5

Algebraische Datentypen

Wir haben in früheren Kapiteln schon gesehen, dass in einer funktionalen Sprache alle zusammengesetzten Datentypen wie Tupel, Listen und Bäume algebraische Datentypen sind.

In diesem Kapitel werden wir die verschiedenen Arten von algebraischen Datentypen – Aufzählungstypen, Produkttypen, Summentypen mit und ohne Rekursion – genauer ansehen.

5.1 Aufzählungstypen

In vielen Programmiersprachen können Aufzählungen von Werten als Datentypen definiert werden, z.B. die Wochentage als die Menge

$$\mathit{Weekday} = \{\mathit{Mo}, \mathit{Tu}, \mathit{We}, \mathit{Th}, \mathit{Fr}, \mathit{Sa}, \mathit{Su}\}$$

Die Werte von Aufzählungen werden häufig als implizit definierte ganzzahlige Konstanten betrachtet, z.B.:

$$\mathit{Mo} = 0, \mathit{Tu} = 1, \mathit{We} = 2, \mathit{Th} = 3, \mathit{Fr} = 4, \mathit{Sa} = 5, \mathit{Su} = 6$$

Machen wir uns schnell klar, dass die Typesynonyme von Haskell nicht geeignet sind, um Aufzählungstypen zu realisieren. Zwar könnte man definieren

type *Weekday* = *Int*

und auch die oben genannten Konstanten einführen, doch böte dies keinerlei *Typsicherheit*: Typsynonyme sind nur Abkürzungen, die wie Makros expandiert und dann "vergessen" werden. Die Wochentage sind also ganz gewöhnliche ganzzahlige Konstanten, die wie jeder andere Zahlenwert behandelt werden können. Die Ausdrücke `Fr +15` und `Fr * Mo` wären also zulässig, obwohl die Addition keinen Wochentag liefert und Multiplikation auf Wochentagen keinen Sinn macht.

Wir brauchen aber einen Datentyp, der nur Wochentage *Mo*, *Tu*, ..., *Su* als Werte enthält *und sonst nichts*, und wir wollen für diesen Typ nur Funktionen zulassen, die Sinn machen, *und sonst keine*. Wichtig ist, dass alle Wochentage unterschiedlich sind und nicht mit Werten anderer Datentypen verwechselt werden können. Das heißt, wir wollen *Datenabstraktion* betreiben.

Dies können wir mit der einfachsten Art von algebraischem Datentyp erreichen, dem *Aufzählungstyp*:

```
data Weekday = Mo | Tu | We | Th | Fr | Sa | Su
```

Mit dem Typ `Weekday` werden zugleich automatisch die Konstruktoren definiert:

```
Mo, Tu, We, Th, Fr, Sa, Su :: Weekday,
```

So, wie er hier definiert wurde, sind für den Typ `Weekday` keine weiteren Operationen vordefiniert. Funktionen können durch *pattern matching* definiert werden:

```
isWeekend :: Weekday → Bool
isWeekend Sa = True
isWeekend Su = True
isWeekend _  = False
```

Sicher gäbe es einige Operationen, die Sinn machten: Wochentage könnten verglichen, geordnet, als Zeichenkette gedruckt oder aus einer Zeichenkette gelesen werden, sie sind aufzählbar und beschränkt (in ihrer Wertemenge). Man könnte `Weekday` damit zu einer *Instanz* der Klassen `Eq`, `Ord`, `Show`, `Read` und `Bounded` machen und die entsprechenden Operationen für Wochentage definieren.

- Gleichheit und Ungleichheit lässt sich durch Mustervergleich realisieren.
- Als Ordnung nimmt man die Reihenfolge der Konstruktoren in der Typdefinition, in diesem Fall also `Mo` < `Tu` < `We` < `Th` < `Fr` < `Sa` < `Su`.
- Die Namen der Konstruktoren bilden ihre Darstellung als Zeichenkette, also "`Mo`" ... "`Su`", die von `show` ausgegeben bzw. von `read` gelesen wird.
- Die Funktionen `succ` und `pred` der Klasse `Enum` können aus der Ordnung der Konstruktoren abgeleitet werden, so dass dann Listen-Generatoren wie `[Mo..Fr]` oder `[Mo,We..Su]` gebildet werden können. Die Konstanten der Klasse `Bounded` können definiert werden als *minBound* = *Mo* und *maxBound* = *Su*.

