2 Abstrakte Datentypen

2.1 Konkrete und abstrakte Datentypen

Unter einem *Datentyp* oder kurz *Typ* (im dt. auch häufig *Rechenstruktur*) versteht man die Zusammenfassung von Wertebereichen und Operationen zu einer Einheit. Der Zweck von Datentypen besteht in der Zerlegung des Universums der in einer Programmiersprache vorkommenden Werte in Klassen mit gemeinsamen Merkmalen bezgl. Darstellung und erlaubten Operationen.

Bisher haben wir Datentypen nahezu ausschließlich folgendermaßen definiert:

- Wir haben die Elemente der Wertemenge des Datentyps explizit aufgelistet;
- wir haben die Wirkungsweise der Operationen auf die so vorgegebenen Elemente des Datentyps genau beschrieben.

Beispiel: Den Datentyp nat haben wir als Paar (W,R) in der Standarddarstellung definiert mit

W={0,1,2,3,...} und

R={pred,succ,+,-,...}

mit

$$\text{pred}(i) = \begin{cases} i-1, & i \geq 1 \\ \perp, & \text{sonst;} \end{cases}$$

succ(i)= ...

usw.

Durch diese Art der Definition haben wir uns allerdings einer Reihe von Freiheiten beraubt. Wir sind nun gezwungen, natürliche Zahlen nur in Form ihrer Dezimaldarstellung und Operationen nur in Form der gängigen arithmetischen Operationen zu akzeptieren. Nun wissen wir aber, daß es eine Reihe weiterer Repräsentationen für natürliche Zahlen gibt, die genauso verwendet werden können, für einige Anwendungen sogar zweckmäßiger, für andere aber auch weniger sinnvoll sein können. Es erscheint nicht praktisch, uns der Möglichkeit zu entheben, mal diese oder mal jene Darstellung zu wählen oder es einfach der Implementierung zu überlassen, welche Repräsentation sie für zweckmäßig hält. Wichtig ist aber, daß alle Darstellungen mit ihren zugehörigen Operationen die wesentlichen Eigenschaften natürlicher Zahlen erfüllen, dazu gehören z.B. die Rechengesetze.

Beispiele für andere Repräsentationen von nat:

1) „Bierdeckelnotation“: Offenbar hat sich die Darstellung natürlicher Zahlen durch Striche

   ε, |, ||, |||, ||||, ...

   0 1 2 3 4 ...

   in der „freien Wirtschaft“ als geeigneter erwiesen als die Standarddarstellung. Sie ist

aber bei entsprechender Definition der Operationen

$$\text{pred}(|^n) = \begin{cases} |^{n-1}, & n \geq 1 \\ \perp, & n = 0. \end{cases}$$

usw.

gleichwertig („isomorph“) zur Standarddarstellung.

2) Darstellung mit den Peano-Axiomen: Sei M eine Menge, 0 ein Objekt und f: $M \to M$ eine Abbildung. M heißt Menge der natürlichen Zahlen, wenn gilt:
   (P1) $0 \in M$
   (P2) $x \in M \Rightarrow f(x) \in M$
   (P3) $x \in M \Rightarrow f(x) \neq 0$
   (P4) $x,y \in M, x \neq y \Rightarrow f(x) \neq f(y)$
   (P5) $(0 \in A \wedge (\copyright x \in A \Rightarrow f(x) \in A)) \Rightarrow M \subseteq A$.
   Hier sind die Elemente von IN also repräsentiert durch
   0, f(0), f(f(0)), f(f(f(0))), ...
   0 1 2 3 ...
   f ist die Nachfolgerfunktion. Entsprechend dieser Darstellung – ebenfalls isomorph zur Standarddarstellung – sind die übrigen Operationen zu realisieren.

