LEHRBUCH

Kurt-Ulrich Witt
Martin Eric Müller

Algorithmische Informationstheorie

Berechenbarkeit und Komplexität verstehen

Springer Spektrum

Algorithmische Informationstheorie

Kurt-Ulrich Witt · Martin Eric Müller

Algorithmische Informationstheorie

Berechenbarkeit und Komplexität verstehen

Springer Spektrum

Kurt-Ulrich Witt
Fachbereich Informatik
Hochschule Bonn-Rhein-Sieg
Sankt Augustin, Deutschland

Martin Eric Müller
Fachbereich Informatik
Hochschule Bonn-Rhein-Sieg
Sankt Augustin, Deutschland

ISBN 978-3-662-61693-2 ISBN 978-3-662-61694-9 (eBook)
https://doi.org/10.1007/978-3-662-61694-9

Die Deutsche Nationalbibliothek verzeichnet diese Publikation in der Deutschen Nationalbibliografie; detaillierte bibliografische Daten sind im Internet über http://dnb.d-nb.de abrufbar.

Planung und Lektorat: Iris Ruhmann
Springer Spektrum ist ein Imprint der eingetragenen Gesellschaft Springer-Verlag GmbH, DE und ist ein Teil von Springer Nature.
Die Anschrift der Gesellschaft ist: Heidelberger Platz 3, 14197 Berlin, Germany

Vorwort

Unser *Informationszeitalter* mit der zunehmenden *Digitalisierung* von Informationen und deren Verarbeitung und Verbreitung verlangen *effektive* und *effiziente* Konzepte, Methoden, Techniken und Verfahren für die Speicherung, Übertragung und Analyse von Daten. *Effektivität* bedeutet, dass *Algorithmen* zur Verfügung stehen, mit denen diese Verfahren implementiert werden können. *Effizienz* bedeutet, dass die Ausführungszeit der Algorithmen möglichst optimal ist und für die Speicherung und Übertragung der Daten möglichst wenig Platz benötigt wird.

Dieses Lehrbuch gibt eine *Einführung in die theoretischen Grundlagen der Algorithmischen Informationstheorie*, in der die oben genannten Aspekte *Algorithmen*, *Laufzeit-Komplexität* und *Informationskomplexität* eine wesentliche Rolle spielen.

Zumeist werden Kernthemen zu *Grundlagen der Theoretischen Informatik* und zur *Berechenbarkeits- und Komplexitätstheorie* sowie weiterführende Veranstaltungen zur *Algorithmischen Informationstheorie* in den Curricula und in der Literatur getrennt betrachtet. All diese Aspekte sind jedoch eng miteinander verknüpft, und dass man sie häufig getrennt betrachtet, ist lediglich der Tradition geschuldet. Wir sehen es jedoch als sinnvoll an, den inhaltlichen Zusammenhang auch vereinheitlicht im textuellen Zusammenhang darzustellen.

Dazu ist zwar für einführende Veranstaltungen ein unter Umständen etwas höherer Anspruch an die formal präzise mathematische Darstellung notwendig – umgekehrt ermöglicht gerade das in Verbindung mit den Konzepten der algorithmischen Informationstheorie ein der Zeit angepasstes Verständnis der theoretischen Konzepte.

Insofern richten wir uns insbesondere an *Studierende in mathematisch-theoretisch ausgerichteten Studiengängen*; sie werden durch das Buch auf das weitere Studium und Tätigkeiten in Forschung und Entwicklung vorbereitet.

Wir integrieren bisher getrennt behandelte, aber zusammengehörende Inhalte in konsistenten Darstellungen, wobei wir die Konzepte aus den verschiedenen Blickrichtungen zur Vereinheitlichung, Vereinfachung und Verdeutlichung wechselseitig nutzen. Wir möchten zum Ausdruck bringen, dass *Informatik* die Wissenschaft von der *algorithmischen Informationsverarbeitung* ist, die auf theoretischen Konzepten für Algorithmen und deren Komplexität sowie auf theoretischen Konzepten für die Beschreibung der Komplexität von Informationsdarstellungen beruhen.

Das Buch richtet sich an *Studierende in Mathematik- und Informatik-Studiengängen*. Es ist als Begleitlektüre zu entsprechenden Lehrveranstaltungen an Hochschulen aller Art und insbesondere zum *Selbststudium* geeignet. Jedes Kapitel be-

ginnt mit einer seinen Inhalt motivierenden Einleitung. Zusammenfassungen am Ende von Kapiteln bieten Gelegenheit, den Stoff zu reflektieren. Die meisten Beweise sind vergleichsweise ausführlich und mit Querverweisen versehen, aus denen Zusammenhänge hervorgehen. Eingestreut sind Beispiele und Aufgaben, deren Bearbeitung zur Festigung des Wissens und zum Einüben der dargestellten Methoden und Verfahren dient. Zu fast allen Aufgaben sind am Ende des Buches oder im Text Musterlösungen aufgeführt. Die Aufgaben und Lösungen sind als integraler Bestandteil des Buches konzipiert.

