3

Terme und Algebren

Fast alle formalen Kalküle definieren eine Notation für Formeln, in denen Operanden mit Operationen verknüpft werden, die für den Kalkül spezifisch sind. Arithmetik, Logik, Mengenlehre und auch die Wertebereiche aus Kapitel 2 sind Beispiele für solche Kalküle. In diesem Kapitel führen wir Terme ein, als Grundlage für Formeln aller Kalküle. Der Begriff ist so universell, weil zunächst nur die Struktur der Terme betrachtet und von der Bedeutung der Operation abstrahiert wird. Darauf bauen Begriffe auf zum Umformen von Termen, zur Definition und zur Anwendung von Rechenregeln. Damit können dann Kalküle algebraisch definiert werden. Das gilt nicht nur für bekannte Kalküle wie die Mengenalgebra oder die Boolesche Algebra, sondern auch für neue anwendungsspezifische. So werden z. B. auch die Eigenschaften von dynamisch veränderlichen Datenstrukturen mit Termen und Rechenregeln algebraisch spezifiziert.

In den ersten beiden Abschnitten dieses Kapitels führen wir Terme und ihre Notation ein. Sie bestehen aus Symbolen für Operatoren, Variablen und Konstanten, z.B.

```
a + 5        blau & gelb
```

Auch wenn die verwendeten Symbole hier arithmetische Addition oder das Mischen von Farben suggerieren, verbinden wir zunächst keine Bedeutung mit Termen. Sie werden auch nicht „ausgerechnet“. Wir werden Begriffe einführen, um die Struktur von Termen zu beschreiben:

a + 5 ist ein Term der Sorte *ganzeZahl* mit einem zweistelligen Operator +. Seine beiden Operanden sind auch Terme der gleichen Sorte. *blau & gelb* ist ein Term der Sorte *Farbe*, *&* bezeichnet einen 2-stelligen Operator mit zwei Operanden, die auch Terme der Sorte *Farbe* sind. + und *&* bezeichnen *Operatoren*; *5*, *blau* und *gelb* bezeichnen *Konstanten*, *a* eine *Variable*.

Wir stellen dann unterschiedliche Schreibweisen von Termen vor: Die Beispiele oben sind in der *Infix-Form* notiert; dabei steht der Operator zwischen seinen Operanden. In der *Präfix-Form* steht er stattdessen vor seinen Operanden:

```
+ a 5        &  blau gelb
```

Im Abschnitt 3.3 führen wir den Begriff der *Substitution* ein. Sie regelt, wie man für Variable in einem Term spezielle Terme einsetzen kann. So wird z. B. aus dem Term

```
a + 1 > a
```

durch Substitution von *2* für *a* der Term

```
2 + 1 > 2
```

Dies ist eine Formalisierung des „Einsetzens von Variablen“, wie man es intuitiv vom Umgang mit Variablen kennt.

Schließlich erweitern wir in Abschnitt 3.4 eine Menge von Termen um Rechenregeln zu einer Algebra. Rechenregeln wie

$x + 0 \equiv x$ $\quad$ blau & gelb ≡ grün

identifizieren Terme, die man als gleichbedeutend ansehen will. Man kann sie anwenden, um Terme umzuformen und nach den Gesetzen der jeweiligen Algebra „auszurechnen“. Damit haben wir ein mächtiges Werkzeug, mit dem einerseits klassische Algebren über Mengen, Zahlen oder logische Werte definiert sind. Andererseits kann man damit Operationen dynamisch veränderlicher Datenstrukturen wie Keller oder Listen algebraisch spezifizieren. Sogar für Anwendungen wie den Getränkeautomaten kann man die Abfolge von Bedienoperationen algebraisch spezifizieren.

3.1 Terme

Fast jeder formale Kalkül definiert eine Schreibweise für Formeln, mit denen man Eigenschaften und Zusammenhänge in dem Kalkül ausdrücken kann, z. B. Formeln im Mengenkalkül, in der Arithmetik oder in der Logik. Sie bestehen aus Operatoren, die Operanden verknüpfen. Operanden sind wiederum Formeln oder elementare Konstante oder Variable. Die Operatoren und Konstanten haben eine für den Kalkül spezifische Bedeutung. Wenn wir von der Bedeutung abstrahieren wollen, sprechen wir von Termen statt von Formeln. Wir sehen ihre Operatoren und Konstanten als Symbole an, die nur für sich selbst stehen und keine weitergehende Bedeutung haben. Die strukturellen Eigenschaften von Termen sind unabhängig von der Bedeutung, die sie haben, wenn man sie als Formeln interpretiert. Solche strukturellen Regeln geben an, wie viele Operanden welcher Art ein jedes Operatorsymbol verknüpft. Auch der Umgang mit Variablen in Termen sowie das Zuordnen von Termen, die als gleichbedeutend angesehen werden sollen, wird exakt geregelt, ohne dass auf den speziellen Kalkül und die Bedeutung der zugehörigen Formel Bezug genommen zu werden braucht. In diesem Sinne ist der Begriff der Terme grundlegend und universell für jegliche Art von Formeln.

Im Folgenden führen wir die Begriffe *Sorten* und *Signaturen* ein, um damit strukturell korrekte Terme zu definieren. Dann stellen wir unterschiedliche Notationen von Termen vor.

3.1.1 Sorten und Signaturen

Um strukturell korrekte Terme zu definieren, gibt man zu jedem Operatorsymbol an, wie viele Operanden es verknüpft, zu welcher Sorte von Termen sie gehören und welcher Sorte der gesamte Term angehört. Wir notieren diese Angaben zu Operatoren so, wie wir in Kapitel 2 die Signaturen von Funktionen angegeben haben:

Beispiel 3.1: Operatorbeschreibungen

+:	ARITH	×	ARITH	→ ARITH
<:	ARITH	×	ARITH	→ BOOL
∧:	BOOL	×	BOOL	→ BOOL
true:				→ BOOL
1:				→ ARITH

Da wir bei Termen von ihrer Bedeutung abstrahieren, benennen ARITH und BOOL *Sorten* und nicht *Wertebereiche*.

Definition 3.1: Sorte

Eine Menge von Termen τ kann in disjunkte Teilmengen eingeteilt werden, um die strukturelle Korrektheit der Terme zu definieren. Diese Teilmengen heißen dann ***Sorten der Terme*** *in τ.* ■

In Beispiel 3.1 ist festgelegt, dass ein <-Operator einen Term der Sorte BOOL bildet und zwei Terme der Sorte ARITH verknüpft. 1 ist ein Term der Sorte ARITH und hat keine Operanden. Die Terme

1 < 1 1 < 1 + 1

erfüllen obige Anforderungen; aber die Terme

1 ∧ true + 1

verletzen sie: 1 ist ein Term der Sorte ARITH, aber der linke Operand von ∧ muss der Sorte BOOL angehören. Der Operator + muss zwei Operanden haben.

Definition 3.2: Stelligkeit

Ein Operator ist ***n-stellig*** *mit n ≥ 0, wenn er n Operanden hat.* ***0-stellige*** *Operatoren sind* ***Konstanten****.* ■

Den hier für Terme definierten Begriff der Stelligkeit haben wir schon mit gleicher Bedeutung in Kapitel 2 für Funktionen und Relationen verwendet.

Eine Menge von Strukturbeschreibungen für Operatoren fasst man mit den darin vorkommenden Sorten zu einer Signatur zusammen:

Definition 3.3: Signatur

Eine ***Signatur*** $\sum := (S, F)$ *ist ein Paar aus einer Menge von Sorten S und einer Menge von Strukturbeschreibungen F. Eine Strukturbeschreibung aus F beschreibt ein n-stelliges Operatorsymbol op und hat die Form:*

$op: \quad s_1 \times \ldots \times s_n \rightarrow s_0$ *mit* $s_i \in S$

Sein i-ter Operand muss ein Term der Sorte s_i *sein. Der mit op gebildete Term gehört der Sorte* s_0 *an.* ■

Wir können das Beispiel 3.1 zu einer Signatur $\sum_{3.1} := (S_{3.1}, F_{3.1})$ vervollständigen. Dabei ist

$S_{3.1}$:= {ARITH, BOOL}

und $F_{3.1}$ enthält die in Beispiel 3.1 angegebenen Operatorbeschreibungen.

Eine Signatur mit ausschließlich Booleschen Operatorsymbolen kommt mit einer einzigen Sorte aus:

Beispiel 3.2: Signatur

Signatur $\sum_{Bool} = (S_{Bool}, F_{Bool})$ mit
$S_{Bool} = \{BOOL\}$ und

$$
\begin{array}{lllll}
F_{Bool} = \{ & \text{true:} & & \rightarrow & \text{BOOL}, \\
 & \text{false:} & & \rightarrow & \text{BOOL}, \\
 & \wedge: & \text{BOOL} \times \text{BOOL} & \rightarrow & \text{BOOL}, \\
 & \vee: & \text{BOOL} \times \text{BOOL} & \rightarrow & \text{BOOL}, \\
 & \neg: & \text{BOOL} & \rightarrow & \text{BOOL}\}
\end{array}
$$

Das Beispiel 3.2 definiert zwei *Konstante*, zwei *2-stellige* und einen *1-stelligen Operator*. Alle Terme gehören der einzigen Sorte BOOL an.

Leider wird der Begriff Signatur mit unterschiedlichen Bedeutungen verwendet: Einerseits gemäß Definition 3.3 zur Strukturierung von Termen, andererseits aber auch, um für eine einzelne konkrete Funktion oder einen Operator festzulegen, aus welchem Wertebereich die Parameter und das Ergebnis stammen, siehe Abschnitt 2.7.

