Unterabschnitt 3.1.12

Benutzerdefinierte Datentypen

Benutzerdefinierte Datentypen

Bisher betrachtet:

- Einfache Typen: Unit, Bool, Int, Integer, Float, Double, Char, String (?), ...
- Zusammengesetzte Typen: List, Tuple

Benutzerdefinierte Datentypen (2)

Fast alle modernen Spezifikations- und Programmiersprachen gestatten es dem Benutzer, „neue“ Typen zu definieren.

- Vereinbarung von Typbezeichnern
- Deklaration neuer Typen
- Summentypen
- Rekursive Datentypen

Haskell erlaubt es, Bezeichner für Typen zu deklarieren (vgl. F. 235):

Benutzerdefinierte Datentypen (3)

```
type IntPaar       =  (Int,Int) ;
type CharList      =  [Char] ;
type Telefonbuch   =
         [((String,String,String,Int),[String])] ;

type IntegerNachInteger  =   Integer -> Integer ;

fakultaet :: IntegerNachInteger ;
-- Argument muss >= 0 sein
fakultaet = fac ;
```

Dabei wird *kein* neuer Typ definiert, sondern nur ein “neuer” Bezeichner an einen bekannten Typ gebunden.

Bemerkungen: (Typvereinbarungen)

- Typvereinbarungen können zur Abkürzung oder zur Verdeutlichung benutzt werden (siehe Typ `Telefonbuch`).
- Zwei unterschiedliche Bezeichner können den gleichen Typ bezeichnen; z.B.:

```
type IntTriple =  (Int,Int,Int)
type Date      =  (Int,Int,Int)

kalenderwoche :: Date -> Int
-- Parameter muss existierenden Kalendertag sein
kalenderwoche (tag,monat,jahr) = ...

kalenderwoche (11,12,2003)
```

Deklaration neuer Typen

Neue Typen werden in Haskell mit der datatype-Deklaration definiert, die im Folgenden schrittweise erläutert wird.

Definition eines neuen Typs und *Konstruktors*:

```
data <NeuerTyp> =  <Konstruktor>  <Typ1> ... <TypN>
```

Die obige Datatypdeklaration definiert:

- einen neuen Typ und bindet ihn an <NeuerTyp>
- eine Konstruktorfunktion mit Signatur

  ```
  <Konstruktor>:: <Typ1> -> ... -> <TypN> -> <NeuerTyp>
  ```

  Die Konstruktorfunktion ist injektiv.

Typ- und Konstruktorbezeichner müssen mit einem Großbuchstaben beginnen.

Beispiel: (Definition von Typ, Konstruktor, Selektoren)

```
data Person = Student String String Int Int
```

definiert den neuen Typ `Person` und den Konstruktor

```
Student :: String -> String -> Int -> Int -> Person
```

Wir definieren dazu folgende *Selektorfunktionen* (Engl. ***accessors***):

```
vorname :: Person -> String
vorname (Student v n g m)      = v

name :: Person -> String
name (Student v n g m)         = n

geburtsdatum :: Person -> Int
geburtsdatum (Student v n g m) = g

matriknr :: Person -> Int
matriknr (Student v n g m)     = m
```

Bemerkungen:

Jede Datentypdeklaration definiert einen neuen Typ, d.h. insbesondere:

- die Werte des neuen Typs sind inkompatibel mit allen anderen Typen;
- auch Werte strukturgleicher benutzerdefinierter Typen sind inkompatibel.

Beispiele: (Typkompatibilität)

1. Der Typ `Person` ist inkompatibel mit dem Tupeltyp

   ```
   type Person2 =  (String,String,Int,Int)
   ```

   Insbesondere ist `vorname ("Niels","Bohr",18851007,221)` nicht typkorrekt.

2. `Person` ist inkompatibel mit dem strukturgleichen Typ `Adresse`:

   ```
   data Adresse = Wohnung String String Int Int
   ```

   Insbesondere ist

   ```
   name ( Wohnung "Casimirring" "Lautern" 27 67663 )
   ```

   nicht typkorrekt.

Bemerkung:

Den Konstruktor kann man sich als eine Markierung der Werte seines Argumentbereichs vorstellen.