Das Schreiben und Publizieren eines solchen Buches ist nicht möglich ohne die Hilfe und Unterstützung vieler Personen, von denen wir an dieser Stelle allerdings nur einige nennen können: Als erstes möchten wir die Autoren der Publikationen erwähnen, die im Literaturverzeichnis aufgeführt sind. Alle dort aufgeführten Werke haben wir für den einen oder anderen Aspekt verwendet. Wir können sie allesamt für weitere ergänzende Studien empfehlen. Zu Dank verpflichtet sind wir Linda Koine, Oliver Lanzerath und Dominic Witt für ihre Hinweise, die an vielen Stellen zu einer präziseren und lesbareren Darstellung geführt haben. Trotz dieser Unterstützung wird das Buch Fehler und Unzulänglichkeiten enthalten. Diese verantworten wir allein – für Hinweise zu ihrer Beseitigung sind wir dankbar.

Die Publikation eines Buches ist auch nicht möglich ohne einen Verlag, der es herausgibt. Wir danken dem Springer-Verlag für die Bereitschaft zur Publikation und insbesondere Frau Ruhmann und Frau Groth für ihre Ermunterung zur und ihre Unterstützung bei der Publikation des Buches.

Bedburg und Sankt Augustin, im März 2020

K.-U. Witt und M. E. Müller

Inhaltsverzeichnis

Kapitel 1

Einführung und Übersicht

Endliche Bitfolgen, also Zeichenfolgen, die aus Nullen und Einsen bestehen, sind *die* Datenstruktur zur logischen Repräsentation von Informationen in der Informations- und Kommunikationstechnologie. Auf der Anwendungsebene können sie – geeignet decodiert – z. B. Texte, Grafiken, Bilder, Musik oder Videos darstellen; auf der technischen Ebene wird ihre Speicherung und Übertragung elektrotechnisch realisiert. Wir werden fast ausschließlich die logische Ebene, d. h. Bitfolgen betrachten.

Die Datenübertragung spielte schon vor Beginn des Computerzeitalters Ende der fünfziger und Anfang der sechziger Jahre des letzten Jahrhunderts eine wichtige Rolle in der Telekommunikation, z. B. bei der Übertragung von Sprache beim Telefonieren sowie bei der Übertragung von Radio- und Fernsehsignalen. Dabei wurden zu dieser Zeit die Signale *analog* übertragen, d. h., ein elektrisches Signal stellte keinen eindeutigen Wert dar, sondern im Prinzip unendlich viele Werte aus einem Intervall.

Im Hinblick auf die Qualität und die Effizienz der Datenübertragung standen u. a. die folgenden zwei wichtigen Problemstellungen zur Lösung an:

- **Fehlertoleranz:** Wie kann auf der Empfängerseite festgestellt werden, ob Signale bei der Übertragung verfälscht wurden? In der Nachrichtentechnik werden solche Phänomene als **Rauschen** bezeichnet. In der digitalen Datenübertragung, bei der jedes Signal eindeutig einem von zwei Werten, dargestellt durch 0 und 1, zugeordnet werden kann, bedeutet Rauschen das Auftreten von Bitfehlern, d. h. das Kippen von Bits: Eine gesendete Eins wird zur Null oder eine gesendete Null wird zur Eins verfälscht. Auch das Zerstören von Bits, z. B. durch Kratzer auf einer DVD, stellt eine Fehlersituation dar.

- **Effizienz:** Wie viele Bits sind ausreichend, um eine Information zu übertragen oder zu speichern? Bei dieser Frage geht es um Methoden und Verfahren zur **Datenkompression**: Wie weit kann eine Bitfolge ohne Informationsverlust komprimiert werden?

K.-U. Witt und M. E. Müller, *Algorithmische Informationstheorie*,
https://doi.org/10.1007/978-3-662-61694-9_1

1.1 Informationskomplexität

Mit der Lösung dieser beiden Problemstellungen befassen sich die *Codierungstheorie* und die *Informationstheorie*, als deren Pioniere Richard Hamming[1] und Claude Shannon[2] gelten.

Mithilfe mathematischer Konzepte und Methoden aus der *Linearen Algebra und der Zahlentheorie* können fehlertolerante Codierungsverfahren entwickelt werden. Solche Verfahren erkennen Fehler und können diese möglicherweise sogar korrigieren, d. h. auf der Empfängerseite kann erkannt werden, ob Bits verfälscht wurden, bzw. es kann sogar erkannt werden, welche Bits gekippt sind, die dann korrigiert werden können. Notwendig dafür ist, dass einer (zu sendenden) Bitfolge sogenannte *redundante* Bits hinzugefügt werden, die dann die Fehlertoleranz ermöglichen. Das bedeutet, dass der eigentlich zu übertragende Datenstrom ergänzt werden muss, um Fehlererkennung und -korrektur zu realisieren.

Eine zentrale Fragestellung der Informationstheorie ist, inwieweit ein Datenstrom komprimiert werden kann, ohne dass die Information, die er codiert, verfälscht wird. Im Gegensatz zur fehlertoleranten Codierung, bei der Redundanz von essentieller Bedeutung ist, strebt die Informationstheorie also gerade das Gegenteil an, nämlich die Verkürzung von Codierungen: Wie lang muss eine Bitfolge mindestens sein, um eine Information zu übertragen oder zu speichern?