Wir können nun die Menge korrekter Terme, die mit den Operatoren einer Signatur erzeugt werden können, induktiv definieren.

Definition 3.4: Korrekte Terme

Sei eine Signatur $\sum = (S, F)$ gegeben. Dann enthält die Menge τ der ***korrekten Terme*** *zu $\sum$ den Term t der Sorte $s \in S$, wenn gilt,*

a) t = v und v ist ein Name einer Variablen der Sorte s, der verschieden ist von allen Operatorsymbolen in F, oder

b) $t = op\ (t_1, t_2, ..., t_n)$ die Anwendung eines Operators op auf Terme t_i mit $n \geq 0$ und $i \in \{1, ..., n\}$, wobei F eine Strukturbeschreibung für op enthält:

$op: s_1 \times s_2 \times ... \times s_n \rightarrow s_0$

und jedes t_i ein korrekter Term der Sorte s_i ist. ■

Definition 3.5: Grundterm

Ein korrekter Term, der keine Variablen enthält, heißt ***Grundterm****.* ■

Nach Definition 3.4 ist

$\neg$ ($\wedge$ (a, true))

ein korrekter Term zur Signatur $\sum_{Bool}$. Wir überprüfen das, indem wir den Term rekursiv gemäß Definition 3.4 in seine Unterterme zerlegen und dabei die korrekte Anwendung der Operatorbeschreibungen prüfen. Wenn wir annehmen, dass *a* eine *Variable* der *Sorte* BOOL ist, ist der Term korrekt.

Wir können die Definition 3.4 auch benutzen, um die Menge τ der *korrekten Terme* zur *Signatur* $\sum_{Bool}$ zu konstruieren:

Wir setzen

$\tau_0 := \{a \mid a \text{ ist eine Variable oder eine Konstante der Sorte BOOL}\}$

$\tau_{i+1} := \tau_i \cup \{ f(t_1, t_2, \ldots t_n) \mid f \text{ ist ein Operator aus F und die } t_j \text{ sind Terme aus } \tau_k, \text{ mit } k \leq i, \text{ deren Sorte zur Beschreibung von f passt}\}$

τ_i enthält dann alle Terme der Schachtelungstiefe *i*.

In Definition 3.4 werden die Terme notiert, indem man das Operatorsymbol der geklammerten Folge der Teilterme voranstellt:

$\neg (\wedge (a, \text{true}))$

Bisher hatten wir das Operatorsymbol bei 2-stelligen Operatoren zwischen seine Operanden geschrieben:

$\neg (a \wedge \text{true})$

Im folgenden Abschnitt definieren wir verschiedene Notationen von Termen und untersuchen ihre Eigenschaften.

3.1.2 Notationen für Terme

In diesem Abschnitt definieren wir vier textuelle und eine grafische Darstellung von Termen. Wir zeigen, welche Eigenschaften die Notationen haben und wie man Terme von einer in eine andere umformen kann.

Definition 3.6: Termnotationen

Ein n-stelliger Term mit der Operation f und den Termen $t_1, t_2, \ldots, t_n$ als Operanden wird notiert in der

a) ***Funktionsform*** *als $f(t_1, t_2, \ldots, t_n)$, wobei die t_i auch in der* ***Funktionsform*** *notiert sind;*
b) ***Präfixform*** *als $f\ t_1\ t_2 \ldots t_n$, wobei die t_i auch in der* ***Präfixform*** *notiert sind;*
c) ***Postfixform*** *als $t_1\ t_2 \ldots t_n\ f$, wobei die t_i auch in der* ***Postfixform*** *notiert sind;*
d) ***Infixform*** *als $(t_1\ f\ t_2)$ für zweistellige Operatoren f, wobei die t_i auch in der* ***Infixform*** *notiert sind. Bei höherer Stelligkeit als 2 kann der Operator f auch aus mehreren Teilen $f_1 \ldots f_{n-1}$ bestehen: $(t_1 f_1\ t_2 \ldots f_{n-1}\ t_n)$*
Wenn die Zuordnung der Operanden zu den Operatoren eindeutig ist, kann

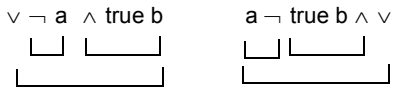
die Klammerung entfallen. Die hier definierte Form heißt ***vollständig geklammert****.* ■

Als Beispiel notieren wir folgenden Term zu Σ_{Bool} in allen vier Formen

Funktionsform:	$\vee (\neg a, \wedge (\text{true}, b))$
Präfixform:	$\vee \neg a \wedge \text{true } b$
Postfixform:	$a \neg \text{true } b \wedge \vee$
Infixform:	$((\neg a) \vee (\text{true} \wedge b))$

In der *Funktionsform* kann man die Operanden eines Operators leicht in der geklammerten Folge nach dem Operator erkennen.

In der *Präfixform* und der *Postfixform* werden keine Klammern verwendet. Wenn man die Stelligkeit der Operatoren kennt, kann man die Operanden ihren Operatoren einfach von innen nach außen zuordnen:



Die *Infixform* benötigt im Allgemeinen Klammern, um Operanden ihren Operatoren eindeutig zuzuordnen. Würde man sie in obigem Beispiel weglassen, also

$\neg a \vee \text{true} \wedge b$

dann könnte man die Operanden auch völlig anders an die Operatoren binden und erhielte einen anderen Term mit anderer Struktur, z. B.

$((\neg (a \vee \text{true})) \wedge b)$

Man kann auch Eindeutigkeit der Termstruktur in der Infixnotation erreichen, wenn man den Operatoren unterschiedliche *Bindungsstärken (Präzedenzen)* zuordnet. Konkurrieren zwei Operatoren um einen Operanden, so gewinnt der mit höherer Präzedenz. In logischen Formeln hat vereinbarungsgemäß $\neg$ höhere Präzedenz als $\wedge$, und dessen Präzedenz ist höher als $\vee$. Damit wäre der Term

$\neg a \vee \text{true} \wedge b$

gleich mit

$((\neg a) \vee (\text{true} \wedge b))$

Man muss dann außerdem noch regeln, ob beim Konkurrieren von Operatoren gleicher Präzedenz der linke oder der rechte gewinnt, also ob sie ihre Operanden *links-* oder *rechtsassoziativ* binden, z. B. bedeutet $a \vee b \vee c$

linksassoziativ	$(a \vee b) \vee c$	und
rechtsassoziativ	$a \vee (b \vee c)$	(linksassoziativ ist gebräuchlicher)

Eine Infixform mit mehrteiligen Operatorsymbolen, wie in Definition 3.6 zugelassen, ist recht ungebräuchlich. Ein Beispiel dafür ist etwa der Operator für bedingte Ausdrücke in

der Programmiersprache C. Er ist 3-stellig, und seine Infixform hat die beiden Teile *?* und *:* , z. B.

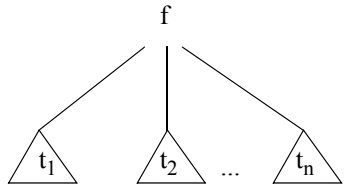
```
i > = 0 ? a [i] : 0
```

Man beachte, dass all diese Regelungen zu Bindungsstärken, Assoziativität und mehrteiligen Operatorsymbolen nur die Infixform betreffen. Die anderen drei Formen kommen ohne sie aus.

Schließlich kann man die Struktur von Termen auch sehr anschaulich grafisch durch Bäume darstellen.

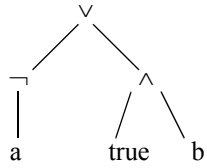
Definition 3.7: Baumdarstellung

*Ein n-stelliger Term mit der Operation f und den Untertermen t_1, t_2, ..., t_n wird als **Baum** durch*



dargestellt, wobei die  *die Baumdarstellung von t_i ist.* ■

Man nennt solche Bäume, deren Knoten mit den Operatoren markiert sind, auch *Kantorowitsch-Bäume*. Das Beispiel zu Definition 3.6 wird dann durch folgenden Baum dargestellt.



Diese Darstellung gibt die Struktur des Terms natürlich ohne weitere Festlegungen eindeutig wieder. Die Baumdarstellung kann man auch gut verwenden, um daraus systematisch die Textformen zu erzeugen. Dazu durchläuft man den Baum *links-abwärts.* Das bedeutet:

- besuche den Wurzelknoten,
- durchlaufe jeden Unterbaum links-abwärts und besuche dann jeweils wieder den Wurzelknoten.

Die Abb. 3.1 gibt solch einen Durchlauf an, die Besuche der Knoten sind als Punkte gekennzeichnet.

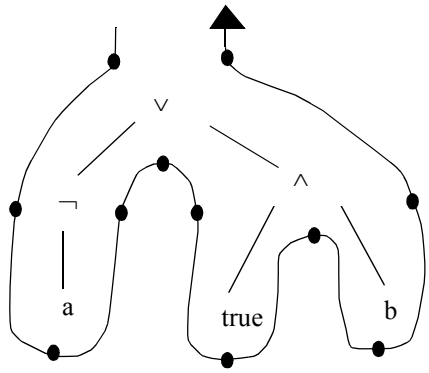


Abbildung 3.1: Links-abwärts-Durchlauf durch einen Baum

Die *Präfixform (Postfixform)* erhält man, wenn man die Inschrift des Knotens beim ersten (letzten) Besuch eines Knotens ausgibt. Die *Funktionsform* erhält man wie die Präfixform, wenn man zusätzlich Klammern und Kommata einfügt.

