Algebraische Typen

...selbstdefinierte (neue) Typen in Haskell!

Algebraische Typen erlauben uns zu definieren...

- Aufzählungstypen
- Produkttypen
- Summentypen

In der Praxis besonders wichtige Varianten...

- Rekursive Typen ($\leadsto$ "unendliche" Datentypen)
- Polymorphe Typen ($\leadsto$ Wiederverwendung)

Zur Erinnerung... (1)

Typdefinitionen in imperativen Sprachen am Bsp. von Pascal:

- *Aufzählungstypen*

```
TYPE spielfarbe = (kreuz, pik, herz, karo);
     wochenende = (sonnabend, sonntag);
     familienstand = (led, verh, gesch, verw);
     form = (kreis, rechteck, quadrat, dreieck);
```

- *Produkttypen*:

```
TYPE person = RECORD
                name: ARRAY [1..25] OF char;
                geschlecht: (maennlich, weiblich);
                alter: integer
              END;
```

Zur Erinnerung... (2)

- *Summentypen*:

```
TYPE mitglied =
       RECORD
         CASE fmstd: familienstand OF
           led: (hobby: boolean);
           verh: (nameGatte: ARRAY [1..25] OF char;
                  istMitglied: boolean);
           gesch: (aktAnschr: ARRAY [1..100] OF char;
                   interessen: ARRAY [1..250] OF char)
       END;
     figur =
       RECORD
         CASE frm: form OF
           kreis: (radius: real);
           rechteck : (breite, hoehe: real);
           dreieck: (s1, s2, s3: real; rechtwkg: boolean);
       END;
```

Zurück zu Haskell: Das allg. Muster...

...einer algebraischen Typdefinition:

```
data TypeName
  = Con1 t11 ... t1k1 |
    Con2 t21 ... t2k2 |
        ...           |
    Conn tn1 ... tnkn
```

Sprechweisen:

- `TypeName` ...*Typname/-identifikator*
- $\texttt{Con}_i$, $i = 1..n$ ...*Konstruktor(en)/-identifatoren*
- k_i, $i = 1..n$ ...*Stelligkeit* des Konstruktors $\texttt{Con}_i$, $\mathsf{k}_i \geq 0$, $i = 1..n$

...wobei Typ- und Konstruktoridentifikatoren mit einem Großbuchstaben beginnen müssen.

Konstruktoren...

...können als Funktionsdefinitionen gelesen werden:

$\mathtt{Con}_i$ `::` $\mathtt{t}_{i1}$ `-> ... ->` $\mathtt{t}_{ik_i}$ `-> TypeName`

Elemente eines algebraischen Datentyps konstruiert durch...

...Anwendung eines Konstruktors auf Werte "passenden" Typs, d.h....

$\mathtt{Con}_i\ \mathtt{v}_{i1}\ \ldots\ \mathtt{v}_{ik_i}$

wobei $\mathtt{v}_{i1}$ `::` $\mathtt{t}_{i1}$ ... $\mathtt{v}_{ik_i}$ `::` $\mathtt{t}_{ik_i}$, $j = 1..k_i$

Aufzählungstypen (1)

Nullstellige Konstruktoren führen auf *Aufzählungstypen*...

Beispiele:

```
data Spielfarbe    = Kreuz | Pik | Herz | Karo
data Wochenende    = Sonnabend | Sonntag
data Geschlecht    = Maennlich | Weiblich
data Familienstand = Ledig | Verheiratet | Geschieden | Verwitwet
data Form          = Kreis | Rechteck | Quadrat | Dreieck
```

Aber auch...

```
data Bool = True | False
```

...ein vordefinierter (Aufzählungs-) Typ in Haskell.

Aufzählungstypen (2)

Funktionsdefinitionen über Aufzählungstypen...

...üblicherweise mit Hilfe von Pattern-matching.

Beispiel:

```
hatEcken :: Form -> Bool
hatEcken Kreis = False
hatEcken _     = True
```

Produkttypen

Alternativenlose mehrstellige Konstruktoren führen auf *Produkttypen*...

Beispiel:

```
type Name  = String
type Alter = Int
data Leute = Person Name Geschlecht Alter
```

Beispiele: ...für Werte des Typs `Leute`.

```
Person "Paul Pfiffig" Maennlich 23
Person "Paula Plietsch" Weiblich 22
```

Beachte: Funktionalität der Konstruktorfunktion ist hier...

```
Person :: Name -> Geschlecht -> Alter -> Leute
```

Produkttypen vs. Tupeltypen (1)

Der Typ `Leute` als...

- *Produkttyp*

```
data Leute = Person Name Geschlecht Alter
```

- *Tupeltyp*

```
type Leute = (Name, Geschlecht, Alter)
```

Vordergründiger Unterschied:
...in der Variante als Tupeltyp fehlt der Konstruktor
(in diesem Bsp.: `Person`)

Produkttypen vs. Tupeltypen (2)

...eine Abwägung von Vor- und Nachteilen.