Dazu benötigt man ein Maß, mit dem der Informationsgehalt eines Datenstroms gemessen werden kann. Ein solches Maß hat Claude Shannon mit der **Entropie einer Informationsquelle** eingeführt. Eine Quelle Q von Datenströmen besteht aus Symbolen der Menge $A_Q = \{a_1, a_1, \ldots, a_N\}$, die mit den unabhängigen Wahrscheinlichkeiten p_i auftreten (Quellen mit unabhängigen Wahrscheinlichkeiten für das Auftreten der Symbole werden gedächtnislos genannt). Eine kompakte Darstellung einer solchen Quelle ist die Folgende:

$$Q = \begin{pmatrix} a_1 & a_2 & \ldots & a_N \\ p_1 & p_2 & \ldots & p_N \end{pmatrix}$$

Dabei ist $p_i > 0$ und $\sum_{i=1}^{N} p_i = 1$. Je niedriger p_i ist, um so höher ist der Informationsgehalt der Nachricht a_i. Nachrichten, die mit hoher Wahrscheinlichkeit auftreten, tragen wenig Information, denn ihr Auftreten ist nicht unerwartet; sie haben also keinen besonderen Neuigkeitswert. Eine mathematische Modellierung dieser Annahmen führt zur Definition des Informationsgehalts

$$I(a_i) = -\log p_i \tag{1.1}$$

[1] Richard Hamming (1915–1998), amerikanischer Mathematiker und Ingenieur, entwickelte die Grundlagen für fehlertolerante Codes. Außerdem leistete er u. a. Beiträge zur numerischen Analysis und zur Lösung von Differentialgleichungen, und er war an der Entwicklung der ersten IBM 650-Rechner beteiligt.

[2] Claude E. Shannon (1916–2001) war ein amerikanischer Mathematiker und Elektrotechniker, der 1948 mit seiner Arbeit *A Mathematical Theory of Communication* die Informationstheorie begründete. Shannon war vielseitig interessiert und begabt. So lieferte er unter anderem beachtete Beiträge zur Booleschen Algebra; er beschäftigte sich mit dem Bau von Schachcomputern und der Entwicklung von Schachprogrammen, und er lieferte Beiträge zur Lösung wirtschaftswissenschaftlicher Probleme.

des Nachrichtensymbols a_i. Die Entropie von Q

$$H(Q) = -\sum_{i=1}^{N} p_i \log p_i \tag{1.2}$$

gibt ihren mittleren Informationsgehalt an.

Die Symbole von A_Q müssen jetzt zur Speicherung auf Datenträgern oder zur digitalen Übertragung durch Bitfolgen codiert werden. Eine Bitfolge besteht aus Nullen und Einsen, kann also als Wort über dem Alphabet $\mathbb{B} = \{0, 1\}$ betrachtet werden. Wenn wir mit $\mathbb{B}^*$ die Menge aller endlichen Bitfolgen bezeichnen, kann eine Codierung von A_Q durch eine Abbildung $c : A_Q \to \mathbb{B}^*$ beschrieben werden. Diese Abbildung sollte injektiv sein, damit eine eindeutige Decodierung der Codewörter möglich ist. Sei A_Q^* die Menge aller endlichen Datenströme, die über A_Q gebildet werden können. Wir erweitern c zur Codierung von Datenströmen zur Abbildung $c^* : A_Q^* \to \mathbb{B}^*$, definiert durch

$$c^*(a_1 a_2 \dots a_k) = c(a_1)c(a_2)\dots c(a_k).$$

Damit auch die Decodierung von Datenströmen eindeutig ist, muss die Menge $c(A_Q)$ der Codewörter präfixfrei sein, d. h. für alle Symbole $a, a' \in A_Q$, $a \neq a'$, muss gelten, dass $c(a)$ kein Präfix von $c(a')$ ist. Das folgende einfache Beispiel zeigt, dass die Präfixfreiheit für die eindeutige Decodierung von Datenströmen erforderlich ist. Sei z. B. $c(a) = 11$ und $c(a') = 111$, dann gilt

$$c^*(aa') = c(a)c(a') = 11\,111 = 111\,11 = c(a')c(a).$$

Die Bitfolge 11111 kann also die Codierung von aa' oder die Codierung von $a'a$ sein.

Es stellt sich jetzt die Frage nach einer optimalen präfixfreien Codierung von A_Q. Optimal soll bedeuten, dass die mittlere Codewortlänge minimal ist. Sei $|x|$ die Länge der Bitfolge $x \in \mathbb{B}^*$, dann ist $\ell_c(Q) = \sum_{i=1}^{N} p_i\,|c(a_i)|$ die mittlere Codewortlänge der Codierung c von A_Q, und

$$\overline{\ell}(Q) = \min\{\ell_c(Q) \mid c \text{ ist präfixfreie Codierung von } A_Q\}$$

ist die minimale mittlere Codewortlänge zur präfixfreien Codierung von A_Q. Es kann gezeigt werden, dass

$$H(Q) \leq \overline{\ell}(Q) < H(Q) + 1$$

gilt. Der Aufwand für eine optimale präfixfreie Codierung einer gedächtnislosen Quelle entspricht quasi deren Entropie. Ein Verfahren, mit dem zu jeder solchen Quelle ein optimaler Code bestimmt werden kann, ist die Huffman-Codierung.[3]

Wesentlich für die Möglichkeiten zur Kompression von Datenströmen einer Quelle sind die Wahrscheinlichkeiten für das Auftreten der Quellensymbole. Völlig unberücksichtigt bleibt dabei die Struktur der Codewörter. Tritt z. B. ein Codewort auf,

[3] David A. Huffman (1925–1999) war ein amerikanischer Informatiker, der 1953 in einer Seminararbeit am Massachussetts Institute of Technology dieses Verfahren entwickelte.

das aus 32 Einsen besteht, könnte man dieses wie folgt beschreiben: Die Länge 32 wird als Dualzahl 100000 codiert und dahinter die 1 geschrieben: 100000 1. Diese Beschreibung der Bitfolge, die aus 32 Einsen besteht, benötigt nur sieben anstelle von 32 Bits, ist also deutlich kürzer. Allerdings muss klar sein, wo der dual codierte Wiederholungsfaktor aufhört und die zu wiederholende Bitfolge beginnt.

