In J. Mühlbacher, editor, Datenstrukturen, Graphen, Algorithmen.
Appl. Comp. Sc. 8, pages 48-61, München, 1978. Hanser-Verlag.

Algebraische Spezifikation von Datenstrukturen

H.-D. Ehrich

Abteilung Informatik
Universität Dortmund
Postfach 50 05 00
D-4600 Dortmund 50

ALGEBRAISCHE SPEZIFIKATION VON DATENSTRUKTUREN

H.-D. Ehrich

Abteilung Informatik/Universität Dortmund

Abstract - The method of algebraic specification by abstract data types is demonstrated by several examples , including general storage and access systems and open hash tables. Then, after sketching the algebraic background, correct realizations of data types are considered. Several correctness proofs are outlined. Finally, problems corresponding to an appropriate notion of realization are discussed.

1. Einleitung

In der Methodologie des Programmentwurfs spielen Abstraktionen eine wesentliche Rolle, denn sie erlauben eine konsequente Verwirklichung der Ideen der strukturierten Programmierung /3/. Während Prozeduren als Mittel der Abstraktion bereits seit langem gebräuchlich sind, werden Datenabstraktionen erst seit relativ kurzer Zeit diskutiert. Wichtige Schritte in deren Entwicklung bilden das Klassenkonzept von SIMULA /2/, die Module von Parnas /16/ und die von Liskov und Zilles kürzlich eingeführten abstrakten Datentypen /12,13/. Wir wollen uns hier mit den letzteren beschäftigen, denn sie haben der Programmiermethodik wichtige Impulse gegeben (vgl. /4,6,11,12/), die sich auch in der Definition neuer Programmiersprachen äußern (z.B. CLU /14/).

Die abstrakten Datentypen haben bereits eine ganze Reihe theoretischer Untersuchungen angeregt. Guttag /11/ und die ADJ-Gruppe /8,9,1o, 17/ verwenden heterogene Algebren (s./1/) zur Beschreibung, und kategorientheoretische Ansätze stammen von Wand /18,19/ und von Goguen /8/. Diese mathematischen Grundlagen, deren Entwicklung bei weitem nicht abgeschlossen ist, eröffnen Möglichkeiten für Korrektheitsbeweise für abstrakte Datentypen, und damit für algebraisch spezifizierte Programmsysteme, und für deren Realisierung.

Im nächsten Abschnitt wird die Methode der algebraischen Spezifikation anhand einiger Beispiele demonstriert, darunter als etwas umfangreicheres Beispiel eine offene Hashtabelle. Danach werden die mathema-

tischen Grundlagen kurz skizziert, wobei wir im wesentlichen den Darstellungen der ADJ-Gruppe folgen /8,9,1o,17/. Im letzten Abschnitt betrachten wir Realisierungen von abstrakten Datentypen. Wir umreißen exemplarisch einige Korrektheitsbeweise für Realisierungen und beleuchten dann die Problematik der adäquaten Präzisierung des Begriffs der Realisierung.

2. Spezifikation

Abstrakte Datentypen werden gebildet aus einem Bündel von Operationen auf verschiedenen Bereichen, zusammen mit Gleichungen - den Axiomen - die die Beziehungen zwischen diesen Operationen festlegen. Die Beschreibungsweise ist also rein operational. Konstante werden als nullstellige Operationen aufgefaßt. Um einen Eindruck von der Spezifikationsmethode zu geben, behandeln wir zunächst einige Beispiele.

(2.1)Beispiel: Der Datentyp bool wird gebildet aus den beiden nullstelligen Operationen true und false:

$$\begin{aligned} \text{true} &: \longrightarrow \underline{\text{bool}} \\ \text{false} &: \longrightarrow \underline{\text{bool}} \end{aligned}$$

Dieser Datentyp hat keine Axiome. Ein offensichtliches Modell ist die Menge $\mathrm{BOOL} = \{\,\mathrm{TRUE}, \mathrm{FALSE}\,\}$.

(2.2)Beispiel: Der Datentyp nat wird gebildet aus den Operationen

$$\begin{aligned} 0 &: \longrightarrow \underline{\text{nat}} \\ +1 &: \underline{\text{nat}} \longrightarrow \underline{\text{nat}} \end{aligned}$$

und hat ebenfalls keine Axiome. Ein Modell für nat sind die natürlichen Zahlen mit der Konstanten 0 und der Nachfolgerfunktion +1 .

