Brauer, Johannes

Working Paper

Typen, Objekte, Klassen - Teil 3: Semantik abstrakter Datentypen

Arbeitspapiere der Nordakademie, No. 2011-08

Provided in Cooperation with:
Nordakademie - Hochschule der Wirtschaft, Elmshorn

Suggested Citation: Brauer, Johannes (2011) : Typen, Objekte, Klassen - Teil 3: Semantik abstrakter Datentypen, Arbeitspapiere der Nordakademie, No. 2011-08, Nordakademie - Hochschule der Wirtschaft, Elmshorn

This Version is available at:
https://hdl.handle.net/10419/67100

ARBEITSPAPIERE DER NORDAKADEMIE
ISSN 1860-0360

Nr. 2011-08

Typen, Objekte, Klassen – Teil 3: Semantik abstrakter Datentypen

Prof. Dr.-Ing. Johannes Brauer

Dezember 2011

Eine elektronische Version dieses Arbeitspapiers ist verfügbar unter:

http://www.nordakademie.de/arbeitspapier.html

NORDAKADEMIE
HOCHSCHULE DER WIRTSCHAFT

Köllner Chaussee 11
25337 Elmshorn
http://www.nordakademie.de

Typen, Objekte, Klassen – Teil 3: Semantik abstrakter Datentypen

Johannes Brauer

12. Januar 2012

Inhaltsverzeichnis

In diesem dritten Teil einer Reihe von Arbeitspapieren[1] wird schwerpunktmäßig die algebraische Spezifikation abstrakter Datentypen behandelt. Ausgehend von den Problemen einer Fixierung auf die syntaktischen Aspekte von Datentypen erfolgt eine Abgrenzung verschiedener Interpretationen des Begriffs *Abstrakter Datentyp*. Vor- und Nachteile verschiedener Implementierungsvarianten werden behandelt. Anschließend werden kurze Hinweise auf weiterführende Literatur und Spezifikationssprachen gegeben. Die Bedeutung formaler Methoden für die Lehre wird angerissen.

[1] Brauer, J.: *Typen, Objekte, Klassen – Teil 1: Grundlagen*. Arbeitspapier 2009-05, NORDAKADEMIE Hochschule der Wirtschaft, Juni 2009

1 Einleitung

Der Begriff *Abstrakter Datentyp*[2] ist unter der Überschrift *Abstraktionsorientierte Sicht auf Datentypen* als Mittel der Datenabstraktion bereits eingeführt worden. Er ist stark geprägt von der Entwicklung der Programmiersprachen[3] in der 1970er Jahren und assoziiert Abstraktion hauptsächlich mit der Trennung von Schnittstelle und Implementierung. Dabei werden durch die Schnittstelle der Typname und die Signaturen der Operationen, die auf Exemplare des Typs angewendet werden dürfen, spezifiziert. Die Datenrepräsentation und die Implementierung der Operationen werden verborgen.

[2] vgl.
Brauer, J.: *Typen, Objekte, Klassen – Teil 2: Sichtweisen auf Typen*. Arbeitspapier 2010-05, NORDAKADEMIE Hochschule der Wirtschaft, Mai 2009

[3] z. B. MODULA-2, ADA oder CLU

Eine genauere Definition des Begriffs *Abstrakter Datentyp* wird in Kapitel 3 vorgenommen.

In diesem Sinne könnten auch vordefinierte Datentypen wie INTEGER oder REAL als abstrakte Datentypen betrachtet werden, denn

- die Operationen können ohne Kenntnis ihrer Implementierung verwendet werden;

In der üblichen Programmiersprachen-Terminologie ist der Begriff aber vom Programmierer selbst definierten Datentypen vorbehalten.

- die maschineninterne Repräsentation der Werte ist (innerhalb gewisser Grenzen) unerheblich. Jedenfalls ist sie unsichtbar.

Es gibt aber einen wichtigen Unterschied zwischen den vordefinierten Datentypen und vom Programmierer selbst definierten abstrakten Datentypen. Während die Semantik der Operationen eines vordefinierten Datentyps durch die Semantik der Programmiersprache exakt definiert ist, verrät das `definition module` eines in MODULA-2 programmierten abstrakten Datentyps über die Semantik der Operationen nichts. Jede Implementierung der Operationen, die der Signatur der Schnittstelle entspricht, wäre zulässig, auch wenn sie der Intention des Datentyps nicht entspricht. Wir haben es hier mit einem rein „syntaktischen" Typbegriff zu tun.

z. B. die arithmetischen Operationen des Typs INTEGER

Zu einer wohlgeformten Typenlehre gehört aber auch, Möglichkeiten aufzuzeigen, wie die Semantik der Operationen eines abstrakten Datentyps implementierungsunabhängig präzise spezifiziert werden können. Nur so kann die Voraussetzung dafür geschaffen werden, die Korrektheit einer Implementierung zu beweisen.

2 *Probleme eines „syntaktischen" Typbegriffs*

Die Vorteile der Verwendung abstrakter Datentypen für die Unterstützung einer modularen Programmentwicklung durch die Trennung von Schnittstelle und Implementierung wurden in Teil 2 der vorliegenden Reihe von Arbeitspapieren[4] bereits erläutert. Allein die durch die Definition eines abstrakten Datentyps geschaffene Möglichkeit, diesen neuen Datentyp wie einen vordefinierten Typ verwenden zu können, ist unzweifelhaft von großem Nutzen.[5]

Die Definition eines Datentyps ausschließlich über seine syntaktische Schnittstelle[6] erlaubt es dem Programmierer aber

- zu einer Schnittstellenspezifikation verschiedene, möglicherweise semantisch inkompatible Implementierungen anzugeben oder
- zu verschiedenen Schnittstellenspezifikationen, die eine unterschiedliche Semantik intendieren, eine identische Implementierung anzugeben.

Als Beispiel für den ersten Fall betrachten wir das folgende Java-Interface, das einen Typ `Tuple` für Objekte mit zwei Komponenten und einer Additionsoperation definiert:

```
public interface Tuple {
    Object first();
    Object second();
    Tuple add(Tuple p);}
```

Eine mögliche Implementierung ist die folgende Klasse `Point`:

```
public class Point implements Tuple {
    private int x;
    private int y;
```

[4] Brauer, J.: *Typen, Objekte, Klassen – Teil 2: Sichtweisen auf Typen*. Arbeitspapier 2010-05, NORDAKADEMIE Hochschule der Wirtschaft, Mai 2009

[5] vgl. in diesem Zusammenhang auch Sivilotti, P. A. und M. Lang: *Interfaces first (and foremost) with Java*. In: *Proceedings of the 41st ACM technical symposium on Computer science education*, SIGCSE '10, S. 515–519, New York, NY, USA, 2010. ACM

[6] d. h. durch seinen Namen und die Signaturen der Operationen, aber ohne Spezifikation ihrer Semantik.

```
    public Point(int xCoord, int yCoord) {
        x = xCoord;
        y = yCoord;}

    public Object second() {
        return y; }

    public Object first() {
        return x;}

    public Tuple add(Tuple p) {
        return new Point((Integer) this.first()
                         + (Integer) p.first(),
                         (Integer) this.second()
                         + (Integer) p.second());}}
```

Die folgende Implementierung von `Tuple` durch die Klasse `Fraction` erfüllt alle syntaktischen Forderungen der Schnittstellendefinition:

Es muss nicht besonders betont werden, dass das Interface `Tuple` und die hier vorgenommenen Implementierungen durch die Klassen `Point` und `Fraction` inhaltlich eher abwegig sind. Es geht hier aber nur um die prinzipielle Möglichkeit, zwei semantisch inkompatible Implementierungen ein und desselben Interfaces anzulegen.

```
public class Fraction implements Tuple {
    private int numer;
    private int denom;

    public Fraction(int numerator, int denominator) {
        numer = numerator;
        denom = denominator; }

    public Object first() {
       return numer; }

    public Object second() {
       return denom;}

    public Tuple add(Tuple p) {
        int num;
        int den;
        den = (Integer) this.second() * (Integer) p.second();
        num = (Integer) this.first() * p.second
              + (Integer) p.first() * this.second;
        return new Fraction(num, den);}}
```

Die unterschiedliche Implementierung der Methode `add` in den Klassen `Point` und `Fraction` entspricht natürlich den Erwartungen hinsichtlich des Verhaltens von Punkten bzw. rationalen Zahlen.[7] Die beiden Klassen `Point` und `Fraction` das Interface `Tuple` implementieren zu lassen, mag als eine „böswillige“ Nutzung der Möglichkeiten von Java angesehen werden. Dennoch ist es problematisch, dass auf der Grundlage der oben angegebenen Klassendefinitionen z. B. die folgende Anweisungssequenz syntaktisch korrekt ist:

[7] Auf das Kürzen der Summe von zwei Brüchen wurde in der `add`-Methode von `Fraction` verzichtet.

```
p = new Point(2, 3);
f = new Fraction(3, 6);
s1 = f.add(p);
s2 = p.add(f);
```

Dabei sei vorausgesetzt, dass die Variablen `p`, `f`, `s1` und `s2` alle vom deklarierten Typ `Tuple` sind. Man beachte, dass `s1.first()` und `s2.first()` unterschiedliche Ergebnisse liefern.

An diesem Beispiel wird deutlich, dass bei der Definition eines Typs[8] auf die Spezifikation der Semantik der Operationen zu verzichten, zumindest fragwürdig ist. Betrachtet man hingegen die Semantik der Operationen als Bestandteil einer Typdefinition, wären die von den Klassendefinitionen `Point` und `Fraction` erzeugten Typen wegen der unterschiedlichen Implementierung der `add`-Methode nicht kompatibel, bzw. das Interface `Tuple` wäre unvollständig, da die Spezifikation der Operationssemantiken fehlt.

[8] hier durch das Interface `Tuple`

An dieser Stelle ist der Einwand, dass Java-Interfaces nicht in erster Linie der Definition von abstrakten Datentypen dienen, sondern als Ersatz fehlender Mehrfachvererbung eine Form der Polymorphie unterstützen sollen, nicht unberechtigt. In diesem Fall ist unterschiedliches Verhalten beim Senden der gleichen Nachricht (z. B. `add`) gerade erwünscht.

Als Beispiel für zwei „isomorphe“ Interfaces, die eine unterschiedliche Semantik zwar intendieren, aber eben nicht gewährleisten können, sollen die beiden folgenden dienen:

```
public interface Queue {
    public void put(Object element);
    public Object get();
    public void init();}

public interface Stack {
    public void push(Object element);
    public Object pop();
    public void init();}
```

Die Charakterisierung dieser beiden Interfaces als „isomorph“ soll nur darauf hindeuten, dass sie sich nur in ihrem Namen und der Benennung der Operationen unterscheiden, im Übrigen aber strukturgleich sind.

Die Namensgebung legt natürlich nahe, dass eine Implementierung von `Queue` das FIFO[9]- und eine Implementierung von `Stack` das LIFO-Prinzip[10] realisieren sollte, aber der rein syntaktische Typbegriff bietet keine Handhabe dafür, die Korrektheit der Implementierungen formal sicherzustellen.

[9] first in first out

[10] last in first out

Im Kapitel 4 wird ein „semantischer“ Typbegriff eingeführt, bei dem die Typspezifikation die präzise Definition der Semantik der Operationen einschließt, ohne dafür eine Implementierung angeben zu müssen.

3 *Begriffliche Abgrenzung*

Die Ausführungen im vorangegangenen Kapitel dienten zunächst einmal der „Sensibilisierung“ für die Probleme, die mit Typspezifikationen ohne Angabe der Semantik der Operationen verbunden sein können. An dieser Stelle sollen die in der Literatur vorkommenden Definitionen des Begriffs *Abstrakter Datentyp* gegeneinander abgegrenzt werden. Leider sind diese Definitionen nicht immer miteinander verträglich, was bei der Vermittlung dieses Begriffs im Unterricht problematisch ist.

Ausgangspunkt der Betrachtung soll die folgende „gängige“ Definition des Begriffs sein:

Ein *Abstrakter Datentyp* ist eine Menge von Werten in Verbindung mit einer Menge von Operationen auf diesen Werten.

Unter einem strengen Blickwinkel müssen die *Werte*, von denen hier die Rede ist, als unveränderlich angesehen werden. Insofern repräsentieren abstrakte Datentypen mehr oder weniger „theoretische Konzepte", die letztlich unserer Gedankenwelt entspringen. Die Anwendung von Operationen auf Exemplaren eines abstrakten Datentyps liefert immer einen neuen Wert, niemals wird ein Wert „verändert".

Auf der anderen Seite beschäftigt sich die Programmierung fortwährend damit, einen realen Weltausschnitt im Rechner zu rekonstruieren. Dies führt dazu, dass Programme mit Daten operieren, die stellvertretend für real existierende Objekte stehen, wie z. B. Personen, Fahrzeuge oder Bankkonten. Derartige Daten können daher nicht als Exemplare abstrakter Datentypen in diesem strengen Sinne betrachtet werden. Denn für sie ist kennzeichnend, dass sie durch die Anwendung von Operationen verändert werden.

Die Gutschrift eines Betrags auf ein Konto führt zu einer Zustandsänderung des Kontos und nicht zu einem neuen Konto.

Platonische Typen nennen Buck und Stucki[11] abstrakte Datentypen in dem genannten strengen Sinn. Kennzeichnend für sie ist, dass es sich um unveränderliche Werte handelt. Das Musterbeispiel für unveränderliche Werte sind Zahlen. So kann der von einer Programmiersprache üblicherweise bereitgestellte Datentyp *Integer* als platonischer Typ betrachtet werden: Er ist definiert durch eine Menge unveränderlicher Werte[12] und einer Menge wohldefinierter Operationen auf diesen Werten. Ob z. B. auch ein Datentyp *String* als platonischer Typ betrachtet werden kann, hängt von seiner Implementierung in der jeweiligen Programmiersprache ab.

In diesem Aufsatz findet eine grundlegende Auseinandersetzung mit diesem Thema statt:
[11] Buck, D. und D. J. Stucki: *The hidden injuries of overloading 'ADT'*. In: *Proceedings of the 40th ACM technical symposium on Computer science education*, SIGCSE '09, S. 256–259, New York, NY, USA, 2009. ACM

[12] eine Teilmenge der ganzen Zahlen

Der in Kapitel 2 gezeigte Datentyp `Fraction` ist als platonischer Typ realisiert.