Dabei werden Werte mit unterschiedlicher Markierung als verschieden betrachtet.

Konstruktoren erlauben es in gewisser Weise neue Produkttypen zu definieren.

Summentypen

Ein *Summentyp* stellt die disjunkte Vereinigung der Elemente anderen Typen zu einem neuen Typ dar.

Die meisten modernen Programmiersprachen unterstützen die Deklaration von Summentypen.

In Haskell definiert man Summentypen durch Angabe von Alternativen bei der datatype-Deklaration:

```
data <NeuerTyp> =
        <Konstruktor1>  <Typ1_1> ... <Typ1_N1>
    |   <Konstruktor2>  <Typ2_1> ... <Typ2_N2>
        ...
    |   <KonstruktorM>  <TypM_1> ... <TypM_NM>
```

Beispiele: (Summentypen)

1. Ein anderer Datentyp zur Behandlung von Personen:

```
data Person2 =
     Student     String String Int Int
   | Mitarbeiter String String Int Int
   | Professor   String String Int Int String
```

Beispiele: (Summentypen) (2)

2. Eine benutzerdefinierte Datenstruktur für Zahlen:

```
data MyNumber = Intc    Int
              | Floatc  Float

isInt :: MyNumber -> Bool
isInt (Intc m)      = True
isInt (Floatc r)    = False

isFloat :: MyNumber -> Bool
isFloat (Intc m)    = False
isFloat (Floatc r) = True

neg :: MyNumber -> MyNumber
neg (Intc m)    = Intc (-m)
neg (Floatc r) = Floatc (-r)
```

Beispiele: (Summentypen) (3)

```
plus :: MyNumber -> MyNumber -> MyNumber
plus (Intc m) (Intc n)     = Intc (m+n)
plus (Intc m) (Floatc r)   =
           Floatc ((fromInteger (toInteger m))+r)
plus (Floatc r) (Intc m)   =
           Floatc (r+(fromInteger (toInteger m)))
plus (Floatc r) (Floatc q) = Floatc (r+q)
```

Beispiele: (Summentypen) (4)

Bemerkung: Dies sind Zahlen mit “dynamic typing” und sie funktionieren ohne Überladen.

```
data Any =
    INT Int
  | FLOAT Float
  | NIL
  | CONS (Any,Any)
  | STRING String
```

Begriffsklärung:

Konstruktorfunktionen oder ***Konstruktoren*** (***constructors***) liefern Werte des neu definierten Datentyps. Sie können in Mustern verwendet werden (z.B.: Student, Intc).

Diskriminatorfunktionen oder ***Diskriminatoren*** prüfen, ob der Wert eines benutzerdefinierten Datentyps zu einer bestimmten Alternative gehört (Beispiel: isInt).

Selektorfunktionen oder ***Selektoren*** liefern Komponenten von Werten des definierten Datentyps (z.B.: vorname, name, . . .).

Begriffsklärung: (2)

Selektorfunktionen helfen mit ***Abstraktion*** von der ***Repräsentation***:
Vorher:

```
data Person = Student String String Int Int
vorname (Student v n g m)       = v

hallo0 (Student v n g m) = "hallo, "++v
hallo student = "hallo, " ++ (vorname student)
```

Nachher:

```
data Person = Student Int String String Int Int
vorname (Student k v n g m)       = v

hallo0 (Student v n g m) = "hallo, "++v   <--- FEHLER
hallo student = "hallo, " ++ (vorname student)
```

Weitere Operationen auf neu deklarierten Typen:

Konstruktoren und Selektoren erlauben das Aufbauen und Zerlegen der Werte neu deklarierter Typen. Durch den Zusatz:

```
deriving (Eq,Show)
```

liefert Haskell auch eine standardmäßige Gleichheit und die Möglichkeit, Werte des neuen Typs mittels `print` auszugeben. Zum Beipiel:

```
data MyNumber = Intc    Int
              | Floatc  Float
    deriving (Eq,Show)
```

Bemerkung:
Haskell ermöglich es dem Benutzer auch, die Gleichheit oder andere Operationen auf neu definierten Typen selbst zu definieren.

Weitere Anwendungen der datatype-Deklaration:

Die datatype-Deklaration kann auch verwendet werden, um ***Aufzählungstypen*** zu definieren, indem nur null-stellige Konstruktoren benutzt werden.