(2.3)Beispiel: Sei E irgendein Datentyp. Ein Stack S mit Einträgen in E ist definiert durch die folgenden Operationen und Axiome:

$$\begin{aligned} \emptyset &: \longrightarrow S \\ \mathrm{push} &: S \times E \longrightarrow S \\ \mathrm{pop} &: S \longrightarrow S \\ \mathrm{top} &: S \longrightarrow E \end{aligned}$$

$$\begin{aligned} \mathrm{pop}(\emptyset) &= \mathrm{error} \\ \mathrm{pop}(\mathrm{push}(s,e)) &= s \\ \mathrm{top}(\emptyset) &= \mathrm{error} \\ \mathrm{top}(\mathrm{push}(s,e)) &= e \end{aligned}$$

Die Bezeichnung error bedeutet, daß der entsprechende Wert undefiniert ist und eine Implementierung an dieser Stelle eine Fehlermeldung vorzusehen hat. Ein Modell für einen Stack ist besonders naheliegend: sei $\bar{S}$ die Menge der Terme $push[....push[push[\emptyset,e_1],e_2],....,e_n]$, wobei $e_i \in E$ ist für $i=1,...,n$. Dann ist $\emptyset$ ein konstanter Term, push überführt einen Term $s \in \bar{S}$ und einen Eintrag $e \in E$ in den Term $push[s,e]$, und die Axiome definieren die Operationen pop und top explizit als partielle Funktionen auf $\bar{S}$.

(2.4)Beispiel: Sei E irgendein Datentyp. Eine Queue Q mit Einträgen in E ist definiert durch die folgenden Operationen und Axiome:

$$\begin{aligned} \emptyset &: \longrightarrow Q \\ put &: Q \times E \longrightarrow Q \\ get &: Q \longrightarrow Q \\ front &: Q \longrightarrow E \end{aligned}$$

$$\begin{aligned} get(\emptyset) &= error \\ get(put(\emptyset,e)) &= \emptyset \\ get(put(put(q,e),f)) &= put(get(put(q,e)),f) \end{aligned}$$

$$\begin{aligned} front(\emptyset) &= error \\ front(put(\emptyset,e)) &= e \\ front(put(put(q,e),f)) &= front(put(q,e)) \end{aligned}$$

Hier läßt sich ein ähnliches Modell konstruieren wie beim Stack: sei $\bar{Q}$ die Menge der Terme in $\emptyset$ und put, wobei die Operationen $\emptyset$ und put auf $\bar{Q}$ wie oben definiert sind. Dann geben die Axiome eine primitiv rekursive Definition der partiellen Funktionen get und front.

(2.5)Beispiel: Seien K und E irgendwelche Datentypen, und sei ein Prädikat "gleich", = : $K \times K \longrightarrow$ bool , erklärt. Dann ist ein Speicher- und Zugriffssystem Z mit Schlüssel K und Einträgen E wie folgt definiert. Man beachte, daß Bezug genommen wird auf den Datentyp nat von Beispiel 2.2 .

$$\begin{aligned} \underline{0} &: \longrightarrow Z \\ insert &: Z \times K \times E \longrightarrow Z \\ delete &: Z \times K \longrightarrow Z \\ value &: Z \times K \longrightarrow E \end{aligned}$$

$$\begin{aligned} delete(\underline{0},k) &= error \\ delete(insert(z,k,e),h) &= \underline{if}\ k=h\ \underline{then}\ z \\ &\qquad \underline{else}\ insert(delete(z,h),k,e)\ \underline{fi} \end{aligned}$$

$$\begin{aligned} value(\underline{O},k) &= error \\ value(insert(z,k,e),h) &= \underline{if}\ \ k{=}h\ \ \underline{then}\ \ e \\ &\qquad\qquad\qquad\quad \underline{else}\ \ value(z,h)\ \ \underline{fi} \end{aligned}$$

Die Bedeutung der Operationen ist offensichtlich und bedarf keiner weiteren Erklärungen. Bemerkenswert ist, daß eine speziell eingeschränkte Version von Z sich wie ein Stack verhält. Arbeitet man immer nur mit einem festen Schlüssel $k_o \in K$, interpretiert dann $\underline{O}$ als $\emptyset$, insert als push, delete als pop und value als top, so sind diese vier Axiome gerade diejenigen des Stack, da immer $k{=}h{=}k_o$ ist. Dies ist ein einfaches Beispiel für eine Realisierung von S durch Z. Auf Realisierungen werden wir im letzten Abschnitt zurückkommen.

(2.6)Beispiel: Eine andere Art von Speicher- und Zugriffssystem mit Schlüssel K und Einträgen E ist der folgende Datentyp Y.

$$\begin{aligned} \underline{O} &: \longrightarrow Y \\ in &: Y \times K \times E \longrightarrow Y \\ del &: Y \times K \longrightarrow Y \\ val &: Y \times K \longrightarrow E \end{aligned}$$

$$\begin{aligned} val(\underline{O},k) &= error \\ val(in(y,k,e),h) &= \underline{if}\ k{=}h\ \underline{then}\ e\ \underline{else}\ val(y,h)\ \underline{fi} \\ val(del(y,k),h) &= \underline{if}\ k{=}h\ \underline{then}\ error\ \underline{else}\ val(y,h)\ \underline{fi} \end{aligned}$$

Der Unterschied zu Z ist, daß durchaus $del(in(y,k,e),k) \neq y$ sein kann, und daß z.B. in Y gilt: $val(in(del(in(\underline{O},k_1,e_1),k_2),k_3,e_3),k_1)=e_1$, falls die k_i alle verschieden sind, während in Z gilt: $value(insert(delete(insert(\underline{O},k_1,e_1),k_2),k_3,e_3),k_1)=error$.

