Algebraische Spezifikation

Prof. Dr. Hans-Jörg Kreowski

Studiengang Informatik

Wintersemester 2001/02

MZH 3260
Tel.: 2956, 3697 (Sekr.), Fax: 4322

E-Mail: kreo@informatik.uni-bremen.de
www.informatik.uni-bremen.de/theorie

1 Einleitung

Dieses Material und die zugehörige Lehrveranstaltung sollen die mathematischen Grundlagen für die algebraische Spezifikation von Datentypen bereitstellen. Ist das wirklich nötig? Was hat Informatik überhaupt mit Mathematik zu tun, was speziell Datentypen mit Algebren? Diese Fragen sind Schlüssel zum Verständnis der algebraischen Spezifikation von abstrakten Datentypen und der dafür erforderlichen mathematischen Theorie der universellen Algebren; es soll deshalb eine Antwort versucht werden.

Was hat Informatik mit Mathematik zu tun?

> *Every program performs some task correctly. What is of interest to computer scientists is whether a program performs its intended task.*
>
> B. Liskov und V. Berzins

Zu den wichtigsten Aufgaben der Informatikerinnen und Informatiker zählt der Entwurf von Algorithmen (Programmen, Programmpaketen, großen Softwaresystemen), die vorgegebene Probleme lösen. Typische Aufgabenstellungen sind beispielsweise: Übersetze eine Programmiersprache; betreibe eine Rechenanlage; verwalte eine Bibliothek! Sie werden bewältigt durch die Entwicklung entsprechender Compiler, Betriebs- und Datenbanksysteme. Die Betonung muss dabei auf "entsprechend" liegen, denn es ist evident, dass ein Texteditor keine der gestellten Aufgaben erledigen kann. Deshalb lautet eine zentrale – leider für nichtnumerische Algorithmen äußerst selten explizit gestellte und noch seltener beantwortete – Frage:

Löst Algorithmus A Problem B?

Diese Frage nach der Korrektheit eines Algorithmus ist vorrangig; denn erst wenn sie positiv beantwortet ist, können ernsthaft und sinnvoll andere erwünschte Eigenschaften von Algorithmen bzw. Softwaresystemen wie Strukturiertheit, Effizienz, Zuverlässigkeit, Stabilität, Portabilität usw. erörtert werden. Dass sie dennoch so selten diskutiert wird, hat sicherlich sehr vielschichtige Gründe:

- Häufig erscheinen die Probleme (größter gemeinsamer Teiler; sortiere ganze Zahlen nach Größe) so einfach, dass man den Algorithmen intuitiv ansieht, ob sie das Gewünschte leisten.
- Unterstützt wird das dazu nötige Vorstellungsvermögen durch Namensgebung und Kommentierung der Algorithmen (GGT, LIST-OF-INT, CHANGE, ...), was einen Zusammenhang zu gegebenen Problemen suggeriert.
- Wo Intuition und Suggestion versagen, kann man dann testen (und bei negativem Ausgang die Programme revidieren und erneut testen ...).

– Es sind viele ausgezeichnete, sehr bequeme, leicht handhabbare Konzepte bekannt, Algorithmen zu schreiben – insbesondere aus den höheren Programmiersprachen; dagegen steht zum Aufschreiben der Probleme selten mehr als die natürliche Sprache zur Verfügung.
– Dieses Missverhältnis macht einen fundierten Vergleich von Algorithmen und Problemen nahezu unmöglich, so dass manche ihn sogar für unnötig halten.

Aus wissenschaftlicher Sicht ist dieses Dilemma fehlender Möglichkeiten (und Einsicht) ganz unbefriedigend und Abhilfe dringlich. Um aber Problemstellungen exakt formulieren und Algorithmen daraufhin überprüfen zu können, ob sie die gewünschten Aufgaben erfüllen, ob sie also korrekt sind, ist eine gemeinsame Sprachebene zum Schreiben von Problemen und Algorithmen erforderlich. Es werden also in der Informatik Spezifikations- und Implementierungssprachen gebraucht, die Korrektheitsbeweise unterstützen bzw. überhaupt erst erlauben.

Welchen Charakter wird eine solche Sprache haben müssen? Es werden einerseits Sprachmittel benötigt, mit denen Probleme und sie lösende Algorithmen ausgedrückt werden können; welche Konsequenzen ergeben sich aber daraus, dass zusätzlich beantwortbar sein soll, ob Algorithmus A Problem B löst?

Betrachtet man die beiden möglichen Antworten "A löst B" und "A löst B nicht" (das bisher in der Informatik übliche Achselzucken – teils wortreich verbrämt – nicht gerechnet), so handelt es sich dabei um Aussagen, um Behauptungen, die wahr sein können oder falsch. Will man definitiv wissen, welche Antwort zutrifft, so muss man eine der beiden verifizieren oder falsifizieren. In einem exakten Sinne lassen sich Richtigkeit oder Falschheit von Aussagen jedoch ausschließlich in mathematischen Theorien nachweisen.

Einer der bedeutendsten Zusammenhänge von Mathematik und Informatik spiegelt sich deshalb in folgender These wider.

These 1
Korrektheit erfordert mathematische Theorien.

Was haben Datentypen mit Algebren zu tun?

Types are not sets.

F.L. Morris

Algorithmen (Programme, große Softwaresysteme) sind aus kleinen "Bausteinen" zusammengesetzt, zu denen die Kontrollstrukturen wie *if–then–else*, *while–do*, ... zählen. Diese sind bei praktisch allen Algorithmen in ähnlicher Weise zu finden, und ihre Wahl hängt mehr vom Programmierstil, der Programmdisziplin und vom persönlichen Geschmack der einzelnen Programmiererin bzw. des einzelnen Programmierers ab als von den konkreten Programmieraufgaben. Die problemorientierte Sprachebene von Algorithmen und damit ihre Fähigkeit, bestimmte Probleme zu lösen, ist dagegen erst durch den Datentyp gegeben. Einige bekannte oder typische Beispiele für Datentypen, die auch abstrakte Daten-

typen genannt werden, wenn sie unabhängig von der Repräsentation der Daten behandelt werden (können), sind in folgender Liste zusammengestellt:

Datentyp	Bezeichnung
Wahrheitswerte, Boolesche Werte	BOOL
endliche Menge, Alphabet	A
natürliche Zahlen	NAT
ganze Zahlen	INT
Vektoren, Matrizen, Arrays	ARRAY
Wörter	STRING
Keller	STACK
Schlangen	QUEUE
Teilmengen, Potenzmenge	SET
Bäume	TREE
verkettete Listen	LIST
Graphen	GRAPH
Symboltabellen (Teil eines Compilers)	SYMTAB
Filesystem (Teil eines Betriebssystems)	FILE
Airportschedule (spezielles Datenbanksystem)	APS

Damit die Charakteristika von Datentypen erfasst werden können, sollen einige dieser Beispiele genauer betrachtet werden.

1.1 Beispiel

1. BOOL, der Datentyp der Wahrheitswerte, besteht aus einer zweielementigen Menge $\mathbb{B} = \{\mathsf{true}, \mathsf{false}\}$ mit dem Namen *Bool* und zwei direkten Zugriffen mit den Bezeichnungen *true* und *false* auf die beiden Elemente von $\mathbb{B}$.

   Formal lässt sich das in folgender Weise aufschreiben:

   spec BOOL =
   sorts *Bool*
   opns *true*: $\rightarrow$ *Bool*
   false: $\rightarrow$ *Bool*

   Für die beiden direkten Zugriffe *true* und *false*, die als spezielle Operatoren unter dem Schlüsselwort **opns** aufgelistet sind, ist neben ihrem Namen noch angegeben, in welchem Datenbereich sie Werte abliefern bzw. auf Werte zugreifen – nämlich *Bool*. Da die Namen von Datenmengen auch Sorten (oder Typen) genannt werden, ist *Bool* unter **sort** aufgeführt.

   BOOL sieht noch reichlich ärmlich aus; das wird sich später jedoch mit Einführung der Operatoren "nicht", "und", "oder", ... ändern (siehe Beispiel 4.3.3).

2. Der Datentyp der natürlichen Zahlen NAT hat eine unendliche Menge von Daten $\mathbb{N} = \{0, 1, 2, 3, \ldots\}$. Er besitzt in 0 ein ausgezeichnetes, "kleinstes" Element und ist

total geordnet, d.h. jede natürliche Zahl n besitzt in $n+1$ eine direkte Nachfolgerin: $\mathsf{succ}(n) = n + 1$.

Gibt man der Menge der natürlichen Zahlen den Namen *Nat* und der Nachfolgeoperation den Namen *succ*, so lässt sich diese Situation so formalisieren:

spec NAT =
sorts *Nat*
opns $0\colon \to \mathit{Nat}$
$\mathit{succ}\colon \mathit{Nat} \to \mathit{Nat}$

Im Gegensatz zu 0 ist *succ* kein direkter Zugriff auf ein Element, sondern liefert natürliche Zahlen als Werte (*Nat* als Ziel des Pfeiles) abhängig von den natürlichen Zahlen im Argument (*Nat* als Quelle des Pfeiles). Dadurch gelingt eine endliche sprachliche Beschreibung, die bei unendlich vielen Elementen durch direkte Zugriffe nicht möglich ist.

Weitere Operationen auf den natürlichen Zahlen wie Addition und Multiplikation werden später eingeführt (siehe Beispiel 4.3.4).

3. Der Datentyp STRING der Wörter über einem Alphabet A ist etwas komplizierter aufgebaut. Er besteht zum einen aus einer endlichen Menge A mit n Elementen, auf die mit a_1, a_2, ... , a_n direkt zugegriffen werden kann: $A = \{a_1, a_2, \ldots, a_n\}$, zum anderen aus der Menge A^* aller Wörter über A, die rekursiv definiert ist durch:

   (i) Das leere Worte λ ist in A^*;

   (ii) für jeden Buchstaben a in A und jedes Wort w in A^* ist auch aw in A^* (für $a\lambda$ wird oft kurz a geschrieben).

   Durch (ii) wird eine Operation definiert, die jedem Element des Alphabets und jedem Wort w das Wort aw zuordnet. Nennt man sie *insert*, so erhält man folgende formale Gestalt des Datentyps:

   spec STRING =
   sorts A, *String*
   opns $a_i\colon \to A$ für $i = 1, \ldots, n$
   $\lambda\colon \to \mathit{String}$
   $\mathit{insert}\colon A \times \mathit{String} \to \mathit{String}$

   Unterschiedlich zu den Beispielen BOOL und NAT ist, dass es mehrere Sorten (A und *String*) gibt und dass die Operation *insert* mehrere Argumente hat (aufgelistet vor dem Pfeil).

Diese drei Beispiele und ganz analog die anderen genannten Datentypen haben einige gemeinsame Merkmale: Sie besitzen Daten, die auf eine oder mehrere Datenmengen verteilt sind, und um auf die Daten zugreifen und sie verändern zu können, gibt es Operationen. In der Mathematik werden solche Gesamt- und Einheiten aus Daten und Operationen

Algebren genannt (und insbesondere in der universellen Algebra untersucht). Diese Erkenntnis berechtigt zu der

These 2
(Abstrakte) Datentypen sind Algebren.

Es lohnt sich angesichts dieser beiden Thesen also zu fragen, ob und welche Konzepte in der Theorie der universellen Algebren bekannt sind, die sich sowohl zur Präzisierung von Syntax und Semantik abstrakter Datentypen als auch zur Einführung eines Korrektheitsbegriffs und entsprechender Beweisprinzipien eignen. Die universelle Algebra verspricht, bezüglich Datentypen eine gemeinsame Sprachebene für die Spezifikation von Problemen (und Anforderungen) sowie für die Implementierung der sie lösenden Algorithmen zu bilden und zusätzlich als mathematische Theorie Korrektheitsbeweise zuzulassen, ja sogar zu unterstützen.

Alle Begriffsbildungen und Ergebnisse werden ausführlich an den Wahrheitswerten, den natürlichen und ganzen Zahlen, den Wörtern über und Teilmengen aus einem Alphabet A und den binären Bäumen und ähnlichen Beispielen erläutert. Das sind zwar typische Datentypen, und sie bilden den Ausgangspunkt für viele andere Datentypen, sie sind allerdings alle verhältnismäßig "klein" und deshalb möglicherweise als Beispiele für den Entwurf von Softwaresystemen nicht voll überzeugend. Mit den hier entwickelten Methoden können aber auch aus der Sicht der Spezifikationstechniken aussagekräftigere und instruktivere Datentypen wie die Symboltabellen eines Compilers, ein "security file system" als Teil eines Betriebssystems, ein spezielles Datenbanksystem "airport schedule" sowie weitere ähnliche Beispiele behandelt werden.

2 Spezifikation von Algebren durch Signaturen

In diesem Abschnitt sollen die bereits für die Beispiele der Wahrheitswerte, der natürlichen Zahlen und der Wörter über einem Alphabet intuitiv und informell verwendeten Konzepte zur Spezifikation und Behandlung von Algebren eingeführt werden. Das geschieht in zwei Schritten.

Zuerst werden Möglichkeiten geschaffen, Namen für Datenmengen und Namen, Argument- und Wertebereiche für Operationen zu vereinbaren. Diese Deklaration von Sorten und Operationssymbolen bildet den Begriff der Signatur (Definition 2.1). Auf dieser Stufe wird die Form von Algebren festgelegt. Es handelt sich um eine rein syntaktische Ebene. Das kommt auch dadurch zum Ausdruck, dass sich eine Grammatik (in einer Backus-Naur-ähnlichen Form) angeben lässt (Bemerkung 2.3), die alle Signaturen mit endlich vielen Sorten und Operationen analog zur formalen Beschreibung von BOOL, NAT und STRING in Beispiel 1.1 erzeugt. Diese Darstellung von Signaturen wird im folgenden für alle Beispiele verwendet. Sie bildet darüber hinaus den syntaktischen Kern von algebraischen Spezifikationssprachen wie beispielsweise CASL (siehe `http://www.brics.dk/Projects/CoFI/Documents/CASL/Summary/`).

In einem zweiten semantischen Schritt können nun die Algebren definiert werden (Definition 2.7), die der durch eine Signatur vorgegebenen Form genügen, indem sie korrespondierend zu Sorten und Operationssymbolen Datenmengen und Operationen besitzen. Dieser Zusammenhang von Signatur und Algebra ist in Beispiel 1.1 beispielhaft bereits im Verhältnis von NAT zu $\mathbb{N}$ und STRING zu A^* vorgekommen. Es gibt allerdings (fast) immer unendlich viele Algebren, die zu einer Signatur gehören, weil Benennungen von Daten und Operationen diese nicht inhaltlich festlegen.

Die Kopplung von Signatur und zugehörigen Algebren grenzt jedoch den Bereich ein, in dem im Sinne der These 2 nach einem Kandidaten gesucht werden kann, der sich als abstrakter Datentyp und damit als Semantik zur Spezifikation von Algebren und Datentypen durch Signaturen eignet. Ergebnis dieser Suche ist die aus einer Signatur erzeugte Termalgebra (Definitionen 2.10 und 2.16), die sowohl die Anforderungen erfüllt, die an abstrakte Datentypen gestellt werden müssen (Theoreme 2.25 und 2.28) als auch mathematische Eigenschaften besitzt, die sie unter allen Algebren auszeichnet und für Korrektheitsbeweise prädestiniert (Theorem 2.20).

2.1 Definition (Signaturen)
Eine *Signatur* $SIG = <S, OP>$ besteht aus einer Menge S, deren Elemente *Sorten* genannt werden, und aus einer Mengenfamilie $OP = (OP_{w,s})_{w \in S^*, s \in S}$ (für S^* siehe Beispiel 1.1.3).

Dabei ist für jedes $w \in S^*$ und $s \in S$ $OP_{w,s}$ eine Menge, deren Elemente *op* als *Operationssymbole* (auch *Operatoren* und – wenn keine Verwechslungen möglich – als *Operationen*) bezeichnet werden.

Für jedes Operationssymbol $op \in OP_{w,s}$ legt w die *Argumente* und s den *Wertebereich* von *op* fest. w wird auch die *Stelligkeit* von *op* genannt; insbesondere heißt eine Operation *0-stellig* oder *Konstante*, wenn $w = \lambda$.

2.2 Beispiel
1. Die Signatur BOOL besitzt eine einelementige Sortenmenge $S = \{Bool\}$ und von der Mengenfamilie der Operationssymbole sind alle Mengen $OP_{w,Bool}$ leer außer die mit dem leeren Wort λ als Argument: $OP_{\lambda,Bool} = \{true, false\}$ und $OP_{w,Bool} = \emptyset$ sonst.
2. Bei NAT ist $S = \{Nat\}$, $OP_{\lambda,Nat} = \{0\}$, $OP_{Nat,Nat} = \{succ\}$ und $OP_{w,Nat} = \emptyset$ sonst.
3. Bei STRING ist $S = \{A, String\}$ zweielementig. Die Mengen der Operationssymbole sind für diese Signatur definiert durch:
   - $OP_{\lambda,A} = \{a_1, \ldots, a_n\} = A$,
   - $OP_{\lambda,String} = \{\lambda\}$,
   - $OP_{A\ String,String} = \{insert\}$ und
   - $OP_{w,s} = \emptyset$ sonst.
4. Als neues Beispiel wird die Signatur der binären Bäume angegeben.

   Die Sortenmenge von BINTREE ist $S = \{BinTree\}$. Die Mengen der Operationssymbole sind

- $OP_{\lambda, BinTree} = \{leaf\}$, was anschaulich einem Knoten ohne linkes und rechtes Kind entspricht;
- $OP_{BinTree, BinTree} = \{left, right\}$, wobei *left* (*right*) anschaulich an einen Knoten ein linkes (rechtes) Kind – das ist ein binärer Baum – anhängt, jedoch kein rechtes (linkes);
- $OP_{BinTree\,BinTree, BinTree} = \{both\}$, wodurch an einen Knoten sowohl ein linkes als auch ein rechtes Kind gehängt wird.
- Ansonsten sind die $OP_{w, BinTree}$ leer.

Man sieht an diesen Beispielen sehr deutlich, dass die für die allgemeine Situation sehr knappe mathematische Darstellung von Signaturen durch $< S, OP >$ in konkreten Fällen durch die vielen Mengenklammern, durch die häufige Wiederholung des Zeichens *OP* und durch die explizite Nennung der leeren Mengen länglich wird.

