Polymorphie und Typklassen

Unsere Themen heute...

- Polymorphie
  - Polymorphe Typen
  - Polymorphe Funktionen

  ...und ihre Vorteile für die Programmierung.
- Typklassen
  ...und ihr Zusammenhang zur Polymorphie.
- Polymorphie vs. Überladung (ad-hoc Polymorphie)
  ...eine Präzisierung ihrer Abgrenzung.

Polymorphe Typen

...ein (Daten-) Typ `T` heißt *polymorph*, wenn bei der Deklaration von `T` der Grundtyp der Elemente (in Form einer Typvariablen) als Parameter angegeben wird.

Typische Beispiele:

```
data Tree a = Nil |
              Node a (Tree a) (Tree a)

data List a = Empty |
              (Head a) (List a)
```

Beachte: Ein Typsynoym `type List = [a]` ist nicht möglich.

Polymorphe Funktionen

...eine Funktion `f` heißt *polymorph*, wenn deren Parameter (in Form einer oder mehrerer Typvariablen) für Argumente unterschiedlicher Typen definiert sind.

Typische Beispiele:

```
depth  :: Tree a -> Int
depth Nil             = 0
depth (Node _ t1 t2) = 1 + max (depth t1) (depth t2)

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

llength :: List a -> Int
llength Empty       = 0
llength (Head _ hs) = 1 + llength hs
```


Warum polymorphe Typen und Funktionen? (1)

...Wiederverwendung durch Parametrisierung!

$\rightsquigarrow$ *ein typisches Vorgehen in der Informatik*!

...die Essenz eines Datentyps (einer Datenstruktur) ist wie die Essenz darauf arbeitender Funktionen oft unabhängig von gewissen typspezifischen Details.

(Vgl. dazu Bäume über ganzen Zahlen, Zeichenreihen, Personen,... oder Listen über Gleitkommazahlen, Wahrheitswerten, Bäumen,... und Funktionen darauf wie zur Bestimmung der Tiefe von Bäumen und der Länge von Listen.

Warum polymorphe Typen und Funktionen? (2)

...durch Parametrisierung werden gleiche Teile ausgeklammert und somit der Wiederverwendung zugänglich!

...(i.w.) gleiche Codeteile müssen nicht (länger) mehrfach geschrieben werden.

...man spricht deshalb auch von *parametrischer Polymorphie*.

Warum polymorphe Typen und Funktionen? (3)

Polymorphie und die mit ihr verbundene Wiederverwendung unterstützt...

- Ökonomie der Programmierung (vulgo: "Schreibfaulheit")

Aber auch...

- Transparenz und Lesbarkeit
  ...durch Betonung der Gemeinsamkeiten, nicht der Unterschiede!
- Verlässlichkeit und Wartbarkeit (Mehrere Dimensionen: Fehlersuche, Weiterentwicklung,...)
  ...als erwünschte Seiteneffekte!
- ...
- Effizienz (der Programmierung)
  $\rightsquigarrow$ *time-to-market*

Warum polymorphe Typen und Funktionen? (4)

Beachte: Auch in anderen Paradigmen wie etwa imperativer und objektorientierter Programmierung lernt man nach und nach den Nutzen polymorpher Konzepte zu schätzen (Aktuelles Stichwort: Generic Java).

Von Polymorphie zu ad-hoc Polymorphie (1)

Zur Erinnerung...

- Polymorphie ("Vielgestaltigkeit)
- Überladen (engl. Overloading) oder ad-hoc Polymorphie

Vordergründig...

...ein Funktionsname kann auf Argumente unterschiedlichen Typs angewendet werden.

Bsp.:

```
Polymorphie                              Ueberladen
-----------                              -----------
length :: [a] -> Int                     equal :: a -> a -> Bool
length []     = 0               "naiv":  equal x y = x == y
length (_:xs) = 1 + length xs
```

Beachte: Anders als für `length` ist für `gleich` keine typunabhängige einheitliche Implementierung möglich (denke an Gleichheit auf Basistypen, Paaren, Listen, Bäumen, etc.)

Von Polymorphie zu ad-hoc Polymorphie (2)

Was heißt das für die Funktion `equal` konkret?

Betrachte dazu beispielsweise...

```
-- Gleichheit auf ganzen Zahlen
equal :: Int -> Int -> Bool
equal x y = x == y

-- Gleichheit auf Paaren von Zeichen und ganzen Zahlen
equal :: (Char,Int) -> (Char,Int) -> Bool
equal (c,i) (d,j) = (c == d) && (i == j)

-- Gleichheit auf Baeumen ueber Zeichenreihen
equal :: Tree String -> Tree String -> Bool
equal Nil Nil                     = True
equal Nil _                       = False
equal _   Nil                     = False
equal (Node s t1 t2) (Node t u1 u2) = (s == t) && (equal t1 u1)
                                               && (equal t2 u2)
```

Von Polymorphie zu ad-hoc Polymorphie (3)

Beobachtung:

> ...*gleicher* (Funktions-) Name, *ähnliche* Funktionalität, *unterschiedlicher* Code.

