3.3.6 Rekursive Datentypen

Die Rekursion von Datentypen bewirkt den Übergang zu abzählbar unendlichen Wertebereichen, deren Elemente nach einem festen Schema gleichartig aus einfacheren Elementen eines Wertebereichs aufgebaut sind. Demgegenüber erlauben die bisher eingeführten Konstruktoren nur die Definition von Datentypen fester endlicher Struktur. Einen rekursiven Datentyp D definiert man in ML mittels des Generalisationskonstruktors schematisch in der Form

datatype D=c_0 of D' | c of D''.

Hierbei bezeichnet D' den terminalen Datentyp mit dem Konstruktor c_0 und D'' mit Konstruktor c einen Datentyp, in dessen Definition wiederum D vorkommt.

Beispiele:

1) Wir definieren den Datentyp nat (natürliche Zahlen):

   datatype nat=null | succ of nat.

   Ein Element $succ^n(null)$, $n \geq 0$, repräsentiert die natürliche Zahl n. Zur Konversion der Objekte $succ^n(null)$ in die lesbare Darstellung n und umgekehrt eignen sich die Funktionen

```
fun  conv_to_int null = 0 |
     conv_to_int (succ x) = 1+ (conv_to_int x);
fun  conv_to_nat 0 = null |
     conv_to_nat n = succ (conv_to_nat (n-1)).
```

   Man beachte, daß conv_to_nat für n<0 nicht definiert ist.

2) Ein Text über dem Alphabet {a,b} ist entweder das leere Wort eps, oder er besteht aus einem einzelnen Zeichen a oder b, dem ein Text folgt. Wir definieren daher

```
datatype zeichen= a | b;
datatype text=eps | conc of zeichen*text.
```

   Elemente des Typs text sind also z.B.

```
eps, conc(a,eps), conc(a,conc(b,conc(a,eps)))
 ↓       ↓                  ↓
 ε       a                 aba
```

   Verallgemeinern wir den Datentyp text durch Übergang zu einem Text über einem beliebigen Alphabet, so definieren wir den polymorphen Typ

```
datatype 'alphabet text=eps | conc of 'alphabet * 'alphabet text.
```

   Den obigen Typ text über {a,b} erhalten wir dann auch durch

```
type abtext=zeichen text.
```

   Typische Funktionen auf Texten sind z.B.:

   Erstes Zeichen eines Textes: `fun first(conc(x,y))=x;`

   Text ohne das erste Zeichen: `fun rest(conc(x,y))=y;`

Konkatenation zweier Texte:

```
fun append eps x = x |
    append (conc(x,y)) z = conc(x,append(y,z)).
```

Gleichheit auf Datentypen.
Auf jedem Datentyp, vorausgesetzt es handelt sich um einen Gleichheitstyp, ist in ML automatisch die Gleichheitsoperation vordefiniert. Diese stimmt häufig nicht mit der intendierten Gleichheit der Objekte des Typs überein.

Beispiel: Wir definieren den Typ der ganzen Zahlen int:

```
datatype int=null | succ of int | pred of int.
```

Hier ist nun +n standardmäßig durch $succ^n(null)$ und -n durch $pred^n(null)$ repräsentiert. Es gibt aber noch unendlich viele weitere Repräsentationen einer Zahl, z.B. für die Null:

null, succ(pred(null)), pred(succ(null)), $pred^n(succ^n(null))$, $n \geq 0$, usw.

Die vordefinierte Gleichheitsoperation unterscheidet jedoch die einzelnen Repräsentationen der Null, denn jeder Konstruktor erzeugt ein eindeutiges Objekt, das verschieden ist von allen Objekten, die auf eine andere Art erzeugt wurden; es gilt also

$null \neq succ(pred(null)) \neq pred(succ(null))$ usw.

Die ML-Sicht des Datentyps int stimmt also nicht mit der wirklichen Struktur des Typs int überein.