Die Wertemenge eines Aufzählungstyps ist eine endliche Menge (von Namen).

Beispiel: (Aufzählungstypen)

```
data Wochentag  =
    Montag   | Dienstag | Mittwoch | Donnerstag
  | Freitag | Samstag  | Sonntag
  deriving (Eq,Show)

istMittwoch :: Wochentag -> Bool
istMittwoch Mittwoch =  True
istMittwoch _        =  False
```

Oder knapper:

```
istMittwoch w  =  (w==Mittwoch)
```

Konstruktoren mit beliebiger Stelligkeit

In einer Datentypdeklaration können Konstruktoren mit beliebiger Stelligkeit kombiniert werden; z.B.:

```
data  MaybeInt =  Nothing
               |  Just Int
```

Haskell sieht dafür im Prelude den folgenden parametrisierten Typ vor (vgl. 3.3):

```
data Maybe a   =  Nothing
               |  Just a
```

Rekursive Datentypen

Von großer Bedeutung in allen Paradigmen der Programmierung sind rekursive Datentypen. Sie erlauben es insbesondere:

- Listen beliebiger Länge
- Bäume beliebiger Höhe

behandeln zu können.

Definition: (rekursive Datentypen)

Eine *Datentypdeklaration* heißt ***direkt rekursiv***, wenn der neu definierte Typ in einer der Alternativen der Datentypdeklaration vorkommt.

Wie bei Funktionen gibt es auch ***verschränkt rekursive*** Datentypdeklarationen.

Eine *Datentypdeklaration* heißt ***rekursiv***, wenn sie direkt rekursiv ist oder Element einer Menge verschränkt rekursiver Datentypdeklarationen ist.

Ein *Datentyp* heißt ***rekursiv***, wenn er mit einer rekursiven Datentypdeklaration definiert wurde.

Beispiele: (Listendatentypen)

1. Ein Datentyp für Integer-Listen:

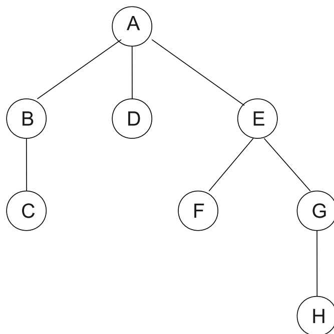
```
data  Intlist  =
                  Nil
              |   Cons Int Intlist
```

2. Ein Datentyp für homogene Listen mit Elementen von beliebigem Typ:

```
data List a  =
                  Nil
              |   Cons  a (List a)
```

Baumartige Datenstrukturen:

Ottmann, Widmayer:
„Bäume gehören zu den wichtigsten in der Informatik auftretenden Datenstrukturen“.



Begriffsklärungen: (zu Bäumen)

- In einem endlich verzweigten ***Baum*** hat jeder ***Knoten*** endlich viele ***Kinder***.
- Üblicherweise sagt man, die Kinder sind von *links nach rechts geordnet*.
- Einen Knoten ohne Kinder nennt man ein ***Blatt***, einen Knoten mit Kindern einen ***inneren*** Knoten oder ***Zweig***.
- Den Knoten ohne Elter nennt man ***Wurzel***.
- Ein Baum heißt ***markiert***, wenn jeder Knoten k eine Markierung $m(k)$ besitzt.
- In einem ***Binärbaum*** hat jeder Knoten maximal zwei Kinder.
- Zu jedem Knoten k gehört ein ***Unterbaum***, nämlich der Baum der k als Wurzel hat.

Datentyp für markierte Binärbäume:

```
data  IntBBaum  =
     Blatt  Int
  |  Zweig  Int IntBBaum IntBBaum
  deriving (Eq,Show)

einbaum = Zweig 7 (Zweig 3 (Blatt 2) (Blatt 4)) (Blatt 5)

mark :: IntBBaum -> Int
mark (Blatt n)         =  n
mark (Zweig n lk rk) =  n
```

Definition: (Sortiertheit markierter Binärbäume)

Ein mit ganzen Zahlen markierter Binärbaum heißt ***sortiert***, wenn für alle Knoten k gilt:

- Alle Markierungen der linken Nachkommen von k sind kleiner als $m(k)$.
- Alle Markierungen der rechten Nachkommen von k sind größer als $m(k)$.

