# Algebraische Spezifikation

Vorlesung an der Fakultät für Mathematik und Informatik der Universität Leipzig [1]

Dr. Rolf Hartwig

gelesen im Sommersemester 1993 und Wintersemster 1996/97

[1] nach einer vom Autor durchgesehenen Mitschrift von stud.inf. Dirk Weigenand

Quelle: http://www.informatik.uni-leipzig.de/~rhartwig

Der Autor ist jederzeit für inhaltliche Hinweise zur Verbesserung der Vorlesung als auch für Hinweise zu immer noch verbliebenen Druckfehlern dankbar.

# Inhaltsverzeichnis

# Kapitel 1

# Einführung

Im Softwarelebenszyklus gibt es Phasen, in denen eine Spezifikation (der Anforderungen an das Produkt, des Problems, der Programme) erforderlich oder wünschenswert ist.

## 1.1 Spezifikationen im Software–Lebenslauf

1. Problemspezifikation:

   - Leistungsbeschreibung
   - Pflichtenheft
   - Anforderungsdefinition

   z.B. als Grundlage eines Vertrages zwischen Auftraggeber und –nehmer.

   Dies ist noch keine Grundlage zum sofortigen Erstellen einer Software, sondern für die Erstellung einer formalen Spezifikation. Die Beschreibung des Modells erfolgt in dieser Stufe verbal.

2. Systemspezifikation:

   Ist Resultat des Systementwurfs und Grundlage des Implementierungsprozesses. Zur Beschreibung sind formale und halbformale Methoden zweckmäßig.

3. Programmspezifikation:

   Anforderungsbeschreibung der Einzelprogramme des Systems nach dessen Modularisierung.

4. Datentypspezifikation:

   Teil der Programmspezifikation; zweckmäßig beim Entwurf von Programmmen als Algorithmen über Datenstrukturen.

**Aufgaben einer Spezifikation:**

1. **Korrektheit der Spezifikation**

   Eine Spezifikation muß sich verifizieren lassen auf:

   - Übereinstimmung mit den Vorstellungen, Absichten, gemeinten Eigenschaften
   - Widerspruchsfreiheit muß auf jeder Ebene überprüfbar sein (Konsistenz)
   - hinreichende Vollständigkeit, d.h. Ausschluß unerwünschter Realisierungen

2. **Korrektheit der Implementierung**

   Die Spezifikation muß es gestatten, eine Implementierung auf Übereinstimmung mit ihr zu überprüfen („relative Korrektheit“)

3. **Automatische Entwurfshilfen**

   Die Spezifikation soll es erlauben, automatische Entwurfshilfen zu entwickeln und einzusetzen.

Daraus ergeben sich verschiedene **Anforderungen** an die Spezifikation:

1. Verständlichkeit, Überschaubarkeit (Modularität)

2. präzise Semantik, d.h. der Spezifikationstext soll eindeutig interpretierbar sein.

3. Ein gewisser Abstraktiongrad sollte gegeben sein (Freiheit bei der Realisierung der Implementierung bezüglich der Algorithmen, etc.), also ohne irrelevante Details ausdrückbar.

## 1.2 Datentypen und abstrakte Datentypen

Unter Datentypen versteht man im allgemeinen eine Zusammenfassung gewisser Mengen von „Werten", z.B. nat, real, array zusammen mit Operationen, die auf diesen Mengen erklärt sind und nicht herausführen.

Damit ist aus mathematischer Sicht klar, daß Datentypen heterogene (mehrsortige) Algebren sind.

**Beispiel**

**NATVERGL:**

| | | | |
|---|---|---|---|
| Wertemengen | bool | = | {true, false} |
| | nat | = | $\{0, 1, 2, \ldots\}$ |
| Operationen | true: | $\mapsto$ | bool |
| | false: | $\mapsto$ | bool |
| | 0 : | $\mapsto$ | nat |
| | succ: | $\mapsto$ | nat |
| | $\neg$ :bool | $\mapsto$ | bool |
| | $\vee$ :bool $\times$ bool | $\mapsto$ | bool |
| | $\leq$ :nat $\times$ nat | $\mapsto$ | bool |

In der Praxis werden kompliziertere Datentypen oft sukzessive aus einzelnen Modulen aufgebaut. Im Beispiel bietet sich an, die Datentypen BOOL und NAT zu erklären und diese dann zusammenzusetzen und Vergleichsoperationen $\leq$ hinzuzufügen.

Für die Verwendung von Datentypen im Softwareentwurf sind diese in geeigneter Form festzulegen, zu definieren, zu spezifizieren. Dabei beschränkt man sich auf die wesentlichen Eigenschaften, die der Datentyp haben soll.

Eine eindeutige und vollständige Beschreibung wird im allgemeinen nicht gelingen und ist auch nicht erwünscht (Implementierungsfreiheiten, Effizienz!, schrittweise Verfeinerung).

Eine solche unvollständige Charakterisierung eines Datentyps meint man, wenn man von einem abstrakten Datentyp spricht. Damit ist ein abstrakter Datentyp als eine Klasse von Datentypen, also als eine Algebrenklasse anzusehen. In diese Klasse gehören alle diejenigen konkreten Datentypen, die der Beschreibung/Charakterisierung genügen.

**Definition 1.1**

(vorläufige)

Ein abstrakter Datentyp ist eine Klasse von Algebren (Datentypen).

**Beispiel**

abstrakter Datentyp ORD „geordnete endliche Menge“ ist endliche Menge $M$ mit einer darauf erklärten Ordnung $\leq$.

Verwendung dieses abstrakten Datentyps zum Entwurf von Sortierverfahren.

Zu diesem abstrakten Datentyp gehören z.B.:

- endliche Menge nat. Zahlen mit $\leq$
- endliche Menge ganzer Zahlen mit $\leq$
- endliche Menge rat. Zahlen mit $\leq$
- endliche Menge reeller Zahlen mit $\leq$
- endliche Menge von Wörtern mit einer lexikographischen Ordnung
- Menge von Angestellten mit Ordnung bezüglich des Gehalts.

Zur Präzisierung:

Um $(M, \leq)$ als Algebra auffassen zu können, füge man bool hinzu und fasse $\leq$ als Operation auf:

$$\leq: M \times M \mapsto \text{bool}$$

ORD : Klasse von Algebren $(M, \text{bool}, \leq)$.

**Beispiel**

Abstrakter Datentyp COUNT8: Zähler modulo 8.

8elementige Menge count8 mit zwei Operationen:

- reset:$\mapsto$ count8 (Rücksetzen/Löschen)
- increment:count8$\mapsto$ count8 (Weiterzählen)

mit den Eigenschaften:

- 8malige Anwendung von increment nach reset führt zum selben Resultat wie reset.
- alle Resultate/Werte sind von reset aus mittels increment erreichbar.

Zu COUNT8 gehört dann z.B. der Datentyp mit
count8 $= \{0, 1, 2, 3, 4, 5, 6, 7\}$,

$$\text{reset} = 0, \qquad \text{increment(n)} = \begin{cases} n+1 & \text{für } 0 \leq n \leq 6 \\ 0 & \text{für n=7} \end{cases},$$

aber auch der Datentyp
count8 $= \{0, -1, -2, -3, -4, -5, -6, -7\}$,

$$\text{reset} = 0, \qquad \text{increment(n)} = \begin{cases} n-1 & \text{für } -6 \leq n \leq 0 \\ 0 & \text{für } n = -7 \end{cases}$$

oder
count8 $= \{-10, -8, -6, -4, -2, 0, 2, 4\}$,

$$\text{reset} = -10, \qquad \text{increment(n)} = \begin{cases} -10 & \text{für n} = 4 \\ n+2 & \text{sonst} \end{cases}.$$

Es fällt auf, daß alle Datentypen, die zu COUNT8 gehören, isomorph zueinander sind.

Im Gegensatz dazu gehören zu ORD sehr verschiedene nichtisomorphe Datentypen. Man sagt, COUNT8 ist ein monomorpher abstrakter Datentyp, der abstrakte Datentyp ORD dagegen ein polymorpher abstrakter Datentyp.

**Definition 1.2**

Ist ein abstrakter Datentyp eine Isomorphieklasse von Algebren, so heißt der abstrakte Datentyp *monomorph*, sonst *polymorph*.

## 1.3 Spezifikationsmethoden

1. Verbale Spezifikation

   Gebrauch der natürlichen Sprache.

   Eine präzise Semantik ist nicht gegeben und damit sind automatische Entwurfshilfen unmöglich. Auch ist die Verifikation der Korrektheit zumindest fraglich.

   Das heißt also, daß eine solche Spezifikation nur in den ersten Stadien des Softwareentwurfes einsetzbar ist (Vorstellungen, Ideen umreißen usw.).

2. Formale Spezifikation
   Es ist klar, daß formale Methoden unumgänglich sind, die Frage ist, welche ?
   Es gibt eine Fülle verschiedener Konzepte!

   - imperative Methoden (zustandsorientiert)
   - applikative Methoden (Funktionen, Ausdrücke)

   Jede dieser Methoden kann jeweils

   | *exemplarisch* | | *axiomatisch* |
   |---|---|---|
   | | oder | |
   | Angabe eines konkreten Repräsentanten (bei Kenntnis der klassenbildenden Äquivalenzrelation) | | Angabe der klassendefinierenden Eigenschaften |

   orientiert sein.

1. **Imperative Methoden**

   Benutzung von Mitteln höherer Programmiersprachen

   (a) **exemplarische imperative Spezifikation**

   - Datenelemente entsprechen der Darstellung in einer Programmiersprache, z. B. kann ein Keller durch arrays dargestellt werden.
   - Operationen sind dann Prozeduren oder Funktionen, z. B. ist `push` eine Funktionsprozedur, die ein Keller–array um ein Element erweitert.

   **Nachteil:** Diese Spezifikation ist *nicht* abstrakt genug. Es ist keine eigentliche Spezifikation, sondern eher eine konkrete Implementierung.

   (b) **axiomatische imperative Spezifikation** Benutzung einer Programmlogik, z. B. haben die Spezifikationsformeln folgende Gestalt:

   $$\{p\}S\{q\},\ S \text{ –Programmstück},\ p, q \text{ Aussagen eines geeigneten Logikkalküls (PK)}$$

Hier werden die Operationen (der Datentypen) durch ihre Vor– und Nachbedingungen axiomatisch spezifiziert (= „abstrakte Prozedur“, die nur durch ihre Eigenschaften gegeben ist, nicht programmiert). Zum Beispiel:

$$\{\text{true}\}\,\text{push}(s,x)\{s \neq \Lambda \wedge \text{top}(s) = x\}$$
$$\{s = s'\}\,\text{push}(s,x);\text{pop}(s)\{s = s'\}$$

**Vorteil:**

- guter Abstraktionsgrad
- erfüllt Anforderungen an Spezifikation

**Nachteil:** Datentypen sind Algebren, imperative Methoden sind dafür nicht adäquat. Imperative Methoden sind z. B. für Datenbanken sinnvoller einsetzbar, da dort Speichertransformationen im Vordergrund stehen.

2. **Applikative Methoden**

(a) **exemplarische applikative Spezifikation**

Hier kommen die gleichen Konzepte wie bei der denotationalen Semantik zum Tragen: Beschreibung eines mathematischen Modells für Datenelemente und –operationen in üblicher mathematischer Notation. Z. B. ist ein Keller eine endliche Folge (von Datenelementen):

$$\text{stack} = \text{char}^* \;\; \text{(Zeichenkeller)} \;\; = \{(z_1, z_2, \ldots, z_n) | n \in I\!N,\; z_i \in \text{char}\} \cup \{\epsilon\},\; \Lambda = \epsilon = ()$$

$$\begin{aligned}
\text{push}((z_1, z_2, \ldots, z_n), x) &= (x, z_1, z_2, \ldots, z_n) \\
\text{pop}((z_1, z_2, \ldots, z_n)) &= \begin{cases} (z_2, \ldots, z_n), & \text{falls } n > 0 \\ \text{undefiniert} & \text{falls } n = 0 \end{cases} \\
\text{top}((z_1, \ldots, z_n)) &= \begin{cases} z_1, & \text{falls } n > 0 \\ \text{undefiniert}, & \text{falls } n = 0 \end{cases}
\end{aligned}$$

**Vorteile:**

- gute Abstraktion
- gute Verständlichkeit

**Nachteile:**

- Probleme bei automatischen Entwurfshilfen
- Äquivalenz zu anderen Modellen ist oft unklar, z.B. ist die Frage, ob ein anderes Modell zum selben abstrakten Datentyp gehört, oft nicht zu beantworten.

Daher ist dieser Ansatz für die Spezifikation von *monomorphen* Datentypen unter Umständen noch brauchbar, bei *polymorphen* abstrakten Datentypen aber ungeeignet.