Die *Infixform* erhält man, wenn man bei mehrstelligen Operatoren beim ersten Besuch die öffnende, beim letzten die schließende Klammer ausgibt und bei den dazwischen liegenden Besuchen die Operatorteile. Bei einstelligen Operatoren gibt man die öffnende Klammer und den Operator beim ersten Besuch und die schließende Klammer beim letzten Besuch aus. Bei 0-stelligen Operatoren wird das geklammerte Symbol bei dem einzigen Besuch ausgegeben. Variablenknoten werden bei dem einzigen Besuch ohne Klammern ausgegeben.

An diesem Verfahren wird auch deutlich, dass die Reihenfolge der Variablen und Konstanten in allen Notationsformen übereinstimmt, im Beispiel: *a true b*.

3.2 Substitution und Unifikation

Eine Variable in einem korrekten Term kann man durch einen Term passender Sorte ersetzen und erhält dann wieder einen korrekten Term. Solche Umformungen sind auch mit Formeln intuitiv bekannt. Eine einfache Rechenregel wie

$$a + a = 2 * a$$

gilt natürlich auch, wenn man für die Variable *a* die Konstante *3* einsetzt:

$$3 + 3 = 2 * 3$$

Solche elementaren Operationen auf Termen heißen *Substitutionen*. Im Folgenden definieren wir Regeln und Schreibweise für ihre Anwendung.

Häufig werden Terme mit Variablen als ein Muster für konkretere Terme angesehen, etwa *a + a* als Muster für *3 + 3*. Die Frage, ob ein Term wie *a + a* als Muster auf einen Term

wie *3 + 3* passt, kann man präziser formulieren: Gibt es eine Substitution von Variablen, die die beiden Terme gleich macht? Dies führt zum Begriff der *Unifikation*, den wir im zweiten Teil dieses Abschnittes einführen.

3.2.1 Substitution

Eine Substitution beschreibt, wie in einem Term vorkommende Variable durch Terme ersetzt werden.

Definition 3.8: Einfache Substitution

*Eine **einfache Substitution** $\sigma = [v / t]$ ist ein Paar aus einer Variablen v und einem Term t zur Signatur $\sum$. v und t gehören beide zu derselben Sorte s.* ■

Beispiele für einfache Substitutionen sind

[a / 3] [a / 5 ∗ b] [a / 5 ∗ b] [x / 2 ∗ x]

Substitutionen werden auf Terme angewandt und liefern als Ergebnisse wieder Terme, in denen einige Variablen ersetzt sind. Die Regeln dafür gibt folgende Definition an.

Definition 3.9: Anwendung einer Substitution

*Die **Anwendung einer einfachen Substitution [v / t]** auf einen **Term** u schreibt man u [v / t]. Ihr Ergebnis ist ein Term, der durch einen der drei folgenden Fälle bestimmt ist:*

a) u [v / t] = t, falls u die zu ersetzende Variable v ist,
b) u [v / t] = u, falls u eine Konstante oder eine andere Variable als v ist,
c) u [v / t] = f (u_1 [v / t], u_2 [v / t], ..., u_n [v / t]),
falls u = f (u_1, u_2, ..., u_n) ■

Wir wenden die Definition 3.9 in allen Schritten einzeln auf das Beispiel

x + x - 1 [x / 2 ∗ b]

in Funktionsform notiert an:

- (+ (x, x), 1) [x / ∗ (2, b)] =
- (+ (x, x) [x / ∗ (2, b)] , 1 [x / ∗ (2, b)]) =
- (+ (x, x) [x / ∗ (2, b)], 1) =
- (+ (x[x / ∗ (2, b)], x [x / ∗ (2, b)]), 1) =
- (+ (∗ (2, b), ∗ (2, b)), 1)

Insbesondere der dritte Fall in Definition 3.9 macht klar, dass in *u* **alle** Vorkommen der Variablen durch den *Term t* ersetzt werden. Außerdem werden die Ersetzungen in jedem Teilterm unabhängig voneinander und nur einmal ausgeführt. D.h. wenn die zu ersetzende *Variable v* auch in dem eingesetzen *Term* vorkommt, dann wird sie nicht nochmals ersetzt, sondern bleibt im Ergebnis stehen, z. B. in

(x + y) [y / y ∗ y] = (x + y ∗ y)

Wir erweitern nun die einfache zu einer mehrfachen Substitution mit entsprechender Bedeutung:

Definition 3.10: Mehrfache Substitution

*In einer **mehrfachen Substitution** $\sigma = [v_1 / t_1, ..., v_n / t_n]$ mit $n \geq 1$ müssen alle Variablen v_i paarweise verschieden sein. Die Paare v_i / t_i müssen jeweils zur selben Sorte s_i gehören. σ wird dann auf einen Term u angewandt:*

a) $u\,\sigma = t_i$, falls $u = v_i$ für ein $i \in \{1, ..., n\}$

b) $u\,\sigma = u$, falls u eine Konstante oder eine Variable ist, die nicht unter v_i, $i \in \{1, ..., n\}$ vorkommt.

c) $u\,\sigma = f(u_1\,\sigma, u_2\,\sigma, ..., u_n\,\sigma)$, falls $u = f(u_1, u_2, ..., u_n)$ ■

Mit einer mehrfachen Substitution werden also mehrere einfache Substitutionen gleichzeitig durchgeführt, z.B.

(x + y) [x / 2 ∗ b, y / 3] = (2 ∗ b + 3)
(x + y) [x / y, y / y ∗ y] = (y + y ∗ y)

Wiederum ist es wichtig, dass alle Unterterme unabhängig und nur einmal substituiert werden.

Da die Anwendung einer Substitution auf einen Term wieder einen Term liefert, kann man auch mehrere Substitutionen hintereinander ausführen, z.B.

(x + y) [x / y ∗ x] [y / 3] [x / a] =
(y ∗ x + y) [y / 3] [x / a] =
(3 ∗ x + 3) [x / a] =
(3 ∗ a + 3)

Das Hintereinanderausführen von Substitutionen liefert im Allgemeinen ein anderes Ergebnis als die mehrfache Substitution mit den gleichen Variablen-Term-Paaren, z.B.

(x + y) [x / y] [y / y ∗ y] = (y ∗ y + y ∗ y)
(x + y) [x / y, y / y ∗ y] = (y + y ∗ y)

Andererseits kann man aber zwei hintereinander auszuführende Substitutionen umrechnen in eine einzige Substitution mit gleicher Wirkung:

Die Hintereinanderausführung der Substitutionen $[x_1 / t_1, ..., x_n / t_n]$ $[y / r]$ hat auf jeden Term die gleiche Wirkung wie die Substitution

- $[x_1 / t_1\,[y / r], ..., x_n / t_n\,[y / r\,]]$ falls $y = x_i$ für ein $i \in \{1, ..., n\}$
- $[x_1 / t_1\,[y / r]\,, ..., x_n / t_n\,[y / r], y / r]$ falls $y \neq x_i$ für alle $i \in \{1, .., n\}$

Damit können wir obiges Beispiel schrittweise umrechnen:

[x / y ∗ x] [y / 3] [x / a] =
[x / (y ∗ x [y / 3]), y / 3] [x / a] =
[x / 3 ∗ x, y / 3] [x / a] =
[x / (3 ∗ x [x / a]), y /(3 [x / a])] =
[x / 3 ∗ a, y / 3]

Natürlich liefert dann das Beispiel das gleiche Ergebnis wie oben

(x + y) [x / 3 ∗ a, y / 3] = (3 ∗ a + 3)

Allgemein gilt für alle Terme *u* und alle Substitutionen σ_1 und σ_2:

$(u\ \sigma_1)\ \sigma_2 = u\ (\sigma_1\ \sigma_2)$

Damit Substitutionen beim Umrechnen und Hintereinanderausführen nicht unnötig komplex werden, führen wir die leere, wirkungslose Substitution ein:

Definition 3.11: Leere Substitution

*[] bezeichnet die **leere Substitution**. Für alle Terme t gilt t [] = t. Außerdem gilt [v/v] = [] für jede Variable v.* ■

Bisher haben wir eine gegebene *Substitution* σ auf einen *Term u* angewandt und damit einen *Term v = u* σ bestimmt. Nun wollen wir stattdessen Fragen der Form untersuchen: Gibt es eine *Substitution* σ, die einen *Term u* in einen *Term v* transformiert? Solch eine Frage stellt sich z.B., wenn man Rechenregeln anwendet. Nehmen wir als Beispiel eine Formulierung des Distributivgesetzes:

a ∗ (b + c) ≡ a ∗ b + a ∗ c

Das Anwenden solcher Rechenregeln wird in Abschnitt 3.4 präzise definiert. Wenn es auf die Formel

2 ∗ (3 + 4 ∗ x)

angewandt werden soll, dann muss man eine Substitution finden, die die linke Seite der Regel auf die anzuwendende Formel transformiert. Solch eine Substitution existiert:

σ = [a/2, b/3, c/4*x]

Wenn wir sie auf die rechte Seite der Regel anwenden, erhalten wir die nach dem Distributivgesetz transformierte Formel

2 ∗ 3 + 2 ∗ 4 ∗ x

Wir können die „Anwendbarkeit" der Formel aus der Rechenregel auf eine andere Formel durch eine Relation zwischen Termen ausdrücken:

Definition 3.12: Relation umfasst

*Ein **Term s umfasst einen Term t**, wenn es eine Substitution* σ *gibt mit* $s\ \sigma = t$. ■

In unserem Beispiel gilt dann

a ∗ (b + c) umfasst 2 ∗ (3 + 4 ∗ x)

Die Relation *umfasst* hat folgende Eigenschaften: Sie ist

- transitiv, denn aus *r* umfasst *s* und *s* umfasst *t* folgt, dass es σ_1 und σ_2 gibt mit $r\ \sigma_1 = s$, $s\ \sigma_2 = t$. Also gilt $(r\ \sigma_1)\ \sigma_2 = t$ und $r\ (\sigma_1\ \sigma_2) = t$;

- reflexiv, denn es gilt immer t [x / x] = t;
- nicht antisymmetrisch, denn es gibt Terme mit *s* umfasst *t* und *t* umfasst *s*, z. B. 2 ∗ x [x / y] = 2 ∗ y und 2 ∗ y [y / x] = 2 ∗ x.