Immer wenn man sich bei der Definition eines Datentyps zugleich für eine konkrete Repräsentation der Datenobjekte entscheidet, spricht man von einem *konkreten Datentyp*. Beschränkt man sich hingegen nur auf die Eigenschaften, die die Operationen und Wertebereiche besitzen, so handelt es sich um einen *abstrakten Datentyp*. In abstrakten Datentypen kommt es nicht auf die tatsächliche Darstellung der Daten an, entscheidend sind allein die Operationen und ihr Verhalten.
Beispiel: Beim Datentyp nat ist die konkrete Darstellung der natürlichen Zahlen bedeutungslos. Wichtig ist nur, daß auf den Darstellungen die Rechengesetze in exakt der Form gelten wie bei der Standarddarstellung.

Verfügt man also über ein programmiersprachliches Konzept für abstrakte Datentypen, so spezifiziert man bei der Softwareentwicklung zunächst den abstrakten Datentyp, implementiert diesen dann anschließend in Übereinstimmung mit der Spezifikation möglichst effizient.

2.2 Abstrakte Datentypen

Abstrakte Datentypen definiert man mithilfe eines formalen Schemas, der sog. *Signatur*, bei der man sich noch nicht auf konkrete Mengen und Operationen festlegt, sondern zunächst nur die Mengensymbole (*Sorten*) sowie Operationssymbole und deren Funktionalität angibt.

Definition A:
Eine **Signatur** Σ ist ein Paar $\Sigma = (S,F)$ mit:
- S ist eine Menge, die Menge der **Sorten**,
- F ist eine Menge von Mengen F(w,s): $F \subseteq \{F(w,s) \mid w \in S^*, s \in S\}$. Die Elemente $f^{w,s} \in F(w,s)$ heißen **Operator**- oder **Funktionssymbole** der Funktionalität $w \rightarrow s$, w ist die **Stelligkeit**, s die **Zielsorte** von $f^{w,s}$.

Ein Funktionssymbol $f^{w,s}$, $w=s_1...s_n \in S^*$, $s \in S$, bezeichnet hier also eine Funktion

$$f^{w,s}: s_1 \times ... \times s_n \rightarrow s.$$

Ist $w=\varepsilon$, so ist $f^{\varepsilon,s}$ eine nullstellige Funktion, also eine *Konstante* der Sorte s.

Man beachte, daß die obige Definition nur ein Schema für einen Datentyp vorgibt; die Symbole besitzen noch keine Bedeutung. Erst durch die Zuordnung der Symbole zu konkreten Mengen und Funktionen (sog. *Interpretation*) erhält die Signatur eine Bedeutung.

Meist stellt man eine Signatur übersichtlicher in Form einer Datentypdefinition dar, wie sie aus Programmiersprachen bekannt ist:

```
type Σ =
    sorts <Auflistung der Sorten>
    functions
        <Auflistung der Funktionen f^{w,s} ∈ F in der Form
        f^{w,s}: s_1×...×s_n→s für w=s_1...s_n>
end.
```

Beispiele:

1) Gegeben sei die Signatur $\Sigma=(S,F)$ mit

   $S=\{b\}$, $F=F(\varepsilon,b) \cup F(b,b) \cup F(bb,b)$,

   wobei $F(\varepsilon,b)=\{t,f\}$,
   $F(b,b)=\{n\}$,
   $F(bb,b)=\{u,o\}$.

   Anschaulich

```
type Σ =
    sorts b
    functions
        t: →b
        f: →b
        n: b→b
        u: b×b→b
        o: b×b→b
end.
```

2) Sei die Signatur $\Sigma'=(S',F')$ mit

$S'=\{b,nz\}$, $F'=F'(\varepsilon,nz)\cup F'(nz,nz)\cup F'(nznz,nz)\cup F'(nznz,b)$,

wobei $F'(\varepsilon,nz)=\{0\}$,

$F'(nz,nz)=\{s\}$,

$F'(nznz,nz)=\{p,m\}$,

$F'(nznz,b)=\{kl,gr,gl\}$.

Anschaulich

type Σ' =
  sorts b,nz
  functions
    0: $\rightarrow$nz
    s: nz$\rightarrow$nz
    p: nz$\times$nz$\rightarrow$nz
    m: nz$\times$nz$\rightarrow$nz
    kl: nz$\times$nz$\rightarrow$b
    gr: nz$\times$nz$\rightarrow$b
    gl: nz$\times$nz$\rightarrow$b
end.