(b) **axiomatische applikative Spezifikation**

Die gewünschten Eigenschaften des zu spezifizierenden abstrakten Datentyps werden in der Form von Axiomen einer geeigneten logischen Sprache aufgeschrieben, z. B.:

Ein Kellerspeicher über einer Menge char ist eine Menge stack mit folgenden Eigenschaften:

i.

$$\Lambda \in \text{stack}$$
$$\forall s \forall x (s \in \text{stack} \wedge\, x \in \text{char} \rightarrow \text{push}(s,x) \in \text{stack})$$
$$\forall s (s \in \text{stack} \wedge\, s \neq \Lambda \rightarrow \text{pop}(s) \in \text{stack})$$
$$\forall s (s \in \text{stack} \wedge\, s \neq \Lambda \rightarrow \text{top}(s) \in \text{char})$$
$$\forall s (s \in \text{stack} \rightarrow \text{empty}(s) \in \{\text{true}, \text{false}\})$$

ii.

$$\forall s \forall x (s \in \text{stack} \wedge\ x \in \text{char} \rightarrow \text{push}(s, x) \neq \Lambda)$$
$$\forall s \forall x \forall y (s \in \text{stack} \wedge\ x, y \in \text{char} \rightarrow (\text{push}(s, x) = \text{push}(s, y) \rightarrow x = y))$$
$$\forall s_1 \forall s_2 \forall x (s_1, s_2 \in \text{stack} \wedge\ x \in \text{char} \rightarrow (\text{push}(s_1, x) = \text{push}(s_2, x) \rightarrow s_1 = s_2))$$

iii.

$$\forall s \forall x (s \in \text{stack} \wedge\ x \in \text{char} \rightarrow \text{top}(\text{push}(s, x)) = x \wedge\ \text{pop}(\text{push}(s, x)) = s)$$
$$\forall s \forall x (s \in \text{stack} \wedge\ x \in \text{char} \rightarrow \text{empty}(\text{push}(s, x)) = \text{false})$$
$$\text{empty}(\Lambda) = \text{true}$$

iv.

$$\forall \mathcal{M} (\mathcal{M} \subseteq \text{stack}\ \wedge\ \Lambda \in \mathcal{M}\ \wedge \forall s \forall x (s \in \mathcal{M} \wedge\ x \in \text{char} \rightarrow \text{push}(s, x) \in \mathcal{M}) \rightarrow \mathcal{M} = \text{stack})$$

**Bemerkung**
Man vergleiche diese Axiome mit den PEANO–Axiomen für natürliche Zahlen bzw. mit den verallgemeinerten PEANO–Axiomen in der Definition einer PEANO–Algebra.

Eine solche axiomatische Spezifikation ist sehr abstrakt. Sie sagt absolut nichts über die Art der Implementation aus. Daher ist die Anwendung von automatischen Entwurfshilfen problematisch. Algebraisch gesprochen: *Wie* ein Modell beschaffen sein könnte, ist im allgemeinen aus den Axiomen *nicht* abzulesen. Es werden – wie beabsichtigt – nur Eigenschaften beschrieben. Dabei tauchen zwei wichtige Fragen auf:

i. Ist die Spezifikation *widerspruchsfrei* (konsistent), d. h. existiert überhaupt ein Modell?
ii. Ist die Spezifikation *vollständig*? Das heißt, beschreibt sie den abstrakten Datentyp genau genug? Lassen sich alle gewünschten Eigenschaften ableiten? Oder gibt es noch Modelle (die zwar alle Axiome erfüllen), die sich aber anders als gewünscht verhalten?
(Ist $s \neq \Lambda\ \text{push}(\text{pop}(s), \text{top}(s)) = s$ aus den obigen Axiomen ableitbar?)

Im allgemeinen ist es sehr schwierig, diese Fragen zu beantworten. Speziell die Frage nach der Existenz von Modellen. Deshalb ist es zweckmäßig, die Ausdrucksmittel für die Axiome zu beschränken.

Für den Prädikatenkalkül sind *Termgleichungen* die einfachsten prädikativen Ausdrücke. Beschränkt man sich auf Axiome in der Form *generalisierter Gleichungen*

$$\forall x_1 \forall x_2 \ldots \forall x_n (t_1 = t_2),$$

wobei $t_1, t_2$ Terme in den Variablen $x_1, x_2, \ldots, x_n$ sind, so weiß man aus der Universellen Algebra, daß eine solche Spezifikation *stets* Modelle besitzt, nämlich die entsprechende Klasse gleichungsdefinierter Algebren. (vgl. Vorlesung „Algebraische Grundlagen der Informatik“). Deshalb beschäftigen wir uns im folgenden mit der *axiomatischen applikativen Spezifikation* in der Spezialform der *algebraischen Gleichungsspezifikation*. Viele Ergebnisse lassen sich z. B. auch auf sogenannte „bedingte Gleichungen“

$$t_{11} = t_{21} \wedge t_{12} = t_{22} \wedge \ldots \wedge t_{1n} = t_{2n} \rightarrow t = t'$$

verallgemeinern.

# Kapitel 2

# Terme, Gleichungen, Gleichungsspezifikationen

$S$ sei eine endliche Menge. Eine $S$–sortige *Signatur* $\Sigma$ ist eine Familie von Mengen

$$\Sigma = (\Sigma_{w,s})_{w \in S^*, s \in S}.$$

Die Elemente von $S$ heißen in diesem Zusammenhang *Sorten*, die von $\Sigma_{w,s}$ *Operatoren* (oder Operationssymbole) mit *Eingang* $w$, *Ausgang* oder *Zielsorte* $s$ und der *Arität* $(w, s)$.

**Bemerkung**

Manchmal wird die Signatur auch als Tripel $\Sigma = (S, \Omega, \alpha)$ gegeben mit $\Omega$ als Menge der Operatoren

$$\Omega = \bigcup_{w \in S^*, s \in S} \Sigma_{w,s}$$

und der Aritätsfunktion $\alpha : \Omega \mapsto S^* \times S$ . Dann ist

$$\Sigma_{w,s} = \{\omega \mid \omega \in \Omega \wedge \alpha(\omega) = (w, s)\}.$$

Man beachte, daß bei Anwendungen fast alle $\Sigma_{w,s}$ leer sind.
Wir schreiben künftig

$$\omega \in \Sigma \qquad \text{für : es existiert ein } (w, s) \text{ mit } \omega \in \Sigma_{w,s}.$$

Eine $\Sigma$–*Algebra* $\mathcal{A}$ ist ein Paar

$$\mathcal{A} = ((A_s)_{s \in S}, (f_\sigma)_{\sigma \in \Sigma}),$$

wobei für $\sigma \in \Sigma_{w,s}$ :

$$\begin{aligned} f_\sigma &: A^w \mapsto A_s, \\ \text{das heißt } f_\sigma &: A_{s_1} \times A_{s_2} \times \ldots \times A_{s_n} \mapsto A_s \end{aligned}$$

bei $w = s_1 s_2 \ldots s_n$.

Mit $\underline{T_\Sigma(X)}$ bezeichnen wir die $\Sigma$–*Termalgebra* über dem Variablensystem $X = (X_s)_{s \in S}$ (das heißt, in der Regel die „Standardtermalgebra“ oder eine zu ihr isomorphe übliche Termalgebra).
$T_\Sigma$ bezeichnet die Familie der *Grundterme* (variablenfreie Terme) , $\underline{T_\Sigma}$ die entsprechende Term–*algebra*.

$$T_\Sigma =_{df} T_\Sigma(\emptyset)$$

Ein $\Sigma$–*Gleichungssystem* über $X$ ist eine Familie $E = (E_s)_{s \in S}$ von Mengen $E_s$ von *Gleichungen* der Sorte $s$:

$$(t, t') \in E_s, t, t' \in T_\Sigma(X)_s, \quad \text{also} \quad E_s \subseteq T_\Sigma(X)_s \times T_\Sigma(X)_s.$$

Übliche Schreibweise für Gleichung $(t, t')$:

$$t = t'.$$

**Definition 2.1**

Eine (Gleichungs–)*Spezifikation* ist ein Paar

$$\underline{\underline{\text{spec}}} = (\Sigma, E),$$

wobei $\Sigma$ eine Signatur und $E$ ein $\Sigma$–Gleichungssystem ist.

**Bemerkung**

Zur Angabe von Spezifikationen benutzen wir eine einfache sich selbst erklärende Sprache.

**Beispiel**

1.

| | | | | | |
|---|---|---|---|---|---|
| stack | = | sorts | | | stack, char, bool |
| | | oprs | push | : | stack, char $\mapsto$ stack |
| | | | top | : | stack $\mapsto$ char |
| | | | pop | : | stack $\mapsto$ stack |
| | | | clear | : | $\mapsto$ stack |
| | | | empty | : | stack $\mapsto$ bool |
| | | | true | : | $\mapsto$ bool |
| | | | false | : | $\mapsto$ bool |
| | | vars | $s$ | : | stack; |
| | | | $x$ | : | char |
| | | eqns | | | empty(push$(s, x)$) = false |
| | | | | | empty(clear) = true |
| | | | | | top(push$(s, x)$) $= x$ |
| | | | | | pop(push$(s, x)$) $= s$ |
| endstack | | | | | |

2.

| | | | | | |
|---|---|---|---|---|---|
| group | = | sorts | | | $G$ |
| | | oprs | $*$ | : | $G, G \mapsto G$ |
| | | | $e$ | : | $\mapsto G$ |
| | | | $i$ | : | $G \mapsto G$ |
| | | vars | $x, y, z$ | : | $G$ |
| | | eqns | | | $(x * y) * z = x * (y * z)$ |
| | | | | | $e * x = x$ |
| | | | | | $i(x) * x = e$ |
| endgroup | | | | | |

Eine $\Sigma$–Algebra $\mathcal{A}$, die das Gleichungssystem $E$ erfüllt, heißt *Modell* von $E$ oder $(\Sigma, E)$–*Algebra* beziehungsweise spec–*Algebra*. Damit bestimmt eine Spezifikation $\underline{\underline{\text{spec}}}$ eine ganze Klasse von Algebren $\underline{\underline{\text{Alg}}}_{\underline{\underline{\text{spec}}}}$ bzw. $\underline{\underline{\text{Alg}}}_{\Sigma, E}$.

Zum obigen Beispiel:

Mit der 1. Spezifikation wird die Klasse aller $\underline{\underline{\text{stack}}}$–Algebren beschrieben, mit der 2. Spezifikation die Klasse $\underline{\underline{\text{Alg}}}_{\underline{\underline{\text{group}}}}$. Am 2. Beispiel erkennt man, daß mit einer Gleichungsspezifikation auf direkte Weise in der Regel nur *polymorphe* Datentypen spezifiziert werden können. Die Klasse aller Gruppen beinhaltet sehr verschiedenartige nichtisomorphe Gruppen. Auch das 1. Beispiel besitzt Nichtstandardmodelle („unendliche Folgen“!).

Generell gilt, daß es mit Formeln des PK1 *nicht* gelingt, *monomorphe* abstrakte Datentypen direkt zu spezifizieren. In unserem Beispiel zur axiomatischen applikativen Spezifikation (Bsp. 2.b) ist das Axiom *iv* dafür verantwortlich, Nichtstandardmodelle auszuschließen. Gleichungen sind Formeln erster Stufe, d. h. es werden im allgemeinen *zu große* Klassen von Algebren beschrieben.

**Frage:** *Wie* lassen sich trotzdem mit Hilfe dieser bequemen, einfachen Ausdrucksmittel die häufig gebrauchten *monomorphen* abstrakten Datentypen spezifizieren?

# Kapitel 3

# Initialität

Wir betrachten eine beliebige gleichungsdefinierbare Klasse von Algebren: $\underline{\underline{\mathrm{Alg}}}_{\Sigma,E}$. Eine solche Klasse kann – unabhängig von der Gestalt der Gleichungen – *niemals* ein monomorpher Datentyp sein. Denn eine $\Sigma$–Algebra mit genau einem Element in jeder Trägermenge muß natürlich alle Gleichungen erfüllen. Bei der Auswertung der Terme kommt als Resultat stets *das* Element der Trägermenge der Sorte des Terms heraus. Gleichungen sind Paare von Termen gleicher Sorte, werden hier also stets erfüllt. Damit gehört diese entartete Algebra (und alle zu ihr isomorphen) zu $\underline{\underline{\mathrm{Alg}}}_{\Sigma,E}$. Ein monomorpher abstrakter Datentyp könnte also nur aus dieser trivialen Klassen bestehen.

**Folgerung**

Zu $\underline{\underline{\mathrm{Alg}}}_{\Sigma,E}$ gehört immer die „entartete Algebra“ (einelementig in jeder Sorte). Ein monomorpher abstrakter Datentyp, der *direkt* durch Gleichungen spezifiziert wurde, würde also nur aus der Isomorphieklasse dieser trivialen Algebra bestehen.

Wenn wir einen nichttrivialen abstrakten Datentyp innerhalb von $\underline{\underline{\mathrm{Alg}}}_{\Sigma,E}$ auszeichnen wollen, brauchen wir weitere – nicht durch Gleichungen ausdrückbare – Bedingungen.

Dazu einige Vorüberlegungen:

**Definition 3.1**

Eine $\Sigma$–Algebra $\mathcal{A}$ heißt *minimal* genau dann, wenn sie keine echten Unteralgebren besitzt.

**Lemma**

Jede $\Sigma$–Algebra besitzt genau eine minimale $\Sigma$–Unteralgebra.

**Beweis**

Der Durchschnitt beliebig vieler $\Sigma$–Unteralgebren ist wieder eine $\Sigma$–Unteralgebra. Daraus folgt die Behauptung.

q.e.d.

**Definition 3.2**

Sei $\mathcal{A}$ eine $\Sigma$–Algebra. Ein Element $a \in A$ heißt *Termelement* genau dann, wenn es einen Grundterm $t \in T_\Sigma$ gibt mit $\text{wert}_\mathcal{A}(t) = a$.
(Man beachte, daß bei der Auswertung variablenfreier Terme $\mathcal{A}$–Belegungen keine Rolle spielen; deshalb diese vereinfachte Schreibweise. Wenn $h_\mathcal{A}$ den eindeutig existierenden Homomorphismus von $T_\Sigma$ in $\mathcal{A}$ bezeichnet, dannn ist $\text{wert}_\mathcal{A} = h_\mathcal{A}$.)

**Lemma**

Die Termelemente von $\mathcal{A}$ bilden die minimale Unteralgebra $\mathcal{A}^0$ von $\mathcal{A}$.