Deshalb erfüllt *umfasst* nicht die Bedingungen für eine Halbordnung, sondern nur die für eine Quasiordnung (siehe Abschnitt 2.6).

3.2.2 Unifikation

Im Beispiel der Anwendung von Rechenregeln haben wir den Term der Regel als den allgemeineren angesehen und ihn durch eine Substitution in einen konkreteren transformiert. Nun betrachten wir stattdessen die symmetrische Fragestellung: Gibt es zu zwei gegebenen Termen eine Substitution, die die Terme in dasselbe Ergebnis transformiert?

Definition 3.13: Unifikation

*Die beiden **Terme s und t sind unifizierbar,** wenn es eine Substitution σ gibt mit s σ = t σ. σ ist dann ein **Unifikator** von s und t.* ■

Die folgende Tabelle zeigt Beispiele für Unifikatoren, sofern sie existieren.

s	x + y	x + 3	x + x	2 ∗ x + 3
t	2 + 3	2 + y	2 + 3	z + x
σ	[x / 2, y / 3]	[x / 2, y / 3]	./.	[x / 3, z / 2 ∗ 3]
s σ = t σ	2 + 3	2 + 3	./.	2 ∗ 3 + 3

Man beachte, dass für das letzte Beispiel die Substitution [x / 3, z / 2 ∗ x] nicht die Bedingung für einen Unifikator erfüllt.

Betrachten wir folgendes Beispiel etwas genauer:

s = (x + y) t = (2 + z)

Folgende Substitutionen sind Unifikatoren für *s* und *t*:

σ_1 = [x / 2, y / z] σ_2 = [x / 2, z / y]

σ_3 = [x / 2, y/1, z/1] σ_4 = [x /2, y/2, z/2]

Es gibt noch beliebig viele weitere Unifikatoren für *s* und *t*, die dieselben Terme für *y* und *z* substituieren. Intuitiv sehen wir σ_1 und σ_2 als allgemeiner als die übrigen Unifikatoren an, denn sie legen den Ergebnisterm weniger speziell fest:

s σ_1 = 2 + z

s σ_2 = 2 + y

s σ_3 = 2 + 1

s σ_1 und s σ_2 umfassen jeweils s σ_3, s σ_4, usw. Aus jedem der Unifikatoren σ_1 und σ_2 kann man auch alle übrigen erzeugen, z.B.

σ_1 [z / y] = [x / 2, y / y, z / y] = [x / 2, z / y] = σ_2

σ_1 [z / 1] = [x / 2, y / 1, z / 1] = σ_3

Daher sind wir bei der Unifikation nur an den allgemeinsten Unifikatoren wie hier σ_1 und σ_2, interessiert.

Definition 3.14: Allgemeinster Unifikator

*Ein Unifikator σ_a heißt **allgemeinster Unifikator der Terme s und t**, wenn es zu jedem Unifikator σ_i von s und t eine Substitution τ_i gibt, die σ_i aus σ_a erzeugt: $\sigma_a \tau_i = \sigma_i$.* ■

In obigem Beispiel sind nur σ_1 und σ_2 allgemeinste Unifikatoren. Alle übrigen Unifikatoren haben die Form

σ_r = [x / 2, y / r, z / r]

mit einem beliebigen Term *r*. Sie können aus σ_1 und σ_2 erzeugt werden:

σ_1 [z / r] = σ_2 [y / r] = σ_r

Aber aus keinem σ_r kann σ_1 oder σ_2 erzeugt werden.

Wir stellen nun ein Verfahren vor, das feststellt, ob zwei Terme unifizierbar sind, und in diesem Fall einen allgemeinsten Unifikator liefert. Zunächst definieren wir eine Funktion *Abweichungspaar*. Sie wird auf zwei verschiedene Terme *s* und *t* angewandt und liefert dann das Paar (*u, v*) von korrespondierenden Untertermen von *s* und *t*, sodass *u* und *v* verschieden sind, möglichst weit links in *s* und *t* vorkommen und möglichst klein sind. Sie ist präzise definiert durch

Abweichungspaar (*s, t*) = (*u, v*) für $s \neq t$: (*u, v*) = (*s, t*) falls

- $s = f(...)$ und $t = g(...)$ und $f \neq g$
- $s = f(...)$ und $t = x$
- $s = x$ und $t = f(...)$
- $s = x$ und $t = y$ und $x \neq y$

sonst ist $s = f(t_1, ..., t_i, ...)$ und $t = f(t_1, ..., t_i', ...)$ und *i* ist der kleinste Index, sodass $t_i \neq t_i'$, dann ist (*u, v*) = Abweichungspaar (t_i, t_i').

Damit lautet der Algorithmus zur Unifikation der Terme *s* und *t*:

1. Setze σ = [ ].
2. Solange $s\,\sigma \neq t\,\sigma$ wiederhole Schritte 3 und 4:
3. Setze (*u, v*) = Abweichungspaar (s σ, t σ).
4. Falls
   – *u* ist eine Variable *x,* die in *v* nicht vorkommt, dann ersetze σ durch σ [*x / v*];

- v ist eine Variable x, die in u nicht vorkommt, dann ersetze σ durch $\sigma\ [x / u]$
- sonst sind diese Terme nicht unifizierbar; Abbruch des Algorithmus.

5. Bei Erfolg gilt $s\ \sigma = t\ \sigma$ und σ ist ein allgemeinster Unifikator.

Wir wenden den Algorithmus zum Beispiel auf die Terme $x + y$ und $1 + 2 * x$ an. Man findet die Abweichungspaare am einfachsten, wenn man die Terme in Funktionsform untereinander schreibt und die korrespondierenden Unterterme ausrichtet:

s [ ] = + (x, y)

t [ ] = + (1, * (2, x))

Das Abweichungspaar (x, 1) liefert [x /1].

s [ ] [x / 1] = + (1, y)

t [ ] [x / 1] = + (1, * (2, 1))

Das Abweichungspaar (y, * (2, 1)) liefert [y / * (2, 1)].

s [ ] [x / 1] [y / * (2, 1)] = + (1, * (2, 1))

t [ ] [x / 1] [y / * (2, 1)] = + (1, * (2, 1))

Ein allgemeinster Unifikator ist gefunden:

[x / 1] [y / * (2, 1)] = [x / 1, y / 2 * 1]

3.3 Algebren

Eine Algebra ist eine formale Struktur. Sie besteht aus einer Menge grundlegender Elemente (*Trägermenge*), aus Operationen über dieser Menge und aus Gesetzen, die die Operationen zueinander in Bezug setzen. Damit kann man formale Kalküle definieren, wie z. B. die Algebra der Mengenoperationen, die Boolesche Algebra oder die Arithmetik ganzer Zahlen. In der Modellierung der Informatik spezifiziert man mit Algebren auch veränderliche Datenstrukturen wie Keller und Listen oder Abläufe in dynamischen Systemen, wie die Bedienung unseres Getränkeautomaten.

Wir unterscheiden zwei Ebenen: abstrakte Algebren und konkrete Algebren. Eine abstrakte Algebra spezifiziert Eigenschaften abstrakter Operationen, die durch eine Signatur beschrieben werden. Es bleibt absichtlich offen, wie die Operatoren durch Funktionen über konkreten Wertemengen realisiert werden. Die Trägermenge einer abstrakten Algebra sind die Grundterme der Signatur. Die Gesetze der Algebra identifizieren Terme mit gleicher Bedeutung. So können Eigenschaften der Operationen oder das Verhalten des Systems abstrakt vorgegeben werden. Zu einer abstrakten Algebra kann man konkrete Algebren angeben. Ihre konkreten Funktionen passen zu den abstrakten Operationen der Signatur. Wenn man zeigen kann, dass sie alle Gesetze der abstrakten Algebra erfüllen, beschreibt die konkrete Algebra eine Realisierung oder Implementierung der abstrakten Spezifikation.

3.3.1 Abstrakte Algebra

Definition 3.15: Abstrakte Algebra

*Eine **abstrakte Algebra*** A = (τ, ∑, Q) *ist definiert durch die Signatur ∑, die Menge der korrekten Terme τ zu ∑ und eine Menge von Axiomen (Gesetzen)* Q. *Ein Axiom hat die Form $t_1 \rightarrow t_2$, wobei t_1 und t_2 korrekte Terme gleicher Sorte sind und Variable enthalten können. Kommen in der Signatur mehrere Sorten vor, so sagen wir: die Algebra ist **heterogen**. Eine der Sorten, die auch als Ergebnissorte von Operatoren vorkommt, kann als **die von der Algebra definierte Sorte** ausgezeichnet werden.* ■

Da die Trägermenge die zu ∑ gebildeten Terme sind, nennt man solche Algebren auch *Term-Algebren*.