Wie erhält man aus einer Signatur einen konkreten Datentyp?
Bei den in der Signatur vorkommenden Sorten und Operationen handelt es sich nur um bedeutungslose Symbole für Mengen und Funktionen. Um einen konkreten Datentyp zu erhalten, muß man den Sorten konkrete Mengen und den Funktionssymbolen konkrete Funktionen zuordnen, wobei die Funktionalitäten zu beachten sind. Eine solche Zuordnung bezeichnet man als Interpretation. Man kann z.B. in der obigen Signatur Σ der Sorte b die Menge IB={wahr,falsch} und den Funktionssymbolen n, o, u die Funktionen not, and, or auf IB zuordnen. So erhält man den konkreten Datentyp bool.

Definition B:
Gegeben sei eine Struktur $A = (M_1,M_2, ...; g_1,g_2,...)$, wobei $M_1,M_2,...$ Mengen und $g_1,g_2,...$ darauf definierte Funktionen sind. A heißt **konkreter Datentyp**, **konkrete Rechenstruktur** oder konkrete heterogene **Algebra** („heterogen“, da die Mengen M_i im allgemeinen verschieden sind).

Definition C:
Seien $\Sigma = (S,F)$ eine Signatur und $A = (M_1,M_2, ...; g_1,g_2,...)$ ein konkreter Datentyp.
Eine Abbildung

$\varphi: \Sigma \rightarrow A$

heißt **Interpretation von** Σ **in** A, falls gilt:

i) $\varphi = (\varphi_S, \varphi_F)$ mit

φ_S: S $\rightarrow$ {M_1,M_2,...} total definiert

φ_F: F $\rightarrow$ {g_1,g_2,...} total definiert.

ii) φ_S und φ_F sind surjektiv.

iii) Für jedes f$\in$ F(s_1...s_r, s) gilt:

$\varphi_F(f)$: $\varphi_S(s_1) \times \varphi_S(s_2) \times ... \times \varphi_S(s_r) \rightarrow \varphi_S(s)$.

Beispiel: Wir interpretieren Σ in der Algebra A = (IB; immerwahr, immerfalsch, not, and, or) mit IB={true,false} und setzen:

$\varphi_S(b)$ = IB,

$\varphi_F(t)$ = immerwahr: $\rightarrow$IB mit immerwahr()=true,

$\varphi_F(f)$ = immerfalsch: $\rightarrow$IB mit immerfalsch()=false,

$\varphi_F(n)$ = not: IB$\rightarrow$IB,

$\varphi_F(u)$ = and: IB$\times$IB$\rightarrow$IB,

$\varphi_F(o)$ = or: IB$\times$IB$\rightarrow$IB.

Da Signaturen nur Schemata sind, kann man ihnen in der Regel viele verschiedene konkrete Datentypen zuordnen. Zu einer gegebenen Signatur existieren daher sehr viele Interpretationen.

Beispiel: Wir interpretieren Σ in A'=(IN_0; 0,1,+,-,succ) und setzen:

$\psi_S(b)$ = IN_0,

$\psi_F(t)$ = 1: $\rightarrow$ IN_0,

$\psi_F(f)$ = 0: $\rightarrow$ IN_0,

$\psi_F(n)$ = succ: $IN_0 \rightarrow IN_0$,

$\psi_F(u)$ = +: $IN_0 \times IN_0 \rightarrow IN_0$,

$\psi_F(o)$ = -: $IN_0 \times IN_0 \rightarrow IN_0$.

Auch wenn diese Interpretation nicht beabsichtigt sein sollte, treten hierbei keine Widersprüche auf, da bisher nicht festgelegt wurde, welche Eigenschaften die konkreten Mengen und Funktionen besitzen sollen.

Signaturen allein reichen also zur eindeutigen Beschreibung konkreter Datentypen nicht aus. Vielmehr muß man die Eigenschaften, die der später zu verwendende Datentyp besitzen soll, zur Signatur hinzufügen.