**Beweis**

1. Die Menge der Termelemente $A^T$ ist abgeschlossen gegenüber allen Operationen aus $\mathcal{A}$; $\mathcal{A}^T$ ist also Unteralgebra.

   Sei $\sigma \in \Sigma_{w,s}$ mit $w = s_1 s_2 \ldots s_n$ und sei jeweils $a_i \in A^T_{s_i}$. D. h. es existieren jeweils $t_i \in (T_\Sigma)_{s_i}$ mit $\text{wert}_\mathcal{A}(t_i) = a_i$. Dann ist

   $$\begin{aligned} \text{wert}_\mathcal{A}(\sigma t_1 t_2 \ldots t_n) &= f_\sigma(\text{wert}_\mathcal{A}(t_1), \ldots, \text{wert}_\mathcal{A}(t_n)) \\ &= f_\sigma(a_1, a_2, \ldots, a_n). \end{aligned}$$

   Damit ist $f_\sigma(a_1, a_2, \ldots, a_n)$ auch ein Termelement, also in $A^T_s$.

2. Es muß sein: $A^T \subseteq A^0$:

   Sei $\sigma \in \Sigma_s$ nullstelliger Operator. Dann $\sigma \in T_\Sigma$ und $f^\mathcal{A}_\sigma \in A^T$ Konstante in $\mathcal{A}$. Alle Konstanten sind Elemente in allen $\Sigma$–Unteralgebren. Daraus folgt $f^\mathcal{A}_\sigma \in A^0$. Sei $\omega \in \Sigma_{w,s}$ und $w = s_1 s_2 \ldots s_n$. Wenn $a_1, a_2, \ldots, a_n$ Termelemente der Sorten $s_1, s_2, \ldots, s_n$ in $A^0$ sind mit $a_i = \text{wert}_\mathcal{A}(t_i)$, so muß auch $f^\mathcal{A}_\sigma(a_1, \ldots, a_n) \in A^0$ sein (Unteralgebraeigenschaft). $f^\mathcal{A}_\omega(a_1, \ldots, a_n)$ ist aber auch Termelement, weil $\text{wert}_\mathcal{A}(\omega t_1 \ldots t_n) = f^\mathcal{A}_\omega(a_1, \ldots, a_n)$. Also $A^T \subseteq A^0$.

q.e.d.

Zurück zur gesuchten Bedingung, um einen nichttrivialen abstrakten Datentyp auszuzeichnen. Bekanntester (zuerst axiomatisch charakterisierter monomorpher abstrakter Datentyp) ist die Menge Nat mit 0 und der Nachfolgerbildung. Die PEANO–Axiome legen Nat bis auf Isomorphie fest:

1. $0 \in \text{Nat}$
2. $\forall n(n \in \text{Nat} \to \text{suc}(n) \in \text{Nat})$
3. $\forall n(n \in \text{Nat} \to \text{suc}(n) \neq 0)$
4. $\forall n \forall m(n, m \in \text{Nat} \wedge \text{suc}(n) = \text{suc}(m) \to n = m)$
5. Induktionsaxiom:

   Jede Teilmenge von Nat, die 0 und mit $n$ auch $\text{suc}(n)$ enthält, ist gleich Nat.

Wir können diese Axiome aus algebraischer Sichtweise umformen:

Nat ist eine Algebra der Signatur

$$\begin{array}{lll} \underline{\text{nat}} = & \underline{\text{sorts}} & \text{nat} \\ & \underline{\text{oprs}} & 0 :\mapsto \text{nat} \\ & & \text{suc} : \text{nat} \mapsto \text{nat} \end{array}$$

Damit sind die Axiome (1) und (2) beschrieben. (3) und (4) besagen, daß

$$0, \mathrm{suc}(0), \mathrm{suc}(\mathrm{suc}(0)), \ldots$$

alles verschiedene Elemente in Nat bezeichnen: es gelten *keine* nichttrivialen Gleichungen. (5) besagt, daß Nat minimale nat–Algebra ist ! Damit besteht Nat nur aus Termelementen. Anders gesagt, die Termalgebra $T_{nat}$ mit den Termen

$$0, \mathrm{suc}(0), \mathrm{suc}(\mathrm{suc}(0)), \ldots$$

ist ein Modell von nat.

Das sind – wenn man dies auf beliebige Signaturen verallgemeinert – genau die Forderungen, die aus der Definition einer $\Sigma$–PEANO–Algebra bekannt sind.

Eine $\Sigma$–Algebra, für die keine nichttrivialen Gleichungen gelten, (es fällt nichts zusammen/„no confusion“) und die minimal ist (d.h. keine Elemente zuviel enthält/„no junk“), ist $\Sigma$–PEANO–Algebra (mit leerer PEANO–Basis) beziehungsweise isomorph zu $T_\Sigma$. Wir wissen, daß zu jeder $\Sigma$–Algebra $\mathcal{A}$ genau ein $\Sigma$–Homomorphismus $h$ von $T_\Sigma$ in $\mathcal{A}$ existiert,

$$h : T_\Sigma \mapsto \mathcal{A},$$

daß $T_\Sigma$ also *initiale* Algebra in $\mathrm{Alg}_\Sigma$ ist.

Allgemein heißt eine Algebra $\mathcal{A}$ *initial* in einer Klasse Alg von Algebren genau dann, wenn zu jeder Algebra $\mathcal{B} \in$ Alg genau ein Homomorphismus $h$ von $\mathcal{A}$ in $\mathcal{B}$ existiert:

$$h : \mathcal{A} \mapsto \mathcal{B}.$$

**Satz 3.1**

Sei $\mathcal{A}$ initial in einer Klasse Alg von Algebren. Dann ist eine $\Sigma$–Algebra $\mathcal{B}$ genau dann initial in Alg, wenn $\mathcal{A} \cong \mathcal{B}$.

**Beweis**

1. Sei $\mathcal{B}$ initial in Alg. Es existieren nach Voraussetzung Homomorphismen

   $$h : \mathcal{A} \mapsto \mathcal{B} \text{ und } g : \mathcal{B} \mapsto \mathcal{A}.$$

   $g \circ h$ ist dann Homomorphismus von $\mathcal{A}$ in $\mathcal{A}$. Wegen Initialität von $\mathcal{A}$ muß sein

   $$g \circ h = id_{\mathcal{A}}.$$

   Analog $h \circ g = id_{\mathcal{B}}$. Dann $\mathcal{A} \cong \mathcal{B}$.

2. Sei nun $\mathcal{A} \cong \mathcal{B}$. Sei $i : \mathcal{A} \mapsto \mathcal{B}$ Isomorphismus.

   Sei $\mathcal{C} \in$ Alg beliebig. $\mathcal{A}$ initial in Alg, also existiert Homomorphismus

   $$h : \mathcal{A} \mapsto \mathcal{C}.$$

   Damit ist $h \circ i^{-1}$ Homomorphismus von $\mathcal{B}$ in $\mathcal{C}$. Sei $g$ irgendein Homomorphismus von $\mathcal{B}$ in $\mathcal{C}$. Dann ist $g \circ i$ Homomorphismus von $\mathcal{A}$ in $\mathcal{C}$. Wegen Initialität von $\mathcal{A}$ also $g \circ i = h$. Damit $g = h \circ i^{-1}$, d. h. $g$ ist eindeutig festgelegt. Folglich ist $\mathcal{B}$ initial.

q.e.d.

**Folgerung**

Die Klasse der initialen Algebren in einer Algebrenklasse $\underline{\underline{\text{Alg}}}$ ist eine Isomorphieklasse.

Die Forderung der Initialität eignet sich also zur Festlegung monomorpher abstrakter Datentypen.

In jeder gleichungsdefinierbaren Klasse $\underline{\underline{\text{Alg}}}_{\Sigma,E}$ gibt es eine initiale Algebra und damit auch die Isomorphieklasse der initialen Algebren. Man erhält die initiale Algebra, indem man in der Termalgebra $T_\Sigma$ jeweils die Terme identifiziert, die man mit Hilfe des gegebenen Gleichungssystems $E$ „ineinander umformen“ kann.

Das heißt, man hat die Faktoralgebra $T_\Sigma/\equiv_E$ von $T_\Sigma$ nach der vom Gleichungssystem $E$ induzierten Kongruenz $\equiv_E$ zu bilden:
Sei $E$ ein $\Sigma$–Gleichungssystem über $X$, dann bezeichnen wir mit $\mathcal{S}(E)$ den *stabilen Abschluß* von $E$, d. h. das Gleichungssystem, das aus $E$ durch Einsetzung hervorgeht. Eine Gleichung $t = t'$ gehört zu $\mathcal{S}(E)$ genau dann, wenn es eine Einsetzungsabbildung

$$s : X \mapsto T_\Sigma(X)$$

und eine Gleichung $t_1 = t_2$ aus $E$ gibt, so daß $t = s(t_1)$ $(= s^*(t_1) = \text{sub}(t_1, s))$ und $t' = s(t_2)$ ist.
Die *Kongruenz modulo* $E$ (von $E$ erzeugte Kongruenz oder syntaktische Äquivalenz in $\underline{\underline{\text{Alg}}}_{\Sigma,E}$) $\equiv_E$ ist nun die kleinste $\Sigma$–Kongruenz, die $\mathcal{S}(E)$ umfaßt:

$$\equiv_E =_{df} \bigcap\{\, R \mid R \text{ ist } \Sigma\text{–Kongruenz über } \underline{T_\Sigma(X)} \text{ und } \mathcal{S}(E) \subseteq R\}.$$

Man erhält sie als transitiven, kompatiblen, reflexiven und symmetrischen Abschluß von $\mathcal{S}(E)$.

$$\equiv_E = \ (\text{kom}(\mathcal{S}(E) \cup \mathcal{S}(E)^{-1}))^*.$$

Dabei ist der *kompatible Abschluß* $\text{kom}(R)$ einer binären Relation $R$ in $\mathcal{A}$ erklärt wie folgt:

1. $R \subseteq \text{kom}(R)$.
2. $\forall \sigma \in \Sigma_{w,s}$ mit $w = s_1, s_2, \ldots, s_n$:
   Wenn für $1 \leq i \leq n$ gilt $(a_i, b_i) \in \text{kom}(R)$, so gilt auch $(f_\sigma(a_1, \ldots, a_n), f_\sigma(b_1, \ldots, b_n)) \in \text{kom}(R)$.
3. $(a, b) \in \text{kom}(R)$ nur auf Grund von 1. oder 2.

Zur *Faktoralgebra* $\underline{T_\Sigma}/\equiv_E$ bzw. $\underline{T_{\Sigma,E}}$:

- Elemente sind die Äquivalenzklassen (Kongruenzklassen) $[t]_{\equiv_E}$ der Terme bezüglich $\equiv_E$
- Operationen $f'_\sigma$ werden von den Operationen in $\underline{T_\Sigma}$ induziert:

$$f'_\sigma([t_1], \ldots, [t_n]) = [f(t_1, \ldots, t_n)]$$

Es gilt der

**Satz 3.2**

Die Faktoralgebra $\underline{T_\Sigma}/\equiv_E$ ist initial in der Klasse aller $(\Sigma, E)$–Algebren, d. h. zu jeder $(\Sigma, E)$–Algebra $\mathcal{A}$ gibt es genau einen Homomorphismus

$$h : \underline{T_\Sigma}/\equiv_E \mapsto \mathcal{A}.$$

Damit hat man die Existenz von initialen Modellen zu jedem beliebigen Gleichungssystem $E$. Das rechtfertigt die folgende

**Definition 3.3**

Es sei $\underline{\underline{\text{spec}}} = (\Sigma, E)$ eine Spezifikation. Dann heißt die Isomorphieklasse der initialen Algebra in $\underline{\underline{\text{Alg}}}_{\underline{\underline{\text{spec}}}}$ der durch $\underline{\underline{\text{spec}}}$ gegebenene *abstrakte Datentyp* $\underline{\text{ADT}}_{\underline{\underline{\text{spec}}}}$ .

**Bemerkung**

1. Zu $\underline{\text{ADT}}_{\underline{\underline{\text{spec}}}}$ gehört immer
$$\underline{T_{\Sigma,E}} \ (= \underline{T_\Sigma}/\equiv_E)$$
2. Deshalb spricht man auch von der *initialen Semantik* der abstrakten Datentypen.

**Beispiel**

| | | | | |
|---|---|---|---|---|
| $\underline{\underline{\text{nat}}}$ | = | $\underline{\text{sorts}}$ | nat; | |
| | | $\underline{\text{oprs}}$ | $0$ | : nat; |
| | | | suc | : nat $\mapsto$ nat; |
| | | | $+$ | : nat, nat $\mapsto$ nat; |
| | | $\underline{\text{vars}}$ | $n, m$ | : nat; |
| | | $\underline{\text{eqns}}$ | | $n + 0 = n$ |
| | | | | $n + \text{suc}(m) = \text{suc}(n + m)$ |
| $\underline{\text{end}}$ $\underline{\underline{\text{nat}}}$ | | | | |

Der durch $\underline{\underline{\text{nat}}}$ gegebene abstrakte Datentyp $\underline{\text{ADT}}_{\underline{\underline{\text{nat}}}}$ ist die Isomorphieklasse von $\underline{T_{\underline{\underline{\text{nat}}}}}$.