Mit einem Wort:

Überladung (*ad-hoc Polymorphie*)!

Von Polymorphie zu ad-hoc Polymorphie (4)

Problem...

> ...welchen *polymorphen* oder *allgemeinsten* Typ hat die Funktion `equal` bzw. die verschiedenen Varianten der Funktion `equal`?

Gleichheit auf dem polymorphen Typ `PolType`...

```
equal :: PolType -> PolType -> Bool
```

...mit `PolType` gleich `a`? Also:

```
equal :: a -> a -> Bool   -- ???
```

Sicher nicht. Was wäre z.B. mit Aufrufen wie...

`equal fib fixfib` oder `equal fib fac`,...?

Beachte: ...Gleichheit von Funktionen ist unentscheidbar!

Von ad-hoc Polymorphie zu Typklassen (1)

...offenbar kann nicht für jeden Typ eine Funktion mit der Funktionalität von `equal` angegeben werden.

...die Teilmenge der Typen, für die das möglich ist, bzw. eine Teilmenge davon, für die das in einem konkreten Programm tatsächlich gemacht wird, wird in Haskell als *Sammlung* (engl. *collection*) von Typen, als sog.

- *Typklasse*

bezeichnet.

Intuitiv:

> ..."die Typklasse zu `equal` umfasst `Int`, `(Char,Int)` und `Tree String`".

Von ad-hoc Polymorphie zu Typklassen (2)

Betrachte ein Haskell-Skript mit folgender Definition...

```
-- allEqual :: String -> String -> Bool
allEqual s t = (s == t) && (t == "Fkt. Programmierung")
```

Typabfrage in Hugs liefert...

```
Main> :t allEqual <return>
Main> allEqual :: [Char] -> [Char] -> Bool
```

Nicht wirklich überraschend, aber...

Von ad-hoc Polymorphie zu Typklassen (3)

...wenn wir das wie folgt geänderte Haskell-Skript betrachten:

```
-- allEqual :: String -> String -> String -> Bool
allEqual s t u = (s == t) && (t == u)
```

...liefert die Typabfrage in Hugs jetzt:

```
Main> :t allEqual <return>
Main> allEqual :: Eq a => a -> a -> a -> Bool
```

...und führt uns so (am Bsp. der vordefinierten Klasse `Eq`) auf Haskell's *Typklassen*.

Typklassen in Haskell (1)

...werden gebraucht, wenn eine Funktion "im wesentlichen polymorph" ist, aber gewissen Randbedingungen unterliegt (Bsp.: `equal`, `==`,...)

Die Funktionen `equal` und `allEqual` haben die Typen...

```
equal :: a -> a -> Bool
allEqual :: a -> a -> a -> Bool
```

...nur für solche Instanzen `I` von `a`, für die es Implementierungen der Form

- `equal :: I -> I -> Bool`
- `(==) :: I -> I -> Bool`

gibt.

Typklassen in Haskell (2)

Wir brauchen...

> ...ein sprachliches Konstrukt, solche Randbedingungen auszudrücken.

Haskell's Angebot dafür:

Typklassen

Beachte: Andere Sprachen wie z.B. ML und Opal gehen hier einen anderen Weg und bieten andere Konzepte.

Definition von Typklassen (1)

Zunächst illustriert anhand zweier Beispiele ...

```
-- Selbstdefiniert
class MyEq a where
  equal :: a -> a -> Bool

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


Definition von Typklassen (2)

Allgemeines Muster einer Typklassendefinition...

```
class Name tv where
  ...signature involving the type variable tv...
```

wobei

- `Name` ...Identifikator der Klasse
- `tv` ...Typvariable
- `signature` ...Liste von Namen zusammen mit ihren Typen

Instanzen von Typklassen (1)

Am Beispiel der (Typ-) Klassen `MyEq` und `Eq`...

```
instance MyEq (Tree String) where
  equal Nil Nil                     = True
  equal Nil _                       = False
  equal _   Nil                     = False
  equal (Node s t1 t2) (Node t u1 u2) = (s == t)
                       && (equal t1 u1) && (equal t2 u2)

instance Eq Bool where
  True  == True  = True
  False == False = True
  _     == _     = False
```

Beachte: Der Ausdruck "Instanz" im Haskell-Jargon ist selbst überladen!

- Bislang: Typ `T` ist Instanz eines Typs `U`
- Jetzt zusätzlich: Typ `T` ist Instanz einer (Typ-) Klasse `C`

Instanzen von Typklassen (2)

Am Beispiel der (Typ-) Klassen `MyEq` und `Eq` sowie eines selbst-definierten algebraischen Typs...