2.3 Bemerkung (Signaturen)
Für die Beispiele wird daher eine bequemere Art der Signaturangabe gewählt, die – grob gesprochen – dadurch entsteht, dass alle Mengenklammern weggelassen sowie *S* und *OP* durch mnemotechnisch geeignetere Ausdrücke ersetzt werden. Die Signatur erhält einen in Kapitälchen gesetzten Namen hinter dem Schlüsselwort **spec**. Die Sorten werden kursiv hinter dem Schlüsselwort **sorts** aufgelistet. Die Operationssymbole werden hinter dem Schlüsselwort **opns** aufgelistet, wobei statt $c \in OP_{\lambda,s}$ nun $c: \to s$ und für $op \in OP_{s_1 \ldots s_n, s}$, $n \neq 0$, $op: s_1 \times \cdots \times s_n \to s$ geschrieben wird. Formal lassen sich Signaturen in dieser Form durch folgende Grammatik erzeugen:

<sign> := **spec** <specname> = <sorts> <opns>
<specname> := <identifier>
<sorts> := **sorts** list1-of-<sortname>
<sortname> := <identifier>
<opns> := **opns** list1-of-<opnsymbol>
<opnsymbol> := <identifier>: list2-of-<sortname> → <sortname>

Dabei sind die Ausdrücke in den spitzen Klammern nichtterminale Zeichen, für das Nonterminal < identifier > kann eine beliebige erzeugende Grammatik gewählt werden, und das Konstrukt "list-of" erlaubt, beliebige Ketten des dahinterstehenden Nonterminals zu erzeugen. Eine "1" hinter "list" bedeutet, dass die Elemente durch Kommata oder Zeilenwechsel getrennt werden. Eine "2" hinter "list" bedeutet, dass die Elemente durch "×" getrennt werden. Die übrigen Zeichen sind terminal: **spec**, =, **sorts**, **opns**, :, →.

2.4 Beispiel
1. Bool, Nat und String aus Beispiel 1.1 sind Signaturen in der verkürzten Darstellung.
2. Die entsprechend geänderte Signatur für binäre Bäume hat die Gestalt:

spec BINTREE =
sorts *BinTree*
opns *leaf*: $\rightarrow$ *BinTree*
left: *BinTree* $\rightarrow$ *BinTree*
right: *BinTree* $\rightarrow$ *BinTree*
both: *BinTree* $\times$ *BinTree* $\rightarrow$ *BinTree*

2.5 Bemerkung
Sehr anschaulich lassen sich Signaturen auch graphisch darstellen. Dabei werden die Sorten als Knoten und die Operationssymbole als Kanten gewählt. Das Ziel einer Kante ist immer der zugehörige Wertebereich des Operationssymbols. Im Gegensatz zu herkömmlichen Graphen haben die Kanten aber nicht immer genau eine Quelle, da die Operationssymbole auch nicht immer nur ein Argument haben, sondern alle Argumentsorten bilden die Quellen der Kante. Hat das Operationssymbol jedoch keine Argumente ($op \in OP_{\lambda,s}$), so besitzt auch die zugehörige Kante keine Quelle. Wie das zu verstehen ist, wird an den Graphen für die vier Beispiele klar:

Eine formale Definition der Signaturgraphen ist überflüssig, da mit ihr im folgenden nicht gearbeitet wird.

Nun wieder anknüpfend an die Definition 2.1, die immer herangezogen wird, wenn Signaturen allgemein behandelt oder verwendet werden, können die durch eine Signatur festgelegten, spezifizierten Algebren definiert werden (Definition 2.7). Da allen weiteren Überlegungen dieses Kapitels eine beliebige, aber fest gewählte Signatur zugrunde liegt, wird vereinbart:

2.6 Generalvoraussetzung
$SIG = <S, OP>$ sei eine Signatur im Sinne der Definition 2.1.

2.7 Definition (*SIG*-Algebren)
Eine *SIG-Algebra* A (auch *OP-Algebra* oder *Algebra vom Typ SIG*) besteht aus:

(i) *Datenmengen* A_s für alle $s \in S$, das ist eine Mengenfamilie $(\mathsf{A}_s)_{s \in S}$,

(ii) *ausgezeichneten Elementen* $c_{\mathsf{A}} \in \mathsf{A}_s$ für alle $c \in OP_{\lambda,s}$, $s \in S$,

(iii) *Operationen* $op_{\mathsf{A}} \colon \mathsf{A}_{s_1} \times \mathsf{A}_{s_2} \times \cdots \times \mathsf{A}_{s_n} \to \mathsf{A}_s$ für alle $op \in OP_{s_1 \ldots s_n, s}$; $s_i, s \in S$; $i = 1, \ldots, n$; $n \geq 1$.

2.8 Bemerkung
$\mathsf{A}_{s_1} \times \cdots \times \mathsf{A}_{s_n}$ ist das kartesische Produkt der zu den Sorten s_i, $i = 1, \ldots, n$ gehörenden Datenmengen. Operationen sind mengentheoretische Abbildungen; d.h. für alle $a_i \in \mathsf{A}_{s_i}$, $i = 1, \ldots, n$ ist $op_{\mathsf{A}}(a_1, \ldots, a_n)$ ein Wert in A_s.

Im folgenden werden oft auch die ausgezeichneten Elemente in (ii) als Operationen bezeichnet. Das ist deshalb gerechtfertigt, weil jedes Element a einer Menge A genau eine Abbildung $\bar{a} : \{*\} \to A$ definiert durch $\bar{a}(*) = a$ und umgekehrt. In diesem Sinne ist (ii) ein Spezialfall von (iii), wenn man dort $n = 0$ zulässt.

Statt Datenmengen ist auch die Bezeichnung Basis- oder Trägermengen für die A_s in (i) gebräuchlich.

2.9 Beispiel

1. Eine bereits aus Beispiel 1.1 bekannte BOOL-Algebra bilden die Booleschen Werte:

   $$\mathsf{BOOL} = (\{\mathsf{true}, \mathsf{false}\}, \mathsf{true}, \mathsf{false}).$$

   Ein weiteres Beispiel einer BOOL-Algebra ist:

   $$(\mathbb{N}, 0, 1) \quad \text{(vgl. Beispiel 1.1.2).}$$

   Analog ist jede Menge mit zwei ausgezeichneten Elementen (die nicht verschieden sein müssen) eine Algebra von Typ BOOL.

2. Bereits aus Beispiel 1.1.2 ist die NAT-Algebra der natürlichen Zahlen bekannt:

   $$\mathsf{NAT} = (\mathbb{N}, 0, \mathsf{succ})$$

   mit $\mathsf{succ}(n) = n + 1$ für alle $n \in \mathbb{N}$.
   Zwei weitere Algebren dieses Typs können mit Hilfe der ganzen Zahlen definiert werden:

   $$(\mathbb{Z}, 0, \mathsf{succ}) \quad \text{und} \quad (\mathbb{Z}, 0, \mathsf{pred})$$

   mit $\mathbb{Z} = \{\ldots, -2, -1, 0, +1, +2, \ldots\}$, $\mathsf{succ}(z) = z + 1$ und $\mathsf{pred}(z) = z - 1$ für alle $z \in \mathbb{Z}$.

Analog bildet jede Menge M mit einem ausgezeichneten Element $x \in M$ und einer Abbildung $f\colon M \to M$ von der Menge auf sich als *succ*-Operation eine NAT-Algebra (M, x, f).

Anmerkung zur Schreibweise

Welche Datenmenge zu welcher Sorte und welches ausgezeichnete Element oder welche Operation zu welchem Operationssymbol gehört, wird durch die Reihenfolge von Sorten und Operationssymbolen bestimmt, die durch die Listendefinition list1-of-<sortname> und list1-of-<opnsymbol> in Bemerkung 2.3 gegeben ist.

Es mag darüber hinaus im ersten Moment etwas überraschen, dass die Datenmengen und Operationen andere Bezeichnungen tragen können, als die Sorten und Operationssymbole der Signatur vorgeben. Das wird dann jedoch verständlicher, wenn man sich Sorten und Operationssymbole als formale Parameter denkt, die durch die Datenmengen und Operationen aktualisiert werden. Lediglich die Zuordnung wird nicht explizit vorgenommen, sondern ist durch die Reihenfolge in der Signatur festgeschrieben.

Als Beispiel ist für die obigen NAT-Algebren die Korrespondenz zwischen formalen und aktuellen Bezeichnern in der folgenden Tabelle angegeben:

	Nat	0	*succ*
1	$\mathbb{N}$	0	succ
2	$\mathbb{Z}$	0	succ
3	$\mathbb{Z}$	0	pred
4	M	x	f

3. Eine STRING-Algebra ist durch A^* aus Beispiel 1.1.3 gegeben:

$$A^* = (A, A^*, a_1, \ldots, a_n, \lambda, \mathsf{insert})$$

mit $\mathsf{insert}(a, w) = aw$ für alle $a \in A$, $w \in A^*$.

Aber auch die natürlichen Zahlen lassen sich als STRING-Algebra auffassen:

$$(\{*\}, \mathbb{N}, \underbrace{*, \ldots\ldots, *}_{n-\text{mal}}, 0, \mathit{insert}_{\mathbb{N}})$$

mit $\mathit{insert}_{\mathbb{N}}(*, n) = \mathsf{succ}(n)$ für alle $n \in \mathbb{N}$.

4. Die Menge B der binären Bäume lässt sich folgendermaßen rekursiv definieren:
   (i) $(\,,\,) \in B$ (ein Knoten ohne Kinder).
   (ii) mit $t, t' \in B$ sind auch
   $(t,\,) \in B$ (ein Knoten mit linkem, ohne rechtes Kind)
   $(\,, t) \in B$ (ein Knoten mit rechtem, ohne linkes Kind)
   $(t, t') \in B$ (ein Knoten mit zwei Kindern)

Das erlaubt, folgende BinTree-Algebra zu definieren:

$$\mathsf{BINTREE} = (B,\ (\ ,\ ),\ \mathsf{left},\ \mathsf{right},\ \mathsf{both})$$

mit $\mathsf{left}(t) = (t,\ )$, $\mathsf{right}(t') = (\ , t')$ und $\mathsf{both}(t, t') = (t, t')$ für alle $t, t' \in B$.

Aber auch die natürlichen Zahlen geben Anlass zu einer BinTree-Algebra, wobei man die *left*- und *right*-Operation als succ wählt und *both* als Addition:

$$(\mathbb{N}, 0, \mathsf{succ}, \mathsf{succ}, +).$$

Bei allen Beispielen fällt auf (bzw. sollte auffallen), dass das jeweils erste nicht nur wegen der Namensgebung sehr genau zur vorgegebenen Signatur passt und sozusagen die erwartete Algebra, der intendierte Datentyp ist, während die anderen Beispielalgebren etwas konstruiert, unnatürlich, gezwungen wirken. Das ist kein Zufall. Dem Phänomen aber auf den Grund zu gehen, die ausgezeichnete Rolle jeweils einer Algebra mathematisch zu präzisieren, erfordert einige zusätzliche Überlegungen, die zur Konstruktion und Untersuchung der sogenannten Termalgebren führen.

Die Signatur legt für jede *SIG*-Algebra A die Form, die Struktur fest. Sie kann aber auch als eine Art "Benutzungsschnittstelle" der *SIG*-Algebra gedeutet werden, mit deren Hilfe "von außen" auf Daten von A zugegriffen werden kann. Zum einen ist für jedes $c \in OP_{\lambda,s}$ ($s \in S$) der direkte Zugriff auf das ausgezeichnete Element c_{A} von A möglich. Zum anderen erhält man für $op \in OP_{s_1 \ldots s_n, s}$ einen Zugriff auf den Wert $op_{\mathsf{A}}(x_1, \ldots, x_n)$, wenn man bereits Zugriff auf die Argumente x_i, $i = 1, \ldots, n$ hat.

Durch wiederholten Aufruf der Operationssymbole lassen sich also Daten jeder zugehörigen Algebra erzeugen. In manchen Fällen erhält man auf diese Weise alle Daten. Bei BOOL aus Beispiel 2.9.1 liefern die Aufrufe von true und false gerade die beiden Elemente gleichen Namens; bei $\mathbb{N}$ aus Beispiel 2.9.2 kann auf die 0 direkt zugegriffen werden, während man alle anderen natürlichen Zahlen durch mehrmaliges Anwenden von succ erhält.

In anderen Fällen erzeugen die Operationen nur einen Teil der Daten. Bei $(\mathbb{Z}, 0, \mathsf{succ})$ aus Beispiel 2.9.2 etwa werden durch die Aufrufe von 0 und succ lediglich die nicht-negativen ganzen Zahlen erreicht.

Den zusammengesetzten Aufrufen der Operationssymbole, die oft auch Terme, Ausdrücke, Formeln und ähnlich genannt werden, kommt also eine besondere Bedeutung zu. Sie lassen sich folgendermaßen exakt definieren.

2.10 Definition (*SIG*-Terme)
Für jede Sorte $s \in S$ ist die Menge $T_{SIG,s}$ der *SIG-Terme* zur Sorte s rekursiv gegeben durch:

(i) $OP_{\lambda,s} \subseteq T_{SIG,s}$ für alle $s \in S$;
(ii) $op \in OP_{s_1 \ldots s_n, s}$, $t_i \in T_{SIG,s_i}$, $i = 1, \ldots, n$ impliziert $op(t_1, \ldots, t_n) \in T_{SIG,s}$.

2.11 Bemerkung
Das Präfix "*SIG*-" bei diesem und anderen Begriffen wird im folgenden häufig weggelassen, wenn aufgrund des Zusammenhangs Missverständnisse nicht zu erwarten sind.

2.12 Beispiel

1. Die einzigen BOOL-Terme sind: *true* und *false*.
2. Die NAT-Terme sind:
$$0, succ(0), succ(succ(0)), \ldots$$
was man auch so schreiben kann:
$$succ^n(0) \text{ für alle } n \in \mathbb{N},$$
wobei $succ^0(0) = 0$ und $succ^{n+1}(0) = succ(succ^n(0))$ gilt.
3. Für die Signatur STRING gibt es Terme zweier Sorten:
   - a_i, $i = 1, \ldots, n$, haben die Sorte A,
   - λ, $insert(a_{i_1}, \lambda)$, $insert(a_{i_2}, insert(a_{i_1}, \lambda))$, ... gehören zur Sorte *String*, wobei $a_{i_1}, a_{i_2}, \ldots$ beliebig aus A gewählt werden können.
4. Dem binären Baum in der bildlichen Darstellung

entspricht der BINTREE-Term:
$$both(both(right(both(leaf, leaf)), leaf), leaf).$$

Mit Hilfe der Definition 2.10 lässt sich nun auch präzisieren, wie die Terme als Operationsaufrufe in *SIG*-Algebren auf Daten zugreifen: durch Übergang von den formalen Operationssymbolen zu den zugeordneten tatsächlichen Operationen der Algebra definiert jeder Term in jeder Algebra einen Wert. Diese Interpretation der Terme in den Algebren wird durch die Interpretationsfunktionen vorgenommen, deren Werte rekursiv über den Aufbau der Terme gemäß Definition 2.10 berechnet werden.

2.13 Definition (Interpretationsfunktionen)
Sei A eine *SIG*-Algebra. Dann ist für jedes $s \in S$ die *Interpretation(sfunktion)*
$$eval_s : T_{SIG,s} \to \mathsf{A}_s$$
rekursiv gegeben durch:

(i) für $c \in OP_{\lambda,s}$, $s \in S$ sei $eval_s(c) := c_{\mathsf{A}}$;

(ii) für $op \in OP_{s_1 \ldots s_n,s}$, $t_i \in T_{SIG,s_i}$, $i = 1, \ldots, n$ sei

$$eval_s(op(t_1, \ldots, t_n)) := op_{\mathsf{A}}(eval_{s_1}(t_1), \ldots, eval_{s_n}(t_n)).$$

Die Bezeichnung "Funktion" rechtfertigt folgendes Lemma.

2.14 Lemma
Die Interpretationen $eval_s : T_{SIG,s} \to \mathsf{A}_s$ sind für alle $s \in S$ Abbildungen.

Beweis.
Es ist zu zeigen, dass für jeden Term $t \in T_{SIG,s}$ $eval_s(t)$ genau ein Element aus A_s bestimmt.

Das wird durch Induktion über den Aufbau der Terme bzw. durch Induktion über die Zahl der in den Termen vorkommenden Operationssymbole gezeigt.

Für $c \in OP_{\lambda,s}$ ist diese Zahl 1. Ferner ist c_{A} gerade ein ausgezeichnetes Element, so dass die Behauptung für den Term c gilt. Das dient als Induktionsanfang.

Sei nun $op \in OP_{s_1 \ldots s_n,s}$ und $op(t_1, \ldots, t_n)$ (mit $t_i \in T_{SIG,s_i}$, $i = 1, \ldots, n$) ein Term, in dem k Operationssymbole vorkommen. Sei weiterhin als Induktionsvoraussetzung $eval_s(t)$ für alle $t \in T_{SIG,s}$, $s \in S$ mit weniger als k Operationssymbolen genau ein Element in A_s. Dann gilt dies insbesondere für die Terme t_i. Da außerdem op_{A} als Operation eine Abbildung ist, bestimmt auch

$$eval_s(op(t_1, \ldots, t_n)) = op_{\mathsf{A}}(eval_{s_1}(t_1), \ldots, eval_{s_n}(t_n))$$

genau ein Element von A_s.

Da alle Terme nach Definition entweder aus $OP_{\lambda,s}$ sind oder die Form $op(t_1, \ldots, t_n)$ haben, ist damit alles gezeigt. □

2.15 Beispiel

1. Für die BOOL-Algebra $(\mathbb{N}, 0, 1)$ liefert die Interpretationsfunktion *eval* (der Index *Bool* wird weggelassen, da es nur eine Sorte gibt):

$$eval(true) = 0 \text{ und } eval(false) = 1.$$

2. Für die NAT-Algebra NAT ordnet die Interpretationsfunktion *eval* jedem NAT-Term gerade die Zahl der darin vorkommenden *succ* zu:

$$eval(succ^n(0)) = n \text{ für alle } n \in \mathbb{N}.$$

Für die NAT-Algebra $(\mathbb{Z}, 0, \mathsf{pred})$ ergibt die Interpretation $eval'$ die entsprechenden negativen ganzen Zahlen:

$$eval'(succ^n(0)) = -n \text{ für alle } n \in \mathbb{N}.$$

3. Bei der STRING-Algebra A^* nehmen die Interpretationen folgende Zuordnung vor:
   - $eval_A(a_i) = a_i$ für $i = 1, \ldots, n$,
   - $eval_{String}(\lambda) = \lambda$ und
   - $eval_{String}(insert(a, t)) = aw$ für alle $a \in A$ und *String*-Terme t, wenn $eval_{String}$ den Term t auf das Wort w abbildet.