Beispiel: Gesetzt den Fall die Signatur Σ soll den Datentyp bool beschreiben, so erwarten wir z.B., daß die Menge $\varphi_S(b)$, als die wir die Sorte b interpretieren, genau zweielementig ist, oder daß die konkrete Funktion $\varphi_F(n)$, als die wir das Funktionssymbol n: b$\rightarrow$b aus Σ interpretieren, die Eigenschaften besitzt, die wir von der not-Funktion erwarten, z.B. die Eigenschaft

$\varphi_F(n)(\varphi_F(n)(x))=x$,

d.h. die not-Funktion zweimal hintereinander angewendet liefert wieder das ursprüngliche Objekt.
Für die obige Interpretation ψ mit $\psi_F(n)$=succ trifft dies jedoch nicht zu, da

succ(succ(0))=2≠0

gilt. Die Definition von Σ ist also so zu erweitern, daß diese unerwünschte Interpretationen ausscheidet..

Aus diesem Beispiel sieht man, daß man Zusatzbedingungen einführen muß, um Signaturen „trennscharf“ zu machen. Der Lösungsweg besteht darin, Gleichungen (*Gesetze*) einzuführen, um die Abhängigkeiten zwischen den Funktionssymbolen zu beschreiben. Gesetze sind Ausdrücke der Form

$t \equiv t'$,

wobei t und t’ *Terme* sind, die (zumindest) die gleiche Zielsorte besitzen müssen, und $\equiv$ besagt, daß die beiden Terme in einem konkreten Datentyp den gleichen Wert besitzen müssen. Die Interpretationen φ müssen also die Gesetze respektieren, d.h. aus $t \equiv t'$ muß stets $\varphi(t)=\varphi(t')$ folgen.

Was sind Terme? Terme sind anschaulich Ausdrücke, die man mit den Funktionssymbolen einer Signatur sowie mit Variablen bilden kann, z.B. in Σ:

n(f) oder
u(x,o(t,y)) oder
n(n(y)).

Deshalb legt man zuerst für eine gegebene Signatur $\Sigma=(S,F)$ eine Menge X von Variablen fest, wobei gilt

$$X = \bigcup_{s \in S} X_s \quad \text{und} \quad X_s \cap X_{s'} = \emptyset, \text{ falls } s \neq s'.$$

Jeder Sorte $s \in S$ wird also eine Menge X_s von Variablen zugeordnet; diese sind untereinander paarweise disjunkt. Die korrekten Ausdrücke über der Signatur Σ und den Variablen X, die auch als Terme bezeichnet werden, sind dann wie folgt definiert:

Definition D:
Sei $\Sigma=(S,F)$ eine Signatur. Jeder Sorte $s \in S$ sei eine Menge X_s von Variablen zugeordnet. Dann sei

$$X = \bigcup_{s \in S} X_s \quad \text{und} \quad X_s \cap X_{s'} = \emptyset, \text{ für } s \neq s'.$$

Dann ist die Menge $T_{\Sigma,X}$ der **Terme über Σ und** X zusammen mit einer Funktion Ziel:$T_{\Sigma,X} \rightarrow S$ die kleinste Menge mit den folgenden Eigenschaften:

i) Für alle $x \in X_s$ ist $x \in T_{\Sigma,X}$ mit Ziel(x)=s.
ii) Für alle $f \in F(\varepsilon,s)$ ist $f \in T_{\Sigma,X}$ mit Ziel(f)=s.

iii) Falls $f \in F(s_1 s_2 ... s_n, s)$ und $t_i \in T_{\Sigma,X}$ und $Ziel(t_i)=s_i$ für i=1,...,n und $n \geq 1$, dann ist auch $f(t_1,...,t_n) \in T_{\Sigma,X}$, und es gilt $Ziel(f(t_1,...,t_n))=s$.

Terme werden also wie die üblichen arithmetischen oder logischen Ausdrücke konstruiert: Ausgehend von Variablen und Konstanten (=nullstelligen Funktionssymbolen) als elementaren Termen kann man durch sukzessive Anwendung von Operatoren unter Beachtung der Stelligkeiten immer komplexere Terme erzeugen. Umgekehrt sind nur solche Terme korrekt, die sich durch die Anwendung der obigen Regeln in endlich vielen Schritten ergeben.
Mithilfe von Termen kann man Gesetze bilden, die das Verhalten der Operationen einer Signatur beschreiben.