Klassische Datenstrukturen, wie Stapel, Listen oder Bäume werden in vielen Literaturquellen, die sich mit abstrakten Datentypen beschäftigen, gerne als typische Vertreter abstrakter Datentypen eingeführt. Wie im Kapitel 4 noch detailliert dargelegt wird, können diese abstrakt durchaus als platonische Typen betrachtet werden. Da derartige Datenstrukturen aus Effizienzgründen aber meistens als veränderliche Objekte implementiert werden, ist eine begriffliche Trennung durchaus sinnvoll. Buck und Stucki bezeichnen diese Variante abstrakter Datentypen als *Container*.

Steht die Trennung von Schnittstelle und Implementierung im Vordergrund[13], wie sie in diversen Programmiersprachen vorgesehen ist, kommt eine dritte Variante der Verwendung des Begriffs *Abstrakter Datentypen* ins Spiel. Sie ist insofern sehr unscharf, als jedwede Implementierung, die syntaktisch zur Schnittstelle „passt", erlaubt ist. Neben der Implementierung als platonische Typen sind aber beliebige andere Implementierungsformen möglich.

[13] vgl. Kapitel 2

Sie wird von Buck und Stucki als *interface* bezeichnet.

Im folgenden Kapitel werden abstrakte Datentypen immer als platonische Typen betrachtet. Das gilt auch für Spezifikationen

von Typen, die im Sinne von Buck und Stucki als *Container* zu bezeichnen sind. Auf Fragen der effizienten Implementierung solcher Container wird in Abschnitt 4.5 näher eingegangen.

4 *Formale Spezifikation abstrakter Datentypen*

Unter einer *Spezifikation* einer Software wird im Rahmen des Software-Lebenszyklus zum einen der Prozess verstanden, der zu einer Beschreibung der von einer Software zu erfüllenden Aufgabe führt, zum anderen das Ergebnis dieses Prozesses in Form eines geeigneten Artefakts. Eine solche Spezifikation beschreibt, *was* eine Software tun soll, aber nicht *wie* und stellt im Idealfall das Dokument dar, das für Kunden und Hersteller eine gemeinsame Grundlage bzw. Sicht auf den zu implementierenden Funktionsumfang der Software dar.

Dafür muss eine Spezifikation sowohl präzise – um eine korrekte Implementierung zu ermöglichen – wie auch verständlich – damit sie auch für „Nicht-Softwaretechniker" lesbar ist – sein. Leider ist dies nicht immer einfach: Beispielsweise ist eine umgangssprachliche Spezifikation in der Regel verständlicher als eine formale; umgekehrt ist eine formale Spezifikation eher frei von Ungenauigkeiten und Mehrdeutigkeiten. *Formal* bedeutet, dass die Spezifikation in einer formalen Sprache mit wohldefinierter Semantik abgefasst ist.

Da beide Ziele nur schwer gleichzeitig zu erreichen sind, besteht der Spezifikationsprozess meistens aus mehreren Schritten. Ausgehend von einer informellen Beschreibung der Kundenwünsche wird über mehrere Stufen hinweg eine formale Spezifikation entwickelt, die dann häufig als *Entwurfsspezifikation* bezeichnet wird.

Der Konflikt zwischen Präzision einerseits und Verständlichkeit andererseits tritt nicht nur bei der Spezifikation von Datentypen auf; er ist die Herausforderung an die Informatik schlechthin.

Aufbauend auf dem Konzept der Modularisierung[14] wird für die folgenden Betrachtungen ein Software-System als Sammlung von Modulen, die ihrerseits als abstrakte Datentypen verstanden werden, aufgefasst. Die formale Spezifikation von Software wird hier demnach als formale Spezifikation abstrakter Datentypen verstanden[15]. Das Ergebnis ist die präzise Definition von Syntax *und* Semantik der Schnittstelle eines Moduls.

[14] vgl. Abschnitt *Abstraktionsorientierte Sicht auf Datentypen* in
BRAUER, J.: *Typen, Objekte, Klassen – Teil 2: Sichtweisen auf Typen*. Arbeitspapier 2010-05, NORDAKADEMIE Hochschule der Wirtschaft, Mai 2009

[15] Für eine tiefer gehende Betrachtung dieses Themas beachte man
LOECKX, J., H.-D. EHRICH und M. WOLF: *Specification of Abstract Data Types*. Wiley-Teubner, 1996

> **Anmerkung:** An dieser Stelle wird eingeräumt, dass bei dieser Sichtweise auf ein Software-System der Begriff *Abstrakter Datentyp*, gemessen an den Darstellungen in Kapitel 3, in einem „umfassenden" Sinn verwendet wird. D. h. insbesondere, dass auch die Kategorie der *Interfaces* mit eingeschlossen ist. Nicht platonische Typen werden also nicht ausgeschlossen. Auf die Probleme ihrer Implementierung wird in Abschnitt 4.5 eingegangen.

Zunächst seien wichtige Eigenschaften von formalen Spezifikationen zusammengefasst:

- Eine *Spezifikation* stellt eine implementierungsunabhängige Beschreibung eines Softwaresystems dar; und zwar unabhängig von Programmiersprache und Rechnersystem.

- Eine Spezifikation ist dann wiederverwendbar (*generisch*) in dem Sinne, dass sie für irgendeine Programmiersprache und irgendeinen Rechnertyp verwendbar ist. Insofern sind Spezifikationen ausdrucksstärker als Programme.
- Eine Spezifikation eines abstrakten Datentyps legt das Verhalten (die Semantik) der Operationen des Datentyps fest, ohne eine Implementierung anzugeben.

Die Vorteile einer formalen Spezifikationsmethode sind in erster Linie:

1. Durch den Rückgriff auf bekannte mathematische Methoden kann das Vertrauen in die Korrektheit der Spezifikation erhöht werden.
2. Erst durch eine formale Spezifikation wird die Voraussetzung geschaffen, die Korrektheit einer Implementierung zu beweisen.

4.1 Algebraische Spezifikation von abstrakten Datentypen

Betrachten wir als erstes Beispiel einen abstrakten Datentyp für Stacks[16]. Eine erste informelle Beschreibung könnte wie folgt aussehen:

- Ein Stack ist eine Folge von Datenelementen, die alle vom gleichen Typ sind.
- Daten werden der Folge hinzugefügt oder aus ihr entfernt, und zwar immer am selben Ende der Folge, genannt *top of stack*. Nur auf das Top-Element kann zugegriffen werden.

Eine „Analyse“ dieses Textes führt zur Identifikation von drei Operationen für die Bearbeitung von Stacks: `top`, `pop` und `push`. Deren Wirkungsweise wird nun etwas detaillierter beschrieben:

- Die Operation `top` liefert als Ergebnis das Element des Stacks, das an der Top-Position steht.
- Die Operation `push` liefert, aufgerufen mit einem Stack und einem Datenelement, einen Stack mit dem neuen Datenelement an der Top-Position als Ergebnis.
- Die Operation `pop` nimmt einen Stack und liefert einen neuen[17] Stack ohne das Datenelement an der vorigen Top-Position.