   $eval_{String}$ listet also für jedenTerm von links nach rechts die darin vorkommenden Terme der Sorte A auf, die ja gleichzeitig Buchstaben des Alphabets A sind.

   Auch für die zweite STRING-Algebra in Beispiel 2.9.3 hat die Interpretation in der Sorte *String* eine interessante Bedeutung. Sie ordnet jedem Term die Zahl der darin vorkommenden Buchstaben aus A zu und damit gerade die Länge des Wortes, auf das dieser Term durch das obige $eval_{String}$ abgebildet wird.
4. Entsprechend dem rekursiven Aufbau der BINTREE-Terme einerseits und der binären Bäume andererseits (vgl. Definition 2.10 und Beispiel 2.9.4) ist die Interpretation gegeben durch:
   - $eval(leaf) = (\,,\,)$,
   - $eval(left(t)) = (b,\,)$,
   - $eval(right(t')) = (\,, b')$ und
   - $eval(both(t, t')) = (b, b')$,

   wenn t bzw. t' durch $eval$ als b bzw. b' interpretiert werden.

   Für den BINTREE-Term aus Beispiel 2.12.4 bedeutet das:

$$eval(both(both(right(both(leaf, leaf)), leaf), leaf)) = (((\,, ((\,,\,), (\,,\,))), (\,,\,)), (\,,\,)).$$

Die *SIG*-Terme sind unabhängig von *SIG*-Algebren rein syntaktisch definiert, greifen aber vermittels der Interpretationsfunktionen auf diejenigen Daten der *SIG*-Algebren zu, die durch die Operationen erzeugt werden. Ihre Bedeutung rührt also daher, dass sie den "von außen" zugänglichen Teil aller *SIG*-Algebren formal beschreiben. Der Wert der *SIG*-Terme wird dadurch erhöht, dass sie selbst die Datenmengen einer *SIG*-Algebra, der sogenannten Termalgebra, bilden und dass die Interpretationsfunktionen mit der Struktur dieser Algebra verträglich sind (Definition 2.16 und Beobachtung 2.17).

2.16 Definition (*SIG*-Termalgebra)
Die *SIG-Algebra* T_{SIG} ist gegeben durch

(i) die Datenmengen $T_{SIG,s}$ aller *SIG*-Terme für alle $s \in S$,

(ii) die ausgezeichneten Elemente $c_T := c$ für alle $c \in OP_{\lambda,s}$, $s \in S$,

(iii) die Operationen $op_T \colon T_{SIG,s_1} \times \cdots \times T_{SIG,s_n} \to T_{SIG,s}$ mit

$$op_T(t_1, \ldots, t_n) := op(t_1, \ldots, t_n)$$

für alle $op \in OP_{s_1 \ldots s_n, s}$, $t_i \in T_{SIG,s_i}$, $i = 1, \ldots, n$.

2.17 Beobachtung

1. *T_{SIG} ist eine SIG-Algebra.*
2. *Sei A eine SIG-Algebra. Dann ist die Familie der Interpretationen*

$$eval = (eval_s : T_{SIG,s} \to \mathsf{A}_s)_{s \in S}$$

mit der Struktur von T_{SIG} und A im folgenden Sinne verträglich:

(i) $eval_s(c_T) = c_{\mathsf{A}}$ *für alle* $c \in OP_{\lambda,s}$, $s \in S$ *und*

(ii) $eval_s(op_T(t_1, \ldots, t_n)) = op_{\mathsf{A}}(eval_{s_1}(t_1), \ldots, eval_{s_n}(t_n))$ *für alle* $op \in OP_{s_1 \ldots s_n, s}$, $t_i \in T_{SIG,s_i}$, $i = 1, \ldots, n$.

Beweis.

1. Wegen $OP_{\lambda,s} \subseteq T_{SIG,s}$ ist Definition 2.16.(ii) korrekt, und nach Definition der Terme ist op_T in Definition 2.16.(iii) eine Abbildung der angegebenen Form.
2. Die Behauptung folgt direkt aus der Definition der Operationen op_T und der Abbildungen $eval_s$:

   (i) $eval_s(c_T) = eval_s(c) = c_{\mathsf{A}}$

   (ii) $eval_s(op_T(t_1, \ldots, t_n)) = eval_s(op(t_1, \ldots, t_n)) = op_{\mathsf{A}}(eval_{s_1}(t_1), \ldots, eval_{s_n}(t_n))$

□

2.18 Beispiel

1. Für BOOL stimmt die Termalgebra genau mit BOOL aus Beispiel 2.9.1 überein (vgl. Beispiel 2.12.1).
2. T_{NAT} hat als Datenmenge $\{succ^n(0) \mid n \in \mathbb{N}\}$ (vgl. Beispiel 2.12.2), und die Operationen sind: $0_T = 0$ und $succ_T(succ^n(0)) = succ^{n+1}(0)$ für alle $n \in \mathbb{N}$.

   Die Strukturverträglichkeit der Interpretation gemäß Beobachtung 2.17.2 kann in diesem Fall leicht noch einmal explizit nachgewiesen werden:

$$eval(0_T) = eval(0) = 0$$

und für alle $n \in \mathbb{N}$:

$$\begin{aligned} eval(succ_T(succ^n(0))) &= eval(succ^{n+1}(0)) \\ &= n + 1 \\ &= \mathsf{succ}(n) \\ &= \mathsf{succ}(eval(succ^n(0))) \end{aligned}$$

Ganz analog erhält man die Strukturverträglichkeit bei der NAT-Algebra $(\mathbb{Z}, 0, \mathsf{pred})$:

$$eval'(0_T) = eval'(0) = 0$$

und für alle $n \in \mathbb{N}$:

$$\begin{aligned} eval'(succ_T(succ^n(0))) &= eval'(succ^{n+1}(0)) \\ &= -(n+1) \\ &= -n-1 \\ &= \mathsf{pred}(-n) \\ &= \mathsf{pred}(eval'(succ^n(0))) \end{aligned}$$

Da im folgenden Strukturverträglichkeiten, wie sie die Interpretationsfunktionen besitzen, noch häufig vorkommen und verwendet werden, erhalten Abbildungen mit dieser Eigenschaft einen besonderen Namen und werden als eigenes Konstrukt eingeführt: der Homomorphismus (Definition 2.19). Mit Hilfe dieses Begriffs kann dann auch die Sonderstellung der Termalgebra unter den Algebren charakterisiert werden (Theorem 2.20), wobei die Beobachtung 2.17.2 wesentlich verschärft wird.

2.19 Definition (*SIG*-Homomorphismus)
Seien A und A' *SIG*-Algebren. Dann wird eine Familie von Abbildungen

$$h = (h_s : \mathsf{A}_s \to \mathsf{A}'_s)_{s \in S}$$

SIG-Homomorphismus von A in A' (in Zeichen: $h : \mathsf{A} \to \mathsf{A}'$) genannt, wenn gilt:

(i) $h_s(c_{\mathsf{A}}) = c_{\mathsf{A}'}$ für alle $c \in OP_{\lambda,s}$, $s \in S$

(ii) $h_s(op_{\mathsf{A}}(a_1, \ldots, a_n)) = op_{\mathsf{A}'}(h_{s_1}(a_1), \ldots, h_{s_n}(a_n))$ für alle $op \in OP_{s_1 \ldots s_n, s}$, $a_i \in \mathsf{A}_{s_i}$, $i = 1, \ldots, n$.

2.20 Theorem
Sei T_{SIG} die SIG-Termalgebra. Dann ist für jede SIG-Algebra A die Familie der Interpretation der einzige SIG-Homomorphismus $eval : T_{SIG} \to \mathsf{A}$.

Beweis.
Nach Beobachtung 2.17.2 in Verbindung mit Definition 2.19 bilden die Interpretationen einen *SIG*-Homomorphismus $eval : T_{SIG} \to \mathsf{A}$. Es bleibt zu zeigen, dass es keinen anderen gibt. Sei dazu $h : T_{SIG} \to \mathsf{A}$ ein beliebiger *SIG*-Homomorphismus. Es muss nun gezeigt werden, dass $eval = h$ gilt, was für Abbildungsfamilien bedeutet $eval_s = h_s$ für alle $s \in S$. Dieser Beweis lässt sich durch Induktion über den Aufbau der Terme führen:

(i) Für $c \in OP_{\lambda,s}$, $s \in S$ gilt als Induktionsanfang:

$$\begin{aligned} eval_s(c) &= eval_s(c_T) && \text{(Def. } c_T\text{)} \\ &= c_{\mathsf{A}} && (eval \text{ Hom.}) \\ &= h_s(c_T) && (h \text{ Hom.}) \\ &= h_s(c) && \text{(Def. } c_T\text{)} \end{aligned}$$

(ii) Für $op \in OP_{s_1 \ldots s_n, s}$, $t_i \in T_{SIG,s_i}$, $i = 1, \ldots, n$ gilt unter der Induktionsvoraussetzung, dass bereits $eval_{s_i}(t_i) = h_{s_i}(t_i)$ für $i = 1, \ldots, n$ ist:

$$\begin{aligned}
eval_s(op(t_1, \ldots, t_n)) &= eval_s(op_T(t_1, \ldots, t_n)) && \text{(Def. } op_T\text{)} \\
&= op_{\mathsf{A}}(eval_{s_1}(t_1), \ldots, eval_{s_n}(t_n)) && \text{(}eval\text{ Hom.)} \\
&= op_{\mathsf{A}}(h_{s_1}(t_1), \ldots, h_{s_n}(t_n)) && \text{(IV)} \\
&= h_s(op_T(t_1, \ldots, t_n)) && \text{(}h\text{ Hom.)} \\
&= h_s(op(t_1, \ldots, t_n)) && \text{(Def. } op_T\text{)}
\end{aligned}$$

□

Die universelle Algebra hat für die Eigenschaft der Termalgebra in Theorem 2.20 den Begriff der Initialität geprägt.

2.21 Definition (Initiale *SIG*-Algebren)
Eine *SIG*-Algebra $\mathsf{A}^{\S}$ heißt *initial*, wenn für jede *SIG*-Algebra A genau ein *SIG*-Homomorphismus $h\colon \mathsf{A}^{\S} \to \mathsf{A}$ existiert.

Hat $\mathsf{A}^{\S}$ diese Eigenschaft nicht bezüglich aller *SIG*-Algebren, sondern nur für eine Unterklasse C, so heißt $\mathsf{A}^{\S}$ *initial in* C.

Unter Verwendung dieser Definition lässt sich das zentrale Ergebnis dieses Kapitels, das Theorem 2.20, kurz und prägnant formulieren.

2.22 Theorem (Initialität von T_{SIG})
Die Termalgebra T_{SIG} ist initial.

Aus Sicht der Datentypen sind die Terme als formale Operationsaufrufe interessant und die Interpretationen, weil sie den aktuellen Wert dieser Aufrufe berechnen. Die Überlegungen von Definition 2.16 bis Definition 2.21 zeigen, dass sich diese Eigenschaften algebraisch deuten lassen. Die Initialität der Termalgebra erschließt ganz im Sinne der These 1 die mathematische Theorie der universellen Algebra für Korrektheitserwägungen bei Datentypen. Wie, wird sich zeigen (siehe Theorem 2.28 und 4.10).

Die Initialität macht die Termalgebra im wahrsten Sinne des Wortes einmalig. Um das allerdings begrifflich fassen zu können, wird als Spezialfall der Homomorphie die Isomorphie benötigt (Definition 2.23), und einige einfache Eigenschaften vom Homo– und Isomorphismus müssen bewiesen werden (Lemma 2.24). Es ist leicht einzusehen, dass die Termalgebra initial bleibt, wenn man nur die Bezeichnungen einiger Daten ändert. Solche Umbenennungen von Elementen der Datenmenge, ohne dass dabei die Struktur der Algebra verletzt wird, sind gerade die Isomorphismen. Es kann dann festgestellt werden, dass nicht nur die Initialität bei Umbenennung nicht verloren geht, sondern sogar je zwei initiale Algebren durch Umbenennung auseinander hervorgehen (Theorem 2.25).

2.23 Definition (*SIG*-Isomorphismen)
1. Ein *SIG*-Homomorphismus $h\colon \mathsf{A} \to \mathsf{A}'$ ist ein *SIG-Isomorphismus*, wenn für jedes $s \in S$ die Abbildung h_s bijektiv ist.

2. Zwei *SIG*-Algebren A und A' sind *isomorph*, wenn es einen *SIG*-Isomorphismus $h\colon \mathsf{A} \to \mathsf{A}'$ gibt.

2.24 Lemma

1. *Sei A eine SIG-Algebra. Dann bilden die identischen Abbildungen $1_s\colon A_s \to A_s$ für $s \in S$ einen SIG-Isomorphismus $1_{\mathsf{A}}\colon \mathsf{A} \to \mathsf{A}$, die* Identität *auf A.*
2. *Seien $h\colon \mathsf{A} \to \mathsf{A}'$ und $h'\colon \mathsf{A}' \to \mathsf{A}''$ SIG-Homomorphismen. Dann definiert auch die Familie der Hintereinanderschaltungen $h' \circ h = (h'_s \circ h_s)_{s\in S}$ der Abbildungen h_s und h'_s für alle $s \in S$ einen SIG-Homomorphismus $h' \circ h\colon \mathsf{A} \to \mathsf{A}''$.*
3. *Ein SIG-Homomorphismus $h\colon \mathsf{A} \to \mathsf{A}'$ ist genau dann ein SIG-Isomorphismus, wenn ein SIG-Homomorphismus $h'\colon \mathsf{A}' \to \mathsf{A}$ existiert mit $h' \circ h = 1_{\mathsf{A}}$ und $h \circ h' = 1_{\mathsf{A}'}$.*
4. *Wenn h' existiert, ist h' ebenfalls SIG-Isomorphismus, die* Umkehrung *von h, die auch mit h^{-1} bezeichnet wird.*

(*Ohne Beweis.*)

2.25 Theorem (initiale Algebren)

1. *Sei A eine initiale SIG-Algebra und $h\colon \mathsf{A} \to \mathsf{A}'$ ein SIG-Isomorphismus. Dann ist auch A' initial.*
2. *Initiale SIG-Algebren sind isomorph.*

2.26 Bemerkung

Theorem 2.25.2 ist auch für initiale *SIG*-Algebren in einer Unterklasse C von *SIG*-Algebren richtig. Dasselbe gilt für Theorem 2.25.1, wenn A, A' aus C sind.

Beweis.

1. Sei h^{-1} die Umkehrung von h gemäß Lemma 2.24.3 und 2.24.4, B eine beliebige *SIG*-Algebra und $f\colon \mathsf{A} \to \mathsf{B}$ der aufgrund der Initialität von A existierende *SIG*-Homomorphismus. Dann ist auch $f \circ h^{-1}\colon \mathsf{A}' \to \mathsf{B}$ ein *SIG*-Homomorphismus (Lemma 2.24.2). Es bleibt zu zeigen, dass er der einzige ist. Sei dazu $g\colon \mathsf{A}' \to \mathsf{B}$ ein beliebiger Homomorphismus, dann ist auch $g \circ h\colon \mathsf{A} \to \mathsf{B}$ ein Homomorphismus, und wegen der Initialität von A folgt $g \circ h = f$. Daraus ergibt sich mit Lemma 2.24.3 $f \circ h^{-1} = g \circ h \circ h^{-1} = g$, was zu zeigen war.
2. Seien A, A' initiale *SIG*-Algebren. Aufgrund dieser Eigenschaft existieren *SIG*-Homomorphismen $h\colon \mathsf{A} \to \mathsf{A}'$ und $h'\colon \mathsf{A}' \to \mathsf{A}$. Mit Lemma 2.24.2 erhält man daraus die *SIG*-Homomorphismen $h' \circ h\colon \mathsf{A} \to \mathsf{A}$ und $h \circ h'\colon \mathsf{A}' \to \mathsf{A}'$. Nach Lemma 2.24.1 gibt es daneben die Identitäten $1_{\mathsf{A}}\colon \mathsf{A} \to \mathsf{A}$ und $1_{\mathsf{A}'}\colon \mathsf{A}' \to \mathsf{A}'$. Mit der Initialität von A und A' folgt: $h' \circ h = 1_{\mathsf{A}}$ und $h \circ h' = 1_{\mathsf{A}'}$, was nach Lemma 2.24.3 bedeutet, dass h ein *SIG*-Isomorphismus ist, dass also A und A' isomorph sind. □

Die Eigenschaft, initial zu sein, ist somit bis auf Isomorphie eindeutig, hängt also nicht von der konkreten Darstellung der Daten ab. In diesem Sinne ist die Termalgebra als

initiale Algebra "repräsentationsunabhängig". Ferner sind alle Daten, die Terme, "operationserzeugt", denn jeder Term ist sein eigener Operationsaufruf, weil in der Termalgebra die formalen mit den konkreten Operationen übereinstimmen. Diese Eigenschaft der Termalgebra lässt sich sehr einfach präzisieren. (Vgl. dazu die Kommentierung vor Definition 2.10)

2.27 Definition (*SIG*-erzeugt)
Sei A eine *SIG*-Algebra und $eval\colon T_{SIG} \to \mathsf{A}$ die Interpretation. Dann wird A *SIG-erzeugt* genannt, wenn für jedes $s \in S$ und $a \in \mathsf{A}_s$ ein Term $t \in T_{SIG,s}$ existiert mit $eval_s(t) = a$, wenn also die Abbildungen $eval_s$ für alle $s \in S$ surjektiv sind.

2.28 Theorem
Die Termalgebra T_{SIG} ist SIG-erzeugt.

Beweis.
Nach Lemma 2.24.1 ist die Identität $1_T\colon T_{SIG} \to T_{SIG}$ ein *SIG*-Homomorphismus, der wegen der Initialität von T_{SIG} die Interpretation sein muss. Da die identischen Abbildungen aber insbesondere surjektiv sind, ist T_{SIG} *SIG*-erzeugt. □

"Repräsentationsunabhängig" und "operationserzeugt" zu sein, sind aber gerade die beiden zentralen, charakteristischen Anforderungen an abstrakte Datentypen (siehe [DF77, 9.2]). Das rechtfertigt, These 2 des vorigen Abschnitts nahezu auf den Kopf zu stellen.