Wenn jedes der durch einen Datentyp beschriebenen Objekte durch genau ein Datenobjekt dargestellt werden kann, dann spricht man von einer *freien* Repräsentation, anderenfalls von einer *unfreien*. Verwendet man für die Beschreibung von Objekten eine freie Repräsentation, so stimmt die vordefinierte Gleichheitsrelation mit der intendierten Gleichheit überein, anderenfalls muß man selbst eine neue Gleichheitsrelation definieren, die alle Datenobjekte gleichmacht, die das gleiche Objekt darstellen. Eine Menge von Objekten heißt *frei*, wenn es eine freie Repräsentation der Menge gibt.

3.3.6.1 Lineare Listen

Den obigen Typ text mit den angegebenen Operationen first, rest, append kann man auch als (polymorphe) lineare Liste auffassen. Dieser Typ wird von ML auch standardmäßig unter der Typfunktion list zur Verfügung gestellt. Sind $d_1,...,d_n$ Objekte eines beliebigen Datentyps Δ, so faßt der Konstruktor $[\,\cdot\,,...,\,\cdot\,]$ diese zu einer Liste

$[d_1,...,d_n]: \Delta$ list

zusammen. Die Konstanten

[ ] oder nil

repräsentieren die leere Liste.

Beispiele: [2,3]: int list
[(2,3),(4,5)]: (int*int) list
[sin,sqr,cos]: (real -> real) list

Zur Selektion von Listenelementen kann man die Standardfunktionen (s.u.) heranziehen, oder man verwendet die elegantere Methode des Pattern-Matching. Die hierfür wichtigste Operation auf Listen ist die Infix-Operation

:: : $\Delta \times \Delta^* \rightarrow \Delta^*$ mit

$a::b=[a,b_1,...,b_k]$, falls $b=[b_1,...,b_k]$, $k \geq 0$,

die aus einem Objekt a von Typ Δ und einer Liste b über Δ eine neue Liste generiert, in der a erstes Element vor den Elementen von b ist.
Mit der Konstanten [ ] und dieser Operation kann man Listen also als einen Datentyp auffassen, der wie folgt definiert ist:

```
datatype 'a list= nil | :: of 'a*'a list.
                        ↑  ↑
                        [ ]   infix-Konstruktor
```

Standardfunktionen auf Listen.
Test auf Leerheit:

```
fun  null [ ] = true |
     null (_::_) = false.
```

Erstes Element (head) einer Liste:

```
fun  hd (x::_) = x.
```

Liste ohne ihr erstes Element (tail):

```
fun  tl (_::x) = x.
```

Länge einer Liste:

```
fun  len [ ] = 0 |
     len (_::x) = 1+len(x).
```

Konkatenation zweier Listen:

```
fun  append [ ] x = x |
     append (x::y) z = x::(append y z).
```

Hier kann man auch die vordefinierte Infixoperation @ verwenden.
Elementtest:

```
fun  member (x::_) x = true |
     member (_::y) x = member x y.
```

Standardfunktionale auf Listen.

Generation von Listen: Ausgehend von einem Anfangswert a wird mithilfe einer Funktion f eine Liste der Form

$[a,f(a),f(f(a)),...,f^k(a)]$

bis zu einem Index k gebildet:

```
fun  generate f a 0 = [a] |
     generate f a k = [a]@generate f (f a) (k-1).
```

Beispiel:

```
generate (fn x => x+1) 0 10;
> [0,1,2,3,4,5,6,7,8,9,10]: int list
```

Transformation einer Liste, d.h. Anwendung einer Funktion f auf alle Listenelemente:

```
fun  map f [ ] = [ ] |
     map f (x::y) = (f x)::(map f y).
```

Beispiele:

```
map sin [0.1,0.2,0.3,0.4,0.5];
> [0.0998,0.1987,0.2955,0.3894,0.4794]: real list
map (fn x => x+1) [1,2,3,4];
> [2,3,4,5]: int list
map (fn x => (x,2*x)) [1,2,3,4];
> [(1,2),(2,4),(3,6),(4,8)]: (int*int) list
```