Ganz im Sinne der Datenabstraktion ist in den obigen Aussagen nur von Operationen, aber nicht von Daten die Rede. Stacks erhält man nur durch Anwendung von Operationen.[18] Die Operationen `push` und `pop` liefern zwar einen Stack als Resultat, verlangen aber auch nach einem Stack als Argument. Es muss also auch eine Operation geben, die eine Art „Ur-Stack“ erzeugt, von dem ausgehend weitere Stacks erzeugt werden können. Wir führen daher zwei weitere Operationen ein:

- createstack: liefert einen leeren Stack
- `isemptystack`: liefert true, wenn der Stack leer ist, sonst false

Vor jeder anderen Operation mit einem Stack muss ein solcher mit `createstack` erzeugt werden. Erst dann kann mit `push` ein Element auf den Stack gebracht werden. Die Anwendung von `top` oder `pop`

[16] Auf die deutsche Bezeichnung *Stapel* wird hier verzichtet, da sie auch in der einschlägigen deutschen Fachliteratur nur selten benutzt wird.

Welcher Natur die *Datenelemente* sind, die auf dem Stack gespeichert werden sollen, ist insofern unerheblich, als die Wirkung der Stackoperationen von ihr nicht abhängig ist.

[17] Derartige Formulierungen legen nahe, dass hier implizit von einer Implementierung als platonischer Typ (vgl. Kapitel 3) ausgegangen wird.

[18] Man sagt auch: Der Datentyp ist *operationserzeugt*.

ist nur auf einen nicht leeren Stack sinnvoll.

EINE WEITERE STEIGERUNG DER PRÄZISION in der Beschreibung der Operationen lässt sich dadurch erzielen, dass man die Wechselbeziehungen der Stack-Operationen untereinander explizit formuliert:

- Wenn ein Datenelement `d` mit `push` auf den Stack gebracht wird und die Operation `top` auf diesen Stack angewendet wird, dann wird das Datenelement `d` als Ergebnis von `top` zurückgeliefert.
- Wenn ein Stack mit `createstack` erzeugt wird und anschließend `isemptystack` auf diesen Stack angewendet wird, dann ist das Ergebnis `true`.

Ähnliche Wechselbeziehungen lassen sich natürlich auch für andere Paare[19] von Operationen formulieren. Es handelt sich dabei um Aussagen, die immer wahr sind, bzw. von jeder Implementierung des Datentyps erfüllt werden müssen.

Diese Aussagen lassen sich nun recht einfach formalisieren. Die o. g. Wechselbeziehung zwischen `push` und `top` könnte mathematisch auch so formuliert werden:

$$\bigwedge_{d \in elem, s \in stack} top(push(s,d)) = d$$

DATENTYPEN, wie wir sie bisher betrachtet haben, bestehen aus einem oder mehreren Datenbereichen[20], die auch als *Sorten* bezeichnet werden, und einer Menge von Operationen auf diesen Datenbereichen. Derartige Strukturen sind aus mathematischer Sicht *Algebren* im Sinne der *universellen Algebra*[21]. Diese Wesensverwandtschaft zwischen Datentypen und Algebren legt die Technik der *algebraischen Spezifikation abstrakter Datentypen* nahe.

Im folgenden Abschnitt wird schrittweise in diese Technik am Beispiel des Datentyps *Stack* eingeführt.

4.2 Algebraische Spezifikation eines Stacks

Für das Aufschreiben von algebraischen Spezifikationen werden in der Literatur verschiedene Notationen verwendet. Wir lehnen uns hier an eine einfache Darstellungsweise an, wie sie z. B. in ähnlicher Form im Informatik-Handbuch[22] verwendet wird.

Der folgende Text zeigt die *Signatur* einer Stack-Spezifikation:

```
datatype  stack
    sorts stack, item, bool
    operations
        push:        stack, item → stack
        pop:         stack       → stack
        top:         stack       → item
        createstack:             → stack
        isempty:     stack       → bool
end
```

[19] z. B. zwischen `pop` und `push`: Wenn mit `push` ein Datenelement auf den Stack `s` gebracht wird und anschließend auf den neu entstandenen Stack `pop` angewendet wird, ist das Resultat der Stack `s`.

Dabei bezeichnet *elem* die Menge, deren Elemente auf dem Stack gespeichert werden sollen, und *stack* die Menge aller Stacks.

[20] Im Stack-Beispiel sind das die Datenelemente und die Stacks selbst

[21] Die universelle Algebra beschäftigt sich im Gegensatz zur klassischen Algebra insbesondere mit Strukturen mit mehreren Basismengen und mehrstelligen Operationen.

[22] RECHENBERG, P. und G. POMBERGER (Hrsg.): *Informatik-Handbuch.* Carl Hanser Verlag, 2 Aufl., 1999

Unter der Signatur versteht man – vereinfacht gesprochen – die Benennung der Operationen[23] zusammen mit ihren *Eingabe-* oder *Argumentsorten*[24] und ihrer *Ergebnis-* oder *Ausgabesorte*[25]. Die *Sorten* benennen abstrakte Datenbereiche, die in einer Implementierung durch konkrete Mengen zu ersetzen wären. Die Operationen können demnach als abstrakte Abbildungen aufgefasst werden, deren Argumentsorten den Definitionsbereichen und deren Ergebnissorte dem Wertebereich entspricht.

[23] hinter dem Schlüsselwort `operations`

[24] links vom Rechtspfeil

[25] rechts vom Rechtspfeil

In der Terminologie der abstrakten Datentypen entsprechen die Sorten im Grunde den Typen. Das heißt, dass die Signatur der Operation `push` auch so gelesen werden kann:

> Die Operation `push` erwartet als Argumente je ein Exemplar der Typen `stack` und `item` und liefert als Resultat ein Exemplar des Typs `stack`.

Anscheinend gibt es keinen Wesensunterschied zwischen *Sorten* und *Typen*, zumindest ist er für die praktische Anwendung algebraischer Spezifikationen nicht relevant. Insofern könnte die **`sorts`**-Zeile in der Signatur auch weggelassen werden. Da aber die Typen `item` und `bool` in der `stack`-Spezifikation nicht definiert, sondern quasi importiert werden, behalten wir auch in den folgenden Spezifikationen die Aufzählung aller vorkommenden Sorten bei.

Operationen ohne Argumentsorte[26] heißen auch *Konstanten* der Spezifikation. Das Aufschreiben einer Konstanten stellt die einfachste Form eines *Terms* einer Spezifikation dar. Der einfachste Stack-Term lautet also: `createstack`. Argumente von Operationen werden in runde Klammer hinter den Operationsnamen geschrieben, z. B.

```
isempty(createstack).
```

[26] z. B. `createstack`; wenn alle Exemplare des Typs ausschließlich durch Operationen erzeugt werden können, bezeichnet man den Typ als *operationserzeugt*.

Terme dürfen auch Variablen enthalten:

```
push(createstack, i).
```

Terme ohne Variablen heißen *Grundterme*.

Die Variable `i` steht hier gemäß der Signatur für einen Term, der ein Exemplar des Typs `item` repräsentiert.