Wir werden im folgenden Abschnitt beispielhaft Kompressionsmöglichkeiten für Bitfolgen betrachten und im Kapitel 7 ein allgemeines Maß für die Kompression von Bitfolgen kennenlernen.

1.2 Beispiele für die Komprimierung von Bitfolgen

In diesem Abschnitt betrachten wir zur Einstimmung in die Thematik beispielhaft einige Möglichkeiten zur Kompression von Bitfolgen. Dazu müssen wir zunächst ein paar grundlegende Begriffe für Bitfolgen und Mengen von Bitfolgen kennenlernen. Diese und weitere Begriffe werden wir im folgenden Kapitel noch einmal aufgreifen und verallgemeinert betrachten.

1.2.1 Grundlegende Definitionen für Bitfolgen

Bitfolgen sind endliche Zeichenketten, die mit den Elementen 0 und 1 der Menge $\mathbb{B} = \{0, 1\}$ gebildet werden können. Das Symbol ε stellt die leere Zeichenkette dar, sie hat die Länge 0. Mit $\mathbb{B}^*$ bezeichnen wir die Menge aller endlichen Bitfolgen. Diese lässt sich wie folgt mathematisch präzise definieren:

(i) $\varepsilon \in \mathbb{B}^*$: Die **leere Bitfolge** gehört zu $\mathbb{B}^*$.

(ii) Für $x \in \mathbb{B}^*$ sind auch $x0, x1 \in \mathbb{B}^*$: Neue Bitfolgen werden konstruiert, indem an vorhandene Nullen oder Einsen angehängt werden.

Mit $\mathbb{B}^+ = \mathbb{B}^* - \{\varepsilon\}$ bezeichnen wir die Menge, die alle Bitfolgen außer der leeren Folge enthält.

Die **Länge einer Bitfolge** ist die Anzahl der in ihr vorkommenden Nullen und Einsen. Diese Länge können wir wie folgt mathematisch – gemäß dem in (i) und (ii) definierten rekursiven Aufbau von Bitfolgen – festlegen durch die Abbildung

$$|\ |: \mathbb{B}^* \to \mathbb{N}_0$$

definiert durch

$$\begin{aligned} |\varepsilon| &= 0 \\ |xb| &= |x| + 1 \text{ für } x \in \mathbb{B}^* \text{ und } b \in \mathbb{B} \end{aligned} \tag{1.3}$$

Damit lässt sich beispielsweise berechnen:

$$|101| = |10| + 1 = |1| + 1 + 1 = |\varepsilon| + 1 + 1 + 1 = 0 + 1 + 1 + 1 = 3$$

Der Betrag $|x|_0$ gibt die Anzahl der Nullen in x an, entsprechend gibt $|x|_1$ die Anzahl der Einsen in x an.

Übung 1.1 *Überlegen Sie sich eine Definition für $|x|_0$ und $|x|_1$!*

Eine mathematische Definition für diese Funktion ist:

$$|\cdot|_\cdot : \mathbb{B}^* \times \mathbb{B} \to \mathbb{N}_0,$$

definiert durch

$$\begin{aligned} |\varepsilon|_b &= 0, \\ |xc|_b &= \begin{cases} |x|_b + 1, & c = b \\ |x|_b, & c \neq b \end{cases} \quad \text{für } x \in \mathbb{B}^* \text{ und } b, c \in \mathbb{B}. \end{aligned} \tag{1.4}$$

Damit lässt sich z. B. berechnen:

$$|101|_1 = |10|_1 + 1 = |1|_1 + 1 = |\varepsilon|_1 + 1 + 1 = 0 + 1 + 1 = 2$$

Übung 1.2 *Wie viele Bitfolgen der Länge $n \in \mathbb{N}_0$ gibt es?*

Bei der Bildung einer Bitfolge $b_1 b_2 \ldots b_n$ haben wir für jedes Bit b_i, $1 \leq i \leq n$, zwei Möglichkeiten, für n Bits also

$$\underbrace{2 \cdot 2 \cdot \ldots \cdot 2}_{n\text{-mal}} = 2^n \tag{1.5}$$

Möglichkeiten.