Für den algebraischen Typ `Weekday` kann man eine *kanonische Instanz* dieser Klassen auch automatisch aus der Typdefinition *herleiten*. Dies kann man in Haskell so spezifizieren:

```
data Weekday = Mo | Tu | We | Th | Fr | Sa | Su
               deriving (Eq, Ord, Show, Read, Enum, Bounded)
```

Dann könnte die Funktion `isWorkday` auch implementiert werden mithilfe der Funktion `elem:: (Eq a) ⇒a → [a] →Bool` und des Listengenerators "`..`", der nur für Instanzen von `Enum` definiert ist:

```
isWorkday    :: Weekday → Bool
isWorkday d =  d `elem` [Mo .. Fr]
```

5.2 Produkttypen

Das *kartesische Produkt* fasst mehrere Werte zu einem einzigen zusammen:

$$P = T_1 \times \cdots \times T_k \qquad (k \geqslant 2)$$

In Haskell sind Produkttypen als *Tupeltypen* vordefiniert. Man könnte also schreiben:

$$\textbf{type}\ P = (T_1, \cdots, T_k)$$

Beispiel 5.1 (Koordinaten und komplexe Zahlen). Koordinaten in der Ebene und komplexe Zahlen können definiert werden als

```
type Point   = (Double,Double)
type Complex = (Double,Double)
```

Typdefinitionen führen jedoch nur Abkürzungen für Typausdrücke ein, die Typen von Werten `p:: Point`, `c:: Complex` und `x::(Double,Double)` können nicht unterschieden werden; es darf also jeder für den anderen eingesetzt werden.

Will man unterschiedliche Typen mit disjunkten Wertemenge definieren, muss man schreiben

```
data Point   = Point   Double Double
data Complex = Complex Double Double
```

Werte dieser Typen werden als `p=Point 0.0 0.0` bzw. `c=Complex 0.0 -1.0` geschrieben und können wegen der vorangestellten Konstruktoren

```
Point   :: Double → Double → Point
Complex :: Double → Double → Complex
```

nicht mit einander verwechselt werden. In Haskell besteht die Konvention, für den einzigen (Wert-) Konstruktor eines Produkttyps denselben Namen zu verwenden wie für den Typ (-Konstruktor) selber. Das ist nur auf den ersten Blick verwirrend.

Beispiel 5.2 (Datum). Ein *Datum* besteht aus Tag, Monat, und Jahr. Wir könnten es als Tupeltyp definieren:

```
type Date' = (Day, Month, Year)
type Day   = Int
data Month = Jan | Feb | Mar | Apr | May | Jun
           | Jul | Aug | Sep | Oct | Nov | Dec
             deriving (Eq, Ord, Show, Read, Enum, Bounded)
type Year  = Int
```

Damit hätten wir die ähnliche Probleme wie mit der Darstellung von Koordinaten und komplexen Zahlen: Werte jedes anderen Tupeltyps, der aus denselben Komponenten besteht, könnten mit den Werten von `Date'` verwechselt werden. Deshalb ist es besser, ein Datum als algebraischen Datentyp mit einem Konstruktor zu definieren:

```
data Date  = Date Year Month Day
             deriving (Eq, Ord, Show, Read, Bounded)
```

Für den Typ `Date` wird wieder automatisch der *Konstruktor* definiert, nur ist es in diesem Fall eine dreistellige Funktion:

```
Date :: Year→ Month→ Day→ Date
```

Wie bei den Wochentagen können wir `Date` als kanonische Instanz der Klassen `Eq`, `Ord`, `Show`, `Read` und `Bounded` ableiten lassen, da der in `Date` benutzte Typ (`Int`) ebenfalls Instanz dieser Klassen ist. Da bei der Ableitung der Vergleichsoperationen die lexikographische Ordnung verwendet wird, in der zunächst die ersten Komponenten verglichen werden und, wenn die gleich sind, die weiteren, haben wir die Reihenfolge der Komponenten umgedreht, so dass die Ordnung Sinn macht.

