3.3 Strukturierte Datentypen

Im letzten Abschnitt haben wir schon zwei der wichtigsten Typkonstruktoren von ML angeschnitten, die Bildung von Paaren

D * D'

und die Bildung von Funktionsräumen

$D \rightarrow D'$

für zwei beliebige Datentypen D und D'.
Im folgenden besprechen wir diese und die übrigen Konstruktoren genauer.

3.3.1 Enumeration

Enumeration ist ein Konstruktor, mit dem man durch Einführung endlicher Mengen neue Datentypen definieren kann, indem man alle zugehörigen Elemente auflistet. Man erhält einen sog. *Aufzählungstyp*. In ML definiert man einen Aufzählungstyp D schematisch durch

datatype D= d_1 | d_2 | ... | d_n.

d_1, d_2, ..., d_n sind die Elemente des Wertebereichs von D. Hierbei ist jedes d_i explizit anzugeben. In Form eines abstrakten Datentyps lautet D:

```
type D =
    sorts Dsorte
    functions
        d1: →Dsorte
        ...
        dn: →Dsorte
    laws
end.
```

Faktisch ist die Enumeration in ML nur ein Spezialfall der Generalisation (s. 3.3.5).

Beispiele:

1)
```
datatype farbe = blau | gruen | rot;
> datatype farbe = blau | gruen | rot
    con rot=rot: farbe
    con blau=blau: farbe
    con gruen=gruen: farbe
```

2) Der elementare Standardtyp bool ist ein Aufzählungstyp und definiert durch

```
datatype bool = true | false;
```

Die zugehörige not-Funktion lautet:

```
fun not x = if x then false else true;
```

3.3.2 Aggregation

Aggregation nennt man das Zusammensetzen mehrerer (möglicherweise verschiedener) Datentypen $D_1,...,D_n$ zu einem n-Tupel. Schematisch

type $D \equiv (D_1,...,D_n)$.

Mathematisch ist D das kartesische Produkt der Typen $D_1,...,D_n$. $D_1,...,D_n$ heißen *Komponenten* von D. Die Elemente von D sind n-Tupel der Form $d=(d_1,...,d_n)$ mit d_i vom Typ D_i für $1 \le i \le n$. Gilt $D_1=...=D_n$, so nennt man D *homogen*, anderenfalls *inhomogen*.

ML besitzt kein besonderes Sprachelement für die Unterscheidung von homogenen und inhomogenen Aggregationen, wie sie in anderen Sprachen üblich sind (array versus record). Vielmehr unterscheidet ML zwischen

- anonymen unter Verwendung des polymorphen Standardkonstruktors $(\cdot,\cdot,...,\cdot)$ gebildeten **Tupeln**,
- selbstdefinierten mit einem Bezeichner versehenen **Tupeln**,
- **Records**.

Der Unterschied zwischen Tupeln und Records besteht darin, daß die einzelnen Komponenten bei Records durch frei wählbare Bezeichner selektiert werden können. Bei Tupeln ist dagegen kein direkter Zugriff auf die Komponenten möglich, sie können nur indirekt über das sog. Pattern-Matching (s. 3.3.3) selektiert werden.

Tupelbildung.

Sind $d_1,...,d_n$ Werte der beliebigen Datentypen $D_1,...,D_n$, so bildet man durch

$(d_1,...,d_n)$

ein Standardtupel des Datentyps

$D=D_1 \times ... \times D_n$.

D definiert man in ML durch

type $D=D_1*...*D_n$.

Beispiele:

1) Bruchrechnung:

```
type rational=int*int
(2,7);
> (2,7): int*int
```

Hier kann ML natürlich nicht unterscheiden, ob es sich bei dem Paar um eine rationale Zahl im Sinne der Typdefinition oder um ein Paar ganzer Zahlen handeln soll.

```
(2,7): rational;
```

```
> (2,7): rational
fun bruchmult((a,b): rational,(a',b'): rational )=(a*a',b*b'): rational;
> val bruchmult=fn: rational * rational -> rational
bruchmult((2,7),(3,6));
> (6,42): rational
```

2) Die polymorphe Funktion paar bildet aus zwei Argumenten x,y zweier beliebiger Datentypen das Paar (x,y):

```
fun paar x y=(x,y);
> val paar=fn: 'a -> ('b -> 'a*'b)
paar true (paar 2 "x");
> (true, (2, "x")): bool*(int*string)
```

3) Die folgende polymorphe Funktion verschiebt die Elemente eines Tripels zyklisch nach rechts:

```
fun cycleshift (x,y,z)=(z,x,y);
> val cycleshift=fn: 'a*'b*'c -> 'c*'a*'b
cycleshift((2,3),18.4,(true,false));
> ((true,false),(2,3),18.4): (bool*bool)*(int*int)*real
```

Selbstdefinierte mit einem Bezeichner versehene Tupeltypen D über den Datentypen $D_1,...,D_n$ definiert man allgemein in der Form

```
datatype D = c of D1*...*Dn.
```