Die Menge aller Bitfolgen der Länge n ist

$$\mathbb{B}^n = \{x \in \mathbb{B}^* : |x| = n\}. \tag{1.6}$$

Es gilt also z. B.

$$\mathbb{B}^0 = \{\varepsilon\} \text{ sowie } \mathbb{B}^3 = \{000, 001, 010, 011, 100, 101, 110, 111\}$$

Wegen (1.5) gilt

$$|\mathbb{B}^n| = 2^n. \tag{1.7}$$

Die Menge

$$\mathbb{B}^{\leq n} = \{x \in \mathbb{B}^* : |x| \leq n\} \tag{1.8}$$

enthält alle Bitfolgen der Länge kleiner gleich n. Es ist also z. B.

$$\mathbb{B}^{\leq 3} = \{\varepsilon, 0, 1, 00, 01, 10, 11, 000, 001, 010, 011, 100, 101, 110, 111\}.$$

Übung 1.3 *Überlegen Sie, dass folgende Beziehungen gelten:*

$$\mathbb{B}^{\leq n} = \bigcup_{i=0}^{n} \mathbb{B}^i = \mathbb{B}^0 \cup \mathbb{B}^1 \cup \mathbb{B}^2 \cup \ldots \cup \mathbb{B}^n$$

$$\left|\mathbb{B}^{\leq n}\right| = \sum_{i=0}^{n} 2^i = 2^{n+1} - 1$$

$$\mathbb{B}^* = \bigcup_{n \in \mathbb{N}_0} \mathbb{B}^n = \mathbb{B}^0 \cup \mathbb{B}^1 \cup \mathbb{B}^2 \cup \mathbb{B}^3 \cup \ldots$$

In Kapitel 8.6 betrachten wir unendliche Bitfolgen. Die Menge aller unendlich langen Bitfolgen bezeichnen wir mit $\mathbb{B}^\omega$. Für $w \in \mathbb{B}^\omega$ und $n \in \mathbb{N}$ ist dann $w[1, n]$ das Anfangsstück der Länge n von w, und für $i \in \mathbb{N}$ ist $w[i]$ das i-te Bit in w.

1.2.2 Beispiele

Betrachten wir die Bitfolge

$$v = 101101101101101101101101. \tag{1.9}$$

so erkennen wir eine Regelmäßigkeit, nämlich dass sich v aus acht 101-Gruppen zusammensetzt. v lässt sich mit bekannten mathematischen Notationen kürzer darstellen – also „komprimieren“ – etwa durch

$$v_1 = [101]^8. \tag{1.10}$$

Betrachten wir als nächstes die Bitfolge

$$w = 110101110111001010101111. \tag{1.11}$$

so erscheint diese „komplexer“ zu sein, da keine Regelmäßigkeit zu erkennen ist und man keine Komprimierung findet. Man könnte w als komplexer ansehen als v, da sich der Informationsgehalt von v wesentlich kürzer darstellen lässt als ausgeschrieben wie in (1.9). Wegen seiner regelmäßigen Struktur erscheint das Wort v, bezogen auf seine Länge, weniger Information zu tragen als das unregelmäßige Wort w, denn es lässt sich ohne Informationsverlust komprimieren.

Es stellt sich die Frage nach einem Komplexitätsbegriff für Informationen, der die Regelmäßigkeit bzw. die Zufälligkeit von Bitfolgen besser erfasst. Wie schon angedeutet, könnte die Komprimierbarkeit von Bitfolgen ein Kandidat für ein entsprechendes Informationsmaß sein. Dazu müssen wir präzise, d. h. mit mathematischen Mitteln, definieren, was wir unter einer Komprimierung einer Bitfolge verstehen wollen.

Übung 1.4 *Überlegen Sie sich eine mathematische Definition für die Komprimierung von Bitfolgen!*

Eine Komprimierung sollte Folgendes leisten:

(1) Komprimierungen von Bitfolgen sollen wieder Bitfolgen sein, und

(2) ein Kompressionsverfahren soll alle Bitfolgen komprimieren können, d. h. eine Komprimierung ordnet allen Bitfolgen eine Bitfolge zu.

(3) Des Weiteren soll die Komprimierung kürzer sein als die ursprüngliche Folge.

(4) Die Komprimierung soll verlustlos sein, d. h. aus der Komprimierung muss die ursprüngliche Folge rekonstruiert werden können.

Wir geben eine für die Betrachtungen in diesem Kapitel hinreichende Definition für die Komprimierung von Bitfolgen an; im Kapitel 7 werden wir diese noch weiter präzisieren.

Definition 1.1 *Wir nennen eine totale Abbildung* $\kappa : \mathbb{B}^* \to \mathbb{B}^*$ *eine* **Komprimierung** *von* $\mathbb{B}^*$*. Eine Komprimierung* κ *heißt* **echte Komprimierung** *von* $\mathbb{B}^*$*, falls* $|\kappa(x)| < |x|$ *für fast alle* $x \in \mathbb{B}^*$ *ist.*

Eine Komprimierung κ ordnet also jeder Bitfolge eine Bitfolge zu, weil κ total sein soll; damit sind die Anforderungen (1) und (2) erfüllt. Wir sind natürlich an echten Komprimierungen interessiert. Diese können allerdings Anforderung (3) nicht erfüllen. Auf diese Problematik werden wir im Kapitel 7 näher eingehen.

Die Zeichenkette v_1, siehe (1.10), ist keine Bitfolge, also gemäß Definition 1.1 keine zulässige Komprimierung von v. Mit dem Ziel, v echt zu komprimieren, stellen wir als nächstes den Exponenten dual dar:

$$v_2 = [101]\,1000$$

v_2 ist nun eine Zeichenkette, die aus den Symbolen der Menge $\{\,0, 1, [\,,]\,\}$ besteht. Es müssen jetzt also noch die eckigen Klammern dual codiert werden – natürlich so, dass daraus unsere ursprüngliche Folge v zurückgewonnen werden kann; die Komprimierung soll ja verlustlos sein.