Kongruenzklassen von $T_{\underline{\underline{\text{nat}}}}$

$$\begin{array}{lcl} [0] & = & \{0, 0+0, 0+(0+0), \ldots\} \\ [\text{suc}(0)] & = & \{\text{suc}(0), \text{suc}(0)+0, 0+\text{suc}(0), \text{suc}(0+0), \text{suc}((0+0)+0), \ldots\} \\ [\text{suc}(\text{suc}(0))] & = & \{\text{suc}(\text{suc}(0)), \text{suc}(0)+\text{suc}(0), \ldots\} \end{array}$$

Offensichtlich ist $\underline{\text{ADT}}_{\underline{\underline{\text{nat}}}}$ die Isomorphieklasse der Algebra der natürlichen Zahlen mit 0, Nachfolgerbildung und Addition.

# Kapitel 4

# Gleichungskalkül und Induktion

Mit den Gleichungen $E$ aus einer Spezifikation haben wir die Kongruenzrelation $\equiv_E$ gegeben. Sie kann rein syntaktisch charakterisiert werden:

transitiver, kompatibler, reflexiver, symmetrischer und stabiler Abschluß von $E$

und damit hat man einen Gleichungskalkül:

alle Gleichungen, die zu $\equiv_E$ gehören, können *abgeleitet* werden. (Deswegen spricht man von „syntaktischer Äquivalenz“ in $\underline{\underline{\mathrm{Alg}}}_{\Sigma,E}$) Dieser Gleichungskalkül ist widerspruchsfrei, d. h. alle Gleichungen aus $\equiv_E$ gelten in allen Algebren von $\underline{\underline{\mathrm{Alg}}}_{\Sigma,E}$.

Umgekehrt kann man nach den Gleichungen fragen, die in *allen* Algebren aus $\underline{\underline{\mathrm{Alg}}}_{\Sigma,E}$ gelten:

das ist die *Theorie* (Menge der gültigen Sätze/Gleichungen) dieser Klasse
oder die *semantische Äquivalenz*.

Man kann zeigen (vergleiche Vorlesung „Algebraische Grundlagen der Informatik“), daß die semantische Äquivalenz mit der syntaktischen zusammenfällt.

Der Gleichungskalkül ist also *vollständig*!

Sei $\mathrm{MOD}(E)$ die Modellklasse von $E$, d. h.

$$\mathrm{MOD}(E) = \underline{\underline{\mathrm{Alg}}}_{\Sigma,E},$$

und bezeichnen weiter $\mathrm{TH}(\mathcal{C})$ für eine Algebrenklasse $\mathcal{C}$ der Signatur $\Sigma$ die Menge aller Gleichungen, die in allen Algebren von $\mathcal{C}$ gelten – die *Theorie* von $\mathcal{C}$. Dann gilt der

**Satz 4.1**

$$\mathrm{TH}(\mathrm{MOD}(E)) \ = \equiv_E$$

Das heißt speziell, daß alle abgeleiteten Gleichungen einer Spezifikation in allen Modellen dieser Spezifikation gelten. In initialen Modellen können natürlich wesentlich mehr Gleichungen gelten !

Betrachten wir das frühere **Beispiel** $\underline{\underline{\mathrm{nat}}}$:

In $\underline{\mathrm{ADT}}_{\underline{\underline{\mathrm{nat}}}}$ (der Klasse der initialen Modelle!) gelten z. B. folgende Gleichungen:

$$\begin{aligned} m+n &= n+m \\ (k+m)+n &= k+(m+n). \end{aligned}$$

Es gelingt aber nicht, diese aus $\underline{\underline{\mathrm{nat}}}$ im Gleichungskalkül abzuleiten.
Wir betrachten als konkretes Beispiel für diese Gleichung in der Algebra mit den Elementen

$$0, \mathrm{suc}(0), \mathrm{suc}(\mathrm{suc}(0)), \mathrm{suc}(\mathrm{suc}(\mathrm{suc}(0))), \ldots$$

die Gleichung:

$$\mathrm{suc}(0) + \mathrm{suc}(\mathrm{suc}(0)) \;=\; \mathrm{suc}(\mathrm{suc}(0)) + \mathrm{suc}(0).$$

Diese Gleichung allerdings ist aus den Gleichungen von $E$ ableitbar :

$$\begin{array}{lrcl} (1) & n + 0 & = & n \\ (2) & n + \mathrm{suc}(m) & = & \mathrm{suc}(n + m). \end{array}$$

$$\begin{array}{rcl} \mathrm{suc}(0) + \mathrm{suc}(\mathrm{suc}(0)) & =_2 & \mathrm{suc}(\mathrm{suc}(0) + \mathrm{suc}(0)) \\ & =_2 & \mathrm{suc}(\mathrm{suc}(\mathrm{suc}(0) + 0)) \\ & =_1 & \mathrm{suc}(\mathrm{suc}(\mathrm{suc}(0))) \\ & =_2 & \mathrm{suc}(\mathrm{suc}(\mathrm{suc}(0)) + 0) \\ & =_2 & \mathrm{suc}(\mathrm{suc}(0)) + \mathrm{suc}(0). \end{array}$$

Man kann sich auch leicht überlegen, daß auch alle anderen „konkreten“ Gleichungen, d. h. Gleichungen, in denen für die Variablen nur Grundterme (variablenfreie Terme) eingesetzt werden (= „konstante Gleichungen“), herleitbar sind.

Allgemein gilt, daß alle *konstanten Gleichungen*, die in initialen Modellen gelten, stets herleitbar sind. Denn sie müssen in *allen* Modellen gelten (wegen der Initialität und weil Homomorphismen Konstanten auf Konstanten abbilden).

Damit ist der Gleichungskalkül für die „konstante Theorie“ der initialen Algebren ebenfalls widerspruchsfrei und vollständig.

Allerdings gilt dies **nicht** für die „ganze“ Theorie initialer Algebren. Durch Anwendung eines *Induktionsbeweises* lassen sich sehr viele Gleichungen, die in initialen Algebren gelten, beweisen. Dazu ist zu überlegen, daß Gleichungen, die in minimalen Algebren gelten (bestehen nur aus Termelementen), auch in initialen gelten.

Eine Gleichung $t_1 = t_2$ gehört zur Theorie der minimalen Algebren genau dann, wenn für jede minimale Algebra $\mathcal{M}$ und jede $\mathcal{M}$–Belegung $h_{\mathcal{M}}$ von $X$ gilt:

$$h^*_{\mathcal{M}}(t_1) = h^*_{\mathcal{M}}(t_2).$$

Jede minimale Algebra $\mathcal{M}$ erhält man (Termelemente!) als volles Bild von $\underline{T_\Sigma}$:

$$\mathcal{M} = h_{\mathcal{M}}(\underline{T_\Sigma}).$$

Damit gilt eine Gleichung $t_1 = t_2$ in allen minimalen Algebren schon dann, wenn alle Einsetzungen

$$s : \; X \mapsto \underline{T_\Sigma}$$

von Grundtermen für Variablen die Gleichung erfüllen:

$$s^*(t_1) \equiv_E s^*(t_2)$$

(bzw. in der von $E$ induzierten Kongruenz liegen).

$$\underbrace{s^*(t_1)}_{\in T_\Sigma} \equiv_E \underbrace{s^*(t_2)}_{\in T_\Sigma}$$

Damit erhält man ein *Induktionsbeweisschema* für die Gültigkeit von Gleichungen in minimalen (und damit in initialen) Algebren:

Gilt für alle Grundterm–Einsetzungen

$$\begin{array}{lcl} s: X & \mapsto & T_\Sigma \\ s^*(t_1) & \equiv_E & s^*(t_2), \end{array}$$

so ist

$$t_1 = t_2.$$

in allen initialen $(\Sigma, E)$–Algebren gültig.

**Beispiel**

Kommutativgesetz in initialen $\underline{\underline{\text{nat}}}$–Algebren, also in $\underline{\text{ADT}}_{\underline{\underline{\text{nat}}}}$:

$$m + n = n + m.$$

Die Grundterme der Signatur von $\underline{\underline{\text{nat}}}$ werden aus $0, \text{suc}, \text{und } +$ gebildet. Die Grundterme

$$suc^n(0), \quad \text{d. h.} \quad 0,\ \text{suc}(0),\ \text{suc}(\text{suc}(0)),\ \ldots,$$

bezeichnen wir als suc–Terme.

1. Jeder Grundterm ist einem suc–Term äquivalent (im Sinne der von $\underline{\underline{\text{nat}}}$ erzeugten Kongruenz):
   Sei $t$ ein Grundterm. Falls $t$ kein „+“ enthält, so ist $t$ selbst suc–Term. Sonst hat $t$ die Gestalt:

   $$t' + \text{suc}^n(0).$$

   Für $n = 0: \ t' + 0 \equiv_E t'$ .
   Für $n \neq 0$ gilt

   $$\begin{array}{lcll} t' + \text{suc}^n(0) & = & t' + \text{suc}(\text{suc}^{n-1}(0)) & \equiv_E \quad \text{suc}(t' + \text{suc}^{n-1}(0)) \\ & & & \vdots \quad \text{n-1 Schritte} \\ & & & \equiv_E \quad \text{suc}^n(t' + 0) \\ & & & \equiv_E \quad \text{suc}^n(t') \end{array}$$

   Damit ist $t \equiv_E \text{suc}^n(t')$ mit einem „ + “ weniger. Vollständige Induktion über die Anzahl der Vorkommen von „ + “ in $t$ ergibt:
   Es existiert ein $\overline{t}$ mit $\overline{t}$ ist suc–Term und $t \equiv_E \overline{t}$.

2. $m + n = n + m$
   wird bewiesen durch Anwendung des Induktionsbeweisschemas:
   Betrachte beliebige Grundterm–Einsetzung $s$ mit $s(m) = t_1, \ s(n) = t_2, \ t_1, t_2 \in T_\Sigma$.
   Zu zeigen:

   $$t_1 + t_2 \equiv_E t_2 + t_1.$$

   Dazu seien $\overline{t_1}, \overline{t_2}$ suc–Terme mit $t_1 \equiv_E \overline{t_1}, t_2 \equiv_E \overline{t_2}$.

   $$\begin{array}{lcl} t_1 + t_2 \equiv_E \overline{t_1} + \overline{t_2} & = & \text{suc}^i(0) + \text{suc}^k(0) \\ & = & \text{suc}^i(0) + \text{suc}(\text{suc}^{k-1}(0)) \\ & \equiv_E & \text{suc}(\text{suc}^i(0) + \text{suc}^{k-1}(0)) \\ & \vdots & \text{k-1 Schritte} \\ & \equiv_E & \text{suc}^k(\text{suc}^i(0) + 0) \end{array}$$

$$
\begin{array}{rl}
\equiv_E & \mathrm{suc}^k(\mathrm{suc}^i(0)) \\
\equiv_E & \mathrm{suc}^i(\mathrm{suc}^k(0)) \\
\equiv_E & \mathrm{suc}^i(\mathrm{suc}^k(0)+0) \\
\vdots & \text{analog wie oben} \\
\equiv_E & \mathrm{suc}^k(0)+\mathrm{suc}^i(0) \\
= & \overline{t_2}+\overline{t_1}.
\end{array}
$$

# Kapitel 5

# Erweiterung von Gleichungsspezifikationen

In der Praxis möchte man umfangreichere Datentypen nicht auf einmal und vielleicht jedesmal neu spezifizieren, sonder möchte Standarddatentypen wie $\underline{\underline{\text{nat}}}, \underline{\underline{\text{bool}}}, \ldots$ und bereits spezifizierte Datentypen als Bausteine verwenden können. Eine einfache Form des Aufbaus der abstrakten Datentypen ist die *Erweiterung* vorhandener um neue Sorten, neue Operatoren, neue Gleichungen.

Dazu definieren wir

**Definition 5.1**

Es sei $\underline{\underline{\text{spec}}} = (\Sigma, E)$ eine Spezifikation mit einer $S$–sortigen Signatur $\Sigma$. Dann heißt die Spezifikation $\underline{\underline{\text{spec}}} = (\Sigma', E')$ eine *Kombination* aus $\underline{\underline{\text{spec}}}$ und der *Ergänzung* $(S_0, \Sigma_0, E_0)$, wenn $S_0$ eine Menge (neuer) Sorten, $\Sigma_0$ eine $(S \cup S_0)$–sortige Signatur und $E_0$ eine Menge von $(\Sigma \dot\cup \Sigma_0)$–Gleichungen ist, und es gilt

$$\Sigma' = \Sigma \dot\cup \Sigma_0, \ E' = E \dot\cup E_0.$$

$S_0, \Sigma_0, E_0$ können auch leer sein. ( $\dot\cup$ bezeichnet die disjunkte Vereinigung.)

Man schreibt einfach

$$\underline{\underline{\text{spec}}}' = \underline{\underline{\text{spec}}} + (S_0, \Sigma_0, E_0).$$

Man beachte, daß die Ergänzung $(S_0, \Sigma_0, E_0)$ für sich keine Spezifikation sein muß.

In der Regel möchte man, daß bei einer solchen Kombination durch die Hinzunahme neuer Sorten, Operatoren oder Gleichungen die Eigenschaften und die Anzahl der ursprünglichen Elemente *nicht* verändert werden.
Bei einer Kombination aufbauend auf z. B. $\underline{\underline{\text{nat}}}$ durfen zur Sorte nat nicht plötzlich neue „Zahlen“ hinzukommen oder gewisse „Zahlen“ identifiziert werden.
Dies kann aber durch die initiale Semantik durchaus passieren!
Zunächst einige Beispiele:

**Beispiel**

Betrachte die Spezifikation bool:

bool
- sorts: bool;
- oprs: true, false :↦ bool
  not : bool ↦ bool
  , or, implies : bool, bool ↦ bool
- vars: $p, q$ : bool
- eqns: not(true) = false
  not(false) = true
  not(not($p$)) = $p$
  (true, $p$) = $p$
  (false, $p$) = false
  ($p, q$) = ($q, p$)
  or($p, q$) = not((not($p$), not($q$)))
  implies($p, q$) = $or$(not($p$), $q$)

end bool

und deren „Erweiterung“

ternary = bool +
- oprs: maybe :↦ bool
- eqns: not(maybe) = maybe
  (true, maybe) = maybe
  (false, maybe) = false
  or(true, maybe) = true

end ternary.

