3.4 Abstrakte Datentypen

In ML stehen zwei verschiedenen Konzepte zur Definition abstrakter Datentypen zur Verfügung:

- ein eingeschränktes Konzept geeignet für kleinere Anwendungen unter dem Schlüsselwort abstype: Hierbei handelt es sich um die Zusammenfassung von Daten zusammen mit darauf definierten Operationen zu einer Einheit. Jedoch kann die Spezifikation der Operationen nicht von ihrer Implementierung getrennt werden. In der üblichen Sprechweise handelt es sich also um konkrete Datentypen.
- ein leistungsfähiges Konzept, sog. *Module* oder *Strukturen*, für die Konstruktion und Spezifikation größerer Anwendungen unter dem Schlüsselwort structure. Auch hier handelt es sich um die Zusammenfassung von Daten und Operationen zu einer Einheit, jedoch kann man bei den Strukturen den Spezifikationsteil mit dem Schlüsselwort signature vom Implementierungsteil trennen. Ferner kann man Module mithilfe von Funktoren (Schlüsselwort: functor) sehr elegant parametrisieren. Dieses Konzept beruht auf einem Vorschlag von D. MacQueen: "Modules for Standard ML", Proc. of the ACM Symposium on LISP and Functional Programming 1984.

Darüberhinaus gehende spezifikatorische Elemente wie z.B. Gesetze sind in beiden Konzepten nicht vorhanden.

3.4.1 Konkrete Datentypen: abstype

Allgemein definiert man einen konkreten Datentyp in ML durch

```
abstype <Typbezeichner> = <Repräsentationstyp>
with <Deklaration der Operationen>
end.
```

Hierbei ist <Typbezeichner> wie üblich eine Typkonstante, Typfunktion usw. Der *Repräsentationstyp* beschreibt den konkreten Typ, durch den der abstrakte Typ realisiert werden soll. Die Deklaration der Operationen erfolgt in der üblichen Form durch fun oder val.

Beispiel: Wir definieren einen konkreten Datentyp queue auf der Basis von linearen Listen (=Repräsentationstyp) mit den Operationen empty, enter, remove, first und is_empty:

```
abstype 'type queue = q of 'type list
with
    val empty = q [ ];
    fun   enter x (q [ ]) = q [x] |
          enter x (q(_::b)) = enter x (q b);
    fun   remove (q(_::b)) = q b;
    fun   first (q(x::_)) = x;
    fun   is_empty (q [ ]) = true |
```

```
        is_empty _ = false
    end.
```

Definiert wird hier ein polymorpher Typ 'type queue. Der Repräsentationstyp, über dem dieser Typ realisiert wurde, ist eine lineare Liste q of 'type list. Zugriffe auf Objekte vom Typ queue sind nur über die Operationen möglich. Die Repräsentation ist gegenüber der Außenwelt verborgen. So ist z.B. ein Ausdruck der Form

```
q [1,2,3]
```

verboten. Vielmehr kann eine Queue dieser Form nur erzeugt werden durch den Ausdruck

```
enter 3 (enter 2 (enter 1 empty))
> - : int queue
```

Der Strich "-", den das System hier ausgibt, deutet an, daß die Repräsentation geheim ist.

Die obige Implementierung ist nicht sonderlich effizient, da die enter-Operation bei einer Queue mit n Einträgen O(n) Operationen benötigt. Alternativ implementiert man eine Queue durch ein Paar von Listen entry und exit. exit verknüpft mit der gespiegelten Version von entry sei die gesamte Queue. Man fügt neue Elemente immer vorne in entry ein und löscht sie immer vorne in exit. Ist exit leer, so kopiert man die gespiegelte Version von entry nach exit.

Aufgabe: Man definiere einen konkreten Datentyp queue auf der Basis des obigen Vorschlags und vergleiche die Effizienz diese Implementierung in obigem Beispiel.

Lösung:

```
abstype 'type queue = q of 'type list*'type list
with
    val  empty = q([ ],[ ]);
    fun  enter x (q(entry,exit)) = q(x::entry,exit);
    fun  remove (q(entry,_::exit)) = q(entry,exit) |
         remove (q(entry,[ ])) = remove (q([ ],rev entry));
    fun  first (q(_,x::_)) = x |
         first (q(entry,[ ])) = first (q([ ],rev entry));
    fun  is_empty (q([ ],[ ])) = true |
         is_empty _ = false
end.
```

Hier ist die Funktion rev die vordefinierte Funktion zum Spiegeln einer Liste.
Effizienzvergleich: selbst erledigen.