Beispielwerte vom Typ `Date` können jetzt anhand des Konstruktors von allen anderen Tupeltypen und Produkttypen mit anderen Konstruktornamen unterschieden werden:

```
today, bloomsday, fstday :: Date
today     = Date 2009 Nov 25
bloomsday = Date 1904 Jun 16
fstday    = Date 1 Jan 1
```

Funktionen für Datum können mit Mustervergleich definiert werden. Damit kann auf die Komponenten des Datums zugegriffen werden.

```
day  :: Date→ Day
day  (Date y m d) = d
year :: Date→ Year
year (Date y m d) = y
```

Der Mustervergleich kann für einen Produkttyp wie `Date` nie scheitern. Man braucht also jeweils nur eine Gleichung für die Definition.

Die oben definierten Funktionen `day`, `year` sind *Selektoren*:

```
day  today     = 26
year bloomsday = 1904
```

Ein Spezialfall von Produkten sind die *homogenen Produkte*, die aus k Komponenten des gleichen Typs bestehen. Mathematisch werden sie oft so geschrieben:

$$P = T^k \qquad (k \geqslant 2)$$

Hierbei kann man sich fragen, weshalb k größer als 1 sein muss. "Eintupel" definieren sicher keinen sinnvollen neuen Typ; deshalb werden Klammern ohne Kommata dazwischen auch einfach nur zur syntaktischen Gruppierung in Ausdrücken verwendet.

Aber was ist mit dem Fall $k = 0$? Das *Nulltupel* () definiert den *Einheitstyp*, der nur einen einzigen Wert enthält, der – genau wie der Typ selbst – mit () bezeichnet wird. Dieser Typ kann – genau wie der Typ **void** in JAVA – überall da eingesetzt werden, wo ein Typ hingeschrieben werden muss, aber kein "interessanter" Wert erwartet wird. Wir werden später sehen, dass es auch in Haskell solche Situationen gibt.

5.3 Summentypen

Summentypen setzen Wertemengen als *disjunkte Vereinigung* andere Wertemengen zusammen:

$$S = T_1 + \cdots + T_k \qquad (k \geqslant 2)$$

Hierbei können die T_i wieder zusammengesetzte Typen, insbesondere Produkttypen sein. Summentypen können als algebraischen Datentypen mit mehreren Konstruktoren definiert werden, wobei die Konstruktoren mehrstellig sein können.

Beispiel 5.3 (Geometrische Figuren). Eine *geometrische Figur* soll sein:

- entweder ein *Kreis*, gegeben durch Mittelpunkt und Durchmesser,
- oder ein *Rechteck*, gegeben durch zwei Eckpunkte,
- oder ein *Polygon*, gegeben durch die Liste seiner Eckpunkte.

Geometrische Figuren können so definiert werden:

```
type Point = (Double, Double)
data Shape = Circ Point Double
           | Rect Point Point
           | Poly [Point]
           deriving (Eq,Show,Read)
```

Für jede Art von Figur gibt es einen eigenen Konstruktor:

```
Circ :: Point→ Double→ Shape
Rect :: Point→ Point→ Shape
Poly :: [Point] Shape
```

Funktionen werden wieder durch *pattern matching* definiert. Die Funktion `corners` berechnet beispielsweise die Anzahl der Eckpunkte.

```
corners :: Shape→ Int
corners (Circ _ _)       = 0
corners (Rect _ _)       = 4
corners (Poly ps)        = length ps
```

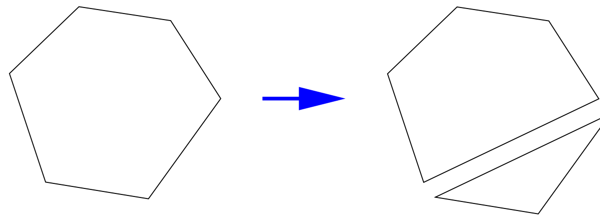
Die Translation eines Punktes wird durch Addition berechnet.

```
translation :: Point→ Point→ Point
translation (x, y) (dx, dy) = (x+ dx, y+ dy)
```

Damit kann die Verschiebung einer Figur so definiert werden:

```
move :: Shape→ Point→ Shape
move (Circ c r)    dp = Circ (translation c dp) r
move (Rect c1 c2)  dp = Rect (translation c1 dp) (translation c2 dp)
move (Poly ps)     dp = Poly [ translation p dp | p ← ps ]
```

