Skriptum zur Vorlesung

Einführung in die Theoretische Informatik

Georg Moser

Wintersemester 2011

Dieses Dokument wurde mit Hilfe von KOMA-Script und LaTeX erstellt. Ich möchte mich an dieser Stelle herzlich bei meiner Kollegin Maria Schett und meinen Kollegen Simon Legner, Thomas Sternagel und Andreas Thalhammer für die Unterstützung bei der Erstellung dieses Skriptums bedanken.

Inhaltsverzeichnis

Vorwort

Absolventinnen und Absolventen des Moduls „Einführung in die Theoretische Informatik“ sollen die Inhalte der Vorlesung verstehen sowie diese wiedergeben und anwenden können. Sie sollen die Fähigkeit erworben haben, sich ähnliche Inhalte selbständig zu erarbeiten. Im Weiteren sollen sie ein Grundverständnis für die Methoden der theoretischen Informatik erlangt haben.

In der Vorlesung „Einführung in die Theoretische Informatik“ werden die folgenden Themen behandelt, die im Proseminar in weiterführenden Übungen vertieft werden.

- *Aussagenlogik*
- *Schaltkreise*
- *Grammatiken*
- *Chomsky-Hierarchie*
- *Formale Modelle*
- *Berechenbarkeit*
- *Gleichungslogik*
- *Programmverifikation*

In Kapitel 1 wird die *Aussagenlogik* besprochen. In Kapitel 2 werden *Schaltkreise* und die *Gleichungslogik* dargestellt. In Kapitel 3 werden *Grammatiken*, die *Chomsky-Hierarchie* und die ersten *formalen Modelle* eingeführt. *Formale Modelle* werden in Kapitel 4 vertieft, in dem auch die Theorie der *Berechenbarkeit* behandelt wird. Abschließend wird *Programmverifikation* in Kapitel 5 besprochen. Im Anhang A dieses Skriptums werden gängige Beweismethoden eingeführt, welche wir im Laufe der Vorlesung verwenden werden. Dieser Anhang dient zur Vertiefung des Verständnisses der verwendeten Beweisprinzipien und ist nicht Teil der Vorlesung.

Das vorliegende Skriptum ersetzt nicht den Besuch der Vorlesung, sondern ist vorlesungsbegleitend konzipiert. Etwa enthält das Skriptum keine, oder so gut wie keine, erläuternden Beispiele.

Um die Lesbarkeit zu erleichtern, wird in der direkten Anrede des Lesers, der Leserin prinzipiell die weibliche Form gewählt. In der Erstellung des Skriptums habe ich mich auf die folgende Literatur gestützt (in der Reihenfolge der Wichtigkeit für dieses Skriptum) [9, 8, 7, 1, 6, 10].

1

Einführung in die Logik

In diesem Kapitel wird die *Aussagenlogik* eingeführt. Zum leichteren Verständnis geschieht dies in zwei Stufen. In Abschnitt 1.1 werden gängige Schlussprinzipien, wie sie in allgemeingültigen Argumenten oft vorkommen, besprochen. In Abschnitt 1.2 wird die Sprache der Aussagenlogik formal eingeführt sowie die Semantik der Aussagenlogik definiert. Der Abschnitt 1.3 widmet sich der Bedeutung beziehungsweise der Manipulation von aussagenlogischen Formeln. In Abschnitt 1.4 betrachten wir ein korrektes und vollständiges Beweissystem für die Aussagenlogik. In Abschnitt 1.5 studieren wir Wahrheitsfunktionen und Normalformen von Formeln. Schließlich gehen wir in Abschnitt 1.6 kurz auf die Bedeutung der Logik für die Informatik ein und skizzieren mögliche Erweiterungen, der hier besprochenen Inhalte.

1.1 Elementare Schlussweisen

Das Fachgebiet der *Logik* beschäftigt sich ganz allgemein mit der Korrektheit von Argumenten: Wie muss ein Argument aussehen, sodass wir es als allgemeingültig betrachten? Oder, negativ ausgedrückt: Wann ist ein Argument nicht korrekt?

Wie argumentieren wir im täglichen Leben? Betrachten wir beispielsweise die folgende Sequenz von Behauptungen, die wir wohl als wahr ansehen können:

> *Sokrates ist ein Mensch.*
> *Alle Menschen sind sterblich.*
> *Somit ist Sokrates sterblich.*

Diese Schlussfigur wird *Syllogismus* genannt und wurde bereits in der Antike untersucht. Aus zwei *Prämissen*, im Fall des Beispiels „Sokrates ist ein Mensch“ und „Alle Menschen sind sterblich“, wird ein dritter Satz geschlossen, also „Somit ist Sokrates sterblich“. Diesen dritten Satz nennt man *Konklusion*. Das Wichtige bei dieser Folge von Behauptungen ist

$\neg$	
T	F
F	T

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

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

$\rightarrow$	T	F
T	T	F
F	T	T

Abbildung 1.1: Wahrheitstabellen

nicht, dass die Prämissen wahr sind, sondern dass die *Schlussfigur* korrekt ist: Wenn die Prämissen wahr sind, dann muss auch die Konklusion wahr sein. Um dies zweifelsfrei behaupten zu können, sucht man nach allgemeinen Strukturen, die Argumentformen aufweisen können. Schlussfolgerungen wie die in dem oben angegebenen Syllogismus sind ein wenig speziell und verwenden implizit bereits Quantifizierungen über eine Menge von Objekten. Aber ein Argument wie das Folgende, das in der Logik *Modus Ponens* genannt wird, ist einleuchtend.

> *Wenn das Kind schreit, hat es Hunger.*
> *Das Kind schreit.*
> *Also hat das Kind Hunger.*

Da die Korrektheit des *Modus Ponens* unabhängig vom Wahrheitsgehalt der eigentlichen Aussagen ist, können wir diese Schlussfigur wie folgt verallgemeinern.

> *Wenn A, dann B.*
> *A gilt.*
> *Also gilt B.*

Hierbei stehen A und B für *Aussagen*, die entweder wahr oder falsch sein können. Weil A und B als Platzhalter für beliebige Aussagen stehen können, sprechen wir auch von *Aussagenvariablen*. Wie in der gerade dargestellten Argumentkette, gilt im Allgemeinen, dass die Korrektheit oder Gültigkeit einer Schlussfigur nicht von den Aussagenvariablen beziehungsweise deren Werten abhängt. Nur die Art wie diese Aussagenvariablen verbunden sind, ist wichtig.

Um aus Aussagenvariablen komplexere Sachverhalte aufzubauen, verwendet man sogenannte *Junktoren*. Beispiele für Junktoren wären etwa die *Negation* (symbolisch $\neg$), *Konjunktion* ($\wedge$) und *Disjunktion* ($\vee$) sowie die *Implikation* ($\rightarrow$). Die Bedeutung dieser Symbole wird durch die *Wahrheitstafeln* in Abbildung 1.1 definiert. Hier schreiben wir T für eine wahre Aussage und F für eine falsche.

Mit Hilfe dieser Junktoren lässt sich nun der *Modus Ponens* konzise fassen und mit Hilfe der Wahrheitstafeln (oder Wahrheitstabellen) in Abbildung 1.1 überprüft man leicht die Allgemeingültigkeit dieser Schlussfigur. Üblicherweise schreibt man den *Modus Ponens* als Inferenzregel, wie folgt.

$$\frac{A \to B \quad A}{B}$$

Diese Schreibweise trennt die beiden Prämissen $A \to B$ und A durch einen Querstrich von der Konklusion B. Wir können uns diese Regel wie eine Rechenregel vorstellen: Haben wir uns von A und $A \to B$ überzeugt, dann können wir auch B schließen.

Während uns die Wahrheitstafeln die *Bedeutung* (auch die *Semantik*) der Junktoren angibt, erlaubt eine Inferenzregel die *syntaktische* Manipulation mit Aussagenvariablen oder zusammengesetzten Aussagen.

Im Allgemeinen spricht man von einem *Kalkül*, wenn eine fixe (formale) Sprache und Regeln zum Formen bestimmter Ausdrücke in dieser Sprache gegeben sind. Im Weiteren muss den Ausdrücken eine Bedeutung zuordenbar sein und es muss eindeutige Regeln geben, wie ein Ausdruck in einen anderen Ausdruck umgewandelt wird. Etwa können wir die Theorie der natürlichen Zahlen samt ihrer Rechenregeln als Kalkül verstehen. Im nächsten Abschnitt definieren wir die Sprache der Aussagenlogik formal und geben einen korrekten und vollständigen Kalkül der Aussagenlogik an.

1.2 Syntax und Semantik der Aussagenlogik

Gegeben sei eine unendliche Menge AT von *atomaren Formeln* (kurz *Atome* genannt), deren Elemente mit p, q, r, ... bezeichnet werden.

Definition 1.1 (Syntax der Aussagenlogik)**.** Die *Wahrheitswertsymbole* der Aussagenlogik sind

$$\mathsf{True} \qquad \mathsf{False}\,,$$

und die *Junktoren* der Aussagenlogik sind

$$\neg \qquad \wedge \qquad \vee \qquad \to\ .$$

Hier ist $\neg$ der einzige unäre Operator und die anderen Operatoren sind alle zweistellig. Die *Formeln* der Aussagenlogik sind induktiv definiert:

1. Eine atomare Formel p ist eine Formel,
2. ein Wahrheitswertsymbol (True, False) ist eine Formel und,
3. wenn A und B Formeln sind, dann sind

$$\neg A \qquad (A \wedge B) \qquad (A \vee B) \qquad (A \to B)\,,$$

auch Formeln.

Konvention. Zur Erleichterung der Lesbarkeit werden die Klammern um binäre Junktoren oft weggelassen. Dies ist möglich, wenn die folgende Präzedenz der Junktoren gilt: Der Operator $\neg$ bindet stärker als $\vee$ und $\wedge$, welche wiederum stärker als $\rightarrow$ binden.

Die Namen der in Definition 1.1 verwendeten Junktoren wurden schon in Abschnitt 1.1 eingeführt. Die Wahrheitswertsymbole ($\mathsf{True}, \mathsf{False}$) dienen der syntaktischen Repräsentation der oben betrachteten Wahrheitswerte. Damit ist die Sprache der Aussagenlogik, also die *Syntax* vollständig definiert.

Wir schreiben T und F für die beiden betrachteten *Wahrheitswerte*. Wie schon in Abschnitt 1.1 bezeichnet T eine wahre und F eine falsche Aussage.

Definition 1.2 (Wahrheitswert)**.** Eine *Belegung* $\mathsf{v}\colon \mathsf{AT} \rightarrow \{\mathsf{T}, \mathsf{F}\}$ ist eine Abbildung, die Atome mit Wahrheitswerten assoziiert. Wir schreiben $\bar{\mathsf{v}}(A)$ für den *Wahrheitswert* einer Formel A. Der Wahrheitswert $\bar{\mathsf{v}}(A)$ ist definiert als die Erweiterung der Belegung v mit Hilfe der Wahrheitstafeln in Abbildung 1.1.

Eine Formel A für die es zumindest eine Belegung gibt, sodass $\bar{\mathsf{v}}(A) = \mathsf{T}$ nennt man *erfüllbar*. Gibt es keine Belegung, sodass $\bar{\mathsf{v}}(A) = \mathsf{T}$, nennt man A *unerfüllbar*.

Definition 1.3 (Konsequenzrelation)**.** Die *Konsequenzrelation* $\{A_1, \ldots, A_n\} \models B$ (oder kurz *Konsequenz*), beschreibt, dass $\bar{\mathsf{v}}(B) = \mathsf{T}$, wenn gilt $\bar{\mathsf{v}}(A_1) = \mathsf{T}$, $\ldots$, $\bar{\mathsf{v}}(A_n) = \mathsf{T}$ für jede Belegung v. Wir schreiben $\models A$, statt $\varnothing \models A$. Gilt $\models A$, dann heißt die Formel A eine *Tautologie*, beziehungsweise *gültig*. Außerdem schreiben wir der Einfachheit halber oft $A_1, \ldots, A_n \models B$, statt $\{A_1, \ldots, A_n\} \models B$.

Informell bezeugt die Konsequenzrelation, dass aus der Wahrheit der Prämissen A_1, $\ldots$, A_n, die Wahrheit der Konklusion B folgt.

Satz 1.1. *Eine Formel A ist eine Tautologie gdw.*[1] *$\neg A$ unerfüllbar ist.*

Beweis.

1. Wir zeigen zunächst die Richtung von links nach rechts. Angenommen A ist eine Tautologie, dann gilt $\bar{\mathsf{v}}(A) = \mathsf{T}$ für alle Belegungen v, also im Besonderen gilt $\bar{\mathsf{v}}(\neg A) = \mathsf{F}$ für alle Belegungen v. Somit ist $\neg A$ unerfüllbar.

2. Nun zeigen wir die Richtung von rechts nach links. Angenommen $\neg A$ ist unerfüllbar. Dann gilt für alle Belegungen v, dass $\bar{\mathsf{v}}(\neg A) = \mathsf{F}$. Somit gilt für alle Belegungen, dass $\bar{\mathsf{v}}(A) = \mathsf{T}$ und A ist eine Tautologie.

□

[1] Wir schreiben gdw. als Abkürzung für „genau dann, wenn“.

Um die Gültigkeit einer Formel A beziehungsweise ihre Erfüllbarkeit oder Unerfüllbarkeit festzustellen, genügt es, alle möglichen Belegungen v zu betrachten und die entsprechenden Wahrheitswerte zu bestimmen. Obwohl es unendlich viele Belegungen v für A gibt, ist leicht einzusehen, dass es genügt die Belegungen zu betrachten, die Atome in A mit verschiedenen Wahrheitswerten belegen. Die Auflistung aller relevanten Belegungen v, zusammen mit dem Wahrheitswert $\bar{\mathsf{v}}(A)$ wird *Wahrheitstabelle von* A genannt. Wenn A aus n verschiedenen Atomen zusammengesetzt ist, hat die Wahrheitstabelle für A maximal 2^n Zeilen, die wir prüfen müssen. Offensichtlich ist dieses Verfahren sehr einfach und bei kleinen Formeln auch recht schnell durchzuführen. Genauso offensichtlich aber ist die inhärente Komplexität.

1.3 Äquivalenz von Formeln

Definition 1.4 (Äquivalenz)**.** Zwei Formeln A, B sind *(logisch) äquivalent*, wenn $A \models B$ und $B \models A$ gilt. Wenn A und B äquivalent sind, dann schreiben wir kurz $A \approx B$.

Satz 1.2. *$A \approx B$ gilt gdw. $(A \to B) \wedge (B \to A)$ eine Tautologie ist.*

Beweis. Zunächst überlegt man sich leicht anhand der Wahrheitstabelle für $\wedge$, dass $(A \to B) \wedge (B \to A)$ eine Tautologie ist gdw. $(A \to B)$ und $(B \to A)$ Tautologien sind.

Nun betrachten wir die Annahme $A \models B$. Dann gilt für alle Belegungen v, dass $\bar{\mathsf{v}}(A) = \mathsf{T}$ $\bar{\mathsf{v}}(B) = \mathsf{T}$ impliziert. Somit gilt aber auch (kontrollieren Sie mit Hilfe der Wahrheitstabelle für $\to$), dass $\bar{\mathsf{v}}(A \to B) = \mathsf{T}$ für alle v. Also ist $A \to B$ eine Tautologie. Ähnlich folgt aus $B \models A$, dass $B \to A$ eine Tautologie ist. Somit haben wir gezeigt, dass $A \approx B$ impliziert, dass $A \to B$ und $B \to A$ Tautologien sind. Mit Hilfe der anfänglichen Bemerkungen folgt also die Richtung von links nach rechts des Satzes. Die Umkehrung folgt in der gleichen Weise und wird der Leserin überlassen. □

Die Konjunktion ist assoziativ, das heißt wir unterscheiden nicht zwischen $(A \wedge B) \wedge C$, $A \wedge (B \wedge C)$ und $A \wedge B \wedge C$. Statt $A_1 \wedge \cdots \wedge A_n$ schreiben wir auch $\bigwedge_{i=1}^{n} A_i$. Wenn $n = 0$, dann setzen wir $\bigwedge_{i=1}^{0} A_i := \mathsf{True}$. Das gleiche gilt für die Disjunktion und wir verwenden $\bigvee_{i=1}^{n} A_i$ für $A_1 \vee \cdots \vee A_n$ mit $\bigvee_{i=1}^{0} A_i := \mathsf{False}$. Zudem sind diese Junktoren kommutativ. Das heißt $A \wedge B$ und $B \wedge A$ haben die gleiche Bedeutung.

Lemma 1.1. *Es gelten die folgenden elementaren Äquivalenzen:*

$$
\begin{array}{llll}
\neg\neg A \approx A & A \vee \mathsf{True} \approx \mathsf{True} & A \wedge \mathsf{True} \approx A & A \to \mathsf{True} \approx \mathsf{True} \\
 & A \vee \mathsf{False} \approx A & A \wedge \mathsf{False} \approx \mathsf{False} & A \to \mathsf{False} \approx \neg A \\
 & A \vee A \approx A & A \wedge A \approx A & \mathsf{True} \to A \approx A \\
 & A \vee \neg A \approx \mathsf{True} & A \wedge \neg A \approx \mathsf{False} & \mathsf{False} \to A \approx \mathsf{True} \\
 & & & A \to A \approx \mathsf{True}
\end{array}
$$

Lemma 1.2. *Es gelten die folgenden Äquivalenzen zur Umformung verschiedener Junktoren:*

$$A \to B \approx \neg A \vee B \qquad \neg(A \to B) \approx A \wedge \neg B$$
$$A \wedge (B \vee C) \approx (A \wedge B) \vee (A \wedge C) \quad A \vee (B \wedge C) \approx (A \vee B) \wedge (A \vee C)\ .$$

Die letzten beiden Äquivalenzen bedeuten, dass die Distributivgesetze *für Konjunktion und Disjunktion gelten.*

Lemma 1.3. *Es gelten die folgenden* Absorptionsgesetze *zwischen Disjunktion und Konjunktion:*

$$A \wedge (A \vee B) \approx A \qquad A \vee (A \wedge B) \approx A$$
$$A \wedge (\neg A \vee B) \approx A \wedge B \quad A \vee (\neg A \wedge B) \approx A \vee B\ .$$

Lemma 1.4. *Es gelten die* Gesetze von de Morgan*:*

$$\neg(A \wedge B) \approx \neg A \vee \neg B \quad \neg(A \vee B) \approx \neg A \wedge \neg B\ .$$

Alle oben angegebenen Äquivalenzen können durch das Aufstellen von Wahrheitstabellen nachgewiesen werden. Wir werden die Beweise für die Lemmata 1.1– 1.4 im Rahmen von Booleschen Algebren in Kapitel 2 nachholen.

Eine *Teilformel* A einer Formel B ist ein Teilausdruck von B, der wiederum eine Formel ist. Den nächsten Satz geben wir ohne Beweis an, die interessierte Leserin wird an [6] verwiesen.

Satz 1.3. *Seien A und B Formeln und E und F Teilformeln von A und B. Gelte $E \approx F$ und sei B das Resultat der Ersetzung von E durch F in A. Dann gilt $A \approx B$.*

Sei A eine Formel und sei p ein Atom in A. Dann bezeichnet $A\{p \mapsto \mathsf{True}\}$ jene Formel, in welcher alle Vorkommnisse von p durch True ersetzt werden. Synonym definiert man $A\{p \mapsto \mathsf{False}\}$.

Lemma 1.5. *Sei A eine Formel und p ein Atom in A. Es gelten die folgenden Äquivalenzen:*

1. *A ist eine Tautologie gdw. $A\{p \mapsto \mathsf{True}\}$ und $A\{p \mapsto \mathsf{False}\}$ Tautologien sind.*
2. *A ist unerfüllbar gdw. $A\{p \mapsto \mathsf{True}\}$ und $A\{p \mapsto \mathsf{False}\}$ unerfüllbar sind.*

Das obige Lemma liefert die Grundlage für eine Methode die Gültigkeit von Formeln zu überprüfen, welche *Methode von Quine* heißt [8]. Die Atome in der gegebenen Formel werden sukzessive durch True beziehungsweise False ersetzt, sodass grundlegende Äquivalenzen, wie die obigen, verwendbar werden. Diese Methode ist, im Gegensatz zur Wahrheitstabellenmethode, auch für größere Formeln verwendbar. Trotzdem bleibt sie ineffizient.

1.4 Formales Beweisen

Wir wenden uns nun rein syntaktischen Methoden zur Bestimmung der Gültigkeit einer Formel zu, dem *formalen Beweisen*. Dazu wiederholen wir die Inferenzregel *Modus Ponens*:

$$\frac{A \to B \quad A}{B} \tag{1.1}$$

Außerdem führen wir die folgenden *Axiome* ein, die auf *Gottlob Frege* (1848–1925) und *Jan Łukasiewicz* (1878–1956) zurückgehen.

$$A \to (B \to A) \tag{1.2}$$
$$(A \to (B \to C)) \to ((A \to B) \to (A \to C)) \tag{1.3}$$
$$(\neg A \to \neg B) \to (B \to A) \tag{1.4}$$

Definition 1.5 (Ableitung)**.** Sei $\mathcal{G}$ eine endliche Menge von Formeln und F eine Formel.

1. Ein *Beweis* von F aus $\mathcal{G}$ ist eine Sequenz von Formeln $A_1, \ldots, A_n, F$, sodass für $i = 1, \ldots, n$ gilt: entweder $A_i \in \mathcal{G}$ oder A_i ist eines der Axiome (1.2)–(1.4) oder A_i folgt mittels *Modus Ponens* (1.1) aus den Formeln A_{i_1} und A_{i_2}, wobei $i_1, i_2 < i$.

2. Eine Formel F heißt *beweisbar* aus den Annahmen $\mathcal{G}$, wenn es einen Beweis von F aus $\mathcal{G}$ gibt. Ein Beweis wird oft auch als *Ableitung*, *Herleitung* oder *Deduktion* bezeichnet.

Definition 1.6 (Beweisbarkeitsrelation)**.** Die *Beweisbarkeitsrelation* $\{A_1, \ldots, A_n\} \vdash B$ zeigt an, dass B aus $A_1, \ldots, A_n$ beweisbar ist. Wir schreiben $\vdash A$ statt $\varnothing \vdash A$ und nennen A in diesem Fall *beweisbar*. Der Einfachheit halber schreiben wir oft $A_1, \ldots, A_n \vdash B$, statt $\{A_1, \ldots, A_n\} \vdash B$.

Der Beweis des folgenden Satzes kann zum Beispiel in [8] nachgelesen werden.

Satz 1.4. *Das angegebene Axiomensystem mit der Inferenzregel* Modus Ponens *ist* korrekt *und* vollständig *für die Aussagenlogik:*

$$A_1, \ldots, A_n \models B \quad \textit{gdw.} \quad A_1, \ldots, A_n \vdash B \ .$$

Der nächste Satz, das *Deduktionstheorem* kann das formale Argumentieren erheblich erleichtern.

Satz 1.5. *Wenn $A_1, \ldots, A, \ldots A_n \vdash B$ gilt, dann auch $A_1, \ldots, A_n \vdash A \to B$, das heißt, wenn A eine Prämisse in einem Beweis von B ist, dann existiert ein Beweis von $A \to B$, der A nicht als Prämisse hat.*

Beweis. Angenommen B wird mit einem Beweis der Form $A_1, \ldots, A_n, B$ nachgewiesen, der A als Prämisse verwendet. Obda.[2] können wir annehmen, dass $A_1 = A$. Wir zeigen mit Induktion nach k ($1 \leqslant k \leqslant n$), dass für jedes k ein Beweis von $A \to A_k$ existiert, der die Prämisse A nicht verwendet.

1. Sei $k = 1$. Dann gilt $A_1 = A$ und die Behauptung gilt, da $A \to A$ beweisbar ist.

2. Sei $k > 1$. Nach Induktionsvoraussetzung gilt für alle $l < k$, dass $A \to A_l$ ohne die Prämisse A beweisbar ist. Wir suchen einen Beweis von $A \to A_k$, der A nicht als Prämisse verwendet. Wenn $A_k = A$, dann argumentieren wir wie im Basisfall. Andererseits, wenn A_k ein Axiom oder eine andere Prämisse als A, dann argumentieren wir wie folgt:

1.	A_k	Ein Axiom oder eine andere Prämisse als A
2.	$A_k \to (A \to A_k)$	Axiom (1.2)
3.	$A \to A_k$	1, 2, *Modus Ponens*

   Wenn A_k nun weder ein Axiom noch eine Prämisse ist, dann folgt A_k im ursprünglichen Beweis $A_1, \ldots, A_n, B$ mittels *Modus Ponens*. Etwa folgt A_k aus den Formeln A_i, $A_j = (A_i \to A_k)$, sodass $i, j < k$. Nach Induktionsvoraussetzung existieren Beweise von $A \to A_i$ und $A \to (A_i \to A_k)$, sodass diese Beweise A nicht als Prämisse verwenden. Wir erweitern diese Beweise wie folgt.

