Heutiges Thema

Polymorphie und Typklassen...

- Polymorphie
  - Polymorphe Typen
  - Polymorphe Funktionen

  ...und ihre Vorteile für die Programmierung.
- Typklassen
  ...und ihr Zusammenhang mit 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)
```


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)

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

-- Zum Vergleich die Standardfunktion length auf Listen
length :: [a] -> Int
length []     = 0
length (_:xs) = 1 + length xs
```


Seitenblick: Polymorphe Typsynonyme

...auch *polymorphe Typsynonyme* und *Funktionen* darauf sind möglich:

Beispiel:

```
type List a = [a]

lengthTypeList :: List a -> Int
lengthTypeList []     = 0
lengthTypeList (x:xs) = 1 + lengthTypeList xs
```

Oder kürzer:

```
lengthTypeList :: List a -> Int
lengthTypeList x = length x
```

...abstützen auf Standardfunktion `length` möglich, da `List a` Typsynonym, kein neuer Typ.

Beachte: `type List = [a]` ist nicht möglich ($\rightsquigarrow$ *Typfehler!*)

Warum polymorphe Typen und Funktionen? (1)

...Wiederverwendung durch Parametrisierung!

⇝ *ein typisches Vorgehen in der Informatik*!

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

(Man überlege sich das z.B. anhand von Bäumen über ganzen Zahlen, Zeichenreihen, Personen,... oder Listen über Gleitkommazahlen, Wahrheitswerten, Bäumen,... und typischen Funktionen darauf wie etwa zur Bestimmung der Tiefe von Bäumen oder 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")

Vor allem aber auch...

- Transparenz und Lesbarkeit
  ...durch Betonung der Gemeinsamkeiten, nicht der Unterschiede!
- Verlässlichkeit und Wartbarkeit (ein Aspekt mit mehreren Dimensionen: Fehlersuche, Weiterentwicklung,...)
  ...als erwünschte Seiteneffekte!
- ...
- Effizienz (der Programmierung)
  ⇝ *beschleunigter Markteintritt* (*time-to-market*)

Warum polymorphe Typen und Funktionen? (4)

Beachte...

- ...auch in anderen Paradigmen wie etwa imperativer und objektorientierter Programmierung lernt man den Nutzen und die Vorteile polymorpher Konzepte zunehmend zu schätzen!
  ⇝ aktuelles Stichwort: *Generic Java*

Problemfall: 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                         gleich :: a -> a -> Bool
length []     = 0                  "naiv":   gleich x y = x == y
length (_:xs) = 1 + length xs
```

Beachte: Anders als für `length` ist obige Implementierung von `gleich` naiv und in dieser Generalität nicht möglich!)

Informell... Gleichheit keine typunabhängige Eigenschaft! (denke an Gleichheit auf Basistypen, Paaren, Listen, Bäumen, etc.)

Problemfall: Ad-hoc Polymorphie (2)

Problem...

> ...welchen *polymorphen*, welchen *allgemeinsten* Typ sollte die Funktion `gleich` haben?

Gleichheit auf einem polymorphen Typ `PolType`...

```
gleich :: PolType -> PolType -> Bool
```

...mit `PolType` gleich `a`? Mithin also:

```
gleich :: a -> a -> Bool
```

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

`gleich fib fixfib` oder `gleich fib fac`, etc.?

Denn... Gleichheit von Funktionen ist unentscheidbar!

Problemfall: Ad-hoc Polymorphie (3)

Erinnerung:

"Gleichheit von Funktionen ist unentscheidbar" heißt informell...

- ...es gibt keinen Algorithmus, der für zwei beliebig vorgelegte Funktionen stets nach endlich vielen Schritten entscheidet, ob diese Funktionen gleich sind oder nicht.

Zur Überlegung: Folgt aus obigem Resultat, dass die Gleichheit zweier Funktionen nie (nach endlicher Zeit) entschieden werden kann? Wie begründen Sie Ihre Antwort?

Problemfall: Ad-hoc Polymorphie (4)

Andererseits...

- ...für viele Typen ist der Test auf Gleichheit sicher implementierbar

Betrachte z.B. Gleichheit auf...

- ganzen Zahlen
- Paaren aus Zeichen und ganzen Zahlen
- Bäumen über Zeichenreichen
- ...

zusammen mit den...

Problemfall: Ad-hoc Polymorphie (5)

...folgenden typabhängigen Implementierungen des Gleichheitstests auf diesen Typen:

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

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

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


Problemfall: Ad-hoc Polymorphie (6)

Beobachtung:

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

Unschön und lästig (denke auch an $+_{Int}$, $+_{Float}$, $*_{Int}$, $*_{Float}$, etc.). Wünschenswert wäre (wie auch bei $+$, $*$, etc. und dort schon gewohnt):

- ...bei *ähnlicher* Funktionalität auch *gleicher* (Funktions-) Name trotz *unterschiedlichen* Codes.

Verdichtet zu einem Wort:
Überladung bzw. (*ad-hoc Polymorphie*)!

Problemfall: Ad-hoc Polymorphie (7)

Mithin: Wir hätten gerne einen Mechanismus, der uns erlaubte, `gleich` so zu spezifizieren, dass...