Die Berechnung der Fläche ist für Kreise und Rechtecke einfach. Für Polygone beschränken wir uns aus Bequemlichkeit halber auf *konvexe* Polygone, und reduzieren die Berechnung durch Abspalten eines Teildreiecks rekursiv auf die Flächenberechnung eines einfacheren Polygons:



```
area :: Shape→ Double
area (Circ _ d) = pi∗ d
area (Rect (x1, y1) (x2, y2)) =
      abs ((x2- x1)∗ (y2- y1))
area (Poly ps) | length ps < 3  = 0
area (Poly (p1:p2:p3:ps)) =
           triArea p1 p2 p3 +
           area (Poly (p1:p3:ps))
```

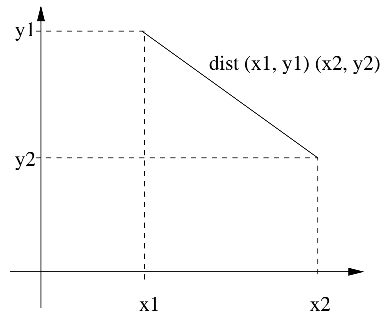
Dabei ergibt sich die Fläche eines Dreieck mit den Eckpunkten (x_1, y_1), (x_2, y_1), (x_3, y_3) als

$$A = \frac{|(x_2 - x_1)(y_3 - y_1) - (x_3 - x_1)(y_2 - y_1)|}{2}$$

Kodiert in Haskell sieht das so aus:

```
triArea :: Point→ Point→ Point→ Double
triArea (x1,y1) (x2,y2) (x3, y3) =
    abs ((x2-x1)∗(y3-y1) - (x3-x1)∗(y2-y1)) / 2
```

Die Distanz zwischen zwei Punkten ist nach Pythagoras so definiert:




```
dist :: Point→ Point→ Double
dist (x1, y1) (x2, y2) =
  sqrt((x1-x2)^2+ (y2- y1)^2)
```

Algebraische Datentypen können auch mit einer Typvariablen *parametrisiert* sein. Weiter unten werden wir Bäume als ein Beispiel für solche Typen betrachten. Hier wollen wir nur kurz einen vordefinierten Typ erwähnen, mit dem Fehler behandelt werden können.

Beispiel 5.4 (Maybe). Beim Programmieren kommt man oft in die Verlegenheit, dass eine Funktion in bestimmten Fehlersituationen keine gültige Ergebnisse ihres Ergebnistyps α liefern kann. Betrachten wir eine Funktion `lookup'`, die in einer Paarliste den Eintrag für einen Schlüssel sucht.

```
lookup' :: Eq a ⇒ a → [(a,b)] → b
lookup' k []       = ...          — ??
lookup' k ((k',e):xs)
    | k == k'    = e
    | otherwise = lookup' k xs
```

Welchen Wert soll sie liefern, wenn der Schlüssel in der Liste gar nicht auftaucht? Man könnte irgendeinen Wert von `b` als den *Fehlerwert* vereinbaren; das wäre aber nicht sehr sauber programmiert. Besser wäre es, den Ergebnistyp `b` um einen Fehlerwert zu erweitern. Dies kann man für beliebige Typen mithilfe des parametrisierten Summentyps `Maybe` tun, der in Haskell folgendermaßen vordefiniert ist

```
data Maybe a = Nothing | Just a deriving (Eq, Ord, Read, Show)
```

`Nothing` repräsentiert den Fehlerwert, und der Konstruktor `Just` kennzeichnet die "richtigen" Werte. Dann können wir die Funktion so definieren:

```
lookup :: Eq a ⇒ a → [(a,b)] → Maybe b
lookup k []      = Nothing
lookup k ((k',e):xs)
   | k == k'    = Just e
   | otherwise = lookup k xs
```

Brauchen wir aber nicht, denn sie ist vordefiniert.

