Heutiges Thema

Fortführung vom letzten Mal...

- Polymorphie
  - auf Funktionen
    - Parametrische Polymorphie ("Echte Polymorphie")
    - Ad hoc Polymorphie (synonym: Überladung)
      $\leadsto$ Haskell-spezifisch: Typklassen
  - auf Datentypen
    - Algebraische Datentypen
    - Typsynonymen

Polymorphe Typsynonyme

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

Beispiel:

```
type List a = [a]

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

Oder kürzer:

```
lengthList :: List a -> Int
lengthList = length
```

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

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

Zusammenfassung und erste Schlussfolgerungen

Zu...

- Polymorphen Funktionen,
- Polymorphen Datentypen und
- Vorteilen, die aus ihrer Verwendung resultieren.

Polymorphe Typen

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

Zwei Beispiele:

```
data Tree a b = Leaf a |
                Node b (Tree a b) (Tree a b)

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 b) -> Int
depth Leaf _          = 1
depth (Node _ t1 t2) = 1 + max (depth t1) (depth t2)

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

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

Warum polymorphe Typen und Funktionen? (1)

...Wiederverwendung durch Parametrisierung!

$\rightsquigarrow$ *wie schon gesagt, 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 vergegenwärtige sich das noch einmal 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 die...

- Ökonomie der Programmierung (vulgo: "*Schreibfaulheit*")

Insbesondere aber trägt sie bei zu höherer...

- 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)
  $\leadsto$ *höhere Produktivität, früherer Markteintritt* (*time-to-market*)

Warum polymorphe Typen und Funktionen? (4)

Beachte...

- ...auch in anderen Paradigmen wie etwa imperativer und speziell objektorientierter Programmierung lernt man, den Nutzen und die Vorteile polymorpher Konzepte zunehmend zu schätzen!

  $\leadsto$ aktuelles Stichwort: *Generic Java*

Ad hoc Polymorphie

Bisher haben wir besprochen...

- Polymorphie in Form von...
  - (Parametrischer) Polymorphie
  - Polymorphie auf Datentypen

Jetzt ergänzen wir diese Betrachtung um...

- *Ad hoc* Polymorphie (Überladen, "unechte" Polymorphie)

Zur Motivation von Ad hoc Polymorphie

Ausdrücke der Form

```
(+) 2 3          => 5
(+) 27.55 12.8   => 39.63
(+) 12.42 3      => 15.42
```

...sind Beispiele wohlgeformter Haskell-Ausdrücke, wohingegen

```
(+) True False
(+) 'a' 'b'
(+) [1,2,3] [4,5,6]
```

...Beispiele nicht wohlgeformter Haskell-Ausdrücke sind.

Zur Motivation von Ad hoc Polymorphie

Offenbar...

- ist `(+)` nicht monomorph
  ...da `(+)` für mehr als einen Argumenttyp arbeitet

- ist der Typ von `(+)` verschieden von `a -> a -> a`
  ...da `(+)` nicht für jeden Argumenttyp arbeitet

Tatsächlich...

- ist `(+)` typisches Beispiel eines *überladenen* Operators.

Das Kommando `:t (+)` in Hugs liefert

- `(+) :: Num a => a -> a -> a`

Polymorphie vs. Ad hoc Polymorphie

Intuitiv

- Polymorphie

  Der polymorphe Typ `(a -> a)` wie in der Funktion
  `id :: a -> a` steht abkürzend für:

  `∀(a) a -> a` "...für alle Typen"

- Ad hoc Polymorphie

  Der Typ `(Num a => a -> a -> a)` wie in der Funktion
  `(+) :: Num a => a -> a -> a` steht abkürzend für:

  `∀(a∈Num) a -> a -> a` "...für alle Typen aus `Num`"

Im Haskell-Jargon ist `Num` eine sog.

- *Typklasse*, eine von vielen vordefinierten Typklassen.