1.	Beweis von $A \to A_i$	mit Induktion
2.	Beweis von $A \to (A_i \to A_k)$	mit Induktion
3.	$(A \to (A_i \to A_k)) \to (A \to A_i) \to (A \to A_k)$	Axiom (1.3)
4.	$(A \to A_i) \to (A \to A_k)$	2, 3, *Modus Ponens*
5.	$A \to A_k$	1, 4, *Modus Ponens*

□

Die angegebene Axiomatisierung ist nicht die einzige korrekte und vollständige Menge von Axiomen, die für die Aussagenlogik angegeben werden können. Es existiert eine Vielzahl von Axiomen- und anderen Beweissystemen, die korrekt und vollständig sind. Als Beispiel wollen wir hier das auf Georg Gentzen (1909–1945) zurückgehende *natürliche Schließen* nennen [10]. Es soll auch nicht unerwähnt bleiben, dass Schlussfolgerungen in der Aussagenlogik nicht nur formalisiert werden können, sondern dass es auch Kalküle gibt, die es erlauben einen Beweis automatisch zu finden. Obwohl diese Kalküle im schlimmsten Fall genauso ineffizient sind wie die Wahrheitstabellenmethode, sind diese Methoden in der Praxis sehr schnell in der Lage die Gültigkeit einer Formel zu verifizieren [11].

Definition 1.7. Eine Logik heißt *endlich axiomatisierbar*, wenn es eine *endliche* Menge von Axiomen und Inferenzregeln gibt, die korrekt und vollständig für diese Logik ist.

[2] Wir schreiben ObdA. als Abkürzung für „Ohne Beschränkung der Allgemeinheit“.

1.5 Konjunktive und Disjunktive Normalform

Definition 1.8. Eine *Wahrheitsfunktion* $f\colon \{\mathsf{T},\mathsf{F}\}^n \to \{\mathsf{T},\mathsf{F}\}$ ist eine Funktion die n Wahrheitswerten, einen Wahrheitswert zuordnet.

Das folgende Lemma folgt unmittelbar aus der Definition.

Lemma 1.6. *Zu jeder Formel A existiert eine Wahrheitsfunktion f, die mit Hilfe der Wahrheitstabelle von A definiert wird.*

In diesem Abschnitt werden wir zeigen, dass auch jeder Wahrheitsfunktion in eindeutiger Weise eine Formel zugeordnet werden kann.

Definition 1.9. Sei $f\colon \{\mathsf{T},\mathsf{F}\}^n \to \{\mathsf{T},\mathsf{F}\}$ eine Wahrheitsfunktion. Wir definieren die Menge aller Argumentsequenzen $\mathsf{TV}(f)$, sodass die Wahrheitsfunktion f, T liefert:

$$\mathsf{TV}(f) := \{(s_1,\ldots,s_n) \mid f(s_1,\ldots,s_n) = \mathsf{T}\}\ .$$

Definition 1.10 (Konjunktive und Disjunktive Normalform)**.** Sei A eine Formel.

1. Ein *Literal* ist ein Atom oder die Negation eines Atoms.
2. A ist in *konjunktiver Normalform* (kurz *KNF*), wenn A eine Konjunktion von Disjunktionen von Literalen ist.
3. A ist in *disjunktiver Normalform* (kurz *DNF*), wenn A eine Disjunktion von Konjunktionen von Literalen ist.

Lemma 1.7. *Sei $f\colon \{\mathsf{T},\mathsf{F}\}^n \to \{\mathsf{T},\mathsf{F}\}$ eine Wahrheitsfunktion, sodass $\mathsf{TV}(f) \neq \varnothing$ und $\mathsf{TV}(f) \neq \{\mathsf{T},\mathsf{F}\}^n$. Im Weiteren seien $p_1,\ldots,p_n$ atomare Formeln.*

1. *Wir definieren:*
$$D := \bigvee_{(s_1,\ldots,s_n)\in\mathsf{TV}(f)} \bigwedge_{i=1}^{n} A_i\ ,$$
wobei $A_i = p_i$, wenn $s_i = \mathsf{T}$ und $A_i = \neg p_i$ sonst. Dann ist D eine DNF, deren Wahrheitstabelle der Funktion f entspricht.

2. *Wir definieren:*
$$K := \bigwedge_{(s_1,\ldots,s_n)\notin\mathsf{TV}(f)} \bigvee_{j=1}^{n} B_j\ ,$$
wobei $B_j = p_j$, wenn $s_j = \mathsf{F}$ und $B_j = \neg p_j$ sonst. Dann ist K eine KNF, deren Wahrheitstabelle der Funktion f entspricht.

Beweis. Zunächst betrachten wir die erste Behauptung: Jede Argumentfolge $\overline{s} = (s_1, \ldots, s_n)$ induziert eine Belegung $\mathsf{v}_{\overline{s}}$ in eindeutiger Weise. Sei $A_{\overline{s}} = \bigwedge_{i=1}^{n} A_i$ eine Konjunktion in D, sodass $A_i = p_i$, wenn $s_i = \mathsf{T}$ und $A_i = \neg p_i$ sonst. Es gilt $\bar{\mathsf{v}}(A_{\overline{s}}) = \mathsf{T}$ gdw. $\mathsf{v} = \mathsf{v}_{\overline{s}}$. Da D eine Disjunktion ist, die genau aus den Konjunktionen $A_{(s_1,\ldots,s_n)}$ mit $f(s_1, \ldots, s_n) = \mathsf{T}$ zusammengesetzt ist, gilt:

$$\bar{\mathsf{v}}(D) = \mathsf{T} \qquad \text{gdw.} \qquad \mathsf{v} = \mathsf{v}_{(s_1,\ldots,s_n)} \quad \text{und} \quad f(s_1, \ldots, s_n) = \mathsf{T}\,.$$

Somit stimmen die Wahrheitstabelle von D und die Funktion f überein.

Im Falle der zweiten Behauptung betrachte man eine Disjunktion $B_{\overline{s}}$ in K. Es gilt: $\bar{\mathsf{v}}(B_{\overline{s}}) = \mathsf{F}$ gdw. $\mathsf{v} = \mathsf{v}_{\overline{s}}$. Somit gilt:

$$\bar{\mathsf{v}}(K) = \mathsf{F} \qquad \text{gdw.} \qquad \mathsf{v} = \mathsf{v}_{(s_1,\ldots,s_n)} \quad \text{und} \quad f(s_1, \ldots, s_n) = \mathsf{F}\,.$$

Also stimmen die Wahrheitstabelle von K und die Funktion f überein. □

Der nächste Satz folgt aus den Lemmata 1.6 und 1.7.

Satz 1.6. *Jeder Wahrheitsfunktion ist auf eine eindeutige Weise eine DNF und eine KNF zugeordnet. Umgekehrt induziert jede Formel mit n Atomen eine eindeutige Wahrheitsfunktion in n Variablen.*

Folgerung. *Für jede Formel A existiert eine DNF D und eine KNF K, sodass $A \approx D \approx K$ gilt.*

Beweis. Es genügt auf die Fälle einzugehen, in denen A eine Tautologie oder unerfüllbar ist. Wenn A eine Tautologie, dann setzen wir $D = K := p \vee \neg p$. Andererseits, wenn A unerfüllbar ist, dann setzen wir $D = K := p \wedge \neg p$, wobei p ein beliebiges Atom ist. □

1.6 Zusammenfassung

Obwohl die Logik ursprünglich eine philosophische Disziplin mit einer starken mathematisch-formalen Komponente ist, hat sich die (mathematische) Logik in den letzten Dekaden als die entscheidende Grundlagendisziplin der Informatik herausgestellt. Dies ist leicht erklärbar: Die Hauptaufgabe einer Informatikerin ist es, eine informelle Beschreibung eines Problems (eine *Spezifikation*) durch Abstraktion so umwandeln zu können, dass dieses Problem in einer *formalen Sprache* (also mit einem Programm) gelöst werden kann. In der Logik wurden (über Jahrhunderte) Methoden und Werkzeuge untersucht, die es erlauben diesen Übergang leicht und korrekt durchzuführen.

Die hier betrachtete Aussagenlogik behandelt nur Aussagen, die entweder wahr oder falsch sind. Das heißt die Logik ist *zweiwertig*, da wir genau zwei Wahrheitswerte (T und F) haben

und es gilt das *Gesetz des ausgeschlossenen Dritten*: Entweder gilt eine Aussage A oder ihr Gegenteil $\neg A$. Anders ausgedrückt: $A \vee \neg A$ ist eine Tautologie. Wir können auch Logiken betrachten, für die dies nicht gilt. Einerseits gibt es Logiken, die das Gesetz des ausgeschlossenen Dritten nicht beinhalten. Solche Logiken nennt man *intuitionistisch*. Der zentrale Unterschied zwischen einer intuitionistischen Aussagenlogik und der Aussagenlogik, die wir in diesem Kapitel behandelt haben, ist, dass erstere *indirekte* Beweise nicht erlaubt: jeder Beweis muss direkt sein. Andererseits gibt es Logiken mit mehr als zwei Wahrheitswerten. Diese Logiken nennt man *mehrwertig*.

Wir gehen kurz auf ein Beispiel und eine Anwendung von mehrwertigen Logiken ein. Sei $V \subseteq [0,1]$ eine Menge von Wahrheitswerten, sodass V zumindest die Werte 0 und 1 enthält. Hier steht 0 für eine zweifelsfrei falsche Aussage und 1 für eine zweifelsfrei richtige Aussage. Eine *Łukasiewicz-Belegung* (basierend auf V) ist eine Abbildung $\mathsf{v}\colon \mathsf{AT} \to V$ und diese Belegung wird wie folgt zu einem *Wahrheitswert* erweitert:

$$\begin{aligned}
\bar{\mathsf{v}}(\neg A) &= 1 - \bar{\mathsf{v}}(A) \\
\bar{\mathsf{v}}(A \wedge B) &= \min\{\bar{\mathsf{v}}(A), \bar{\mathsf{v}}(B)\} \\
\bar{\mathsf{v}}(A \vee B) &= \max\{\bar{\mathsf{v}}(A), \bar{\mathsf{v}}(B)\} \\
\bar{\mathsf{v}}(A \to B) &= \min\{1, 1 - \bar{\mathsf{v}}(A) + \bar{\mathsf{v}}(B)\}
\end{aligned}$$

Eine Formel A heißt *gültig*, wenn $\bar{\mathsf{v}}(A) = 1$ für alle Łukasiewicz-Belegungen v.

Mehrwertige Logiken, die auf einer Łukasiewicz-Belegung aufbauen, werden Łukasiewicz-Logiken genannt. Manchmal werden solche Logiken auch *Fuzzy*-Logiken genannt. Obwohl diese Logiken unendlich viele Wahrheitswerte verwenden können, sind sie endlich axiomatisierbar. Die Bedeutung solcher mehrwertigen Logiken in der Informatik wird durch die Möglichkeit gegeben, Unsicherheit von Information auszudrücken. Etwa finden Łukasiewicz-Logiken praktische Anwendung in medizinischen Expertensystemen.

2

Einführung in die Algebra

In diesem Kapitel werden *Boolesche Algebren* sowie *Logische Schaltkreise* eingeführt. Zum leichteren Verständnis der Begrifflichkeiten werden zunächst in Abschnitt 2.1 allgemeine Sachverhalte zu *Algebren* besprochen, die dann im Kapitel 2.2 für Boolesche Algebren verfeinert werden. In Abschnitt 2.3 gehen wir auf universelle Algebren und im Besonderen auf *Gleichungslogik* ein. Das Kapitel 2.4 stellt eine Anwendung der Theorie von Booleschen Algebren in der Realisierung von Schaltkreisen vor. Schließlich gehen wir in Abschnitt 2.5 kurz auf die Bedeutung der Algebra für die Informatik ein und stellen betrachtete Konzepte in einen historischen Kontext.

2.1 Algebraische Strukturen

Algebren erlauben eine abstrakte Beschreibung von Objekten, in dem die Eigenschaften dieser Objekte durch die Operationen, die mit diesen Objekten möglich sind, beschrieben werden.

Definition 2.1 (Algebra)**.** Eine *Algebra* $\mathcal{A} = \langle A_1, \ldots, A_n; \circ_1, \ldots, \circ_m \rangle$ ist eine Struktur, die aus den Mengen $A_1, \ldots, A_n$ und den Operationen $\circ_1, \ldots, \circ_m$ auf diesen Mengen besteht. Die Mengen $A_1, \ldots, A_n$ werden *Träger* (oder auch *Trägermengen*) genannt und nullstellige Operationen nennt man *Konstanten*.

Definition 2.2 (Algebraischer Ausdruck)**.** Sei $\mathcal{A}$ eine Algebra und sei eine unendliche Menge von Variablen $x_1, x_2, \ldots$ gegeben, die als Platzhalter für Objekte in A verwendet werden. Wir definieren die *algebraischen Ausdrücke von* $\mathcal{A}$ induktiv:

1. Konstanten und Variablen sind algebraische Ausdrücke.
2. Wenn $\circ$ eine Operation von $\mathcal{A}$ ist, die n Argumente hat und $A_1, \ldots, A_n$ algebraische Ausdrücke sind, dann ist $\circ(A_1, \ldots, A_n)$ ein algebraischer Ausdruck.

Konvention. Wann immer möglich schreiben wir Operationen in Infixnotation: Also bei einer zweistelligen Operation $\circ$, schreiben wir $A_1 \circ A_2$ statt $\circ(A_1, A_2)$.

Algebraische Ausdrücke spielen eine ähnliche Rolle wie Formeln der Aussagenlogik. Sie stellen eine textuelle Beschreibung bestimmter Objekte der Trägermenge dar.

Definition 2.3. Sei $\mathcal{A}$ eine Algebra und seien A und B algebraische Ausdrücke über der Algebra $\mathcal{A}$. Die Ausdrücke A und B nennen wir *äquivalent*, wenn für alle Instanzen der in A und B verwendeten Variablen die Ausdrücke in der Algebra $\mathcal{A}$ gleich werden. Das heißt für alle Instanzen A' und B' der Ausdrücke A und B, wobei alle Platzhalter in A und B in gleicher Weise durch Elemente in $\mathcal{A}$ ersetzt werden, gilt $A' = B'$. Wenn A äquivalent zu B ist schreiben wir kurz $A \approx B$.[1]

Sei $\mathcal{A}$ eine Algebra mit endlichem Träger A. Dann nennen wir $\mathcal{A}$ *endlich*. Für endliche Algebren können die Operationen anhand einer *Operationstabelle*, die die Ergebnisse der Operationen auf den Trägerelementen angibt, festgelegt werden.

Definition 2.4. Sei $\circ$ eine binäre Operation auf der Menge A.

- Wenn $0 \in A$ existiert, sodass für alle $a \in A$: $a \circ 0 = 0 \circ a = 0$, dann heißt 0 *Nullelement* für $\circ$.
- Wenn $1 \in A$ existiert, sodass für alle $a \in A$: $a \circ 1 = 1 \circ a = a$, dann heißt 1 *Einselement* oder *neutrales Element* für $\circ$.
- Sei 1 das neutrale Element für $\circ$ und angenommen für $a \in A$, existiert $b \in A$, sodass $a \circ b = b \circ a = 1$. Dann heißt b das *Inverse* oder das *Komplement* von a.

Lemma 2.1. *Jede binäre Operation hat maximal ein neutrales Element.*

Beweis. Sei $\circ$ eine binäre Operation auf der Menge A und angenommen e und u sind neutrale Elemente für $\circ$. Wir zeigen, dass $e = u$. Somit kann es nur ein neutrales Element geben.

$$\begin{aligned} e &= e \circ u && \text{da } u \text{ Einselement} \\ &= u && \text{da } e \text{ Einselement .} \end{aligned}$$

□

Definition 2.5. Sei $\mathcal{A} = \langle A; \circ \rangle$ eine Algebra. Dann heißt $\mathcal{A}$

[1] Die Relation $\approx$ wurde bereits für logische Äquivalenz von Formeln eingeführt. Wir überladen dieses Symbol. Einerseits wird immer aus dem Kontext klar sein, ob A, B algebraische Ausdrücke oder logische Formeln sind, andererseits deuten wir dadurch die starke Nähe zwischen den Konzepten der Logik und der Algebra an.

- *Halbgruppe*, wenn $\circ$ assoziativ ist.
- *Monoid*, wenn $\mathcal{A}$ eine Halbgruppe ist und $\circ$ ein neutrales Element besitzt.
- *Gruppe*, wenn $\mathcal{A}$ ein Monoid ist und jedes Element ein Inverses besitzt.

Eine Halbgruppe, ein Monoid oder eine Gruppe heißt *kommutativ*, wenn $\circ$ kommutativ ist.

Lemma 2.2. *Wenn $\mathcal{A} = \langle A; \circ, 1\rangle$ ein Monoid ist, dann ist das Inverse eindeutig.*

Beweis. Sei $a \in A$ und seien b, c Inverse von a. Wir zeigen $b = c$.

$$\begin{aligned} b &= b \circ 1 && 1 \text{ ist neutrales Element} \\ &= b \circ (a \circ c) && c \text{ ist Komplement von } a \\ &= (b \circ a) \circ c && \text{Assoziativität von } \circ \\ &= 1 \circ c && b \text{ ist Komplement von } a \\ &= c && 1 \text{ ist neutrales Element .} \end{aligned}$$

□

Definition 2.6. Sei $\mathcal{A} = \langle A; +, \cdot, 0, 1\rangle$ eine Algebra.

- $\mathcal{A}$ heißt *Ring*, wenn $\langle A; +, 0\rangle$ eine kommutative Gruppe ist und $\langle A; \cdot, 1\rangle$ ein Monoid. Im weiteren muss gelten, dass $\cdot$ über $+$ distribuiert und zwar sowohl von links als auch von rechts. Das heißt für alle $a, b, c \in A$ gilt:

$$a \cdot (b + c) = (a \cdot b) + (a \cdot c) \qquad (b + c) \cdot a = (b \cdot a) + (c \cdot a) \; .$$

- Wenn $\mathcal{A}$ ein Ring ist und darüber hinaus $\langle A \setminus \{0\}; \cdot, 1\rangle$ eine kommutative Gruppe, dann heißt $\mathcal{A}$ ein *Körper*.

2.2 Boolesche Algebra

Definition 2.7 (Boolesche Algebra)**.** Eine Algebra $\mathcal{B} = \langle B; +, \cdot, \bar{}, 0, 1\rangle$ heißt *Boolesche Algebra* wenn gilt:

1. $\langle B; +, 0\rangle$ und $\langle B; \cdot, 1\rangle$ sind kommutative Monoide.
2. Die Operationen $+$ und $\cdot$ distribuieren übereinander. Es gilt also für alle $a, b, c \in B$:

$$a \cdot (b + c) = (a \cdot b) + (a \cdot c) \qquad a + (b \cdot c) = (a + b) \cdot (a + c) \; .$$

3. Für alle $a \in B$ gilt $a + \overline{a} = 1$ und $a \cdot \overline{a} = 0$. Das Element $\overline{a}$ heißt das *Komplement* oder die *Negation* von a.

Konvention. Zur Verbesserung der Lesbarkeit lassen wir das Zeichen $\cdot$ oft weg und schreiben ab statt $a \cdot b$. Außerdem gilt die folgende Präzedenz der Operationen: Am stärksten bindet der Operator $^{-}$, welcher stärker als $\cdot$ und $+$ bindet.

Für Algebren haben wir die Sprache der algebraischen Ausdrücke eingeführt. Spezialisiert auf Boolesche Algebren bezeichnet man solche Ausdrücke als *Boolesche Ausdrücke*.

Definition 2.8 (Boolescher Ausdruck)**.** Sei eine unendliche Menge von Variablen $x_1, x_2, \ldots$ gegeben, die als Platzhalter für Objekte einer Booleschen Algebra verwendet werden. Diese Variablen heißen *Boolesche Variablen*. Wir definieren *Boolesche Ausdrücke* induktiv:

1. 0, 1 und Variablen sind Boolesche Ausdrücke.

2. Wenn A und B Boolesche Ausdrücke sind, dann sind

$$\overline{A} \qquad (A \cdot B) \qquad (A + B)\,,$$

   Boolesche Ausdrücke.

In der Folge definieren wir eine Reihe von Booleschen Algebren, die in vielfacher Weise Anwendung finden.

Definition 2.9. Sei M eine Menge. Wir bezeichnen mit $\mathcal{P}(M)$ die *Potenzmenge* von M, also $\mathcal{P}(M) := \{N \mid N \subseteq M\}$. Wir betrachten die Algebra $\langle \mathcal{P}(M); \cup, \cap, ^{-}, \varnothing, \mathcal{P}(M)\rangle$ wobei $\cup$ die Mengenvereinigung, $\cap$ die Schnittmenge und $^{-}$ die Komplementärmenge bezeichnet. Diese Algebra nennt man *Mengenalgebra*.

Lemma 2.3. *Die Mengenalgebra ist eine Boolesche Algebra.*

Beweis. Es lässt sich leicht nachprüfen, dass alle Axiome der Booleschen Algebra, wie in Definition 2.7 definiert, erfüllt sind. □

Definition 2.10. Sei $\mathbb{B} := \{0, 1\}$, wobei $0, 1 \in \mathbb{N}$. Wir betrachten die Algebra $\langle \mathbb{B}; +, \cdot, ^{-}, 0, 1\rangle$, wobei die Operationen $+, \cdot, ^{-}$ wie in Abbildung 2.1 definiert sind. Die Algebra $\langle \mathbb{B}; +, \cdot, ^{-}, 0, 1\rangle$ nennt man *binäre Algebra*.

Lemma 2.4. *Die binäre Algebra ist eine Boolesche Algebra.*

Beweis. Es lässt sich leicht nachprüfen, dass alle Axiome der Booleschen Algebra, wie in Definition 2.10 definiert, erfüllt sind. □

$\cdot$	1	0
1	1	0
0	0	0

$+$	1	0
1	1	1
1	1	0

$^-$	
1	0
0	1

Abbildung 2.1: Operationen auf $\mathbb{B}$

Ein Vergleich von Abbildung 2.1 mit den in Abbildung 1.1 definierten Junktoren legt einen Zusammenhang von Booleschen Algebren und der Aussagenlogik nahe. Dabei wird das Komplement als Negation, $\cdot$ als Konjunktion und $+$ als Disjunktion interpretiert. Darüber hinaus werden die Zahlen 0, 1 als die Wahrheitswerte F und T interpretiert. In dieser Interpretation spricht man auch von der *Algebra der Wahrheitswerte*.

Umgekehrt können wir die Menge der aussagenlogischen Formeln zusammen mit den in Kapitel 1 definierten Junktoren als Algebra betrachten.

Definition 2.11. Sei Frm die Menge der aussagenlogischen Formeln. Wir betrachten die Algebra $\langle \mathsf{Frm}; \vee, \wedge, \neg, \mathsf{False}, \mathsf{True}\rangle$. In der Algebra $\langle \mathsf{Frm}; \vee, \wedge, \neg, \mathsf{False}, \mathsf{True}\rangle$ entspricht die Äquivalenz von Booleschen Ausdrücken der logischen Äquivalenz.