Definition E:
Gegeben seien eine Signatur $\Sigma=(S,F)$ und eine Variablenmenge X. $b \in S$ sei die Sorte, die zu den Wahrheitswerten gehört, sofern vorhanden
Die Menge der **Gesetze** $G_{\Sigma,X}$ **über** Σ **und** X ist die kleinste Menge mit folgenden Eigenschaften:

i) Falls $t \in T_{\Sigma,X}$ und $Ziel(t)=b$, dann ist $t \in G_{\Sigma,X}$. (Dies entfällt, falls keine Zielsorte b für die Wahrheitswerte vorgesehen ist.)
ii) Falls $t_1, t_2 \in T_{\Sigma,X}$ und $Ziel(t_1)=Ziel(t_2)$, dann ist $(t_1 \equiv t_2) \in G_{\Sigma,X}$.
iii) Falls $g, g_1, g_2 \in G_{\Sigma,X}$, dann sind auch
not (g) , (g_1 and g_2), (g_1 or g_2), ($g_1 \Rightarrow g_2$) und ($g_1 \Leftrightarrow g_2$).
iv) Falls $g \in G_{\Sigma,X}$ und $x \in X$ ist, dann sind auch
$\exists x\ (g) \in G_{\Sigma,X}$ und $©x\ (g) \in G_{\Sigma,X}$.

Gesetze erhält man also, indem man Terme mit gleicher Zielsorte durch das Symbol „≡“ identifiziert und derartig gebildete Gleichungen zusammen mit logischen Ausdrücken (=Terme der Zielsorte b) mit den üblichen Operationen der Logik verknüpft. not, and, or , ⇒ und ⇔ entsprechen den logischen Operationen „nicht“, „und“, „oder“, „impliziert“ und „gleichwertig“, ©x(g) und ∃x(g) bezeichnen den Allquantor („für alle x gilt …“) und den Existenzquantor („es gibt ein x, so daß gilt ...“). Diese intendierte Bedeutung, ferner die *Gültigkeit* von Gesetzen, müßten wir nun eigentlich genauer definieren. Da diese Begriffe aus mathematischen Vorlesungen bekannt sein sollten, wollen wir die Definitionen hier vernachlässigen, um den definitorischen Apparat nicht zu sehr aufzublähen.

Beispiel: Über der Signatur Σ', die die natürlichen Zahlen beschreiben soll, kann man mit der Variablenmenge $X_{nz}=\{x,y\}$ z.B. folgende Gesetze formulieren:

©x (p(x,0) ≡ x),
©ξ (m(x,x) ≡ 0),
©x (∃ y (gr(y,x))).

Das erste dieser Gesetze soll aussagen, daß die Addition einer gegebenen natürlichen Zahl (ausgedrückt durch eine Variable x) mit der Zahl 0 stets die gegebene Zahl liefert. Das letzte dieser Gesetze soll umgangssprachlich besagen: Zu jeder natürlichen Zahl existiert eine größere natürliche Zahl. Diese Deutungen setzen voraus, daß wir die Signatur Σ' in geeigneter Weise in einem konkreten Datentyp nat interpretieren.

Eine Signatur zusammen mit Gesetzen nennt man einen abstrakten Datentyp.

Definition F:
Sei Σ =(S,F) eine Signatur, X eine Menge von Variablen und $E \subseteq G_{\Sigma,X}$ eine Menge von Gesetzen über Σ und X.

a) Dann heißt D=(Σ,E) **abstrakter Datentyp**.
b) Ein konkreter Datentyp A heißt **Modell** eines abstrakten Datentyps D=(Σ,E), wenn es eine Interpretation φ: $\Sigma \rightarrow$A gibt, so daß die Gesetze E unter φ simultan gültig sind.

Anschaulich stellt man einen abstrakten Datentyp dar, indem man die programmiersprachliche Darstellung der Signatur am Schluß um die Klausel

laws <Gesetze>

ergänzt.