Hierbei ist c der frei gewählte Bezeichner des Konstruktors für Objekte aus D, also eine Abbildung

$$c: D_1 \times ... \times D_n \rightarrow D.$$

Beispiel:

```
datatype rational = Bruch of int*int;
> con Bruch=fn: int*int -> rational
Bruch(2,7);
> Bruch(2,7): rational
fun Bruchmult (Bruch(a,b),Bruch(a',b'))=Bruch(a*a',b*b');
> val Bruchmult=fn: rational*rational -> rational
Bruchmult(Bruch(2,7),Bruch(3,6));
> Bruch(6,42): rational
```

Auf die einzelnen Komponenten eines aggregierten Tupeltyps kann man nicht direkt zugreifen. Es gibt also keine *Selektoren*. Ein indirekter Zugriff ist über *Pattern-Matching* möglich. Man verwendet hierbei als formalen Parameter keinen einzelnen Bezeichner, sondern ein Muster. Die einzelnen Bestandteile des aktuellen Parameters werden dann mit den Symbolen des Musters assoziiert.

Beispiel: In der obigen Funktion Bruchmult haben wir bereits intuitiv als ersten formalen Parameter das Muster

Bruch(a,b)

verwendet. Der aktuelle Parameter

Bruch(2,7)

wird beim Aufruf der Funktion mit dem formalen "gematcht"

```
Bruch ( a , b )
   ↓   ↓ ↓
Bruch ( 2 , 7 )
```

Hiebei werden die entsprechenden Variablenbindungen a=2, b=7 hergestellt. Auf die genauen Abläufe beim Pattern-Matching gehen wir in Abschnitt 3.3.3 ein.

Beispiel: Selektoren eine aggregierten Typs kann man nun simulieren, z.B. durch

```
fun Zaehler(Bruch(a,b))=a;
> val Zaehler=fn: rational -> int
```

oder allgemein für ein Paar durch polymorphe Projektionsfunktionen

```
fun Px(x,y)=x;
> val Px=fn: 'a*'b -> 'a
fun Py(x,y)=y;
> val Px=fn: 'a*'b -> 'b
```

Records.

Einen Recordtyp D über den Typen $D_1,...,D_n$ definiert man schematisch in der Form

type D = {s_1:D_1, s_2:D_2, ..., s_n:D_n}.

$s_1,...,s_n$ sind die Bezeichner der einzelnen Komponenten von D (die *Selektoren*).

Beispiel:

```
type rational={Zaehler:int, Nenner:int};
> type rational={Nenner:int, Zaehler:int}
val Bruch1={Zaehler=2, Nenner=7};
> val Bruch1= {Nenner=7, Zaehler=2} : {Nenner:int, Zaehler:int}
val Bruch2={Zaehler=3, Nenner=6};
> val Bruch2= {Nenner=6, Zaehler=3} : {Nenner:int, Zaehler:int}
```

Eine Komponente eines Records selektiert man, indem man dem Komponenten-bezeichner ein # voranstellt. Dann erhält man eine Funktion, die als Parameter den Bezeichner eines Records erwartet.

Beispiel:

```
#Zaehler Bruch2;
> 3:int
val Bruch1malBruch2={Zaehler=(#Zaehler Bruch1) * (#Zaehler Bruch2),
                     Nenner=(#Nenner Bruch1) * (#Nenner Bruch2)};
> val Bruch1malBruch2= {Nenner=18, Zaehler=6} : {Nenner:int, Zaehler:int}
fun Bruchmult(x:rational,y:rational)={Zaehler=(#Zaehler x) * (#Zaehler y),
                     Nenner=(#Nenner x) * (#Nenner y)}: rational;
> val Bruchmult=fn: rational*rational -> rational
```

Bemerkung: Zwischen Standardtupeln und Records besteht in ML eine enge Beziehung: Tupel der Form

$d=(d_1,...,d_n)$: $D_1*...*D_n$

werden implizit als Records aufgefaßt, wobei die Selektoren den Indizes der Tupel entsprechen. Das Tupel d ist also äquivalent zu einer Wertdeklaration der Form

val $d=\{1=d_1,2=d_2,...,n=d_n\}$.

Auf die einzelnen Komponenten eines Tupels kann dann wie bei Records zugegriffen werden, z.B.

```
#3 d;
> d3:D3.
```

In diesem Abschnitt haben wir zwei Formen von Typdeklarationen kennengelernt, die type-Deklaration und die datatype-Deklaration. Während die datatype-Deklaration einen neuen Datentyp zusammen mit benutzerdefinierten Konstruktoren beschreibt, werden bei der type-Deklaration bereits bekannte Datentypen mittels der *Standardkonstruktoren*

$D*D'$	(Tupelbildung)
$D \rightarrow D'$	(Funktionsraumbildung)
$\{s_1:D_1, s_2:D_2, ..., s_n:D_n\}$	(Record-Bildung)

zu neuen Datentypen verknüpft.