Lemma 2.5. *Die in Definition 2.11 definierte Algebra ist eine Boolesche Algebra.*

Beweis. Es lässt sich leicht nachprüfen, dass alle Axiome der Booleschen Algebra, wie in Definition 2.11 definiert, erfüllt sind. □

Definition 2.12. Sei n eine natürliche Zahl und sei $\mathbb{B}^n$ das n-fache kartesische Produkt von $\mathbb{B}$: $\mathbb{B}^n := \{(a_1, \ldots, a_n) \mid a_i \in \mathbb{B}\}$. Wir betrachten die Algebra

$$\langle \mathbb{B}^n; +, \cdot, ^-, (0, \ldots, 0), (1, \ldots, 1)\rangle \ ,$$

wobei die Operationen $+$, $\cdot$, $^-$ als die komponentenweise Erweiterung der Operationen in Definition 2.10 definiert sind:

$$\begin{aligned}(a_1, \ldots, a_n) + (b_1, \ldots, b_n) &:= (a_1 + b_1, \ldots, a_n + b_n)\\ (a_1, \ldots, a_n) \cdot (b_1, \ldots, b_n) &:= (a_1 \cdot b_1, \ldots, a_n \cdot b_n)\\ \overline{(a_1, \ldots, a_n)} &:= (\overline{a_1}, \ldots, \overline{a_n}) \ .\end{aligned}$$

Lemma 2.6. *Die in Definition 2.12 definierte Algebra ist eine Boolesche Algebra.*

Beweis. Es lässt sich leicht nachprüfen, dass alle Axiome der Booleschen Algebra, wie in Definition 2.12 definiert, erfüllt sind. □

Definition 2.13. Seien n, m natürliche Zahlen. Sei Abb die Menge der Abbildungen von $\mathbb{B}^n$ nach $\mathbb{B}^m$. Wir betrachten die Algebra

$$\langle \mathsf{Abb}; +, \cdot, \overline{\ }, (\mathbf{0}, \ldots, \mathbf{0}), (\mathbf{1}, \ldots, \mathbf{1}) \rangle \ ,$$

wobei $(\mathbf{0}, \ldots, \mathbf{0})$ und $(\mathbf{1}, \ldots, \mathbf{1})$ konstante Funktionen auf $\mathbb{B}^n \to \mathbb{B}^m$ bezeichnen:

$$\begin{aligned} (\mathbf{0}, \ldots, \mathbf{0}) &\colon (a_1, \ldots, a_n) \mapsto (0, \ldots, 0) \\ (\mathbf{1}, \ldots, \mathbf{1}) &\colon (a_1, \ldots, a_n) \mapsto (1, \ldots, 1) \ . \end{aligned}$$

Wir betrachten die in Definition 2.12 eingeführte Algebra mit Trägermenge $\mathbb{B}^m$. Die dort eingeführten Operationen werden punktweise erweitert:

$$\begin{aligned} (f + g)(a_1, \ldots, a_n) &:= f(a_1, \ldots, a_n) + g(a_1, \ldots, a_n) \\ (f \cdot g)(a_1, \ldots, a_n) &:= f(a_1, \ldots, a_n) \cdot g(a_1, \ldots, a_n) \\ \overline{f}(a_1, \ldots, a_n) &:= \overline{f(a_1, \ldots, a_n)} \ . \end{aligned}$$

Diese Algebra nennt man die *Algebra der n-stelligen Booleschen Funktionen.*

Lemma 2.7. *Die Algebra der n-stelligen Booleschen Funktionen ist eine Boolesche Algebra.*

Beweis. Es lässt sich leicht nachprüfen, dass alle Axiome der Booleschen Algebra, wie in Definition 2.13 definiert, erfüllt sind. □

Aufbauend auf den Axiomen einer Booleschen Algebra gelten eine Reihe von Gleichheiten, die den in Kapitel 1 studierten logischen Äquivalenzen entsprechen. Diese werden in der Folge betrachtet. Wenn die Beweise leicht nachvollziehbar sind, überlassen wir diese der Leserin. Für Boolesche Algebren gilt das *Dualitätsprinzip*. Sei $\mathcal{B}$ eine Boolesche Algebra und gelte die Gleichheit A für $\mathcal{B}$, dann gilt eine entsprechende Gleichheit A' bei der alle Vorkommnisse von $+$ durch $\cdot$ (und umgekehrt) ersetzt werden sowie 0 und 1 vertauscht werden.

Lemma 2.8. *Sei $\mathcal{B}$ eine Boolesche Algebra und sei B die Trägermenge von $\mathcal{B}$. Für alle $a \in B$ gelten die folgenden* Idempotenzgesetze*:*

$$a \cdot a = a \qquad a + a = a \ ,$$

und die folgenden Gesetze für 0 *und* 1*:*

$$0 \cdot a = 0 \qquad 1 + a = 1 \ .$$

Lemma 2.9. *Sei $\mathcal{B}$ eine Boolesche Algebra und sei B die Trägermenge von $\mathcal{B}$. Für alle $a, b \in B$ gelten die folgenden* Absorptionsgesetze*:*

$$a + ab = a \qquad a(a + b) = a$$
$$a + \overline{a}b = a + b \qquad a(\overline{a} + b) = ab$$

Lemma 2.10. *Sei $\mathcal{B}$ eine Boolesche Algebra und sei B die Trägermenge von $\mathcal{B}$. Für alle $a, b \in B$ gilt die Eindeutigkeit des Komplements:*

$$\textit{Wenn } a + b = 1 \textit{ und } ab = 0, \textit{ dann } b = \overline{a}\ .$$

Beweis. Unter der Annahme von $a + b = 1$ und $ab = 0$ gelten die folgenden Gleichheiten:

$$\begin{aligned} b &= b1 = b(a + \overline{a}) \\ &= ba + b\overline{a} = 0 + b\overline{a} && \text{da } ab = 0 \\ &= a\overline{a} + b\overline{a} = (a + b)\overline{a} \\ &= 1\overline{a} && \text{da } a + b = 1 \\ &= \overline{a}\ . \end{aligned}$$

□

Lemma 2.11. *Sei $\mathcal{B}$ eine Boolesche Algebra und sei B die Trägermenge von $\mathcal{B}$. Für alle $a \in B$ gilt das* Involutionsgesetz*:*

$$\overline{\overline{a}} = a\ .$$

Beweis. Nach Definition einer Booleschen Algebra und Kommutativität von $+$ beziehungsweise $\cdot$ gilt $\overline{a} + a = 1$ und $\overline{a}a = 0$. Also ist a das Komplement von $\overline{a}$. □

Lemma 2.12. *Sei $\mathcal{B}$ eine Boolesche Algebra und sei B die Trägermenge von $\mathcal{B}$. Für alle $a, b \in B$ gelten die* Gesetze von de Morgan*:*

$$\overline{a + b} = \overline{a} \cdot \overline{b} \qquad \overline{a \cdot b} = \overline{a} + \overline{b}\ .$$

Beweis. Wir zeigen nur die erste Gleichung, die zweite überlassen wir der Leserin. Zunächst

zeigen wir $(a + b) + \overline{a} \cdot \overline{b} = 1$:

$$\begin{aligned}(a + b) + \overline{a} \cdot \overline{b} &= (a + b + \overline{a})(a + b + \overline{b}) \\ &= (a + \overline{a} + b)(a + b + \overline{b}) \\ &= (1 + b)(a + 1) \\ &= 1 \cdot 1 = 1 \,.\end{aligned}$$

Nun zeigen wir $(a + b) \cdot \overline{a} \cdot \overline{b} = 0$:

$$\begin{aligned}(a + b) \cdot \overline{a} \cdot \overline{b} &= a \cdot \overline{a} \cdot \overline{b} + b \cdot \overline{a} \cdot \overline{b} \\ &= a \cdot \overline{a} \cdot \overline{b} + \overline{a} \cdot b \cdot \overline{b} \\ &= 0 \cdot \overline{b} + \overline{a} \cdot 0 \\ &= 0 + 0 = 0 \,.\end{aligned}$$

Zusammen haben wir die Voraussetzungen von Lemma 2.10 gezeigt. Somit ist $\overline{a} \cdot \overline{b}$ das Komplement von $a + b$. □

Jeder Boolesche Ausdruck repräsentiert eine Boolesche Funktion über dem Träger der zugrundeliegenden Booleschen Algebra und umgekehrt kann jeder Booleschen Funktion ein Boolescher Ausdruck zugeordnet werden.

Definition 2.14. Sei $\mathbb{B} = \{0, 1\}$.

1. Sei F ein Boolescher Ausdruck in den Variablen $x_1, \ldots, x_n$ und bezeichne $F(s_1, \ldots, s_n)$ die Instanz von F, die wir durch Ersetzung von $x_1, \ldots, x_n$ durch die Objekte $s_1, \ldots, s_n$ erhalten.

   Wir definieren die Funktion $f \colon \mathbb{B}^n \to \mathbb{B}$ wie folgt:

   $$f(s_1, \ldots, s_n) := F(s_1, \ldots, s_n) \,.$$

   Dann heißt f die *Boolesche Funktion* zum Ausdruck F.

2. Sei $f \colon \mathbb{B}^n \to \mathbb{B}$ eine Boolesche Funktion und sei F ein Boolescher Ausdruck, dessen Boolesche Funktion gleich f. Dann nennen wir F den Booleschen Ausdruck von f.

Boolesche Funktionen erlauben uns eine direkte Definition der Äquivalenz von Booleschen Ausdrücken.

Satz 2.1. *Seien A, B Boolesche Ausdrücke und seien f, g ihre Booleschen Funktionen. Dann gilt $A \approx B$ gdw. $f = g$ in der Algebra der Booleschen Funktionen.*

Nach Definition sind zwei Boolesche Ausdrücke äquivalent (für die betrachtete Boolesche Algebra $\mathcal{B}$), wenn das Einsetzen von Elementen aus $\mathcal{B}$ zum gleichen Ergebnis führt. Satz 2.1 erlaubt es nun diesen Zusammenhang in Bezug auf eine spezielle Boolesche Algebra, der Algebra der Booleschen Funktionen, zu verifizieren und so Äquivalenzen für alle Booleschen Algebren zu erhalten. Grundlage des Satzes ist der Darstellungssatz von Stone, der besagt, dass jede Boolesche Algebra isomorph zu einer Mengenalgebra ist.

In Kapitel 1 haben wir die konjunktive und disjunktive Normalform eingeführt. Diese Begrifflichkeiten werden auch für Boolesche Algebren verwendet. Da wir alle Gleichheiten nachweisen konnten, die notwendig sind, um die Existenz von konjunktiven und disjunktiven Normalformen zu beweisen, erhalten wir den folgenden Satz.

Satz 2.2. *Jeder Boolesche Ausdruck hat eine konjunktive beziehungsweise eine disjunktive Normalform.*

2.3 Universelle Algebra

Gleichheit ist ein intuitives Konzept und tatsächlich haben wir in den vorhergehenden Abschnitten bereits die Gleichheit von Elementen von Algebren untersucht. In diesen Argumentationen war es nicht notwendig genau darauf einzugehen welche Gesetze für die Gleichheit beziehungsweise das Symbol $=$ überhaupt gelten, die Argumente waren einleuchtend. Es mag den Anschein haben als wäre es nicht wirklich erforderlich den Begriff der Gleichheit näher zu begründen, trotzdem werden wir uns in diesem Abschnitt etwas näher mit ihm beschäftigen und führen ein System von Inferenzregeln ein, die *Gleichungslogik*. Zunächst betrachten wir ein paar Grundbegriffe der *universellen Algebra*.

Definition 2.15 (Signatur)**.** Eine *Signatur* F ist eine Menge von *Funktionssymbolen*, sodass jedem Symbol $f \in \mathsf{F}$ eine *Stelligkeit* n zugeordnet wird. Symbole mit Stelligkeit 0 werden auch *Konstanten* genannt.

Definition 2.16 (Term)**.** Sei F eine Signatur und sei V eine (unendliche) Menge von *Variablen*, sodass $\mathsf{F} \cap \mathsf{V} = \varnothing$. Die Menge $\mathsf{T}(\mathsf{F}, \mathsf{V})$ aller *Terme (über* F*)* ist induktiv definiert:

1. Jedes Element von V ist ein Term.
2. Wenn $n \in \mathbb{N}$ und $f \in \mathsf{F}$ mit Stelligkeit n sowie $t_1, \ldots, t_n$ Terme, dann ist auch $f(t_1, \ldots, t_n)$ ein Term.

Funktionssymbole sind eine *Darstellung* von Operationen und Terme präzisieren den Begriff der *algebraischen Ausdrücke*, den wir nur in Bezug auf eine gegebene Algebra verwendet haben.

Definition 2.17 (Substitution)**.** Sei F eine Signatur und V eine Menge von Variablen. Eine *Substitution* ist eine Abbildung $\sigma\colon \mathsf{V} \to \mathsf{T}(\mathsf{F},\mathsf{V})$, sodass $\sigma(x) \neq x$ nur für endlich viele Variablen x. Die (endliche) Menge der Variablen, die durch die Abbildung σ nicht auf sich selber abgebildet werden, nennt man den *Definitionsbereich* $\mathsf{dom}(\sigma)$ von σ. Wenn $\mathsf{dom}(\sigma) = \{x_1, \ldots, x_n\}$ können wir σ wie folgt schreiben:

$$\sigma = \{x_1 \mapsto \sigma(x_1), \ldots, x_n \mapsto \sigma(x_n)\} .$$

Der *Wertebereich* $\mathsf{range}(\sigma)$ von σ ist definiert als $\mathsf{range}(\sigma) := \{\sigma(x) \mid x \in \mathsf{dom}(\sigma)\}$.

Man sagt oft dass eine Variable x durch eine Substitution σ *instanziiert* wird, wenn gilt $x \in \mathsf{dom}(\sigma)$.

Definition 2.18. Jede Substitution σ kann zu einer Abbildung $\overline{\sigma}\colon \mathsf{T}(\mathsf{F},\mathsf{V}) \to \mathsf{T}(\mathsf{F},\mathsf{V})$ erweitert werden:

$$\overline{\sigma}(t) := \begin{cases} \sigma(t) & \text{wenn } t \in \mathsf{V} \\ f(\overline{\sigma}(t_1), \ldots, \overline{\sigma}(t_n)) & \text{wenn } t = f(t_1, \ldots, t_n) . \end{cases}$$

Die Anwendung (der Erweiterung) einer Substitution auf einen Term ersetzt simultan alle Variablen im Definitionsbereich durch ihr Bild.

Wenn es nicht zu Verwechslungen kommen kann, dann bezeichnen wir die Erweiterung $\overline{\sigma}$ einer Substitution σ, wiederum mit σ. Wir können nun den Begriff einer *Gleichung* syntaktisch über Terme einführen.

Definition 2.19 (Gleichung)**.** Sei F eine Signatur und V eine Menge von Variablen. Eine *Gleichung über der Signatur* F ist ein Paar $s = t$ von Termen $s, t \in \mathsf{T}(\mathsf{F},\mathsf{V})$. Wenn die Signatur F aus dem Kontext ersichtlich ist, dann sprechen wir einfach von einer *Gleichung*. Wir bezeichnen s (t) als die linke (rechte) Seite der Gleichung.

Sei E eine Menge von Gleichungen. Üblicherweise sind wir nicht nur an den Gleichungen in E interessiert, sondern wollen Gleichungen verwenden um die Äquivalenz bestimmter Ausdrücke nachzuweisen. Dafür führen wir die Inferenzregeln in Abbildung 5.1 ein. Die Notation $E \vdash s = t$ drückt aus, dass die Gleichung $s = t$ *syntaktisch* aus den Gleichungen in E folgt. Der horizontale Strich dient dazu die Prämissen der Inferenzregeln von der Konklusion zu trennen. Die ersten drei Regeln a)-c) drücken die *Reflexivität*, *Symmetrie* und *Transitivität* von Gleichungen aus. Die Relation $=$ ist also eine *Äquivalenzrelation*. Die nächste Regel d) drückt aus, dass alle Gleichungen in E auch aus E folgen. Schließlich drücken die beiden letzten Regeln e) und f) den Abschluss unter Substitutionen und Kontexten aus.

a) $\dfrac{}{E \vdash t = t}$ b) $\dfrac{E \vdash s = t}{E \vdash t = s}$

c) $\dfrac{E \vdash s = t \quad E \vdash t = u}{E \vdash s = u}$ d) $\dfrac{s = t \in E}{E \vdash s = t}$

e) $\dfrac{E \vdash s = t}{E \vdash \sigma(s) = \sigma(t)}$ σ eine Substitution

f) $\dfrac{E \vdash s_1 = t_1 \quad \ldots \quad E \vdash s_n = t_n}{E \vdash f(s_1, \ldots, s_n) = f(t_1, \ldots, t_n)}$

Abbildung 2.2: Gleichungslogik

Manchmal ist es nützlich die Gleichungen in E von den syntaktischen Folgerungen dieser Gleichungen in der Notation abzugrenzen. In diesem Fall bezeichnen wir die Elemente von E als *Identitäten*.

Definition 2.20. Sei E eine Menge von Identitäten. Wir definieren die von E induzierte Äquivalenzrelation $=_E$ wie folgt:

$$=_E := \{(s,t) \in \mathsf{T}(\mathsf{F},\mathsf{V}) \times \mathsf{T}(\mathsf{F},\mathsf{V}) \mid E \vdash s = t\} \; .$$

Die Schlussregeln in Abbildung 5.1 formalisieren das Argumentieren mit Gleichungen. Aber sind die angegebenen Regeln *korrekt*? Im vorherigen Abschnitt haben wir dargelegt wie Äquivalenzen zwischen Booleschen Ausdrücken nachgewiesen werden können, indem sie auf Gleichungen reduziert werden. Wenn nun diese Gleichungen in der Gleichungslogik nachgewiesen werden, folgt dann auch die Äquivalenz? Andererseits, sind die Regeln *vollständig*? Können alle Äquivalenzen von algebraischen Ausdrücken auf diese Weise überprüft werden? Wir werden sehen, dass die Antwort auf beide Fragen positiv ist, aber dazu ist es notwendig den Begriff der Gleichheit *semantisch* zu fassen.

Definition 2.21 (Algebra über der Signatur F)**.** Sei F eine Signatur. Eine *Algebra $\mathcal{A}$ über der Signatur* F setzt sich zusammen aus:

1. Einer *Trägermenge* A und
2. einer Abbildung, die jedem Funktionssymbol $f \in \mathsf{F}$ mit Stelligkeit n eine Funktion $f^{\mathcal{A}} \colon A^n \to A$ zuordnet.

Eine Algebra über einer bestimmten Signatur ist natürlich eine Algebra nach Definition 2.1. Allerdings ist die Signatur, also die Operationen auf der Trägermenge der Algebra konkretisiert. Wir sprechen auch einfach von *Algebra*, wenn die Signatur F aus dem Kontext ersichtlich ist.

Definition 2.22. Sei F eine Signatur und $\mathcal{A}$ eine Algebra (über der Signatur F). Eine Gleichung $s = t$ (über der Signatur F) *gilt* in $\mathcal{A}$ gdw. für alle Instanzen der in s und t verwendeten Variablen, die entsprechenden Terme gleich sind.

Analog zu Definition 2.3 definieren wir die Äquivalenz von Termen in einer Algebra.

Definition 2.23 (Semantische Konsequenz)**.** Sei F eine Signatur und E eine Menge von F-Identitäten.

1. Eine Algebra (über der Signatur F) $\mathcal{A}$ heißt *Modell* von E, wenn jede Identität in E in $\mathcal{A}$ gilt. Wir schreiben kurz: $\mathcal{A} \models E$.
2. Eine Gleichung $s = t$ (über der Signatur F) ist eine *semantische Konsequenz* von E (kurz $E \models s = t$), wenn in allen Modellen $\mathcal{A}$ von E gilt: $\mathcal{A} \models s = t$.
3. Wir definieren, die von E induzierte Äquivalenzrelation $\approx_E$ wie folgt:

$$\approx_E := \{(s,t) \in \mathsf{T}(\mathsf{F},\mathsf{V}) \times \mathsf{T}(\mathsf{F},\mathsf{V}) \mid E \models s = t\} .$$

Der nächste Satz zeigt, dass alle syntaktischen Konsequenzen von Identitäten auch semantische Konsequenzen sind und umgekehrt. Für einen Beweis dieses Satzes sei auf [1] verwiesen.

Satz 2.3 (Satz von Birkhoff)**.** *Sei E eine Menge von Identitäten. Die beiden Relationen $=_E$ und $\approx_E$ sind gleich, das heißt für beliebige Terme s, t gilt: $s =_E t$ gdw. $s \approx_E t$.*

2.4 Logische Schaltkreise

In diesem Abschnitt wenden wir Boolesche Algebren auf *logische Schaltkreise* (oft auch *Schaltnetze* genannt) an. Ein logischer Schaltkreis ist eine abstrakte Repräsentation eines *elektronischen Schaltkreises* der eine Funktion implementiert, die zum Beispiel als Eingabewert eine hohe/niedere Spannung erhält und als Ausgangswert eine hohe/niedere Spannung zurückliefert. Elektronische Schaltkreise bilden die Grundlage der Informationsverarbeitung in heutigen Rechnerarchitekturen. Wir beschäftigen uns hier nicht mit der tatsächlichen *Realisierung* von logischen Schaltkreisen, dieses Gebiet wird in der technischen Informatik untersucht, sondern betrachten logische Schaltkreise abstrakt als besondere Instanz einer Booleschen Algebra.

Definition 2.24 (Logischer Schaltkreis)**.** Sei $\mathbb{B} = \{0,1\}$, wobei 0 als niedere Spannung und 1 als hohe Spannung interpretiert wird. Wir betrachten die Algebra

$$\langle \mathbb{B}; +, \cdot, \overline{\ }, 0, 1 \rangle ,$$

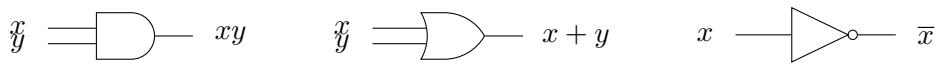

Abbildung 2.3: Logische Gatter

wobei die Operationen $+$, $\cdot$, $^-$ wie in Abbildung 2.1 definiert sind. Diese Boolesche Algebra heißt *Schaltalgebra*. Ein *logischer Schaltkreis* ist ein algebraischer Ausdruck der Schaltalgebra, wobei die Operationen $+$, $\cdot$, $^-$ als *logische Gatter* dargestellt werden. Diese Gatter heißen das *Und*-, das *Oder*- und das *Nicht-Gatter*. Die drei Gatter sind in Abbildung 2.3 dargestellt.

Alternativ zu Definition 2.24 können wir die betrachteten Gatter als Repräsentationen von logischen Junktoren $\wedge$, $\vee$ und $\neg$ betrachten und die Spannungswerte 0 und 1 als F beziehungsweise T interpretieren. Analog zu Definition 2.13 erweitern wir die Schaltalgebra zu einer Algebra von *Schaltfunktionen*.

Definition 2.25. Seien n, m natürliche Zahlen. Sei Abb die Menge der Abbildungen von $\mathbb{B}^n$ nach $\mathbb{B}^m$. Wir betrachten die Boolesche Algebra

$$\langle \mathsf{Abb}; +, \cdot, ^-, (\mathbf{0}, \ldots, \mathbf{0}), (\mathbf{1}, \ldots, \mathbf{1}) \rangle \,,$$

wobei $(\mathbf{0}, \ldots, \mathbf{0})$ und $(\mathbf{1}, \ldots, \mathbf{1})$ konstante Funktionen bezeichnen und $+$, $\cdot$, $^-$ punktweise definiert sind. Diese Boolesche Algebra wird als *Algebra der n-stelligen Schaltfunktionen bezeichnet*.

Wie schon bei Booleschen Ausdrücken entspricht jeder logische Schaltkreis einer Schaltfunktion und umgekehrt. Außerdem gilt Satz 2.1 analog für Schaltfunktionen.

Satz 2.4. *Seien A, B logische Schaltkreise und seien f, g ihre Schaltfunktionen. Dann gilt $A \approx B$ gdw. $f = g$ in der Algebra der Schaltfunktionen.*

Da logische Schaltkreise nur eine andere Darstellung von Booleschen Ausdrücken sind, folgt, dass jede Aussage über die Gleichheit von Schaltfunktionen eine Aussage über die Äquivalenz von Booleschen Ausdrücken ist und somit für *alle* Booleschen Algebren gilt.

Logische Schaltkreise können in vielfältiger Weise kombiniert werden, um einfache arithmetische Operationen, etwa binäre Addition, zu implementieren. Wir realisieren die Funktionen „Übertrag“ und „Summand“ und es ist leicht einzusehen wie aus dem so erhaltenen *Halbaddierer* die binäre Addition zu implementieren ist [8].