Übung 1.5 *Überlegen Sie, wie wir die Elemente der Menge* $\{0, 1, [\,,]\}$ *so mit Bitfolgen darstellen können, dass* v_2 *vollständig als Bitfolge codiert werden kann!*

Wir erreichen eine zulässige Komprimierung etwa mithilfe der Abbildung

$$\eta : \{\, 0, 1, [\ ,]\,\} \to \mathbb{B}^2,$$

definiert durch

$$\eta(0) = 00,\ \ \eta(1) = 11,\ \ \eta([) = 10,\ \ \eta(]) = 01.$$

Wir erweitern η zu $\eta^* : \{0, 1, [\ ,]\}^* \to (\mathbb{B}^2)^*$ definiert durch

$$\begin{aligned} \eta^*(\varepsilon) &= \varepsilon, \\ \eta^*(bw) &= \eta(b)\eta^*(w), \ \text{ für } b \in \{0, 1, [\ ,]\} \ \text{ und } \ w \in \{0, 1, [\ ,]\}^*. \end{aligned}$$

Wir werden im Folgenden nicht zwischen η und η^* unterscheiden und beide Funktionen mit η bezeichnen.

Für unser Beispiel ergibt sich dann

$$v_3 = \eta(v_2) = 10\,110011\,01\,11000000$$

als eine mögliche Komprimierung von v. Es gilt

$$|v_3| = 18 < 24 = |v|\,.$$

Mit unserer Methode haben wir also v echt und rekonstruierbar zu v_3 komprimiert.

Übung 1.6 *Konstruieren Sie für die Bitfolge*

$$x = 000000110101010101010101010101010101010101010101010101$$

mit dem oben beispielhaft vorgestellten Verfahren eine echte, rekonstruierbare Komprimierung!

Die Bitfolge x lässt sich zunächst wie folgt darstellen:

$$x_1 = [0]^6\,[1]^1\,[10]^{22}$$

Daraus ergibt sich mit der Dualdarstellung der Exponenten

$$x_2 = [0]\,110\,[1]\,1\,[10]\,10110$$

und schließlich mit der Abbildung η

$$x_3 = \eta(x_2) = 10\,00\,01\,111100\,10\,11\,01\,11\,10\,1100\,01\,1100111100$$

als eine mögliche echte, rekonstruierbare Komprimierung von x mit

$$|x_3| = 38 < 51 = |x|\,.$$

Als weiteres Beispiel für Kompressionsmöglichkeiten betrachten wir den Fall, dass der Wiederholungsfaktor eine Mehrfachpotenz ist. Sei etwa die Folge

$$x = \underbrace{1010\ldots10}_{4\,294\,967\,296\text{-mal}} \tag{1.12}$$

mit der Potenzdarstellung

$$x_{11} = [10]^{4\,294\,967\,296} \tag{1.13}$$

gegeben. Dann kann man diese etwa durch

$$x_{12} = [10]^{2^{32}} \tag{1.14}$$

kürzer darstellen, oder noch kürzer durch

$$x_{13} = [10]^{2^{2^5}}. \tag{1.15}$$

Diese Art der Darstellung kann man beliebig weiter treiben, um kurze Darstellungen für

$$[10]^{2^n},\ [10]^{2^{2^n}},\ [10]^{2^{2^{2^n}}},\ \ldots$$

zu erhalten. Wir nennen diese Art der Darstellungen **iterierte Zweierpotenzen**.

Übung 1.7 **a)** *Konstruieren Sie für die Bitfolge (1.13) mit dem oben beispielhaft vorgestellten Verfahren eine echte, rekonstruierbare Komprimierung!*

b) *Überlegen Sie sich für iterierte Zweierpotenzen eine Komprimierung!*

c) *Komprimieren Sie mit diesem Verfahren die Bitfolgen (1.14) und (1.15)!*

d) *Vergleichen Sie die Ergebnisse von a) und c)!*

Zum Abschluss dieser ersten Beispiele für die Komprimierung von Bitfolgen wollen wir eine weitere Möglichkeit für deren Komprimierung ebenfalls beispielhaft betrachten. Wir können alle Elemente von $\mathbb{B}^+$ als **Dualdarstellungen** natürlicher Zahlen auffassen. Für $x = x_{n-1}\ldots x_0 \in \mathbb{B}^n$, $n \geq 1$, sei

$$wert(x) = \sum_{i=0}^{n-1} x_i \cdot 2^i$$

der Wert von x dargestellt im Dezimalsystem.

Jede Zahl $n \in \mathbb{N}$, $n \geq 2$, kann faktorisiert werden, d.h. als Produkt von Primzahlpotenzen dargestellt werden:

$$n = \prod_{j=1}^{k} p_j^{\alpha_j} \tag{1.16}$$

Dabei ist $p_j \in \mathbb{P}$, $1 \leq j \leq k$, und $p_j < p_{j+1}$, $1 \leq j \leq k-1$, sowie $\alpha_j \in \mathbb{N}$, $1 \leq j \leq k$. Die Darstellung in dieser Art, d. h. mehrfach vorkommende Primfaktoren zu Potenzen zusammengefasst und die Primzahlen der Größe nach aufgereiht, ist eindeutig. Diese **(Prim-) Faktorisierung** kann man mit den Symbolen der Menge $\{0, 1, [\ , \ ]\}$ wie folgt darstellen:

$$dual(p_1)\,[dual(\alpha_1)]\,dual(p_2)\,[dual(\alpha_2)]\ldots dual(p_k)\,[dual(\alpha_k)]\,. \tag{1.17}$$

Dabei ist $dual(n)$ die Dualdarstellung von $n \in \mathbb{N}_0$, es gilt also $wert(dual(n)) = n$. Mithilfe der Abbildung η ergibt sich daraus wieder eine Darstellung über $\mathbb{B}$.