Beispiel: Konkreter Datentyp set implementiert durch eine lineare Liste:

```
abstype 'type set = s of 'type list
with
    val  empty = s [ ];
```

```
local
    fun  member (x::_) x = true |
         member (_::y) x = member y x;
    fun  filter p [ ] = [ ] |
         filter p (x::y) = if p x then x::(filter p y) else filter p y;
    fun  compose f g x = f(g(x));
in
    fun  insert x (s y) = if member y x then s y else s(x::y);
    fun  is_elem x (s y) = member y x;
    fun  union (s x) (s y) = s(x@(filter (compose not (member x)) y));
    fun  intersect (s x) (s y) = s(filter (member y) x)
end
end.
```

In dieser Definition haben wir erstmals ein Sprachelement verwendet, mit dem man lokale Objekte (Funktionen, Werte) definieren kann (sog. *private* Objekte). In der allgemeinen Form

```
local
    <Definitionen 1>
in
    <Definitionen 2>
end
```

können die in Definitionsteil 1 deklarierten Objekte in Definitionsteil 2 beliebig verwendet werden. Außerhalb dieses Bereichs sind sie nicht sichtbar.

Beispiel: Unendliche Mengen.

Auf den ersten Blick scheint es unmöglich, in einer Programmiersprache unendliche Mengen zu definieren. Tatsächlich ist die Definition aber denkbar einfach: ein weiterer Beweis für die Leistungsfähigkeit der funktionalen Programmierung.

Die zentrale Idee besteht darin, von einer *unendlichen* (entscheidbaren!) Menge $M \subseteq X$ (X Universum) zur *endlichen* Beschreibung der Menge mithilfe der charakteristischen Funktion

$\chi: X \rightarrow bool$ mit

$$\chi(x) = \begin{cases} true, & \text{falls } x \in M, \\ false, & \text{sonst} \end{cases}$$

überzugehen. Dann repräsentiert z.B. die konstante Funktion false die leere Menge, die konstante Funktion true die Menge X, die Funktion even die Menge aller geraden Zahlen, falls X=int ist. Der Elementtest beschränkt sich dann auf den Aufruf der charakteristischen Funktion, Vereinigung bzw. Schnitt zweier Mengen wird realisiert durch Disjunktion bzw.

Konjunktion der beiden charakteristischen Funktionen. Eine weitere Funktion makeset vereinfacht das erstmalige Erzeugen einer unendlichen Menge: Sie wandelt eine vorgegebene charakteristische Funktion in die zugehörige Repräsentation um. Der Datentyp:

```
abstype 'type set = s of 'type -> bool
with
    val   empty = s (fn x => false);
    fun   member x (s f) = f x;
    fun   insert x (s f) = s(fn y => x=y orelse f x);
    fun   union (s f) (s g) = s(fn x => f x orelse g x);
    fun   intersect (s f) (s g) = s(fn x => f x andalso g x);
    local
        fun   compose f g x = f(g(x));
    in
        fun   complement (s f) = s(compose not f);
    end;
    fun   makeset p = s p;
end.
```

Sei prime ein bereits definiertes Prädikat, das zu einer Zahl ausgibt, ob es sich um eine Primzahl handelt oder nicht. Dann erzeugt man mit

```
val primes = makeset prime;
> val primes: int set
```

die unendliche Menge aller Primzahlen. Weitere Beispiele:

```
val evennumbers = makeset (fn x=>(x mod 2=0));
> val evennumbers: int set
val pe = union primes evennumbers;
> val pe: int set
member 15 pe;
> false: bool
```

3.4.2 Module: structure, signature, functor

Module sind in ML meist zweigeteilt: Sie bestehen aus einer Signatur (Schlüsselwort: signature) und einem davon getrennten Implementierungsteil, in dem die durch die Signatur gegebenen Objekte, Typen und Operationen ausprogrammiert werden. Dieser Teil wird durch das Schlüsselwort structure eingeleitet und besitzt einen syntaktischen Aufbau ähnlich konkreten Datentypen à la abstype.