Signatur ∑ = (S, F), S = {BOOL} Operation F:

true:		→ BOOL
false:		→ BOOL
∧:	BOOL × BOOL	→ BOOL
∨:	BOOL × BOOL	→ BOOL
¬:	BOOL	→ BOOL

Axiome *Q*: für alle *x, y* der Sorte BOOL gilt

Q_1:	¬ true	→ false
Q_2:	¬ false	→ true
Q_3:	true ∧ x	→ x
Q_4:	false ∧ x	→ false
Q_5:	x ∧ y	→ ¬ (¬ x ∨ ¬ y)

Abbildung 3.2: Eine abstrakte Boolesche Algebra

Ein einfaches und grundlegendes Beispiel für eine abstrakte Algebra ist eine Boolesche Algebra. In Abb. 3.2 werden ihre Signatur und ihre Axiome definiert. Die Menge der korrekten Terme ergibt sich aus der Signatur. Man beachte, dass die Operationen definitionsgemäß abstrakt sind, obwohl die Symbole die Bedeutung logischer Verknüpfungen suggerieren. Eigenschaften der Operationen können wir nur dadurch ermitteln, dass wir die Axiome anwenden, um Terme in solche umzuformen, die im Sinne der Algebra gleichbedeutend (siehe Definition 3.19) sind. Durch Anwendung von Q_1 und Q_2 kann man z.B. folgende Terme umformen:

¬ ¬ true → ¬ false → true

¬¬ false → ¬ true → false

Wir haben in dieser Booleschen Algebra nur eine Auswahl von Axiomen als Beispiele angegeben. Um damit Terme systematisch umzuformen, müssten weitere Axiome angegeben werden, wie Kommutativ- und Assoziativgesetze.

Grundsätzlich definieren die Axiome einer Algebra, wie Terme in andere Terme umgeformt werden können.

Definition 3.16: Mit Axiomen umformen

Mit Axiomen umformen *heißt: unter Anwenden eines Axioms $t_1 \to t_2$ einen Term s_1 in einen Term s_2 umformen. Wir schreiben $s_1 \to s_2$, wenn gilt:*

1. *s_1 und s_2 stimmen überein bis auf einen Unterterm an entsprechender Position, d. h., es gibt einen Term s, in dem die Variable v genau einmal vorkommt, und $s_1 = s\,[v / r_1]$, $s_2 = s\,[v / r_2]$.*
2. *Es gibt eine Substitution σ mit $t_1\,\sigma = r_1$ und $t_2\,\sigma = r_2$.*

s ist in t umformbar, wenn es eine Folge von Termen $s = s_0, s_1, ..., s_n = t$ mit $s_{i-1} \to s_i$ gibt; wir schreiben dann $s \to t$.

$\to$ ist transitiv. Die Axiome sollten so gewählt werden, dass $\to$ auch irreflexiv ist; dann ist $\to$ eine strenge Halbordnung. ■

Definition 3.16 kann man auch grafisch veranschaulichen:

$$\begin{array}{lcl} \text{Term } s_1 = \ldots\ r_1 \ldots\ \ldots.\ r_2 \ldots = s_2 \\ & \| \qquad \| \\ & t_1\,\sigma \quad t_2\,\sigma \\ \text{Axiom} \qquad t_1 & \to & t_2 \end{array}$$

Als Beispiel formen wir mit dem Axiom Q_3 der Algebra Bool den Term s_1 in s_2 um:

$$\begin{array}{lcccl} s_1 = & y \vee (\text{true} \wedge (\neg z)) & & y \vee (\neg z) = s_2 \\ & \overline{\qquad\qquad} & & \overline{\qquad} \\ & r_1 & & r_2 \\ & \| & & \| \\ & t_1\,[x / \neg z] & & t_2\,[x / \neg z] \\ & \| & & \| \\ Q_3: & \text{true} \wedge x & \to & x \end{array}$$

3.3.2 Konkrete Algebra

Wir zeigen nun, wie man die Vorgaben einer abstrakten Algebra realisiert, indem man eine Algebra mit konkreten Funktionen über bestimmten Wertemengen angibt.

Definition 3.17: Konkrete Algebra

Einer abstrakten Algebra $A_a = (\tau, (S, F), Q)$ lässt sich eine konkrete Algebra $A_k = (W_k, F_k, Q)$ zuordnen, *wenn gilt:*

1. *Jeder Sorte $s \in S$ ist ein Wertebereich $w \in W_k$ zugeordnet.*
2. *Jeder Operation $g \in F$ ist eine Funktion $f \in F_k$ zugeordnet. Der Definitions- und der Bildbereich von **f** ergeben sich durch die Abbildung der Sorten von g.*
3. *Den Axiomen in Q müssen Gleichungen zwischen den Funktionstermen in den Wertebereichen entsprechen.*

A_k nennt man auch ein Modell der abstrakten Algebra A. ■

In der *konkreten Algebra* können außer den Q entsprechenden noch weitere Gleichungen gelten. In diesem Sinne kann A_k spezieller als A_a sein. Zu der *abstrakten Booleschen Algebra* aus Abb. 3.2 kann man natürlich als Modell eine konkrete Algebra mit den entsprechenden logischen Formen angeben. Um das Prinzip zu verdeutlichen, zeigen wir hier als Beispiel die konkrete Algebra *FSet*:

Der einzigen Sorte BOOL wird die Wertemenge $\{\emptyset, \{1\}\}$ zugeordnet. Die Funktionen ordnen wir den abstrakten Operationen zu:

konstante Funktion, die $\{1\}$ liefert	zu *true*
konstante Funktion, die $\emptyset$ liefert	zu *false*
Mengendurchschnitt $\cap$	zu $\wedge$
Mengenvereinigung $\cup$	zu $\vee$
Mengenkomplement bezüglich $\{1\}$	zu $\neg$

Wir zeigen nur beispielhaft an Q_4, dass eine entsprechende Gleichung zwischen Funktionstermen gilt:

Für alle $x \in \{\emptyset, \{1\}\}$ gilt $\emptyset \cap x = \emptyset$. Mit der Zuordnung von $\emptyset$ zu *false* entspricht das Q_4: $false \wedge x \rightarrow false$.

3.4 Algebraische Spezifikation von Datenstrukturen

Ein typisches Einsatzgebiet für algebraische Spezifikationen sind dynamisch veränderliche Datenstrukturen: Mit einer abstrakten Algebra kann man die Wechselwirkungen zwischen den Operationen präzise beschreiben, ohne etwas über deren Realisierung sagen zu müssen. Anhand der Axiome lässt sich dann die Korrektheit von Implementierungen nachweisen. Auch steht mit der Term-Notation eine gut geeignete Schreibweise für Abfolgen von Operatoren auf der Datenstruktur zur Verfügung.

Wir wollen solch eine algebraische Spezifikation am Beispiel der Datenstruktur *Keller* demonstrieren. Ein *Keller* (engl. stack) ist eine Datenstruktur, in die man Elemente einfügen und aus der man Elemente entfernen kann. Dabei wird das so genannte *Keller-Prinzip* befolgt: Was zuletzt eingefügt wurde, wird als Erstes wieder entfernt (engl. last-in-first-out, LIFO).

Anwendungen des Kellerprinzips findet man auch im Alltag, z. B. die Tellerstapel an einer Essensausgabe; frisch gespülte Teller werden oben auf den Stapel gelegt; wer einen Teller benötigt, nimmt ihn auch oben weg. In der Informatik ist das *Keller-Prinzip* Grundlage für die Lösung sehr vieler Aufgaben: z. B. der *Laufzeitkeller* für die Implementierung von Programmiersprachen; dort werden die Daten für Funktionsaufrufe untergebracht. Bei Beginn des Aufrufes werden sie gekellert, am Ende wieder entkellert; der zuletzt begonnene Aufruf wird als Erster beendet.

Bevor wir die abstrakte Algebra spezifizieren, verschaffen wir uns einen informellen Überblick über die Operationen auf einem Keller:

createStack: liefert einen leeren Keller

push: fügt ein Element in den Keller ein

pop: entfernt das zuletzt eingefügte Element

top: liefert das zuletzt eingefügte und noch nicht wieder entfernte Element

empty: gibt an, ob der Keller leer ist

Abb. 3.3 zeigt die Signatur und die Axiome der abstrakten Algebra Keller. Terme der Sorte *Keller* modellieren Inhalte bzw. Werte der spezifizierten Datenstruktur. *Element* ist eine *Hilfssorte*, die die Daten im Keller modelliert. Die zweite Hilfssorte BOOL mit den Konstanten true und false tritt als Ergebnis der als Prädikat intendierten Operation empty auf. Um uns mit der Keller-Algebra vertraut zu machen, formulieren wir einige Terme und formen sie mit den Axiomen um. Der Term

push (push (push (createStack, 1), 2), 3)

soll zum Beispiel einen Keller modellieren, in den nacheinander die Zahlen 1, 2, 3 eingefügt wurden. Entfernen wir ein Element, bevor die dritte Zahl eingefügt wird, so lautet die Operationsfolge

push (pop (push (push (createStack, 1) , 2)), 3)

Mit dem Axiom K_3 können wir sie in folgenden gleichbedeutenden Term umformen (siehe Definition 3.19):

push (push (createStack, 1), 3)

Daran erkennen wir auch, dass das Axiom *K3* das Keller-Prinzip formalisiert: Die Operation pop macht die Wirkung der letzten Anwendung einer push-Operation rückgängig.

Weiter sehen wir, dass top die einzige Operation ist, die ein Keller-Element als Ergebnis liefert – und zwar immer das zuletzt eingefügte und noch nicht wieder entfernte. Auf andere Elemente kann man mit diesen Operationen nicht zugreifen.

Definition 3.16 empfiehlt, dass die Axiome so definiert werden, dass die Relation $\rightarrow$ über Terme irreflexiv ist. Die Algebra Keller folgt der Empfehlung, denn die rechte Seite jedes Axioms enthält weniger Operationen als die entsprechende linke Seite. Also ist $t \rightarrow t$ für keinen Term t umformbar.