Bevor wir die mathematischen Grundlagen behandeln, wollen wir ein etwas umfangreicheres Beispiel erörtern, denn Beispiele des bisherigen Umfangs lassen die Vorzüge der algebraischen Spezifikationsmethode vielleicht noch nicht sehr deutlich in Erscheinung treten.

(2.7)Beispiel: Wir spezifizieren den Datentyp offene Hashtabelle H (vgl./15/,S.218) und idealisieren insoweit, als die Tabelle nicht beschränkt sein soll. Im nächsten Beispiel betrachten wir dann beschränkte Hashtabellen. Wir definieren H als spezielles Speicher- und Zugriffssystem der Art Y mit Schlüsselmenge nat, hier auch "Adressen" genannt, und Einträgen aus $K \times E$. Das heißt: die Operationen und Axiome für Y im vorigen Beispiel gehören zu H. Darüberhinaus sei

$$hash : K \times nat \longrightarrow nat$$

eine gegebene - hier nicht näher spezifizierte - Operation. Wir nennen hash(k,i) die "i-te Hashadresse" des Schlüssels k. Für i=0 heißt sie auch "Hausadresse" und für i>0 "i-te Ausweichadresse". Anstelle der Operation val : H × nat ⟶ K × E führen wir der Kürze halber die beiden Operationen

$$\text{key} = \text{pr}_1 \circ \text{val}$$
$$\text{ent} = \text{pr}_2 \circ \text{val}$$

ein. Zusätzlich benötigen wir folgende Operationen:

$$\text{occup} : H \times \underline{\text{nat}} \longrightarrow \underline{\text{bool}}$$
$$\text{used} : H \times \underline{\text{nat}} \longrightarrow \underline{\text{bool}}$$
$$\text{free} : H \times K \times \underline{\text{nat}} \longrightarrow \underline{\text{nat}}$$
$$\text{find} : H \times K \times \underline{\text{nat}} \longrightarrow \underline{\text{nat}}$$

occup und used testen, ob die Adresse besetzt ist bzw. jemals benutzt wurde, free(h,k,i) ist die erste freie Ausweichadresse nach der i-ten Hashadresse von k, und find(h,k,i) ist die erste Ausweichadresse nach der i-ten Hashadresse von k, in der ein Eintrag (k,e) mit Schlüssel k gespeichert ist. Die Axiome präzisieren dies:

occup(0,i) = false
occup(in(h,i,k,e),j) = if i=j then true else occup(h,j) fi
occup(del(h,i),j) = if i=j then false else occup(h,j) fi

Die Axiome für used unterscheiden sich von denen für occup nur in der letzten Zeile: dort steht auf der rechten Seite true statt false.

```
free(h,k,i)=if occup(h,hash(k,i))
            then if k=key(h,hash(k,i)) then error else free(h,k,i+1) fi
            else  i      fi

find(h,k,i)=if occup(h,hash(k,i))
            then if k=key(h,hash(k,i)) then i else find(h,k,i+1) fi
            else if used(h,hash(k,i)) then find(h,k,i+1) else error fi fi
```

Einfügen, Löschen und Zugriff in einer Hashtabelle geschieht dann durch folgende zusammengesetzten Operationen:

inhash(h,k,e) = in(h,hash(k,free(h,k,0)),k,e)
delhash(h,k) = del(h,hash(k,find(h,k,0)))
valhash(h,k) = ent(h,hash(k,find(h,k,0)))

Sofern hash(k,nat) für jeden Schlüssel k eine unendliche Menge ist, realisieren diese Operationen von H ein Speicher- und Zugriffssystem der Art Y (s. Abschnitt 4). Hierbei treten die Schlüssel von Y als

als Teil der Einträge in H auf, und nat wird als Standard-Schlüsselmenge benutzt.

(2.8)Beispiel: Eine unbeschränkte Hashtabelle ist nicht sehr realistisch. Wir können die obige Spezifikation für H leicht in eine für eine beschränkte Hashtabelle H_n mit höchstens n Einträgen umwandeln. Dazu sei

$$\underline{n} = \{ 0,1,\ldots,n-1 \}$$

ein Datentyp mit n Konstanten, den "Adressen", einer einstelligen Operation +1 mit 0+1=1, 1+1=2, ... , (n-2)+1 = n-1, (n-1)+1=error , und einem Gleichheitsprädikat. Dann ergibt sich die Spezifikation für H_n aus der von H, indem man überall nat durch $\underline{n}$ ersetzt. Der error, der bei (n-1)+1 auftritt, pflanzt sich dann in den Operationen free und find fort und bedeutet dort Überlauf bzw. Fehlanzeige. Als Hashfunktion für eine beschränkte Hashtabelle H_n kann irgendeine der aus der Literatur bekannten genommen werden (s. etwa /15/,S.219ff).