Der „Übertrag“ $\mathsf{carry}(a, b)$ ist 1 gdw. $a = 1$ und $b = 1$. Also können wir carry mit Hilfe einer Konjunktion darstellen: $\mathsf{carry}(a, b) = ab$. Nun betrachten wir die „Summand“-Funktion $\mathsf{summand}(a, b)$. Hier erkennen wir, dass $\mathsf{summand}(a, b) = 1$ gdw. $a = 0$ und $b = 1$ gilt oder $a = 1$ und $b = 0$. Somit erhalten wir die folgende Schaltfunktion: $\mathsf{summand}(a, b) = \overline{a}b + a\overline{b}$.

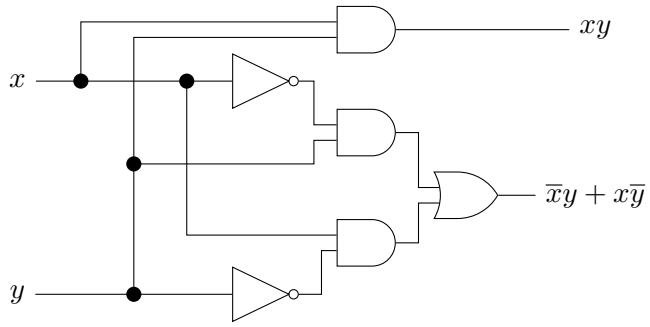

Abbildung 2.4: Halbaddierer

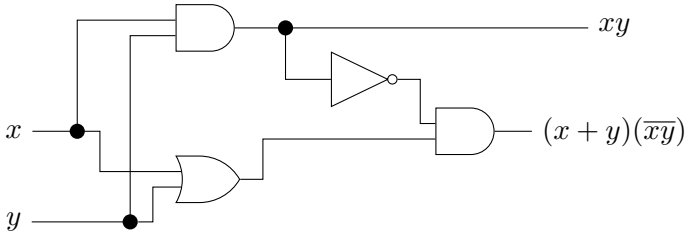

Abbildung 2.5: Einfacherer Halbaddierer

Wenn wir die Operationen carry und summand in geeigneter Weise kombinieren, erhalten wir einen sogenannten *Halbaddierer*. Eine mögliche Realisierung ist in Abbildung 2.4 dargestellt, wo x und y Boolesche Variablen bezeichnen.

Es ist leicht einzusehen, dass der angegebene Schaltkreis zur Realisierung eines Halbaddierers nicht optimal ist. Wir verwenden zwei Nicht-Gatter, drei Und-Gatter und ein Oder-Gatter. Eine Anwendung der Gesetze der Schaltalgebra zeigt, dass wir den Halbaddierer auch mit 4 Gattern realisieren können.

$$\begin{aligned} \overline{a}b + a\overline{b} &= (\overline{a}b + a)(\overline{a}b + \overline{b}) \\ &(a + b)(\overline{a} + \overline{b}) \\ &(a + b)\overline{ab}\ . \end{aligned}$$

Die vereinfachte Realisierung des Halbaddierers ist in Abbildung 2.5 dargestellt.

Dieses Beispiel zeigt, dass es möglich ist die selbe Schaltfunktion mit einfacheren Schaltkreisen zu realisieren, wenn die entsprechenden Booleschen Ausdrücke vereinfacht werden. Üblicherweise wird die Größe eines Schaltkreises über die Anzahl der notwendigen Komponenten einer logisch äquivalenten DNF oder KNF gemessen.

Definition 2.26. Eine *minimale DNF* D eines Booleschen Ausdruckes A ist eine DNF von A, sodass die Anzahl der Konjunktionen in D minimal ist. Wenn zwei DNFs von A die

gleiche Anzahl von Konjunktionen haben, dann ist jene DNF minimal, deren Anzahl von Literalen minimal ist. Die *minimale KNF* ist analog definiert.

Aus Satz 2.2 folgt, dass eine minimale DNF beziehungsweise minimale KNF immer existiert. Allerdings ist es nicht immer leicht die minimale DNF oder KNF zu finden. Auf die dazu entwickelten Verfahren, wie etwa das Erstellen von Karnaugh-Veitch-Diagrammen gehen wir hier nicht näher ein, sondern verweisen auf die Vorlesung „Einführung in die Technische Informatik“ [4].

2.5 Zusammenfassung

Das Wort „Algebra“ kommt von dem arabischen Wort „al-jabr“ im Titel des Lehrbuches „Hisàb al-jabr w'al-muqâbala“, geschrieben um 820 vom Mathematiker und Astronom Al-Khowârizmi. Der Titel bedeutet etwa „Berechnungen durch Sanierung und Vereinfachung“. Wobei „Sanierung“, in Arabisch „al-jabr“, das vielfache Kürzen von Gleichungen bedeutet. Hier sei auch darauf hingewiesen, dass das Wort „Algorithmus“ auf eine fehlerhafte Übersetzung eines anderen Lehrbuches von Al-Khowârizmi zurückgeht. Statt den Titel des Buches zu zitieren wurde der Autor zitiert.

Untersuchungen zur Booleschen Algebra gehen auf den englischen Philosophen George Boole (1815–1864) zurück, der in seinem Hauptwerk „An Investigation of the Laws of Thought“ als Erster ähnliche Strukturen untersucht hat. Wie die Logik ist auch die Boolesche Algebra ursprünglich ein mathematisches Fachgebiet, das in der Informatik Anwendung findet.

Neben den in Abschnitt 2.4 angesprochenen Anwendungen zur Vereinfachung von logischen Schaltkreisen, findet die Boolesche Algebra Anwendung in der Logik, in der Theorie der formalen Sprachen, in der Programmierung sowie in der Statistik. Häufige Anwendungsbereiche von Algebren in der Programmierung sind etwa *abstrakte Datentypen*. Ein abstrakter Datentyp ist benutzerdefiniert und besteht aus einer Menge von Objekten, wie etwa Listen und Operationen auf diesen Objekten. Diese Datentypen werden *abstrakt* genannt, da die *Objekte* und die *Operationen* auf diesen Objekten im Vordergrund stehen. So kann ein Datentyp unabhängig von seiner Implementierung beschrieben werden.

3

Einführung in die Theorie der Formalen Sprachen

In diesem Kapitel werden *formale Sprachen* und *Grammatiken* eingeführt. Im Weiteren wird die *Chomsky-Hierarchie* definiert und es werden zwei Sprachklassen der Chomsky-Hierarchie, die *regulären* und die *kontextfreien* Sprachen näher betrachtet. In Abschnitt 3.1 liegt unser Hauptaugenmerk auf den Grundbegriffen der formalen Sprachen. In Abschnitt 3.2 erklären wir, wie formale Sprache mittels Grammatiken erzeugt werden können und typische Klassen von Sprachen werden eingeführt. In Abschnitt 3.3 konzentrieren wir uns auf eine sehr einfache Klasse von formalen Sprachen, den regulären Sprachen, und beschreiben alternative Repräsentationsformen dieser Sprachen. In Abschnitt 3.4 behandeln wir kurz kontextfreie Sprachen und besprechen eine Anwendung von diesen in Abschnitt 3.5. Schließlich stellen wir in Abschnitt 3.6 die betrachteten Konzepte in einen historischen Kontext und gehen kurz auf die Bedeutung der formalen Sprachen für die Informatik ein.

3.1 Alphabete, Wörter, Sprachen

Ein *Alphabet* Σ ist eine endliche, nicht leere Menge von Symbolen. Gemäß Konvention wird ein Alphabet durch das Symbol Σ dargestellt.

Eine *Zeichenreihe* (auch *Wort* oder *String* genannt) ist eine endliche Folge von Symbolen über Σ. Das *Leerwort* bezeichnet das kleinste vorstellbare Wort: ein Wort ohne Buchstaben. Das Leerwort wird mit ϵ dargestellt.

Konvention. Wir verwenden üblicherweise lateinische Buchstaben vom Anfang des Alphabets, um Elemente des Alphabets zu beschreiben. Im weiteren schreiben wir Buchstaben vom Ende des Alphabets (x, y, ...), um Zeichenreihen zu bezeichnen. Um Verwechslungen auszuschließen führen wir die Konvention ein, dass $\epsilon \notin \Sigma$.

Definition 3.1. Die *Länge* eines Wortes w ist als die Anzahl der Symbole in w definiert. Die Länge von w wird mit $|w|$ bezeichnet; das Leerwort ϵ hat die Länge 0.

Definition 3.2. Wenn Σ ein Alphabet ist, können wir die Menge aller Zeichenreihen einer bestimmten Länge über Σ durch eine Potenznotation bezeichnen. Wir definieren Σ^k als die Menge der Wörter der Länge k, deren Symbole aus Σ stammen. Wir verwenden auch die folgenden Definitionen:

$$\Sigma^+ := \Sigma^1 \cup \Sigma^2 \cup \cdots$$
$$\Sigma^* := \Sigma^+ \cup \{\epsilon\}$$

Jedes Wort w über Σ ist Element von Σ^*.

Formal gilt es zwischen dem *Alphabet* Σ und Σ^1, der Menge der *Wörter* mit Länge 1 über dem Alphabet Σ, zu unterscheiden. Wir identifizieren jedoch Zeichen des Alphabets mit den Wörtern der Länge 1.

Definition 3.3. Angenommen x, y sind Wörter, dann schreiben wir $x \cdot y$ für die *Konkatenation* von x und y. Genauer sei $x = a_1 a_2 \cdots a_i$, $y = b_1 b_2 \cdots b_j$, dann gilt:

$$x \cdot y = a_1 a_2 \cdots a_i b_1 b_2 \cdots b_j \ .$$

Üblicherweise schreibt man die Worte x und y direkt hintereinander als xy, das Zeichen für die Konkatenationsoperation $\cdot$ wird also weggelassen.

Sei Σ ein Alphabet. Wir betrachten die Algebra $\langle \Sigma^*; \cdot \rangle$, wobei $\cdot$ wie in Definition 4.8 definiert ist. Die Konkatenation ist assoziativ und besitzt das Leerwort ϵ als neutrales Element. Also ist die Algebra $\langle \Sigma^*; \cdot \rangle$ ein Monoid. Diese Algebra wird auch als *Wortmonoid* bezeichnet.

Definition 3.4 (Formale Sprache)**.** Eine Teilmenge L von Σ^* heißt eine *formale Sprache* über dem *Alphabet* Σ.

Definition 3.5. Seien L, M formale Sprachen über dem Alphabet Σ. Die *Vereinigung* von L und M ist wie in der Mengenlehre definiert:

$$L \cup M := \{x \mid x \in L \text{ oder } x \in M\} \ .$$

Wir definieren das *Komplement von* L:

$$\sim L = \Sigma^* \setminus L := \{x \in \Sigma^* \mid x \notin L\} \ .$$

Der *Durchschnitt* von L und M ist wie folgt definiert:

$$L \cap M := \{x \mid x \in L \text{ und } x \in M\} .$$

Das *Produkt* von L und M, auch Verkettung von L und M genannt, ist definiert als:

$$LM := \{xy \mid x \in L,\, y \in M\} .$$

Das nächste Lemma folgt unmittelbar aus den Definitionen und der Tatsache, dass $\langle \Sigma^*; \cdot \rangle$ ein Monoid ist.

Lemma 3.1. *Seien L, L_1, L_2, L_3 formale Sprachen, dann gilt*

$$(L_1 L_2) L_3 = L_1 (L_2 L_3) \qquad L\{\epsilon\} = \{\epsilon\} L = L .$$

In der nächsten Definition erweitern wir die Potenznotation für das Alphabet Σ, siehe Definition 3.2, auf Sprachen.

Definition 3.6. Sei L eine formale Sprache und $k \in \mathbb{N}$. Dann ist die *k-te Potenz* von L definiert als:

$$L^k := \begin{cases} \{\epsilon\} & \text{falls } k = 0 \\ L & \text{falls } k = 1 \\ \underbrace{LL \cdots L}_{k\text{-mal}} & \text{falls } k > 1 \end{cases}$$

Der *Kleene-Stern* * oder *Abschluss* von L ist wie folgt definiert:

$$L^* := \bigcup_{k \geqslant 0} L^k = \{x_1 \cdots x_k \mid x_1, \ldots, x_k \in L \text{ und } k \in \mathbb{N}, k \geqslant 0\} .$$

Und wir definieren:

$$L^+ := \bigcup_{k \geqslant 1} L^k = \{x_1 \cdots x_k \mid x_1, \ldots, x_k \in L \text{ und } k \in \mathbb{N}, k > 0\} .$$

3.2 Grammatiken und Formale Sprachen

Grammatiken sind nützliche Modelle zum Entwurf von Software, die Daten mit einer rekursiven Struktur verarbeiten. Das bekannteste Beispiel ist ein *Parser*, also die Komponente eines Compilers, die mit den rekursiv verschachtelten Elementen einer typischen Programmiersprache umgeht, wie beispielsweise arithmetische Ausdrücke und Bedingungsausdrücke.

Vereinfacht ausgedrückt dienen Grammatiken als Regelwerk zur Bildung von *Sätzen* einer Sprache. Die Grammatik der deutschen Sprache etwa, ist ein, meist als höchst kompliziert empfundenes, Regelwerk zur richtigen Bildung deutscher Sätze, die wiederum die deutsche Sprache ausmachen.[1]

Vereinfacht können Sätze als Sequenzen von Wörtern verstanden werden. Also beschreiben Grammatiken, abstrakt gesprochen das „richtige“ Bilden von Mengen von Buchstabensequenzen, also etwa formalen Sprachen. Im Allgemeinen ist unser Interesse aber nicht auf die syntaktische Simulierung von real existierenden Sprachen bezogen, sondern vielmehr auf die Analyse rekursiver Strukturen, wie etwa Programmiersprachen, gerichtet.

Definition 3.7 (Grammatik)**.** Eine Grammatik G ist ein Quadrupel $G = (V, \Sigma, R, \mathsf{S})$, wobei

1. V eine endliche Menge von *Variablen* (oder *Nichtterminale*),
2. Σ ein Alphabet, die *Terminale*, $V \cap \Sigma = \varnothing$,
3. R eine endliche Menge von *Regeln*.
4. $\mathsf{S} \in V$ das *Startsymbol* von G.

Eine Regel ist ein Paar $P \to Q$ von Wörtern, sodass $P, Q \in (V \cup \Sigma)^*$ und in P mindestens eine Variable vorkommt. P nennen wir auch die *Prämisse* und Q die *Konklusion* der Regel.

Konvention. Variablen werden üblicherweise als Großbuchstaben geschrieben und Terminale als Kleinbuchstaben. Wenn es mehrere Regeln mit der gleichen Prämisse gibt werden die Konklusionen auf der rechten Seite zusammengefasst: Statt $P \to Q_1$, $P \to Q_2$, $P \to Q_3$ schreiben wir $P \to Q_1 \mid Q_2 \mid Q_3$.

Definition 3.8. Sei $G = (V, \Sigma, R, \mathsf{S})$ eine Grammatik und sei $x, y \in (V \cup \Sigma)^*$, dann heißt y aus x in G *direkt ableitbar*, wenn gilt:

$$\exists\, u, v \in (V \cup \Sigma)^*,\ \exists\, (P \to Q) \in R \ \text{ sodass } \ (x = uPv \text{ und } y = uQv)\ .$$

In diesem Fall schreiben wir kurz $x \underset{G}{\Rightarrow} y$. Wenn die Grammatik G aus dem Kontext folgt, dann schreiben wir $x \Rightarrow y$.

Definition 3.9. Sei $G = (V, \Sigma, R, \mathsf{S})$ eine Grammatik und $x, y \in (V \cup \Sigma)^*$ Wörter. Dann ist y aus x in G *ableitbar*, wenn es eine natürliche Zahl $k \in \mathbb{N}$ und Wörter $w_0, w_1, \ldots, w_k \in (V \cup \Sigma)^*$ gibt, sodass

$$x = w_0 \Rightarrow w_1 \Rightarrow \ldots \Rightarrow w_k = y\ ,$$

das heißt $x = y$ für $k = 0$. Symbolisch schreiben wir $x \overset{*}{\underset{G}{\Rightarrow}} y$, beziehungsweise $x \overset{*}{\Rightarrow} y$.

[1] Die Leserin kann die Behauptung, dass die deutsche Sprache eine komplizierte Grammatik besitzt, leicht selbst überprüfen, indem sie versucht die Zeichensetzung in diesem Satz auf ihre Korrektheit hin zu untersuchen.

Die vom Startsymbol S ableitbaren Wörter heißen *Satzformen*. Elemente von Σ^* werden *Terminalwörter* genannt. Satzformen, die Terminalwörter sind, heißen *Sätze*. Sätze können mehrere Ableitungen haben und es kann Satzformen geben, die nicht weiter abgeleitet werden können.

Definition 3.10 (Sprache einer Grammatik)**.** Die Menge aller Sätze

$$\mathsf{L}(G) := \{x \in \Sigma^* \mid \mathsf{S} \overset{*}{\underset{G}{\Rightarrow}} x\} \ ,$$

wird die von der Grammatik G *erzeugte Sprache* $\mathsf{L}(G)$ genannt. Zwei Grammatiken G_1 und G_2 heißen *äquivalent*, wenn $\mathsf{L}(G_1) = \mathsf{L}(G_2)$ gilt.

Anhand der zugelassenen Form der Regeln unterscheidet man verschiedene Klassen von Grammatiken.

Definition 3.11. Sei $G = (V, \Sigma, R, \mathsf{S})$ eine Grammatik. Dann heißt G

- *rechtslinear*, wenn für alle Regeln $P \to Q$ gilt, dass $P \in V$ und $Q \in \Sigma^* \cup \Sigma^+ V$,
- *kontextfrei*, wenn für alle Regeln $P \to Q$ gilt, dass $P \in V$ und $Q \in (V \cup \Sigma)^*$,
- *kontextsensitiv*, wenn für alle Regeln $P \to Q$ gilt:
  1. entweder es existieren $u, v, w \in (V \cup \Sigma)^*$ und $A \in V$, sodass
  $$P = uAv \text{ und } Q = uwv \text{ wobei } |w| \geqslant 1 \ ,$$
  2. oder $P = \mathsf{S}$ und $Q = \epsilon$,

  Wenn $\mathsf{S} \to \epsilon \in G$, dann kommt S nicht in einer Konklusion vor.
- *beschränkt*, wenn für alle Regeln $P \to Q$ entweder gilt:
  1. $|P| \leqslant |Q|$ oder
  2. $P = \mathsf{S}$ und $Q = \epsilon$.

  Wenn $\mathsf{S} \to \epsilon \in G$, dann kommt S nicht in einer Konklusion vor.

Aufbauend auf die eingeführten Klassen von Grammatiken, werden entsprechend Klassen von formalen Sprachen definiert.

Definition 3.12. Eine formale Sprache L heißt

- *regulär* oder vom *Typ 3*, wenn eine rechtslineare Grammatik G existiert, sodass $L = \mathsf{L}(G)$,

- *kontextfrei* oder vom *Typ 2*, wenn eine kontextfreie Grammatik G existiert, sodass $L = \mathsf{L}(G)$,
- *kontextsensitiv* oder vom *Typ 1*, wenn eine kontextsensitive Grammatik G existiert, sodass $L = \mathsf{L}(G)$,
- *beschränkt*, wenn eine beschränkte Grammatik G existiert, sodass $L = \mathsf{L}(G)$,
- *rekursiv aufzählbar* oder vom *Typ 0*, wenn eine Grammatik G existiert, sodass $L = \mathsf{L}(G)$.

Zu beachten ist, dass es formale Sprachen gibt, die gar nicht durch eine Grammatik beschrieben werden können. Für formale Sprachen gelten die folgenden Inklusionen, die als *Chomsky-Hierarchie* bekannt sind.

$$\mathcal{L}_3 \subsetneq \mathcal{L}_2 \subsetneq \mathcal{L}_1 \subsetneq \mathcal{L}_0 \subsetneq \mathcal{L} \,,$$

wobei $\mathcal{L}_i$ $(i = 0, 1, 2, 3)$ die Klasse der Sprachen von Typ i und $\mathcal{L}$, die Klasse der formalen Sprachen bezeichnet.

Satz 3.1. *Die Chomsky-Hierarchie ist eine Hierarchie, das heißt alle Inklusionen gelten und sind strikt.*

Beweis. Wir werden hier nur den Beweis führen, dass alle Inklusionen gelten. Für die entscheidende (und wesentlich aufwändigere) Argumentation, dass alle Inklusionen strikt sind, verweisen wir auf [7].

Anhand der Definitionen der Grammatiken ist leicht einzusehen, dass eine rechtslineare Grammatik auch kontextfrei ist. Somit gilt der Zusammenhang $\mathcal{L}_3 \subseteq \mathcal{L}_2$. Jede kontextfreie Grammatik, die keine Regeln der Form $A \to \epsilon$ verwendet, ist laut Definition auch eine kontextsensitiv Grammatik. Regeln der Form $A \to \epsilon$ werden *ϵ-Produktionen* genannt. Man kann zeigen, dass man jede kontextfreie Grammatik mit ϵ-Produktionen in eine kontextfreie Grammatik, die auch eine kontextsensitive Grammatik ist, umschreiben kann [7]. Zusammenfassend gilt $\mathcal{L}_2 \subseteq \mathcal{L}_1$. Im weiteren gilt offensichtlich, dass eine kontextsensitive Grammatik überhaupt eine Grammatik ist, somit folgt $\mathcal{L}_1 \subseteq \mathcal{L}_0$. Schließlich beschreibt jede Grammatik eine formale Sprache, also folgt $\mathcal{L}_0 \subseteq \mathcal{L}$. □

Die Chomsky-Hierarchie erwähnt beschränkte Grammatiken nicht. Das erklärt sich durch den nächsten Satz, für dessen Beweis wir ebenfalls auf [7] verweisen.

Satz 3.2. *Eine Sprache L ist kontextsensitiv gdw. L beschränkt ist.*

3.3 Reguläre Sprachen

Wir haben oben festgelegt, dass eine Sprache *regulär* heißt, wenn sie von einer rechtslinearen Grammatik beschrieben wird. Reguläre Sprachen sind die einfachste Klasse von Sprachen in der Chomsky-Hierarchie. Reguläre Sprachen haben eine derart große Bedeutung, dass neben der Charakterisierung von regulären Sprachen durch Grammatiken auch Beschreibungen mit Hilfe von *endlichen Automaten* und *regulären Ausdrücken* untersucht werden [9].

Reguläre Sprachen finden etwa in den folgenden Bereichen ihre Anwendung.

- Software zum Entwurf und Testen von *digitalen Schaltkreisen.*
- Softwarebausteine eines Compilers. Der *lexikalische Scanner* („*Lexer*“) eines Compilers wird üblicherweise mit Hilfe von endlichen Automaten implementiert und dient zur Aufteilung des Eingabetextes in logische Einheiten, wie Bezeichner oder Schlüsselwörter.
- Software zum *Durchsuchen* umfangreicher Texte, wie Sammlungen von Webseiten, um Vorkommen von Wörtern, Ausdrücken oder anderer Muster zu finden.
- Software zur Verifizierung aller Arten von Systemen, die eine endliche Anzahl verschiedener Zustände besitzen, wie Kommunikationsprotokolle oder *Protokolle* zum sicheren Datenaustausch.
- Softwarebausteine eines Computerspiels. Die Logik bei der *Kontrolle von Spielfiguren* kann mit Hilfe eines endlichen Automaten implementiert werden. Dies erlaubt eine bessere Modularisierung des Codes.

In diesem Abschnitt werden zunächst endliche Automaten informell anhand einer kleinen Anwendung eingeführt. Endliche Automaten stellen ein einfaches formales Modell dar, das trotz oder gerade wegen seiner Einfachheit vielfache Verwendung findet. Anschließend an die Motivation wird der Zusammenhang zwischen endlichen Automaten und rechtslinearen Grammatiken skizziert. Vertiefungen der präsentierten Begriffe sowie Beweise für die aufgestellten Behauptungen werden in der Vorlesung „Diskrete Mathematik“ behandelt [5, 12].

Im einführenden Beispiel untersuchen wir Protokolle, die den Gebrauch elektronischen „Geldes“ ermöglichen. Mit elektronischem „Geld“ sind Dateien gemeint, mit denen Kunden Waren im Internet bezahlen können. Es handeln drei Parteien: der *Kunde*, die *Bank* und das *Geschäft*. Die Interaktion zwischen diesen Parteien ist auf die folgenden fünf Aktionen beschränkt:

- Der Kunde kann *zahlen*, das heißt der Kunde sendet das Geld beziehungsweise weist die Bank an, an seiner Stelle zu zahlen.
- Der Kunde kann das Geld *löschen.*

- Das Geschäft kann dem Kunden Waren *zusenden*.
- Das Geschäft kann Geld *einlösen*.
- Die Bank kann Geld *überweisen*.