Betrachten wir als Beispiel die Zahl

$$n = 1\,428\,273\,264\,576\,872\,238\,279\,737\,182\,200\,000$$

mit der Dualdarstellung $z = dual(n) =$

$$\begin{aligned}&1000110011010110101011111010001111000101100010001101101 0\\ &1011111000011111111111000011000100010110010000011000000.\end{aligned}$$

Die Faktorisierung von z ist

$$2^6 \cdot 3^9 \cdot 5^5 \cdot 7^{11} \cdot 11^3.$$

Ihre Darstellung über $\{\,0, 1, [\ , \ ]\,\}$ ist

$$z_{\mathbb{P}} = 10\,[110]\,11\,[1001]\,101\,[101]\,111\,[1011]\,.1011\,[11]\,.$$

Die Anwendung von η liefert die Darstellung

$$\begin{aligned}\eta(z_{\mathbb{P}}) =\,&1100\,10\,111100\,01\,1111\,10\,11000011\,01\,110011\,10\,110011\,01\\ &111111\,10\,11001111\,01\,11001111\,10\,1111\,01.\end{aligned}$$

Es ist

$$|\eta(z_{\mathbb{P}})| = 80 < 111 = |z|\,.$$

Man könnte im Übrigen anstelle einer Primzahl selbst ihre Nummer innerhalb einer aufsteigenden Auflistung der Primzahlen in der Darstellung einer Zahl verwenden, d. h. anstelle von p_i ihre Nummer i. Also z. B. anstelle von $p_1 = 2$ verwenden wir 1 und anstelle von $p_5 = 11$ verwenden wir 5. Damit ergäbe sich für die obige Bitfolge z anstelle von $z_{\mathbb{P}}$ die Darstellung

$$z'_{\mathbb{P}} = 1\,[110]\,10\,[1001]\,11\,[101]\,100\,[1011]\,101\,[11]$$

worauf die Anwendung von η die Darstellung

$$\begin{aligned}\eta(z'_{\mathbb{P}}) =\,&11\,10\,111100\,01\,1100\,10\,11000011\,01\,1111\,10\,110011\,01\\ &110000\,10\,11001111\,01\,110011\,10\,1111\,01\end{aligned}$$

mit

$$|\eta(z'_{\mathbb{P}})| = 74$$

ergibt.

Übung 1.8 *Überlegen Sie sich Optimierungen der beiden vorgestellten Methoden oder weitere Methoden, die möglicherweise noch bessere Komprimierungen erreichen!*

Bei allen Kompressionsmethoden, die wir uns bisher überlegt haben oder uns neu ausdenken, stellen sich Fragen wie: Welche davon ist besser, welche ist die beste? Die Methoden sind kaum vergleichbar. Die eine Methode komprimiert manche Zeichenketten besser als die andere, und umgekehrt. Es stellt sich also die Frage nach einer Komprimierungsmethode, die allgemeingültige Aussagen zulässt und die zudem noch in sinnvoller Weise die Komplexität einer Zeichenkette beschreibt. Etwas allgemeiner betrachtet geht es um ein

Komplexitätsmaß für den Informationsgehalt von Zeichenfolgen.

Wegen ihrer wesentlichen Beiträge zu dieser Thematik spricht man auch von

Kolmogorov-, *Solomonoff*- oder *Chaitin-Komplexität*.[4]

Eine heute gängige Bezeichnung, die die genannten und weitere damit zusammenhängende Aspekte umfasst, ist

Algorithmische Informationstheorie,

in die dieses Buch eine Einführung gibt.

1.3 Inhaltsübersicht

Für die Einführung in die Algorithmische Informationstheorie benötigen wir Kenntnisse über formale Sprachen und Berechenbarkeit; deshalb betrachten wir vorher grundlegende Konzepte, Methoden und Verfahren aus diesen beiden Themengebieten.

Im Kapitel 2 werden Alphabete, Wörter und formale Sprachen sowie wichtige auf ihnen operierende Funktionen und Verknüpfungen allgemein eingeführt. Des Weiteren werden Codierungen und Nummerierungen von Alphabeten und Wörtern durch Bitfolgen und natürliche Zahlen vorgestellt. Mithilfe solcher Codierungen und Nummerierungen können wir bei späteren Betrachtungen eine im Einzelfall geeignete Repräsentation von Daten wählen; die Betrachtungen treffen dann auch auf andere Repräsentationen zu.

[4] A. N. Kolmogorov (1903–1987) war ein russischer Mathematiker, der als einer der bedeutendsten Mathematiker des 20. Jahrhunderts gilt. Er lieferte wesentliche Beiträge zu vielen Gebieten der Mathematik und auch zur Physik. Einer seiner grundlegenden Beiträge ist die Axiomatisierung der Wahrscheinlichkeitstheorie.