Was passiert nun ? Zu $\mathrm{ADT}_{\mathrm{bool}}$ gehört $T_{\mathrm{bool}}$ : 2 Elemente (Kongruenzklassen) [true], [false].
In $T_{\mathrm{ternary}}$ sind aber mindestens 3 Elemente [true], [false], [maybe]. Es kommen – möglicherweise unbeabsichtigt – unendlich viele weitere hinzu:

- [or(maybe, maybe)],
- [or(maybe, or(maybe, maybe)],
- [or(maybe, or(maybe, or(maybe, maybe)))],
- ...

Um dies zu verhindern, hätte die Gleichung

$$\mathrm{or}(\mathrm{maybe}, \mathrm{maybe}) = \mathrm{maybe}$$

oder eine entsprechende andere nicht „vergessen“ werden dürfen.

- Absicht: 1 weiterer neuer Wahrheitswert
- Initiale Semantik und ein Versehen erzeugen unendlich viele neue boolesche Werte.

**Beispiel**

Sei nat die Spezifikation wie im vorherigen Abschnitt, bool wie oben. Betrachte

$$
\begin{array}{lll}
\underline{\underline{\text{order}}} = \underline{\underline{\text{bool}}} + \underline{\underline{\text{nat}}} + & & \\
 & \underline{\text{oprs}} & \leq\colon \text{nat}, \text{nat} \mapsto \text{bool} \\
 & \underline{\text{vars}} & m, n : \text{nat} \\
 & \underline{\text{eqns}} & (0 \leq n) = \text{true} \\
 & & (\text{suc}(n) \leq 0) = \text{false} \\
 & & (\text{suc}(n) \leq \text{suc}(m)) = (n \leq m) \\
\underline{\text{end}}\ \underline{\underline{\text{order}}} & &
\end{array}
$$

Hier handelt es sich um eine „Erweiterung“ von $\underline{\underline{\text{bool}}} + \underline{\underline{\text{nat}}}$, bei der *keine* neuen Elemente entstehen. Durch die neuen Gleichungen werden die „Zahlen“ $[\overline{\text{suc}^n(0)}]$ nicht verändert.

Andererseits läßt sich jeder Term (der Sorte bool) der Gestalt $(t_1 \leq t_2)$ durch die Gleichungen (aus $\underline{\underline{\text{nat}}}$) auf $(\overline{t_1} \leq \overline{t_2})$ zurückführen, wobei $\overline{t_1}, \overline{t_2}$ suc–Terme sind, und dann weiter mit den neuen Gleichungen auf true oder false. Damit entstehen auch keine neuen Elemente der Sorte bool.

**Beispiel**

$$
\begin{array}{lll}
\underline{\underline{\text{testbool}}} = \underline{\underline{\text{bool}}} + & & \\
 & \underline{\text{eqns}} & \text{implies}(\text{true}, \text{false}) = \text{true} \\
\underline{\text{end}}\ \underline{\underline{\text{testbool}}} & &
\end{array}
$$

Hier erhält man

$$[\text{true}] = [\text{false}],$$

denn man kann umformen

$$
\begin{array}{rcl}
\text{true} & = & \text{implies}(\text{true}, \text{false}) \\
 & = & \text{or}(\text{not}(\text{true}), \text{false}) \\
 & = & \text{or}(\text{false}, \text{false}) \\
 & \vdots & \\
 & = & \text{false}
\end{array}
$$

Damit ist diese Kombination *keine* „Erweiterung“ von bool.

Die Beispiele legen es nahe, festzulegen und zu untersuchen, wann man es bei einer Kombination mit einer *echten Erweiterung* zu tun hat.

**Definition 5.2**

Sei $\underline{\underline{\text{spec}}} = (\Sigma, E)$ eine Spezifikation und $\underline{\underline{\text{spec}'}} = \underline{\underline{\text{spec}}} + (S_0, \Sigma_0, E_0)$.
Das $\underline{\underline{\text{spec}}}$–*Redukt* von $\underline{\underline{T_{\text{spec}'}}}$ ist die $\Sigma$–Algebra

$$\left(\underline{\underline{T_{\text{spec}'}}}\right)_{\underline{\underline{\text{spec}}}} = \left((T_{\underline{\underline{\text{spec}'}}})_{\underline{\underline{\text{spec}}}},\ (f^*_\sigma)_{\sigma \in \Sigma}\right)$$

mit

$$(T_{\underline{\underline{\text{spec}'}}})_{\underline{\underline{\text{spec}}}} = (T_{\underline{\underline{\text{spec}'}}, s})_{s \in S}$$

(nur *die* Sorten und Operatoren der Kombination, die auch in $\underline{\underline{\text{spec}}}$ vorkommen!).

**Definition 5.3**

$\underline{\underline{\text{spec}'}}$ heißt eine *Erweiterung* von $\underline{\underline{\text{spec}}}$ genau dann, wenn

$$\left(T_{\underline{\underline{\text{spec}'}}}\right)_{\underline{\underline{\text{spec}}}} \cong T_{\underline{\underline{\text{spec}}}}.$$

**Bemerkung**

Bei obigen Beispielen trifft dies nur auf das 2. Beispiel zu !

Die Bedingung in der Definition „Erweiterung“ ist letzten Endes *semantischer* Natur (Kongruenzklassenbildung entspricht Semantik). Damit dürfte klar sein, daß es allgemein nicht gelingt, rein syntaktische Krtierien für beliebige $\underline{\underline{\text{spec}}}$ und $\underline{\underline{\text{spec}'}}$ aufzustellen, die feststellen, ob eine Erweiterung vorliegt.
Leider gilt, daß die Frage, ob $\underline{\underline{\text{spec}'}}$ eine Erweiterung von $\underline{\underline{\text{spec}}}$ ist, im allgemeinen *unentscheidbar* ist.

Also Suche nach *hinreichenden* Kriterien !

**Definition 5.4**

Sei $\underline{\underline{\text{spec}}} = (\Sigma, E)$ eine Spezifikation und $\underline{\underline{\text{spec}'}} = \underline{\underline{\text{spec}}} + (S_0, \Sigma_0, E_0)$. $h$ sei der eindeutig bestimmte Homomorphismus

$$h \ : \ T_{\underline{\underline{\text{spec}}}} \mapsto \left(T_{\underline{\underline{\text{spec}'}}}\right)_{\underline{\underline{\text{spec}}}}.$$

Dann heißt $\underline{\underline{\text{spec}'}}$

*vollständig* bezüglich $\underline{\underline{\text{spec}}}$ genau dann, wenn $h$ surjektiv ist,
und

*konsistent* bezüglich $\underline{\underline{\text{spec}}}$ genau dann, wenn $h$ injektiv ist.

**Beispiel**

Bei den obigen 3 Beispielen haben wir:

- $\underline{\underline{\text{ternary}}}$ ist bezüglich $\underline{\underline{\text{bool}}}$ konsistent, aber nicht vollständig.
- $\underline{\underline{\text{order}}}$ ist bezüglich $\underline{\underline{\text{bool}}} + \underline{\underline{\text{nat}}}$ konsistent und vollständig.
- $\underline{\underline{\text{testbool}}}$ ist bezüglich $\underline{\underline{\text{bool}}}$ vollständig, aber nicht konsistent.

Es gilt trivialerweise der

**Satz 5.1**

$\underline{\underline{\text{spec}'}}$ ist eine Erweiterung von $\underline{\underline{\text{spec}}}$ genau dann, wenn $\underline{\underline{\text{spec}'}}$ bezüglich $\underline{\underline{\text{spec}}}$ konsistent und vollständig ist.

(Zum Beweis: $h$ surjektiv und injektiv genau dann, wenn $h$ Isomorphismus ist!)

Inkonsistente Kombinationen können im Spezifikations–Entwurfsprozeß sehr leicht „passieren“:

**Beispiel**

| | | | |
|---|---|---|---|
| set(nat) | = nat + | | (natwie früher) |
| | | sorts | set |
| | | oprs | $\emptyset :\mapsto$ set |
| | | | insert : set, nat $\mapsto$ set |
| | | vars | $s$ : set |
| | | | $n, m$ : nat |
| (1) | | eqns | insert(insert$(s, n), n)$ = insert$(s, n)$ |
| (2) | | | insert$(s, n), m)$ = insert(insert$(s, m), n)$ |
| end set(nat) | | | |

Will man noch eine Lösch–Operation einführen, so kann man leicht auf folgende „Erweiterung“ kommen:

| | | | |
|---|---|---|---|
| newset(nat) | = set(nat) + | | |
| | | oprs | del : set, nat $\rightarrow$ set |
| | | vars | $s$ : set |
| | | | $n$ : nat |
| (3) | | eqns | del$(\emptyset, n) = \emptyset$ |
| (4) | | | del(insert$(s, n), n) = s$ |
| end newset(nat) | | | |

Dies funktioniert aber nicht in der Weise, wie man dies sicher beabsichtigt. Speziell läßt sich Gleichung (1) ja auch von rechts nach links verwenden ! So ergibt sich:

$$\begin{aligned} \emptyset &=_{(4)} \text{del(insert}(\emptyset, 0), 0) \\ &=_{(1)} \text{del(insert(insert}(\emptyset, 0), 0), 0) \\ &=_{(4)} \text{insert}(\emptyset, 0) \end{aligned}$$

Damit hat die Operation insert keine Wirkung mehr ! Die Sorte set besteht nur noch aus einem einzigen Element:

$$[\emptyset] \qquad !$$

Die Kombination newset(nat) ist bezüglich set(nat) inkonsistent!

**Lemma**

Falls spec$'$ = spec + $(S_0, \emptyset, \emptyset)$, so ist spec$'$ Erweiterung von spec.

– trivial –

**Lemma**

Wenn spec$'$ eine Erweiterung von spec ist, und spec$''$ eine Erweiterung von spec$'$, so ist spec$''$ auch eine Erweiterung von spec.

(Beweis: über Nacheinanderausführung von Homomorphismen)

**Definition 5.5**

Gilt $\underline{\underline{\text{spec}'}} = \underline{\underline{\text{spec}}} + (\emptyset, \Sigma_0, E_0)$, so heißt $\underline{\underline{\text{spec}'}}$ eine *Anreicherung* von $\underline{\underline{\text{spec}}}$.

**Bemerkung**

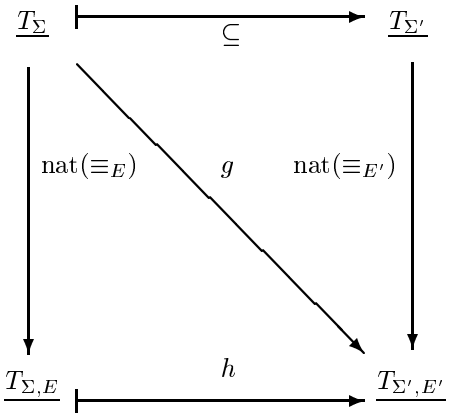
Wegen obiger Lemmata genügt es, Anreicherungen auf Kriterien für Vollständigkeit und Konsistenz zu untersuchen.

**Satz 5.2**

Sei $\underline{\underline{\text{spec}'}}$ eine Anreicherung einer Spezifikation $\underline{\underline{\text{spec}}}$, $\underline{\underline{\text{spec}'}} = \underline{\underline{\text{spec}}} + (\emptyset, \Sigma_0, E_0)$, wobei $\underline{\underline{\text{spec}'}} = (\Sigma', E')$, $\underline{\underline{\text{spec}}} = (\Sigma, E)$, $\Sigma' = \Sigma \dot\cup \Sigma_0$, $E' = E \dot\cup E_0$ (alles $S$–sortig).
Dann ist $\underline{\underline{\text{spec}'}}$

1. vollständig bezüglich $\underline{\underline{\text{spec}}}$ genau dann, wenn $\forall t'(t' \in T_{\Sigma'} \rightarrow \exists t(t \in T_\Sigma \wedge t' \equiv_{E'} t))$
   und
2. konsistent bezüglich $\underline{\underline{\text{spec}}}$ genau dann, wenn $\forall t_1 \forall t_2 (t_1, t_2 \in T_\Sigma \wedge t_1 \equiv_{E'} t_2 \rightarrow t_1 \equiv_E t_2)$.

**Beweis**




Die Homomorphismen $g, h$ und die Inklusion werden jeweils auf den $\underline{\underline{\text{spec}}}$–Redukten betrachtet, wegen Initialität sind all diese Homomorphismen eindeutig bestimmt. Da $g$ eindeutig bestimmt ist, muß auch gelten:

$$(*) \qquad h([t]_{\equiv_{E'}}) = [t]_{\equiv_{E'}} \text{ für alle } t \in T_\Sigma.$$

zu 1.) $\underline{\underline{\text{spec}'}}$ ist vollständig genau dann, wenn $h$ surjektiv ist. Dies gilt genau dann, wenn zu jeder Äquivalenzklasse $[t']_{\equiv_{E'}}$ eine Äquivalenzklasse $[t]_{\equiv_E}$ existiert, so daß

$$h([t]_{\equiv_E}) = [t']_{\equiv_{E'}}.$$

Wegen $(*)$ ist

$$[t']_{\equiv_{E'}} = h([t]_{\equiv_E}) = [t]_{\equiv_{E'}},$$

und dies gilt genau dann, wenn $t' \equiv_{E'} t$. Also existiert zu jedem $t' \in T_{\Sigma'}$ ein $t \in T_\Sigma$ mit $t' \equiv_{E'} t$.

zu 2.) $\underline{\underline{\text{spec}'}}$ konsistent bezüglich $\underline{\underline{\text{spec}}}$ genau dann, wenn $h$ injektiv ist, und dies gilt genau dann wenn

$$\forall t_1, t_2 (t_1, t_2 \in T_\Sigma) : \; h([t_1]_{\equiv_E}) = h([t_2]_{\equiv_E}) \rightarrow [t_1]_{\equiv_E} = [t_2]_{\equiv_E}$$

genau dann, wenn (wegen $(*)$)

$$[t_1]_{\equiv_{E'}} = [t_2]_{\equiv_{E'}} \rightarrow [t_1]_{\equiv_E} = [t_2]_{\equiv_E}$$

genau dann, wenn

$$t_1 \equiv_{E'} t_2 \;\rightarrow\; t_1 \equiv_E t_2.$$

q.e.d.