Wir treffen die folgenden *Grundannahmen*:

- Der Kunde ist *unverantwortlich*.
- Das Geschäft ist *verantwortlich*, aber *gutgläubig*.
- Die Bank ist *strikt*.

Die Handlungen werden in einem *Protokoll* zusammengefasst. Protokolle dieser Art lassen sich durch endliche Automaten darstellen.

- Jeder Zustand repräsentiert die *Situation* eines Partners.
- Zustandsübergänge entsprechen *Aktionen* oder *Handlungen* der Partner.
- Wir betrachten diese Handlungen als „extern“; die Sequenz der Handlungen ist wichtig, nicht wer sie initiiert.

In Abbildung 3.1 präsentieren wir die endlichen Automaten, welche die Aktionen der drei Partner beschreiben. In der Folge bezeichnen wir diese Automaten mit G (für den Geschäftsautomaten), K (für den Kundenautomaten) und B (für den Bankautomaten).

Unsere Spezifikation des Protokolls mit endlichen Automaten ist leicht unterrepräsentiert. Etwa ist derzeit ungeklärt wie die drei Automaten zusammenwirken: In welchen Zustand soll G wechseln, wenn der Kunde beschließt das elektronische Geld zu löschen? Da das Geschäft von dieser Aktion nicht (direkt) betroffen ist, sollte der Zustand beibehalten werden. Dies muss jedoch noch explizit vereinbart werden. Außerdem fehlen noch Übergänge für Verhalten der Agenten, die nicht beabsichtigt sind. Auch in diesem Fall muss vereinbart werden, dass die jeweiligen Automaten in ihrem bisherigen Zustand verharren. In der exakten Formalisierung von endlichen Automaten (siehe Definition 3.13) wird das Problem dadurch gelöst, dass die Automaten gemeinsam auf *alle* möglichen Aktionen reagieren können müssen. Das heißt wir müssen die Automaten explizit dazu befähigen gewisse Aktionen zu ignorieren. Die erweiterten Automaten sind in Abbildung 3.2 dargestellt, dabei verwenden wir die folgenden Abkürzungen:

Zahlen... Z Einlösen... E Löschen... Lö Liefern... L Überweisen... U

Nun erweitern wir unsere Automaten, sodass wir die Interaktionen zwischen den Agenten abbilden können. Um diese Interaktion zu modellieren, genügt es, die Interaktion zwischen dem Automaten B, der die Bankgeschäfte beschreibt, und dem Automaten G, der die Aktionen des Geschäftes darstellt, zu beschreiben. Dies geschieht, indem wir den *Produktautomaten* $B \times G$ aus B und G definieren.

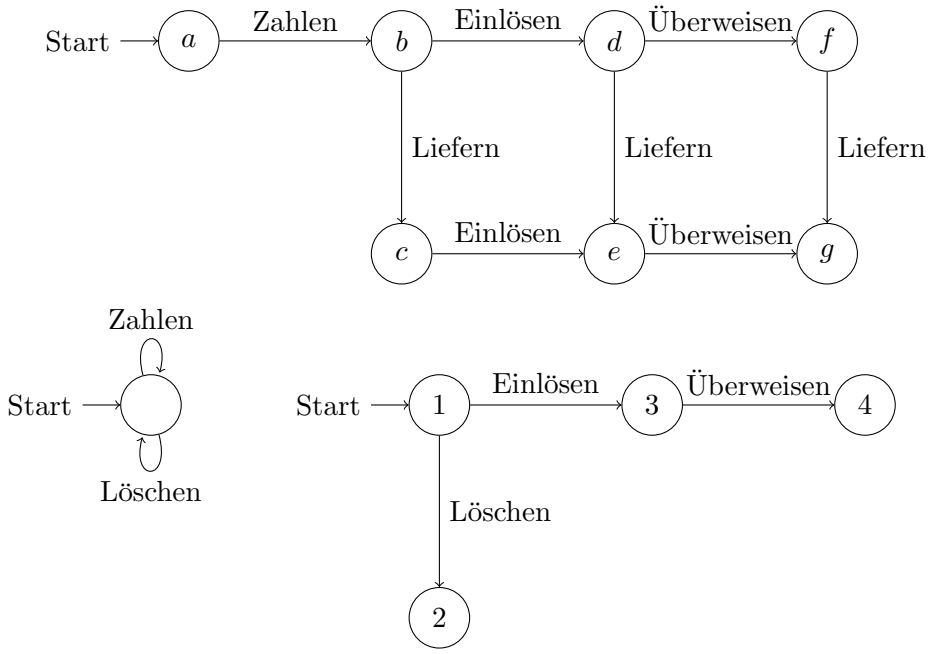

Abbildung 3.1: Endliche Automaten G, K und B

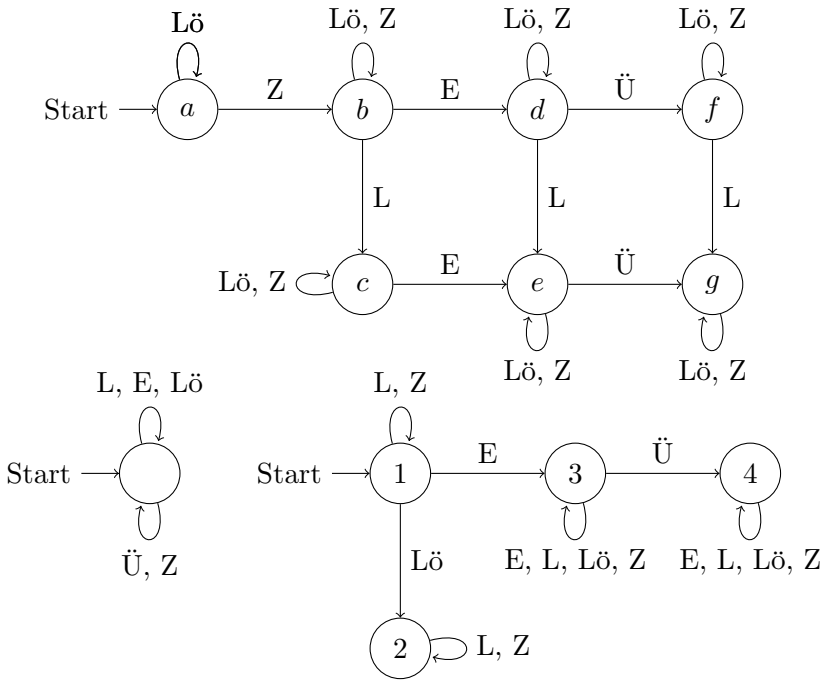

Abbildung 3.2: Endliche Automaten G, K und B

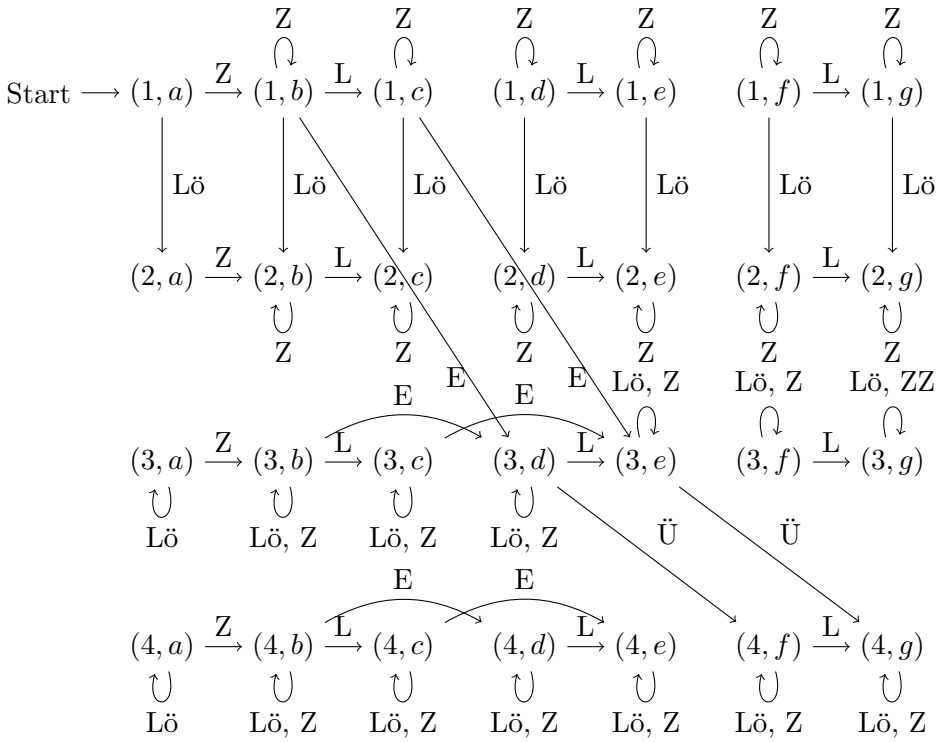

Abbildung 3.3: Produktautomat $B \times G$

1. Die *Zustände* dieses Automaten werden als Paare

$$(i, x) \colon i \in \{1, 2, 3, 4\}, x \in \{a, b, c, d, e, f, g\} \ ,$$

   angegeben.

2. Die *Übergänge* werden durch *paralleles* Ausführen von B und G definiert. Angenommen $B \times G$ ist im Zustand (i, x), das heißt B ist im Zustand i und G im Zustand x. Sei X eine Eingabe. Gelte nun, dass B mit Eingabe X in den Zustand i' wechselt. Andererseits wechselt G bei Eingabe X in den Zustand x'. Dann geht der Zustand (i, x) in $B \times G$ zu (i', x') über. Dieser Übergang wird durch eine Kante mit der Markierung X repräsentiert.

Der Automat, den wir erhalten ist, in Abbildung 3.3 dargestellt. Um Platz zu sparen, ist die Darstellung leicht vereinfacht; auf die Kreise rund um die Zustände wurde verzichtet.

Der Produktautomat $B \times G$ lässt nun einige Schlussfolgerungen zu. Einerseits ist $B \times G$ ineffizient. Von den 28 möglichen Zuständen, sind nur 10 tatsächlich vom Startzustand aus erreichbar, die anderen 18 sind *unerreichbar* und können weggelassen werden. Kritischer ist, dass das Protokoll nicht sicher ist. Der Automat $B \times G$ kann in einen Zustand gelangen, in

	$a_1 \in \Sigma$	$a_2 \in \Sigma$	$\cdots$
$q_1 \in Q$	$\delta(q_1, a_1)$	$\delta(q_1, a_2)$	$\cdots$
$q_2 \in Q$	$\delta(q_2, a_1)$		
$\vdots$	$\vdots$		

Abbildung 3.4: Die Übergangsfunktion dargestellt durch die Zustandstabelle

welchem die Waren geschickt wurden und trotzdem nie eine Überweisung an das Geschäft erfolgen wird. Betrachte etwa den Zustand $(2, c)$. In diesem Zustand hat die Bank einen Antrag das elektronische Geld zu löschen (Lö) bearbeitet. Dies geschah, bevor die Anweisung zu überweisen (Ü) bearbeitet werden konnte. Trotzdem hat das Geschäft die Waren bereits versandt.

Nach dieser informellen Einführung in die Verwendung von endlichen Automaten wenden wir uns der formalen Definition zu.

Definition 3.13 (Deterministischer endlicher Automat)**.** Ein *deterministischer endlicher Automat* (*DEA*) ist ein 5-Tupel $A = (Q, \Sigma, \delta, q_0, F)$, sodass

1. Q eine endliche Menge von *Zuständen*,
2. Σ eine endliche Menge von *Eingabesymbole*, (Σ wird auch *Eingabealphabet* genannt)
3. $\delta\colon Q \times \Sigma \to Q$ die *Übergangsfunktion*,
4. $q_0 \in Q$ der *Startzustand* und
5. $F \subseteq Q$ eine endliche Menge von *akzeptierende Zuständen*.

Die Übergangsfunktion gibt an wie sich der Zustand des Automaten bei einer Eingabe ändern kann. Zu beachten ist, dass δ für alle möglichen Argumente definiert sein muss.

Die Übergangsfunktion kann tabellarisch in der *Zustandstabelle* angegeben werden, siehe Abbildung 3.4. Ähnlich wie in der Motivation kann der Automat durch seinen *Zustandsgraphen* visualisiert werden.

Definition 3.14. Sei $A = (Q, \Sigma, \delta, q_0, F)$ ein DEA, der *Zustandsgraph* ist ein gerichteter Graph, sodass

1. die Ecken die Zustände sind,
2. für Zustände $p, q \in Q$ sind die Kanten von p nach q alle Tripel

$$(p, a, q) \qquad \text{mit} \qquad a \in \Sigma \qquad \text{und} \qquad \delta(p, a) = q\ .$$

Konvention. Üblicherweise schreibt man zu jeder Kante (p, a, q) die Eingabe a, den Startzustand markiert man mit einem Pfeil und die akzeptierenden Zustände werden mit einem doppelten Kreis gekennzeichnet.

Aus Definition 3.13 ist der Zusammenhang von endlichen Automaten zu formalen Sprachen ersichtlich. Die Aktionen, die wir in einem Automaten durchführen können, werden durch Buchstaben eines Alphabets repräsentiert. Sei $A = (Q, \Sigma, \delta, q_0, F)$ und gelte $\delta(p, a) = q$, dann sagen wir der Automat A *liest* den Buchstaben a, wenn er vom Zustand p nach q wechselt. Wir erweitern diesen Zusammenhang auf das Lesen von Wörtern. Dazu definieren wir die *erweiterte Übergangsfunktion*.

Definition 3.15. Sei $A = (Q, \Sigma, \delta, q_0, F)$ ein DEA. Wir definieren $\hat{\delta}\colon Q \times \Sigma^* \to Q$ induktiv.

$$\begin{aligned} \hat{\delta}(q, \epsilon) &:= q \\ \hat{\delta}(q, xa) &:= \delta(\hat{\delta}(q, x), a) \qquad\qquad x \in \Sigma^*,\ a \in \Sigma \end{aligned}$$

Beachten Sie, dass wir hier von unserer Konvention Gebrauch machen, dass Buchstaben vom Ende des Alphabets (in der Definition etwa x) Strings bezeichnen, wohingegen lateinische Buchstaben vom Anfang des Alphabets (etwa a) einzelne Buchstaben im Alphabet bezeichnen.

Sei A wie oben definiert und gelte nun $\hat{\delta}(p, x) = q$, dann sagen wir der Automat A *liest* das Wort x am Weg von Zustand p nach Zustand q. Schließlich können wir die *Sprache* eines DEA definieren.

Definition 3.16 (Sprache eines DEA)**.** Sei $A = (Q, \Sigma, \delta, q_0, F)$ ein DEA. Die Sprache $\mathsf{L}(A)$ von A ist wie folgt definiert:

$$\mathsf{L}(A) := \{x \in \Sigma^* \mid \hat{\delta}(q_0, x) \in F\}\ .$$

Das heißt, die Sprache von A ist die Menge der Zeichenreihen x, die vom Startzustand q_0 in einen akzeptierenden Zustand führen. Wir nennen $\mathsf{L}(A)$ auch die von A *akzeptierte* Sprache.

Der nächste Satz stellt den Zusammenhang zwischen regulären Sprachen und Sprachen, die von einem endlichen Automaten akzeptiert werden können dar. Für den Beweis sei auf [7] verwiesen.

Satz 3.3. *Sei A ein DEA, dann ist $\mathsf{L}(A)$ regulär und umgekehrt existiert zu jeder regulären Sprache L ein DEA A, sodass $L = \mathsf{L}(A)$.*

Für reguläre Sprachen gelten eine Reihe von nützlichen Abschlusseigenschaften.

Satz 3.4. *Die Vereinigung $L \cup M$ zweier regulärer Sprachen L, M ist regulär.*

Satz 3.5. *Sei L regulär (über dem Alphabet Σ), dann ist das Komplement $\sim L = \Sigma^* \setminus L$ ebenfalls regulär.*

Satz 3.6. *Wenn L und M reguläre Sprachen sind, dann ist auch $L \cap M$ regulär.*

Beweis. Wir können das Gesetz von de Morgan anwenden.

$$L \cap M = \sim (\sim L \cup \sim M) \ .$$

Wir haben in den Sätzen 3.4 und 3.5 gezeigt, dass reguläre Sprachen unter Vereinigung und Komplement abgeschlossen sind. □

Satz 3.7. *Wenn L und M reguläre Sprachen sind, dann ist auch $L \setminus M$ regulär.*

Beweis. Dazu verwenden wir:

$$L \setminus M = L \cap \sim M \ .$$

□

Die oben dargestellten Abschlusseigenschaften regulärer Sprachen beziehen sich alle auf Boolesche Operationen. Es gelten aber noch weitere Abschlusseigenschaften, etwa den Abschluss unter *Kleene-Stern*, die *Spiegelung* oder die *Homomorphismusbildung*. Zur weiteren Vertiefung sei auf [9] verwiesen.

3.4 Kontextfreie Sprachen

Eine Sprache heißt *kontextfrei* wenn sie von einer kontextfreien Grammatik (kurz *KFG*) beschrieben wird. Wir skizzieren die Ausdrucksfähigkeit von kontexfreien Sprachen anhand der Sprache der Palindrome über dem Alphabet $\Sigma = \{0, 1\}$. Dabei ist zu beachten, dass diese Sprache nicht regulär ist.

Definition 3.17. Induktive Definition von Palindromen über Σ:

1. ϵ, 0, 1 sind Palindrome.
2. Wenn x ein Palindrom ist, dann sind auch

$$0x0 \qquad 1x1 \ ,$$

   Palindrome.

Die KFG $G = (\{P\}, \Sigma, R, P)$, wobei R wie folgt definiert ist, beschreibt die Sprache der Palindrome.

$$P \to \epsilon \mid 0 \mid 1$$
$$P \to 0P0 \mid 1P1$$

Satz 3.8. $\mathsf{L}(G)$ *ist genau die Menge der Palindrome über dem Alphabet* $\{\mathsf{0}, \mathsf{1}\}$.

Beweis. Wir zeigen, dass $x \in \mathsf{L}(G)$ gdw. x ein Palindrom ist. Wir betrachten die Richtung von links nach rechts, die Umkehrung überlassen wir der Leserin. Angenommen x ist ein Palindrom. Mit Induktion nach $|x|$ zeigt man, dass $x \in \mathsf{L}(G)$.

1. Basis: Wir zeigen die Behauptung für $|x| = 0$ und $|x| = 1$ und betrachten die Worte ϵ, $\mathsf{0}$ und $\mathsf{1}$. Diese sind in $\mathsf{L}(G)$, da die Regeln

$$P \to \epsilon \qquad P \to \mathsf{0} \qquad P \to \mathsf{1}\ ,$$

   in R sind.

2. Schritt: Wir können $|w| \geqslant 2$ annehmen. Da x ein Palindrom ist, muss ein $y \in \Sigma^*$ existieren, sodass:

$$x = \mathsf{0}y\mathsf{0} \quad \text{oder} \quad x = \mathsf{1}y\mathsf{1}\ .$$

   ObdA. sei $x = \mathsf{0}y\mathsf{0}$. Dann ist die Induktionshypothese auf y anwendbar und wir wissen, dass $y \in \mathsf{L}(G)$. Somit gilt $P \overset{*}{\Rightarrow} y$ und daher auch:

$$P \Rightarrow \mathsf{0}P\mathsf{0} \overset{*}{\Rightarrow} \mathsf{0}y\mathsf{0} = x\ .$$

□

Im Induktionschritt des obigen Beweises haben wir implizit den folgenden Sachverhalt zu Ableitungen in der Grammatik G verwendet.

Lemma 3.2. *Sei G eine KFG und sei $A \overset{*}{\Rightarrow} x$ eine Ableitung in G und seien $u, v \in (V \cup \Sigma)^*$. Dann ist auch $uAv \overset{*}{\Rightarrow} uxv$ eine Ableitung in G.*

Beweis. Angenommen es existieren Wörter $w_1, \ldots, w_{k_1} \in (V \cup \Sigma)^*$, sodass

$$A \Rightarrow w_1 \Rightarrow \cdots \Rightarrow w_{k-1} \Rightarrow x\ .$$

Wir zeigen das Lemma mittels Induktion nach k.

1. Basis. Wenn $k = 0$ dann ist nichts zu zeigen.

2. Schritt. Sei $k > 0$. Wir betrachten die Ableitung $A \overset{*}{\Rightarrow} w_{k-1}$. Nach Induktionshypothese existiert eine Ableitung $uAv \overset{*}{\Rightarrow} uw_{k-1}v$. Es genügt also die Ableitung $w_{k-1} \Rightarrow x$ in die Ableitung $uw_{k-1}v \Rightarrow uxv$ umzuschreiben.

□

Wir bezeichnen diesen Vorgang als das *Einbetten* von Ableitungen. Bei der *Linksableitung* wird in jeder Satzform das am weitesten links stehende Nichtterminalsymbol ersetzt, bei der *Rechtsableitung* das am weitesten rechts stehende Nichtterminalsymbol. Sei $G = (V, \Sigma, R, \mathsf{S})$ eine kontexfreie Grammatik und seien $x, y \in (V \cup \Sigma)^*$ Wörter. Wir schreiben $x \underset{\ell}{\Rightarrow} y$, wenn y aus x in G mit Hilfe einer Linksableitung ableitbar ist und $x \underset{\mathrm{r}}{\Rightarrow} y$, wenn y aus x mit Hilfe einer Rechtsableitung folgt.

Definition 3.18. Eine kontexfreie Grammatik G heißt *eindeutig*, wenn jedes Wort $x \in \mathsf{L}(G)$ genau eine Linksableitung besitzt, ansonsten nennt man G *mehrdeutig*.

Statt Regeln von links nach rechts, also *top-down*, auszuwerten, können wir das auch von rechts nach links, also *bottom-up*, tun.

Definition 3.19. Sei $G = (V, \Sigma, R, \mathsf{S})$ eine kontexfreie Grammatik und sei $A \to x$ mit $x \in (V \cup \Sigma)^*$ eine Regel in G. Die *rekursive Inferenz* $\mathsf{L}(A)$ ist induktiv definiert:

1. Wenn $x \in \Sigma^*$, dann $x \in \mathsf{L}(A)$.
2. Wenn $x = X_1 \cdots X_n$ und $x_i \in \mathsf{L}(X_i)$, dann gilt $x_1 x_2 \cdots x_n \in \mathsf{L}(A)$.

Definition 3.20 (Syntaxbaum)**.** Sei $G = (V, \Sigma, R, \mathsf{S})$ eine kontexfreie Grammatik. Ein *Syntaxbaum* für G ist ein Baum S, sodass die folgenden Bedingungen gelten:

1. Jeder innere Knoten von S ist eine Variable in V.
2. Jedes Blatt in S ist entweder ein Terminal aus Σ, ein Nichtterminal aus V oder ϵ. Wenn das Blatt ϵ ist, dann ist dieser Knoten das einzige Kind seines Vorgängers.
3. Sei A ein innerer Knoten, $X_1, \ldots, X_n$ seine Kinder. Dann ist

$$A \to X_1 \cdots X_n \in R\,.$$

Definition 3.21. Sei $G = (V, \Sigma, R, \mathsf{S})$ eine KFG. Das *Ergebnis* eines Syntaxbaums S für G ist das Wort über $(V \cup \Sigma)^*$, das wir erhalten, wenn wir die Blätter in S von links nach rechts lesen.

Satz 3.9. *Sei Σ ein Alphabet und sei $x \in \Sigma^*$. Die folgenden Beschreibungen kontextfreier Sprachen sind* äquivalent*.*

1. *$x \in \mathsf{L}(A)$ nach dem rekursiven Inferenzverfahren.*
2. *$A \overset{*}{\Rightarrow} x$.*
3. *$A \underset{\ell}{\overset{*}{\Rightarrow}} x$.*

4. $A \overset{*}{\underset{r}{\Rightarrow}} x$.

5. *Es existiert ein Syntaxbaum mit Wurzel A und Ergebnis x.*

Beweis. Der Satz folgt aus den Sätzen 3.10 – 3.13. □

Im restlichen Abschnitt werden wir Satz 3.9 beweisen.