These 3
Termalgebren als initiale Algebren sind abstrakte Datentypen.

Die Syntax abstrakter Datentypen in Form der Signatur hat damit unter den zugehörigen *SIG*-Algebren in der Termalgebra bzw. allen dazu isomorphen Algebren eine eindeutige Semantik erhalten.

Es lässt sich nun auch aufklären, warum und in welchem Sinne die Algebren BOOL, NAT, STRING und BINTREE "besser" zu den Signaturen Bool, Nat, String bzw. BinTree passen als die anderen Beispiele in 2.9. Sie sind isomorph zu den jeweiligen Termalgebren, also initial, und repräsentieren damit die Semantik der Signaturen.

2.29 Beispiel

1. Wie bereits in Beispiel 2.18.1 bemerkt, ist BOOL die Termalgebra des Typs Bool.
2. Nach Beispiel 2.18.2 ist durch die Interpretation $eval(succ^n(0)) = n$ ein Nat-Homomorphismus $eval\colon T_{\mathrm{NAT}} \to \mathsf{NAT}$ definiert, der ja offenbar auch bijektiv ist. Denn jedes $n \in \mathbb{N}$ hat in $succ^n(0)$ ein Urbild, so dass $eval$ surjektiv ist, und zwei Terme $succ^m(0)$ und $succ^n(0)$ sind genau dann verschieden, wenn $m \neq n$, so dass $eval$ injektiv ist. Somit ist NAT isomorph zu T_{NAT}.

   Es gibt noch eine weitere Möglichkeit, diese Isomorphie zu beweisen. Man zeigt, dass NAT initiale Nat-Algebra ist, so dass die Isomorphie aus Theorem 2.25.2 folgt.

3. Nach Beispiel 2.15.3 ist durch

   - $eval(a_i) = a_i$ für $i = 1, \ldots, n$,
   - $eval(\lambda) = \lambda$ und
   - $eval(insert(a, t)) = a\, eval(t)$ für alle $a \in A$ und Terme t

   der STRING-Homomorphismus $eval\colon T_{\text{STRING}} \to A^*$ gegeben, wobei die Indizes für die Sorten weggelassen sind, da sie durch die Argumente eindeutig rekonstruiert werden können.

   Auf der Sorte A ist $eval$ als Identität bijektiv.

   Auf der Sorte *String* ergibt sich (a) die Surjektivität von $eval$ über die Länge der Wörter $w \in A^*$ (vgl. Beispiel 2.15.3) und (b) die Injektivität durch Induktion über die Zahl der *insert*-Operationen:

   (a) Für das leere Wort λ der Länge 0 ist λ Urbild. Sei nun aw ein Wort der Länge $n+1$ ($a \in A$, $w \in A^*$) und t ein Urbild von w nach Induktionsvoraussetzung. Dann besitzt auch aw ein Urbild:

   $$eval(insert(a, t)) = a\, eval(t) = aw.$$

   (b) Nur der Term λ mit 0 *insert*-Operationen wird auf λ abgebildet, denn alle anderen Bilder haben eine von 0 verschiedene Länge.

   Sei nun t' ein Term mit $n+1$ *insert*-Operationen, und sei $eval$ injektiv auf allen kürzeren Termen (Induktionsvoraussetzung). Sei u' ein weiterer Term mit $eval(u') = eval(t')$. Damit ist $u' \neq \lambda$, und u' sowie t' haben nach Definition die Form

   $$t' = insert(a, t) \text{ und } u' = insert(b, u)$$

   für geeignete $a, b \in A$ und Terme t, u.

   Es folgt:

   $$\begin{aligned} a\, eval(t) = eval(insert(a, t)) &= eval(t') \\ &= eval(u') = eval(insert(b, u)) = b\, eval(u), \end{aligned}$$

   was $a = b$ und $eval(t) = eval(u)$ impliziert. Die Zahl der *insert*-Operationen in t und u entspricht nach Beispiel 2.15.3 der Länge von $eval(t)$ und $eval(u)$, die n ist. Nach Induktionsvoraussetzung ist also $t = u$, so dass auch gilt:

   $$t' = insert(a, t) = insert(b, u) = u'.$$

4. Analog zeigt man die Isomorphie von T_{BINTREE} und BINTREE.

3 Termalgebren mit Variablen

Die im vorigen Abschnitt eingeführten syntaktischen und semantischen Konzepte zur Spezifikation von Algebren und damit von abstrakten Datentypen sind geeignet, um die

Wahrheitswerte, natürlichen Zahlen, Wörter über A und binären Bäume sowie analoge Datentypen zu erzeugen (vgl. Beispiel 2.29). Sie reichen jedoch nicht aus, wenn etwa diese Datentypen mit weiteren Operationen ausgestattet werden sollen oder wenn es gilt, die ganzen Zahlen oder die Potenzmenge über A zu spezifizieren.

3.1 Beispiel

1. Bei der Verwendung von Wahrheitswerten zum Beispiel in Prädikaten und Booleschen Ausdrücken treten neben "wahr" und "falsch" häufig Boolesche Operationen wie "nicht", "und", "oder", ... auf. Konkret kann etwa auf der BOOL-Algebra BOOL eine Operation non definiert werden durch non(true) = false und non(false) = true. BOOL, erweitert durch non, ist nun eine Algebra des Typs

   spec BOOL1 =
   sorts *Bool*
   opns $true, false\colon \to Bool$
   $non\colon Bool \to Bool$

   und repräsentiert nach wie vor den Datentyp, den man haben will. Die BOOL1-Termalgebra T_{BOOL1} jedoch enthält nun die unendlich vielen Terme

   $$non^m(true) \quad \text{und} \quad non^n(false) \quad \text{für alle } m, n \in \mathbb{N},$$

   so dass T_{BOOL1} und BOOL nicht isomorph sind. Mit anderen Worten stimmt die durch BOOL1 spezifizierte Semantik in Form der Termalgebra T_{BOOL1} nicht mit der intendierten Semantik überein. In diesem Sinne ist BOOL1, obwohl syntaktisch korrekt, eine "falsche" Spezifikation. Das ist auch vollkommen klar, weil die Eigenschaft von *non* in BOOL, "wahr" zu "falsch" und "falsch" zu "wahr" zu machen, nicht durch die Form $non\colon Bool \to Bool$, die einzig in BOOL1 festgelegt wird, reflektiert und respektiert ist.

2. Das Rechnen auf natürlichen Zahlen ohne Addition und Multiplikation, nur mit 0 und *succ* ist äußerst unbequem. Deklariert man jedoch beispielsweise die Operation + zusätzlich:

   spec NAT1 =
   sorts *Nat*
   opns $0\colon \to Nat$
   $succ\colon Nat \to Nat$
   $+\colon Nat \times Nat \to Nat$

   so erhält man neue Terme, u.a.

   $$succ^m(0) + succ^n(0) \quad \text{für alle } m, n \in \mathbb{N}.$$

   Interpretiert man + als übliche Addition in $\mathbb{N}$, so ordnet die Interpretationsfunktion $eval\colon T_{\text{NAT1}} \to \mathbb{N}$ ($\mathbb{N}$ ist NAT1-Algebra!) diesen Termen als Wert $m + n$ zu. Also

ist *eval* nicht injektiv, da beispielsweise $3 + 7 = 2 + 8$, aber $succ^3(0) + succ^7(0) \neq succ^2(0) + succ^8(0)$. Die zur Spezifikation NAT1 gehörende Semantik stammt also nicht mit den gewünschten natürlichen Zahlen überein. Das liegt daran, dass die Deklaration von + z.B. die Eigenschaft der Addition in $\mathbb{N}$, dass $m+n = \mathsf{succ}^{m+n}(0)$ ist, nicht berücksichtigt.

3. Nimmt man sich vor, die ganzen Zahlen zu spezifizieren (vgl. Beispiel 2.9.2), so bemerkt man sehr schnell, dass sich alle positiven ganzen Zahlen als Nachfolgerinnen und alle negativen als Vorgängerinnen der Null erzeugen lassen. Das legt folgende Signatur nahe:

   spec INT =
   sorts *Int*
   opns $0\colon \to \mathit{Int}$
   $\mathit{succ}, \mathit{pred}\colon \mathit{Int} \to \mathit{Int}$

   Die ganzen Zahlen bilden tatsächlich eine INT-Algebra:

   $$\mathbb{Z} = (\mathbb{Z}, 0, \mathsf{succ}, \mathsf{pred})$$

   mit $\mathsf{succ}(z) = z + 1$ und $\mathsf{pred}(z) = z - 1$ für alle $z \in \mathbb{Z}$. $\mathbb{Z}$ ist auch INT-erzeugt im Sinne der Definition 2.27, denn unter der Interpretation $eval\colon T_{\text{INT}} \to \mathbb{Z}$ haben die ganzen Zahlen folgende Urbilder:

   - $eval(0) = 0$,
   - $eval(succ^m(0)) = +m$ für alle $m \geq 1$ und
   - $eval(pred^n(0)) = -n$ für alle $n \geq 1$.

   Also ist *eval* surjektiv; es ist jedoch nicht injektiv, denn neben 0, $succ^m(0)$ und $pred^n(0))$ gibt es ja noch viele weitere INT-Terme.
   Beispielsweise wird $pred(succ(pred(0)))$ wie $pred(0)$ als -1 interpretiert:

   $$\begin{aligned} eval(pred(succ(pred(0)))) &= \mathsf{pred}(eval(succ(pred(0)))) \\ &= \mathsf{pred}(\mathsf{succ}(eval(pred(0)))) \\ &= \mathsf{pred}(\mathsf{succ}(\mathsf{pred}(eval(0)))) \\ &= \mathsf{pred}(\mathsf{succ}(\mathsf{pred}(0))) \\ &= \mathsf{pred}(\mathsf{succ}(-1)) \\ &= \mathsf{pred}(-1 + 1) \\ &= -1 + 1 - 1 = -1 \end{aligned}$$

   Es gilt sogar allgemein, dass für $t \in T_{\text{INT}}$ die Interpretationen von t, $succ(pred(t))$ und $pred(succ(t))$ übereinstimmen.

   Die durch INT spezifizierte Termalgebra ist verschieden von der intendierten Semantik $\mathbb{Z}$, weil die Eigenschaft der Operationen succ und pred in $\mathbb{Z}$, sich gegenseitig

aufzuheben, also

$$\mathsf{succ}(\mathsf{pred}(z)) = z \quad \text{und} \quad \mathsf{pred}(\mathsf{succ}(z)) = z \quad \text{für alle } z \in \mathbb{Z},$$

in der Termalgebra nicht erfüllt ist.

4. Sei A eine Menge mit den Elementen $a_1, \ldots, a_n$. Dann lassen sich alle Teilmengen von A, die die Potenzmenge $\mathcal{P}(A)$ bilden, aus der leeren Menge $\emptyset$ durch Hinzunahme einzelner Elemente aus A erzeugen. Daraus resultiert die folgende formale Spezifikation.

   spec SET =
   sorts A, Set
   opns $a_i\colon \to A$ für $i = 1, \ldots, n$
   $\emptyset\colon \to Set$
   $insert\colon A \times Set \to Set$

   Bis auf Umbenennung ist das die Signatur STRING, so dass T_{SET} zu A^* isomorph ist und damit nicht zur intendierten Algebra

   $$\mathcal{P}(A) = (A, \mathcal{P}(A), a_1, \ldots, a_n, \emptyset, \mathsf{ins})$$

   mit $\mathsf{ins}(a, M) = M \cup \{a\}$ für alle $a \in A$, $M \in \mathcal{P}(A)$. Insbesondere sind die Terme $insert(a, M)$ und $insert(a, insert(a, M))$ für beliebige $a \in A$ und Set-Terme M nach Konstruktion verschieden, ihre Interpretation in $\mathcal{P}(A)$ jedoch ist gleich:

   $$\begin{aligned} \mathsf{ins}(a, \mathsf{ins}(a, M)) &= \mathsf{ins}(a, M \cup \{a\}) \\ &= (M \cup \{a\}) \cup \{a\} \\ &= M \cup (\{a\} \cup \{a\}) \\ &= M \cup \{a\} \\ &= \mathsf{ins}(a, M). \end{aligned}$$

Zusammenfassend zeigen die Beispiele, dass die Signaturen als Syntax und die Termalgebren als Semantik konzeptionell nicht ausreichen, um zu bereits definierten Algebren bzw. Datentypen weitere Operationen hinzunehmen zu können noch um andere überhaupt zu definieren. Ursache ist in allen Fällen, dass es nicht möglich ist, Eigenschaften der Operationen auszudrücken und zu fordern. Eine genauere Analyse der Beispiele zeigt, dass jeweils bestimmte Gleichungen, die in den Vergleichsalgebren gelten, von den Termen bzw. Operationen der Termalgebra nicht erfüllt werden. Positiv gewendet, ist neben den Sorten und Operationen ein Konstrukt "Gleichungen" erforderlich. Bevor das jedoch im vierten Kapitel eingeführt werden kann, muss der "Stoff" untersucht werden, aus dem die Gleichungen sind.

Im Beispiel 3.1 kamen folgende – vorläufig also intuitiv verwendete – Gleichungen vor:

$$\begin{aligned} non(true) &= false \\ non(false) &= true \\ succ^m(0) + succ^n(0) &= succ^{m+n}(0) \quad \text{für alle } m, n \in \mathbb{N} \\ succ(pred(z)) &= z \\ pred(succ(z)) &= z \\ insert(a, insert(a, M)) &= insert(a, M) \end{aligned}$$

Gemeinsames Merkmal dieser Gleichungen ist, dass sie aus linken und rechten Seiten bestehen, die in den ersten drei Fällen Terme sind sowie in den letzten drei zu Termen sehr ähnlich, nur dass z und M keine 0-stelligen Operationen in *Int* bzw. *Set* sind. Die Rolle von z und M in diesen Ausdrücken ist die von Variablen. Denn die Gleichung $pred(succ(z)) = z$ gilt beispielsweise in $\mathbb{Z}$, d.h. linke und rechte Seite sind gleich, wenn z die ganzen Zahlen durchläuft und *pred* und *succ* durch die entsprechenden Operationen in $\mathbb{Z}$ ersetzt werden. Analog gilt die letzte Gleichung in $\mathcal{P}(A)$ für alle Belegungen von M durch Teilmengen von A und für alle $a \in A$, was in Beispiel 3.1.4 nachgewiesen ist.

Bevor Gleichungen eingeführt und Algebren untersucht werden können, in denen die gegebenen Gleichungen gelten, muss der Begriff "Term mit Variablen" präzisiert werden (Definition 3.4).

Als Grundbausteine für Gleichungen sind parametrisierte Terme bzw. Terme mit Variablen vor allem deshalb interessant, weil sie einen allgemeineren Datenzugriff erlauben und syntaktisch beschreiben als die bisher bekannten Terme. Durch das zusätzliche Hilfsmittel der Variablen gelingt es, einen parametrisierten Datenzugriff und zusammengesetzte Operationen unabhängig von konkreten Algebren auszudrücken (Theorem 3.5 und Korollar 3.9). Terme mit Variablen als baumartig verzweigte Operationsaufrufe, deren formale Parameter, die Variablen, durch Werte in Algebren aktualisiert werden können, lassen sich als eine einfache Form des Programmierens auf Datentypen (ohne Rekursion und Iteration) deuten. Auch dafür vorweg einige Beispiele.

3.2 Beispiel

1. In den natürlichen Zahlen wird durch zweimaliges Anwenden der Nachfolgerfunktion zu jeder Zahl 2 addiert:

   $$\mathsf{succ}(\mathsf{succ}(n)) = n + 2 \quad \text{für alle } n \in \mathbb{N}.$$

   Behandelt man nun die Variable n wie die 0 der natürlichen Zahlen, so wird das Problem "addiere 2" durch den Term $succ(succ(n))$ formal gelöst.

2. Will man bei der Wortalgebra A^* vor jedes Wort w noch die Buchstaben ab schreiben $(a, b \in A)$, so entspricht dem der formale Ausdruck $insert(a, insert(b, s))$, wobei s eine Variable der Sorte *String* ist. Ersetzt man darin den formalen Parameter s durch ein beliebiges Wort w aus A^* und das formale Operationssymbol *insert* durch

die zugehörige A^*-Operation, wird aus dem Ausdruck das Wort abw. Der Term $insert(a, insert(b, s))$ ist also ein "Programm", das die Abbildung $f: A^* \to A^*$ mit $f(w) = abw$ für alle $w \in A^*$ berechnet.

3. Wie kann man aus zwei beliebigen binären Bäumen b und b' den binären Baum

machen?

Dieses Problem wird von folgendem "Programm" gelöst:

$$both(both(b, b), both(b', b')).$$

Wiederum ist das ein Term, wenn man statt des 0-stelligen Operationssymbols *leaf* auch die beiden Variablen b und b' zulässt.

Das lässt sich verallgemeinern. Man erhält Terme mit Variablen als Terme über einer Signatur, zu der die Variablen als 0-stellige Operationssymbole hinzugenommen werden (Definition 3.4). Wie die einfachen Terme lassen sich Terme mit Variablen in *SIG*-Algebren interpretieren. Sie bilden darüber hinaus ebenfalls die Datenmengen von *SIG*-Algebren, die unter Einbeziehung der besonderen Rolle der Variablen ganz ähnliche Eigenschaften besitzen wie initiale Algebren (Theorem 3.5).

Zur Vereinfachung wird für das restliche Kapitel folgendes vereinbart:

3.3 Generalvoraussetzung
Sei $SIG = <S, OP>$ eine Signatur und $X = (X_s)_{s \in S}$ eine Mengenfamilie, die mit OP keine Elemente gemeinsam hat. X dient als Vorrat an Variablen, sollte deshalb immer entsprechend groß gewählt werden.

3.4 Definition (SIG-Terme mit Variablen und ihre Algebra)

1. Für $s \in S$ heißt ein Element $x \in X_s$ *Variable* der Sorte s.
2. Für eine *SIG*-Algebra A wird eine Familie von Abbildungen

$$ass: X \to \mathsf{A} = (ass_s: X_s \to \mathsf{A}_s)_{s \in S}$$

Wertzuweisung an die (oder *Belegung* der) Variablen aus X in A genannt.