Prüfung von Sortiertheit

Aufgabe:
Prüfe, ob ein Baum vom Typ `IntBBaum` sortiert ist.

Idee:
Berechne zu jedem Unterbaum die minimale und maximale Markierung und prüfe rekursiv die Sortiertheitseigenschaft für alle Knoten/Unterbäume.

```
maxmark, minmark :: IntBBaum -> Int
maxmark (Blatt n)         = n
maxmark (Zweig n lk rk) =
    n `max` (maxmark lk `max` maxmark rk)

minmark (Blatt n)         = n
minmark (Zweig n lk rk) =
    n `min` (minmark lk `min` minmark rk)
```

Prüfung von Sortiertheit (2)

```
istsortiert :: IntBBaum -> Bool
istsortiert (Blatt n)          = True
istsortiert (Zweig n lk rk) =
     istsortiert lk && istsortiert rk &&
     (maxmark lk)<n && n<(minmark rk)

result = istsortiert einbaum
```

Wenig effiziente Lösung! Besser ist es, die Berechnung von Minima und Maxima mit der Sortiertheitsprüfung zu verschränken.
Idee:
Entwickle eine Funktion istsortiert3 mit drei Ergebniswerten:

- Angabe, ob Baum sortiert
- minimale Markierung des Baums
- maximale Markierung des Baums

Prüfung von Sortiertheit (3)

```
istsortiert3 :: IntBBaum -> (Bool,Int,Int)
-- Sei  istsortiert3 b == (srt,mn,mx) ; dann ist
--   srt das Ergebnis der Sortiertheitspruefung von b
--   mn  die minimale Markierung von b
--   mx  die maximale Markierung von b

istsortiert3 (Blatt n)        = (True, n, n)
istsortiert3 (Zweig n lk rk) =
    let (lsrt, lmn, lmx) = istsortiert3 lk
        (rsrt, rmn, rmx) = istsortiert3 rk
    in  ((lsrt && rsrt && lmx<n && n<rmn), lmn, rmx)

istsortiert b = let (srtflag,_,_) = (istsortiert3 b)
                in srtflag
```

Begriffsklärung: (Weitere Begriffe zu Bäumen)

Der ***leere Baum*** ist ein Baum ohne Knoten.

Die ***Tiefe*** eines Knotens in einem Baum ist sein Abstand zur Wurzel. Der Wurzelknoten hat die Tiefe 0. Für alle anderen Knoten k gilt:

$$\text{tiefe}(k) = \text{tiefe}(\text{elternknoten}(k)) + 1$$

Die Knoten gleicher Tiefe t nennt man das ***Niveau*** t.

Die ***Höhe*** des leeren Baumes ist 0. Die Höhe eines nicht-leeren Baumes b ist die maximale Knotentiefe plus 1:

$$\text{höhe}(\ b\ ) = \max \{\ \text{tiefe}(k) \mid k \text{ Knoten von } b\ \} + 1.$$

Die ***Größe*** eines Baums ist die Anzahl seiner Knoten.

Datentyp möglicherweise leerer Binärbäume:

```
data IntBBaum2  =
     Leer
   | Knoten Int IntBBaum2 IntBBaum2
```

Bäume mit variabler Kinderzahl:

Bäume mit variabler Kinderzahl lassen sich realisieren:

- durch mehrere Alternativen für Zweige (begrenzte Anzahl von Kindern)
- durch Listen von Unterbäumen:

```
data VBaum = Kn Int [VBaum]  deriving (Eq,Show)
```

Der Rekursionsanfang ergibt sich durch Knoten mit leerer Unterbaumliste.

Bäume mit variabler Kinderzahl: (2)

```
zaehleKnVBaum    :: VBaum -> Int
zaehleKnVBaumLst :: [VBaum] -> Int

zaehleKnVBaum (Kn _ xs) =  1 + (zaehleKnVBaumLst xs)

zaehleKnVBaumLst []     = 0
zaehleKnVBaumLst (x:xs) =
           (zaehleKnVBaum x) + (zaehleKnVBaumLst xs)
```

Bemerkungen:

- Bäume mit variabler Kinderzahl lassen sich z.B. zur Repräsentation von Syntaxbäumen verwenden, indem das Terminal- bzw. Nichtterminalsymbol als Markierung verwendet wird.