Akkumulation von Listenelementen: Gesucht ist eine Funktion akku, die zu einer Funktion f, einem Anfangswert a und einer Liste $[x_1,...,x_n]$ den Wert

$f(x_1,f(x_2,...,f(x_n,a)...))$

berechnet:

```
fun  akku f a [ ] = a |
     akku f a (x::y) = f(x,akku f a y).
```

Beispiel:

```
val sum = akku (fn (x,y) => x+y) 0;
sum [7,8,10,17];
> 42: int
```

Heraus*filtern* aller Elemente einer Liste, die ein bestimmtes Prädikat p erfüllen:

```
fun  filter p [ ] = [ ] |
     filter p (x::y) = if p x then x::(filter p y) else filter p y.
```

Beispiel:

```
filter (fn x => x>0) [-7,-3,8,-2,5,3,6,-1];
> [8,5,3,6]: int list
```

Existenzquantor. Existieren Elemente in einer Liste, die das Prädikat p erfüllen:

```
fun  exists p [ ] = false |
     exists p (x::y) = (px) orelse (exists p b).
```

Beispiel:

```
exists (fn x => x<0) [-7,-3,8,-2,5,3,6,-1];
> true: bool
```

Lazy lists.

Lazy lists sind in der in Abschnitt 2.2.5.1 besprochenen Form nur möglich in funktionalen Programmiersprachen, die als Substitutionsregel call-by-need verwenden. In ML mit seiner strikten Auswertungsstrategie kann man daher lazy lists nicht unmittelbar definieren. Man kann sich jedoch eines Tricks bedienen, um die Parameterauswertung zumindest zu verzögern. Zur Motivation betrachte man die Funktion

```
fun nonsense x y = if x then y else 0.
```

Der Aufruf

```
nonsense false (2.0/0.0)
```

führt in ML, nicht aber in Sprachen mit lazy evaluation, zu einem Laufzeitfehler, weil der Ausdruck 2.0/0.0 bereits beim Aufruf ausgewertet wird, obwohl er im Rumpf gar nicht benötigt wird. Diese Auswertung kann man nun verzögern, indem man den Ausdruck in einer (parameterlosen, besser auf dem Typ unit definierten) Funktion verbirgt:

```
fun junk ()=2.0/0.0
```

und die Funktion nonsense geeignet anpaßt:

```
fun nonsense x y = if x then y() else 0.
```

Beim Aufruf

```
nonsense false junk
```

wird nun junk zwar übergeben, jedoch nicht aufgerufen, weil x=false ist.
Analog geht man auch bei lazy lists in ML vor und definiert

```
datatype 'type lazylist = empty |
        c of 'type * (unit -> 'type lazylist).
```

Hier wird also von der naiven Definition

```
c of 'type * 'type lazylist
```

abgewichen und der Rest einer Liste in einer unit-Funktion versteckt und damit die Auswertung verzögert.
Eine lazy list ist also entweder leer (empty), oder sie hat die Form

```
c(x,y),
```

wobei x das erste Element ist und y eine Funktion ist, die den Rest der Liste berechnet. Man beachte den Unterschied: Bei endlichen Listen *ist* y der Rest der Liste, bei lazy lists *berechnet* y den Rest der Liste.
Die beiden Standardfunktionen head und tail überträgt man nun leicht:

```
fun head (c(x,_))=x;
> val head=fn: 'a lazylist -> 'a
fun tail (c(_,y))=y();
> val tail=fn: 'a lazylist -> 'a lazylist.
```

Wir wollen nun die Standardfunktion :: von endlichen Listen auf lazy lists übertragen. Die zugehörige Funktion erscheint auf den ersten Blick ungewohnt: Ist x ein Element und y eine lazy list, so definiert man die Liste x gefolgt von y durch:

```
fun colon x y=c(x,fn () => y).
```