Das Kriterium für die Vollständigkeit läßt sich noch verschärfen: Man muß nur solche Terme beachten, bei denen die neuen Operatoren nur als Hauptverknüpfungsfaktoren auftreten.

**Definition 5.6**

Es sei $\underline{\text{spec}'} = \underline{\text{spec}} + (\emptyset, \Sigma_0, E_0)$ mit $\underline{\text{spec}'} = (\Sigma', E')$, $\underline{\underline{\text{spec}}} = (\Sigma, E)$. Ein Term $t \in T_{\Sigma'}$ heißt ein $\Sigma_0$–*Term* genau dann, wenn $t = \sigma t_1 \ldots t_n$ für ein $\sigma \in \Sigma_0$. Ein $\overline{\Sigma_0}$–Term $\sigma t_1 \ldots t_n$ heißt $\Sigma_0$–*normaler Term* genau dann, wenn $\forall i (1 \leq i \leq n) : \; t_i \in T_\Sigma$.

**Satz 5.3**

Mit den obigen Bezeichnungen gilt: $\underline{\text{spec}'}$ ist vollständig bezüglich $\underline{\underline{\text{spec}}}$ genau dann, wenn für alle $\Sigma_0$–normalen Terme $t_0 \in T_{\Sigma'}$ ein $t \in T_\Sigma$ existiert mit $t_0 \equiv_{E'} t$.

**Beweis**

($\rightarrow$) trivial

($\leftarrow$) Sei $t' \in T_{\Sigma'}$. Wir zeigen induktiv über die Anzahl der Vorkommen von Operatoren $\sigma \in \Sigma_0$ in $t'$:
es existiert stets ein $t \in T_\Sigma$ mit $t' \equiv_{E'} t$ (d. h. $\underline{\underline{\text{spec}'}}$ ist vollständig nach obigem Satz).

(IA) In $t'$ gebe es kein $\sigma \in \Sigma_0$ . Dann ist $t' \in T_\Sigma$, womit die Behauptung trivialerweise gilt.

(IV) Die Behauptung möge gelten für alle $t'' \in T_\Sigma$, die weniger als $n$ $(n > 0)$ Vorkommen von $\sigma$, $\sigma \in \Sigma_0$, enthalten.

(IS) $t'$ enthalte genau $n$ Vorkommen von $\sigma \in \Sigma_0$.
Sei $t_0$ ein innerster $\Sigma_0$–normaler Teilterm von $t'$. Nach Voraussetzung existiert ein $t_{00} \in T_\Sigma$ mit $t_0 \equiv_{E'} t_{00}$. Ersetzt man nun in $t'$ den Teilterm $t_0$ durch den Term $t_{00}$ und erhält dadurch den Term $t''$, so gilt nach dem Gleichungskalkül $t' \equiv_{E'} t''$. $t''$ enthält nur noch $n-1$ Vorkommen von Operatoren aus $\Sigma_0$. Also existiert nach (IV) ein Term $t \in T_\Sigma$ mit $t'' \equiv_{E'} t$. Damit ist auch $t' \equiv_{E'} t$.

q.e.d.

**Bemerkung**

Die vorgestellten Kriterien für Vollständigkeit und Konsistenz erfordern die Prüfung der Äquivalenz von Termen:

$$t \equiv_E t' \; ?$$

d. h., die Lösung des Wortproblems. Folglich ist die Frage im allgemeinen unentscheidbar!
Es sind weitere hinreichende Kriterien dafür notwendig oder man benötigt die Konfluenz und Nötherizität des Gleichungssystems (vgl. Termersetzungssysteme).

- Für die Vollständigkeit lassen sich relativ einfache hinreichende Kriterien finden, speziell für sogenannte kanonische Termalgebren als Modell der Basisspezifikation $\underline{\underline{\text{spec}}}$. Aus Zeitgründen werden wir das hier aber nicht behandeln.

- Für die Konsistenz sind bisher vergleichbar einfache Kriterien nicht bekannt.

# Kapitel 6

# Finale Semantik und beobachtbares Verhalten

Bei der Suche nach einer Definitionsmethode für monomorphe, abstrakte Datentypen hatte die Isomorphieklasse der initialen Algebren aus verschiedenen Gründen eine bevorzugte Stellung bekommen. Aber sie ist nicht a priori vor anderen Isomorphieklassen ausgezeichnet. Das heißt, generell sind auch andere als die initiale Semantik möglich. Nachteile der initialen Semantik wie zum Beispiel bei den „Erweiterungen“ gesehen (leicht möglich, inkonsistente „Erweiterungen“ zu entwerfen), gebieten es, auch nach anderen Semantiken zu suchen.
Als Motivation für die Richtung der Suche betrachten wir wieder ein Beispiel.

**Beispiel**

Zunächst sei nat und bool wie oben, dann sei

$$
\begin{array}{lll}
\underline{\underline{\text{nateq}}} = \underline{\underline{\text{bool}}} + \underline{\underline{\text{nat}}} + & & \\
 & \underline{\text{oprs}} & \text{eq} : \text{nat}, \text{nat} \mapsto \text{bool} \\
 & \underline{\text{vars}} & n, m : \text{nat} \\
 & \underline{\text{eqns}} & 0 \,\text{eq}\, 0 = \text{true} \\
 & & 0 \,\text{eq}\, \text{suc}(m) = \text{false} \\
 & & \text{suc}(n) \,\text{eq}\, 0 = \text{false} \\
 & & \text{suc}(n) \,\text{eq}\, \text{suc}(m) = n \,\text{eq}\, m \\
\underline{\text{end}}\ \underline{\underline{\text{nateq}}} & &
\end{array}
$$

Man kann leicht feststellen, daß nateq eine Erweiterung von nat ist.

Betrachte weiter (anders als im Beispiel von Kapitel 5)

$\underline{\underline{\text{set(nat)}}} = \underline{\underline{\text{nateq}}} +$

$\underline{\text{sorts}}$ set

$\underline{\text{oprs}}$ $\emptyset : \mapsto \text{set}$
insert: $\text{set}, \text{nat} \mapsto \text{set}$
del: $\text{set}, \text{nat} \mapsto \text{set}$
if–then–else–set: $\text{bool}, \text{set}, \text{set} \mapsto \text{set}$
if–then–else–nat: $\text{bool}, \text{nat}, \text{nat} \mapsto \text{nat}$
if–then–else–bool: $\text{bool}, \text{bool}, \text{bool} \mapsto \text{bool}$
$\in$: $\text{nat}, \text{set} \mapsto \text{bool}$
equal : $\text{set}, \text{set} \mapsto \text{bool}$
$\subseteq$: $\text{set}, \text{set} \mapsto \text{bool}$
card : $\text{set} \mapsto \text{nat}$

$\underline{\text{vars}}$ $s, t : \text{set}$
$m, n : \text{nat}$
$p, q : \text{bool}$

$\underline{\text{eqns}}$ if true then $s$ else $t$ set $= s$
if false then $s$ else $t$ set $= t$
if true then $m$ else $n$ nat $= m$
if false then $m$ else $n$ nat $= n$
if true then $p$ else $q$ bool $= p$
if false then $p$ else $q$ bool $= q$
$n \in \emptyset = \text{false}$
$n \in \text{insert}(s, m) = \text{if}(n \,\text{eq}\, m)$ then true else $n \in s$ bool
$\text{card}(\emptyset) = 0$
$\text{card}(\text{insert}(s, n)) = \text{if}(n \in s)$ then $\text{card}(s)$ else $\text{suc}(\text{card}(s))$ nat
$\text{del}(\emptyset, n) = \emptyset$
$\text{del}(\text{insert}(s, n), m) = \text{if}(n \,\text{eq}\, m)$ then $\text{del}(s, m)$ else $\text{insert}(\text{del}(s, m), n)$ set
$(\emptyset \subseteq \emptyset) = \text{true}$
$(\emptyset \subseteq \text{insert}(s, n) = \text{true}$
$(\text{insert}(s, n) \subseteq t) = (n \in t, s \subseteq t)$
$(s \,\text{equal}\, t) = (s \subseteq t, t \subseteq s))$

$\underline{\text{end}}$ $\underline{\underline{\text{set(nat)}}}$

Für die initiale Semantik betrachten wir wieder die Algebra

$$\underline{\underline{T_{\text{set(nat)}}}}.$$

Anders als in dem früheren Beispiel (zu $\underline{\underline{set(\text{nat})}}$) gilt hier:

$$[\text{insert}(\text{insert}(\emptyset, 0), 0)]_{\equiv_E} \neq [\text{insert}(\emptyset, 0)]_{\equiv_E} \qquad \text{usw.}$$

Verschiedene Darstellungen „derselben Menge“ werden also durch die initiale Semantik unterschieden!

Oft wird bei Datentypen folgender Ansatz verfolgt:
Man unterscheidet zwischen „sichtbaren“ oder „erlaubten“ bzw. „beobachtbaren“ Sorten und Operationen und „unsichtbaren“ bzw. „versteckten“ („hidden“) Sorten bzw. Operationen. Wenn man auf diese Weise die Beispiel–Spezifikation betrachtet mit

$\underline{\underline{\text{set(nat)}}}$

$\vdots$

$\underline{\text{hidden}}$ $\underline{\text{sorts}}$ set ,
$\underline{\text{oprs}}$ $\emptyset, \text{insert}, \text{del}$

$\vdots$

d.h. , daß man über die „neuen“ Elemente (die der Sorte set) nur etwas erfährt durch „Beobachtung“ der „erlaubten“ Operationen $\in$, equal, $\subseteq$, card, dann kann man die obigen verschiedenen Darstellungen nicht mehr unterscheiden! Z. B. :

$$\begin{aligned} \mathrm{card}(\mathrm{insert}(\mathrm{insert}(\emptyset,0),0) &= \text{if } \underbrace{(0 \in \mathrm{insert}(\emptyset,0))}_{=\mathrm{true}} \text{ then } \mathrm{card}(\mathrm{insert}(\emptyset,0)) \\ &\quad \text{else } \mathrm{suc}(\mathrm{card}(\mathrm{insert}(\emptyset,0))) \text{ nat} \\ &= \mathrm{card}(\mathrm{insert}(\emptyset,0)) \end{aligned}$$

Analog für $\in$, equal usw.

Also:

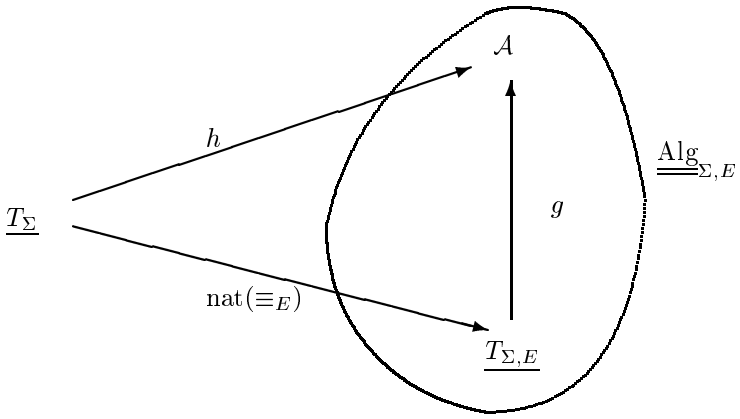
Das *beobachtbare Verhalten* der verschiedenen Darstellungen für dieselbe Menge ist gleich!

Wenn man sich also nur für das beobachtbare Verhalten interessiert, ist das initiale Modell zu groß! Generell ist das initiale Modell oft zu groß (vergleiche z. B. ternary weiter vorn).

Die initiale Algebra $T_{\Sigma,E}$ in $\mathrm{Alg}_{\Sigma,E}$ ist eine „möglichst große“ Algebra, die vom leeren Erzeugendensystem erzeugt wird. Es gilt für zwei Terme $t_1, t_2 \in T_\Sigma$, daß

$$[t_1]_{\equiv_E} \neq [t_2]_{\equiv_E} \quad \text{bzw.} \quad t_1 \not\equiv_E t_2\,,$$

wenn sie nur in irgendeiner Algebra $\mathcal{A} \in \mathrm{Alg}_{\Sigma,E}$ verschiedene Werte haben.




Wenn $h(t_1) \neq h(t_2)$, so muß auch sein $[t_1]_{\equiv_E} \neq [t_2]_{\equiv_E}$, sonst wäre das obige Diagramm nicht kommutativ:

$$\begin{aligned} \text{wäre } [t_1] &= [t_2], \text{ so wäre} \\ g([t_1]) &= g([t_2]) \end{aligned}$$

und mit $i = \mathrm{nat}(\equiv_E)$ müßte sein

$$i(g([t_1])) = i(g[t_2])),$$

dies bedeutet aber wegen $h = i \circ g$

$$h(t_1) = h(t_2).$$

Das ist aber ein Widerspruch.