3. *SIG* lässt sich zu einer *Signatur mit Variablen* $SIG(X) = <S, OP(X)>$ erweitern, die die Variablen als 0-stellige Operationssymbole besitzt:

$$OP(X) = (OP(X)_{w,s})_{w \in S^*, s \in S}$$

mit $OP(X)_{\lambda,s} := OP_{\lambda,s} \cup X_s$ für alle $s \in S$ und $OP(X)_{w,s} := OP_{w,s}$ sonst.

4. Die Elemente der $SIG(X)$-Termalgebra $T_{SIG(X)}$, also die $SIG(X)$-Terme, werden *SIG-Terme mit Variablen in X* genannt.
5. Durch Vergessen der 0-stelligen Operationen $x_T = x$ für alle $x \in X_s$, $s \in S$ in $T_{SIG(X)}$ wird daraus eine SIG-Algebra $T_{SIG}(X)$, die *SIG-Termalgebra mit Variablen in X*:
   (i) $T_{SIG}(X)_s := T_{SIG(X),s}$ für alle $s \in S$,
   (ii) $c_{T_{SIG}(X)} := c_{T_{SIG(X)}}$ für alle $c \in OP_{w,s}$.

Analog zu den SIG-Termen lassen sich SIG-Terme mit Variablen in SIG-Algebren interpretieren, wenn man nicht nur die formalen Operationssymbole durch die aktuellen Operationen ersetzt, sondern nun auch die Variablen mit aktuellen Werten belegt. Ähnlich wie bei Termalgebren definieren die Interpretationen dieser parametrisierten Operationsaufrufe eindeutige SIG-Homomorphismen.

3.5 Theorem (SIG-Termalgebra mit Variablen)
Zu jeder SIG-Algebra A und jeder Wertzuweisung $ass\colon X \to \mathsf{A}$ (gemäß Definition 3.4.2) existiert genau ein SIG-Homomorphismus $ass^{\S}\colon T_{SIG}(X) \to \mathsf{A}$ mit $ass^{\S}_s(x) = ass_s(x)$ für alle $x \in X_s$, $s \in S$.

Explizit ist $ass^{\S}$ definiert durch:

(i) $ass^{\S}_s(c) = c_{\mathsf{A}}$ *für alle* $c \in OP_{\lambda,s}$, $s \in S$,
(ii) $ass^{\S}_s(x) = ass_s(x)$ *für alle* $x \in X_s$, $s \in S$,
(iii) $ass^{\S}_s(op(t_1, \ldots, t_n)) = op_{\mathsf{A}}(ass^{\S}_{s_1}(t_1)), \ldots, ass^{\S}_{s_n}(t_n))$
für alle $op \in OP_{s_1 \ldots s_n, s}$, $t_i \in T_{SIG}(X)_{s_i}$, $i = 1, \ldots, n$.

Beweis.
Da $ass_s(x)$ für alle $x \in X_s$, $s \in S$ ein Element von A_s auszeichnet, wird durch $x_{\mathsf{A}} := ass_s(x)$ A zu einer $SIG(X)$-Algebra $\overline{\mathsf{A}}$ erweitert, deren Datenmengen und SIG-Operationen mit denen von A identisch sind.

Wegen der Initialität (Theorem 2.22) gibt es dann einen $SIG(X)$-Homomorphismus $ass^{\S}\colon T_{SIG(X)} \to \overline{\mathsf{A}}$.

$ass^{\S}$ ist mit den $SIG(X)$-Operationen verträglich, also erst recht mit den SIG-Operationen ($OP \subseteq OP(X)$). Damit erweist sich $ass^{\S}\colon T_{SIG}(X) \to \mathsf{A}$ als SIG-Homomorphismus. Und die Verträglichkeit mit den 0-stelligen Operationen x_T und x_{A} zusammen mit deren Definition liefert:

$$ass^{\S}_s(x) = ass^{\S}_s(x_T) = x_{\mathsf{A}} = ass_s(x) \quad \text{für alle } x \in X_s,\ s \in S.$$

Es bleibt die Eindeutigkeit von $ass^{\S}$ zu zeigen. Sei dazu $g\colon T_{SIG}(X) \to \mathsf{A}$ ein SIG-Homomorphismus mit $g_s(x) = ass_s(x)$. Dann ist g aber auch mit den Operationen x_T und x_{A} verträglich:

$$g_s(x_T) = g_s(x) = ass_s(x) = x_{\mathsf{A}} \quad \text{für alle } x \in X_s,\ s \in S,$$

so dass $g\colon T_{SIG(X)} \to \overline{\mathsf{A}}$ auch einen $SIG(X)$-Homomorphismus bildet. Wegen der Initialität der Termalgebra $T_{SIG(X)}$ folgt daraus wie gewünscht: $g = ass^{\S}$. $ass^{\S}$ ist als Interpretation der $SIG(X)$-Terme in $\overline{\mathsf{A}}$ definiert, so dass die Richtigkeit der angegebenen expliziten Definition im Theorem aus Definition 2.13 folgt. □

3.6 Bemerkung
Wie bei der Initialität der Termalgebra kennt die universelle Algebra für die spezifische Eigenschaft der Termalgebra mit Variablen im obigen Theorem eine besondere Bezeichnung. In diesem Sinne ist $T_{SIG}(X)$ *freie SIG*-Algebra über X. Für freie Algebren gilt analog zu initialen, dass sie durch diese Eigenschaft bis auf Isomorphie eindeutig bestimmt sind (vgl. Theorem 2.25).

3.7 Beispiel

1. Wählt man als $X_{Bool} = \{bool1, bool2\}$, so ist die BOOL-Termalgebra mit Variablen in X:
$$T_{\text{BOOL}}(X) = (\{true, false, bool1, bool2\}, true, false).$$

2. Interessanter wird die Situation für $X_{Nat} = \{N\}$. Die Datenmenge der Termalgebra mit der Variablen N enthält die Terme:
$$succ^m(0) \quad \text{und} \quad succ^n(N) \quad \text{für alle } m, n \in \mathbb{N} \quad \text{(vgl. Beispiel 2.12.2).}$$
Die 0-stellige Operation ist wiederum $0_T := succ^0(0) = 0$, und die Nachfolgeoperation ist
$$succ_T(succ^m(0)) = succ^{m+1}(0) \quad \text{und} \quad succ_T(succ^n(N)) = succ^{n+1}(N).$$
Aktualisiert man die Variable N durch den Wert 7 in der NAT-Algebra $\mathbb{N}$, d.h. $ass(N) = 7$, so erhält der Term $succ^3(N)$ die Interpretation 10:
$$\begin{aligned} ass^{\S}(succ^3(N)) &= \mathsf{succ}(ass^{\S}(succ^2(N))) \\ &= \mathsf{succ}^2(ass^{\S}(succ(N))) \\ &= \mathsf{succ}^3(ass^{\S}(N)) \\ &= \mathsf{succ}^3(ass(N)) \\ &= \mathsf{succ}^3(7) \\ &= 7 + 3 = 10 \end{aligned}$$
Aber der Term $succ^3(0)$ wird genauso interpretiert wie in der Termalgebra ohne Variable, weil die Variable N nicht vorkommt:
$$\begin{aligned} ass^{\S}(succ^3(0)) &= \ldots \\ &= \mathsf{succ}^3(ass^{\S}(0)) \\ &= \mathsf{succ}^3(0) \\ &= 3 \end{aligned}$$

3. Bei BinTree können bemerkenswerterweise die Terme mit Variablen auch als binäre Bäume gedeutet werden (vgl. Beispiel 3.2.3), bei denen in den Blättern neben *leaf* auch die Variablen stehen können.

   Diese binären Bäume bzw. Terme mit Variablen haben dann in der BinTree-Algebra

   $$(\mathbb{N}, 1, \mathsf{succ}, \mathsf{succ}, \mathsf{l{+}r})$$

   mit $\mathsf{l{+}r}(m,n) = m + n + 1$ für alle $m, n \in \mathbb{N}$ folgende Interpretation: Die Zahl der Knoten, die nicht Variable sind, wird erhöht um $ass(x)$ für alle Variablen x, die als Blätter vorkommen, wobei $ass\colon X \to \mathbb{N}$ die vorgegebene Belegung der Variablen sei. Jedem binären Baum wird also die Zahl der Knoten mit "gewichteten" Blättern zugeordnet. Insbesondere wird für binäre Bäume, in denen keine Variable vorkommt, von der Interpretation die Knotenzahl berechnet.

Lässt man nicht wie in Theorem 3.5 für jede Belegung der Variablen die Terme variieren, sondern für einen festen Term die Wertzuweisung an die Parameter, so können Terme als (zusammengesetzte) Operationen gedeutet werden. Um das präzisieren zu können (Korollar 3.9), werden die in einem Term vorkommenden Variablen bestimmt:

3.8 Definition
Für *SIG*-Terme t mit Variablen in X sind die Mengen der *vorkommenden Variablen* $var(t)$ rekursiv gegeben durch:

(i) $var(c) = \emptyset$ für alle $c \in OP_{\lambda,s}$, $s \in S$,

(ii) $var(x) = \{x\}$ für alle $x \in X_s$, $s \in S$,

(iii) $var(op(t_1, \ldots, t_n)) = \bigcup_{i=1}^{n} var(t_i)$ für alle $op \in OP_{s_1 \ldots s_n, s}$, $t_i \in T_{SIG}(X)_{s_i}$.

(Dabei bezeichnet $\bigcup_{i=1}^{n}$ die n-fache Vereinigung von Mengen.)

Manchmal werden die in t vorkommenden Variablen auch nach Sorten sortiert als Mengenfamilie $(var(t)_s)_{s \in S}$ mit $var(t)_s = var(t) \cap X_s$. Auch dafür wird die Bezeichnung $var(t)$ verwendet, da Missverständnisse unwahrscheinlich sind.

3.9 Korollar
Sei t ein SIG-Term der Sorte s mit Variablen in X; sei $s_1 \ldots s_n \in S^*$; sei $x_i \in X_{s_i}$ für $i = 1, \ldots, n$; und sei $var(t) \subseteq \{x_i \mid i = 1, \ldots, n\}$.

Dann induziert der Term t für jede SIG-Algebra A eine abgeleitete Operation

$$t_{\mathsf{A}}\colon \mathsf{A}_{s_1} \times \cdots \times \mathsf{A}_{s_n} \to \mathsf{A}_s$$

mit $t_{\mathsf{A}}(a_1, \ldots, a_n) = ass^{\S}(t)$ für alle $a_i \in \mathsf{A}_{s_i}$, $i = 1, \ldots, n$, wobei die Wertzuweisungen $ass\colon X \to \mathsf{A}$ gegeben sind durch $ass(x_i) = a_i$ für $i = 1, \ldots, n$ und sonst beliebig gewählt werden können.

Beweis.
Nach Theorem 3.5 ist für jedes $ass\colon X \to \mathsf{A}$ und $t \in T_{SIG}(X)_s$ $ass^{\S}(t)$ ein Wert in A_s, der nur von den aktuellen Werten der Variablen $var(t)$ unter ass abhängt, so dass t_{A} eine Abbildung der gewünschten Form ist. □

Etwas kompliziert an dieser Auffassung von den Termen als Operationen ist der Zusammenhang von Argumenten der Operation und Variablen des Terms:

(a) Die Argumente sind geordnet, die Variablen nicht; man muss daher die Reihenfolge vorgeben, sie ist durch $var(t)$ nicht festgelegt.

(b) Die aktuellen Argumente müssen die aktuellen Werte der Variablen von t festlegen, damit genau ein Wert zugeordnet wird; deshalb muss $var(t)$ Teil der Variablen sein, die den Argumenten zugeordnet sind.

(c) Auf den Wert einer Abbildung müssen nicht alle Argumente Einfluss haben. Typisch dafür ist die Projektion $p_A\colon A \times B \to A$ mit $p_A(a,b) = a$, bei der das Ergebnis nicht von der zweiten Komponente abhängt. Aus diesem Grunde wird nicht $var(t) = \{x_1, \ldots, x_n\}$ gefordert.

Das wird auch an den folgenden Beispielen klar.

3.10 Beispiel
Betrachte den STRING-Term

$$t = \mathit{insert}(a, \mathit{insert}(a, s))$$

mit den Variablen a zur Sorte A und s zur Sorte *String*. Für *String A* als Argumente liefert das in A^* die Abbildung

$$t^1_{A^*}(w,a) = \mathsf{insert}(a, \mathsf{insert}(a,w)) = \mathsf{insert}(a,aw) = aaw.$$

$t^1_{A^*}$ schreibt also vor sein erstes zweimal sein zweites Argument.

Für die Argumente *A A String* jedoch, wobei die Variable a dem ersten Argument zugeordnet ist, erhält man

$$t^2_{A^*}(b,c,v) = \mathsf{insert}(b, \mathsf{insert}(b,v)) = bbv \qquad \text{für alle } b,c \in A,\ v \in A^*,$$

d.h. $t^2_{A^*}$ schreibt vor das dritte Argument zweimal das erste und vergisst das zweite.

Dagegen lässt sich *String* allein nicht als Argument wählen, weil dann die Argumente nicht die Wertzuweisung an die Variable a festlegen.

4 Spezifikation von Algebren mit Gleichungen

Aufbauend auf Terme mit Variablen ist es nun möglich, auf der syntaktischen Ebene das Konzept der Signaturen um Gleichungen zu ergänzen und damit zu Spezifikationen von Algebren zu erweitern (Definition 4.1) sowie auf der semantischen Ebene zu präzisieren,

wann eine Algebra die durch die Gleichungen festgelegten Eigenschaften besitzt, wann die Gleichungen in ihr gelten (Definition 4.5).

Leider gehört zu diesen Algebren i.a. nicht die Termalgebra, da Terme nur gleich sind, wenn sie formal übereinstimmen, und sonst in der Termalgebra keine weiteren Gleichheiten auffindbar sind. Trotz der guten Erfahrungen mit der Termalgebra muss also für die Semantik einer Spezifikation, für die von ihr erzeugte Algebra nach einer neuen geeigneten Kandidatin Ausschau gehalten werden. Die Termalgebra jedoch scheidet einzig und allein deshalb aus, weil sie in der Regel den Gleichungen nicht genügt; als Datentyp hat sie in der Repräsentationsunabhängigkeit und Operationserzeugtheit Eigenschaften, die es Wert sind, bewahrt zu werden. So kann man sich fragen, um beiden Aspekten gerecht zu werden: Lassen sich die Gleichungen nicht der Termalgebra aufzwingen? Können nicht die Terme, die verschieden sind, aber aufgrund der vorgegebenen Gleichungen übereinstimmen müssten, gleichgemacht werden?

In Definition 4.7 wird das bejaht; es lässt sich durch einen Identifizierungsprozess schrittweise bewerkstelligen. Bemerkenswerterweise erhält man durch diese Konstruktion zu jeder Spezifikation eine ausgezeichnete Algebra mit Eigenschaften weitgehend analog zur Termalgebra (Theorem 4.9). Die resultierende – so genannte – Quotiententermalgebra kann daher mit einigem Recht als abstrakter Datentyp bezeichnet werden. Der Fortschritt gegenüber Signatur und Termalgebra liegt darin, dass die in Beispiel 3.1 aufgedeckten Mängel nicht mehr auftreten: Die ganzen Zahlen und die Potenzmenge von A lassen sich mit Hilfe von Gleichungen spezifizieren; in BOOL, NAT, STRING und BINTREE lassen sich durch Gleichungen zusätzliche Operationen definieren, so dass auch weiterhin die Wahrheitswerte, natürlichen Zahlen, Wörter über A bzw. binären Bäume ihre Semantik sind (Beispiel 4.10).

4.1 Definition (Gleichungen und Spezifikationen)
Sei $SIG = <S, OP>$ eine Signatur und $X = (X_s)_{s \in S}$ eine Mengenfamilie von Variablen.

1. Für $s \in S$ heißt ein Paar $e = (L, R)$ von SIG-Termen L und R der Sorte s mit Variablen in X *Gleichung* der Sorte s ($L, R \in T_{SIG}(X)_s$).
2. Eine *Spezifikation* $SPEC = <SIG, EQ>$ besteht aus einer Signatur $SIG = <S, OP>$ und einer Mengenfamilie $EQ = (EQ_s)_{s \in S}$, wobei die Elemente von EQ_s Gleichungen der Sorte s sind.

4.2 Bemerkung
Wie schon bei Signaturen bietet sich auch für konkrete Spezifikationen eine abgekürzte, intuitive Schreibweise der Gleichungen an, indem alle Elemente von EQ hinter dem Schlüsselwort **eqns** aufgelistet und die linke und rechte Seite jeder Gleichung durch ein (formales) Gleichheitszeichen getrennt werden (was jedoch nicht verwechselt werden darf mit der tatsächlichen Gleichheit von Elementen einer Menge).

Formal werden Spezifikationen dieser Art von folgender Grammatik generiert:

$$
\begin{array}{lll}
<\text{spec}> := <\text{sign}> <\text{eqns}> & (<\text{sign}> \text{ aus Bemerkung 2.3}) \\
<\text{eqns}> := \textbf{eqns } \text{list1-of-}<\text{singleq}> & \\
<\text{singleq}> := <s\text{-term}> = <s\text{-term}> & \\
<s\text{-term}> := x & \text{für } x \in X_s \\
<s\text{-term}> := c & \text{für } c \in OP_{\lambda,s} \\
<s\text{-term}> := op(<s_1\text{-term}>, \ldots, <s_n\text{-term}>) & \text{für } op \in OP_{s_1 \ldots s_n, s}
\end{array}
$$

Um die Schreibarbeit für die Beispiele möglichst gering zu halten, werden die in den <eqns> verwendeten Variablen vorher nicht deklariert. Man erkennt sie jedoch als die Identifizierer, die nicht als Operationen eingeführt worden sind.

Abgesehen davon und von anderen syntaktischen Unerheblichkeiten weist dieses Spezifikationsprimitiv große Ähnlichkeit mit dem Elementarkonzept der algebraischen Spezifikationssprache CASL (siehe `http://www.brics.dk/Projects/CoFI/Documents/CASL/Summary/`) auf.