Der parametrisierte Summentyp schlechthin ist der vordefinierte Typ `Either a b`, der entweder Werte des Typs `a` oder des Typs `b` enthält.

```
data Either a b = Left a | Right b deriving (Eq,Ord,Read,Show)
```

5.4 Rekursive algebraische Datentypen

In Summentypen kann der definierte Typ auf der rechten Seite benutzt werden; dann ist der Datentyp *rekursiv*. Funktionen auf solchen Typen sind meist auch rekursiv. (Rekursive Produkttypen machen dagegen keinen Sinn, weil es dann keinen Konstruktor gäbe, der einen Anfangswert erzeugen könnte.)

Beispiel 5.5 (Arithmetische Ausdrücke). Einfache arithmetische Ausdrücke bestehen entweder aus Zahlen (Literalen), oder aus der Addition bwz. der Subtraktion zweier Ausdrücke.

```
data Expr = Lit Int
          | Add Expr Expr
          | Sub Expr Expr
          deriving (Eq,Read,Show)
```

Betrachten wir Funktionen zum Auswerten und Drucken eines Ausdrucks:

```
eval :: Expr→ Int
eval (Lit n)= n
eval (Add e1 e2) = eval e1+ eval e2
eval (Sub e1 e2) = eval e1- eval e2

print :: Expr→ String
print (Lit n) = show n
print (Add e1 e2) = "("++ print e1++ "+"++ print e2++ ")"
print (Sub e1 e2) = "("++ print e1++ "-"++ print e2++ ")"
```

An diesen Beispielen können wir ein Prinzip der *primitive Rekursion* auf Ausdrücken erkennen, das der primitiven Rekursion aus Listen entspricht, die wir in Abschnitt 3.3.1 entdeckt haben. Primitiv rekursive Funktionen über Ausdrücken definieren eine Gleichung für jeden Konstruktor:

- Für Literale, die Rekursionsverankerung, wird ein Wert eingesetzt.

- Für die rekursiven Fälle, Addition, bzw. Subtraktion, werden jeweils binäre Funktionen eingesetzt.

Kombinatoren wie `map` und *`fold`* machen nicht nur Sinn für Listen. Sie können für viele algebraische Datentypen definiert werden, insbesondere wenn sie rekursiv sind.

Beispiel 5.6 (Strukturelle Rekursion über Ausdrücken). In Beispiel 5.5 haben wir ein Prinzip der primitiven Rekursion über Ausdrücken entdeckt, dass wir jetzt in einem Kombinator kapseln können.

Für jeden Wertkonstruktor des Typs *`Expr`* müssen wir beim Falten eine Funktion entsprechenden Typs angeben:

```
data Expr = Lit Int
          | Add Expr Expr
          | Sub Expr Expr
            deriving (Eq,Read,Show)

foldE :: (Int→ a)→ (a→ a→ a)→ (a→ a→ a) → Expr→ a
foldE b a s (Lit n)     = b n
foldE b a s (Add e1 e2) = a (foldE b a s e1) (foldE b a s e2)
foldE b a s (Sub e1 e2) = s (foldE b a s e1) (foldE b a s e2)
```

Damit kann die Auswertung und die Ausgabe so definiert werden:

```
eval' :: Expr→ Int
print':: Expr→ String

eval'  = foldE id (+) (-)
print' = foldE show (\s1 s2 → "("++ s1++ "+"++ s2++ ")")
                    (\s1 s2 → "("++ s1++ "-"++ s2++ ")")
```

5.5 Bäume

Bäume sind der rekursive algebraische Datentyp schlechthin. Ein binärer Baum ist hier

- entweder leer,
- oder ein Knoten mit genau *zwei* Unterbäumen, wobei die Knoten Markierungen tragen.

```
data Tree a = Null
            | Node (Tree a) a (Tree a)
              deriving (Eq, Read, Show)
```

Einen Test auf Enthaltensein kann man wie gewohnt mit Mustervergleich definieren:

```
member :: Eq a⇒ Tree a→ a→ Bool
member Null _ = False
member (Node l a r) b =
  a == b || (member l b) || (member r b)
```

Diese Funktion benutzt ein Schema für *primitive Rekursion* auf Bäumen:

- Der leere Baum bildet den Rekursionsanfang; dann wird ein Wert zurückgeliefert.
- Im Rekursionsschritt wird für jeden Knoten aus seiner Markierung und den rekursiv bestimmten Werten für die Unterbäume der Rückgabewert berechnet.