Am Beispiel der Keller-Algebra wollen wir weitere Eigenschaften solcher Spezifikationen herausarbeiten. Im Unterschied zur Booleschen Algebra enthält die Keller-Algebra neben der hier definierten Sorte Keller weitere Sorten, d. h. sie ist heterogen. Die Terme der Sorte Keller sollen Werte der Datenstruktur Keller modellieren. Sie wird deshalb als die *von der Algebra definierte Sorte* ausgezeichnet. Dies führt zu einer Klassifikation der Operationen der Algebra:

Signatur $\sum = (S, F)$

Sorten s = {Keller, Element, BOOL}

Operationen F:

CreateStack:		$\rightarrow$ Keller
push:	Keller $\times$ Element	$\rightarrow$ Keller
pop:	Keller	$\rightarrow$ Keller
top:	Keller	$\rightarrow$ Element
emtpy:	Keller	$\rightarrow$ BOOL

Axiome Q: für beliebige Terme t der Sorte Element und k der Sorte Keller gilt:

K1: empty (createStack) $\rightarrow$ true

K2: empty (push(k, t)) $\rightarrow$ false

K3: pop (push(k, t)) $\rightarrow$ k

K4: top (push(k, t)) $\rightarrow$ t

Abbildung 3.3: Abstrakte Algebra Keller

Definition 3.18: Klassifikation von Operationen

Die Operationen einer abstrakten Algebra teilt man in disjunkte Teilmengen ein: ***Konstruktoren****,* ***Hilfskonstruktoren*** *und* ***Projektionen****. Konstruktoren und Hilfskonstruktoren haben die von der Algebra definierte Sorte als Ergebnissorte; Projektionen haben eine andere Sorte als Ergebnissorte.* ■

Konstruktoren und Hilfskonstruktoren werden nach den Rollen unterschieden, die sie in der Normalform zu der Algebra spielen:

Definition 3.19: Terme in Normalform

Ein Term einer Algebra, der keine Variablen der definierten Sorte enthält, in dem nur Konstruktoren vorkommen und auf den kein Axiom der Algebra anwendbar ist, ist in ***Normalform****. Ein Term in Normalform ist ein minimales Element bezüglich der strengen Halbordnung* $\rightarrow$.

Können zwei Terme s und t in dieselbe Normalform umgeformt werden, nennen wir sie ***gleichbedeutend****,* $s \equiv t$. ■

Für die Keller-Algebra ist eine Klassifikation der Operationen leicht zu finden: push und createStack sind Konstruktoren, pop ist ein Hilfskonstruktor und top und empty sind Projektionen. Jeder Term in Normalform hat folgende Struktur:

(push (push ... (push (createStack, n_1), ...), n_m) mit $m \geq 0$

Mit dem Axiom K_3 kann man Unterterme der Form pop (push (r, t)) in *r* umformen und so die Anwendung des Hilfskonstruktors pop entfernen. Terme, die so in dieselbe Normalform umgeformt werden können, modellieren denselben Kellerinhalt und abstrahieren von unterschiedlichen Operationsfolgen, die ihn erzeugt haben.

Durch Induktion über die Vorkommen von pop in einem Term kann man beweisen, dass alle Vorkommen durch Anwenden von K_3 eliminiert werden können, außer im Term pop (createStack). Deshalb sehen wir diesen Term als undefiniert an. Er modelliert den fehlerhaften Versuch, aus einem leeren Keller ein Element zu entfernen.

Definition 3.20: Undefinierte Terme

Jeden Term der definierten Sorte einer Algebra, der nicht durch Anwenden von Axiomen in Normalform umgeformt werden kann, sehen wir als ***undefiniert*** *an. Er modelliert eine Fehlersituation. Ein Term t = op (nf) ist ebenfalls undefiniert, wenn op eine Projektion ist, nf ein Term in Normalform und auf t kein Axiom anwendbar ist.* ■

Gemäß dem zweiten Teil der Definition 3.20 ist auch top (createStack) ein undefinierter Term der Keller-Algebra.

Für die abstrakte Algebra Bool aus Abb. 3.2 gibt es drei Möglichkeiten zur Klassifikation der Operationen: Man kann alle Operationen zu Konstruktoren erklären oder $\wedge$ oder $\vee$ jeweils zum Hilfskonstruktor machen. Nur die beiden letzten Fälle sind wegen ihrer nichttrivialen Normalformen interessant.

Bei dem Entwurf einer abstrakten Algebra ist es nützlich, die Normalform zu kennen: zur Definition von Hilfskonstruktoren und Projektionen braucht man nur solche Axiome anzugeben, sodass die Operation für jeden Term der Normalform definiert ist oder undefiniert bleibt. Die Bedeutung für alle übrigen Terme der definierten Sorte kann man durch Umformen in die Normalform herleiten. Betrachten wir dazu die Axiome der Keller-Algebra: Aus K_1 und K_2 lässt sich die Anwendung von empty für jeden Term in Normalform herleiten. Das gilt auch für K_3 und pop sowie K_4 und top. Hier ist jedoch das zu K_1 ent-

sprechende Axiom für createStack weggelassen, weil pop (createStack) und top (createStack) als undefiniert angesehen werden sollen.

Dieses Prinzip wenden wir auch an, wenn wir eine Algebra um neue Operationen erweitern. Als Beispiel erweitern wir die Keller-Algebra um eine Operation size. Sie soll die Anzahl der Elemente im Keller liefern. Dazu fügen wir die Operationsbeschreibung in die Signatur ein:

size: Keller → NAT

size ist also eine *Projektion*. Die neue Ergebnissorte müssen wir den Sorten hinzufügen:

s = {Keller, Element, BOOL, NAT}

Und schließlich benötigen wir Axiome, die size für alle Terme in Normalform definieren:

K_5: size (createStack) → null

K_6: size (push (k, t)) → succ (size (k))

Wir setzen dabei voraus, dass folgende Algebra bekannt ist:

Sorten: S = {NAT}

Operationen: null: → NAT, succ: NAT → NAT
(succ modelliert den Nachfolger von n, also n+1.)

size, angewandt auf einen Normalform-Term,

size (push (push (createStack, a), b))

kann man dann mit K_5 und K_6 umformen in

succ (succ (null)).

Mit einer passend definierten konkreten Algebra für natürliche Zahlen kann man dem Term dann den Wert 2 zuordnen.

Wir geben nun eine konkrete Algebra zur Keller-Algebra an und zeigen, dass sie tatsächlich die abstrakte Spezifikation erfüllt. Dabei wählen wir eine Realisierung durch Funktionen über Folgen auf natürlichen Zahlen. Die Wertebereiche ordnen wir den Sorten wie folgt zu:

konkret	**abstrakt**
FSet	BOOL
$\mathbb{N}$	Element
N-Folge = $\mathbb{N}^*$	Keller

Die Funktionen werden folgendermaßen zugeordnet:

konkret			**abstrakt**
newFolge:		→ N-Folge	create Stack
append:	N-Folge × $\mathbb{N}$	→ N-Folge	push
remove:	N-Folge	→ N-Folge	pop
last:	N-Folge	→ $\mathbb{N}$	top
noElem:	N-Folge	→ FSet	empty

Die Wirkung der Funktionen wird wie folgt definiert:

newFolge ( )		liefert die leere Folge ()
append ($(a_1, ..., a_n)$, x)		liefert ($a_1, ..., a_n$, x)
remove ($(a_1, ..., a_{n-1}, a_n)$)	für $u \geq 1$	liefert ($a_1, ..., a_{n-1}$)
last ($(a_1, ..., a_n)$)	für $u \geq 1$	liefert a_n
noElem (f)		liefert {1}, falls f = ( ), sonst ∅

Nun zeigen wir, dass die Axiome der Keller-Algebra auch für unsere Realisierung gelten:

Axiom K_1: empty (createStack) → true

Setzen wir statt der abstrakten Operationen die zugeordneten Funktionen ein, so erhalten wir als Gleichung über Funktionsterm:

noElem (newFolge()) = {1}.

Mit der Definition der Funktionen noElem und newFolge gilt also die K_1 entsprechende Gleichung. Ebenso zeigt man die Gültigkeit der K_3 entsprechenden Gleichung:

K3: pop (push(k, t)) → k

remove (append (k, x))	= k
remove (append ($(a_1, ..., a_n)$, x))	= k
remove ($(a_1, ..., a_n$, x))	= k
($a_1, ..., a_n$)	= k
k	= k

Für die übrigen Axiome weist man Entsprechendes nach.

Zum Schluss der Betrachtung der Keller-Algebra wollen wir sie anwenden, um die konkrete Verwendung eines Kellers in einem Algorithmus nachzuweisen. Unser Algorithmus soll Terme aus der Infixform in die Postfixform umwandeln:

Gegeben: ein Term *t* in Infixform, nur mit 2-stelligen Operationen unterschiedlicher Präzedenz, zunächst ohne Klammern.

Gesucht: der Term *t* in Postfixform. (Die Aufgabenstellung lässt sich leicht erweitern durch Hinzunehmen von Klammern, *1*-stelligen und *n*-stelligen Operatoren.)

Als Beispiel nehmen wir an, dass + und – geringste, ∗ und / mittlere und ↑ höchste Präzedenz haben und alle linksassoziativ sind.

Der Algorithmus soll z. B. den Term

a - b * c in a b c * -

umwandeln. Dazu liest er die Symbole des Eingabeterms nacheinander und gibt sie in der benötigten Reihenfolge aus. Da die Reihenfolge der Variablen und Konstanten im Ergebnis unverändert ist und sie vor ihrem Operator stehen, werden sie sofort ausgegeben, wenn sie gelesen werden.