4.3 Beispiel

1. Eine erste Spezifikation von ganzen Zahlen mit Gleichungen lautet, aufbauend auf der Signatur in Beispiel 3.1.3:

   spec INT =
   sorts Int
   opns $0\colon \to Int$
   $succ, pred\colon Int \to Int$
   eqns $succ(pred(X)) = X$
   $pred(succ(X)) = X$

   Ob dadurch allerdings tatsächlich die ganzen Zahlen spezifiziert sind, kann erst in Beispiel 4.10 geklärt werden.

2. Auch die Spezifikation der Potenzmenge von einer Menge $A = \{a_1, \ldots, a_n\}$ aus Beispiel 3.1.4 lässt sich nun geeignet ergänzen:

   spec SET =
   sorts A, Set
   opns $a_i\colon \to A$ für $i = 1, \ldots, n$
   $\emptyset\colon \to Set$
   $insert\colon A \times Set \to Set$
   eqns $insert(a, insert(a, M)) = insert(a, M)$
   $insert(a, insert(b, M)) = insert(b, insert(a, M))$

   Dabei sind a, b Variable der Sorte A, und M ist Set-Variable.

3. Die Spezifikation von BOOL aus Beispiel 1.1.1 wird um die Operationen *non*, *and*, *or*, *impl*(ication) und *equiv*(alence) erweitert:

$$
\begin{array}{ll}
\textbf{spec} & \textsc{Bool} = \\
& \textbf{sorts}\ \ Bool \\
& \textbf{opns}\ \ true, false\colon \to Bool \\
& \qquad non\colon Bool \to Bool \\
& \qquad and, or, impl, equiv\colon Bool \times Bool \to Bool \\
& \textbf{eqns}\ \ non(true) = false \\
& \qquad non(non(B)) = B \\
& \qquad and(true, B) = B \\
& \qquad and(false, B) = false \\
& \qquad or(B, B') = non(and(non(B), non(B'))) \\
& \qquad impl(B, B') = or(non(B), B') \\
& \qquad equiv(B, B') = and(impl(B, B'), impl(B', B))
\end{array}
$$

Dabei ist zu beachten, dass zur weiteren Vereinfachung der Schreibweise Operationsnamen in einer Liste zusammengefasst sind, wenn sie gleiche Argumente und Wertebereiche haben. Die Präfixnotation für die Operationen rührt von der Definition der Terme her. Sie ist jedoch für Boolesche Operationen unüblicher als die Infixnotation. In diesem Ausnahmefall und in ähnlichen Situationen (wie 4. unten) werden deshalb auch andere Termschreibweisen verwendet:

$$
\begin{array}{ll}
\textbf{eqns} & non\ true = false \\
& non\ non\ B = B \\
& true\ and\ B = B \\
& false\ and\ B = false \\
& B\ or\ B' = non((non\ B)\ and\ (non\ B')) \\
& B\ impl\ B' = (non\ B)\ or\ B' \\
& B\ equiv\ B' = (B\ impl\ B')\ and\ (B'\ impl\ B)
\end{array}
$$

4. Zur Spezifikation von NAT aus Beispiel 1.1.2 können nun folgendermaßen die Addition und Multiplikation als neue Operationen hinzugenommen werden:

$$
\begin{array}{ll}
\textbf{spec} & \textsc{Nat} = \\
& \textbf{sorts}\ \ Nat \\
& \textbf{opns}\ \ 0\colon \to Nat \\
& \qquad succ\colon Nat \to Nat \\
& \qquad +, *\colon Nat \times Nat \to Nat \\
& \textbf{eqns}\ \ M + 0 = M \\
& \qquad M + succ(N) = succ(M + N) \\
& \qquad M * 0 = 0 \\
& \qquad M * succ(N) = (M * N) + M
\end{array}
$$

Die Gleichungen einer Spezifikation $SPEC = <SIG, EQ>$ sind als Anforderungen an SIG-Algebren zu verstehen. Ist $e = (L, R)$ eine Gleichung aus EQ, A eine SIG-Algebra und $ass\colon X \to \mathsf{A}$ eine Wertzuweisung an die Variablen, dann greifen die Terme L und R auf die Daten $ass^{\S}(L)$ und $ass^{\S}(R)$ in A zu (vgl. Theorem 3.5). Sind sie gleich, so kann man davon sprechen, dass A bzgl. ass die Gleichung e erfüllt. Andernfalls verletzt A die geforderte Gleichung. Bevor das als Definition noch einmal genau aufgeschrieben wird, sei für den Rest des Kapitels folgendes vorausgesetzt.

4.4 Generalvoraussetzung
Sei $SPEC = <SIG, EQ>$ mit $SIG = <S, OP>$ eine Spezifikation im Sinne der Definition 4.1, wobei die in EQ vorkommenden Variablen alle aus einer Mengenfamilie X stammen (vgl. Voraussetzung 3.3).

4.5 Definition (*SPEC*-Algebren)

1. In einer SIG-Algebra A *gilt* die Gleichung $e = (L, R)$ aus EQ, wenn linke und rechte Seite von e immer gleich interpretiert werden, wenn also für alle Wertzuweisungen $ass\colon X \to \mathsf{A}$ schon $ass^{\S}(L) = ass^{\S}(R)$ gilt.
2. A ist eine *SPEC-Algebra* (auch: *Algebra des Typs SPEC*), wenn A SIG-Algebra ist und alle Gleichungen aus EQ in ihr gelten.

Anmerkung zur Schreibweise

In Begriffen dieses Kapitels wird oft das Präfix "*SPEC*" weggelassen.

Beachte außerdem, dass eigentlich $ass^{\S}$ in Definition 4.5.1 einen Index tragen müsste (vgl. Theorem 3.5). Da aber jede Gleichung, jeder Term und jede Variable nach Definition immer mit genau einer Sorte assoziiert ist und diese Sorte auch diejenige Abbildung eines Homomorphismus oder einer Wertzuweisung bestimmt, die anwendbar ist, wird im folgenden häufig auf die explizite Angabe der Indizes verzichtet.

4.6 Beispiel

1. Die ganzen Zahlen $\mathbb{Z} = (\mathbb{Z}, 0, \mathsf{succ}, \mathsf{pred})$ (vgl. Beispiel 2.9.2 und 3.1.3) definieren eine INT-Algebra für INT aus Beispiel 4.3.1. Denn die Operationen genügen wegen

$$\mathsf{succ}(\mathsf{pred}(z)) = \mathsf{succ}(z - 1) = z - 1 + 1 = z$$

   und

$$\mathsf{pred}(\mathsf{succ}(z)) = \mathsf{pred}(z + 1) = z + 1 - 1 = z$$

   für beliebige Werte z der Variablen X den beiden Gleichungen.
2. In der Potenzmenge $\mathcal{P}(A)$ aus Beispiel 3.1.4 gelten die Gleichungen in der Spezifikation SET aus Beispiel 4.3.2. Denn für alle Teilmengen M von A (das ist der Wert der Variablen M in $\mathcal{P}(A)$) und für alle Elemente $a, b \in A$ (die Werte der Variablen a, b) gilt:

$$\begin{aligned}\mathsf{insert}(a, \mathsf{insert}(b, M)) &= \mathsf{insert}(a, M \cup \{b\}) \\ &= M \cup \{b\} \cup \{a\} \\ &= M \cup \{a\} \cup \{b\} \\ &= \mathsf{insert}(b, M \cup \{a\}) \\ &= \mathsf{insert}(b, \mathsf{insert}(a, M)),\end{aligned}$$

während die Gültigkeit der ersten Gleichung bereits in Beispiel 3.1.4 nachgewiesen ist.

3. Für BOOL aus Beispiel 2.9.1 lassen sich bekanntermaßen die Operationen $\vee, \wedge, \Rightarrow, \Leftrightarrow$ durch Wahrheitstafeln definieren, wobei allerdings true und false durch T bzw. F abgekürzt werden:

$\wedge$	T	F
T	T	F
F	F	F

$\vee$	T	F
T	T	T
F	T	F

$\Rightarrow$	T	F
T	T	F
F	T	T

$\Leftrightarrow$	T	F
T	T	F
F	F	T

Hier stehen die ersten Argumente unter und die zweiten neben dem Operationszeichen, der Wert befindet sich jeweils dort, wo sich Zeile und Spalte der Argumente treffen.

Außerdem lässt sich die Negation definieren durch $\neg\,\mathsf{T} = \mathsf{F}$ und $\neg\,\mathsf{F} = \mathsf{T}$.

Damit BOOL mit den neuen Operationen eine BOOL-Algebra zur Spezifikation in Beispiel 4.3.3 bildet, müssen die Gleichungen gelten. Für die erste ist das nach Definition von $\neg$ richtig. Für die zweite, dritte und vierte betrachte die beiden einzig möglichen Belegungen der Variablen B durch T und F, wofür man nach Definition von $\neg$ und $\wedge$ erhält:

2. $\neg\neg\,\mathsf{T} = \neg\,\mathsf{F} = \mathsf{T}$ und $\neg\neg\,\mathsf{F} = \neg\,\mathsf{T} = \mathsf{F}$
3. $\mathsf{T} \wedge \mathsf{T} = \mathsf{T}$ und $\mathsf{T} \wedge \mathsf{F} = \mathsf{F}$
4. $\mathsf{F} \wedge \mathsf{T} = \mathsf{F}$ und $\mathsf{F} \wedge \mathsf{F} = \mathsf{F}$

Die Gültigkeit der übrigen Gleichungen kann den beiden folgenden Tabellen entnommen werden:

1	2	3	4	5	6	7
B	B'	$\neg B$	$\neg B'$	$\neg B \wedge \neg B'$	$\neg(\neg B \wedge \neg B')$	$B \vee B'$
T	T	F	F	F	T	T
T	F	F	T	F	T	T
F	T	T	F	F	T	T
F	F	T	T	T	F	F

1	2	3	8	9	10	11	12
B	B'	$\neg B$	$\neg B \vee B'$	$B \Rightarrow B'$	$B' \Rightarrow B$	$(B \Rightarrow B') \wedge (B' \Rightarrow B)$	$B \Leftrightarrow B'$
T	T	F	T	T	T	T	T
T	F	F	F	F	T	F	F
F	T	T	T	T	F	F	F
F	F	T	T	T	T	T	T

Dabei geben die Spalten 1 und 2 die vier möglichen Belegungen der beiden Variablen B und B' an und, davon abhängig, die übrigen Spalten die Werte der oben stehenden Terme.

Dass die Gleichungen gültig sind, besagt die Übereinstimmung der Spalten 6 und 7, 8 und 9, 11 und 12; die anderen Spalten geben Zwischenrechnungen wieder.

4. Die natürlichen Zahlen $\mathbb{N} = (\mathbb{N}, 0, \mathsf{succ}, +, *)$ bilden mit der üblichen Addition und Multiplikation eine Algebra des Typs NAT aus Beispiel 4.3.4. Denn bei Belegung der Variablen M und N mit beliebigen natürlichen Zahlen $m, n \in \mathbb{N}$ ergeben sich die Gleichungen aus bekannten Eigenschaften der natürlichen Zahlen:

$$\begin{aligned} &n + 0 = n \\ &m + \mathsf{succ}(n) = m + (n + 1) = (m + n) + 1 = \mathsf{succ}(m + n) \\ &n * 0 = 0 \\ &m * \mathsf{succ}(n) = m * (n + 1) = m * n + m * 1 = m * n + m \end{aligned}$$

Wie bereits erwähnt, gelten in der *SIG*-Termalgebra i.a. nicht die Gleichungen in EQ (es sei denn, diese haben die Form $t = t$ oder EQ enthält überhaupt keine Elemente). Dennoch kann man natürlich auch in diesem Fall für alle Wertzuweisungen $ass\colon X \to T_{SIG}$ und für alle Gleichungen $e = (L, R) \in EQ$ diejenigen Terme benennen, die gleich sein sollten: $ass^{\S}(L)$ und $ass^{\S}(R)$.

Identifiziert man nun alle diese Paare von Termen, so ist im Resultat alles gleich, was aufgrund der Gleichungen übereinstimmen soll. Leider ist das Ergebnis dieses Identifizierungsvorgangs noch keine *SIG*-Algebra. Aber auch dieser Mangel lässt sich beheben, wenn man zusätzlich zu den Paaren $ass^{\S}(L)$ und $ass^{\S}(R)$ auch deren Bilder unter Operationen identifiziert:

$$\begin{aligned} &op(ass^{\S}(L_1), \ldots, ass^{\S}(L_n)) \quad \text{und} \quad op(ass^{\S}(R_1), \ldots, ass^{\S}(R_n)) \\ &\text{für } op \in OP_{s_1 \ldots s_n, s},\ (L_i, R_i) \in EQ_{si},\ i = 1, \ldots, n. \end{aligned}$$

Diese Konstruktion kann folgendermaßen präzisiert werden.

4.7 Definition und Konstruktion (Quotiententermalgebra)

1. Die Gleichungen induzieren auf den Termen aus T_{SIG} eine Familie von Relationen $\equiv_{EQ} = (\equiv_{EQ,s})_{s \in S}$, die *Äquivalenz* genannt wird und rekursiv definiert ist durch:
   (i) $ass^{\S}(L) \equiv_{EQ,s} ass^{\S}(R)$
   für alle $(L, R) \in EQ_s$, $s \in S$ und $ass\colon X \to T_{SIG}$,
   (ii) $t_i \equiv_{EQ,s_i} t_i'$, $i = 1, \ldots, n$ impliziert $op(t_1, \ldots, t_n) \equiv_{EQ,s} op(t_1', \ldots, t_n')$
   für alle $op \in OP_{s_1 \ldots s_n, s}$ und $t_i, t_i' \in T_{SIG,s_i}$,
   (iii) $t \equiv_{EQ,s} t$,
   (iv) $t \equiv_{EQ,s} t'$ impliziert $t' \equiv_{EQ,s} t$,
   (v) $t \equiv_{EQ,s} t'$, $t' \equiv_{EQ,s} t''$ impliziert $t \equiv_{EQ,s} t''$.

2. Für jeden Term $t \in T_{SIG,s}$ werden in der *Äquivalenzklasse* $[t]$ alle zu t äquivalenten Terme zusammengefasst und damit identifiziert:

$$[t] = \{t' \in T_{SIG,s} \mid t \equiv_{EQ,s} t'\} \quad (s \in S).$$

3. Die Mengen aller Äquivalenzklassen $T_{SPEC,s} = \{[t] \mid t \in T_{SIG,s}\}$ für alle $s \in S$ bilden die Datenmengen der *Quotiententermalgebra* $T_{SIG,EQ}$, deren Operationen definiert sind durch:

   (i) $\overline{c} = [c]$ für alle $c \in OP_{\lambda,s}$, $s \in S$,

   (ii) $\overline{op}([t_1], \ldots, [t_n]) = [op(t_1, \ldots, t_n)]$ für alle $op \in OP_{s_1 \ldots s_n, s}$, $t_i \in T_{SIG,s_i}$, $i = 1, \ldots, n$.

4.8 Bemerkung
Für $s \in S$ ist $\equiv_{EQ,s}$ eine Relation auf der Datenmenge $T_{SIG,s}$, d.h. eine Teilmenge des kartesischen Produkts $T_{SIG,s} \times T_{SIG,s}$. Dabei wird statt $(t, t') \in \equiv_{EQ,s}$ $t \equiv_{EQ,s} t'$ geschrieben und t *äquivalent zu* t' gesagt. Die Äquivalenz ist nach Definition die kleinste Relationenfamilie mit den Eigenschaften (i) bis (v). Die Eigenschaften (iii), (iv) und (v) besagen, mathematisch gesehen, dass $\equiv_{EQ}$ eine Familie von Äquivalenzrelationen ist, die aufgrund von (ii) strukturverträglich mit den *SIG*-Operationen ist und deshalb in der Algebra als *Kongruenz* bezeichnet wird. Die Forderung (i) bedeutet schließlich, dass die Familie von Äquivalenz- bzw. Kongruenzrelationen $\equiv_{EQ}$ von EQ erzeugt wird.

Dass die Konstruktion in Definition 4.7 ein brauchbares Ergebnis liefert, zeigt das nächste Theorem.

4.9 Theorem (Quotiententermalgebra)
T_{SPEC} ist eine SIG-erzeugte, initiale SPEC-Algebra.

Beweis wird nachgeliefert**.**

4.10 Beispiel

1. Sei INT0 die Spezifikation, die durch Weglassen der Gleichungen aus INT entsteht; d.h. die Signatur von INT (siehe Beispiel 4.3.1). Die INT0-Terme haben die Gestalt $op_1 \ldots op_n(0)$ für $n \in \mathbb{N}$, $op_i \in \{succ, pred\}$, $i = 1, \ldots, n$.

   Sei $\#_{succ}(t)$ die Zahl der *succ*-Operationen in jedem dieser Terme t und $\#_{pred}(t)$ die Zahl der *pred*-Operationen. Sei ferner $diff(t) = \#_{succ}(t) - \#_{pred}(t)$. Man kann sich nun überlegen, dass das so definierte *diff*: $T_{\text{INT0}} \to \mathbb{Z}$ gerade die Interpretation der INT0-Terme in $\mathbb{Z}$ gemäß Definition 2.13 ist.

   Da in $\mathbb{Z}$ die Gleichungen $EQ(\text{INT})$ von INT gelten (INT $= <\text{INT0}, EQ(\text{INT})>$!), gilt $\equiv_{EQ(\text{INT})} \subseteq \equiv_{diff}$, wobei $t \equiv_{diff} t'$ für zwei Terme t, t' definiert ist durch $diff(t) = diff(t')$, d.h. die Differenzen von *succ*- und *pred*-Operationen äquivalenter Terme sind gleich. Die Inklusion der Äquivalenzrelation ist gegeben, weil $\equiv_{diff}$ die Eigenschaften von Definition 4.7.1 hat, $\equiv_{EQ(\text{INT})}$ aber die kleinste derartige ist. Das sieht man

auch leicht direkt ein; denn das Anwenden einer Gleichung entfernt aus jedem Term gerade ein *succ* und ein *pred*, wodurch die Differenz natürlich nicht verändert wird.