3. Mathematische Grundlagen

Ziel dieses Abschnitts ist es, den Begriff des abstrakten Datentyps nach Goguen et al /9/ exakt zu definieren. Die dazu nötigen algebraischen Begriffe und Grundlagen werden nur ganz kurz skizziert. Detailliertere Beschreibungen findet man in /8,9,1o,17/, und andere Ansätze in /7,8,18, 19/ .

Sei S eine Menge von Sorten oder Bereichssymbolen. Zu einem Wort $w \in S^*$ und einem Symbol $s \in S$ sei Σ_{ws} eine Menge von Operationssymbolen; für jedes $\sigma \in \Sigma_{ws}$ sind w der Rang und s die Sorte von σ . Eine mit S^+ indizierte Familie von Mengen $\Sigma = \{ \Sigma_{ws} \}_{ws \in S^+}$ nennen wir eine Signatur.

(3.1)Definition: Eine Σ-Algebra A ist eine heterogene Algebra mit Trägermengen A_s für jede Sorte $s \in S$ und Operationen $\sigma_A : A_w \longrightarrow A_s$ für jedes $\sigma \in \Sigma_{ws}$.
Hierbei ist $A_w = A_{s_1} \times \ldots \times A_{s_n}$, falls $w = s_1 \ldots s_n$ ist.
Bzgl. des Begriffs der heterogenen Algebra sei auf /1/ verwiesen.

(3.2)Beispiel: Ein Stack S nach Beispiel 2.3 ist eine Σ-Algebra mit $\Sigma_S = \{\emptyset\}$, $\Sigma_{SES} = \{push\}$, $\Sigma_{SS} = \{pop\}$, $\Sigma_{SE} = \{top\}$ und $\Sigma_{ws} = \emptyset$ für alle anderen $ws \in \{S,E\}^+$.

(3.3)Definition: Seien A,B zwei Σ-Algebren. Ein Σ-Homomorphismus $h : A \longrightarrow B$ ist eine Familie von Abbildungen $\{ h_s : A_s \longrightarrow B_s \}_{s \in S}$, so daß $h_s(\sigma_A(a)) = \sigma_B(h_w(a))$ ist für alle $s \in S$, $\sigma \in \Sigma_{ws}$ und $a \in A_w$.

Hierbei ist $h_w(a)=(h_{s_1}(a_1),\ldots,h_{s_n}(a_n))$, falls $w=s_1\ldots s_n$ ist und $a=(a_1,\ldots,a_n)$.

Die Kategorie aller Σ-Algebren mit allen ihren Homomorphismen bezeichnen wir mit $\underline{\underline{Alg}}_\Sigma$.

Von besonderem Interesse sind Terme, die sich aus Variablen und den Operationssymbolen einer Signatur zusammensetzen. Sei $Y=\{Y_s\}_{s\in S}$ eine Familie von Variablenmengen. Zu $y\in Y_s$ sagen wir, y sei eine Variable der Sorte s .

(3.4)Definition: Die Menge der Terme der Sorte s bzgl. einer Signatur und einer Familie Y von Variablenmengen ist

$$T_\Sigma(Y)_s = Y_s \cup \{\sigma[t_1,\ldots,t_n] \mid \sigma\in\Sigma_{s_1\ldots s_n s} \wedge \bigwedge_{i=1}^{n} t_i\in T_\Sigma(Y)_{s_i}\}$$

Aus der Familie $T_\Sigma(Y)=\{T_\Sigma(Y)_s\}_{s\in S}$ kann man eine Σ-Algebra konstruieren, indem man für jedes $\sigma\in\Sigma_{ws}$ die Operation

$$\sigma_T(t_1,\ldots,t_n) = \sigma[t_1,\ldots,t_n]$$

einführt. Diese Algebra bezeichnen wir ebenfalls mit $T_\Sigma(Y)$. Wir zitieren ohne Beweis einen wichtigen Satz über $T_\Sigma(Y)$. Den Beweis findet man in /9/ .

(3.5)Satz: $T_\Sigma(Y)$ ist eine frei von Y erzeugte Σ-Algebra in dem Sinne, daß für alle A in $\underline{\underline{Alg}}_\Sigma$ und für jede Familie $\vartheta=\{\vartheta_s:Y_s\longrightarrow A_s\}_{s\in S}$ von Abbildungen ein eindeutiger Homomorphismus $\bar{\vartheta}: T_\Sigma(Y)\longrightarrow A$ existiert, der ϑ erweitert.

Daraus ergibt sich die Interpretation von Termen aus $T_\Sigma(Y)$ als abgeleiteten Operationen in einer Σ-Algebra A : ϑ beschreibt eine Belegung der Variablen mit Werten, und der Satz garantiert die eindeutige Auswertung. Ein Term $\tau\langle y_1,\ldots,y_m\rangle$ der Sorte s, in dem die Variablen $y_i\in Y_{s_i}$ für i=1,...,m vorkommen, beschreibt somit eine Operation : $A_{s_1}\times\ldots\times A_{s_m}\longrightarrow A_s$.