Wenn in der Infixform Operatoren mit steigender Präzedenz aufeinander folgen, z. B. auf ein - ein ∗ folgt, stehen sie in der Postfixform in umgekehrter Reihenfolge, also ∗ vor -.

Das Umkehren von Folgen ist eine typische Aufgabe für Keller. Bevor wir den Algorithmus aufschreiben, legen wir fest, wie wir den Keller verwenden wollen, und formulieren es durch eine Keller-Invariante:

Keller-Invariante KI:

Der Keller enthält Operatoren mit echt steigender Präzedenz, d. h. sei

push (...(push (createStack, opr_1), ...), opr),

dann gilt Präzedenz (opr_i) < Präzedenz (opr_{i+1}) für alle i = 1, ..., n - 1

An den Stellen, wo in Abb. 3.4 $\{KI\}_i$ angegeben ist, gilt die Keller-Invariante. Nur um darauf Bezug zu nehmen, haben wir sie indiziert: Für den leeren Keller gilt natürlich $\{KI\}_1$. Da in der ersten Schleife beide *falls*-Zweige wieder auf {KI} führen, gilt {KI} vor, am Anfang, am Ende und nach der ersten Schleife (eine Schleifen-Invariante). Der erste *falls*-Zweig verändert den Keller nicht, deshalb gilt $\{KI\}_3$. Im zweiten Zweig gilt nach der inneren Schleife vor der *push*-Operation: Entweder ist der Keller leer, oder die Präzedenz des anstehenden Symbols ist größer als die des obersten Kellerelementes. Deshalb kann es nun gekellert werden, ohne *KI* zu verletzen. Die letzte Schleife leert den Keller, ohne *KI* zu verletzen. Damit ist bewiesen, dass der Algorithmus elementare Operanden vor ihren Operatoren angibt und Folgen von Operatoren steigender Präzedenz umkehrt und deshalb die Postfixform erzeugt.

.

```
keller ∈ Keller
symbol ∈ Operator ∪ Variable
keller = createStack();                    {KI}1
solange Eingabe nicht leer, wiederhole {KI}2
        lies symbol;
        falls symbol ∈ Variable ∪ Konstante
                gib symbol aus             {KI}3
        falls symbol ∈ Operator
                solange not empty(keller) ∧
                                Präzedenz (top(keller)) ≥
                                Präzedenz (symbol)
                wiederhole
                        gib top(keller) aus;
                        keller = pop(keller);
                        keller = push(keller, symbol). {KI}4
                solange not empty(keller), wiederhole
                gib top(keller) aus;
                keller = pop(keller);      {KI}5
```

Abbildung 3.4: Algorithmus zum Umformen von Infix-in Postfixform

Abb. 3.5 demonstriert den Ablauf des Algorithmus anschaulich an einem Gleissystem: Der Infix-Term fährt von rechts nach links auf die Weiche zu. Variable fahren sofort geradeaus durch. Ein Operator fährt in den Keller ein, sobald dort kein Operator mit gleicher oder höherer Präzedenz steht. Solche werden über das Ausfahrtgleis in den Postfixterm entkellert.

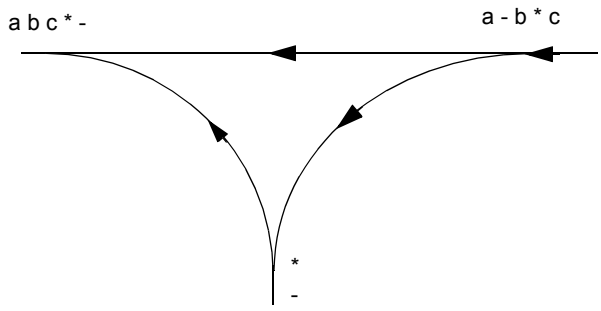


Abbildung 3.5: Gleissystem zum Rangieren von Termen

3.5 Algebraische Spezifikation für den Getränkeautomaten

Wir wollen nun zeigen, wie man einen Teilaspekt unserer Beispielaufgabe „Getränkeautomat“ mit einer abstrakten Algebra spezifiziert.

Unser Automat hat zwei Knöpfe, mit denen man auswählen kann, ob dem Getränk Zucker oder Milch oder beides zugegeben wird; wir nennen sie hier *sweet* und *white*. Eine bereits getroffene Wahl soll man zurücknehmen können, indem man den Knopf noch einmal drückt. Es soll gleichgültig sein, in welcher Reihenfolge die Knöpfe gedrückt werden. Solche Eigenschaften kann man gut algebraisch spezifizieren.

Wir definieren eine Signatur mit zwei Sorten s = {Add, Choice} und folgende Operationen:

sweet:		$\rightarrow$ Add
white:		$\rightarrow$ Add
noChoice:		$\rightarrow$ Choice
press:	Add $\times$ Choice	$\rightarrow$ Choice

press modelliert die zentrale Operation. Der schon getätigten Auswahl wird ein weiterer Tastendruck hinzugefügt. Die Sorte Choice ist die hier definierte Sorte, Add wird als Hilfssorte benutzt, um die beiden Konstanten sweet und white zusammenzufassen. noChoice modelliert eine leere, erweiterbare Folge von Tastendrücken.

Ein Term beschreibt dann eine Folge von Tastendrücken, z. B.

press (sweet, press (white, press (sweet, noChoice)))

Die beiden Anforderungen an die Bedeutung der Terme formulieren wir durch Axiome, für alle Terme a, b ∈ Add. und c ∈ Choice gilt:

Q_1: press (a, press (a, c)) → c

Q_2: press (sweet, press (white, c)) → press (white, press (sweet, c))

Q_2 identifiziert alle Terme, die sich nur in der Reihenfolge unterscheiden, in der die Tasten gedrückt wurden. Q_2 legt eine Reihenfolge für die Normalform fest: erst sweet, dann white. Q_1 drückt aus, dass sich aufeinander folgende Betätigungen derselben Taste paarweise aufheben; zusammen mit Q_2 gilt das auch für weiter auseinander liegende Betätigungen von Tasten. Den oben angegebenen Term kann man mit den Axiomen vereinfachen zu press (white, noChoice). Zu dieser Algebra gibt es nur vier verschiedene Terme, die in der Normalform sind; alle anderen können in einen davon umgeformt werden:

noChoice
press (white, noChoice)
press (sweet, noChoice)
press (white, press (sweet, noChoice))

Diese vier Wahlmöglichkeiten hatten wir ja auch intendiert.

Die Spezifikation lässt sich sehr einfach erweitern: Wenn eine zukünftige Luxus-Variante des Automaten auch die Beigabe von Rum und eine Auswahltaste dafür anbietet, brauchen wir die Signatur nur um eine Operation rum: → Add zu erweitern. Q_1 bleibt unverändert, da Operationen mit der Ergebnissorte Add nicht explizit vorkommen. Zu Q_2 müssen wir Axiome ergänzen, die rum hinter white und sweet in die Normalform einordnet:

Q_3: press (sweet, press (rum, c)) → press (rum, press (sweet, c))

Q_4: press (white, press (rum, c)) → press (rum, press (white, c))

Übungen

3.1 Notation von Termen

Gegeben sei der in Infixnotation dargestellte Term a + (x ∗ (x + 5) - (y + z)). Geben Sie den Term in Postfix- und Präfixnotation an.

3.2 Korrekte Terme zur Signatur

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

Sorten S := {Add, Choice}

F := {	sweet:		→	Add,
	white:		→	Add,
	noAdd:		→	Add,
	noChoice:		→	Choice,
	press:	Add × Choice	→	Add
}				

Geben Sie jeweils zwei korrekte Terme an, welche die Schachtelungstiefen *0, 1, 2, 3* haben.

3.3 Signatur zu Operationen angeben

Auf Zeichenketten sind viele Operationen definiert. Einige davon sind:

- emptyString: eine 0-stellige Operation, die eine leere Zeichenkette erzeugt.
- addChar: liefert die Konkatenation aus einer Zeichenkette und einem Zeichen.
- length: gibt die Länge der Zeichenkette an.
- toLowerCase: wandelt alle Großbuchstaben einer Zeichenkette in Kleinbuchstaben um.
- isSubstring: überprüft, ob eine Zeichenkette in einer anderen Zeichenkette enthalten ist.
- concat: verkettet zwei Zeichenketten.

Sei die Signatur mit $\Sigma = (S, F)$ mit $S := \{$ ZEICHENKETTE, ZEICHEN, NAT, BOOL $\}$ gegeben. Geben Sie F an.

3.4 Substitution

Bestimmen Sie die Ergebnisse der folgenden Substitutionen:

a) f (x, y, g (a, x)) [y / h (a,x)]

b) f (x, y, g (a, x)) [y / g (a, x)]

c) f (x, y, g (a, x)) [x / g (a, x)]

d) f (x, y, g (a, x)) [y / b, x / c)]

e) f (x, y, g (a, x)) [y / g (a, x), x/y]

3.5 Unifikation

Peter und Klaus schauen bei einem Würfelspiel zu. Es werden fünf Würfel geworfen, die zur Unterscheidung mit den Indizes {1, 2, 3, 4, 5} gekennzeichnet sind. Einen Würfelwurf beschreiben wir durch den Term

wurf $(x_1, x_2, x_3, x_4, x_5)$,

wobei das *i-te* Argument die Augenzahl des Würfels *i* angibt.

a) Beschreiben Sie folgende Beobachtungen durch entsprechende Terme:

(1) Peter sagt: Beim letzten Wurf zeigten Würfel 1 und Würfel 3 die gleiche Augenzahl. Würfel 5 zeigte den Wert 6.

