

---

# Rückblick: Das zentrale Thema beim letzten Mal...

...selbstdefinierte (neue) Datentypen in Haskell!

~> Haskell's Vehikel dafür: *Algebraische Typen*

*Algebraische Typen* erlauben uns zu definieren...

- Summentypen
  - Spezialfälle
    - \* Produkttypen
    - \* Aufzählungstypen

In der Praxis besonders wichtige Varianten...

- Rekursive Typen (~> "unendliche" Datenstrukturen)
- Polymorphe Typen (~> Wiederverwendung)

Offen geblieben war die Untersuchung und Diskussion *polymorpher Typen*!

---

# Heutiges Thema...

## Polymorphie

- Bedeutung lt. Duden:
  - *Vielgestaltigkeit, Verschiedengestaltigkeit*  
...mit speziellen fachspezifischen Bedeutungsausprägungen
  - \* In der *Chemie*: das Vorkommen mancher Mineralien in verschiedener Form, mit verschiedenen Eigenschaften, aber gleicher chemischer Zusammensetzung
  - \* In der *Biologie*: Vielgestaltigkeit der Blätter oder der Blüte einer Pflanze
  - \* In der *Sprachwissenschaft*: das Vorhandensein mehrerer sprachlicher Formen für den gleichen Inhalt, die gleiche Funktion (z.B. die verschiedenartigen Pluralbildungen in: die *Wiesen*, die *Felder*, die *Tiere*)
  - \* In der *Informatik*, speziell der *Theorie der Programmiersprachen*:  
~> unser heutiges Thema!

---

# Polymorphie

...im programmiersprachlichen Kontext unterscheiden wir insbesondere zwischen

- Polymorphie
  - auf Datentypen
  - auf Funktionen
    - \* Parametrische Polymorphie (“Echte Polymorphie”)
    - \* Ad hoc Polymorphie (synonym: Überladung)
      - ↪ Haskell-spezifisch: Typklassen

---

# Polymorphie

Wir beginnen mit...

- (Parametrischer) Polymorphie auf Funktionen
  - ...die wir an einigen Beispielen schon kennengelernt haben:
    - Die Funktionale `curry` und `uncurry`
    - Die Funktionen `length`, `head` und `tail`

---

## Rückblick (auf Vorlesungsteil 3)

Die Funktionale `curry` und `uncurry`...

`curry :: ((a,b) -> c) -> (a -> b -> c)`

`curry f x y = f (x,y)`

`uncurry :: (a -> b -> c) -> ((a,b) -> c)`

`uncurry g (x,y) = g x y`

---

# Rückblick (auf Vorlesungsteil 1)

Die Funktionen `length`, `head` und `tail`...

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

```
head :: [a] -> a
head (x:_) = x
```

```
tail :: [a] -> [a]
tail (_:xs) = xs
```

---

# Äußeres Kennzeichen parametrischer Polymorphie

Statt

- (ausschließlich) konkreter Typen (wie Int, Bool, Char,...)

treten in der (Typ-) Signatur der Funktionen

- (auch) *Typparameter*, sog. *Typvariablen*

auf.

*Beispiele:*

```
curry :: ((a,b) -> c) -> (a -> b -> c)
```

```
length :: [a] -> Int
```

---

# Typvariablen in Haskell

Typvariablen in Haskell sind...

- freigewählte Identifikatoren, die mit einem Kleinbuchstaben beginnen müssen  
z.B.: a, b, aber auch fp185161\_WS0506

Im Unterschied dazu sind Typnamen, (Typ-) Konstruktoren in Haskell...

- freigewählte Identifikatoren, die mit einem Großbuchstaben beginnen müssen  
z.B.: A, String, Node, aber auch Fp185161\_WS0506

---

# Warum Polymorphie auf Funktionen?

...Wiederverwendung (durch Abstraktion)!

*~> ein typisches Vorgehen in der Informatik!*

---

# Einfaches Bsp.: Funktionsabstraktion

Sind viele Ausdrücke der Art

```
(5 * 37 + 13) * (37 + 5 * 13)
(15 * 7 + 12) * (7 + 15 * 12)
(25 * 3 + 10) * (3 + 25 * 10)
...
```

zu berechnen, schreibe eine Funktion

```
f :: Int -> Int -> Int -> Int
f a b c = (a * b + c) * (b + a * c)
```

um die Rechenvorschrift  $(a * b + c) * (b + a * c)$  wiederverwenden zu können:

```
f 5 37 13
f 15 7 12
f 25 3 10
...
```

---

# Zur Motivation param. Polymorphie (1)

Listen können Elemente sehr unterschiedlicher Typen zusammenfassen, z.B.