Strukturen.

Obwohl Strukturen in der Regel mit Signaturen verbunden sind, behandeln wir sie zunächst getrennt voneinander. Eine Struktur definiert man in ML allgemein durch

```
structure <Strukturbezeichner> = struct
    <beliebige Definitionen von Typen, Werten und Operationen
    gemäß bekannter ML-Syntax>
end.
```

Beispiel: Der im 1. Beispiel von 3.4.1 definierte Typ queue lautet als Struktur:

```
structure queue = struct
    datatype 'a T = q of 'a list;
    val   empty = q [ ];
    fun   enter x (q [ ]) = q [x] |
          enter x (q(_::b)) = enter x (q b);
    fun   remove (q(_::b)) = q b;
    fun   first (q(x::_)) = x;
    fun   is_empty (q [ ]) = true |
          is_empty _ = false
end.
```

Der Typ einer Queue ist 'a T.

Wie greift man auf die Definitionen einer Struktur zu? Die Bezeichner, die in einer Struktur deklariert worden sind, werden eindeutig identifiziert durch eine dot-Notation (wie bei Records in PASCAL) der Form

<Strukturbezeichner>**.**<Bezeichner>.

Hier unterscheidet sich der Zugriff auf Objekte von Strukturen von dem Zugriff auf Objekte von konkreten abstype-Datentypen. Dies schließt Namenskonflikte aus, wenn man mehrere Strukturen deklariert hat, in denen jeweils Operationen mit gleichen Bezeichnern und Funktionalitäten definiert sind.

Beispiel: Einfügen der Zahlen 1, 2, 3 in eine Queue mit Objekten vom Typ int:

```
queue.enter 3 (queue.enter 2 (queue.enter 1 queue.empty)).
```

Diese manchmal recht umständliche Schreibweise kann man abkürzen, indem man eine Struktur vorher *öffnet*. Anschließend kann man auf die Objekte über ihre Bezeichner ohne dot-Notation zugreifen (wie bei with in PASCAL).

Beispiel (Fortsetzung von oben):

```
open queue;
```

```
enter 3 (enter 2 (enter 1 empty)).
```

Da open-Klauseln nicht durch ein "close" rückgängig gemacht werden können, empfiehlt es sich immer, das Öffnen einer Struktur mittels des local-Konstrukts (s. 3.4.1) auf einen eng begrenzten Bereich zu beschränken:

```
local open queue
in
    enter 3 (enter 2 (enter 1 empty))
end.
```

Anderenfalls kann das Öffnen mehrerer Strukturen mit gleichen internen Bezeichnern zu unübersichtlichen Bindungen führen.

Signaturen.

Strukturen in der obigen Form stimmen im wesentlichen mit abstypes's überein und können damit zu den konkreten Datentypen gerechnet werden. Eine gewisse Nähe zu abstrakten Datentypen erreicht man erst, wenn man Strukturen unter Verwendung von Signaturen und Funktoren (s.u.) um Möglichkeiten zur Datenabstraktion und zum information hiding ergänzt. Mittels Signaturen kann man also die wesentlichen spezifikatorischen Merkmale (die *Exportschnittstelle*) einer Struktur, das sind die Typen sowie die Bezeichner der Operationen und ihre Funktionalität, von der Struktur lösen und getrennt halten.
Eine Signatur definiert man in ML nach folgendem Muster:

```
signature <Signaturbezeichner> = sig
    <Folge von Typdeklarationen und Deklarationen von
    Bezeichnern mit ihrer Funktionalität>
end.
```

Folgende Deklarationen sind innerhalb von Signaturen zugelassen:

- Wert- und Funktionsdeklarationen mittels val zusammen mit ihrem Typ in der Form
  ```
  val <Bezeichner> : <Typ>
  ```
- Typdefinitionen in der Form
  ```
  type <ggf. Liste von Typvariablen> <Typbezeichner>
  ```
- datatype-Definitionen.