(2) Klaus sagt: Beim letzten Wurf zeigten Würfel 2 und Würfel 4 die gleiche Augenzahl. Würfel 3 zeigte den Wert 2.

Hinweis: Verwenden Sie **verschiedene** Variablen für die beiden Beobachtungen!

b) Überprüfen Sie mit Hilfe der Unifikation, ob die beiden Beobachtungen vereinbar sind bzw. ob die beiden Zuschauer falsch beobachtet haben. Geben Sie dazu gegebenenfalls einen Unifikator für die Terme nach (a1) und (a2) an.

3.6 Unifikationsverfahren nach Robinson

Es seien die beiden Terme in Funktionsnotation

f (g (y), h (y, g (y))) und, *f (z, h (g(x), g (g (x))))*

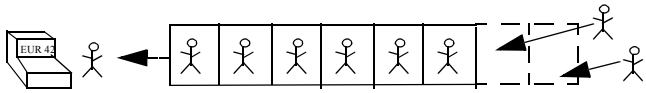
mit den Variablen *x*, *y* und *z* gegeben.

a) Unifizieren Sie die Terme mit Hilfe des Verfahrens von Robinson. Gehen Sie dabei wie folgt vor:
   - Notieren Sie in jedem Schritt den aktuellen Wert der Substitution σ und das Ergebnis der Anwendung dieser Substitution auf die beiden Terme.
   - Kennzeichnen Sie in jedem Schritt das Abweichungspaar durch Unterstreichen.
   - Fassen Sie die einzelnen Substitutionen zu einer gemeinsamen Substitution zusammen.

b) Geben Sie einen zweiten allgemeinsten Unifikator für die beiden Terme an. Begründen Sie, warum das Verfahren von Robinson zu verschiedenen Unifikatoren führen kann.

3.7 Abstrakte Algebra Schlange

Die Datenstruktur *Schlange* verwaltet Elemente in der Reihenfolge ihres Eintreffens. Operationen auf einer Schlange erlauben es, Elemente hinten anzufügen oder vorne die am längsten wartenden Elemente zu entnehmen. Schlangen arbeiten nach dem FIFO-Prinzip (First-In-First-Out). Sie sind z. B. an der Kasse im Supermarkt als Warteschlange zu finden.



Die folgende abstrakte Algebra beschreibt eine Schlange = (τ, Σ, Q) mit der Signatur Σ = (Σ, F):

S = {Queue, Element, BOOL}				(S_1)
F = {	createQueue:		→ Queue,	(F_1)
	enqueue:	Queue × Element	→ Queue,	(F_2)
	dequeue:	Queue	→ Queue	(F_3)
	front:	Queue	→ Element	(F_4)
	empty:	Queue	→ BOOL }	(F_5)

Die Sorte Queue stellt eine Schlange dar, die Sorte Element beschreibt Elemente der Schlange. Für die Sorte BOOL sind die Konstanten true und false definiert.

createQueue bezeichnet eine *0*-stellige Operation und ist damit eine Konstante. Sie steht für die leere Schlange. Die Operation enqueue fügt ein Element am Ende der Schlange an. Die Operation front liefert das erste Element vorne in der Schlange, dequeue entfernt die-

ses Element aus der Schlange. Ob die Schlange leer ist oder nicht, zeigt die Operation empty an.

Für die Menge der Axiome Q seien *x, y* Terme der Sorte Element und *q* ein Term der Sorte Queue:

```
Q ={
    Q1:   dequeue (enqueue (createQueue, x))      → createQueue,
    Q2:   dequeue (enqueue (enqueue (q, y), x))   →
                  enqueue (dequeue (enqueue (q, y)), x),
    Q3:   front (enqueue (createQueue, x))        → x,
    Q4:   front (enqueue (q, x))                  → front (q),
    Q5:   empty (createQueue)                     → true,
    Q6:   empty enqueue (createQueue, x))         → false
    }
```

Hinweis: Beziehen Sie sich in Ihrer Lösung auf die oben angegebenen Namen der Axiome Q_i. Im Folgenden seien *x, y, z* Variable und *a* eine Konstante der Sorte Element.

a) **Korrekte Terme:** Prüfen Sie, ob die folgenden Terme entsprechend der Signatur korrekt sind. Zeichnen Sie die Terme als Baum, und notieren Sie an jedem Blatt und jedem inneren Knoten die entsprechende Sorte. Kennzeichnen Sie die Stellen, an denen Terme nicht korrekt gebildet wurden.

(1) enqueue (enqueue (createQueue, x), y)
(2) front (dequeue (enqueue (createQueue, x)))
(3) front (createQueue, x)
(4) enqueue (a, createQueue)

b) **Axiome anwenden:** Formen Sie die folgenden Terme mit Hilfe der Axiome so um, dass Sie als Ergebnis eine einzelne Konstante oder Variable erhalten. Notieren Sie bei jeder Umformung das benutzte Axiom.

(1) dequeue (enqueue (createQueue, z))
(2) empty (enqueue (createQueue, y))
(3) front (dequeue (enqueue (enqueue (createQueue, z), x)))
(4) front (enqueue (enqueue (createQueue, x), z))
(5) empty (dequeue (enqueue (createQueue, x)))

3.8 Bedeutung der Axiome beschreiben

Beschreiben Sie in möglichst kurzen und präzisen Sätzen die Bedeutung der Axiome Q_1 bis Q_6.

3.9 Klassifikation von Operationen

a) Nehmen Sie an, dass createQueue und enqueue die einzigen Konstruktoren sind. Klassifizieren Sie die übrigen Operationen der Schlange als Hilfskonstruktoren oder Projektionen.

b) Wie für das Beispiel Keller (siehe Abschnitt 3.4) lässt sich auch für die Schlange eine Normalform angeben. Wie sind Terme der Sorte Queue in Normalform aufgebaut? (Hinweis: Geben Sie wie zu Definition 3.20 einen Term mit Lücken an.)

c) Formen Sie die folgenden Terme der Sorte Queue falls möglich in Normalform um. Notieren Sie bei jeder Umformung das benutzte Axiom:

(1) dequeue (enqueue (createQueue, x))
(2) enqueue (dequeue (enqueue (createQueue, x)), y)
(3) dequeue (enqueue (enqueue (createQueue, x), z))
(4) dequeue (enqueue (createQueue, front (enqueue (createQueue, y))))

3.10 Erweiterung der Schlange

Erweitern Sie die Algebra Schlange um eine zusätzliche Operation size, welche die aktuelle Anzahl der Elemente in der Schlange zurückgibt. Kopieren Sie die Definitionen der Schlange nicht, sondern vereinigen Sie die Mengen der oben angegebenen Algebra mit ihren Erweiterungen. Es wird vorausgesetzt, dass für die Sorte NAT wie zur Erweiterung der Keller-Algebra die Operationen null und succ sind.

3.11 Korrektheit einer Schlangen-Implementierung prüfen

Es sei folgende konkrete Algebra zur abstrakten Algebra Schlange gegeben:

Wertebereiche:

Bool	für die Sorte BOOL
$\mathbb{N}_0$	für die Sorte Element
N-Folge = $\mathbb{N}_0^*$	für die Sorte Queue

Funktionen:

new:		$\rightarrow$ N-Folge	für die Operation createQueue
append:	N-Folge $\times$ $\mathbb{N}_0$	$\rightarrow$ N-Folge	für die Operation enqueue
remove:	N-Folge	$\rightarrow$ N-Folge	für die Operation dequeue
first:	N-Folge	$\rightarrow$ $\mathbb{N}_0$	für die Operation front
noElem:	N-Folge	$\rightarrow$ Bool	für die Operation empty

Die Funktionen seien folgendermaßen definiert:

new:		$\rightarrow$ ( )
append:	$((a_1, ..., a_n), x)$	$\rightarrow (a_1, ..., a_n, x)$
remove:	$(a_1, a_2, ..., a_n)$	$\rightarrow (a_2, ..., a_n)$
first:	$(a_1, ..., a_n)$	$\rightarrow a_1$
noElem:	$(a_1, ..., a_n)$	$\rightarrow n = 0$

Beweisen Sie, dass zu den Axiomen Q_1 und Q_2 der abstrakten Algebra entsprechende Gleichungen in dieser Implementierung gelten.

3.12 Getränkeautomat

Die folgenden abstrakten Algebren modellieren auf verschiedene Weise die Abfolge von Bedienoperationen eines Getränkeautomaten:

(1) Sorten S := {Drink}

Operationen F:=

{ none: → Drink,
tea: Drink → Drink,
coffee: Drink → Drink,
chocolate: Drink → Drink}

Axiome Q: für alle Terme x der Sorte Drink gilt:

{ tea (tea (x)) → tea (x),
tea (coffee (x)) → tea (x),
tea (chocolate (x)) → tea (x),
coffee (tea (x)) → coffee (x),
coffee (coffee (x)) → coffee (x),
coffee (chocolate (x)) → coffee (x),
chocolate (tea (x)) → chocolate (x),
chocolate (coffee (x)) → chocolate (x),
chocolate (chocolate (x)) → chocolate(x)}

(2) Sorten S := {Drink, Choice}
Operationen F:=

{ tea: → Drink,
coffee: → Drink,
chocolate: → Drink,
noChoice: → Choice,
press: Drink × Choice → Choice}

Axiome *Q*: für alle Terme *a, b* der Sorte Drink und *c* der Sorte Choice gilt

{ press (a, press (b, c)) → press (a, c)}

a) Geben Sie für beide abstrakten Algebren je zwei verschiedene Terme der Schachtelungstiefe 3 an.

b) Beschreiben Sie die Bedeutung der Axiome in Worten.

c) Welche der beiden abstrakten Algebren lässt sich leichter erweitern? Warum?