- Listen von Basistypeelementen  
`[2,4,23,2,53,4] :: [Int]`
- Listen von Listen  
`[[2,4,23,2,5], [3,4], [], [56,7,6,]] :: [[Int]]`
- Listen von Paaren  
`[(3.14,42.0), (56.1,51.3), (1.12,2.22)] :: [Point]`
- Listen von Bäumen  
`[Nil, Node 42 Nil Nil), Node 17 (Node 4 Nil Nil) Nil)]`
- ...
- Listen von Funktionen  
`[fact, fib, fun91] :: [Integer -> Integer]`

---

## Zur Motivation param. Polymorphie (2)

- *Aufgabe*: Bestimme die Länge einer Liste, d.h. die Anzahl ihrer Elemente.
- *Naive Lösung*: Schreibe für jeden Typ eine entsprechende Funktion.

---

# Zur Motivation param. Polymorphie (4)

Umsetzung der naiven Lösung:

```
lengthIntLst :: [Int] -> Int
lengthIntLst [] = 0
lengthIntLst (_:xs) = 1 + lengthIntLst xs
```

```
lengthIntLstLst :: [[Int]] -> Int
lengthIntLstLst [] = 0
lengthIntLstLst (_:xs) = 1 + lengthIntLstLst xs
```

```
lengthPointLst :: [Point] -> Int
lengthPointLst [] = 0
lengthPointLst (_:xs) = 1 + lengthPointLst xs
```

```
lengthTreeLst :: [BinTree1] -> Int
lengthTreeLst [] = 0
lengthTreeLst (_:xs) = 1 + lengthTreeLst xs
```

```
lengthFunLst :: [Integer -> Integer] -> Int
lengthFunLst [] = 0
lengthFunLst (_:xs) = 1 + lengthFunLst xs
```

---

## Zur Motivation param. Polymorphie (5)

Damit möglich:

```
lengthIntLst      [2,4,23,2,53,4] => 6
lengthIntLstLst  [[2,4,23,2,5],[3,4],[],[56,7,6,]] => 4
lengthPointLst   [(3.14,42.0),(56.1,51.3),(1.12,2.22)] => 3
lengthTreeLst    [Nil,Node 42 Nil Nil, Node 17 (Node 4 Nil Nil) Nil)] => 3
lengthFunLst     [fact, fib, fun91] => 3
```

---

# Zur Motivation param. Polymorphie (6)

Beobachtung:

- Die einzelnen Rechenvorschriften zur Längenberechnung sind i.w. identisch
- Unterschiede beschränken sich auf
  - Funktionsnamen und
  - Typsignaturen

---

# Zur Motivation param. Polymorphie (7)

Sprachen, die parametrische Polymorphie offerieren, erlauben eine elegantere Lösung unserer Aufgabe:

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

```
length [2,4,23,2,53,4] => 6
length [[2,4,23,2,5],[3,4],[],[56,7,6,]] => 4
length [(3.14,42.0),(56.1,51.3),(1.12,2.22)] => 3
length [Nil,Node 42 Nil Nil, Node 17 (Node 4 Nil Nil) Nil]] => 3
length [fact, fib, fun91] => 3
```

Funktionale Sprachen, auch Haskell, offerieren parametrische Polymorphie!

---

## Zur Motivation param. Polymorphie (8)

Unmittelbare Vorteile parametrischer Polymorphie:

- Wiederverwendung von
  - Rechenvorschriften und
  - Funktionsnamen (*Gute Namen sind knapp!*)

---

# Monomorphie vs. Polymorphie

Rechenvorschriften der Form

- `length :: [a] -> Int` heißen *polymorph*.

Rechenvorschriften der Form

- `lengthIntLst :: [Int] -> Int`
- `lengthIntLstLst :: [[Int]] -> Int`
- `lengthPointLst :: [Point] -> Int`
- `lengthFunLst :: [Integer -> Integer] -> Int`
- `lengthTreeLst :: [BinTree1] -> Int` heißen *monomorph*.