Die Axiome eines abstrakten Datentyps sind Gleichungen, bei denen auf der linken und auf der rechten Seite je ein Term der gleichen Sorte steht. Dementsprechend definieren wir:

(3.6)Definition: Ein Σ-Axiom der Sorte s ist ein Paar $e=\langle L,R\rangle$ mit $L,R\in T_\Sigma(Y)_s$. Eine Σ-Algebra erfüllt das Axiom e gdw. $\bar{\vartheta}(L)=\bar{\vartheta}(R)$ ist für alle Abbildungsfamilien $\vartheta: Y\longrightarrow A$.

(3.7)Definition: Sei $\mathcal{E}$ eine Menge von Σ-Axiomen. Dann ist A eine $(\Sigma, \mathcal{E})$-Algebra gdw. A eine Σ-Algebra ist, die alle Σ-Axiome $e \in \mathcal{E}$ erfüllt.

Die Kategorie aller $(\Sigma, \mathcal{E})$-Algebren mit allen ihren Homomorphismen bezeichnen wir mit $\underline{\underline{Alg}}_{\Sigma,\mathcal{E}}$. Wesentlich für die Definition von abstrakten Datentypen ist der Begriff der initialen Algebra (s./9,1o/). Diese ist initiales Element in einer Kategorie von Σ-Algebren.

(3.8)Definition: Eine Σ-Algebra A heißt initial in einer Kategori $\underline{\underline{C}}$ von Σ-Algebren gdw. für jede Σ-Algebra B in $\underline{\underline{C}}$ ein eindeutiger Homomorphismus $h : A \longrightarrow B$ existiert.

Es ist bekannt, daß initiale Algebren in $\underline{\underline{C}}$ isomorph sind, und daß jede zu einer initialen isomorphe Algebra wiederum initial in $\underline{\underline{C}}$ ist. Darüberhinaus gilt der folgende grundlegende Satz.

(3.9)Satz: In $\underline{\underline{Alg}}_{\Sigma,\mathcal{E}}$ existiert eine initiale Algebra.

Ein abstrakter Datentyp wird präsentiert durch die Angabe der Sorten S, der Operationen Σ und der Axiome $\mathcal{E}$. Es liegt daher nahe, die in $\underline{\underline{Alg}}_{\Sigma,\mathcal{E}}$ existierende und damit bis auf Isomorphie eindeutige initiale Algebra als Explikat des Begriffs "abstrakter Datentyp" zu wählen, wie dies in /8/ geschieht.

(3.1o)Definition: Ein abstrakter Datentyp ist die initiale Algebra in einer Kategorie $\underline{\underline{C}}$ von Σ-Algebren.

Wird ein abstrakter Datentyp durch eine Gleichungsmenge $\mathcal{E}$ angegeben, so kann man eine Konstruktion für die initiale Algebra in $\underline{\underline{Alg}}_{\Sigma,\mathcal{E}}$ angeben.

(3.11)Satz: Sei $\equiv_{\mathcal{E}}$ die von $\mathcal{E}$ erzeugte Kongruenzrelation in $T_\Sigma = T_\Sigma(\emptyset)$. Dann ist $T_\Sigma/\equiv_{\mathcal{E}}$ initial in $\underline{\underline{Alg}}_{\Sigma,\mathcal{E}}$.

Bezüglich des Beweises sei wiederum auf /8/ verwiesen.

4. Realisierung

Ein Vorteil der algebraischen Spezifikation liegt in ihrer Abstraktheit. Abstrakt bedeutet in diesem Zusammenhang, daß die Beschreibung unabhängig ist von jeder Implementierung, und das heißt, daß sie auch gültig ist für jede Implementierung. Diese Abstraktheit bedeutet aller-

dings auch, daß man in der Spezifikation keine Hinweise auf mögliche und günstige Implementierungen erhält.

Ein weiterer Vorteil der algebraischen Spezifikation liegt in ihrer Exaktheit. Sie gibt damit eine Grundlage für zweierlei Arten von Korrektheitsbeweisen. Zum einen lassen sich aus der Spezifikation Eigenschaften der Operationen beweisen, woraus sich in gewünschtem Umfang die Sicherheit herleiten läßt, daß die Spezifikation der ursprünglich vorgestellten Intention entspricht. Zum anderen gibt die Spezifikation ein Maß, an dem sich die Korrektheit einer Implementierung messen läßt. Wir wenden uns hier der letzteren Problematik zu und erörtern insbesondere die Frage, welche definierende Beziehung zwischen einem abstrakten Datentyp und seiner Implementierung besteht.