Mit der Signatur eines Datentyps verharren wir aber auf der syntaktischen Ebene. Um die Semantik der Operationen des Datentyps `Stack` zu spezifizieren, werden nun die oben informell eingeführten Wechselbeziehungen zwischen den Operationen durch Gleichungen ausgedrückt. Das führt dann zu der folgenden `Stack`-Spezifikation:

Übertragen auf Programmiersprachen, entspricht einer Signatur in etwa dem, was mit einem Java-Interface ausgedrückt wird.

```
datatype  stack
   sorts stack, item, bool
   operations
       push:        stack, item → stack
       pop:         stack       → stack
       top:         stack       → item
       createstack:             → stack
       isempty:     stack       → bool
   equations for all s in stack, i in item:
   {1} top(push(s,i)) = i
   {2} pop(push(s,i)) = s
   {3} isemptystack(createstack) = true
   {4} isemptystack(push(s,i)) = false
end
```

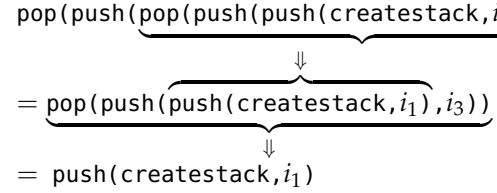
Gleichungen werden hinter dem Schlüsselwort `equations` aufgeschrieben. Durch `for all ...` werden die in den Gleichungen benutzten Variablen allquantifiziert. Text in geschweiften Klammern ist Kommentar.

Mit Hilfe der Gleichungen können nun bestimmte Terme in andere Terme äquivalent umgeformt werden. Zum Beispiel kann der Term

`pop(push(pop(push(push(createstack,`i_1`),`i_2`)),`i_3`))`

$i_1, i_2 i_3$ seien Elemente der Sorte `item`.

durch zweimaliges Anwenden der Gleichung 2 wie folgt umgeformt werden:



Das Ergebnis dieser Umformung entspricht der intuitiven Vorstellung, dass der Ausgangsterm letztlich einem Stack entspricht, auf dem sich nur das Element i_1 befindet.

Durch die Gleichungen ist nun die Semantik des Datentyps `Stack` im Einklang mit der oben angegebenen informellen Spezifikation definiert.

Diese Übereinstimmung ist natürlich nicht formalisierbar.

Die Signatur des Datentyps kennt drei Operationen mit der Ergebnissorte `stack`[27]. Konstruktiv betrachtet, erzeugen wir `stack`-Exemplare, in dem wir, ausgehend vom durch den Grundterm `createstack` repräsentierten leeren Stack, mit `push` weitere `items` einem Stack hinzufügen. D. h. jeder Stack kann mit diesen beiden Operationen erzeugt werden. Das wiederum bedeutet, dass ein Term, der die Operation `pop` enthält, keinen Stack definieren kann, der nicht auch ohne Anwendung von `pop` aufgeschrieben werden kann. Die Gleichung 2 sorgt genau dafür, dass ein beliebiger Term[28] mit `pop` in einen äquivalenten Term ohne `pop` umgewandelt werden kann.

[27] `createstack, push` und `pop`

[28] Wir betrachten selbstverständlich nur syntaktisch korrekte Terme.

Ein Stack-Term, der nur die einmalige Anwendung von `createstack` und die beliebig häufige Anwendung von `push` enthält, liegt also in einer Art „Normalform“ vor. Dies legt es nahe, aus der Menge der Operationen einer Spezifikation die Teilmenge auszuzeichnen, deren Elemente in der Normalform der Terme vorkommen dürfen, d. h. mit denen alle Exemplare des Datentyps erzeugt werden können. Diese Teilmenge bezeichnet man auch als die *aufbauenden Operationen* oder *Konstruktoren*. Für die Stack-Spezifikation schreiben wir nun:

```
datatype  stack
  sorts stack, item, bool
  constructors
       push:        stack, item → stack
       createstack:             → stack
  operations
       pop:         stack       → stack
       top:         stack       → item
       isempty:     stack       → bool
   equations for all s in stack, i in item:
   {1} top(push(s,i)) = i
   {2} pop(push(s,i)) = s
   {3} isemptystack(createstack) = true
   {4} isemptystack(push(s,i)) = false
end
```

Es ist nun naheliegend, zu verlangen, dass die Gleichungen einer Spezifikation erlauben, aus Termen Operationen, die nicht zu den Konstruktoren zählen, durch Anwenden der Gleichungen zu eliminieren. Das führt zu folgender Faustregel für die „Vollständigkeit“[29] einer Spezifikation:

> Man schreibe mindestens je eine Gleichung auf, die die Wechselwirkung jeder nicht-aufbauenden[30] mit jeder aufbauenden[31] Operation beschreibt.

[29] Mathematisch betrachtet, gibt es den Begriff der *Wohlgeformtheit* einer Spezifikation, auf dessen exakte Definition hier verzichtet wird. Vgl. Kapitel 5

[30] den `operations`

[31] den `constructors`

Wenden wir diese Regel auf die Stack-Spezifikation an, bräuchten wir mindestens sechs Gleichungen. In der Tat lässt sich z. B. der Term

```
pop(createstack)
```

mithilfe der existierenden Gleichungen nicht weiter vereinfachen.

In der Regel wird man wohl den Versuch, von einem leeren Stack das oberste Element zu entfernen, als unzulässige Anwendung der Operation `pop` ansehen. Die informelle Spezifikation macht für diesen Fall keine expliziten Angaben. Eine formale Spezifikation darf aber hier keine „Lücke“ aufweisen. In Abschnitt 4.3 wird das Problem der Fehlerbehandlung in algebraischen Spezifikationen behandelt.

> **Anmerkung**: Mathematisch gesehen, versteht man unter einem Datentyp die Angabe einer Algebra, die zur Signatur eines abstrakten Datentyps passt. Dazu gehört, für die Sorten konkrete Mengen anzugeben und für die Operationen ihre Wirkung auf die Elemente dieser Mengen zu definieren. Die Angabe einer solchen Algebra definierte dann die Semantik des Datentyps. Bei der Spezifikation von abstrakten Datentypen will man sich jedoch gerade nicht auf eine konkrete Repräsentation festlegen. Es ist jedoch möglich, für jeden gleichungsdefinierten Datentyp eindeutig eine Algebra der Terme der Spezifikation anzugeben, die die Semantik des abstrakten Datentyps definiert. Diese Algebra heißt *Quotiententermalgebra*.

Detaillierte Darstellungen der Bedeutung von Termalgebren für die exakte Definition der Semantik algebraischer Spezifikationen sind z. B. zu finden in:

Ehrig, H., B. Mahr, M. Grosse-Rhode, F. Cornelius und F. Zeitz: *Mathematisch-strukturelle Grundlagen der Informatik*. Springer-Verlag Berlin Heidelberg New York, 2. Aufl., 2001

4.3 Fehlerbehandlung in algebraischen Spezifikationen

Die unsinnige Stack-Operation `pop(createstack)` könnte man in der algebraischen Spezifikation durch Hinzufügen der folgenden Gleichung berücksichtigen:

```
{5} pop(createstack) = createstack
```

Das hätte zur Folge, dass eine fehlerhafte Operation als solche nicht „erkannt“ würde oder – anders ausgedrückt – die Anwendung von `pop` auf den leeren Stack wirkungslos bliebe.

Diese Möglichkeit, den Fehlerfall zu behandeln, besteht aber nicht, wenn wir den Term `top(createstack)` betrachten. Da die Signatur verlangt, dass ein Element der Sorte `item` als Resultat geliefert wird, bleibt hier nur die Möglichkeit, eine neue `item`-Konstante (z. B. `erroritem`) einzuführen. Verfährt man für die fehlerhafte Anwendung von `pop` in ähnlicher Weise durch Einführung einer Fehlerkonstanten `errorstack`, ergibt sich die folgende Variante der `stack`-Spezifikation:

```
datatype  stack
   sorts stack, item, bool
   constructors
        push:        stack, item → stack
        createstack:             → stack
        errorstack:              → stack
   operations
        pop:         stack       → stack
        top:         stack       → item
        isempty:     stack       → bool
    equations for all s in stack, i in item:
    {1} top(push(s,i)) = i
    {2} pop(push(s,i)) = s
    {3} isemptystack(createstack) = true
    {4} isemptystack(push(s,i)) = false
    {5} pop(createstack) = errorstack
    {6} top(createstack) = erroritem
end
```

Dass an dieser Stelle die Konstante `erroritem` nicht ebenfalls eingeführt wird, ist damit zu begründen, dass angenommen wird, dass diese bereits im hier importierten abstrakten Datentyp `item` als „Konstruktor“ für `items` definiert ist.