Auf die gleiche Weise können wir Funktionen zum *Traversieren* von Bäumen definieren, die alle Markierungen in einer Liste zusammenfassen:

```
preorder, inorder, postorder :: Tree a→ [a]
preorder Null          = []
preorder (Node l a r)  = [a] ++preorder l ++preorder r

inorder  Null          = []
inorder  (Node l a r)  = inorder l ++[a] ++inorder r

postorder Null          = []
postorder (Node l a r)  = postorder l ++postorder r++ [a]
```

Auch für Bäume können Kombinatoren definiert und benutzt werden.

Beispiel 5.7 (Strukturelle Rekursion auf Bäumen). Der Kombinator `foldT` kapselt Rekursion auf Bäumen.

```
foldT :: (a→ b→ b→ b)→ b→ Tree a→ b
foldT f e Null = e
foldT f e (Node l a r) = f a (foldT f e l) (foldT f e r)
```

Damit kann der Elementtest und eine `map`-Funktion für Bäume realisiert werden.

```
member' :: Eq a⇒ Tree a→ a→ Bool
member' t x =
   foldT (\e b1 b2→ e == x || b1 || b2) False t

mapT :: (a→ b)→ Tree a→ Tree b
mapT f = foldT (flip Node ∘ f) Null
```

Dabei ist `flip` die vordefinierte Funktion, die zwei Argumente eine Funktion vertauscht. Traversierung kann dann so beschrieben werden:

```
preorder, inorder, postorder :: Tree a→ [a]
preorder   = foldT (\x t1 t2→ [x] ++ t1  ++ t2 ) []
inorder    = foldT (\x t1 t2→  t1 ++ [x] ++ t2 ) []
postorder  = foldT (\x t1 t2→  t1 ++ t2  ++ [x]) []
```

Beispiel 5.8 (geordnete Bäume). Bäume können dazu benutzt werden, Mengen effizient darzustellen. Eine Voraussetzung dafür ist, dass die Elemente der Menge geordnet sind, d.h. in der Terminologie von Haskell, dass ihr Elementtyp `a` eine Instanz der Klasse `Ord` sein muss.

Für alle Knoten *Node a l r* eines *geordneten Baum* soll gelten, dass der linke Unterbaum nur kleinere und der rechte Unterbaum nur größere Elemente als `a` enthält:

$$\mathit{member}\ x\ l \Rightarrow x < a \wedge \mathit{member}\ x\ r \Rightarrow a < x$$

Eine Konsequenz dieser Bedingung ist, dass jeder Eintrag höchstens einmal im Baum vorkommt. Der Test auf Enthaltensein lässt sich dann so vereinfachen:

```
member :: Ord a => Tree a -> a -> Bool
member Null _ = False
member (Node l a r) b
  | b < a  = member l b
  | a == b = True
  | b > a  = member r b
```

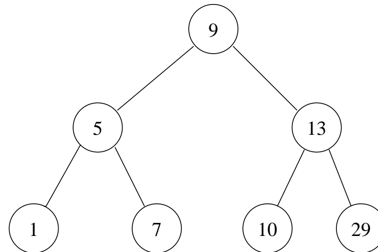
Die Funktion für ordnungserhaltendes Einfügen kann dann so definiert werden:

```
insert :: Ord a => Tree a -> a -> Tree a
insert Null a = Node Null a Null
insert (Node l a r) b
  | b < a  = Node (insert l b) a r
  | b == a = Node l a r
  | b > a  = Node l a (insert r b)
```

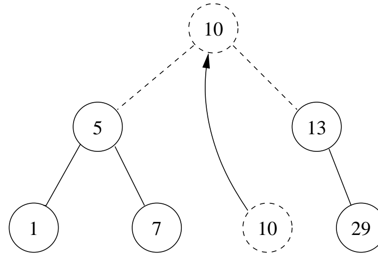
Etwas komplizierter ist das Löschen eines Elements:

```
delete :: Ord a => a -> Tree a -> Tree a
delete x Null = Null
delete x (Node l y r)
  | x < y  = Node (delete x l) y r
  | x == y = join l r
  | x > y  = Node l y (delete x r)
```

Sei beispielsweise folgender Baum gegeben:



Wenn wir danach `delete t 9` ausführen, wird die Wurzel von `t` gelöscht, und die beiden Unterbäume müssen von `join` ordnungserhaltend zusammengefügt werden:



Dabei wird der Knoten *links unten* im *rechten* Teilbaum (oder der rechts unten im linken Teilbaum) die Wurzel des neuen Baums.