(4.1)Beispiel: Betrachten wir den Datentyp Stack von Beispiel 2.3 . Eine gängige Implementierung ist die folgende, wobei wir ALGOL-ähnliche Notation verwenden und unwesentliche Details weglassen:

```
E array S [1:flex] ; int t ;

proc push(S,t,e); ...
     begin t:=t+1; S[t]:=e end  ;

proc pop(S,t); ...
     if t=0 then error else t:=t-1 ;

E proc top(S,t); ...
     if t=0 then error else S[t] ;
```

Um zu zeigen, daß diese Implementierung korrekt ist, müssen wir nachweisen, daß sie sich so verhält, wie es die Spezifikation vorschreibt. Hier ist der Stack durch das Paar (S,t) dargestellt, und t=0 stellt den leeren Stack $\emptyset$ dar. Es ist immer $t \geq 0$. Wenn man den Axiomen intuitiv folgt, wird man folgendes zeigen:

$$t=0 \Rightarrow pop(S,t) = error$$
$$t>0 \Rightarrow \{P\}\ push(S,t,e);pop(S,t)\ \{P\}$$

$$t=0 \Rightarrow top(S,t) = error$$
$$t>0 \Rightarrow \{true\}\ push(S,t,e)\ \{top(S,t) = e\}$$

Die zweite Beziehung sollte für alle "vernünftigen" Zusicherungen P gelten, die den Ausdruck S[t+1] nicht enthalten. Die Beweise sind offensichtlich und werden hier übergangen.

Es ist i.a. keineswegs klar, wie man auf systematische Weise Verifikationsbedingungen für implementierende Programme aus der algebraischen

Spezifikation gewinnt. Ein gangbarer Weg ist es, den Begriff der korrekten Realisierung als Beziehung zwischen abstrakten Datentypen einzuführen. Damit erhält man die attraktive Möglichkeit, die Realisierung stufenweise mit wachsender Detaillierung vorzunehmen, d.h. eine Folge $A=A_0,A_1,\ldots,A_n$ von abstrakten Datentypen anzugeben, so daß A_i den Typ A_{i-1} korrekt realisiert für $i=1,\ldots,n$. Dann realisiert auch A_n den Typ A korrekt. Um die Implementierung durch Programme einer Programmiersprache einzubeziehen, muß man dann ein Programm als Term in einem abstrakten Datentyp auffassen, der durch die Programmiersprache gegeben wird, d.h. man muß die Semantik algebraisch spezifizieren. Dies Problem können wir hier nicht behandeln. Ansätze dazu finden sich in /5,1o,18,19/.

Es liegt nahe, den Begriff der Realisierung eines Datentyps A durch einen anderen Datentyp B folgendermaßen zu präzisieren (vgl./9,11/): man ordne jeder Operation von A eine - i.a. abgeleitete - Operation in B zu und überführe die Axiome von A in Gleichungen in B, indem man die zugeordneten Operationen substituiert. Lassen sich dann diese Gleichungen in B beweisen, so realisiert B den Datentyp A. Dazu betrachten wir einige Beispiele.

(4.2)Beispiel: Im Beispiel 2.5 hatten wir eine einfache Realisierung des Stacks betrachtet: für einen festen Schlüssel $k_o \in K$ hatten wir dabei folgende Zuordnung $s \rightarrow z$ getroffen:

$$\begin{aligned} \emptyset &\longrightarrow \underline{0} \\ push(s,e) &\longrightarrow insert(z,k_o,e) \\ pop(s) &\longrightarrow delete(z,k_o) \\ top(s) &\longrightarrow value(z,k_o) \end{aligned}$$

(4.3)Beispiel: Eine andere Realisierung des Stacks ergibt sich, wenn wir das Speicher- und Zugriffssystem Z mit Schlüsselmenge nat um die folgende Operation erweitern:

$$\#entry : Z \longrightarrow \underline{nat}$$

$$\begin{aligned} \#entry(\underline{0}) &= 0 \\ \#entry(insert(z,k,e)) &= \#entry(z) + 1 \end{aligned}$$

Wir treffen nun folgende Zuordnung $s \rightarrow z$:

$$\begin{aligned} \emptyset &\longrightarrow \underline{0} \\ push(s,e) &\longrightarrow insert(z,\ \#entry(z)+1\ ,\ e\ ) \\ pop(s) &\longrightarrow delete(z,\ \#entry(z)) \\ top(s) &\longrightarrow value(z,\ \#entry(z)) \end{aligned}$$

Den Beweis, daß dies eine Realisierung (in obigem Sinne) ist, führen wir nur für das Axiom pop(push(s,e))=s vor:

pop(push(s,e))
⟶ delete(insert(z, #entry(z)+1 , e) , #entry(insert(z,#entry(z)+1,e)))
= delete(insert(z, #entry(z)+1 , e) , #entry(z) + 1)
= z
⟵ s

Hierbei haben wir nacheinander die jeweils zweiten Axiome für #entry und delete benutzt. Die Verifizierung der übrigen Axiome sei dem Leser überlassen.