Beispiele:

1) Die oben definierte Struktur queue sowie die in der letzten Aufgabe gesuchte besitzen folgende Signatur:

```
signature queue=sig
    type        'a T;
    val   empty: 'a T;
    val   enter: 'a*'a T -> 'a T;
```

```
        val   remove: 'a T -> 'a T;
        val   first: 'a T -> 'a;
        val   is_empty: 'a T -> bool
    end.
```

Jede andere Struktur, in der
- ein polymorpher Typ 'a T,
- eine Konstante empty vom Typ 'a T,
- eine Funktion enter vom Typ 'a×'a T→'a T,
- eine Funktion remove vom Typ 'a T→'a T,
- eine Funktion first vom Typ 'a T→'a,
- eine Funktion is_empty vom Typ 'a T→bool

deklariert ist, besitzt ebenfalls diese Signatur.

2) Es folgt eine Signatur für beliebige Objekte mit zwei Operationen und einem Initialwert:

```
signature any_object=sig
    type        object;
    val   init: object;
    val   grow: object -> object;
    val   shrink: object -> object
end.
```

Dieser Signatur ordnet sich jede Struktur unter, in der ein Typ object, eine Konstante vom Typ object sowie zwei Funktionen grow und shrink der Funktionalität object→object definiert sind.

Die Klauseln type 'a T und type object sind in den obigen Beispielen erforderlich, denn anderenfalls würden sich die nachfolgenden Definitionen auf Typen 'a T und object beziehen, die bereits vorher irgendwo im Programm deklariert worden sind.

Sobald man eine Signatur angegeben hat, kann man Strukturen definieren, die sich dieser Signatur unterordnen. Solch eine Definition erfolgt durch

```
structure <Strukturbezeichner> : <Signaturbezeichner> = struct
    <wie bisher: Folge von Typdeklarationen und Deklarationen von
    Bezeichnern mit ihrer Funktionalität>
end.
```

Das ML-System überprüft automatisch, ob die definierte Struktur die angegebene Signatur besitzt oder nicht.

Beispiele:

1) Einfache Definition der natürlichen Zahlen unter Verwendung der Spezifikation von any_object:

```
structure nat: any_object=struct
    type    object=int;
    val     init=0;
    fun     grow n=n+1;
    fun     shrink 0=0 |
            shrink n=n-1
end.
```

2) Verbesserte Definition der natürlichen Zahlen unter Verwendung der Spezifikation von any_object:

```
structure nat: any_object=struct
    datatype object=null | succ of object;
    val     init=null;
    val     grow=succ;
    fun     shrink null=null |
            shrink (succ x)=x
end.
```

Hier liegt eine Besonderheit vor: Die Signatur von any_object fordert einen type object, in der Struktur wird jedoch ein datatype object definiert. Dennoch erfüllt die Struktur die Spezifikation (was auch durchaus vernünftig erscheint). Umgekehrt: Wäre in der Signatur object als datatype spezifiziert, würde eine Struktur die Signatur *nicht* erfüllen, wenn in ihr object durch type definiert ist. Auch dies erscheint plausibel, denn eine Signatur beschreibt ja nur eine Mindestanforderung, die bei der Implementierung vom Programmierer nach Belieben verschärft werden kann. Genauere Aussagen, wann eine Struktur eine Spezifikation erfüllt, folgen unten.

3) Definition des freien Monoid über dem einelementigen Alphabet {a} unter Verwendung einer Liste:

```
structure monoid: any_object=struct
    datatype symbol=a;
    type    object=symbol list;
    val     init=[];
    fun     grow l=a::l;
    fun     shrink []=[] |
            shrink (a::l)=l;
    fun     concat x y= x@y: object
end.
```

Die Funktion concat ist zwar in der Signatur von object nicht gefordert, dennoch darf man sie definieren, ohne die Spezifikation zu verletzen. concat wie auch der Typ symbol werden jedoch nach außen nicht sichtbar (*private* Objekte, s.u.).

4) Die folgende Definition wird von ML zurückgewiesen, denn die Struktur erfüllt nicht die durch any_object gegebene Spezifikation, weil die Funktion grow nicht die spezifizierte Funktionalität besitzt und shrink nicht definiert ist:

```
structure mistake: any_object=struct
    type object=string;
    val  init="";
    fun  grow x y=x^y
end.
```

Signaturen und Strukturen.