Satz 3.10. *Sei $G = (V, \Sigma, R, \mathsf{S})$ eine KFG. Sei A ein Nichtterminal in V. Angenommen $x \in \mathsf{L}(A)$ nach dem rekursiven Inferenzverfahren, dann gibt es einen Syntaxbaum mit Wurzel A und Ergebnis x.*

Beweis. Wir zeigen den Satz mit Induktion über die Anzahl der Schritte in der rekursiven Inferenz von $x \in \mathsf{L}(A)$.

1. Basis: Angenommen es war ein Schritt nötig, um $x \in \mathsf{L}(A)$ festzustellen. Dann muss es eine Regel $A \to x$ in R geben und es ist leicht einen Syntaxbaum von A zu konstruieren, dessen Ergebnis x ist.

2. Schritt: Angenommen $n + 1$ Inferenzschritte wurden durchgeführt, um $x \in \mathsf{L}(A)$ nachzuweisen. Der $n + 1^{\text{te}}$ Schritt erfordert die Existenz einer Regel $A \to X_1 X_2 \cdots X_n$ in R, wobei gelten muss: $x = x_1 x_2 \cdots x_n$ und für alle $i = 1, \ldots, n$: $x_i \in \mathsf{L}(X_i)$. Es gibt zwei Möglichkeiten:

   - Wenn X_i ein Terminalsymbol ist, dann gilt $x_i = X_i$.
   - Wenn X_i eine Variable ist, dann existiert nach Induktionshypothese ein Syntaxbaum mit Ergebnis x_i, dessen Wurzel X_i ist.

   In beiden Fällen ist leicht einzusehen, wie der Syntaxbaum für A definiert werden muss, sodass x das Ergebnis dieses Syntaxbaumes ist.

□

Satz 3.11. *Sei $G = (V, \Sigma, R, \mathsf{S})$ eine KFG, sodass ein Syntaxbaum S mit Wurzel A und Ergebnis $x \in \Sigma^*$ existiert, dann gibt es eine Linksableitung von x aus A in G.*

Beweis. Wir zeigen den Satz mittels Induktion über die Höhe des Syntaxbaumes, also der maximalen Anzahl von Kanten von der Wurzel zu einem Blatt.

1. Basis: Hat der Baum S die Höhe 1, dann gibt es jeweils nur eine Kante von der Wurzel zu den Blättern. Nach Definition 3.20 ist das nur möglich, wenn auch eine Regel $A \to x$ in R vorkommt.

2. Schritt: Angenommen S hat Höhe $n+1$. Dann existiert eine Regel $A \to X_1 X_2 \cdots X_n$ in R und es existieren Worte $x_i \in (V \cup \Sigma)^*$, sodass $x = x_1 \ldots x_n$. Im weiteren hat S n direkte Teilbäume S_i, deren Wurzeln mit X_i markiert sind und der Ergebnisse jeweils x_i ist. Es gelten die folgenden beiden Fälle:
   - Wenn $X_i \in \Sigma$, dann $x_i = X_i$ oder
   - $X_i \in V$, dann ist die Induktionhypothese anwendbar, und es existiert eine Linksableitung $X_i \overset{*}{\underset{\ell}{\Rightarrow}} x_i$.

   Wir konstruieren eine Linksableitung von $x = x_1 x_2 \cdots x_n$:

$$\begin{aligned} A \underset{\ell}{\Rightarrow} X_1 X_2 \cdots X_n &\overset{*}{\underset{\ell}{\Rightarrow}} x_1 X_2 \cdots X_n \\ &\vdots \\ &\overset{*}{\underset{\ell}{\Rightarrow}} x_1 x_2 \cdots x_{n-1} x_n \ . \end{aligned}$$

   Formal zeigen wir, dass für alle $i = 1, \ldots, n$ gilt

$$A \overset{*}{\underset{\ell}{\Rightarrow}} x_1 x_2 \cdots x_i X_{i+1} \cdots X_n \ .$$

□

Satz 3.12. *Sei $G = (V, \Sigma, R, \mathsf{S})$ eine KFG, sodass ein Syntaxbaum mit Wurzel A und Ergebnis $x \in \Sigma^*$ existiert, dann gibt es eine Rechtsableitung von w aus A in G.*

Beweis. Der Beweis verläuft analog zum Beweis von Satz 3.11. □

Als Vorbereitung für den nächsten Satz stellen wir fest, dass wir Ableitungen *aufbrechen* können. Aufbrechen von Ableitungen ist die zur Einbettung inverse Operation, siehe Lemma 3.2.

Lemma 3.3. *Sei G eine KFG und sei $A \Rightarrow X_1 X_2 \ldots X_n \overset{*}{\Rightarrow} x$ eine Ableitung in G. Dann können wir x in die Stücke x_1, x_2, …, x_n brechen, sodass für alle $i = 1, \ldots, n$, $X_i \overset{*}{\Rightarrow} x_i$ Ableitungen in G sind.*

Beweis. Für $X_i \in \Sigma$ ist die Aussage klar: Wenn X_i ein Terminalsymbol, dann $x_i = X_i$ und die Ableitung enthält keine Schritte. Sonst zeigt man zunächst (mit Induktion nach den Ableitungsschritten), dass wenn

$$X_1 X_2 \ldots X_n \overset{*}{\Rightarrow} x \ ,$$

alle Positionen von Expansionen von X_i in x links von Positionen von Expansionen von X_j vorkommen, wenn $i < j$. Somit, wenn $X_i \in V$, dann erhalten wir $X_i \overset{*}{\Rightarrow} x_i$, indem

– alle Positionen der Satzformen links und rechts von Positionen, die aus X_i abgeleitet werden, eliminiert werden und

– überflüssige Schritte eliminiert werden.

□

Satz 3.13. *Sei $G = (V, \Sigma, R, \mathsf{S})$ eine KFG. Angenommen $A \overset{*}{\Rightarrow} x$ mit $x \in \Sigma^*$, dann liefert das rekursive Inferenzverfahren, dass $x \in \mathsf{L}(A)$.*

Beweis. Wir zeigen den Satz mittels Induktion nach der Länge der Ableitung $A \overset{*}{\Rightarrow} x$.

1. Basis: Sei die Ableitung genau ein Schritt. Dann gilt $A \to x \in R$, also gilt $x \in \mathsf{L}(A)$ nach dem Basisfall des rekursive Inferenzverfahren.

2. Schritt: Angenommen $n + 1$ Schritte sind in der Ableitung notwendig:

$$A \Rightarrow X_1 X_2 \cdots X_n \overset{*}{\Rightarrow} x \; .$$

   Wir können x als $x_1 x_2 \cdots x_n$ schreiben, wobei

   – Wenn $X_i \in \Sigma$, dann $X_i = x_i$ oder

   – $X_i \in V$, dann existiert eine Ableitung der Länge (maximal) n $X_i \overset{*}{\Rightarrow} x_i$. Nach Induktionshypothese folgt mit dem rekursiven Inferenzverfahren, dass $x_i \in \mathsf{L}(X_i)$.

   Nach Annahme existiert eine Produktion $A \to X_1 X_2 \cdots X_n \in R$. Somit folgt, dass das Wort $x_1 x_2 \cdots x_n$ in $\mathsf{L}(A)$ ist.

□

3.5 Anwendung kontextfreier Grammatiken: XML

Die klassische Anwendung von kontextfreien Sprachen findet sich in *Parsergeneratoren* oder allgemeiner im Compilerbau. Ein Parsergenerator verwandelt die Beschreibung einer Sprache in einen Parser für diese Sprache. Die Sprache wird üblicherweise mit Hilfe einer kontextfreie Grammatik angegeben. Parser werden zur syntaktischen Analyse von Programmen verwendet. Neuere Anwendungen finden im Bereich der Wissensrepräsentation statt, etwa in *XML*-Dokumenten. Ein essentieller Teil von XML ist die DTD, die *Document Type Definition*, die im Prinzip eine kontextfreie Sprache ist. In der Folge gehen wir nur auf die modernere Anwendung ein.

XML steht für *eXtensible Markup Language* und ist eine *Markup-Sprache* oder *Kennzeichnungssprache* wie HTML. Im Gegensatz zu HTML, dessen Aufgabe die *Formatierung* des Textes ist, ist die Aufgabe von XML den *Inhalt* des Textes zu beschreiben. In XML

haben wir die Möglichkeit, durch benutzerdefinierte Tags eine Aussage über den Text, der zwischen den Tags steht, zu machen. Angenommen wir wollen ausdrücken, dass ein bestimmter Teil des Textes einen Zeitungsartikel beschreiben soll. Dann führen wir das Tag `ARTICLE` ein und schreiben:

`<ARTICLE>` *Artikelbeschreibung* `</ARTICLE>`

Solche Tags werden *Namenselemente* genannt. Wie aber geben wir einem Namenselement Inhalt? Dazu werden entweder „*Document Type Definitions*“ (DTDs) oder *XML-Schemata* verwendet. Eine *DTD* hat die Form

`<!DOCTYPE` *Name der DTD* `[`
Liste der Elementbeschreibungen `]>`

Um Elementbeschreibungen definieren zu können, verwendet man (erweiterte) *reguläre Ausdrücke* [9]. Wir definieren diese induktiv, wobei wir die Bedeutung der Ausdrücke teilweise nur informell einführen.

1. – Namenselemente sind reguläre Ausdrücke
   – Der Ausdruck `#PCDATA`, der jedes Wort ohne XML-Tags bezeichnet ist ein regulärer Ausdruck.
2. – $E \mid F$ bezeichnet die *Vereinigung* der durch E und F beschriebenen Elemente,
   – E, F bezeichnet die *Konkatenation* der durch E udn F beschriebenen Elemente,
   – $E\star$ ($E+$) steht für die beliebige (beliebige, aber mindestens einmalige) Wiederholung der durch E beschriebenen Elemente,
   – $?E$ steht für die Möglichkeit die durch E beschriebenen Elemente optional anzugeben.

Beispiel 3.1. Wir betrachten die folgende *Document Type Definition*:

```
<!DOCTYPE NEWSPAPER [

<!ELEMENT NEWSPAPER (ARTICLE+)>
<!ELEMENT ARTICLE (HEADLINE,BYLINE,LEAD,BODY,NOTES)>
<!ELEMENT HEADLINE (#PCDATA)>
<!ELEMENT BYLINE (#PCDATA)>
<!ELEMENT LEAD (#PCDATA)>
<!ELEMENT BODY (#PCDATA)>
<!ELEMENT NOTES (#PCDATA)>

]>
```

Der *Name* der DTD ist `NEWSPAPER`. Das erste Element—dem Startsymbol einer Grammatik entsprechend—ist ebenfalls `NEWSPAPER`. Die Elementbeschreibung drückt aus, dass das Element `NEWSPAPER` eine nichtleere Sequenz von Artikeln beschreibt. Schließlich ist ein `ARTICLE` die Verknüpfung der folgenden Textelemente:

`HEADLINE`	die Kopfzeile
`BYLINE`	der Untertitel
`LEAD`	die Einleitung
`BODY`	der eigentliche Artikel
`NOTES`	Anmerkungen

In der Folge wandeln wir exemplarisch zwei Elementbeschreibungen in Produktionsregeln einer kontextfreien Grammatik um. Die Definition

```
<!ELEMENT ARTICLE (HEADLINE,BYLINE,LEAD,BODY,NOTES)> ,
```

entspricht der Produktionsregel

$$\texttt{ARTICLE} \to \texttt{HEADLINE BYLINE LEAD BODY NOTES}\,.$$

Nun betrachten wir die Zeile

```
<!ELEMENT NEWSPAPER (ARTICLE+)> ,
```

und wollen diese Beschreibung durch Regeln einer kontextfreien Grammatik ausdrücken. Wir müssen dazu die (leichte) Schwierigkeit bewältigen, dass in der Elementbeschreibung von einer Variante eines Kleene-Sterns Gebrauch gemacht wird. Diese Schwierigkeit bewältigen wir durch die Einführung eines zusätzlichen Nichtterminals `ARTICLES` und erhalten die folgenden drei Produktionsregeln:

$$\begin{aligned} \texttt{NEWSPAPER} &\to \texttt{ARTICLES}\,, \\ \texttt{ARTICLES} &\to \texttt{ARTICLES} \mid \texttt{ARTICLES ARTICLES}\,. \end{aligned}$$

Man kann allgemein zeigen, dass jede Produktion mit (erweiterten) regulären Ausdrücken im Rumpf durch eine Sammlung äquivalenter gewöhnlicher Produktionen ersetzt werden kann [9].

3.6 Zusammenfassung

In den 1940er und 1950er Jahren wurden von einigen Forschern einfachere Maschinen untersucht, die heute als „endliche Automaten“ bezeichnet werden. Diese Automaten, ursprünglich zur Simulation von Gehirnfunktionen von Warren McCulloch (1898–1969) und

Walter Pitts (1923–1969) eingeführt, haben sich für verschiedene andere Zwecke als nützlich erwiesen. In den 1950er Jahren wurden die von McCulloch und Pitts vorgelegten Definition von Stephen Kleene (1909–1994) aufgenommen und mathematisch präzise gefasst. Auf Kleene geht die Definition von *regulären Sprachen* zurück und in seinem Namen wird der Operator * auch oft als *Kleene-Stern* bezeichnet. In den späten 1950er Jahren begann zudem der Linguist Noam Chomsky (1928–), *formale Grammatiken* zu untersuchen. Diese Grammatiken dienen heute als Grundlage einiger wichtiger Softwarekomponenten, wie etwa Compilern.

Anwendungen von regulären Sprachen beziehungsweise regulären Ausdrücken finden sich etwa auch in der Genese des Betriebssystems Unix. Ken Thompson (1943–) baute reguläre Ausdrücke in den Texteditor `qed` ein und später in den Editor `ed`. Unter anderem für ihre Arbeiten zu Unix wurde Thompson und Dennis Ritchie (1941–2011) 1983 der *Turing Award*, der Nobelpreis der Informatik, verliehen. Reguläre Ausdrücke werden seit Beginn bei Unix verwendet. Beispiele hierfür sind `expr`, `awk`, `Emacs`, `vi`, `lex` und `Perl`. Die bessere Integration von regulären Ausdrücken ist das erklärte Ziel in der Entwicklung von `Perl 6`, siehe http://dev.perl.org/perl6.

4

Einführung in die Berechenbarkeitstheorie

In diesem Kapitel wird das formale Modell des endlichen Automaten zu *Turing-vollständigen Berechenbarkeitsmodellen* erweitert und eine Einführung in die *Berechenbarkeitstheorie* gegeben. In Abschnitt 4.1 liegt unser Hauptaugenmerk auf der Klärung der Frage welche Probleme prinzipiell algorithmisch lösbar sind. Das Berechnungsmodell der *Turingmaschinen* wird in Abschnitt 4.2 eingeführt. Diesem Modell werden in Abschnitt 4.3 Registermaschinen gegenübergestellt. Schließlich gehen wir in Abschnitt 4.4 kurz auf die Bedeutung von Berechnungsmodellen für die Informatik ein und stellen betrachtete Konzepte in einen historischen Kontext.

4.1 Algorithmisch unlösbare Probleme

In diesem Abschnitt wollen wir uns einführend mit der Frage beschäftigen, ob alle Probleme algorithmisch lösbar sind. Hier bedeutet „algorithmisch lösbar“, dass es einen *Algorithmus*, das heißt ein Programm gibt, welches das Problem vollständig, das heißt auf allen möglichen Eingaben, löst. Leider müssen wir diese Frage mit „Nein“ beantworten. In der Folge erklären wir, warum diese Antwort nicht wirklich überraschend ist. Zunächst betrachten wir ein sehr einfaches C-Programm P:

```
int main(void) {
    printf("hello, world");
}
```

Wie leicht einzusehen, gibt P die Worte „hello, world“ aus und terminiert. In der Folge nennen wir jedes Programm, das die Zeichenreihe „hello, world“ als die ersten 12 Buchstaben seiner Ausgabe druckt ein *„hello, world“-Programm*. Wir setzen dabei nicht voraus, dass das Programm tatsächlich terminiert.

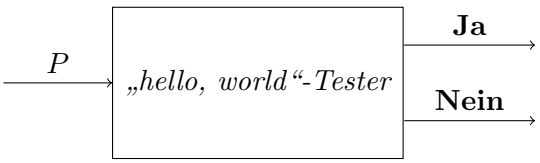

Abbildung 4.1: Ein hypothetischer „hello, world“-Tester

Nun untersuchen wir, ob es möglich ist, ein Programm zu schreiben, das testet, ob ein bestimmtes Programm ein „hello, world“-Programm ist. Schematisch können wir einen hypothetisch angenommenen *„hello, world“-Tester* H wie in Abbildung 4.1 beschreiben. Der Tester H erhält als Eingabe ein Programm P und antwortet entweder mit „Ja“, wenn P ein „hello, world“-Programm ist und sonst mit „Nein“. In Anbetracht der Einfachheit des Programms P erscheint diese Aufgabe recht einfach. Diese Einfachheit ist jedoch trügerisch. Dazu betrachten wir die folgende Variante P_1 des Programms P. Wir setzen die Funktion `exp` voraus, wobei $\texttt{exp}(x, y) = x^y$.

```
int main(void) {
    int n, summe, x, y, z;
    scanf("%d", &n);
    summe = 3;
    while (1) {
        for (x = 1; x <= summe - 2; x++)
            for (y = 1; y <= summe - x - 1; y++) {
                z = summe - x - y;
                if (exp(x, n) + exp(y, n) == exp(z, n))
                    printf("hello, world");
            }
        summe++;
    }
}
```

Es ist nicht schwer einzusehen, dass P_1 die Eingabe einer natürlichen Zahl n erwartet und dann prüft, ob es natürliche Zahlen x, y und z gibt $(x, y, z \geqslant 1)$, sodass die Gleichung

$$x^n + y^n = z^n \,, \tag{4.1}$$

wahr wird. Wenn eine solche Lösung gefunden wird, wird der String „hello, world“ gedruckt. Ist das Programm P_1 ein „hello, world“-Programm? Angenommen als Eingabe setzen wir $n = 2$. Dann ist leicht zu überprüfen, dass $x = 3$, $y = 4$ und $z = 5$ die Gleichung (4.1) löst. Das heißt für $n = 2$ ist P_1 ein „hello, world“-Programm. Genauer sehen wir, dass P_1

„hello, world“ druckt, sobald das erste *pythagoreische Tripel* gefunden wird. Was passiert nun für $n \geqslant 3$? Dann müssen wir feststellen, dass das Programm niemals den String „hello, world“ schreibt, da die Gleichung (4.1) für $n \geqslant 3$ unlösbar ist. Dieser Sachverhalt wurde von Pierre de Fermat (1607–1665) im 17. Jahrhundert vermutet, allerdings dauerte es über 300 Jahre bis der britische Mathematiker Andrew Wiles (1953–) im Jahr 1995 diese Vermutung auch tatsächlich beweisen konnte [14]. Es hat also der Anstrengung von Generationen von Mathematikern und Mathematikerinnen bedurft, um festzustellen, dass das Programm P_1 in bestimmten Fällen *kein* „hello, world“-Programm ist.

Wir betrachten eine weitere Variante P_2 von P. In P_2 setzten wir die Boolesche Funktion `primes` voraus, wobei `primes`$(n) = 1$ gdw. n eine Primzahl ist.[1]

```
int main(void) {
    int summe = 4, x, y, test;
    while (1) {
        test = 0;
        for (x = 2; x <= summe; x++) {
            y = summe - x;
            if (primes(x) && primes(y))
                test = 1;
        }
        if (!test)
            printf("hello, world");
        summe = summe + 2;
    }
}
```

Es ist nicht schwer einzusehen, dass das Programm P_2 ein „hello, world“-Programm ist, wenn eine gerade natürliche Zahl größer als 2 existiert, die nicht als Summe zweier Primzahlen geschrieben werden kann. Ob es überhaupt möglich ist, jede gerade natürliche Zahl größer als 2 als Summe zweier Primzahlen zu schreiben, ist zur Zeit nicht bekannt. Christian Goldbach (1690–1794) hat diese Vermutung (die deshalb *Goldbachsche Vermutung* genannt wird) im 18. Jahrhundert aufgestellt.

Die beiden Varianten des einfachen anfänglichen „hello, world“-Programms zeigen uns, dass die Konstruktion eines „hello, world“-Testers keineswegs eine einfache Angelegenheit ist. In der Tat kann man beweisen, dass es keinen „hello, world“-Tester H geben kann. Wir formulieren allgemein das folgende Problem:

Problem. *Gegeben ein beliebiges Programm P. Ist P ein „hello, world“-Programm?*

[1] Eine solche Funktion wird auch *Primzahltest* genannt. Das Testen, ob eine bestimmte natürliche Zahl eine Primzahl ist, ist ein lösbares Problem.

Dieses Problem ist *algorithmisch nicht lösbar*, da wir keinen Algorithmus dafür angeben können. Probleme, die in diesem Sinn nicht gelöst werden können, nennen wir *unentscheidbar*. Wie kann die algorithmische Unlösbarkeit formal gezeigt werden, wie kommt man also zu der Behauptung, dass etwas nicht algorithmisch lösbar ist? Dazu bräuchte man im Prinzip eine formale Definition was ein „Algorithmus“ ist. Das ist jedoch nicht möglich, da ein „Algorithmus“ ein intuitives Konzept ist. Allerdings hat man so genannte *abstrakte Berechnungsmodelle* studiert, die in einer geeigneten Weise alle Algorithmen darstellen können, die man sich bis jetzt vorstellen hat können. Im weiteren Verlauf dieses Kapitels werden wir zwei solche Modelle studieren: *Turingmaschinen* und *Registermaschinen*. Es kann gezeigt werden, dass diese Modelle äquivalent sind. Jedes Programm, dass auf einer Turingmaschine läuft kann in ein Programm einer Registermaschine umgeschrieben werden und umgekehrt. Ähnliches gilt für alternative Berechnungsmodelle wie etwa Grammatiken, den von Alonzo Church eingeführten *λ-Kalkül* [2] oder *Termersetzungssysteme* [1]: Alle untersuchten Präzisierungen des Begriffs „Algorithmus“ beschreiben exakt die gleiche Menge von Programmen. Diese Beobachtung hat schon in den 1930er Jahren die Grundlage für die so genannte *Church-Turing-These* geliefert:

These. *Jedes algorithmisch lösbare Problem ist auch mit Hilfe einer Turingmaschine lösbar.*

Wir schließen diesen Abschnitt mit einer (sehr) unvollständigen Liste unentscheidbarer Probleme: Die folgenden Probleme sind *unentscheidbar*:

- Das Problem, ob ein beliebiges Programm auf seiner Eingabe terminiert. (*Halteproblem*)
- Postsches Korrespondenzproblem (*PCP*): Gegeben zwei Listen von Wörtern

  $$w_1, w_2, \ldots, w_n \quad \text{und} \quad x_1, x_2, \ldots, x_n \,.$$

  Gesucht sind Indizes $i_1, i_2, \ldots, i_m$, sodass

  $$w_{i_1} w_{i_2} \ldots w_{i_m} = x_{i_1} x_{i_2} \ldots x_{i_m}$$

- Das Problem, ob eine beliebige kontextfreie Grammatik eindeutig ist.

4.2 Turingmaschinen

Im Vergleich zu anderen Konzepten zur Beschreibung der Klasse der berechenbaren Funktionen, stellen Turingmaschinen eines der einfachsten abstrakten Berechnungsmodelle dar.

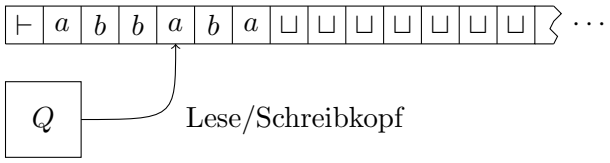

Abbildung 4.2: Schema einer Turingmaschine

Wir beschreiben hier nur deterministische, 1-Band-Turingmaschinen. Äquivalente Formulierungen von Turingmaschinen, wie etwa Maschinen mit mehreren Bändern, mehreren Leseköpfen oder nichtdeterministische Turingmaschinen werden in der Vorlesung „Diskrete Mathematik“ behandelt [5, 12].

Eine *Turingmaschine* (abgekürzt *TM*) besteht aus einer endlichen Anzahl von Zuständen Q, einem einseitig unendlichen Band und einem Lese- und Schreibkopf, der eine Position nach links oder rechts wechseln kann und Symbole lesen beziehungsweise schreiben kann. Das einseitig unendliche Band ist auf der linken Seite durch $\vdash$ begrenzt und unbeschränkt auf der rechten Seite. Skizze 4.2 liefert einen schematisierten Überblick.