(4.4)Beispiel: Wie bereits im Beispiel 2.7 bemerkt, realisiert eine offene Hashtabelle H ein Speicher- und Zugriffssystem Y vermöge der Zuordnung y⟶h :

Ω	⟶	Θ
in(y,k,e)	⟶	in(h,hash(k,free(h,k,0)),k,e)
del(y,k)	⟶	del(h,hash(k,find(h,k,0)))
val(y,k)	⟶	val(h,hash(k,find(h,k,0)))

Zur Unterscheidung haben wir die Operationen Θ,in,del,val in H etwas umbezeichnet. Der Beweis, daß diese Zuordnung eine Realisierung ist, ist bereits ziemlich aufwendig. Wir beschränken uns hier auf das dritte Axiom und skizzieren den Beweis nur grob.

val(del(y,k),k') ⟶ v=val(del(h,j),j')

mit j=hash(k,find(h,k,0)) und j'=hash(k',find(del(h,j),k',0)) .
Sei h'=del(h,j) .

1.Fall: k=k' . In diesem Falle gilt das folgende Lemma, dessen Beweis hier übergangen wird.

Lemma: $\forall i \in nat : k \neq key(h',i)$

Der einzige Ausgang aus der Rekursion für find(h',k',0) ist dann der error-Ausgang, und dieser wird irgendwann erreicht, da $|hash(k',nat)| = \infty$ ist nach Voraussetzung, und nur für endlich viele $i \in nat$ kann occup(h',i)=used(h',i)=true sein. Damit ist find(h',k',0)=error, und dieser error pflanzt sich nach j' und v fort.

2.Fall: k≠k' . In diesem Fall sind k=key(h,j) und occup(h,j)=used(h,j)=true, während occup(h',j)=false ist und used(h',j)=true . Solange hash(k',i)≠j ist, ist find(h',k',i)=find(h,k',i). Ist hash(k',i)=j, so ergibt sich aufgrund der obigen Überlegungen, da

$k' \neq k = key(h,j)$ ist : $find(h',k',i) = find(h',k',i+1)$ und ebenfalls $find(h,k',i) = find(h,k',i+1)$. Durch Induktion folgt, daß $find(h',k',i) = find(h,k',i)$ ist für alle i, und zwar aufgrund der Voraussetzung $|hash(k',\underline{nat})| = \infty$, woraus folgt, daß es ein $c \in \underline{nat}$ gibt mit $hash(k',i+c) \neq hash(k',i)$. Daraus folgt, daß $j' = hash(k',find(h,k',0))$ ist und somit $j' \neq j$. Somit haben wir

$$\begin{aligned} v &= \underline{val}(h',j') = \underline{val}(\underline{del}(h,j),j') = \underline{val}(h,j') \\ &= \underline{val}(h,hash(k',find(h',k',0))) \\ &= \underline{val}(h,hash(k',find(h,k',0))) \\ &\leftarrow val(y,k') \end{aligned}$$

Zur Präzisierung des Begriffs der Realisierung seien A und A' abstrakte Datentypen mit Sorten S und S', Signaturen Σ und Σ', sowie Axiomen $\mathcal{E}$ und $\mathcal{E}'$. Sei $f: S \longrightarrow S'$ ein String-Homomorphismus, und für $ws \in S^+$ sei $(T_{\Sigma'})_{f(ws)}$ die Menge der abgeleiteten Operationen von $A'_{f(w)}$ nach $A'_{f(s)}$ im Datentyp A' (s.o. Abschnitt 3). Sei $d = \{d_{ws}: \Sigma_{ws} \longrightarrow (T_{\Sigma'})_{f(ws)}\}_{ws \in S^+}$ eine Familie von Abbildungen, die den Operationen σ in A abgeleitete Operationen $d(\sigma)$ in A' zuordnen. Dann definieren wir mit Goguen et al /9/ als <u>abgeleitete Algebra</u> $\partial A'/(f,d)$ die Σ-Algebra mit Trägermengen $A'_{f(s)}$ für $s \in S$ und Operationen $d(\sigma)$ für $\sigma \in \Sigma$. Hinreichend für die obigen Beispiele wäre es, zu sagen, daß A' dann A realisiert, wenn A isomorph ist zu $\partial A'/(f,d)$. Daß dies jedoch i.a. zu eng ist, zeigt das folgende einfache Beispiel:

(4.5)<u>Beispiel</u>: Seien N und E zwei Datentypen. Wir definieren <u>Variable</u> mit Namen aus N und Werten aus E auf zwei Arten:

dcl : N ⟶ V1	dcl : N ⟶ V2
:= : V1 ⋉ E ⟶ V1	:= : V2 E ⟶ V2
val : V1 ⟶ E	val : V2 ⟶ E
val(dcl(n)) = undef	(v:=f):=e = v:=e
val(v:=e) = e	val(dcl(n)) = undef
	val(dcl(n):=e) = e

Beide spezifizieren das übliche Verhalten einer Variablen, den zuletzt zugewiesenen Wert zu haben. Die Datentypen V1 und V2 sind jedoch nicht isomorph. Von einem adäquaten Begriff der Realisierung würden wir erwarten, daß sowohl V1 den Typ V2 als auch V2 den Typ V1 realisiert.

