3.3.4 Polymorphe Datentypen

Wie bereits in den vorigen Abschnitten mehrfach sporadisch erwähnt, erlaubt ML die Definition polymorpher Funktionen und Datentypen. Polymorphe Funktionen sind schon hinlänglich benutzt und erläutert worden. Hier wollen wir uns mit polymorphen Typen befassen. Diese Typen definiert man über Typausdrücke, die mithilfe von

- Typvariablen (Hochkomma gefolgt von einem Bezeichner, z.B. 'a,'b, 'alphabet)
- Typkonstanten (int, real, ...)
- Typkonstruktoren (*, $\rightarrow$)
- Typfunktionen (list (s. Abschnitt 3.3.6), selbstdefinierte Typen)
- Klammern, um Abweichungen von der Prioritätenregelung der vorstehenden Konstruktoren und Funktionen festzulegen, die wie folgt gegeben ist:

Typfunktionen	höchste Priorität
*	mittlere Priorität
$\rightarrow$	niedrigste Priorität

Ein wichtiger Unterschied besteht in der Anwendung der Typfunktionen: Der Funktionsname steht hinter seinem Parameter, also z.B. (int*int) list statt list(int*int) für einen Typ Liste mit Elementen aus Paaren des Typs int. Hier ist also int*int der aktuelle Parameter der Typfunktion list.

Beispiele:

1) Den polymorphen Typ aller Paare $\Delta\times\Delta$ über einem beliebigen Datentyp Δ definiert man durch

```
type 'param paare='param*'param.
```

   Hier wird eine Typfunktion paare definiert mit dem Typparameter 'param und Wert 'param*'param. Konkretisierungen dieses Typs sind z.B.

```
type ip=int paare;
type bp=bool paare;
type ipaarvonpaar=(int paare) paare.
```

2) Der Typ

```
type ('a,'b,'c) T='a*'a list -> 'b*'c
```

   beschreibt anschaulich den Typ aller Abbildungen f: $\Delta\times\Delta^*\rightarrow\Delta'\times\Delta$". T ist also eine Typfunktion mit drei Parametern 'a, 'b, 'c und Ergebnis 'a*'a list ->'b*'c. Hierbei ist die Prioritätenregelung zu beachten, nach der

```
'a*'a list ->'b*'c = ('a*('a list)) -> ('b*'c)
```

   gilt. Durch

```
type neu=(int,bool,real) T
```

   konkretisieren wir den polymorphen Typ T zum Typ

```
int*int list -> bool*real
```

3.3.5 Generalisation

Die Generalisation (auch *Summenbildung*) vereinigt disjunkte Datentypen $D_1,...,D_n$ zu einem neuen Datentyp D. In ML schreibt man schematisch

$\underline{\text{datatype}}\ D=c_1\ \text{of}\ D_1 \mid c_2\ \text{of}\ D_2 \mid ... \mid c_n\ \text{of}\ D_n$

oder in parametrisierter Form

$\underline{\text{datatype}}\ (D_1,...,D_n)\ D=c_1\ \text{of}\ D_1 \mid c_2\ \text{of}\ D_2 \mid ... \mid c_n\ \text{of}\ D_n.$

Die durch "|" getrennten Teile heißen *Varianten*. c_i, $i=1,...,n$, sind hier frei gewählte Bezeichner, die Konstruktoren von D. Sie repräsentieren Abbildungen

$c_i: D_i \rightarrow D.$

Man kann sie als Marken betrachten, um die Varianten voneinander zu unterscheiden. Sie besitzen die Funktion der Typdiskriminatoren in PASCAL.

Beispiel: Wir definieren eine Datenstruktur für eine Personaldatei:

```
type grunddaten={vorname: string, gehalt: real};
datatype mitarbeiter=mann of grunddaten*{bart: bool} |
                         frau of grunddaten*{gebname: string};
> datatype mitarbeiter = mann of grunddten*{bart:bool} |
    frau of grunddaten*{gebname:string}
    con mann = fn : (grunddaten*{bart:bool}) -> mitarbeiter
    con frau = fn : (grunddaten*{gebname:string}) -> mitarbeiter
```

Hier wird der Datentyp `mitarbeiter` als disjunkte Vereinigung des Produkts zweier Records definiert. Die beiden Varianten werden über die Konstruktoren `mann` bzw. `frau` angesprochen:

```
val HerrMeier=mann({vorname="Paul", gehalt=3500.0},{bart=true});
> val HerrMeier = mann({gehalt=3500.0,vorname="Paul"},{bart=true}) : mitarbeiter
val FrauMeier=frau({gehalt=2500.0,vorname="Else"},{gebname="Kunze"});
> val FrauMeier = frau({gehalt=2500.0,vorname="Else"},{gebname="Kunze"}) :
                mitarbeiter
```

Die Generalisation von Datentypen findet man vor allem bei rekursiven Datentypen. Hier beschreiben dann einige Varianten die rekursive, andere die terminale Struktur des Datentyps.