Definition 4.1 (Turingmaschine)**.** Eine *deterministische, einbändige Turingmaschine* M ist ein 9-Tupel

$$M = (Q, \Sigma, \Gamma, \vdash, \sqcup, \delta, s, t, r) \ ,$$

sodass

1. Q eine endliche Menge von *Zuständen*,
2. $\Sigma \subseteq \Gamma$ eine endliche Menge von *Eingabesymbolen*,
3. Γ eine endliche Menge von *Bandsymbolen*,
4. $\sqcup \in \Gamma \setminus \Sigma$, das *Blanksymbol*,
5. $\vdash \in \Gamma \setminus \Sigma$, der *linke Endmarker*,
6. $\delta \colon Q \times \Gamma \to Q \times \Gamma \times \{\mathrm{L}, \mathrm{R}\}$ die *Übergangsfunktion*,
7. $s \in Q$, der *Startzustand*,
8. $t \in Q$, der *akzeptierende Zustand* und
9. $r \in Q$, der *verwerfende Zustand* mit $t \neq r$.

Informell bedeutet $\delta(p, a) = (q, b, d)$: „Wenn die TM M im Zustand p das Symbol a liest, dann ersetzt M a durch b, der Lese/Schreibkopf bewegt sich einen Schritt in die Richtung d und M wechselt in den Zustand q.“ Wir verlangen, dass das Symbol $\vdash$ niemals überschrieben werden kann und die Maschine niemals über die linke Begrenzung hinaus fährt. Dies wird

formal durch die folgende Bedingung festgelegt: Für alle $p \in Q$, existiert $q \in Q$ mit:

$$\delta(p, \vdash) = (q, \vdash, \mathrm{R}) \; . \tag{4.2}$$

Außerdem verlangen wir, dass die Maschine, sollte sie den akzeptierenden beziehungsweise verwerfenden Zustand erreicht haben, diesen nicht mehr verlassen kann. Das heißt für alle $b \in \Gamma$ existieren $c, c' \in \Gamma$ und $d, d' \in \{\mathrm{L}, \mathrm{R}\}$ sodass gilt:

$$\delta(t, b) = (t, c, d) \tag{4.3}$$
$$\delta(r, b) = (r, c', d') \tag{4.4}$$

Die Zustandsmenge Q und die Übergangsfunktion δ einer TM M wird auch als die *endliche Kontrolle* von M bezeichnet.

Beachten Sie, dass eine Turingmaschine M nach Definition 4.1 niemals zur Ruhe kommt. Zwar kann M in ihren akzeptierenden oder nicht-akzeptierenden Zustand wechseln und diesen dann auch nicht mehr verlassen, aber M bleibt trotzdem immer in Bewegung. Dennoch sprechen wir vom *Halten* der TM M, wenn M entweder den Zustand t oder r erreicht, andernfalls sagen wir: *M hält nicht* (oder auch *terminiert* nicht).

Zu jedem Zeitpunkt enthält das Band einer TM M ein unendliches Wort der Form $y\sqcup^\infty$, wobei $\sqcup^\infty$ das unendlichen Wort

$$\sqcup\sqcup\sqcup\sqcup\sqcup\sqcup\sqcup\sqcup\sqcup\cdots \; ,$$

bezeichnet. Obwohl das Wort $y\sqcup^\infty$ unendlich ist, ist es endlich repräsentierbar, da nur die Darstellung von y von Interesse ist und $|y| \in \mathbb{N}$.

Definition 4.2 (Konfiguration einer TM)**.** Eine *Konfiguration* einer TM M ist ein Tripel (p, x, n), sodass

- $p \in Q$ der aktuelle Zustand,
- $x = y\sqcup^\infty$ der aktuelle Bandinhalt ($y \in \Gamma^*$) und
- $n \in \mathbb{N}$ die Position des Lese/Schreibkopfes am Band.

Wir verwenden griechische Buchstaben vom Anfang des Alphabets um Konfigurationen zu bezeichnen. Die *Startkonfiguration* bei Eingabe $x \in \Sigma^*$ ist die Konfiguration

$$(s, \vdash x\sqcup^\infty, 0) \; .$$

In der Folge definieren wir eine binäre Relation zwischen Konfigurationen, um einen Rechenschritt einer Turingmaschine konzise formalisieren zu können.

Definition 4.3. Sei $z \in \Gamma^*$, wir schreiben z_n für das n-te Symbol des Wortes z. Die Relation $\xrightarrow[M]{1}$ ist wie folgt definiert:

$$(p, z, n) \xrightarrow[M]{1} \begin{cases} (q, z', n-1) & \text{wenn } \delta(p, z_n) = (q, b, \mathrm{L}) \\ (q, z', n+1) & \text{wenn } \delta(p, z_n) = (q, b, \mathrm{R}) \end{cases}$$

Hier bezeichnet z' das Wort, das wir aus z erhalten, wenn z_n durch b ersetzt wird.

Definition 4.4. Wir definieren die reflexive, transitive Hülle $\xrightarrow[M]{*}$ von $\xrightarrow[M]{1}$ induktiv:

1. $\alpha \xrightarrow[M]{0} \alpha$
2. $\alpha \xrightarrow[M]{n+1} \beta$, wenn $\alpha \xrightarrow[M]{n} \gamma \xrightarrow[M]{1} \beta$ für eine Konfiguration γ und
3. $\alpha \xrightarrow[M]{*} \beta$, wenn $\alpha \xrightarrow[M]{n} \beta$ für ein $n \geqslant 0$.

Definition 4.5 (Sprache einer TM)**.** Sei $M = (Q, \Sigma, \Gamma, \vdash, \sqcup, \delta, s, t, r)$ eine TM. Die Turingmaschine M *akzeptiert* die Eingabe $x \in \Sigma^*$, wenn gilt

$$(s, \vdash x\sqcup^\infty, 0) \xrightarrow[M]{*} (t, y, n) \,,$$

für $y \in \Gamma^*$ und $n \in \mathbb{N}$. M *verwirft* x, wenn

$$(s, \vdash x\sqcup^\infty, 0) \xrightarrow[M]{*} (r, y, n) \,,$$

für $y \in \Gamma^*$ und $n \in \mathbb{N}$. Wir sagen M *hält* bei Eingabe x, wenn M x entweder akzeptiert oder verwirft. Andernfalls *terminiert* M auf x *nicht*. Eine TM M heißt *total*, wenn sie auf allen Eingaben hält. Die Menge $\mathsf{L}(M)$ bezeichnet die Menge aller von M akzeptierten Wörter.

Der folgende Satz zeigt, dass Turingmaschinen die gleichen Sprachen beschreiben können wie Grammatiken. Für den Beweis des Satzes sei auf [7] verwiesen.

Satz 4.1. *Sei L eine Sprache, die von einer TM akzeptiert wird. Dann ist L* rekursiv aufzählbar *(vergleiche Definition 3.12).*

Eine Sprache L heißt *rekursiv*, wenn es eine *totale* TM M gibt, sodass $L = \mathsf{L}(M)$. Die Klasse der rekursiven Sprachen ist echt größer als die Klasse der beschränkten Sprachen, aber echt kleiner als die Klasse der rekursiv aufzählbaren Sprachen [7]. Um Turingmaschinen mit Registermaschinen, die wir im nächsten Abschnitt einführen werden, vergleichen zu können, definieren wir neben der Sprache, die von einer TM akzeptiert wird, wie eine partielle Funktion $f\colon \mathbb{N}^k \to \mathbb{N}$ von einer TM berechnet werden kann. Dazu repräsentieren wir natürliche Zahlen *unär*: Eine Zahl $n \in \mathbb{N}$ wird auf dem Band der TM durch n Wiederholungen des Zeichens $\sqcap$ dargestellt. Wir schreiben abkürzend $\sqcap^n$ für n-maliges Hinschreiben von $\sqcap$.

Definition 4.6 (Berechenbarkeit mit einer TM)**.** Sei

$$M = (Q, \{\sqcap\}, \{\vdash, \sqcup, \sqcap, \square\}, \vdash, \sqcup, \delta, s, t, r) \ ,$$

eine TM. Eine partielle Funktion $f\colon \mathbb{N}^k \to \mathbb{N}$ heißt *M-berechenbar*, wenn gilt

$$f(n_1, \ldots, n_k) = m \quad \text{gdw.} \quad (s, \vdash \ \sqcap^{n_1} \square \cdots \square \sqcap^{n_k} \sqcup^\infty, 0) \xrightarrow[M]{*} (t, \vdash \ \sqcap^m \sqcup^\infty, n) \ .$$

Hier dient das Zeichen $\square$ zur Trennung der unären Repräsentation. Eine partielle Funktion $f\colon \mathbb{N}^k \to \mathbb{N}$ heißt *berechenbar mit einer TM*, wenn eine TM M über dem Alphabet $\{\sqcap\}$ existiert, sodass f M-berechenbar ist.

Beachten Sie, dass die in Definition 4.6 definierte partielle Funktion undefiniert ist, wenn die TM M nicht hält.

4.3 Registermaschinen

Eine *Registermaschine* (abgekürzt *RM*) ist eine Maschine, die eine endliche Anzahl von Registern, $x_1, \ldots, x_n$ besitzt. Die Register enthalten beliebig große natürliche Zahlen. Eine RM führt ein Programm P aus, dessen Befehle an eine stark vereinfachte imperative Sprache erinnern. Es gibt verschiedene, äquivalente Möglichkeiten, die Instruktionen einer RM zu wählen. Wir werden als Programme so genannte while-Programme verwenden. Diese Programme verwenden im Befehlssatz neben einfachen Zuweisungen auch bedingte Schleifenaufrufe, also while-*Schleifen*. Eine andere Möglichkeit wären etwa goto-Programme. In goto-Programmen werden die Befehle durchnummeriert und es kommen zu den einfachen Zuweisungen bedingte Sprunganweisungen, goto-*Befehle* hinzu [7, 13]. Da wir uns in der Folge auf while-Programme beschränken, sprechen wir der Einfachheit halber schlicht von Programmen.

Definition 4.7 (Registermaschine)**.** Eine *Registermaschine* R ist ein Paar

$$R = ((x_i)_{1 \leqslant i \leqslant n}, P) \ ,$$

sodass $(x_i)_{1 \leqslant i \leqslant n}$ eine Sequenz von n *Registern* x_i ist und P ein Programm.

Programme sind endliche Folgen von Befehlen und sind induktiv definiert:

1. Für jedes Register x_i sind die folgenden Instruktionen sowohl Befehle wie Programme:

$$x_i := x_i + 1 \qquad x_i := x_i - 1 \ ,$$

2. und wenn P_1, P_2 Programme sind, dann ist

$$P_1; P_2 \ ,$$

ein Programm und

$$\mathsf{while}\ x_i \neq 0\ \mathsf{do}\ P_1\ \mathsf{end}\ ,$$

ist sowohl ein Befehl als auch ein Programm.

Wir beschreiben die Semantik eines Programms informell. Sei $R = ((x_i)_{1 \leqslant i \leqslant n}, P)$ eine RM, dann bedeuten die Befehle

$$x_i := x_i + 1 \qquad x_i := x_i - 1 \ ,$$

dass der Inhalt des Register x_i entweder um 1 erhöht oder vermindert wird. Das Programm $P_1; P_2$ bedeutet, dass zunächst das Programm S_1 und dann das Programm P_2 ausgeführt wird. Schließlich bedeutet der Befehl (und das Programm)

$$\mathsf{while}\ x_i \neq 0\ \mathsf{do}\ P_1\ \mathsf{end}\ ,$$

dass der Schleifenrumpf P_1 solange ausgeführt werden soll bis die Bedingung $x_i \neq 0$ falsch wird. Das Programmende eines Programms ist erreicht, wenn kein nächster auszuführender Befehl existiert.

Definition 4.8 (Berechenbarkeit mit einer RM)**.** Sei $R = ((x_i)_{1 \leqslant i \leqslant n}, P)$ eine Registermaschine. Eine partielle Funktion $f\colon \mathbb{N}^k \to \mathbb{N}$, heißt R-berechenbar, wenn gilt

$$f(n_1, \ldots, n_k) = m \quad \text{gdw.} \quad \begin{array}{l} R \text{ mit } n_i \text{ in den Registern } x_i \text{ für } 1 \leqslant i \leqslant k \text{ startet und} \\ \text{die Programmausführung mit } n_i \text{ in den Registern } x_i \\ \text{für } 1 \leqslant i \leqslant k \text{ und } m \text{ im Register } x_{k+1} \text{ abbricht.} \end{array}$$

Eine partielle Funktion $f\colon \mathbb{N}^k \to \mathbb{N}$ heißt *berechenbar auf einer RM*, wenn eine RM R existiert, sodass f R-berechenbar ist.

Beachten Sie, dass die in Definition 4.8 definierte partielle Funktion undefiniert ist, wenn die RM R nicht hält.

Jede Funktion $f\colon \mathbb{N}^k \to \mathbb{N}$, die mit einer TM berechenbar ist, ist auch mit einer RM berechenbar und umgekehrt. Für den Beweis des folgenden Satzes wird auf [7] verwiesen.

Satz 4.2. *Jede partielle Funktion $f\colon \mathbb{N}^k \to \mathbb{N}$, die berechenbar auf einer RM ist, ist auf einer TM berechenbar und umgekehrt.*

4.4 Zusammenfassung

Alan Turing (1912–1954) schlug 1936 die Turingmaschine als allgemeines Berechnungsmodell vor. Das Ziel war auf möglichst intuitive Weise zu klären, was eigentlich einen „Algorithmus“ oder eine Berechnung ausmacht. In den 1930er Jahren war dies eine Frage, die eine Reihe von hochkarätigen Forschern wie etwa Alonzo Church (1903–1995), Kurt Gödel (1906–1978), Stephen Kleene (1909–1994) oder John von Neumann (1903–1957) zu lösen versuchten. Turing hatte zum Ziel, die Grenze zwischen dem, was ein Computer berechnen kann, und dem, was er nicht berechnen kann, genau zu beschreiben. Seine Schlussfolgerungen treffen nicht nur auf seine abstrakten Turingmaschinen zu, sondern auch auf die heutigen realen Maschinen. Beispielsweise seien Grenzen der Berechenbarkeit genannt, das sind Entscheidungsprobleme, die rein prinzipiell von einer Maschine nicht gelöst werden können. Interessant ist, dass diese Untersuchungen über die Schranken der Berechenbarkeit zu einer Zeit entwickelt wurden in der die ersten Vorfahren moderner Rechner noch gar nicht gebaut waren. Die von Konrad Zuse (1910–1995) entwickelte Z3 wurde 1941 fertig gestellt und der von John Atanasoff (1903–1995) und Clifford Berry (1918–1963) entwickelte Atanasoff-Berry-Computer (ABC) wird auf 1939 datiert.

Im Jahr 1969 führte Stephen Cook (1939–) Turings Untersuchung mit der Frage fort, was berechnet werden kann und was nicht, indem er untersuchte, welche Probleme *effektiv*, also mit vertretbarem Aufwand, algorithmisch zu lösen sind. Die Klasse der manchmal als *nicht handhabbar* (*intractable*) betrachteten Probleme werden als NP-hart bezeichnet. Diese Begriffsbestimmung geht auf Cook zurück. Für diese Arbeiten ist Cook 1982 mit dem *Turing Award* ausgezeichnet worden.

Es ist sehr unwahrscheinlich, dass die exponentielle Steigerung der Rechengeschwindigkeit, die bei der Computerhardware erzielt worden ist („Moores Gesetz“), sich bemerkenswert auf unsere Fähigkeit auswirken wird, umfangreiche Beispiele solcher nicht handhabbaren Probleme berechnen zu können. Allerdings haben die Forschungen und Erkenntnisse der letzten Jahre gezeigt, dass NP-harte Probleme keineswegs „nicht handhabbar“ sind. Die Frage, ob eine gegebene aussagenlogische Formel erfüllbar ist, wäre ein solches Problem. Wie schon in Kapitel 1 erwähnt wurden in den letzten Jahren (beziehungsweise Jahrzehnten) sehr mächtige automatische Techniken entwickelt, um dieses Problem (fast immer) effizient lösen zu können [11].

5

Einführung in die Programmverifikation

In diesem Kapitel werden wir kurz auf Prinzipien der Analyse von Programmen eingehen und die Begriffe *Verifikation* und *Validierung* von Software klären (Abschnitt 5.1). Darüber hinaus werden wir die Verifikation nach Hoare in Abschnitt 5.2 behandeln. Schließlich diskutieren wir in Abschnitt 5.3 den historischen Kontext.

5.1 Prinzipien der Analyse von Programmen

Grundsätzlich besteht der Wunsch nach fehlerfreier Software. Die Praxis zeigt jedoch, dass dies bisher als nicht realisierbar angesehen werden muss. So ist bei einem größeren Softwareprojekt, wie etwa der Entwicklung eines Betriebssystems mit mehreren Millionen Zeilen Code, auch davon auszugehen, dass wiederum Millionen von Fehlern enthalten sein werden. *Verifikation* dient nun zum Nachweis, dass die Spezifikation eines Programms auch korrekt implementiert wurde. Nur wenn so sichergestellt ist, dass die Implementierung nicht von der Spezifikation abweicht, wenn das Programm also verifiziert ist, können wir davon ausgehen, dass das Programm wirklich fehlerfrei ist. Eine unvollständige Methode der Verifikation stellt das Testen von Software dar. Sehr viele Fehler lassen sich durch Testen finden, aber das Testen von Programmen kann nur die Fehler erkennen, auf die auch tatsächlich getestet wird. Dahingegen versuchen formale Methoden der Verifikation, die Korrektheit eines Programmes zu zeigen. Die Verifikation muss von der *Validierung* abgegrenzt werden. Letztere überprüft nicht, ob das Programm die Spezifikation korrekt implementiert, sonder ob die Spezifikation unseren Anforderungen entspricht. Kurz gefasst kann man sagen, dass die Verifikation überprüft, ob wir die Dinge richtig tun, wohingegen die Validierung überprüft, ob wir das Richtige tun.

Heutzutage werden in der Verifikation immer häufiger formale Methoden angewandt. Diese Methoden erlauben es die Verifikation frühzeitig in den Entwicklungsprozess einzubinden. Das ist wichtig, da Fehler umso leichter ausgebessert werden können, je früher diese erkannt

werden. Formale Methoden erlauben es auch die Verifikationstechniken effizient zu gestalten. Das kann bis zur Automatisierung der Verifikation führen. So kann erreicht werden, dass die Verifikation eines Programms zeitlich abgekürzt wird. Formale Methoden werden mittlerweile auch verwendet, um geeignete Teststrategien beziehungsweise Testmengen zu finden.

Im nächsten Abschnitt werden wir die Verifikation nach Hoare näher betrachten. Es ist wichtig darauf hinzuweisen, dass die Verifikation nach Hoare nicht vollständig automatisiert werden kann, zumindest wenn der Quellcode übliche Programmkonstrukte, wie etwa Prozeduren verwendet. Es ist in modernen Softwarepaketen undenkbar, den Code monolithisch zu entwerfen. Somit stößt die Anwendbarkeit dieser Methode schnell an ihre Grenzen. Deshalb spielt der Hoare-Kalkül in der Praxis kaum mehr eine Rolle. Andererseits bietet der Hoare-Kalkül eine gute Einführung in das Gebiet der Verifikation und die entsprechenden Arbeiten von Hoare gehören zu den meistzitierten Arbeiten in der Informatik.

Ein anderes Beispiel einer formalen Methode, die zur Verifikation verwendet werden kann, ist das *Model Checking*. Model Checking ist eine formale Methode, die gegeben eine endliche Beschreibung eines Systems (also nicht nur eines Programmes) und eine formale Eigenschaft, systematisch nachprüft, ob das System die Eigenschaft hat. Model Checking ist eine automatische Methode [3].

Eine wichtige Begriffsabgrenzung in der Verifikation ist der Unterschied zwischen *totaler* und *partieller* Korrektheit. Totale Korrektheit schließt die Termination des untersuchten Programmes mit ein. Partielle Korrektheit überprüft zwar, dass der Quelltext korrekte Ergebnisse liefert, aber geht von der Termination der untersuchten Programme aus.

5.2 Verifikation nach Hoare

Um interessante Eigenschaften von Programmen ausdrücken zu können, reichen die Möglichkeiten der Aussagenlogik nicht aus. Wir müssen unseren logischen Formalismus erweitern. Die Grundlage der Verifikation nach Hoare sind *prädikatenlogische* Ausdrücke. Eine Einführung in die Prädikatenlogik geht über diese Vorlesung hinaus. Wir werden uns also hier auf eine einfache Teilklasse der Prädikatenlogik beschränken: Wir betrachten *atomare Formeln* und deren Abschluss unter aussagenlogischen Junktoren.[1] Für die vollständige Definition von prädikatenlogischen Formeln wird auf die Vorlesung „Logic in Computer Science“ beziehungsweise auf [10, 6] verwiesen. Die wichtigste Erweiterung sind so genannte *Prädikatensymbole*, die wir anhand eines simplen Beispiels motivieren.

[1] Dies ist eine sehr starke Vereinfachung. In der vollständigen Definition werden den hier betrachteten aussagenlogischen Junktoren auch noch Quantoren ($\forall$, $\exists$) als logische Symbole zur Seite gestellt [10, 6]. Erst dann kann man eigentlich von prädikatenlogischen Formeln sprechen.

Beispiel 5.1. Angenommen wir haben die Konstante 7 und das Prädikatensymbol ist_prim in unserer Sprache, dann können wir ist_prim(7) schreiben, um auszudrücken, dass 7 eine Primzahl ist.

Prädikatensymbole erlauben es uns, über Elemente einer Menge, im Beispiel die natürlichen Zahlen, Aussagen zu treffen. Um Elemente in dieser Menge zu referenzieren, verwendet man *Terme*. Terme haben wir schon in Definition 2.16 eingeführt. Außerdem wurden in Kapitel 2 Gleichungen $s = t$ betrachtet. Hier stehen s und t für Terme. Das Gleichheitszeichen können wir als Prädikatensymbol auffassen, denn die Gleichung drückt eine Beziehung zwischen den Objekten aus, auf die, die Terme s und t verweisen. Wegen seiner Bedeutung wird die Gleichheit hier aber besonders hervorgehoben.

Sei nun P ein Prädikatensymbol und $t_1, \ldots, t_n$ Terme über einer geeignet gewählten Signatur. Der Ausdruck $P(t_1, \ldots, t_n)$ sowie die Gleichung $t_1 = t_2$ wird *Atom* oder *atomare Formel* genannt. Mit Hilfe von atomaren Formeln als Basis können nun *Zusicherungen* definiert werden.

Definition 5.1 (Zusicherungen)**.** Wir definieren *Zusicherungen* induktiv.

1. Atome sind Zusicherungen.
2. Wenn A und B Zusicherungen sind, dann sind $\neg A$, $(A \wedge B)$, $(A \vee B)$ und $(A \to B)$ auch Zusicherungen.

Der Einfachheit halber nennen wir Zusicherungen manchmal auch *Formeln*, da keine Verwechslungsgefahr mit aussagenlogischen Formeln besteht.