Obige Probleme lassen fragen, gibt es auch eine vom leeren Erzeugendensystem erzeugte Algebra in $\mathrm{Alg}_{\Sigma,E}$, die „möglichst klein“ ist? Z. B. sollten zwei Terme $t_1, t_2 \in T_\Sigma$ in der neuen „semantischen Algebra“ nur dann verschiedene Werte haben, wenn sie in *allen* Modellen, d. h. allen $(\Sigma, E)$–Algebren, verschiedene Werte haben.
Vergleiche aber weiter vorn: In $\mathrm{Alg}_{\Sigma,E}$ liegt stets die entartete Algebra, die in jeder Trägermenge genau ein Element enthält. Diese würde obige Bedingung erfüllen, wäre aber *viel zu klein*.

„Initialität“ wurde mit Hilfe des Homomorphiebegriffs definiert. Ein dualer Ansatz führt zum Begriff „final“ bzw. „terminal“ :

**Definition 6.1**

Sei $K \subseteq \underline{\underline{\text{Alg}}}_{\Sigma}$ eine Klasse von $\Sigma$–Algebren. Dann heißt eine Algebra $\mathcal{A} \in K$ *final* in $K$ (*terminal* in $K$) genau dann, wenn für jede Algebra $\mathcal{B} \in K$ genau ein Homomorphismus $h$

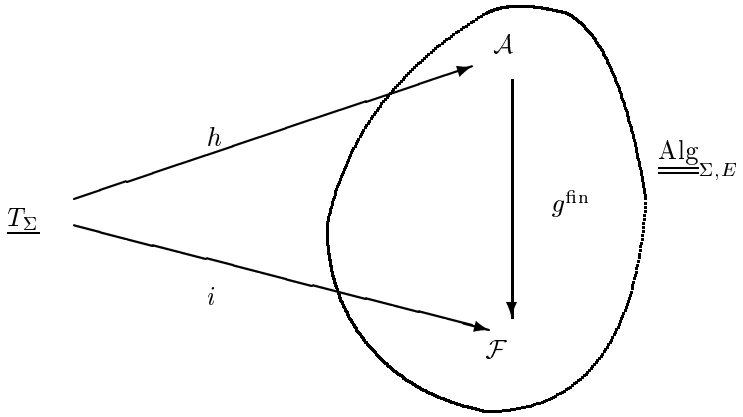
$$h : \ \mathcal{B} \mapsto \mathcal{A}$$

existiert.

**Bemerkung**

Wie im Falle der Initialität kann man leicht zeigen, daß zwei finale Algebren in $K$ stets isomorph sind. Die in $K$ finalen Algebren bilden eine Isomorphieklasse. Aber natürlich gibt es nicht in jeder Klasse $K$ finale Algebren.

Sei $\mathcal{A}$ eine $(\Sigma, E)$–Algebra, dann ergibt sich unter der Annahme, daß es in $\underline{\underline{\text{Alg}}}_{\Sigma,E}$ eine finale Algebra $\mathcal{F}$ gibt:




$i(t_1) = i(t_2) \rightsquigarrow g^{fin}(h(t_1)) = g^{fin}(h(t_2))$.
Umgekehrt: wenn für irgendein Modell $\mathcal{A}$ $\quad h(t_1) = h(t_2)$ ($t_1$ und $t_2$ haben in $\mathcal{A}$ denselben Wert), so gilt auch $i(t_1) = i(t_2)$, d. h. $t_1$ und $t_2$ haben auch in $\mathcal{F}$ denselben Wert. Dies führt also wieder zu der bereits bekannten, uninteressanten, entarteten Algebra! Diese ist „die“ finale in $\underline{\underline{\text{Alg}}}_{\Sigma,E}$.

Will man nach einer sinnvollen *finalen Semantik* suchen, so muß man die Algebrenklasse verkleinern! Dazu werden in der Literatur zwei **Methoden** verfolgt:

- Definiere eine echte Unterklasse $MOD \subset \underline{\underline{\text{Alg}}}_{\Sigma,E}$ von „zulässigen“ Modellen von $\underline{\underline{\text{spec}}} = (\Sigma, E)$. Wenn es in $MOD$ eine finale Algebra $\mathcal{F}$ gibt, so definiere die Isomorphieklasse von $\mathcal{F}$ als den durch die finale Semantik bestimmten abstrakten Datentyp $FIN_{\underline{\underline{\text{spec}}}}$.

- Definiere mittels des Gleichungssystems $E$ eine Kongruenzrelation $\sim_E$ und bilde $\underline{T_\Sigma}/\sim_E$. Zeige, daß $\underline{T_\Sigma}/\sim_E$ final in einer bestimmten Klasse $MOD \subset \underline{\underline{\text{Alg}}}_{\Sigma,E}$ ist, und setze die Isomorphieklasse von $\underline{T_\Sigma}/\sim_E$ als den abstrakten Datentyp $FIN_{\underline{\underline{\text{spec}}}}$.

**Bemerkung**

- „initiale Semantik“ ist fester Begriff
- für die „finale Semantik“ hat man Wahlmöglichkeiten entweder in der Wahl von $MOD$ (dann muß man sichern, daß eine finale Algebra in $MOD$ existiert), oder bei der Wahl von $\sim_E$ (hier ist dann zu sichern, daß $\underline{T_\Sigma}/\sim_E$ final in einer gewissen Klasse $MOD$ ist).

Es existieren deshalb viele verschiedene Ansätze zur finalen Semantik in der Literatur.

Begründer der Anwendung der finalen Semantik auf die Erweiterung von Gleichungsspezifikationen ist M. WAND (1977) mit folgendem Ansatz:

**Definition 6.2**

Sei

$$\underline{\underline{\text{spec}'}} = \underline{\underline{\text{spec}}} + (S_0, \Sigma_0, E_0)$$

eine Erweiterung (Bezeichnungen wie weiter oben).
Die Klasse $EMOD$ von *zulässigen Modellen* der erweiterten Spezifikation $\underline{\underline{\text{spec}'}}$ wird definiert durch:

$$EMOD \subset \underline{\underline{\text{Alg}}}_{\underline{\underline{\text{spec}'}}}$$

mit $\mathcal{A} \in EMOD$ genau dann, wenn

1. $\mathcal{A}$ wird vom leeren Erzeugendensystem $\underline{\emptyset}$ erzeugt (d. h. $\mathcal{A}$ ist minimale Algebra) und
2. der eindeutig bestimmte Homomorphismus

$$h: \underline{\underline{T_{\text{spec}}}} \mapsto (\mathcal{A})_{\underline{\underline{\text{spec}}}}$$

ist injektiv.

**Satz 6.1**

Unter der Voraussetzung der obigen Definition gilt:

1. $EMOD \neq \emptyset$
2. Es gibt eine finale Algebra $\mathcal{F} \in EMOD$.
3. Falls $\underline{\underline{T_{\text{spec}}}}$ keine entartete Algebra (genau ein Element in jeder Sorte) ist, so ist $\mathcal{F}$ ebenfalls nicht entartet.

**Beweis**

zu 1.: $\underline{\underline{T_{\text{spec}'}}} \in EMOD$ (weil $\underline{\underline{spec'}}$ Erweiterung ist, also konsistent und damit $h$ injektiv).

zu 3.: jedes zulässige Modell $\mathcal{A}$ besitzt nach Definition (2.) mindestens so viele Elemente wie $\underline{\underline{T_{\text{spec}}}}$.

zu 2.: hier ohne Beweis
(WAND benutzte eine spezielle Kongruenzrelation $R$ (vgl. später) und zeigte, daß $\mathcal{F} = T_{\underline{\underline{\text{spec}'}}}/R$ final in $EMOD$ ist.)

q.e.d.

**Definition 6.3**

Der durch die *finale Semantik* der erweiterten Spezifikation bestimmte abstrakte Datentyp $FIN_{\underline{\underline{\text{spec}'}}}$ ist die Isomorphieklasse der finalen Algebra $\mathcal{F} \in EMOD$.

Folgende Definition stellt die Konstruktion einer ähnlichen Kongruenzrelation dar, wie sie von WAND benutzt wurde, und liefert gleichzeitig einen Ansatz, das „beobachtbare Verhalten" zu formalisieren.

Im weiteren sei wieder

$$\underline{\underline{\text{spec}'}} = \underline{\underline{\text{spec}}} + (S_0, \Sigma_0, E_0)$$

eine Erweiterung mit

$$\underline{\underline{\text{spec}'}} = (\Sigma', E'),\ \ \underline{\underline{\text{spec}}} = (\Sigma, E),\ \Sigma' = \Sigma \dot\cup \Sigma_0,\ \ E' = E \dot\cup E_0,\ \ S' = S \dot\cup S_0.$$

**Definition 6.4**

Für eine Sorte $s' \in S_0$ sei $x$ eine Variable dieser Sorte $s'$. Nun wird für $\mathcal{A} \in \underline{\underline{\text{Alg}}}_{\underline{\underline{\text{spec}'}}}$ mit Erzeugendensystem $\underline{\emptyset}$ (d. h. $\mathcal{A}$ ist minimal) eine Relation $\equiv_{\mathcal{A}}$ definiert:
Für $a, b \in \mathcal{A}$ gilt, sie sind *abstrakt gleich*, kurz

$$a \equiv_{\mathcal{A}} b$$

genau dann, wenn

$$a, b \in A_s \text{ für ein } s \in S \text{ und } a = b$$

oder

$$a, b \in A_{s'} \text{ für ein } s' \in S_0 \text{ (neue Sorte!) und}$$

$$\text{für alle Terme } t \in T_{\Sigma'}(\{x\})_s \text{ mit } s \in S \text{ (alte Zielsorte) ist}$$

$$\text{wert}_{\mathcal{A}}(t, f_a) = \text{wert}_{\mathcal{A}}(t, f_b)$$

$$\text{für Belegungen } f_a : x \mapsto a \text{ bzw. } f_b : x \mapsto b.$$

Das heißt, bei Auswertung beliebiger Terme mit alten Zielsorten in $\mathcal{A}$ lassen sich $a$ und $b$ nicht unterscheiden; sie zeigen die gleiche Wirkung.

**Bemerkung**

1. $\equiv_{\mathcal{A}}$ ist eine Kongruenzrelation für alle minimalen $\underline{\underline{\text{spec}'}}$–Algebren $\mathcal{A}$.
2. Die Relation $\equiv_{\mathcal{A}}$ (*abstrakt gleich*) ist eine Formalisierung des „beobachtbaren Verhaltens".

**Definition 6.5**

(Bezeichnungen wie oben)
Zwei minimale $\underline{\underline{\text{spec}'}}$–Algebren $\mathcal{A}$ und $\mathcal{B}$ (mit leerem Erzeugendensystem $\underline{\emptyset}$) heißen *austauschbar* genau dann, wenn

1. Für die $\underline{\underline{\text{spec}}}$—Redukte gilt

   $$(\mathcal{A})_{\underline{\underline{\text{spec}}}} \cong (\mathcal{B})_{\underline{\underline{\text{spec}}}}.$$

   „Basistypen sind gleich."

2. Für die eindeutig bestimmten Homomorphismen

   $$h_{\mathcal{A}} : \underline{T_{\Sigma'}} \mapsto \mathcal{A} \qquad \text{und} \qquad h_{\mathcal{B}} : \underline{T_{\Sigma'}} \mapsto \mathcal{B}$$

   und den Isomorphismus (entspechend 1.)

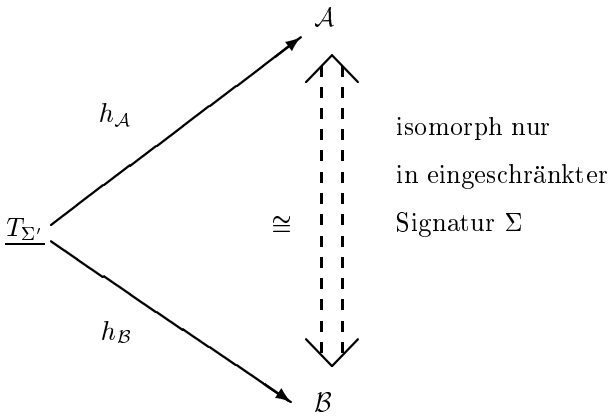
   $$i : (\mathcal{A})_{\underline{\underline{\text{spec}}}} \mapsto (\mathcal{B})_{\underline{\underline{\text{spec}}}}$$

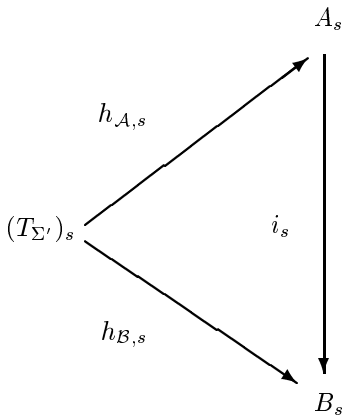
   „Das beobachtbare Verhalten ist gleich."

   gilt:

   $$h_{\mathcal{B}}(t) = i(h_{\mathcal{A}}(t))$$

   für jeden Term $t \in (T_{\Sigma'})_s$ mit $s \in S$.