Das Auftreten von `errorstack` bzw. `erroritem` würde in einer Implementierung des Datentyps sinnvollerweise als Ausnahme (exception) behandelt.

Die konsequente Anwendung der im Abschnitt 4.2 angegebenen Regel für das Aufstellen der Gleichungen erfordert die Angabe von drei weiteren für den Konstruktor `errorstack`:

```
{7} isemptystack(errorstack) = true
{8} pop(errorstack) = errorstack
{9} top(errorstack) = erroritem
```

Die Festlegung auf `true` ist an dieser Stelle willkürlich.

Konsequenterweise müsste man nun auch das Ergebnis der Anwendung von `push` auf `erroritem` bzw. `errorstack` spezifizieren, z. B. durch folgende Gleichungen:

```
{10} push(errorstack, i) = errorstack
{11} push(s, erroritem)  = errorstack
```

Es ist ersichtlich, dass eine algebraische Spezifikation durch die Gleichungen für die Fehlerbehandlung erheblich „aufgebläht“ wird. Zu den ursprünglich vier Gleichungen für die „Gutfälle“ wurden zunächst zwei Gleichungen[32] in Anwendung der in Abschnitt 4.2 aufgestellten Faustregel hinzugefügt. Dies ist schon deshalb sinnvoll, da – wie gezeigt – Gleichung 5 auch anders lauten könnte. Die übrigen Gleichungen[33], die nur durch die Einführung der Fehlerkonstanten zustande kommen, haben aber keinen praktischen Nutzen, wenn wir voraussetzen, dass jedes Auftreten einer Fehlerkonstanten zu einer Ausnahmebehandlung in der Implementierung führt. Auf die Angabe derartiger Gleichungen werden wir daher verzichten.

[32] Gleichungen 5 und 6

[33] hier die Gleichungen 7 bis 11

4.4 *Weitere Beispiele algebraisch spezifizierter Datentypen*

Als Ergänzung im Hinblick auf die in Kapitel 2 erwähnten „isomorphen“ Interfaces[34] wird hier nun auch noch die algebraische Spezifikation eines weiteren Datentyps der Kategorie *Container*[35] angegeben.

[34] `Stack` und `Queue`

[35] vgl. Kapitel 3

ABSTRAKTER DATENTYP FÜR WARTESCHLANGEN:

```
datatype queue
   sorts queue, item, boolean
   constructors
         emptyqueue:               →  queue
         append:       queue, item →  queue
   operations
         remove:       queue       →  queue
         front:        queue       →  item
         isemptyqueue: queue       →  boolean
   equations for all q in queue, i in item
   {1} remove(emptyqueue) = emptyqueue
   {2} remove(append(q, i)) = if isemptyqueue(q)
                              then emptyqueue
                              else append(remove(q), i)
   {3} front(emptyqueue) = erroritem
   {4} front(append(q, i)) = if isemptyqueue(q)
                             then i
                             else front(q)
   {5} isemptyqueue(emptyqueue) = true
   {6} isemptyqueue(append(q, i)) = false
end
```

Hier wird die Festlegung getroffen, dass das Entfernen aus der leeren Warteschlage nicht fehlerhaft ist.

Für die Formulierung der in den Spezifikationen für die Operationen `remove` und `front` notwendigen Fallunterscheidungen sind die Gleichungen 2 und 4 als *bedingte Gleichungen* aufgeschrieben. Alternativ könnten auch jeweils zwei Gleichungen formuliert werden:

```
...
 {2a} remove(append(emptyqueue, i)) = emptyqueue
 {2b} remove(append(append(q, i), j)) = append(remove(append(q,i)), j)

...
 {4a} front(append(emptyqueue, i)) = i
 {4b} front(append(append(q, i), j)) = front(append(q, i))
...
```

ALGEBRAISCHE SPEZIFIKATIONEN PLATONISCHER TYPEN

Der in Kapitel 2 als Java-Interface eingeführte Typ `Tuple` kann bezüglich seiner Implementierungen `Fraction` und `Point` als platonischer Typ angesehen werden. Den Typ `Tuple` algebraisch zu spezifizieren, ist aber nicht sinnvoll, da er keine eigene Operationssemantik besitzt. Die wird ja erst durch die beiden Klassen `Fraction` und `Point` in unterschiedlicher Weise formalisiert. Deswegen werden hier nun die Spezifikationen für zwei abstrakte, platonische Datentypen `fraction` und `point` angegeben:

```
datatype point
   sorts point, integer
   constructors
         makePoint: integer integer → point
```

```
   operations
         xCoord: point              → integer
         yCoord: point              → integer
         add:    point point        → point
   equations for all p, q in point, i, j in integer
   {1} xCoord(makePoint(i, j)) = i
   {2} yCoord(makePoint(i, j)) = j
   {3} add(p, q) = makePoint(xCoord(p) + xCoord(q)
                             yCoord(p) + yCoord(q)
end

datatype fraction
   sorts fraction, integer
   constructors
         makeFraction: integer integer → fraction
   operations
         numer: fraction              → integer
         denom: fraction              → integer
         add:   fraction fraction     → fraction
   equations for all f, g in fraction, i, j in integer
   {1} numer(makeFraction(i, j)) = i
   {2} denom(makeFraction(i, j)) = j
   {3} add(f, g) = makeFraction(numer(f) * denom(g)
                                + numer(g) * denom(f),
                                denom(f) * denom(g)
end
```

Angesichts dieser Spezifikationen wird deutlich, dass die Typen `point` und `fraction` eigentlich nichts[36] gemeinsam haben, so dass der Versuch, einen gemeinsamen „Obertyp“ `Tuple` zu schaffen, abwegig erscheinen muss.

[36] abgesehen von der strukturellen Ähnlichkeit

4.5 Implementierungen abstrakter Datentypen

Platonische Typen[37] auch als solche zu implementieren, scheint auf der Hand zu liegen. Bei Verwendung einer objektorientierten Programmiersprache wie Smalltalk oder Java bedeutete dies, dass für die Exemplarvariablen keine Set-Methoden zur Verfügung gestellt werden dürfen. Operationen auf Exemplaren platonischer Typen erzeugen neue Exemplare, niemals werden vorhandene verändert.

[37] wie z. B. `point` und `fraction` aus Abschnitt 4.4

Für Typen, die aus mathematischen Mengen abgeleitet sind, sollte nur die Implementierung als platonischer Typ in Betracht kommen. Ihre Zahl ist allerdings begrenzt.