Darüber hinaus kann man jeden Term mit Hilfe der Gleichungen solange umformen, bis keine *succ*- und *pred*-Operationen gleichzeitig mehr vorkommen. Jeder Term ist also äquivalent zu einem der Form

$$pred^m(0) \quad \text{oder} \quad 0 \quad \text{oder} \quad succ^m(0) \quad (m \geq 1).$$

Diese sind jedoch untereinander alle inäquivalent wegen

$$\begin{aligned} diff(pred^m(0)) &= -m \\ diff(0) &= 0 \\ diff(succ^m(0)) &= m. \end{aligned}$$

Die Daten von T_{INT} sind daher die Äquivalenzklassen

$$[pred^m(0)], [0], [succ^m(0)] \quad \text{für alle } m \geq 1,$$

so dass der eindeutig existierende Homomorphismus $init\colon T_{\text{INT}} \to \mathbb{Z}$ explizit gegeben ist durch

$$\begin{aligned} init([pred^m(0)]) &= -m \\ init([0]) &= 0 \\ init([succ^m(0)]) &= m. \end{aligned}$$

Diese Zuordnung ist offenbar bijektiv und damit $\mathbb{Z}$ isomorph zu T_{INT}. Fasst man also die Quotiententermalgebra als Semantik einer Spezifikation auf, so kann man sagen, dass INT (bis auf Umbenennung) die ganzen Zahlen $\mathbb{Z}$ spezifiziert.

2. Sei SET0 die Signatur von SET aus Beispiel 4.3.2. Wie bereits in Beispiel 3.1.4 bemerkt, stimmen die SET0-Terme praktisch mit den STRING-Termen überein.

   Durch die beiden SET-Gleichungen lässt sich jeder Term t äquivalent umformen zu

   $$t \equiv insert(a_{i_1}, insert(a_{i_2}, \ldots, insert(a_{i_m}, \emptyset) \ldots))$$

   für ein $m \in \mathbb{N}$ und $1 \leq i_1 < i_2 < \cdots < i_n \leq m$, indem man mit der ersten Gleichung mehrfaches Hinzufügen desselben Elements beseitigt und *insert*-Operationen mit der zweiten umdreht, wenn die Indizes zugefügter Elemente lokal keine aufsteigende Folge der angegebenen Art bilden.

   Damit kennt man aus jeder Äquivalenzklasse von T_{SET} ein Element explizit, so dass sich für jede SET-Algebra B auch explizit angeben lässt, wie der Homomorphismus $init\colon T_{\text{SET}} \to \mathsf{B}$, der aufgrund der Initialität existiert, definiert ist:

   $$init([t]) = insert_{\mathsf{B}}(a_{i_1\mathsf{B}}, insert_{\mathsf{B}}(a_{i_2\mathsf{B}}, \ldots, insert_{\mathsf{B}}(a_{i_m\mathsf{B}}, \emptyset_{\mathsf{B}}) \ldots))$$

Für die SET-Algebra $\mathcal{P}(A)$ bedeutet das speziell:

$$init([t] = \{a_{i_1}, \ldots, a_{i_m}\}.$$

Da sich die Elemente einer Teilmenge von A immer nach der Größe der Indizes sortieren lassen, ist *init* offenbar ein Isomorphismus. Die von der Spezifikation SET erzeugte Algebra T_{SET} stimmt daher bis auf Umbenennung mit der Potenzmenge $\mathcal{P}(A)$ – dem intendierten Datentyp – überein.

3. Sei BOOL0 die Signatur der neuen BOOL-Spezifikation aus Beispiel 4.3.3. Es ergibt sich dann durch vollständige Induktion über die Länge der BOOL0-Terme, dass jeder dieser Terme entweder zu *true* oder zu *false* äquivalent ist.

   Hat ein BOOL0-Term t beispielsweise die Form $and(t', t'')$, dann sind t' und t'' kürzer als t, so dass t' etwa äquivalent ist zu *true* und t'' zu *false* (Induktionsvoraussetzung). Es folgt dann für t mit der dritten BOOL-Gleichung und der Eigenschaft (ii) der Äquivalenz in Definition 4.7:

   $$t = and(t', t'') \equiv and(true, false) \equiv false$$

   (Induktionsschluss).

   Alle anderen Fälle für die Form von t lassen sich analog abhandeln.

   Mit anderen Worten hat T_{BOOL} eine zweielementige Datenmenge und ist deshalb isomorph zur Algebra BOOL aus Beispiel 4.6.3.

4. Jeder NAT0-Term, wobei NAT0 die Signatur von NAT sei, ist äquivalent zu einem Term, in dem nur 0 und *succ* als Operationen vorkommen. Die Äquivalenzklassen von T_{NAT} haben also die Form $[succ^m(0)]$, $m \in \mathbb{N}$. Da aber die Interpretation von $succ^m(0)$ in NAT gerade der Exponent m ist (vgl. Beispiel 2.15.2), trifft der Homomorphismus $init\colon T_{\text{NAT}} \to \mathbb{N}$ die offenbar bijektive Zuordnung

   $$init([succ^m(0)] = m \quad \text{für alle } m \in \mathbb{N}.$$

In der Quotiententermalgebra T_{SPEC} hat man also nach Theorem 4.9 zu einer Spezifikation *SPEC* eine *SPEC*-Algebra gefunden, die initial und im Hinblick auf diese Eigenschaft repräsentationsunabhängig ist (vgl. Definition 2.21 und folgende) sowie von den Operationen erzeugt wird. Sind bei einer Spezifikation keine Gleichungen vorgegeben ($EQ = \emptyset$), so stimmt T_{SPEC} bis auf Isomorphie mit der Termalgebra überein. Die syntaktischen und semantischen Konzepte und Möglichkeiten von Signaturen und Termalgebren beim Entwurf von Algebren bzw. Datentypen sind also vollständig als Spezialfall in dem allgemeineren Prinzip der Spezifikation erhalten geblieben. Die in Beispiel 3.1 entdeckten Mängel aber konnten durch das zusätzliche Konstrukt der Gleichungen beseitigt werden, wie Beispeil 4.10 zeigt. In allen sechs Beispielen BOOL, NAT, INT, STRING, SET und BINTREE ist T_{SPEC} isomorph zu den vorgegeben, intendierten Datentypen der Wahrheitswerte, natürlichen und ganzen Zahlen, Wörter über und Teilmengen von A sowie der binären Bäume. In diesem Sinne sind die sechs Beispiele korrekt spezifiziert durch die

entworfenen Signaturen und Gleichungen. Der Nachweis in Beispiel 2.29 und 4.10 gelang in allen Fällen mit Hilfe der Initialität von T_{SPEC}.

Das alles zusammengenommen berechtigt zur folgenden These.

These 4
Spezifikationen *SPEC* als Syntax mit Quotiententermalgebren T_{SPEC} als Semantik beschreiben abstrakte Datentypen. Die Initialität von T_{SPEC} ermöglicht Korrektheitsbeweise (wobei ein vorgegebener, gewünschter Datentyp dann korrekt spezifiziert ist, wenn er bis auf Umbenennung, bis auf Repräsentation der Daten mit T_{SPEC} übereinstimmt).

5 Korrektheit von Spezifikationen

Die Überlegungen und Resultate am Ende von Kapitel 4 legen nahe, den Begriff der Korrektheit nachzutragen.

Im allgemeinen gilt etwas (eine Spezifikation, Implementierung, ein Programm u.ä.) als korrekt, wenn es das tut, was es soll. Das ist leicht gesagt; die Schwierigkeit besteht jedoch darin, dass Sein und Sollen verglichen werden müssen. Erforderlich ist also eine gemeinsame Sprachebene für die Semantik, die ja widergibt, was ein Softwareprodukt ist oder tut, und für die Intention, die mit dem Produkt verfolgt wird. In dieser Sprachebene müssen darüber hinaus Übereinstimmen und Verschiedensein präzisiert werden können. (Vgl. die Einleitung.)

Im Gegensatz zu nahezu allen Programmier- und Entwurfssprachen ist für die algebraischen Spezifikationskonzepte eine solche Vergleichsebene ohne zusätzlichen Aufwand verfügbar. Da die Semantik einer Spezifikation als Quotiententermalgebra konstruiert ist (Theorem 4.9), erlauben Homomorphismen (Definition 2.19) den Vergleich mit anderen Algebren, wobei das Vorhandensein eines Homomorphismus bereits strukturelle Ähnlichkeiten konstatiert. Von Übereinstimmung zwischen Semantik und Intention, festgelegt durch eine Algebra, lässt sich in dem Falle sprechen, dass beide isomorph sind. Das trifft sich mit der Auffassung, dass bei abstrakten Datentypen die Datenrepräsentation keine Rolle spielen soll, dass eine Umbenennung der Daten, also ein Isomorphismus, die charakteristischen Merkmale nicht verändern darf (vgl. These 3 in Kapitel 2 und These 4 in Kapitel 4). Zusammenfassend erhält man folgende Präzisierung:

5.1 Definition
Eine Spezifikation $SPEC = <SIG, EQ>$ ist *korrekt* bzgl. einer *SIG*-Algebra A, wenn die Quotiententermalgeba T_{SPEC} isomorph ist zu A, d.h.

$$T_{SPEC} \cong \mathsf{A}.$$

Die Ausführungen in den Beispielen 2.29 und 4.10 lassen sich nun prägnanter formulieren.

5.2 Korollar

$$\left.\begin{array}{l} T_{\text{BOOL}} \\ T_{\text{NAT}} \\ T_{\text{STRING}} \\ T_{\text{BINTREE}} \\ T_{\text{INT}} \\ T_{\text{SET}} \end{array}\right\} \textit{ ist korrekt bzgl. } \left\{\begin{array}{ll} \mathsf{BOOL} & \textit{(vgl. Beispiel 4.10.3)} \\ \mathsf{NAT} & \textit{(vgl. Beispiel 4.10.4)} \\ A^* & \textit{(vgl. Beispiel 2.29.3)} \\ B & \textit{(vgl. Beispiel 2.29.4)} \\ \mathbb{Z} & \textit{(vgl. Beispiel 4.10.1)} \\ \mathcal{P}(A) & \textit{(vgl. Beispiel 4.10.2)} \end{array}\right.$$

Zum Nachweis der Korrektheit in den Beispielen 2.29 und 4.10 wurde jeweils ausgenutzt, dass nach Theorem 2.20 und 4.9 ein Homomorphismus $h\colon T_{SPEC} \to \mathsf{A}$ bereits explizit bekannt ist, die Isomorphie also allein den Beweis der Bijektivität von h erfordert. Ein anderer Ansatzpunkt für die Verifikation der Korrektheit ist die Eindeutigkeit initialer Algebren bis auf Isomorphie gemäß Theorem 2.25.2; denn ist A initiale *SPEC*-Algebra, so ist A isomorph zu T_{SPEC}. Dazu wäre zu zeigen, dass eine gegebene *SIG*-Algebra A die Gleichungen *EQ* erfüllt und die Initialitätseigenschaft besitzt. Eine ausführliche, systematische Darstellung dieser und weiterer Korrektheitskriterien muss unterbleiben, da sie den Rahmen dieses Skripts sprengt. (Einige Hinweise dazu findet man in [EKP78, GN78, TWW78]).

Nach Definition muss bei obigem Korrektheitsbegriff die Semantik einer Spezifikation immer in allen Datenbereichen und in allen Operationen mit der intendierten Algebra verglichen werden. Das macht den Begriff etwas unhandlich. Diese mangelnde Flexibilität lässt sich dadurch beseitigen, dass die Überprüfung der Isomorphie auf ausgewählte Teile der Spezifikation, auf eine Unterspezifikation beschränkt werden kann. Für die Algebra T_{SPEC} bedeutet das, die nicht ausgewählten Teile vorübergehend auszublenden, zu vergessen.

5.3 Definition

Seien $SPEC_0 = <S_0, OP_0, EQ_0>$ und $SPEC_1 = <S_1, OP_1, EQ_1>$ Spezifikationen mit $SPEC_0 \subseteq SPEC_1$, d.h. $S_0 \subseteq S_1$, $OP_0 \subseteq OP_1$ und $EQ_0 \subseteq EQ_1$.

1. Für jede $SPEC_1$-Algebra A ist dann der $SPEC_0$-*Anteil* A_{SPEC_0} (auch *Vergissbild* oder *Redukt* genannt) als $SPEC_0$-Algebra definiert durch:
   (i) $(\mathsf{A}_{SPEC_0})_s := \mathsf{A}_s$ für alle $s \in S_0$,
   (ii) $op_{\mathsf{A}_{SPEC_0}} := op_{\mathsf{A}}$ für alle $op \in OP_0$.
2. Die Spezifikation $SPEC_1$ ist *korrekt* bzgl. einer OP_0-Algebra A, wenn der $SPEC_0$-Anteil der Quotiententermalgebra T_{SPEC_1} isomorph ist zu A:

$$(T_{SPEC_1})_{SPEC_0} \cong \mathsf{A}.$$

Der besondere Vorteil des verallgemeinerten Korrektheitsbegriffs ist, dass die Semantik einer Spezifikation nicht nur mit Datentypen verglichen werden kann, deren Herkunft nicht präzisiert ist, die es beim Neuentwurf von abstrakten Datentypen auch schwerlich geben

wird, sondern insbesondere auch mit einem vorausgegangenen Entwurfsschritt. Diese Situation ergibt sich gerade, wenn man die Vergleichsalgebra A aus obiger Definition als T_{SPEC_0} wählt, und wird im nächsten Abschnitt unter dem Begriff Extension ausführlicher behandelt.

Bei dem eingeführten Korrektheitsbegriff verwundert manche, dass die Intention in Form einer vollständigen Algebra angegeben werden muss, um die Semantik daran messen zu können, statt einfach Anforderungen zu stellen und deren Erfülltheit zu überprüfen. Bei diesem Einwand wird übersehen, dass das Spezifikationskonzept mit den Gleichungen ja gerade sprachliche Mittel zur Formulierung von Anforderungen bereitstellt. Als Gleichungen gegebene Anforderungen innerhalb einer Spezifikation (und dort gehören sie hin) werden aber von der Semantik automatisch erfüllt, so dass sich ein derartiger Korrektheitsbegriff erübrigt.

6 Konzepte einer Spezifikationssprache für abstrakte Datentypen

Die Untersuchungen der vorangegangenen Abschnitte lassen sich als Sprachkern einer Spezifikationssprache für abstrakte Datentypen deuten. Die Syntax ist gegeben durch die Spezifikationen in der mathematischen Version der Definition 4.1 und der sprachlichen der Bemerkung 4.2; die Semantik ist definiert durch die Quotiententermalgebren und charakterisiert durch die Eigenschaft der Initialität und Operationserzeugtheit (vgl. Theorem 4.9). Daneben gibt es die Ebene der Korrektheit (und Korrektheitsbeweise), die im vorigen Kapitel angesprochen ist.

Für den Entwurf größerer Systeme (mit einer großen Zahl an Operationen und Gleichungen) ist der Sprachkern jedoch zu eng, da er Strukturierungen und Gruppierungen logisch zusammengehörender Operationen und Gleichungen nicht im ausreichenden Maße gestattet, da das ungeordnete Nebeneinander sehr vieler Operationen und Gleichungen Übersichtlichkeit und Verständlichkeit praktisch ausschließt. Erforderlich wird also eine Spracherweiterung um Konzepte, die einen schrittweisen, strukturierten Entwurf abstrakter Datentypen unterstützen.

Ein erstes Konstrukt dieser Art ist die *KOMBINATION*. Ihr liegt die Beobachtung zugrunde, dass in den meisten Spezifikationen ein Teil der Sorten, Operationen und Gleichungen selbst wieder eine Spezifikation bildet. Diese Unterspezifikation lässt sich dann auslagern, wenn man in der ursprünglichen Spezifikation einen Verweis, eine Benennung der Unterspezifikation, zurücklässt. Syntaktisch wird dieser Name den verbleibenden Teilen der Spezifikation vorangestellt. Ein Beispiel mag dieses Prinzip illustrieren.

6.1 Beispiel

Die Spezifikation

spec BinTree1 =
sorts *BinTree*, *Nat*
opns $leaf\colon \to BinTree$
$left, right\colon BinTree \to BinTree$
$both\colon BinTree \times BinTree \to BinTree$
$height\colon BinTree \to Nat$
$0\colon \to Nat$
$succ\colon Nat \to Nat$
$max\colon Nat \times Nat \to Nat$
eqns $height(leaf) = 0$
$height(left(B)) = height(right(B)) = succ(height(B))$
$height(both(B, B')) = succ(max(height(B), height(B')))$
$max(0, n) = max(n, 0) = n$
$max(succ(m), succ(n)) = succ(max(m, n))$

umfasst die aus Beispiel 2.4.2 bekannte Spezifikation BinTree und eine Spezifikation des Maximums zweier natürlicher Zahlen, die selbst das in Beispiel 1.1.2 spezifizierte Nat enthält. Ein strukturierter Entwurf der Höhe binärer Bäume lässt sich deshalb auch so schreiben:

spec BinTree2 = BinTree + Nat1
opns $height\colon BinTree \to Nat$
eqns $height(leaf) = 0$
$height(left(B)) = height(right(B)) = succ(height(B))$
$height(both(B, B')) = succ(max(height(B), height(B')))$

mit

spec Nat1 = Nat
opns $max\colon Nat \times Nat \to Nat$
eqns $max(0, n) = max(n, 0) = n$
$max(succ(m), succ(n)) = succ(max(m, n))$

BinTree2, das als Kombination von BinTree mit Nat1 und der Höhe definiert ist, unterscheidet sich von der unstrukturierten Spezifikation BinTree1 eigentlich nur in der Sortierung von Sorten, Operationen und Gleichungen. Es liegt daher nahe, als Semantik von BinTree2 gerade den von BinTree1 erzeugten abstrakten Datentyp zu verwenden bzw. allgemein die Semantik einer *KOMBINATION* auf die Spezifikation zurückzuführen, die entsteht, wenn alle vorkommenden Sorten, Operationen und Gleichungen zusammengefasst werden und damit die Strukturierung aufgelöst wird.

6.2 Definition

1. $COMB = SPEC_1 + <S_2, OP_2, EQ_2>$ ist eine *Kombination*, wenn $SPEC_1 = <S_1, OP_1, EQ_1>$ und $SPEC = <S_1 \cup S_2, OP_1 \cup OP_2, EQ_1 \cup EQ_2>$ Spezifikationen sind.
2. Als *Semantik* ist einer Kombination $COMB$ die von der Spezifikation $SPEC$ erzeugte Quotiententermalgebra zugeordnet:

$$T_{COMB} := T_{SPEC}.$$

3. Eine Kombination $COMB$ ist *korrekt* bzgl. A, wenn die Spezifikation $SPEC$ korrekt bzgl. A ist.

6.3 Bemerkung

1. $SPEC$ ist genau dann eine Spezifikation, wenn die Operationen von OP_2 Argumente und Werte in $S_1 \cup S_2$ haben und die Gleichungen in EQ_2 nur Operationen aus $OP_1 \cup OP_2$ verwenden.