Produkttypen und ihre typischen...

- *Vorteile* gegenüber *Tupeltypen*
  - Objekte des Typs sind mit dem Konstruktor "markiert" (trägt zur Dokumentation bei)
  - Tupel mit zufällig passenden Komponenten nicht irrtümlich als Elemente des Produkttyps manipulierbar (Typsicherheit! Vgl. früheres Bsp. zur Umrechnung EUR in Yen!)
  - Aussagekräftigere (Typ-) Fehlermeldungen (Typsynonyme können wg. Expansion in Fehlermeldungen fehlen).
- *Nachteile* gegenüber *Tupeltypen*
  - Produkttypelemente sind weniger kompakt, erfordern längere Definitionen (mehr Schreibarbeit)
  - Auf Tupeln vordefinierte polymorphe Funktionen (z.B. `fst`, `snd`, `zip`, `unzip`, `...`) stehen nicht zur Verfügung.

Andererseits...

```
data Euro        = EUR Float
data Yen         = YEN Float
data Temperature = Temp Float

myPi    :: Float
daumen  :: Float
maxTemp :: Temperature
myPi     = 3.14
daumen   = 5.55
maxTemp  = Temp 43.2
```

Eine Funktionsdefinition im Stile von...

```
currencyConverter :: Euro -> Yen
currencyConverter x = x + myPi * daumen
```

...wird jetzt durch das Typsystem von Haskell verhindert!

Somit als Fazit...

- *Typsynonyme* wie...

```
type Euro        = Float
type Yen         = Float
type Temperature = Float
```

...erben alle Operationen von `Float` und sind damit beliebig austauschbar – mit allen Annehmlichkeiten und Gefahren, sprich Fehlerquellen.

- *Produkttypen* wie...

```
data Euro        = EUR Float
data Yen         = YEN Float
data Temperature = Temp Float
```

...erben keinerlei Operationen von `Float`, bieten dafür aber um den Preis zusätzlicher Schreibarbeit Typsicherheit.

Summentypen (1)

Alternative Konstruktoren führen auf Summentypen...

Beispiel:

```
type Radius       = Float
type Breite       = Float
type Hoehe        = Float
type Seite1       = Float
type Seite2       = Float
type Seite3       = Float
type Rechtwinklig = Bool
data Figur = Kreis Radius |
             Rechteck Breite Hoehe |
             Quadrat Kantenlaenge |
             Dreieck Seite1 Seite2 Seite3 Rechtwinklig
```

Die Varianten einer Summe werden durch "|" getrennt.

Summentypen (2)

Beispiele: ...für Werte des Typs `Figur`

```
Kreis 3.14
Rechteck 17.0 4.0
Quadrat 47.11
Dreieck 3.0 4.0 5.0 True
```

Beachte: Auch Aufzählungstypen sind in Haskell als Summentypen realisiert!

Rekursive Typen (1)

...der Schlüssel zu (potentiell) unendlichen Datenbereichen.

Technisch:
...zu definierende Typnamen können rechtsseitig in der Definition benutzt werden.

Beispiele: ...Ausdrücke und Binärbäume

```
data Expr = Opd Int |
            Add Expr Expr |
            Sub Expr Expr |
            Squ Expr

data BinTree = Nil |
               Node Int BinTree BinTree
```

Rekursive Typen (2)

Beispiele ...für Ausdrücke.

```
Opd 42  -->  42
Add (Opd 17) (Opd 4)  -->  17+4
Add (Squ (Sub (Opd 42) (Squ (2)))) (Opd 12)
                                  --> square(42-square(2))+12
```

...rekursive Typen ermöglichen potentiell unendliche Datenstrukturen!

Rekursive Typen (3)

Beispiele ...für Funktionen über Binärbäumen.

```
valBinTree :: BinTree -> Int
valBinTree Nil = 0
valBinTree (Node n bt1 bt2) = n + valBinTree bt1 + valBinTree bt2

depthBinTree  :: BinTree -> Int
depthBinTree Nil = 0
depthBinTree (Node _ bt1 bt2)
                 = 1 + max (depthBinTree bt1) (depthBinTree bt2)
```

Mit diesen Definitionen...

```
valBinTree (Node 17 Nil (Node 4 Nil Nil)) = 21
depthBinTree (Node 17 Nil (Node 4 Nil Nil)) = 2
```

Rekursive Typen (4)

Spezialfall... wechselweise rekursive Typen

Beispiel:

```
data Person    = Adult Name Address Biography |
                 Child Name

data Biography = Parent String [Person] |
                 NonParent String
```

Polymorphe Typen (1)

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

Beispiele: ...Paare und Bäume

```
data Pairs a = Pr a a


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

Polymorphe Typen (2)

Mit diesen Definitionen...

```
Pr 17 4    :: Pairs Int
Pr [] [42] :: Pairs [Int]
Pr [] []   :: Pairs [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 Typen unterstützen Wiederverwendung!