Die Implementierung von Container-Typen muss hingegen differenzierter betrachtet werden. Die in den Abschnitten 4.2 bzw. 4.4 spezifizierten Typen `stack` und `queue` weisen zunächst – wie alle algebraischen Spezifikationen – einen funktionalen Charakter auf. Darauf deuten auch die „rekursiven“ bedingten Gleichungen[38]

Die Operation `remove` liefert, angewendet auf eine `queue`, eine neue `queue` ohne das ursprünglich erste Element.

[38] Gleichungen 2 und 4 der `queue`-Spezifikation in Abschnitt 4.4

der queue-Spezifikation hin. Für eine funktionale Implementierung z. B. des Typs queue können die Gleichungen mit wenigen syntaktischen Anpassungen in eine funktionale Programmiersprache übertragen werden. Gegen eine derartige Implementierung können zwei Gründe sprechen:

1. Bei einem Container besteht in vielen Fällen sicherlich die intuitive Vorstellung von einem einmal erzeugten Objekt, das durch nachfolgende Operationen verändert wird.
2. Die funktionale Implementierung kann ineffizient sein, wenn statt der Modifikation eines vorhandenen Containers bei jeder Operation ein neues Exemplar erzeugt wird.

Ein Exemplar des Typs queue könnte zur Simulation einer realen Warteschlange verwendet werden, die dann selbstverständlich Zustandsänderungen erfährt.

Interessanterweise enthält die Java-Klassenbibliothek zwei Klassen für die Implementierung von Zeichenketten:

`String` implementiert Strings als Exemplare eines platonischen Typs, während die Exemplare von

`StringBuffer` veränderbar sind.

Die Existenz von `StringBuffer` ist sicher in erster Linie der Effizienzsteigerung geschuldet.

Ein Beispiel für eine platonische Implementierung einer Datenstruktur sind die Listen, wie sie in Programmiersprachen wie Lisp oder Scheme verwendet werden. Alle Standardfunktionen liefern stets neue Listen, die Argumente bleiben unverändert.

> **Anmerkung:** Benutzte man Lisp-Listen zur Implementierung des abstrakten Datentyps queue, stellt die Operation append das Effizienzproblem dar: Das Anfügen eines Elements an die Warteschlange verursacht eine asymptotische Laufzeit $O(n)$[39]. Für Warteschlangen und andere Datenstrukturen[40] gibt es aber funktionale Implementierungen, deren amortisierte Kosten nicht größer als $O(1)$ sind.

[39] mit n = Anzahl der Elemente in der Warteschlange

[40] siehe z. B.: Okasaki, C.: *Purely Functional Data Structures*. Cambrigde University Press, 1998

In den meisten Fällen werden Container wohl imperativ implementiert, d. h. die Operationen verändern den Zustand der Datenstruktur. Dagegen ist auch nichts einzuwenden, solange man sich der Konsequenzen bewusst ist. Eine Konsequenz soll an dieser Stelle nur angedeutet werden. Während für eine funktionale Implementierung von queue die Aussage

```
{q = q1}
append(q, e)
{q = q1}
```

$q, q1$ sind Exemplare von queue.

immer wahr ist, gilt dies für eine imperative Implementierung nicht, da die Operation append ihr Argument q verändert. Dies hat z. B. zur Folge, dass Korrektheitsbeweise schwieriger werden. Der Vorteil der imperativen Implementierung besteht darin, dass die Standard-Operationen für Warteschlangen ohne weiteres mit asymptotischen Kosten von $O(1)$ realisiert werden können.

4.6 Zusammenfassung

Eine *Entwurfsspezifikation* einer Software in Form von algebraisch spezifizierten, abstrakten Datentypen beschreibt ein abstraktes

Modell des Systems, das durch die folgenden Eigenschaften charakterisiert ist:

- Es ist *konstruktiv*[41]. Das bedeutet, dass keinerlei Festlegungen hinsichtlich einer Datenrepräsentation getroffen werden. Jedes „Datum", d. h. jedes Exemplar eines Datentyps, wird durch eine Folge von Operationen erzeugt.
- Die Semantik der Operationen ist demzufolge unabhängig von der Repräsentation der Daten, d. h. in der Regel axiomatisch, erklärt.
- Sie ist prinzipiell durch Rechner „ausführbar", wenn für das Umwandeln der Terme eines abstrakten Datentyps ein geeigneter Interpreter zur Verfügung steht.
- Sie ist formal und damit prinzipiell einer mathematischen Behandlung zugänglich.

[41] Man spricht auch von *operationserzeugt*.

Man könnte es auch so formulieren: Die Operationenfolge steht stellvertretend für das von ihr erzeugte Datum.

Man spricht auch von *gleichungsdefinierten* Datenypen.

Es handelt sich um einen frühen Prototyp. Vgl. auch Kapitel 5.

Die Rechnerausführbarkeit stellt sicher einen bedeutsamen Vorteil algebraischer Spezifikationen dar, da der Entwickler damit in die Lage versetzt wird, die Wirkung seiner Gleichungen quasi zu testen. Damit lassen sich erste Hinweise gewinnen, ob die spezifizierte Semantik im Einklang mit ihrer Intention steht. Ein weiterer Vorteil ist in ihrem formalen Charakter zu sehen. Erst eine formale Spezifikation ermöglicht, die Korrektheit einer Implementierung zu beweisen. Durch den Verzicht auf eine Festlegung der Datenrepräsentation ist eine algebraische Spezifikation wiederverwendbar und außerdem programmiersprachenunabhängig.

Als Nachteile algebraischer Spezifikationen können angeführt werden:

- Der Umgang mit algebraischen Spezifikationen erfordert viel Erfahrung und Übung.
- Das Arbeiten mit Interpretern für das „Testen" einer Spezifikation will ebenfalls gelernt sein.
- Komplexe Gleichungssysteme sind nicht immer leicht verständlich.
- Die Frage, ob ein Gleichungssystem vollständig ist, lässt sich nicht beantworten.

Die Beantwortung der Frage, ob das, was formal spezifiziert wurde, auch das ist, was fachlich beabsichtigt ist, ist leider keiner formalen Behandlung zugänglich.

Unter softwaretechnischen Gesichtspunkten ist die Tatsache, dass eine Modulschnittstelle[42] nicht nur eine Syntax, sondern auch eine Semantik besitzt, eine Selbstverständlichkeit. Dabei kommt der Semantik eigentlich die größere Bedeutung zu. Diesem Umstand wird in der Praxis der Software-Entwicklung, aber auch in der Lehre zu wenig Beachtung geschenkt.

[42] Der Begriff Modul wird hier mal vereinfachend für *Komponente eines Softwaresystems* benutzt. Das kann ein abstrakter Datentyp, aber z. B. auch ein Webservice sein.

5 *Literatur, Sprachen und Systeme*

In dem Werk *Mathematisch-strukturelle Grundlagen der Informatik*[43] werden im Teil *Algebraische Strukturen* die Zusammenhänge zwischen Datenstrukturen und Algebren sehr grundlegend behandelt.

[43] Ehrig, H., B. Mahr, M. Grosse-Rhode, F. Cornelius und F. Zeitz: *Mathematisch-strukturelle Grundlagen der Informatik*. Springer-Verlag Berlin Heidelberg New York, 2. Aufl., 2001

Hier werden insbesondere auch die Termalgebren behandelt, die die mathematische Grundlage der formalen Definition der Semantik algebraischer Spezifikationen darstellen. Im Kapitel über algebraische Spezifikationen werden auch die Anforderungen erläutert, die an eine Gleichungsmenge eines abstrakten Datentyps zu stellen sind, damit das darauf fußende Termersetzungssystem die Eigenschaften *terminierend* und *konfluent* besitzt. Mit diesen Eigenschaften ist – vereinfacht gesagt – sichergestellt, dass jeder zulässige Term einer Spezifikation durch Anwenden der Gleichungen eindeutig in seine Normalform überführt werden kann.