Im folgenden wollen wir genauer festhalten, unter welchen Bedingungen eine Struktur einer vorgegebene Spezifikation, dargestellt durch eine Signatur, erfüllt. Es gelten drei Regeln, von denen die ersten beiden bestimmen, wann die Deklaration einer Struktur zur Signatur paßt. Die dritte Regel schließlich ermöglicht die Definition *privater* Objekte.

Regel 1: Matching von Bezeichnern.

Zu jedem Bezeichner, der in einer Signatur definiert wird, muß es eine zugehörige Definition in der Struktur geben.

Beispiel: Im obigen Beispiel 4 wurde vergessen, den durch die Signatur object gegebenen Bezeichner shrink in der Struktur mistake auszuprogrammieren.

Regel 2: Matching von Typen.

Wenn ein Bezeichner in einer Struktur deklariert wird, zu dem es eine entsprechende Definition in der zugehörigen Signatur gibt, dann muß der Typ des Bezeichners in der Struktur zum Typ des Bezeichners in der Signatur passen. Hierbei gilt:

a) Jede Typdefinition für den Typ T in der Struktur paßt zur Spezifikation

```
type T
```

in der Signatur.

b) Nur eine identische datatype-Definition für den Typ T in der Struktur paßt zur Spezifikation

```
datatype T = ...
```

in der Signatur.

Durch Wahl von type in der Signatur kann man also dem Programmierer noch gewisse Freiheiten lassen, wie er einen Typ implementiert.

Beispiel: Im obigen Beispiel 2 konnte die Definition type object in der Signatur wie erwähnt durch eine datatype-Deklaration ausprogrammiert werden.

In Beispiel 4 stimmt der Typ von grow (object→(object→object)) in der Struktur nicht mit dem Typ object→object in der Signatur überein.

Regel 3: Private Objekte.

Jede Deklaration in einer Struktur, zu der es keine entsprechende Definition in der zugehörigen Signatur gibt, ist bezgl. der Struktur privat, also außerhalb nicht sichtbar. Auf private Objekte kann man von außen weder über die dot-Notation noch über ein Öffnen der Struktur zugreifen.

Beispiel: In Beispiel 3 oben sind concat und symbol privat.

Das folgende Beispiel zeigt noch einmal die wesentlichen Elemente von Signaturen und Strukturen in der Gesamtschau.

Beispiel: Wir definieren zunächst eine Signatur für allgemeine Ordnungsrelationen und anschließend mehrere Ordnungen auf konkreten Datentypen:

```
signature order=sig
    type T;
    val lesseq: T*T->bool
end;
structure intorder: order=struct
    type T=int;
    fun  lesseq(x:T,y:T)=x≤y
end;
structure natorder: order=struct
    datatype T=null | succ of T;
    fun  lesseq(null,_)=true |
         lesseq(_,null)=false |
         lesseq(succ x,succ y)=lesseq(x,y)
end;
structure natpairorder: order=struct
    type T=natorder.T*natorder.T;
    fun  lesseq((x,y),(x',y'))=natorder.lesseq(x,x')
                          orelse (x=x' andalso natorder.lesseq(y,y'))
end.
```

Funktoren.

Zur Motivation betrachte man eine Funktion zur Ermittlung des Minimums einer Liste:

```
fun  min [x]=x: real |
     min (x::y::z)=if x≤y then min(x::z) else min (y::z).
```

Hier ist eine Typeinschränkung (wahlweise int, real oder string) erforderlich, weil die Operation ≤ nur auf diesen Typen zugelassen ist. Dies bedeutet für den Programmierer häufig eine unerwünschte Einschränkung, denn eine Minimumsfunktion ist nicht nur auf int, real, string eine sinnvolle Funktion, sondern allgemein auf beliebigen Typen, deren Wertemenge geordnet ist. Dennoch ist für jeden anderen Typ nach unserem bisherigen Kenntnisstand eine eigene Minimumsfunktion zu schreiben.
Einen Ausweg bilden Funktoren. Mittels Funktoren kann man Strukturen in einer ähnlichen Weise parametrisieren wie Funktionen. Sie bilden Strukturen (=aktuelle Parameter des Funktors) in eine neue Struktur (=Funktorergebnis) ab. Anschaulich besteht zwischen Funktoren und Funktionen folgende einprägsame Analogie:

Funktor	↔	Funktion
Struktur	↔	Wert
Signatur	↔	Typ.