Typklassen in Haskell

Informell

- Eine Typklasse ist eine Kollektion von Typen, auf denen eine in der Typklasse festgelegte Menge von Funktionen definiert ist.

- Die Typklasse `Num` ist die Kollektion der numerischen Typen `Int`, `Integer`, `Float`, etc., auf denen u.a. die Funktionen `(+)`, `(*)`, etc. definiert sind.

Hinweis: Vergleiche dieses Klassenkonzept z.B. mit dem Schnittstellenkonzept aus Java. Gemeinsamkeiten, Unterschiede?

Polymorphie vs. Ad hoc Polymorphie

Informell...

- (*Parametrische*) *Polymorphie* ...
  $\leadsto$ gleicher Code trotz unterschiedlicher Typen

- *ad-hoc Polymorphie* (synonym: *Überladen* (engl. *Overloading*))...
  $\leadsto$ unterschiedlicher Code trotz gleichen Namens (mit sinnvollerweise i.a. ähnlicher Funktionalität)

Ein erweitertes Bsp. zu Typklassen (1)

Wir nehmen an, wir seien an der Größe interessiert von

- Listen und
- Bäumen

Der Begriff "Größe" sei dabei typabhängig, z.B.

- Anzahl der Elemente bei Listen
- Anzahl der
  - Knoten
  - Blätter
  - Benennungen
  - ...

  bei Bäumen

Ein erweitertes Bsp. zu Typklassen (2)

Wir betrachten folgende Baumvarianten...

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

data Tree1 a b = Leaf1 b |
                 Node1 a b (Tree1 a b) (Tree1 a b)

data Tree2 = Leaf2 String |
             Node2 String Tree2 Tree2
```

...und den Haskellstandardtyp für Listen.

Ein erweitertes Beispiel zu Typklassen

Naive Lösung: Schreibe für jeden Typ eine passende Funktion:

```
sizeT :: Tree a -> Int          -- Zaehlen der Knoten
sizeT Nil           = 0
sizeT (Node n l r) = 1 + sizeT l + sizeT r

sizeT1 :: (Tree1 a b) -> Int    -- Zaehlen der Benennungen
sizeT1 (Leaf1 m)        = 1
sizeT1 (Node1 m n l r) = 2 + sizeT1 l + sizeT1 r

sizeT2 :: Tree2 -> Int          -- Summe der Laengen der Benennungen
sizeT2 (Leaf2 m)     = length m
sizeT2 (Node2 m l r) = length m + sizeT2 l + sizeT2 r

sizeLst :: [a] -> Int           -- Zaehlen der Elemente
sizeLst = length
```

Ein erweitertes Bsp. zu Typklassen (3)

"Smarte" Lösung mithilfe von Typklassen:

```
class Size a where                    -- Definition der Typklasse Size
  size :: a -> Int

instance Size (Tree a) where          -- Instanzbildung fuer (Tree a)
  size Nil          = 0
  size (Node n l r) = 1 + size l + size r

instance Size (Tree1 a b) where       -- Instanzbildung fuer (Tree1 a b)
  size (Leaf1 m)       = 1
  size (Node1 m n l r) = 2 + size l + size r

instance Size Tree2 where             -- Instanzbildung fuer Tree2
  size (Leaf2 m)     = length m
  size (Node2 m l r) = length m + size l + size r

instance Size [a] where
  size = length
```

Ein erweitertes Bsp. zu Typklassen (4)

Das Kommando `:t size` liefert:

```
size :: Size a => a -> Int
```

...und wir erhalten wie gewünscht:

```
size Nil => 0
size (Node "asdf" (Node "jk" Nil Nil) Nil) => 2

size (Leaf1 "adf") => 1
size ((Node1 "asdf" 3
          (Node1 "jk" 2 (Leaf1 17) (Leaf1 4))
          (Leaf1 21)) => 7

size (Leaf2 "abc") => 3
size (Node2 "asdf"
       (Node2 "jkertt" (Leaf2 "abc") (Leaf2 "ac"))
       (Leaf "xy"))      => 17

size [5,3,45,676,7] => 5
size [True,False,True] => 3
```

Definition von Typklassen

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

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!

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

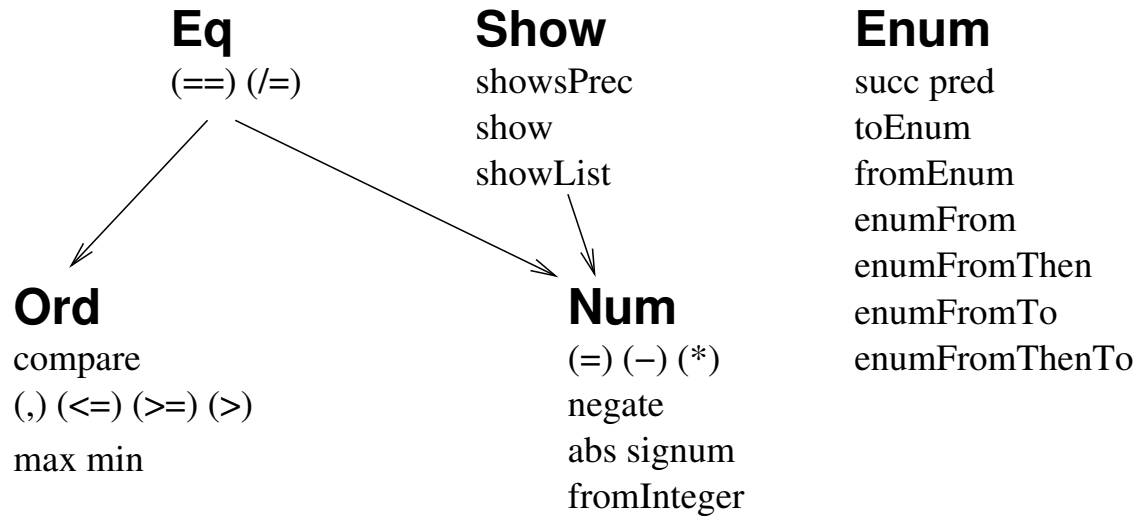
Bsp. einiger in Haskell vordef. Typklassen (1)

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
- ...

Bsp. einiger in Haskell vordef. Typklassen (2)

Auswahl vordefinierter Typklassen, ihrer Abhängigkeiten, Operatoren und Funktionen in Standard Prelude nebst Bibliotheken:



Quelle: Fethi Rabhi, Guy Lapalme. "Algorithms - A Functional Approach", Addison-Wesley, 1999.

Vorschau auf die kommenden Aufgabenblätter...

Ausgabe des...

- fünften Aufgabenblatts: Mo, den 13.11.2006
  ...Abgabetermine: Di, den 21.11.2006, und Di, den 28.11.2006, jeweils 15:00 Uhr

- sechsten Aufgabenblatts: Mo, den 20.11.2006
  ...Abgabetermine: Di, den 28.11.2006, und Di, den 05.12.2006, jeweils 15:00 Uhr

- siebenten Aufgabenblatts: Mo, den 27.11.2006
  ...Abgabetermine: Di, den 05.12.2006, und Di, den 12.12.2006, jeweils 15:00 Uhr

Vorschau auf die nächsten Vorlesungstermine...

- Do, 09.11.2006, Vorlesung von 16:30 Uhr bis 18:00 Uhr im Radinger-Hörsaal
- Do, 16.11.2006: Keine Vorlesung, aber Besprechung von Aufgabenblatt 2 und 3 beginnend um 16:30 Uhr im Radinger-Hörsaal
- Do, 23.11.2006, Vorlesung von 16:30 Uhr bis 18:00 Uhr im Radinger-Hörsaal
- Do, 30.11.2006, Vorlesung von 16:30 Uhr bis 18:00 Uhr im Radinger-Hörsaal