  Besser ist es allerdings, die Information über die Symbole mittels Konstruktoren auszudrücken (vgl. nächstes Beispiel).

- Bäume mit variabler Kinderzahl werden auch zur Repräsentation von strukturierten oder semi-strukturierter Daten verwendet (z.B. XML, ...)

Beispiel: (abstrakte Syntaxbäume)

Der abstrakte Syntaxbaum eines Programms repräsentiert die Programmstruktur unter Verzicht auf Schlüsselworte und Trennzeichen.

Rekursive Datentypen eignen sich sehr gut zur Beschreibung von abstrakter Syntax.

Als Beispiel betrachten wir die abstrakte Syntax von Femto:

```
data Programm = Prog [Wertdekl] Ausdruck
     deriving (Eq,Show)
data Wertdekl = Dekl String Ausdruck
     deriving (Eq,Show)
data Ausdruck = Bzn String
              | Zahl Int
              | Add  Ausdruck Ausdruck
              | Mul  Ausdruck Ausdruck
     deriving (Eq,Show)
```

Beispiel: (abstrakte Syntaxbäume) (2)

Das Femto-Programm

```
a = 73;
main = print ( a + 12 )
```

Wird durch folgenden Baum repräsentiert:

```
Prog  [Dekl "a" (Zahl 73)]  (Add (Bzn "a") (Zahl 12))
```

Die Baumrepräsentation eignet sich besser als die Zeichenreihenrepräsentation zur weiteren Verarbeitung von Programmen.

Verschränkte Datentypdeklarationen:

Haskell unterstützt verschränkt rekursive Datentypdeklarationen.

Die Datentypdeklaration werden einfach hintereinander geschreiben (wie verschränkt rekursiven Funktionsdeklarationen).

Bei abstrakten Syntaxbäumen wird häufig verschränkte Rekursion der Datentypen benötigt.

Beispiel: (verschränkte Datentypen)

Als Beispiel betrachten wir die abstrakte Syntax einer Erweiterung von Femto um let-Ausdrücke:

```
data Programm = Prog [Wertdekl] Ausdruck
     deriving (Eq,Show)
data Wertdekl = Dekl String Ausdruck
     deriving (Eq,Show)
data Ausdruck = Bzn  String
              | Zahl Int
              | Add  Ausdruck Ausdruck
              | Mul  Ausdruck Ausdruck
              | Let  Wertdekl Ausdruck
  deriving (Eq,Show)
```

Die Deklaration von `Wertdekl` benutzt `Ausdruck`; die Deklaration von `Ausdruck` benutzt `Wertdekl`.

Unendliche Datenobjekte:

Zu einer nicht-leeren endlichen Liste kann man sich das erste Element und eine Liste als Rest geben lassen.

Hat die endliche Liste `xl` die Länge $n > 0$, dann hat `(tail xl)` die Länge $n - 1$.

Eine unendliche Liste besitzt keine natürlichzahlige Länge und wird durch Anwendung von `tail` nicht kürzer.

Haskell unterstützt unendliche Liste und andere unendliche Datenobjekte.

Begriffsklärung: (Strom) (***stream***)

Potenziell unendliche Listen (***infinite lists***) werden meist als ***Ströme*** (***stream***) bezeichnet.

In Haskell kann man den Listendatentyp zur Realisierung von Strömen verwenden.

Typische Operationen:

- Lesen des ersten Elements (head)
- Entfernen des ersten Elements (tail)
- Prüfen, ob noch Elemente im Strom vorhanden sind

Beispiel:
Strom von Eingabedaten

Beispiel: (unendliche Liste)

Eine vorstellbare unendliche Liste ist die Liste der natürlichen Zahlen `[0,1,2,3,4,5,...]`. In Haskell lässt sich diese Liste wie folgt definieren:

```
incr :: [Integer] -> [Integer]
incr []       =  []
incr (x:xs) =  (x+1):(incr xs)

natlist = 0:(incr natlist)

natlist2 = [0..]
```