Wie erwähnt besteht eine lazy list aus ihrem ersten Element und einer Funktion, die den Rest der Liste berechnet. Ist nun y eine lazy list, an die vorne das Element x angefügt werden soll, so ist y anschließend der Rest der neuen Liste. Wir müssen also aus dem Rest eine Funktion machen, die y berechnet. Gerade dies leistet die Funktion fn () => y. Diese Funktion colon besitzt jedoch wenig praktischen Nutzen, da man einen Ausdruck der Form colon(x,E) in ML nicht verwenden darf, weil E nicht lazy sondern strikt (also beim Aufruf) ausgewertet wird. Dies wird auch nicht durch den Ausdruck fn () => E im Rumpf von colon verhindert. Anstelle von colon(x,E) muß man also stets c(x, fn () => E) schreiben.

Beispiele: Wir stellen im folgenden zur Veranschaulichung die korrekte Darstellung in ML einer lazy-Darstellung gegenüber, also einer Darstellung, die davon ausgeht, ML verwende lazy evaluation.

1) Wir definieren die bereits aus Abschnitt 2.2.5.1 bekannte Funktion from.

   Lazy Darstellung in ML-Notation:

   ```
   fun from n = n::(from (n+1)).
   ```

   Korrekte Darstellung in ML:

   ```
   fun from n = c(n,fn () => from(n+1)).
   ```

2) Die ebenfalls aus 2.2.5.1 bekannte Funktion take definiert man wie folgt.

   Lazy Darstellung in ML-Notation:

   ```
   fun  take 0 l = [ ] |
        take k [ ] = [ ] |
        take k (x::y)=x::(take (k-1) y).
   ```

   Korrekte Darstellung in ML:

   ```
   fun  take 0 l = [ ] |
        take k empty = [ ] |
        take k (c(x,y))=x::(take (k-1) (y())).
   ```

   Anwendung:

   ```
   take 4 (from 10);
   > [10,11,12,13]: int list
   ```

   Wir betrachten die Auswertung von take 2 (from 10):

   ```
   take 2 (from 10) => take 2 (c(10, fn () => from (10+1)))
        => 10 :: (take 1 (from (10+1)))
        => 10 :: (take 1 (c(11, fn () => from (11+1))))
        => 10 :: 11 :: (take 0 (from (11+1)))
   ```

=> 10 :: 11 :: (take 0 (c(12, fn () => from (12+1))))
=> 10 :: 11 :: [ ] => [10,11].

3) Folge aller Quadratzahlen:

```
fun  squares empty = empty |
     squares (c(x,y)) = c(x*x:int, fn () => squares (y())).
```

Standardfunktionale auf lazy lists.

Die bereits auf Listen definierten Funktionale zur Generierung, Transformation, Akkumulation und zum Filtern übertragen sich auf lazy lists in natürlicher Weise:

Generation einer lazy list, also Erzeugen der unendlichen Liste

[a,f(a),f(f(a)),f(f(f(a))),...]

für eine Funktion f und einen Startwert a erfolgt durch:

```
fun  generate f a = c(a,fn () => generate f (f a)).
```

Beispiel:

```
val  from = generate (fn x => x+1).
```

Transformation einer lazy list, d.h. Erzeugung der Liste

[f(x_1),f(x_2),f(x_3),...]

aus der Liste [x_1,x_2,x_3,...] wird realisiert durch:

```
fun  map f empty = empty |
     map f (c(x,y)) = c((f x),fn () => map f (y())).
```

Beispiele:

```
take 5 (map (fn x => (x,2*x)) (from 7));
> [(7,14),(8,16),(9,18),(10,20),(11,22)]: (int*int) list
```

Akkumulation, also Berechnung der Liste

[f(a,x_1),f(f(a,x_1),x_2), f(f(f(a,x_1),x_2),x_3),...]

aus der Liste [x_1,x_2,x_3,...]:

```
fun  akku f a empty = empty |
     akku f a (c(x,y)) =c(f(a,x),fn () => akku f (f(a,x)) (y())).
```