---

# Sprechweisen im Zshg. mit parametrischer Polymorphie

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

*Sprechweisen:*

- a in der Typsignatur von length heißt *Typvariable*. Typvariablen werden mit Kleinbuchstaben gewöhnlich vom Anfang des Alphabets bezeichnet: a, b, c,...
- Typen der Form...

```
length :: [Point] -> Int
length :: [[Int]] -> Int
length :: [Integer -> Integer] -> Int
...
```

heißen *Instanzen* des Typs [a] -> Int. Letzterer heißt *allgemeinster Typ* der Funktion length.

*Bem.:* Das Hugs-Kommando `:t expr` liefert stets den (eindeutig bestimmten) *allgemeinsten* Typ eines (wohlgeformten) Haskell-Ausdrucks `expr`.

---

# Weitere Bsp. polymorpher Funktionen

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

id 3 => 3
id ["abc","def"] => ["abc","def"]

zip :: [a] -> [b] -> [(a,b)]    -- "Verschmelzen" von Listen
zip (x:xs) (y:ys) = (x,y) : zip xs ys
zip _ _          = []

zip [3,4,5] ['a','b','c','d'] => [(3,'a'),(4,'b'),(5,'c')]
zip ["abc","def","geh"] [(3,4),(5,4)] => [("abc",(3,4)),("def",(5,4))]

unzip :: [(a,b)] -> ([a],[b])    -- "Aufreissen" von Listen
unzip [] = ([],[])
unzip ((x,y):ps) = (x:xs,y:ys)
                  where
                    (xs,ys) = unzip ps

unzip [(3,'a'),(4,'b'),(5,'c')] => ([3,4,5],['a','b','c'])
unzip [("abc",(3,4)),("def",(5,4))] => ("abc","def"),[(3,4),(5,4)]
```

---

## Weitere in Haskell auf Listen vordefinierte (polymorphe) Funktionen

<code>:</code>	<code>:: a -&gt; [a] -&gt; [a]</code>	Listenkonstruktor (rechtsassoziativ)
<code>!!</code>	<code>:: [a] -&gt; Int -&gt; a</code>	Proj. auf i-te Komp., Infixop.
<code>length</code>	<code>:: [a] -&gt; Int</code>	Länge der Liste
<code>++</code>	<code>:: [a] -&gt; [a] -&gt; [a]</code>	Konkatenation zweier Listen
<code>concat</code>	<code>:: [[a]] -&gt; [a]</code>	Konkatenation mehrerer Listen
<code>head</code>	<code>:: [a] -&gt; a</code>	Listenkopf
<code>last</code>	<code>:: [a] -&gt; a</code>	Listen "endelement"
<code>tail</code>	<code>:: [a] -&gt; [a]</code>	Liste ohne Listenkopf
<code>init</code>	<code>:: [a] -&gt; [a]</code>	Liste ohne Listenendelement
<code>splitAt</code>	<code>:: Int -&gt; [a] -&gt; [[a], [a]]</code>	Aufspalten einer Liste an Stelle i
<code>reverse</code>	<code>:: [a] -&gt; [a]</code>	Umdrehen einer Liste
<code>...</code>		

---

# Polymorphie

Soviel zu parametrischer Polymorphie auf Funktionen...

Wir fahren fort mit

- Polymorphie auf Datentypen
  - Algebraische Datentypen
  - Typsynonymen

---

# Polymorphe algebraische Typen (1)

Der Schlüssel dazu...

↪ Definitionen algebraischer Typen dürfen Typvariablen enthalten und werden dadurch polymorph.

*Beispiele:* Paare und Bäume...

```
data Pairs a = Pair a a
```

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

---

## Polymorphe algebraische Typen (2)

Beispiele konkreter Werte von Paaren und Bäumen:

```
data Pairs a = Pair a a
```

```
Pair 17 4      :: Pairs Int
```

```
Pair [] [42]  :: Pairs [Int]
```

```
Pair [] []    :: Pairs [a]
```

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

```
Node 'a' (Node 'b' Nil Nil) (Node 'z' Nil Nil)      :: Tree Char
```

```
Node 3.14 (Node 2.0 Nil Nil) (Node 1.41 Nil Nil)    :: Tree Float
```

```
Node "abc" (Node "b" Nil Nil) (Node "xyzzz" Nil Nil) :: Tree [Char]
```

---

## Polymorphe algebraische Typen (3)

Ähnlich wie parametrische Polymorphie unterstützt auch...

- Polymorphie auf algebraischen Datentypen

Wiederverwendung!

*Vergleiche* dies mit der schon bekannten Situation im Zshg. mit *polymorphen Listen*:

```
length :: [a] -> Int
```

---

## Polymorphe Typen (4)

Ähnlich wie bei der Funktion *Länge* auf Listen...

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

...kann auf algebraischen Typen Typunabhängigkeit generell vorteilhaft ausgenutzt werden, wie hier das Bsp. der Funktion `depth` auf Bäumen zeigt:

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

depth (Node 'a' (Node 'b' Nil Nil) (Node 'z' Nil Nil))      => 2
depth (Node 3.14 (Node 2.0 Nil Nil) (Node 1.41 Nil Nil))    => 2
depth (Node "abc" (Node "b" Nil Nil) (Node "xyzzz" Nil Nil)) => 2
```

---

# Heterogene algebraische Typen

...sind möglich, z.B. *heterogene* Bäume:

```
data HTree = LeafS String      |
           LeafI Int           |
           NodeF Float HTree HTree |
           NodeB Bool  HTree HTree

-- 2 Varianten der Funktion Tiefe auf Werten vom Typ HTree
depth :: HTree -> Int
depth (LeafS _) = 1
depth (LeafI _) = 1
depth (NodeF _ t1 t2) = 1 + max (depth t1) (depth t2)
depth (NodeB _ t1 t2) = 1 + max (depth t1) (depth t2)

depth :: HTree -> Int
depth (NodeF _ t1 t2) = 1 + max (depth t1) (depth t2)
depth (NodeB _ t1 t2) = 1 + max (depth t1) (depth t2)
depth _                = 1
```

*Beachte:* ...folgendes geht nicht  $\rightsquigarrow$  *Syntaxfehler!*

```
depth :: HTree -> Int
depth (_ _ t1 t2) = 1 + max (depth t1) (depth t2)
depth _          = 1
```

---

# Heterogene polymorphe algeb. Typen

...sind ebenfalls möglich, z.B. *heterogene* polymorphe Bäume:

```
data PHTree a b c d = LeafA a                               |
                    LeafB b                               |
                    NodeC c (PHTree a b c d) (PHTree a b c d) |
                    NodeD d c (PHTree a b c d) (PHTree a b c d)

-- 2 Varianten der Funktion Tiefe auf Werten vom Typ PHTree
depth :: (PHTree a b c d) -> Int
depth (LeafA _)      = 1
depth (LeafB _)      = 1
depth (NodeC _ t1 t2) = 1 + max (depth t1) (depth t2)
depth (NodeD _ _ t1 t2) = 1 + max (depth t1) (depth t2)

depth :: (PHTree a b c d) -> Int
depth (NodeC _ t1 t2) = 1 + max (depth t1) (depth t2)
depth (NodeD _ _ t1 t2) = 1 + max (depth t1) (depth t2)
depth _                = 1
```

*Das heißt...*

- Auch “mehrfach” polymorphe Datenstrukturen sind möglich!

---

# Vorschau auf die kommenden Aufgabenblätter...

## Verlängerte Abgabetermine!

*Beachten Sie auch die verlängerten Abgabetermine für Blatt 1&2! (Siehe Webseite der LVA für Details.)*

Ausgabe des...

- dritten Aufgabenblatts: Gestern, Mo, den 30.10.2006  
...Abgabetermine: Di, den 07.11.2006, und Di, den 14.11.2006, jeweils 15:00 Uhr
  - vierten Aufgabenblatts: Gestern, Mo, den 30.10.2006  
...Abgabetermine: Di, den 14.11.2006, und Di, den 21.11.2006, jeweils 15:00 Uhr
  - fünften Aufgabenblatts: Mo, den 13.11.2006  
...Abgabetermine: Di, den 21.11.2006, und Di, den 28.11.2006, jeweils 15:00 Uhr
-

---

## Vorschau auf die nächsten Vorlesungstermine...

- *Do, 02.11.2006: Keine Vorlesung! (Allerseelentag)*
- Di, 07.11.2006, Vorlesung von 13:00 Uhr bis 14:00 Uhr im Informatik-Hörsaal
- 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

---

**Jetzt gleich...**

**Besprechung von Aufgabenblatt 1!**