Beispiel: Der Typ D=(Σ,E) ist ein abstrakter Datentyp

```
type D =
    sorts b
    functions
        t: →b
        f: →b
        n: b→b
        u: b×b→b
        o: b×b→b
    laws
        n(t)≡f
        n(f)≡t
        ©x (©y (u(x,y)⇒o(x,y)))
        ©x (u(f,x)≡f)
        ©x (©y (u(x,y)≡u(y,x)))
        ©x (©y (o(x,y)≡o(y,x)))
        ©x (o(t,x)≡t))
end.
```

Ein Modell für diesen abstrakten Datentyp ist z.B. die Menge der Wahrheitswerte IB mit den üblichen Operationen not, and, or.

Man ist nun bestrebt, die Anzahl der möglichen Modelle zu einem abstrakten Datentyp einzuschränken. Man könnte zum Beispiel die Gesetze so formulieren, daß im wesentlichen nur noch ein Modell übrig bleibt. Wenn es bis auf *Isomorphie* nur ein einziges Modell zu einem abstrakten Datentyp gibt, dann nennt man diesen Datentyp **monomorph**; gibt es dagegen mehrere wesentlich verschiedene Modelle, so heißt er **polymorph**. Man ist aber keineswegs nur an monomorphen abstrakten Datentypen interessiert; denn in der Praxis stellen die Modelle meist die unterschiedlichen Möglichkeiten dar, wie man einen abstrakten Datentyp in einer Programmiersprache implementieren kann. Man will die Implementierung nur insoweit einschränken, als die gewünschten Eigenschaften, die sich in den Gesetzen ausdrücken, alle berücksichtigt werden.

Beispiel: Abschließend definieren wir unter Zusammenfassung aller bisherigen Ergebnisse als Beispiel den (monomorphen) Datentyp bool, so wie er in den meisten Programmiersprachen gegeben ist:

type bool =
  sorts BOOL
  functions
    true: →BOOL
    false: →BOOL
    not: BOOL→BOOL
    and: BOOL×BOOL→BOOL
    or: BOOL×BOOL→BOOL
  laws
    not(true≡false) Hier is not der Operator aus Def. E und nicht die not-Funktion
    not(false)≡true
    not(true)≡false
    ©x (and(true,x)≡x)
    ©x (and(false,x)≡false)
    ©x (©y (or(x,y)≡not(and(not(x),not(y)))) DeMorgansche Regel
end.

Mit zwei weiteren Beispielen, in denen wir zwei fundamentale Datentypen der Informatik einführen und als abstrakte Datentypen beschreiben, schließen wir diesen Abschnitt ab.

Beispiele:

1) Der Datentyp stack.
   Einen Datentyp s über einem Grundtyp T bezeichnet man als **Stack**, **Keller** oder **Stapel**, wenn es drei Zugriffsfunktionen gibt, von denen die eine ein Element von T in s einfügt, die andere stets das zuletzt eingefügte Element von s entfernt und die dritte

das zuletzt eingefügte Element als Ergebnis liefert, ohne es jedoch zu entfernen. Die Einfügeoperation nennt man üblicherweise push, die Ausfügeoperation pop, die dritte Funktion top. Datenelemente müssen aus einem Stack also in genau der umgekehrten Reihenfolge wieder entfernt werden, in der sie eingefügt wurden. Dieses Prinzip bezeichnet man als **LIFO-Prinzip** (Abk. last in first out).
Wir definieren den Stack als abstrakten Datentyp:

type STACK =
  sorts T, bool, stack
  functions
    empty: →stack
    push: stack×T→stack
    pop: stack→stack
    top: stack→T
    is_empty: stack→bool
  laws
    is_empty(empty)
    ©s (©x (not(is_empty(push(s,x)))))
    pop(empty)≡empty
    ©s (©x (pop(push(s,x))≡s))
    ©s (©x (top(push(s,x))≡x))
end.