Zusicherungen bilden den logischen Formalismus, der es uns erlaubt, den Zustand eines Programmes zu beschreiben. Zusicherungen sind rein syntaktisch definiert, das heißt wir müssen diesen Formeln eine Bedeutung zuordnen. Dazu verwendet man *Interpretationen*, die wir vereinfacht als Verallgemeinerungen von Algebren verstehen können. Eine Interpretation $\mathcal{I}$ gibt an, wie die Symbole einer Formel zu verstehen sind. Etwa würden wir in Beispiel 5.1 das Symbol ist_prim so interpretieren wollen, dass das Atom ist_prim(n) wahr wird gdw. wenn n tatsächlich eine Primzahl ist. In ähnlicher Weise werden Gleichungen $t_1 = t_2$ über einer gegeben Interpretation $\mathcal{I}$ wahr genannt werden gdw. wenn die Terme t_1 und t_2 in $\mathcal{I}$ als gleich angesehen werden. Ist die Wahrheit von Atomen in $\mathcal{I}$ definiert, ist es leicht die Wahrheit einer beliebigen Zusicherung über $\mathcal{I}$ zu definieren: Wie in Kapitel 1 verwenden wir die Wahrheitstabellen für die aussagenlogischen Junktoren $\neg$, $\wedge$, $\vee$ und $\to$, um die Wahrheit einer zusammengesetzten Formel zu überprüfen. Wenn eine Formel F in einer Interpretation $\mathcal{I}$ wahr ist, schreibt man $\mathcal{I} \models F$. Schließlich wollen wir noch ausdrücken können, dass eine bestimmte Zusicherung B aus einer Prämisse A folgt. Dazu definiert man $A \models B$ gdw. für alle Interpretationen $\mathcal{I}$, sodass $\mathcal{I} \models A$ auch $\mathcal{I} \models B$ gilt. Die Relation

a) $\dfrac{}{\{Q\{x \mapsto t\}\}\ x := t\ \{Q\}}$ b) $\dfrac{\{Q'\}\ P\{R'\}}{\{Q\}\ P\ \{R\}}\ Q \models Q', R' \models R$

c) $\dfrac{\{Q\}\ P_1\ \{R\} \quad \{R\}\ P_2\ \{S\}}{\{Q\}\ P_1; P_2\ \{S\}}$ d) $\dfrac{\{I \wedge B\}\ P\ \{I\}}{\{I\}\ \mathsf{while}\ B\ \mathsf{do}\ P\ \mathsf{end}\ \{I \wedge \neg B\}}$

Abbildung 5.1: Regeln nach Tony Hoare

$\models$ wird *Konsequenzrelation* genannt und erweitert die in Definition 1.3 eingeführte Konsequenzrelation für aussagenlogische Formeln. Wir motivieren die Konsequenzrelation anhand des nächsten Beispiels.

Beispiel 5.2. Seien $x < 5$ und $x + 1 < 7$ Atome und damit Zusicherungen. Dann gilt intuitiv die folgende Folgerung für alle natürlichen Zahlen, die wir für x einsetzen können:

$$x < 5 \models x + 1 < 7\ .$$

In Definition 2.17 haben wir Substitutionen eingeführt. Substitutionen sind auf Termen definiert, aber es ist intuitiv einleuchtend, wie eine Substitution $\{x \mapsto t\}$ auf eine Formel F angewandt werden soll: Wenn x in F vorkommt werden alle Vorkommnisse von x durch den Term t ersetzt. Für diesen Prozess schreiben wir einfach $F\{x \mapsto t\}$. In Korrektheitsbeweisen werden Ausdrücke die in Programmen auftreten, etwa die rechten Seiten von Zuweisungen, als Terme repräsentiert.

Mit diesem Handwerkszeug wenden wir uns der Verifikation nach Hoare zu. Wir nehmen die in Definition 4.7 definierten while-Programme als Grundlage, um den Hoare Kalkül erklären zu können.[2]

Definition 5.2. Sei P ein while-Programms und seien Q und R Formeln. Ein *Hoare-Tripel* ist wie folgt definiert:

$$\{Q\}\ P\ \{R\}\ ,$$

wobei Q und R Zusicherungen sind. Q wird *Vorbedingung* und R wird *Nachbedingung* genannt.

Definition 5.3 (Hoare-Kalkül)**.** Seien Q, R Zusicherungen und P ein while-Programm. P heißt *korrekt in Bezug auf Q und R* wenn

$$\{Q\}\ P\ \{R\}\ ,$$

[2] Die hier betrachteten Hoare-Regeln bilden einen Teil der üblicherweise in der Literatur betrachteten Regeln, da sie auf while-Programme spezialisiert sind [8].

mit Hilfe der in Abbildung 5.1 dargestellten Regeln abgeleitet werden kann. Die Hoare-Regeln werden üblicherweise von unten nach oben gelesen: Das Problem die Korrektheit des Programmes zu zeigen, wird sukzessive in kleinere Teile aufgespaltet.

Wir nennen P *partiell korrekt* für eine Spezifikation S, wenn P korrekt ist in Bezug auf Zusicherungen Q und R die der Spezifikation S entsprechen.

Wir motivieren die in Abbildung 5.1 dargestellten Regeln kurz. Die erste Regel a), ist ein Axiom des Kalküls, da keine Vorbedingungen erfüllt sein müssen. Das Axiom wird verwendet, um Zuweisungsbefehle korrekt in den Zusicherungen abbilden zu können. Die Regel b) dient dazu, die Vorbedingung abzuschwächen (Nebenbedingung $Q \models Q'$) beziehungsweise die Nachbedingung zu verstärken (Nebenbedingung $R' \models R$). Um diese Regel anwenden zu können, müssen die Nebenbedingungen erfüllt sein. Die Regel c) dient dazu die Korrektheit der einzelnen Teile eines Programms separat zu beweisen. Die vierte Regel d) wird auch while-Regel genannt, da sie die Korrektheit von while-Schleifen überprüft. Wir bezeichnen die Formel I als *Schleifeninvariante* oder kurz *Invariante*. Bei der Anwendung dieser Regel setzt man die Termination des Schleifenrumpfes voraus.

Definition 5.4. Angenommen P ist ein partiell korrektes Programm für eine gegebene Spezifikation. Wenn es noch gelingt die Termination von P nachzuweisen, dann ist auch der Beweis der *totalen Korrektheit* gelungen.

5.3 Zusammenfassung

Schon in den Anfängen der Programmierung wurde erkannt, dass es wünschenswert wäre, die Korrektheit eines Programmes formal verifizieren zu können und sich nur auf Fehlersuche beschränken zu müssen. Etwa hatte sich schon Alan Turing dieser Fragestellung gewidmet. Robert Floyd (1963–2001) hat diese Ideen weitergeführt. Auf die Arbeiten von Floyd aufbauend, hat Charles Antony Richard (oder Tony) Hoare (1934–), den hier vorgestellten Hoare-Kalkül entwickelt. Hoare wurde 1980 der *Turing Award* für seine Arbeiten zu Definition und Design von Programmiersprachen verliehen.

Die Erkenntnis, dass der Hoare-Kalkül für komplexere Programmkonstrukte ungeeignet ist, geht auf Edmund M. Clarke (1945–) zurück, der gemeinsam mit Allen Emerson (1954–) als Erster das Model Checking als formale Verifikationsmethode von endlichen Systemen vorgeschlagen hat. Clarke, Emerson und Joseph Sifakis (1946–) wurden gemeinsam für ihre Arbeiten zu Model Checking im Jahr 2007 mit dem *Turing Award* ausgezeichnet.

A

Beweismethoden

A.1 Deduktive Beweise

Ein *deduktiver Beweis* besteht aus einer Folge von Aussagen, die von einer *Hypothese* zu einer *Konklusion* führen. Jeder Beweisschritt muss sich nach einer akzeptierten logischen Regel aus den gegebenen Fakten oder aus vorangegangenen Aussagen ergeben. In Definition 1.5 in Kapitel 1 haben wir formale, deduktive Beweise kennengelernt.

Der Aussage, dass die Folge der Beweisschritte von einer Hypothese H zu einer Konklusion K führt, entspricht der Satz:

$$\text{Wenn } H\text{, dann } K.$$

Wir betrachten einen Satz dieser Form.

Satz A.1. *Wenn $x \geqslant 4$, dann $2^x \geqslant x^2$.*

Beweisskizze. Für $x = 4$ richtig: $2^4 \geqslant 4^2$. Für $x \geqslant 1$ gilt, dass sich die linke Seite verdoppelt, wenn x um 1 erhöht wird. Die rechte wächst hingegen nur mit $\frac{(x+1)^2}{x^2}$. Gilt nun $x \geqslant 4$, dann muss gelten $\frac{x+1}{x} \leqslant 1{,}25$, und somit $\frac{(x+1)^2}{x^2} \leqslant 1{,}5625 < 2$. □

Die gegebene Argumentation ist akkurat, jedoch informell. Um den Satz (oder besser das Sätzchen) formal beweisen zu können, benötigen wir vollständige Induktion. Wir geben einen Induktionsbeweis in Sektion A.3.1. Der folgende Satz folgt mittels einer stringenten Folge von Aussagen aus seiner Hypothese und Satz A.1.

Satz A.2. *Wenn x die Summe der Quadrate von 4 positiven ganzen Zahlen ist, dann $2^x \geqslant x^2$.*

Beweis. Wir listen die notwendige Folge von Aussagen tabellarisch auf.

$x = a^2 + b^2 + c^2 + d^2$	Hypothese	(A.1)
$a \geqslant 1, b \geqslant 1, c \geqslant 1, d \geqslant 1$	Hypothese	(A.2)
$a^2 \geqslant 1, b^2 \geqslant 1, c^2 \geqslant 1, d^2 \geqslant 1$	(A.2) und elementare Arithmetik	(A.3)
$x \geqslant 4$	folgt aus (A.1) und (A.3)	(A.4)
$2^x \geqslant x^2$	(A.4) und voriger Satz	(A.5)

□

A.1.1 Formen von „Wenn-dann“

„Wenn-dann“-Sätze können auch in anderen Formen auftreten. Beispiele hierzu sind:

- H impliziert K.
- H nur dann, wenn K.
- K, wenn H.
- Wenn H gilt, folgt daraus K.
- $H \to K$.

A.1.2 „Genau dann, wenn“-Sätze

Gelegentlich finden wir Aussagen der Form „A genau dann, wenn B“. Andere Varianten dieses Satzes sind etwa:

- A dann—und nur dann—wenn B.
- $A \approx B$, $A \leftrightarrow B$, $A \approx B$.

„Genau dann, wenn“ Aussagen werden bewiesen indem *zwei* Behauptungen gezeigt werden:

- A impliziert B und
- B impliziert A.

Beachten Sie bitte den Sprachgebrauch. Der Aussage S

A *genau dann, wenn* B,

entsprechen die beiden Aussagen „A *nur dann, wenn* B“ (also „A impliziert B“) und „A, *wenn* B“ („B impliziert A“). Abkürzend schreibt man manchmal für die Richtung der Aussage S von links nach rechts „⇒“ und für die Richtung von rechts nach links „⇐“.

A.2 Beweisformen

Bis jetzt haben wir vor allem die Struktur von Sätzen beziehungsweise Beweisen betrachtet. Im Folgenden gehen wir auf häufig bis sehr häufig auftretende Beweisprinzipien ein.

A.2.1 Reduktion auf Definitionen

Viele Aussagen folgen leicht oder sogar unmittelbar, sobald die in den Hypothesen verwendeten Begriffe in ihre Definitionen umgewandelt werden. Wir geben dazu ein einfaches Beispiel, das gleichzeitig wichtige Grundbegriffe der elementaren Mengenlehre einführt. Der folgende Beweis liefert auch das erste Beispiel eines Widerspruchsbeweises, ein oftmals sehr nützliches Beweisprinzip.

Satz A.3. *Sei S eine endliche Teilmenge einer unendlichen Menge U. T sei die Komplementärmenge von S in Bezug auf U. Dann ist T unendlich.*

Beweis. Laut Definition gilt $S \cup T = U$ und S, T disjunkt, also $|S| + |T| = |U|$, wobei $|S|$ die *Kardinalität* oder Mächtigkeit der Menge S angibt.

Da S endlich, existiert eine bestimmte natürliche Zahl n, sodass $|S| = n$. Andererseits, da U unendlich, existiert *kein* l, sodass $|U| = l$. Nun angenommen T ist endlich, dann existiert m, sodass $|T| = m$. Daraus folgt, dass $|U| = |S| + |T| = n + m$. Somit würde eine natürliche Zahl $l = n + m$ existieren, sodass die Kardinalität von U gleich l. Das steht jedoch im Widerspruch zur Annahme, dass U unendlich ist. Somit muss die Annahme, dass T endlich ist, falsch sein. Es folgt die Aussage des Satzes. □

A.2.2 Beweis in Bezug auf Mengen

Wir geben das folgende einfache Beispiel.

Satz A.4 (Distributivgesetz der Vereinigung)**.**

$$R \cup (S \cap T) = (R \cup S) \cap (R \cup T) \ .$$

Dieser Satz stellt de-facto einen „Genau dann, wenn“-Satz dar.

Beweisansatz. Wir beschreiben hier nur den Beweisansatz. Zunächst formulieren wir die Gleichung zu einem „Genau dann, wenn“-Satz um:

$$x \in R \cup (S \cap T) \text{ gdw } x \in (R \cup S) \cap (R \cup T).$$

Um diesen Satz zeigen zu können müssen wir nun nur noch die folgenden beiden Behauptungen zeigen.

1. $x \in R \cup (S \cap T)$ impliziert $x \in (R \cup S) \cap (R \cup T)$ *und*

2. $x \in (R \cup S) \cap (R \cup T)$ impliziert $x \in R \cup (S \cap T)$

Betrachten wir die folgende Implikation:

$$x \in (R \cup S) \cap (R \cup T) \quad \text{impliziert } x \in R \cup (S \cap T)\,. \tag{A.6}$$

Oft ist es vorteilhaft, statt einer zu beweisenden Implikation, wie der gerade angegebenen, ihre *Kontraposition* zu betrachten. Die Kontraposition von (A.6) ist dann:

$$x \notin R \cup (S \cap T) \quad \text{impliziert} \quad x \notin (R \cup S) \cap (R \cup T)\,.$$

□

Im Beweisansatz haben wir einen typischen Fall aufgelistet. Statt zu zeigen, dass die Aussage A die Aussage B impliziert, wird gezeigt, dass die Negation von B, die Negation von A impliziert. Gerade bei „Genau dann, wenn“ kann diese Umformulierung nützlich sein. Wir stellen den Sachverhalt schematisch dar:

$$\frac{\text{A impliziert B}}{\text{(nicht B) impliziert (nicht A)}} \qquad \frac{\text{(nicht B) impliziert (nicht A)}}{\text{A impliziert B}}$$

Diese Schemata bedeuten, dass die Aussagen „A impliziert B“ und „nicht B impliziert nicht A“ äquivalent sind, das heißt aus dem einen Satz folgt der andere und umgekehrt.

A.2.3 Widerspruchsbeweise

Widerspruchsbeweise sind von entscheidender Bedeutung; da wir bereits einen solchen Beweis angegeben haben, begnügen wir uns im folgenden mit der schematischen Darstellung des Beweisschemas. Im Schema steht $\perp$ für eine unerfüllbare Aussage, um den Widerspruch darzustellen.

Satz A.5. *S sei eine endliche Teilmenge einer unendlichen Menge U. T sei die Komplementärmenge von S in Bezug auf U. Dann ist T unendlich.*

Beweisansatz.

$$\cfrac{\cfrac{\text{Hypothese(n)} \quad \text{Negation der Konklusion}}{\perp}}{\text{Konklusion}}$$

□

A.2.4 Gegenbeispiele

Da Sätze *allgemeine* Aussagen behandeln, genügt es, die Aussage für bestimmte Werte zu widerlegen, um den ganzen Satz zu widerlegen. In dieser Situation haben wir dann ein *Gegenbeispiel* gefunden. Gegenbeispiele können auch verwendet werden, um allgemein gefasste Aussagen soweit zu präzisieren, dass sie dann als Satz gezeigt werden können.

A.3 Induktive Beweise

A.3.1 Induktive Beweise mit ganzen Zahlen

Zunächst behandeln wir Induktionsbeweise mit ganzen Zahlen. Induktionsbeweise sind immer dann erforderlich, wenn eine Aussage $S(n)$ für alle n gezeigt werden soll. In diesem Fall gehen wir wie folgt vor:

- Basis: Zu zeigen, dass S für Startwert gilt, etwa $n = 0$ oder $n = 1$.
- Induktionsschritt: Zu zeigen, dass wenn $S(n)$, dann gilt auch $S(n + 1)$.

Das zugrundeliegende Prinzip wird *Induktionsprinzip* genannt.

> **Induktionsprinzip**: *Wenn wir $S(i)$ bewiesen haben und beweisen können, dass $S(n)$ für alle $n \geqslant i$ $S(n + 1)$ impliziert, dann können wir daraus schließen, dass $S(n)$ für alle $n \geqslant i$ gilt.*

Wir verdeutlichen das Prinzip indem wir Satz A.1 formal beweisen.

Satz A.6. *Wenn $x \geqslant 4$, dann $2^x \geqslant x^2$.*

Beweis.

- Basis: Für $x = 4$ stimmt die Aussage $2^x \geqslant x^2$.
- Schritt: Zu zeigen ist $2^{x+1} \geqslant (x + 1)^2$ unter der Voraussetzung $2^x \geqslant x^2$, der Induktionshypothese. Dazu zeigen wir zunächst die folgende Hilfsüberlegung:

$$2x^2 \geqslant (x + 1)^2 \,. \tag{A.7}$$

  Da $x \geqslant 4$ gilt $x \geqslant 2 + \frac{1}{x}$ und damit auch $x^2 \geqslant 2x + 1$. Somit gilt:

$$2x^2 = x^2 + x^2 \geqslant x^2 + 2x + 1 = (x + 1)^2 \,.$$

Somit folgt (A.7). Schließlich zeigen wir $2^{x+1} \geqslant (x+1)^2$ wie folgt:

$$\begin{aligned} 2^{x+1} &= 2 \cdot 2^x \\ &\geqslant 2 \cdot x^2 \\ &\geqslant (x+1)^2 \ . \end{aligned}$$

Hier haben wir in der zweiten Zeile die Induktionshypothese und in der dritten Zeile die Hilfsüberlegung (A.7) angewandt.

□

Wenn wir kurz (und informell) Gebrauch von Quantoren machen, können wir das Prinzip der vollständigen Induktion auch als Schlussfigur anschreiben:

$$\frac{S(i) \quad \forall n \geqslant i \ (S(n) \to S(n+1))}{\forall n \geqslant i \ S(n)}$$

A.3.2 Allgemeinere Formen der Induktion

Das oben beschriebene Induktionsprinzip ist, in der angegeben Form, oft nicht ausreichend. Zwei *Erweiterungen* sind besonders nützlich. Zunächst müssen wir uns nicht auf einen Basisfall konzentrieren, sondern können mehrere Basisfälle verwenden. Zum Beispiel

$$S(i), S(i+1), \ldots, S(j) \ .$$

Im Weiteren können wir, um $S(n+1)$ zu beweisen, als Hypothesen alle Aussagen

$$S(i), S(i+1), \ldots, S(n) \ ,$$

verwenden. Darüberhinaus, wenn wir mehrere Basisfälle gezeigt haben, dann können wir im Beweis des Induktionsschrittes

$$n \geqslant j \ ,$$

annehmen. Zu beachten ist, dass diese *Erweiterungen* des Induktionsprinzips Erweiterungen in der Anwendbarkeit des Prinzips darstellen, aber der Beweisstärke der Induktion über natürlichen Zahlen nichts hinzufügen. Genauer gesagt folgen die oben erwähnten allgemeineren Formen aus der „einfachen“ Induktion.

A.3.3 Induktive Definitionen und Strukturelle Induktion

In der theoretischen Informatik sind wir weniger an Induktionen über natürliche Zahlen interessiert, sondern mehr an Induktionen über den (rekursiv definierten) Strukturen mit

denen wir ständig arbeiten, wie etwa Listen, Bäumen oder Ausdrücken. Wir beginnen mit zwei einfachen Beispielen von rekursiv definierten Strukturen, solche Definitionen werden auch als *induktive Definitionen* bezeichnet.

Definition A.1. Wir definieren *Bäume* induktiv:
Basis: Ein einzelner Knoten ist ein *Baum*; dieser Knoten ist die *Wurzel*.
Schritt: Wenn $T_1, T_2, \ldots, T_k$ Bäume sind, bilden wir einen neuen *Baum* wie folgt:

1. Man beginnt mit einem neuen Knoten N, der die Wurzel des Baumes darstellt.
2. Schließlich fügt man k Kanten von N zu den Wurzeln der T_i hinzu.

Definition A.2. *Ausdrücke*:
Basis: Jede Zahl und jeder Buchstabe ist ein *Ausdruck*.
Schritt: Wenn E, F Ausdrücke sind, dann sind auch $E + F$, $E \cdot F$ und (E) *Ausdrücke*.

Die Aussage $S(X)$ soll für alle Strukturen X, die durch eine bestimmte induktive beziehungsweise rekursive Definition gegeben sind, gezeigt werden. In diesem Fall gehen wir wie folgt vor:

- Basis: Zunächst beweisen wir $S(X)$ für die Basisstruktur(en) X der induktiven Definition.
- Schritt: Wähle Struktur Y, die rekursiv aus $Y_1, Y_2, \ldots, Y_k$ gebildet wird.

  Induktionshyptothese: $S(Y_1), S(Y_2), \ldots, S(Y_k)$ seien wahr.

  Mit Hilfe der Induktionshyptothese wird nun $S(Y)$ gezeigt.

Das zugrundeliegende Induktionsprinzip wird *Strukturelle Induktion* genannt.

> **Strukturelle Induktion**: *Wenn wir $S(X)$ für alle Basisstrukturen X der induktiven Definition beweisen und beweisen können, dass, wenn Y rekursiv mittels $Y_1, Y_2, \ldots, Y_k$ gebildet wurde und $S(Y_1), S(Y_2), \ldots, S(Y_k)$ für die Teilstrukturen $Y_1, Y_2, \ldots, Y_k$ angenommen wird, dann $S(Y)$ folgt, dann können wir daraus schließen dass $S(X)$ für alle nach der induktiven Definition gebildeten Strukturen X gilt.*

Wir zeigen als Beispiel den folgenden Satz mit struktureller Induktion über Bäume.

Satz A.7. *Jeder Baum besitzt genau einen Knoten mehr als Kanten.*

Beweis. Die Aussage $S(T)$ lautet: „Wenn T ein Baum ist und n Knoten und e Kanten hat, dann gilt $n = e + 1$.“

- Basis: Trivialerweise gilt $n = e + 1$, wenn T nur aus einem Knoten besteht.

– Schritt: Angenommen T habe $T_1, \ldots, T_k$ als direkte Teilbäume und mit Induktionshypothese folgt $S(T_1), \ldots, S(T_k)$.

Seien nun $n_1, \ldots, n_k$ die Anzahlen der Knoten von $T_1, \ldots, T_k$. Und seien $e_1, \ldots, e_k$ die Anzahlen der Kanten von $T_1, \ldots, T_k$.

Für alle $i \in [1, k]$ gilt: $n_i = e_i + 1$. Somit

$$n = 1 + n_1 + \cdots + n_k =$$
$$= 1 + (e_1 + 1) + \cdots + (e_k + 1) = k + e_1 + \cdots e_k + 1 = e + 1 \; .$$

Somit haben wir auch den Induktionschritt gezeigt und damit den Satz vollständig bewiesen.

□

Literaturverzeichnis

[1] F. Baader and T. Nipkow. *Term Rewriting and All That.* Cambridge University Press, 1998.

[2] H. Barendregt. *The Lambda Calculus: Its Syntax and Semantics.* Studies in Logic and the Foundations of Mathematics. Elsevier, zweite edition, 1985.

[3] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking.* MIT Press, 2000.

[4] F. Dressler and S. Podlipnig. *Einführung in die technische Informatik.* Institut für Informatik, 2011. Foliensatz.

[5] A. Dür. *Diskrete Mathematik 1.* Institut für Mathematik, 2011. Skriptum zur Vorlesung, 4. Auflage.

[6] H.-D. Ebbinghaus, J. Flum, and W. Thomas. *Einführung in die mathematische Logik.* Hochschul Taschenbuch. Spektrum Akademischer Verlag, 5te auflage edition, 2007.

[7] K. Erk and L. Priese. *Theoretische Informatik: Eine umfassende Einführung.* Springer Verlag, 3te auflage edition, 2008.

[8] J. L. Hein. *Discrete Structures, Logic, and Computability.* Jones and Bartlett Publishers, 3te auflage edition, 2010.

[9] J. E. Hopcroft, R. Motwani, and J. D. Ullman. *Introduction to Automata Theory, Languages, and Computation.* Addison-Wesley, 2001. 2te Auflage.

[10] M. Huth and M. Ryan. *Logic in Computer Science: Modelling and Reasoning about Systems.* Cambridge University Press, 2te auflage edition, 2004.

[11] D. Kroening and O. Strichman. *Decision Procedures – An Algorithmic Point of View.* Springer Verlag, 2008.

[12] G. Moser. *Diskrete Mathematik 2.* Institut für Informatik, 2011. Skriptum zur Vorlesung, 4. Auflage.

[13] M. Schaper. Programming turing machines. Master's thesis, Institute of Computer Science, 2011. Bachelor Thesis.

[14] S. Singh. *Fermats letzter Satz: Die abenteuerliche Geschichte eines mathematischen Rätsels.* Deutscher Taschenbuch Verlag, 2000.