Zur Lösung des obigen Minimumsproblems definiert man zunächst eine Struktur für die zugrundeliegende Ordnung und übergibt sie dem Funktor als aktuellen Parameter, der daraufhin als Ergebnis eine Struktur liefert, in der die Minimumsbildung an die gewünschte Ordnung angepaßt wurde (s. Beispiel unten).

Einen Funktor definiert man allgemein in einer Funktions-artigen Schreibweise durch

```
functor <Funktorbezeichner> (structure <Bezeichner 1>: <Signatur 1>;...;
                             structure <Bezeichner n>: <Signatur n>): <Signatur>=struct
    <Definitionen wie bei Strukturen>
end.
```

Die Strukturen mit Bezeichner 1 bis n spezifiziert durch die Signaturen 1 bis n sind die formalen Parameter. <Signatur> ist die Signatur der Struktur, die der Funktor als Ergebnis liefert.

Beispiel: Wir lösen das o.g. Minimum-Problem mit einem Funktor. Der Funktor erwartet als aktuellen Parameter eine Struktur, die eine Ordnung spezifiziert, z.B. eine der oben angegebenen Ordnungen intorder, natorder usw. oder irgendeine andere, die der Signatur order genügt. Als Resultat liefert der Funktor eine Struktur einer zunächst noch näher zu definierenden Signatur, die die Eigenschaften der Minimumssuche beschreibt. Der Funktor-Rumpf selbst legt fest, wie die Resultatsstruktur unter Verwendung der Deklarationen der Parameterstruktur ausprogrammiert wird. Zunächst die Signatur des Funktorergebnisses:

```
signature minimum=sig
    type T;
    val min: T list -> T
```

```
end.
```

Nun der Funktor:

```
functor make_min(structure act_order: order): minimum=struct
    type T=act_order.T;
    fun  min [x]=x |
         min (x::y::z)=if act_order.lesseq(x,y) then min(x::z) else min (y::z)
end.
```

Der Funktor besitzt den formalen Parameter act_order, der eine beliebige Struktur der Signatur order bezeichnet. Die Ergebnisstruktur erfüllt die Signatur minimum.
Im Rumpf werden die in minimum definierten Objekte implementiert: Als Typ T wird der Typ act_order.T aus der zugrundeliegenden Ordnung verwendet. Die Funktion min besitzt die bekannte Form, jedoch wird zum Vergleich die zugehörige Ordnungsrelation act_order.lesseq anstelle von ≤ herangezogen.
Zur Abkürzung kann man die Signatur minimum auch unmittelbar in die Funktordefinition einfügen, also

```
functor ... : sig
                type T;
                val min: T list -> T
              end=
struct ....
```

Funktoren ruft man in natürlicher Weise auf:

```
<Funktorbezeichner> (structure <Bezeichner 1>;...; structure <Bezeichner n>).
```

Das Ergebnis ist eine Struktur, die man an einen neuen Bezeichner binden kann.

Beispiel: Definition einer Struktur zur Minimumsbildung auf natpairorder (s.o.):

```
structure natpairmin=make_min(structure natpairorder: order).
```

Benutzung der Struktur z.B. durch Aufruf der Funktion min:

```
natpairmin.min [(null,succ null),(succ null,succ(succ null)),...].
```

Abschließend noch ein Beispiel zur mathematischen Nutzung des Funktorkonzepts.

Beispiel: Wir definieren einen Funktor, der zu einem Ring R=(0,+,-,*) den Ring der 2×2-Matrizen erzeugt:

```
signature ring=sig
    type R;
    val null: R;
    val plus: R*R->R;
```

```
    val minus: R*R->R;
    val mult: R*R->R;
end;
functor make_matrix(structure act_ring: ring): ring=struct
    type      R=(act_ring.R*act_ring.R)*(act_ring.R*act_ring.R);
    val   null=((act_ring.null, act_ring.null),(act_ring.null,act_ring.null));
    fun   plus(((a,b),(c,d)),((a',b'),(c',d')))=
                                ((act_ring.plus(a,a'),  act_ring.plus(b,b')),
                                (act_ring.plus(c,c'),act_ring.plus(d,d')));
    fun   minus ...
    fun   mult ...
end.
```