```
gleich 17 4
gleich ('a',29) ('a',29)
gleich (Node "abc" Nil Nil) (Node "xyz" Nil Nil)
```

...gültige Aufrufe wären.

Offenes Problem:

- ...wie kann das erreicht werden? Insbesondere, wie kann es in Haskell erreicht werden?

*Haskell*s Angebot dafür:
...das Konzept der *Typklassen*!

Ad-hoc Polymorphie und Typklassen

...am Beispiel des Gleichheitstests:

- ...offenbar kann nicht für jeden Typ eine Funktion mit der Funktionalität von `gleich` angegeben werden, sondern nur für eine Teilmenge aller möglichen Typen.
- ...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 `gleich`” umfasst `Int`, `(Char,Int)` und `Tree String` (und darüberhinaus viele weitere Typen).

Unterwegs zu Typklassen (1)

Betrachte ein Haskell-Skript mit folgender Definition...

```
-- alleGleich :: [Char] -> [Char] -> Bool
alleGleich s t = (s == t) && (t == "Fkt. Programmierung")
```

Typabfrage in Hugs liefert...

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

Vermutlich nicht überraschend, aber...

Unterwegs zu Typklassen (2)

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

```
-- alleGleich :: [Char] -> [Char] -> Bool
alleGleich s t u = (s == t) && (t == u)
```

...liefert die Typabfrage in Hugs jetzt:

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

...und hat uns so auf Haskells Konzept der *Typklassen* geführt, und zwar am Bsp. der vordefinierten Typklasse `Eq`.

Typklassen in Haskell (1)

...werden gebraucht, wenn eine Funktion von "polymorphem Charakter" ist, aber typspezifisch individuelle Randbedingungen erfüllen muss ("pseudopolymorph") (Bsp.: `gleich`, `alleGleich`, (==),...)

Die Funktionen `gleich` und `alleGleich`...

```
gleich :: a -> a -> Bool
alleGleich :: a -> a -> a -> Bool
```

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

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

gibt.

Informell... "*Instanz* gleich *bestimmter konkreter Typ*"

Typklassen in Haskell (2)

Gebraucht wird...

> ...ein sprachliches Konstrukt, die Existenz solcher Implementierungen auszudrücken.

Wie schon angedeutet, Haskell's Angebot dafür:

Typklassen

Hinweis: 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 ...

```
-- Selbstdefinierte Typklasse
class Gleich a where
  gleich :: a -> a -> Bool

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

Beachte:

- ...keine vollständige Implementierung angegeben!
- ...vgl. z.B. mit Schnittstellendefinitionen und Definitionen abstrakter Klassen in Java und achte auf Gemeinsamkeiten und Unterschiede!

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

Instanzbildung von Typklassen (1)

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

```
-- Instanzbildung: ...damit Wahrheitswerte mittels (==)
--                    verglichen werden koennen
instance Eq Bool where
  True  == True  = True
  False == False = True
  _     == _     = False

-- Instanzbildung: Damit Baeume ueber Zeichenreihen mittels
--                 gleich verglichen werden koennen
instance Gleich (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)
```

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 `Eq` und `Gleich` sowie eines selbstdefinierten algebraischen Typs...

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

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

instance Gleich Point where
  gleich (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 gleich <return>
Main> gleich :: Gleich a => a -> a -> Bool

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

Sprechweisen:

- Der Teil links von => heißt *Kontext*.
- Wenn `a` Instanz von/aus der Klasse `Gleich` ist, d.h. wenn die Funktion `gleich` auf `a` definiert ist, dann hat die Funktion `gleich` 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. wie unten 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!

So wäre z.B. auch folgendes (nicht gleichzeitiges!) Überschreiben möglich (wenn auch kaum empfehlenswert):

```
c == d = (c+d) > (*2) d

c /= d = (c == d)
```

Mithin: ...Appell an die *Programmierdisziplin* unabdingbar!

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 weiterer Instanzen reicht es, Implementierungen der Funktionen `==` und `<` anzugeben.

Ergänzungen zu Typklassen (3)

Auch *Mehrfachvererbung* ist möglich...

```
class (Eq a, Gleich a) => GuertelundHosentraegerGleich a where
...
...
...
```

- ...vgl. dies auch mit Vererbungskonzepten objektorientierter Sprachen!

Ergänzungen zu Typklassen (4)

Vordefinierte Typklassen in Haskell...

- *Gleichheit* `Eq` ...die Klasse der Typen mit Gleichheitstest
- *Ordnungen* `Ord` ...die Klasse der Typen mit Ordnungsrelationen (wie etwa $<$, $\leq$, $>$, $\geq$, etc.)
- *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 (5)

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: ...auch hier können die Definitionen abgeleiteter Funktionen überschrieben werden.

Ergänzungen zu Typklassen (6)

Mehr zu Typklassen, alles zu Funktionen vordefinierter Typklassen...

- ...in jedem guten Buch über Haskell!

Rückblick: Polymorphie vs. Überladen...

Erinnerung:

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

Präzisierung:

- *Polymorphe Funktionen...*
  - ...haben eine einzige Implementierung, 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 Implementierung (Bsp.: `alleGleich :: Eq a => a -> a -> a -> Bool`)

Ausblick

...auf den kommenden Dienstag:

Funktionen höherer Ordnung!