Goguen et al /9/ definieren den Begriff der Realisierung - dort "Implementierung" genannt - folgendermaßen: A' realisiert A genau dann, wenn A isomorph ist zu einer Faktorstruktur $(\partial A'/(f,d))/\equiv$ nach einer

Kongruenzrelation $\equiv$, die etwa als Kern eines Epimorphismus von $\partial A'/(f,d)$ auf A auftreten kann. In vielen Fällen wird man mit dieser Definition auskommen, im letzten Beispiel jedoch wäre danach V1 zwar eine Realisierung von V2, aber V2 wäre keine von V1. Daher ist wohl eine weitere Verallgemeinerung nötig.

Hierauf bezieht sich auch die Kritik von Giarratana, Gimona und Montanari /7/ . Es ist uns in dem hier gesteckten Rahmen nicht möglich, die Problematik erschöpfend zu behandeln; das soll an anderer Stelle geschehen. Wir beschränken uns auf einige Hinweise: auf der Grundlage der Idee, daß neue Datentypen gewöhnlich unter Verwendung alter, als bekannt vorausgesetzter Datentypen spezifiziert werden, werden in /7/ die Sorten eines Datentyps in "alte" und "neue" unterschieden, und nur die Terme der alten Sorten sind für das "äußere" Verhalten des Datentyps maßgebend (sog. "black-box-Semantik"). Dies führt zu dem Begriff der "äquivalenten Darstellungen" eines Datentyps, die nicht isomorph und auch nicht homomorph zu sein brauchen. Danach wären im Beispiel 4.5 die Typen V1 und V2 äquivalent bzgl. des Verhaltens der val-Operation.

Literatur

1. Birkhoff,G.-Lipson,J.D.: Heterogenous algebras. Journal of Combinatorial Theory 8 (1970), 115-133

2. Dahl,O.J.-Myhrhaug,B.-Nygaard,U.: SIMULA 67, Common Base Language. Norwegian Computing Center, Oslo 1967

3. Dahl,O.J.-Dijkstra,E.W.-Hoare,C.A.R.: Structured programming. Academic Press, London 1972

4. Dennis,J.B.: An example of programming with abstract data types. MIT, Project MAC, Computation Structures Group Memo 131

5. Ehrich,H.-D.: Algebraic semantics of type definitions and structured variables. To be published in the proceedings of FCT'77 in Poznan, Lecture Notes in Computer Science, Springer-Verlag, Berlin 1977

6. Flon,L.: Program design with abstract data types. Res. Report, Dpt. of Computer Science, Carnegie-Mellon University, Pittsburgh, June 1975

7. Giarratana,V.-Gimona,F.-Montanari,U.: Observability concepts in abstract data type specification. In: Proc. 5th MFOC Symposium, ed. by A. Mazurkiewicz, Lecture Notes in Computer Science45, Springer-Verlag, Berlin 1976, 576-587

8. Goguen,J.A.: Correctness and equivalence of data types. In: Proc. Conf. on Alg. Sys. Th., Udine, Lecture Notes in Computer Science, Springer-Verlag, Berlin 1975

9. Goguen,J.A.-Thatcher,J.W.-Wagner,E.G.: An initial algebra approach to the specification, correctness, and implementation of abstract data types, to be published in: Current trends in programming methodology 4 (R.Yeh, ed.)

10. Goguen,J.A.-Thatcher,J.W.-Wagner,E.G.-Wright,J.B.: Initial algebra semantics and continuous algebras. Journal of the ACM 24 (1977), 68-95

11. Guttag,J.V.: The specification and application to programming of abstract data types. Tech. Report CSRG-59, University of Toronto 1975

12. Liskov,B.-Zilles,S.: Programming with abstract data types. In: SIGPLAN Notices 9 (April 1974), 50-59

13. Liskov,B.-Zilles,S.: Specification techniques for data abstractions. IEEE Transactions on Software Engineering, vol. SE-1 (1975), 7-19

14. Liskov,B.: An introduction to CLU. MIT, Project MAC, Computation Structures Group Memo 136, 1976

15. Mühlbacher,J.: Datenstrukturen. Carl Hanser Verlag, München - Wien 1975

16. Parnas,D.L.: A technique for module specification with examples. Communications of the ACM 15 (1972), 330-336

17. Thatcher,J.W.-Wagner,E.G.-Wright,J.B.: Specification of abstract data types using conditional axioms (extended abstract). IBM Res. Report RC-6214, 1976

18. Wand,M.: First-order identities as a defining language. Tech. Report No. 29, Computer Science Dpt., Indiana University, Bloomington 1976

19. Wand,M.: Efficient axioms for algebra semantics. Tech. Report No. 42, Computer Science Dpt., Indiana University, Bloomington 1975