Beispiel:

```
val  teilsumme = akku (fn (x,y) => x+y) 0;
take 3 (teilsumme (from 17));
> [17,35,54]: int list
```

Filtern einer lazy list nach dem Prädikat p:

```
fun  filter p empty = empty |
     filter p (c(x,y)) = if p x then c(x,fn () => filter p (y())) else filter p (y()).
```

Beispiel:

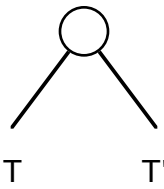
```
take 2 (filter (fn x => x mod 2=0) (teilsumme (from 17)));
> [54,74]: int list
```

3.3.6.2 Bäume

Betrachtet man die natürliche Definition für geordnete binäre Bäume:

(1) Die leere Struktur ist ein Baum.

(2) Sind T und T' Bäume, so auch



so ergibt sich daraus bereits unmittelbar eine Datentypdefinition in ML:

```
datatype bintree = empty | node of bintree*bintree.
```

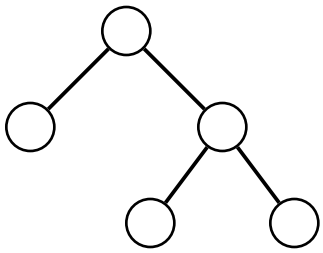
Blätter (äußere Knoten) in einem Baum sind dann Objekte der Form

```
node(empty,empty).
```

Die übrigen Knoten sind *innere* Knoten.

Beispiele:

node(empty,empty): bintree



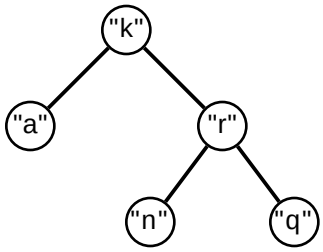
node(node(empty,empty),
node(node(empty,empty),node(empty,empty))): bintree

Von größerer Bedeutung für die Praxis sind markierte binäre Bäume. Als Markierung lassen wir beliebige Datentypen zu, wir definieren daher einen polymorphen Datentyp

```
datatype 'label bintree = empty | node of 'label bintree*'label*'label bintree.
```

Beispiele:

node(empty,1,empty): int bintree



node(node(empty,"a",empty),"k",
node(node(empty,"n",empty),"r",node(empty,"q",empty))):
string bintree

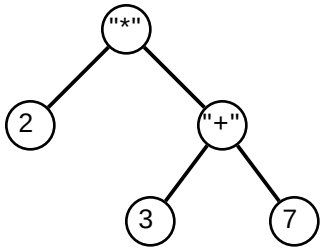
Eine weitere Verallgemeinerung bilden markierte binäre Bäume, bei denen die Blätter und die inneren Knoten Markierungen unterschiedlichen Datentyps besitzen:

```
datatype ('leaflabel,'label) bintree = leaf of 'leaflabel |
        node of ('leaflabel,'label) bintree*'label*('leaflabel,'label) bintree.
```

Ein typisches Anwendungsgebiet dieser Datenstruktur sind Syntaxbäume für arithmetische Ausdrücke.

Beispiele:

leaf(1): (int,'label) bintree



node(leaf(2),"*",node(leaf(3),"+",leaf(7))): (int,string) bintree

Baumdurchlauf.

Ein Baumdurchlauf liefert die Markierungen der Knoten eines Baums als lineare Liste. Bekannte Durchläufe sind inorder, preorder, postorder. In ML formuliert man z.B. den inorder-Durchlauf auf Objekte vom Typ 'label bintree wie folgt:

```
fun  inorder empty = [ ] |
     inorder (node(left,label,right)) = inorder(left)@[label]@inorder(right).
```

Beispiel: Eine Anwendung des inorder-Durchlaufs ist die Funktion swap, die den rechten und den linken Sohn jedes Knoten vertauscht:

```
fun  swap empty = empty |
     swap (node(left,label,right)) = node(swap right,label,swap left).
```