2) Der Datentyp queue.
Einen Datentyp q über einem Grundtyp T bezeichnet man als **Queue** oder **Schlange**, wenn es drei Zugriffsfunktionen gibt, von denen die eine ein Element von T in q einfügt, die andere stets das zuerst eingefügte Element von q entfernt und die dritte das zuerst eingefügte Element als Ergebnis liefert, ohne es jedoch zu entfernen. Die Einfügeoperation nennt man üblicherweise enter oder enqueue, die Ausfügeoperation remove oder dequeue, die dritte Funktion first. Datenelemente müssen aus einer Queue also genau in der gleichen Reihenfolge wieder entfernt werden, in der sie eingefügt wurden. Dieses Prinzip bezeichnet man als **FIFO-Prinzip** (Abk. first in first out).
Wir definieren die Queue als abstrakten Datentyp:

type QUEUE =
  sorts T, bool, queue
  functions
    empty: →queue
    enter: queue×T→queue
    remove: queue→queue
    first: queue→T
    is_empty: queue→bool

```
    laws
        is_empty(empty)
        ©q (©x (not(is_empty(enter(q,x)))))
        remove(empty)≡empty
        ©q (©x (is_empty(q) ⇒ remove(enter(q,x))≡empty))
        ©q (©x (not (is_empty(q)) ⇒
                remove(enter(q,x))≡enter(remove(q,x))))
        ©q (©x (is_empty(q) ⇒ first(enter(q,x))≡x))
        ©q (©x (not (is_empty(q)) ⇒
                first(enter(q,x))≡first(q)))
end.
```

In beiden Beispielen wird – wie beabsichtigt – keinerlei Bezug auf eine mögliche Implementierung der Datentypen genommen; es bleibt also der Implementierungsentscheidung überlassen, ob sie einen Stack oder eine Queue durch ein Array, eine Linkssequenz, eine Rechtssequenz oder was immer realisiert.

2.3 Konkrete Datentypen

Abstrakte Datentypen sind in der bisher besprochenen allgemeinen Form in keiner Programmiersprache realisiert. Stattdessen kann man in allen gängigen Programmiersprachen konkrete Datentypen formulieren, die in mehr oder weniger großem Umfang um Abstraktionsmöglichkeiten angereichert sind. Die entsprechenden Programmstücke bezeichnet man z.B. als *Module* (in MODULA-2), als *Klassen* (in SMALLTALK-80), als *Pakete* (in ADA) oder als *Strukturen* (in ML). Es sind in sich zusammenhängende Bausteine, die stets folgende Eigenschaften besitzen:

- Sie sind logisch oder funktional in sich abgeschlossen.
- Es ist nach außen nur ihre Funktionalität bekannt, aber nicht wie diese Funktionalität realisiert sind (*information hiding*).
- Sie besitzen klar definierte Schnittstellen nach außen.
- Sie sind überschaubar und damit leicht zu testen.

Module (diese Bezeichnung verwendet man vorwiegend) haben den Charakter von *Anwendungen*, also vollständiger, in sich geschlossener Problemlösungen für einen bestimmten Anwendungsbereich. Wird ein Modul an einer allgemein zugänglichen Stelle, in einer sog. *Bibliothek*, im Computersystem gespeichert, so kann es von anderen Programmen genutzt werden. Modularisierung ist das zentrale Prinzip bei der Entwicklung von Software: Man unterteilt ein komplexes Gesamtsystem in unabhängig voneinander realisierbare und in ihrem Zusammenwirken überschaubare Module und beschreibt die Funktion dieser Bausteine und ihre Schnittstellen.

Wir behandeln im folgenden die Ausprägung konkreter Datentypen in ML. Hier stehen

zwei verschiedene Mechanismen zur Verfügung:

- ein eingeschränktes Konzept, geeignet für kleinere Anwendungen unter dem Schlüsselwort abstype: Hierbei handelt es sich um die Zusammenfassung von Daten mit darauf definierten Operationen zu einer Einheit. Jedoch kann die Spezifikation eines Typs nicht von seiner Implementierung getrennt werden. In der Sprechweise von Abschnitt 2.1 handelt es sich also um konkrete Datentypen ohne jedes abstrakte Element.
- ein leistungsfähiges Konzept, sog. *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 (die Signatur des Datentyps) mit dem Schlüsselwort signature vom Implementierungsteil trennen. Ferner kann man Module mithilfe von Funktoren (Schlüsselwort: functor) sehr elegant parametrisieren. Der Abstraktionsgrad ist damit höher als bei den abstypes.