Wir implementieren dazu eine Funktion `splitTree`, die den Baum in den Knoten links unten und den Rest aufspaltet.

```
join :: Tree a→ Tree a→ Tree a
join xt Null = xt
join xt yt   = Node xt u nu
   where
   (u, nu) = splitTree yt
   splitTree :: Tree a→ (a, Tree a)
   splitTree (Node Null a t) = (a, t)
   splitTree (Node lt a rt) = (u, Node nu a rt)
                              where (u, nu) = splitTree lt
```

Dabei nehmen wir an, dass `xt` nicht leer ist, wenn `yt` nicht leer ist.

5.6 Zusammenfassung

Die Definition eines algebraischen Datetypen $T\,\alpha_1 \cdots \alpha_k$ hat im Allgemeinen folgende Form:

$$
\begin{aligned}
\textbf{data}\ T\,\alpha_1 \cdots \alpha_k = {} & K_1\ \mathit{Typ}_{1,1} \ldots \mathit{Typ}_{1,k_1} \\
& \vdots \\
\mid {} & K_n\ \mathit{Typ}_{n,1} \ldots \mathit{Typ}_{n,k_n} \\
& \textbf{deriving}\ (C_1, \ldots, C_n)
\end{aligned}
$$

Hierin sind die $\mathit{Typ}_{i,j}$ Typausdrücke, in denen die Typvariablen $\alpha_1 \cdots \alpha_k$ und auch T vorkommen dürfen. Die Konstruktorfunktionen

$$K_i \ :: \ \mathit{Typ}_{i,1} \to \ldots \to \mathit{Typ}_{i,k_i} \to T\,\alpha_1 \cdots \alpha_k$$

sind verschieden von einander (und verschieden von allen anderen Konstruktoren algebraischer Datentypen).

Algebraische Datentypen haben drei wichtige Eigenschaften:

1. Die Konstruktoren sind *total*: Jede Anwendung eines Konstruktors auf Argumente geeigneten Typs liefert einen Wert.
2. Sie sind *Konstruktor-erzeugt*: Ihre Werte können ausschließlich durch die (verschachtelte) Anwendung von Konstruktoren aufgebaut werden.
3. Die Werte sind *frei erzeugt*: Für verschiedene Argumente liefert die Anwendung eines Konstruktors immer auch verschiedene Werte.

Beispiel 5.9 (Nochmal Bäume). Für den Datentyp `Tree a` folgert aus den oben definierten Eigenschaften:

1. Für alle Werte v vom Typ `a` und alle Bäume vom l, r vom Typ `Tree a` ist $t = \texttt{Node}\, v\, l\, r$ ein Baum (von Typ `Tree a`).
2. Jeder Baum $t :: \texttt{Tree}\, a$ hat entweder die Form $t = \texttt{Null}$ oder die Form $t = \texttt{Node}\, v\, l\, r$, wobei v ein Wert vom Typ `a` und l, r Bäume vom Typ `Tree a` sind.
3. Für Werte $t = \texttt{Node}\, v\, l\, r$ und $t' = \texttt{Node}\, v'\, l'\, r'$ mit $(v, l, r) \neq (v', l', r')$ gilt auch $t \neq t'$. (Umgekehrt folgt aus $t = t'$, dass $(v, l, r) = (v', l', r')$.)

Die erste Eigenschaft macht uns Probleme, wenn wir geordnete Bäume implementieren wollen. Mit den Konstruktoren können jederzeit beliebige, also auch ungeordnete Bäume erzeugt werden. Um die Ordnung zu erhalten, müssten die Konstruktoren `Null` und `Node` *verborgen* werden, und alleine `insert` als *Pseudo-Konstruktor* benutzbar sein. Diese Operation ist total, erzeugt die geordneten Bäume aber nicht frei, denn

$$\texttt{insert}\ (\texttt{insert}\ t\ x)\ x = \texttt{insert}\ t\ x$$

Im nächsten Kapitel werden wir sehen, wie geordnete Bäume als *abstrakte Datentypen* genau so realisiert werden können.