Mit diesem weiteren Begriff, der ebenfalls das beobachtbare Verhalten formalisiert, kann man nun eine Isomorphieklasse von zulässigen Modellen bilden:

**Definition 6.6**

$$\underline{\underline{\text{spec}'}} = \underline{\underline{\text{spec}}} + (S_0, \Sigma_0, E_0)$$

mit Bezeichnungen wie oben sei eine Erweiterung. Sei die Klasse

$$ZMOD \subset \underline{\underline{\text{Alg}}}_{\underline{\underline{\text{spec}'}}}$$

von *zulässigen Modellen* festgelegt durch

1. $\underline{T_{\Sigma'}}/\equiv_{E'} \in ZMOD$ (die initiale $\underline{\underline{\text{spec}'}}$–Algebra gehört dazu).

2. Wenn $\mathcal{A} \in ZMOD$ und $\mathcal{A}$ und $\mathcal{B}$ sind austauschbar, so auch $\mathcal{B} \in ZMOD$.

**Bemerkung**

$ZMOD$ ist damit eine Isomorphieklasse, deren Elemente man erhält, indem man zur initialen Algebra alle Modelle hinzunimmt, die bezüglich der Basissorten das gleiche beobachtbare Verhalten besitzen.

Es gilt der

**Satz 6.2**

Es gibt in $ZMOD$ eine finale Algebra $\mathcal{F}$.
$\mathcal{F}$ ist nur dann die entartete Algebra, wenn auch die initiale spec–Redukt–Algebra $T_\Sigma / \equiv_E$ entartet ist.

Damit hat man als „vernünftige“ *finale Semantik* der erweiterten Spezifikation spec′ die Isomorphieklasse von $\mathcal{F} \in ZMOD$ als den spezifizierten abstrakten Datentyp.

**Beweis**

(Skizze) Betrachte $\mathcal{I} =_{df} T_{\Sigma',E'} = T_{\Sigma'} / \equiv_{E'}$. $\mathcal{I}$ ist also spec′–Algebra. Bilde die Faktoralgebra von $\mathcal{I}$ nach der Kongruenzrelation $\equiv_{\mathcal{I}}$. Es gilt:

$$\mathcal{I} / \equiv_{\mathcal{I}} \in ZMOD, \text{ weil } \mathcal{I} \text{ und } \mathcal{I} / \equiv_{\mathcal{I}}$$

austauschbar sind! Letzteres ist leicht zu zeigen.

$\mathcal{I} / \equiv_{\mathcal{I}}$ ist *final* in $ZMOD$:

Dazu zeigt man, daß es zu jeder Algebra $\mathcal{B} \in ZMOD$ genau einen Homomorphismus

$$h : \mathcal{B} \mapsto \mathcal{I} / \equiv_{\mathcal{I}}$$

gibt. Das geschieht in folgenden Schritten:

Betrachte folgende Abbildung

$$abs : \ B \mapsto \mathcal{I} / \equiv_{\mathcal{I}}$$

mit

$$abs(b) = [h_{\mathcal{I}}(t)]_{\equiv_{\mathcal{I}}},$$

wobei $t \in T_{\Sigma'}$ mit $h_{\mathcal{B}}(t) = b$.
Da $\mathcal{B}$ minimal (von $\emptyset$ erzeugt,), besteht $\mathcal{B}$ nur aus Termelementen, folglich gibt es zu jedem $b \in \mathcal{B}$ ein solches $t$. $h_{\mathcal{I}}$ und $h_{\mathcal{B}}$ sind die wegen der Initialität von $T_{\Sigma'}$ eindeutig existierenden Homomorphismen.
Nun ist zu zeigen:

1. $abs$ ist eine Funktion, d. h. repräsentantenunabhängig definiert (von $t$ unabhängig).
2. $abs$ ist Homomorphismus (trivial).
3. für jeden Homomorphismus $h : \mathcal{B} \mapsto \mathcal{I} / \equiv_{\mathcal{I}}$ gilt $h = abs$.

q.e.d.

Der so konstruierte Homomorphismus $abs$ heißt *Abstraktionsfunktion* von $\mathcal{B}$. $abs$ liefert zu jedem Element $b \in B$ dasjenige Element der finalen Algebra (d. h. der semantischen Algebra !), das durch $b$ „dargestellt“ wird.

**Satz 6.3**

In der finalen semantischen Algebra

$$\mathcal{F} \in \text{ ZMOD}$$

gilt für zwei Elemente $a, b \in \mathcal{F}_{s'}$ mit $s' \in S_0$ (neue Sorte !):

$$\text{wenn } a \equiv_{\mathcal{F}}, \text{ so } a = b,$$

d. h., keine zwei verschiedenen Elemente sind abstrakt gleich.

# Kapitel 7

# Implementation von Spezifikationen

Eine Implementation einer Spezifikation läßt sich abstrakt stets als eine *Algebra* ansehen:

- gewisse Datenbereiche/Zustandsmengen o.ä,
- darüber gewisse Operationen.

Daraus resultiert die Frage:
Wann ist eine gegebene Algebra $\mathcal{A}$ eine Implementation einer Spezifikation $\underline{\underline{\text{spec}}}$ ?

Naheliegender Gedanke:

$$\mathcal{A} \text{ heißt } \textit{Implementation} \text{ von } \underline{\underline{\text{spec}}} = (\Sigma, E) \qquad \text{genau dann, wenn} \qquad \mathcal{A} \cong \underline{T_{\Sigma,E}}.$$

Schwächen dieser „Definition“:

- Es folgt, daß $\mathcal{A}$ $\Sigma$–Algebra ist, d. h. speziell, daß die Implementation genau dieselben Sorten(bezeichnungen) usw. haben müste wie der zu implementierende Datentyp.
- Der Begriff ist praxisfern, denn abgesehen davon, daß $\mathcal{A}$ nicht dieselbe Sortenmenge $S$ haben muß, könnten bei der Implementation *mehr* und *andere Sorten* vorkommen, und eventuell auch *mehr* bzw. *andere* (vielleicht detailliertere) *Operationen*.

Also:
Es müssen auch andere Signaturen $\Sigma'$ zugelassen sein.
Weiter zu klären ist, wie die Implementation $\mathcal{A}$ angegeben werden soll ?

- „konkrete“ Definition der Datenbereiche und Operationen, oder
- abstrakt wieder durch eine Spezifikation.

Im Sinne der „schrittweisen Verfeinerung“, des „strukturierten Entwurfs“ auch von Implementationen bevorzugen wir die 2. Möglichkeit.

Frage:
Wann implementiert eine Spezifikation $\underline{\underline{\text{spec}'}} = (\Sigma', E')$ eine Spezifikation $\underline{\underline{\text{spec}}} = (\Sigma, E)$ korrekt ?

Die bei der Untersuchung dieser Frage auftretenden Probleme sollen an Beispielen erläutert werden.

**Beispiel**

- bool sei dieselbe Spezifikation wie weiter oben. Bekannt ist, daß alle BOOLeschen Funktionen durch eine einzige, z. B. die nand–Funktion $(= \mathrm{not}((-,-)))$, dargestellt werden können. Also muß sich bool durch den folgenden Datentyp implementieren lassen:

$$\begin{array}{lcll} \underline{\underline{\text{bit}}} & = & \underline{\text{sorts}} & \text{bit} \\ & & \underline{\text{oprs}} & 0, 1 :\mapsto \text{bit} \\ & & & \text{nand} : \text{bit}, \text{bit} \mapsto \text{bit} \\ & & \underline{\text{vars}} & b : \text{bit} \\ & & \underline{\text{eqns}} & \text{nand}(0, b) = 1 \\ & & & \text{nand}(b, 0) = 1 \\ & & & \text{nand}(1, 1) = 0 \\ \underline{\text{end}}\ \underline{\underline{\text{bit}}} & & & \end{array}$$

**Klar:** Zur Implementation von bool durch bit muß man die Sorten *umbenennen*:

$$\text{bool} \to \text{bit},$$

die Datenelemente/Konstanten *umbennen*:

$$\text{false} \to 0, \qquad \text{true} \to 1,$$

die Operationen von bool

$$\text{not}, , \text{or}, \text{implies}$$

durch die Operationen von bit erklären:

$$\begin{array}{rcl} \text{not}(p) & := & \text{nand}(p, p) \\ (p, q) & := & \text{nand}(\text{nand}(p, q), \text{nand}(p, q)) \\ \text{or}(p, q) & := & \text{nand}(\text{nand}(p, p), \text{nand}(q, q)) \\ \text{implies}(p, q) & := & \text{nand}(p, \text{nand}(q, q)). \end{array}$$

Das heißt, den Operationen von bool entsprechen *abgeleitete Operationen* von bit. Zusätzlich ist die Operation nand in bool zuviel, sie muß also „*vergessen*“ werden.

- nat wie weiter oben. In „allen“ Programmiersprachen gibt es keinen „eigenen“ Datentyp für die natürlichen Zahlen, sie werden durch **integer** dargestellt. Also muß man nat durch int implementieren können:

$$\begin{array}{lcll} \underline{\underline{\text{int}}} & = & \underline{\text{sorts}} & \text{int} \\ & & \underline{\text{oprs}} & 0 :\mapsto \text{int} \\ & & & \text{suc}, \text{pred} : \text{int} \mapsto \text{int} \\ & & & + : \text{int}, \text{int} \mapsto \text{int} \\ & & \underline{\text{vars}} & k, l : \text{int} \\ & & \underline{\text{eqns}} & \text{pred}(\text{suc}(k)) = k \\ & & & \text{suc}(\text{pred}(k)) = k \\ & & & k + 0 = k \\ & & & k + \text{suc}(l) = \text{suc}(k + l) \\ & & & k + \text{pred}(l) = \text{pred}(k + l) \\ \underline{\text{end}}\ \underline{\underline{\text{int}}} & & & \end{array}$$

Innerhalb von $T_{\underline{\underline{\text{int}}}}$ läßt sich $T_{\underline{\underline{\text{nat}}}}$ „implementieren“ durch „Vergessen“ der Operation pred und *Einschränkung* der Trägermenge, d. h. *Vergessen von Datenelementen*, nämlich aller „negativen Zahlen“ $\text{pred}^n(0)$. Das heißt, man kann auf den Teil von $T_{\underline{\underline{\text{int}}}}$ einschränken, der bei den noch erlaubten Operationen *erreichbar* ist (entspricht der Unteralgebrenbildung).

- int wie eben soll implementiert werden durch:

vorznat = sorts nat, int, bool
oprs $0 :\mapsto \text{nat}$
$\text{suc}, \text{pred} : \text{nat} \mapsto \text{nat}$
$+, \dot{-} : \text{nat}, \text{nat} \mapsto \text{nat}$
$\text{abs} : \text{nat}, \text{bool} \mapsto \text{nat}$
$\text{i} : \text{nat}, \text{bool} \mapsto \text{int}$
if–then–else–fi : bool, int, int $\mapsto$ int
true, false : bool
$\text{sgn} : \text{int} \mapsto \text{bool}$
$\leq : \text{nat}, \text{nat} \mapsto \text{bool}$
$\text{eq} : \text{bool}, \text{bool} \mapsto \text{bool}$
vars $k, l : \text{int}$
$n, m : \text{int}$
$p : \text{bool}$
eqns $\text{pred}(0) = 0$
$\text{pred}(\text{suc}(m)) = m$
$n \dot{-} 0 = n$
$n \dot{-} \text{suc}(m) = \text{pred}(n \dot{-} m)$
$n + 0 = n$
$n + \text{suc}(m) = \text{suc}(n + m)$
$(0 \leq n) = \text{true}$
$(\text{suc}(n) \leq 0) = false$
$(\text{suc}(n) \leq \text{suc}(m)) = (n \leq m)$
$\text{true}\,\text{eq}\,p = p$
$p\,\text{eq}\,\text{true} = p$
$\text{false}\,\text{eq}\,\text{false} = \text{true}$
if true then $k$ else $l$ fi $= k$
if false then $k$ else $l$ fi $= l$
$\text{i}(\text{abs}(k), \text{sgn}(k)) = k$
$\text{abs}(\text{i}(n, p)) = n$
$\text{sgn}(\text{i}(n, p)) = p$
end vorznat

Die Elemente von int sollen also durch Paare von natürlichen Zahlen und „Vorzeichen“ implementiert werden! Dazu müssen die int–Operationen $0, \text{suc}, \text{pred}, +$ durch abgeleitete Operationen aus vorznat erklärt werden, z. B. :

$$\begin{aligned}\text{pred}_{\text{int}}(x) \quad := \quad & \text{if } (0 \leq \text{abs}(x))\,\text{eq}(\text{abs}(x) \leq 0))\ \text{then i}(\text{suc}(0), \text{false}) \\ & \quad \text{else if sgn}(x)\ \text{then pred}_{\text{vorznat}}(x)\ \text{else i}(\text{suc}(\text{abs}(x)), \text{false})\ \text{fi} \\ & \text{fi}\end{aligned}$$

$\vdots$
usw.

Anschließend muß man einschränken auf die Sorte int. Das reicht aber noch nicht aus, weil es zwei Nullen gibt:

$$\text{i}(0, \text{false}),\ \text{i}(0, \text{true}).$$

Diese müssen *identifiziert* werden.

**Bemerkung**

Drei Schritte sind bei Implementationen im allgemeinen notwendig (für einen Implementationsbegriff zu formalisieren):

1. **Synthese**
   Zu implementierende Operationen und Sorten müssen mit Hilfe der Operationen und Sorten der Implementation erklärt werden.

2. **Identifikation**
   In der Implementation kann es zu Mehrfachdarstellungen desselben zu implementierende Datenelements kommen, diese müssen identifiziert werden (Äquivalenzklassenbildung mit Hilfe Kongruenzrelation).

3. **Restriktion**
   Die Trägermengen können zu groß sein: Einschränkung auf Unter– und Teilalgebren.

Aus der Formalisierung dieses Vorgehens ergibt sich eine brauchbare Definition des Implementationsbegriffs.

# Index