Mit diesen Eigenschaften ist ein Anhaltspunkt für die „Vollständigkeit" (vgl. Abschnitt 4.2) eines Gleichungssystems gegeben.

Eine kompakte Einführung in gleichungsdefinierte Datentypen und ihre algebraischen Grundlagen wird von Herbert Klaeren und Michael Sperber[44] gegeben.

Jacques Loeckx and Hans-Dieter Ehrich and Markus Wolf[45] behandeln nicht nur die mathematischen Grundlagen axiomatischer Datentypspezifikationen, sondern stellen diese auch in den Kontext der Entwicklung von Software.

[44] KLAEREN, H. und M. SPERBER: *Die Macht der Abstraktion*. B. G. Teubner, 1. Aufl., 2007

[45] LOECKX, J., H.-D. EHRICH und M. WOLF: *Specification of Abstract Data Types*. Wiley-Teubner, 1996

BEISPIELE FÜR SPEZIFIKATIONSSPRACHEN sind das von der „berliner Schule" entwickelte ACT TWO[46] nebst eigener Entwicklungsumgebung oder das von der Common Framework Initiative (CoFI) vorangetriebene CASL[47]: the Common Algebraic Specification Language. Die Ziele der CoFI werden in ihrem Wiki[48] näher erläutert.

[46] EHRIG, H. und B. MAHR: *Fundamentals of Algebraic Specification 2*. Monographs in Theoretical Computer Science. An EATCS Series; 21. Springer Berlin, 1990

[47] MOSSES, P. D. (Hrsg.): *CASL Reference Manual*. Springer Berlin Heidelberg, 2004

[48] `http://www.informatik.uni-bremen.de/cofi/wiki/index.php/CoFI`

DAS SYSTEM *Meta Environment* ist eigentlich ein Werkzeug für die Sprachentwicklung. Es besteht aus

- einer SDF genannten Komponente für Syntaxdefinition und Parsing sowie
- einer Komponente für algebraische Spezifikationen und Termersetzung, die ASF+SDF[49] genannt wird.

Das System ist frei zugänglich und kann z. B. dazu benutzt werden, die in Kapitel 4 aufgeführten Beispiele algebraischer Spezifikationen praktisch umzusetzen und mithilfe des mitgelieferten Interpreters auch zu „testen".

[49] `http://www.meta-environment.org/`

6 *Schlussbemerkungen*

Die Einführung einer formalen Spezifikationsmethode für Datentypen und damit für Software im Allgemeinen stellt einerseits einen wichtigen Baustein zur Vervollständigung einer im Teil 1[50] dieser Reihe von Arbeitspapieren geforderten Typenlehre dar. Andererseits steht die Behandlung formaler Methoden im Programmierunterricht in einem Kontrast zu ihrer Durchdringung der Praxis der Software-Entwicklung. Insofern wird die Frage, ob formale Methoden in Bachelor-Studiengängen der Informatik einen Platz haben sollten, Diskussionsgegenstand bleiben. Grundlegende Kenntnisse hierüber zu vermitteln, ermöglicht es Absolventen aber erst, zukünftige Entwicklungen auf diesem Gebiet wahrzunehmen und einzuschätzen.

[50] BRAUER, J.: *Typen, Objekte, Klassen – Teil 1: Grundlagen*. Arbeitspapier 2009-05, NORDAKADEMIE Hochschule der Wirtschaft, Juni 2009

Dass formale Methoden in der Praxis wenig Beachtung finden, liegt möglicherweise an der nach wie vor mangelnden Unterstützung durch Werkzeuge. Die Entwicklung beweisbar korrekter Software bleibt aber nicht nur auf der Tagesordnung der Wissenschaft[51]. Auch die Software-Industrie beschäftigt sich mit der Entwicklung von Werkzeugen auf diesem Gebiet, wie das Beispiel der Entwicklung von Spec#[52] bei Microsoft Research zeigt.

Vor diesem Hintergrund wäre es fahrlässig, dieses Thema in der Lehre auszusparen. Den passenden Umfang zu ermitteln und geeignete didaktische Konzepte zu entwickeln, bleibt eine Herausforderung.

[51] HOARE, C. A. R. und J. MISRA: *Preface to special issue on software verification*. ACM Comput. Surv., 41:18:1–18:3, October 2009

[52] BARNETT, M., M. FÄHNDRICH, K. R. M. LEINO, P. MÜLLER, W. SCHULTE und H. VENTER: *Specification and verification: the Spec# experience*. Commun. ACM, 54:81–91, Juni 2011

Literatur

[1] BARNETT, M., M. FÄHNDRICH, K. R. M. LEINO, P. MÜLLER, W. SCHULTE und H. VENTER: *Specification and verification: the Spec# experience*. Commun. ACM, 54:81–91, Juni 2011.

[2] BRAUER, J.: *Typen, Objekte, Klassen – Teil 1: Grundlagen*. Arbeitspapier 2009-05, NORDAKADEMIE Hochschule der Wirtschaft, Juni 2009.

[3] BRAUER, J.: *Typen, Objekte, Klassen – Teil 2: Sichtweisen auf Typen*. Arbeitspapier 2010-05, NORDAKADEMIE Hochschule der Wirtschaft, Mai 2009.

[4] BUCK, D. und D. J. STUCKI: *The hidden injuries of overloading 'ADT'*. In: *Proceedings of the 40th ACM technical symposium on Computer science education*, SIGCSE '09, S. 256–259, New York, NY, USA, 2009. ACM.

[5] EHRIG, H. und B. MAHR: *Fundamentals of Algebraic Specification 2*. Monographs in Theoretical Computer Science. An EATCS Series; 21. Springer Berlin, 1990.

[6] EHRIG, H., B. MAHR, M. GROSSE-RHODE, F. CORNELIUS und F. ZEITZ: *Mathematisch-strukturelle Grundlagen der Informatik*. Springer-Verlag Berlin Heidelberg New York, 2. Aufl., 2001.

[7] HOARE, C. A. R. und J. MISRA: *Preface to special issue on software verification*. ACM Comput. Surv., 41:18:1–18:3, October 2009.

[8] KLAEREN, H. und M. SPERBER: *Die Macht der Abstraktion*. B. G. Teubner, 1. Aufl., 2007.

[9] LOECKX, J., H.-D. EHRICH und M. WOLF: *Specification of Abstract Data Types*. Wiley-Teubner, 1996.

[10] MOSSES, P. D. (Hrsg.): *CASL Reference Manual*. Springer Berlin Heidelberg, 2004.

[11] OKASAKI, C.: *Purely Functional Data Structures*. Cambrigde University Press, 1998.

[12] RECHENBERG, P. und G. POMBERGER (Hrsg.): *Informatik-Handbuch*. Carl Hanser Verlag, 2 Aufl., 1999.

[13] SIVILOTTI, P. A. und M. LANG: *Interfaces first (and foremost) with Java*. In: *Proceedings of the 41st ACM technical symposium on Computer science education*, SIGCSE '10, S. 515–519, New York, NY, USA, 2010. ACM.