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

2.3.1 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 implementiert wird. Die Deklaration der Operationen erfolgt in der üblichen Form durch fun oder val.

Beispiel: Wir definieren einen konkreten (polymorphen) Datentyp queue und implementieren ihn durch lineare Listen (=Repräsentationstyp):

```
abstype ,type queue = q of ,type list
with
    val  empty = q [ ];
    fun  enter x (q y) = q (y@[x]);
    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 definierten Operationen möglich. Die Implementierung der Schlange 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. Für den Benutzer sichtbar ist also nur die Signatur des Datentyps:

```
type Schlange =
    sorts bool, ,type, ,type queue
    functions
        empty: → ,type queue
        enter: ,type → ,type queue → ,type queue
        remove: ,type queue → ,type queue
        first: ,type queue → 'type
        is_empty: ,type queue → bool
end.
```

Beispiel: Der Datentyp set implementiert durch eine lineare Liste:

```
abstype ,type set = s of ,type list
with
    val  empty = s [ ];
    local
        fun  member [ ] x = false |
             member (z::y) x = (z=x) orelse 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* Menge $M \subseteq X$ (X Universum) zur *endlichen* Beschreibung der Menge mithilfe der sog. *charakteristischen Funktion*

χ_M: X→bool mit

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

überzugehen. Die charakteristische Funktion muß natürlich *berechenbar* sein, d.h. es muß eine maschinell nachvollziehbare Beschreibung der Funktion geben.
Dann repräsentiert z.B. die konstante Funktion false die leere Menge, die konstante Funktion true die Menge X, eine 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
```

2.3.2 Module: structure, signature, functor

Module sind in ML meist zweigeteilt: Sie bestehen aus einer Signatur (Schlüsselwort: signature), deren Syntax sich an der Syntax von Signaturen gem. Abschnitt 2.2 orientiert, 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 2.3.1 definierte Typ queue lautet als Struktur:

```
structure queue = struct
    datatype ,a T = q of ,a list;
    val   empty = q [ ];
    fun   enter x (q y) = q (y@[x]);
    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) 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 inspect für Records).

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 eines local-Konstrukts (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 (entsprechen Sortendeklarationen à la sorts ... in 2.2)

  type <ggf. Liste von Typvariablen> <Typbezeichner>

- datatype-Definitionen.

Beispiele:

1) Die oben definierte Struktur queue besitzt 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.
```

Von allen in der Struktur definierten Objekten gehören nur diejenigen zur Exportschnittstelle, die in der Signatur aufgelistet sind. 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 Monoids 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 eine Signatur erfüllt. Es gelten drei Regeln, von denen die ersten beiden bestimmen, wann eine Struktur zu einer 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. Funktoren 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 funktionsartigen 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 Funktorrumpf 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 spezifizierten 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 [(succ(succ null),succ null),(null,succ null),...].
```

Abschließend noch ein Beispiel zur mathematischen Nutzung des Funktorkonzepts.

Beispiel: Ein Ring ist eine allgemeine mathematische Struktur bestehend aus einer Menge M, gewissen auf M definierten Operationen +, -, *, die bestimmte Gesetze erfüllen, u.a. das Kommutativgesetz bei +, und zwei je nach Operation neutralen Elementen $0,1 \in M$. Wir definieren einen Funktor, der zu einem Ring R=(M;0,1,+,-,*) den Ring der 2×2-Matrizen erzeugt. Dabei ist dann das Nullelement die Nullmatrix und das Einselement die Einheitsmatrix, die Operationen sind in der üblichen Weise definiert:

```
signature ring=sig
    type R;
    val null: R;
    val eins: 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));
    val   eins=((act_ring.eins, act_ring.null),(act_ring.null,act_ring.eins));
    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.
```