```
data Point = Point (Float,Float)

instance Eq Point where
  Point x == Point y = (fst(x)==fst(y)) && (snd(x)==snd(y))

instance MyEq Point where
  equal (Point (x,y)) (Point (w,z)) = (x==w) && (y==z))
```

Beachte: Typ- und Konstruktorname (`Point`!) dürfen übereinstimmen.

Mit diesen Definitionen...

...liefern die Typabfragen in Hugs jetzt:

```
Main> :t equal <return>
Main> equal :: MyEq a => a -> a -> Bool

Main> :t allEqual <return>
Main> allEqual :: Eq a => a -> a -> a -> Bool
```

Sprechweisen:

- Der Teil links von `=>` heißt *Kontext*.
- Wenn `a` Instanz/aus der Klasse `MyEq` ist, d.h. wenn die Funktion `equal` auf `a` definiert ist, dann hat die Funktion `equal` den Typ `a -> a -> Bool`.

Vererbung und Überschreiben vererbter Funktionen

...am Beispiel des Typs `Point` der Klasse `Eq`:

- *Vererbung*:
  `Point` erbt die Funktion "ungleich" aus der Klasse `Eq`:
  `Point x /= Point y = not (Point x == Point y)`
- *Überschreiben*:
  Die ererbte (Standard-) Funktion "ungleich" kann überschrieben werden, z.B. hier durch eine (geringfügig) effizientere Variante:

```
instance Eq Point where
  Point (x,y) == Point (w,z) = (x==w) && (y==z)
  Point x /= Point y = if x/=w then True else y/=z
```


Schlussfolgerungen zu Typklassen

Intuitiv...

...Typklassen sind Kollektionen von Typen, für die eine gewisse Menge von Funktionen ("gleicher" Funktionalität) definiert ist.

Beachte...

..."Gleiche" Funktionalität kann nicht syntaktisch erzwungen werden, sondern liegt in der Verantwortung des Programmierers!

Beispiele:

```
equal :: Float -> Float -> Bool
equal p q = (square p) > (sqrt q)

equal :: Char -> Char -> Bool
equal c d = (c /= d)
```

Stichwort:

Programmier(un)disziplin!

Ergänzungen zu Typklassen (1)

Auch komplexere Konstruktionen sind möglich...

```
instance Eq a => Eq (Tree a) where
  Nil == Nil                        = True
  (Node s t1 t2) == (Node t u1 u2) = (s==t) && (t1==u1) && (t2==u2)
  _ == _                            = False
```

Intuitiv...

...`Tree a` gehört zur/ist Instanz der Klasse `Eq`, wenn `a` zu dieser Klasse gehört.

Ergänzungen zu Typklassen (2)

Komplexere Konstruktionen mittels *Vererbung*...

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

...die Klasse `Ord` erweitert die Klasse `Eq`. Ihre Instanzen müssen alle Funktionen aus `Eq` und `Ord` besitzen.

Beachte: ...`Ord` wie oben liefert bereits Definitionen für einige Funktionen mit. Bei der Definition von Instanzen reicht es, Implementierungen der Funktionen `==` und `<` anzugeben.

Ergänzungen zu Typklassen (3)

Vordefinierte Typklassen in Haskell...

- *Gleichheit* `Eq` ...die Klasse der Typen mit Gleichheitstest
- *Ordnungen* `Ord` ...die Klasse der Typen mit Ordnungsrelationen
- *Aufzählung* `Enum` ...die Klasse der Typen, deren Werte aufgezählt werden können (Bsp.: `[2,4..29]`)
- *Werte zu Zeichenreichen* `Show` ...die Klasse der Typen, deren Werte als Zeichenreihen dargestellt werden können
- *Zeichenreihen zu Werten* `Read` ...die Klasse der Typen, deren Werte aus Zeichenreihen herleitbar sind

Ergänzungen zu Typklassen (4)

Abgeleitete Instanzen von Klassen...

```
data Spielfarbe = Kreuz | Pik | Herz | Karo
                  deriving (Eq,Ord,Enum,Show,Read)
data Tree a = Nil |
              Node a (Tree a) (Tree a)
              deriving (Eq,Ord)
```

...algebraische Typen können mithilfe des Schlüsselworts `deriving` als Instanzen vordefinierter Klassen gekennzeichnet werden.

Beachte: ...die Definitionen abgeleiteter Funktionen können überschrieben werden.

Rückblick: Polymorphie vs. Überladen...

Erinnerung:

Vordergründig... ein Funktionsname kann auf Argumente unterschiedlichen Typs angewendet werden.

Präzisierung:

- *Polymorphe Funktionen*...
  - ...haben eine einzige Definition, die für alle (zugelassenen/abgedeckten) Typen arbeitet (Bsp.: `length :: [a] -> Int]`)
- *Überladene Funktionen*...
  - ...arbeiten für Instanzen einer Klasse von Typen mit einer für jede Instanz spezifischen Definition (Bsp.: `allEqual :: Eq a => a -> a -> a -> Bool`)