   Ein wichtiger Spezialfall liegt vor, wenn $<S_2, OP_2, EQ_2>$ selbst eine Spezifikation bildet wie im Falle BinTree + Nat1 in Beispiel 6.1; dann ist die Kombination nur eine gemeinsame Hülle für eigentlich völlig voreinander unabhängige Datentypen, die aber zusammen verwendet werden, um beispielsweise neue Operationen zu definieren. Ein Beispiel dafür ist die Höhe in BinTree2.
2. Die sprachliche Version der Syntax einer Kombination ist durch die folgenden beiden Schemata gegeben:

   spec Spec = Spec1 + Spec2

   und

   spec Spec = Spec1
   sorts ...
   opns ...
   eqns ...

   wobei Spec1 und Spec2 beliebige Spezifikationen (auch Kombinationen) sind und die Schlüsselwörter **sorts**, **opns** und **eqns** weggelassen werden können, wenn die Listen dahinter leer sind.

Um das Prinzip zu verdeutlichen, seien neben den Beispielen in 6.1 weitere angegeben.

6.4 Beispiel

1. Ein schrittweiser Entwurf der ganzen Zahlen, bei dem erst die positiven und dann die negativen erzeugt werden, lässt sich wie folgt realisieren.

spec INT1 =
sorts *Int*
opns $0\colon \rightarrow Int$
$succ\colon Int \rightarrow Int$

spec INT2 = INT1
opns $pred\colon Int \rightarrow Int$
eqns $pred(succ(X)) = X$
$succ(pred(X)) = X$

2. Um die Potenzmenge einer Datenmenge zu spezifizieren, kann man eine Datenmenge mit einem Mechanismus kombinieren, der endliche Teilmengen bildet:

spec DATA =
sorts *Data*

verwendet in

spec SET(DATA) = DATA
sorts *Set*
opns $\emptyset\colon \rightarrow Set$
$insert\colon Data \times Set \rightarrow Set$
eqns $insert(a, insert(a, M)) = insert(a, M)$
$insert(a, insert(b, M)) = insert(b, insert(a, M))$.

Diese und viele andere Beispiele zeigen, dass bereits das vergleichsweise einfache Konzept *KOMBINATION* flexibel und komfortabel Gliederungen und Modularisierungen beim Entwurf abstrakter Datentypen gestattet. Die semantischen Eigenschaften dieses Entwurfsprimitivs bergen jedoch einige Gefahren. Die in einer Kombination *COMB* enthaltene Unterspezifikation $SPEC_1$ besitzt eine eigene Semantik T_{SPEC_1}, definiert andererseits gemäß Definition 5.3.1 in der Semantik von *COMB*, T_{COMB}, einen $SPEC_1$-Anteil. Nach Theorem 4.9 gibt es dann einen Homomorphismus $h\colon T_{SPEC_1} \rightarrow (T_{COMB})_{SPEC_1}$, der im allgemeinen weder surjektiv noch injektiv zu sein braucht. Mit anderen Worten kann sich bei der Konstruktion neuer Sorten und Operationen $(< S_2, OP_2, EQ_2 >)$ der dazu verwendete Datentyp ändern, kann er zerstört werden, indem neue Daten in alten Sorten dazukommen (h nicht surjektiv) oder indem verschiedene alte Daten identifiziert werden (h nicht injektiv). Beispielsweise sind durch INT1 in Beispiel 6.4.1 nach Beispiel 2.29.2 die positiven ganzen bzw. natürlichen Zahlen spezifiziert, während der INT1-Anteil in T_{INT2} alle ganzen Zahlen umfasst, da INT2 nach Definition 6.2.2 und Beispiel 4.10.1 die ganzen Zahlen in der Sorte *Int* erzeugt:

$$T_{\text{INT1}} \cong \mathbb{N} \overset{!}{\subset} (\mathbb{Z}, 0, succ) \cong (T_{\text{INT}})_{\text{INT1}} =: (T_{\text{INT2}})_{\text{INT1}}$$

Will man also, dass sich die Semantik einer Spezifikation bei der Erweiterung durch neue Sorten und Operationen nicht ändert, so ist das durch die Kombination nicht automatisch garantiert, sondern muss zusätzlich gefordert werden. Sinnvoll ist zum Beispiel die Forderung, dass BinTree und Nat1 in BinTree2 sowie Nat in Nat1 nicht zerstört werden, weil man sonst nicht davon sprechen könnte, dass die Operation *height* die Höhe binärer Bäume (aus BinTree) misst (in Nat). Die um eine entsprechende semantische Anforderung verschärfte Kombination definiert den Begriff der *EXTENSION*, die auch *ENRICHMENT* genannt wird, wenn keine neuen Sorten dazukommen.

6.5 Definition

1. $EXT = SPEC_1 + < S_2, OP_2, EQ_2 >$ ist eine *Extension*, wenn es eine Kombination ist und der $SPEC_1$-Anteil des von EXT erzeugten Datentyps isomorph ist zur $SPEC_1$-Quotiententermalgebra:
$$T_{SPEC_1} \cong (T_{EXT})_{SPEC_1}.$$
2. Eine Extension EXT wird auch *Enrichment* genannt, wenn S_2 leer ist.

Die Begriffe Extension und Enrichment werden in [GTW78] eingeführt und dort sowie in [EKP78] ausführlich untersucht. Sie stehen darüber hinaus im engen Zusammenhang mit den Überlegungen und Untersuchungen in [Wan77], [Gog78], [TWW78] und [EKW78]. Extension und Enrichment sind in [EKP78] etwas anders definiert als hier, nämlich dass EXT korrekt sein muss bzgl. T_{SPEC_1}. Ein Vergleich von Definition 5.3.2 mit Definition 6.5 zeigt jedoch, dass beide Begriffsbildungen äquivalent sind.

Bisher wurde Spezifizieren weitgehend als ein statischer Vorgang behandelt: eine Spezifikation beschreibt genau einen abstrakten Datentyp. Diese Betrachtungsweise liefert jedoch für das Beispiel Set(Data) in Beispiel 6.4.2 eher sinnlose Ergebnisse: T_{Data} ist die leere Menge, da es keine 0-stelligen Operationen gibt; $T_{\text{Set(Data)}}$ enthält im *Set*-Datenbereich nur ein Element, nämlich die 0-stellige Operation $\emptyset$, da durch *insert* nichts dazukommt. Beide Datentypen sind also vollständig uninteressant, und mit der Potenzmenge hat das nur soviel zu tun, als die Potenzmenge der leeren Menge die leere Menge als einziges Element enthält.

Tatsächlich jedoch beschreibt Set(Data) nicht nur diese triviale Potenzmenge, sondern alle Potenzmengen (Menge aller endlichen Teilmengen), wenn man in Data eine beliebige Menge einsetzt, $\emptyset$ durch die leere Menge interpretiert und *insert* durch Element-Einfügen, wenn man also Data als formalen Parameter auffasst, dessen Sorte man – um im Rahmen der Spezifikationen zu bleiben – durch beliebige Sorten beliebiger Spezifikationen aktualisieren kann. Beispielsweise erhält man die als Kombination geschriebene Spezifikation Set, wenn man in Set(Data) Data durch

spec A =
sorts A
opns $a_i : \rightarrow A$ für $i = 1, \ldots, n$

ersetzt. Die resultierende Spezifikation SET(A) beschreibt nach Definition 6.2.2 und Beispiel 4.10.2 gerade die Potenzmenge über $A = \{a_1, \ldots, a_n\}$. Aktualisiert man dagegen DATA durch NAT –

spec SET(NAT) = NAT
sorts *Set*
opns $\emptyset$: $\rightarrow$ *Set*
insert: *Nat* $\times$ *Set* $\rightarrow$ *Set*
eqns ...

– erhält man als Semantik analog alle endlichen Teilmengen der natürlichen Zahlen. Ebenso kann man DATA gegen SET(...) austauschen, was dann die Potenzmenge einer Potenzmenge beschreibt.

Dieses Beispiel zeigt, dass sich Kombinationen als Parametrisierungen deuten lassen, wobei die ausgezeichnete Unterspezifikation den formalen Parameterteil darstellt. Er lässt sich aktualisieren, indem seine Sorten, Operationen und Gleichungen gegen entsprechende Bestandteile einer beliebigen Spezifikation ausgetauscht werden. Lässt man einmal außer Acht, dass dabei den formalen Sorten und Operationen andere Namen zugeordnet sein können (z.B. ist *Data* in SET(NAT) durch *Nat* ersetzt), so besagt diese Form der Parameterübergabe, dass der aktuelle Parameterteil den formalen als Unterspezifikation umfasst, konzeptionell also auch eine Kombination ist. Die jeweils aktuelle Spezifikation entsteht dann aus der Parametrisierung einfach durch Ersetzen des formalen durch den aktuellen Parameterteil; sie ist also selbst auch eine Kombination, so dass ihre Semantik – abhängig von dem aktuellen Parameter – durch Definition 6.2.2 erklärt ist. Zusammenfassend ergibt das den folgenden Begriff der *PARAMETRISIERUNG*.

6.6 Definition

1. $PARAM = SPEC_1 + < S_2, OP_2, EQ_2 >$ ist eine *schwache Parametrisierung* mit dem *formalen Parameter(teil)* $FORMAL = SPEC_1$, wenn $PARAM$ eine Kombination ist.
2. Jede Kombination $ACTUAL = FORMAL + < S_3, OP_3, EQ_3 >$ ist ein *aktueller Parameter(teil)* zu einer schwachen Parametrisierung.
3. Austauschen des formalen durch einen aktuellen Parameter in einer schwachen Parametrisierung liefert die *aktualisierte Parametrisierung* $PARAM(ACTUAL) = ACTUAL + < S_2, OP_2, EQ_2 >$.
4. Die *Semantik* einer schwachen Parametrisierung ist durch die von $PARAM(ACTUAL)$ für alle aktuellen Parameter $ACTUAL$ erzeugten Datentypen (gemäß Definition 6.2.2) gegeben:
$$ACTUAL \longmapsto T_{PARAM(ACTUAL)}.$$

6.7 Bemerkung

Diese Zuordnung von aktuellen Parametern zu initialen Algebren lässt sich auch aufschreiben als Familie der von den Spezifikationen $PARAM(ACTUAL)$ erzeugten abstrakten

Datentypen, die über alle aktuellen Parameter $ACTUAL$ indiziert ist:

$$\{T_{PARAM(ACTUAL)}\}_{ACTUAL}$$

Die Parametrisierung in obiger Form wird schwach genannt, weil sie semantisch dieselben "Schwächen" besitzt wie die Kombination: Der $ACTUAL$-Anteil des von der aktualisierten Parametrisierung erzeugten Datentyps stimmt im allgemeinen nicht mit der Semantik des aktuellen Parameters überein:

$$T_{ACTUAL} \ncong (T_{PARAM(ACTUAL)})_{ACTUAL}.$$

Schwaches Parametrisieren kann also die Bedeutung der Parameter verändern.

Dass sich dagegen beim Übergang von formalen zu aktuellen Parametern die Semantik des $FORMAL$-Anteils im allgemeinen ändert –

$$(T_{ACTUAL})_{FORMAL} \ncong T_{FORMAL} \ncong (T_{PARAM(ACTUAL)})_{FORMAL}$$

– ist geradezu erwünscht, wie das Beispiel SET(DATA) zeigt. Denn T_{DATA} ist leer, während der DATA-Anteil von A und SET(A) (sowie analog von NAT und SET(NAT)) übereinstimmt mit der Menge $A = \{a_1, \ldots, a_n\}$ (bzw. mit den natürlichen Zahlen).

Will man aber den Effekt vermeiden, dass der aktuelle Parameter in der aktualisierten Parametrisierung zerstört sein kann, so muss man analog zum Übergang von $KOMBINATION$ zu $EXTENSION$ zusätzlich Forderungen stellen.

6.8 Definition
Eine *(starke) Parametrisierung* ist eine schwache Parametrisierung $PARAM$, für die zusätzlich gilt, dass die Parameter erhalten bleiben; d.h. für alle aktuellen Parameter $ACTUAL$ ist

$$T_{ACTUAL} \cong (T_{PARAM(ACTUAL)})_{ACTUAL}.$$

Ein anderer konzeptioneller Mangel des bisherigen Parametrisierungsbegriffs lässt sich nicht so schnell beseitigen, da die dafür erforderlichen algebraischen Hilfsmittel bisher nicht zur Verfügung stehen. Die Semantik hängt von den Aktualisierungen ab; und es ist unklar, wie der Zusammenhang zur eigentlichen Parametrisierung ist, ob sie nicht bereits die Semantik festlegt, wie es wünschenswert wäre. Tatsächlich lässt sich die Semantik gemäß Definition 6.6.4 näher an die gegebene Parametrisierung $PARAM$ mit ihrem formalen Parameter $FORMAL$ heranbringen, wenn man zu folgender Zuordnung übergeht: dem $FORMAL$-Anteil der Semantik von $ACTUAL$ wird der $PARAM$-Anteil der Semantik von $PARAM(ACTUAL)$ zugeordnet,

$$(T_{ACTUAL})_{FORMAL} \longmapsto (T_{PARAM(ACTUAL)})_{PARAM}.$$

Das verdeutlicht, dass sich die Semantik einer Parametrisierung auch durch eine geeeignete Zuordnung von $FORMAL$-Algebren zu $PARAM$-Algebren beschreiben ließe. Diese Idee

wird in [TWW78] verfolgt, wo als adäquates Beschreibungsmittel der Funktorbegriff der Kategorientheorie (siehe z.B. [GTW75]) herangezogen wird. In dieser Arbeit werden vor allem die semantischen Aspekte von (starken) Parametrisierungen einschließlich Korrektheit behandelt und am Beispiel SET(DATA) erläutert, das gegenüber der hier angegebenen Fassung noch um einige Operationen erweitert ist. Schwache Parametrisierungen werden darüber hinaus in [Ehr78] untersucht; auf diese Arbeit geht die Parameterübergabe in Definition 6.6.2 und 6.6.3 zurück. Außerdem sind "data type extensions" im Sinne von Wand [Wan77] tatsächlich Parametrisierungen.

Es bleibt festzuhalten, dass sich Parametrisierungen von Kombinationen und Extensionen nur durch ihre dynamische Interpretation unterscheiden, wie sich der Definition 6.6 entnehmen lässt, dass alle in diesem Abschnitt vorgestellten Konzepte dasselbe syntaktische Grundmuster besitzen:

$$< S_1, OP_1, EQ_1 > = SPEC_1 \subseteq \left\{ \begin{array}{c} COMB \\ EXT \\ PARAM \end{array} \right\} = SPEC_1 + < S_2, OP_2, EQ_2 >$$

Bemerkenswerterweise lassen sich auch weitere Sprachkonstrukte, die einen schrittweisen Entwurf abstrakter Datentypen unterstützen, aus der Kombination durch geeignete Modifikationen beziehungsweise veränderte Interpretationen gewinnen. Dazu gehören eine konsistente Fehlerbehandlung (vgl. [GTW78]), Implementierungen abstrakter Datentypen (vgl. [GTW78, GN78, Ehr78]), algebraische Spezifikationsschemata (vgl. [EKW78]) und das Prozedurkonzept in CLEAR (vgl. [BG77]). Diese Hilfsmittel zum strukturierten, schrittweisen Spezifizieren und Implementieren, denen man in den zitierten Arbeiten nicht immer ansieht, dass sie syntaktisch und konzeptionell mit der Kombination verwandt sind, können im Rahmen dieses Skripts nicht behandelt werden.

Der aktuelle Stand der Dinge, der in seinen Grundelementen nicht wesentlich über das hier Beschriebene hinausgeht, kann in [Ore99] nachgelesen werden, wo auch neuere Literaturangaben zu finden sind.

Literatur

[BG77] R.M. Burstall and J.A. Goguen. Putting theories together to make specifications. In *Proc. Int. Conf. Artificial Intelligence*, Boston, 1977.

[DF77] Ernst Denert and Reinhold Franck. *Datenstrukturen.* Bibliographisches Institut, Mannheim u.a., 1977.

[Ehr78] H.-D. Ehrich. On the theory of specification, implementation and parametrization of abstract data types. Forschungsbericht, Dortmund, 1978.

[EKP78] Hartmut Ehrig, Hans-Jörg Kreowski, and Peter Padawitz. Stepwise specification and implementation of abstract data types. In *Proc. 5th Int. Colloquium on Automata, Languages and Programming*, Udine, 1978.

[EKW78] Hartmut Ehrig, Hans-Jörg Kreowski, and H. Weber. Algebraic specification schemes for data base systems. In *Proc. 4th Int. Conf. on Very Large Data Bases*, Berlin, 1978.

[GN78] J.A. Goguen and F. Nourani. Some algebraic techniques for proving correctness of data type implementation. Technical report, 1978.

[Gog78] J.A. Goguen. Some design principles and theory for OBJ-O, a language to express and execute algebraic specifications of programs. Technical report, Los Angeles, 1978.

[GTW75] J.A. Goguen, J.W. Thatcher, and E.G. Wagner. An introduction to categories, algebraic theories and algebras. IBM Research Report RC–5369, 1975.

[GTW78] J.A. Goguen, J.W. Thatcher, and E.G. Wagner. An initial algebra approach to the specification, correctness and implementation of abstract data types. In R. Yeh, editor, *Current Trends in Programming Methodology. IV: Data Structuring*. Prentice Hall, New Jersey, 1978. Auch erschienen als IBM Research Report RC–6487, 1976.

[Ore99] Fernando Orejas. Structuring and modularity. In E. Astesiano, H.-J. Kreowski, and B. Krieg-Brückner, editors, *Algebraic Foundations of Systems Specification*, pages 159–200. Springer, 1999.

[TWW78] J.W. Thatcher, E.G. Wagner, and J.B. Wright. Data type specification: Parameterization and the power of specification techniques. In *Proc. 10th SIGACT Symposium on Theory of Computing*, San Diego, 1978.

[Wan77] M. Wand. Final algebra semantics and data type extensions. Technical Report 65, Indiana Univ., Comp. Sci. Dept., 1977.