R. Solomonoff (1926–2009) war ein amerikanischer Mathematiker, der schon zu Beginn der sechziger Jahre des vorigen Jahrhunderts Grundideen zur Algorithmischen Informationstheorie veröffentlichte (also vor Kolmogorov, dem aber diese Arbeiten nicht bekannt waren).

G. Chaitin (*1947) ist ein amerikanischer Mathematiker, der sich mit Fragen zur prinzipiellen Berechenbarkeit beschäftigt. Im Kapitel 8 gehen wir näher auf einige seiner Ansätze ein.

Für die praktische Verwendung von formalen Sprachen muss entscheidbar sein, ob ein Wort zu der Sprache gehört oder nicht. So muss es z. B. für eine Programmiersprache ein Programm geben (etwa als Bestandteil eines Compilers), das überprüft, ob eine Zeichenkette den Syntaxregeln der Sprache genügt. Denn nur syntaktisch korrekte Programme können in Maschinensprache übersetzt und dann ausgeführt werden.

Im Kapitel 3 wird die Turingmaschine als eine mögliche mathematische Präzisierung für die Begriffe Algorithmus und Programm vorgestellt. Damit liegt eine mathematische Definition für die Berechenbarkeit von Funktionen und für die Entscheidbarkeit von Mengen vor. So können Aussagen über solche Funktionen und Mengen getroffen und bewiesen werden. Des Weiteren wird die Äquivalenz der für die Theoretische Informatik wichtigen Varianten von Turingmaschinen, nämlich deterministische und nicht deterministische Turingmaschinen sowie Turing-Verifizierer, bewiesen.

Kapitel 4 behandelt die Laufzeitkomplexität von Algorithmen. Die Komplexitätsklassen P (deterministisch in Polynomzeit entscheidbare Mengen) und NP (nicht deterministisch in Polynomzeit entscheidbare Mengen) werden vorgestellt, und ihre Bedeutung sowohl für die Theoretische Informatik als auch für praktische Anwendungen wird erläutert.

Für Theorie und Praxis von wesentlicher Bedeutung ist auch, dass es realisierbare universelle Berechenbarkeitskonzepte gibt. Das bedeutet, dass es innerhalb eines solchen Konzepts ein Programm gibt, das alle anderen Programme ausführen kann. Sonst müsste für jedes Programm eine eigene Maschine gebaut werden, die nur dieses Programm ausführen kann. Im Kapitel 5 werden das Konzept und wesentliche Eigenschaften der universellen Turingmaschine betrachtet. Die universelle Turingmaschine ist eine theoretische Grundlage für die Existenz von universellen Rechnern, die jedes Programm ausführen können.

Durch Codierung und Nummerierung von Turingmaschinen gelangt man zu einer abstrakten universellen Programmiersprache. Es wird gezeigt, dass diese alle wesentliche Anforderungen erfüllt, die man an eine solche Sprache stellen kann. Des Weiteren werden weitere wichtige Eigenschaften dieser Sprache bewiesen. Diese Eigenschaften gelten für alle universellen Sprachen.

Im Kapitel 6 werden die Grenzen der algorithmischen Berechenbarkeit aufgezeigt. Diese sind nicht nur von theoretischer Bedeutung, sondern auch von praktischer. So wird gezeigt, dass es keinen Programmbeweiser geben kann, der im Allgemeinen entscheiden kann, ob ein Programm seine Spezifikation erfüllt, d. h. „korrekt“ ist. Solche Beweiser wären für die Softwareentwicklung von immenser Bedeutung.

Während im Kapitel 4 die Komplexität von Berechnungen behandelt wird, wird im Kapitel 7 die Beschreibungskomplexität von Datenströmen betrachtet. Ein Maß dafür ist die Kolmogorov-Komplexität. Diese wird vorgestellt, und ihre wesentlichen Eigenschaften werden untersucht. Der Begriff der Kolmogorov-Komplexität ermöglicht, grundsätzliche Aussagen über Komprimierungsmöglichkeiten von Daten zu treffen. Damit kommen wir in diesem Kapitel auf die grundsätzliche Frage nach einem allgemeingültigen Komplexitätsmaß für den Informationsgehalt von Zeichenfolgen zurück, die wir am Ende des vorigen Abschnitts gestellt haben.

Im Kapitel 8 werden Anwendungen der Kolmogorov-Komplexität in der Theoretischen Informatik vorgestellt. Des Weiteren wird die Chaitin-Konstante erläutert. Diese wird auch „Chaitins Zufallszahl der Weisheit“ genannt, da in ihr die Antworten zu allen mathematischen Entscheidungsfragen versteckt sind.

Am Ende des Kapitels wird noch kurz auf Anwendungsmöglichkeiten der Kolmogorov-Komplexität bei der Datenanalyse eingegangen.

1.4 Zusammenfassung und bibliografische Hinweise

In diesem Kapitel haben wir zunächst einen Einblick in grundlegende Konzepte der Shannonschen Informationstheorie gegeben, insbesondere im Hinblick auf Grenzen für die Komprimierung von Informationen. Die Grundlage dafür ist die Wahrscheinlichkeit, mit der die Symbole, aus denen die Nachrichten, welche die Träger der Informationen sind, gebildet werden.

Anschließend haben wir beispielhaft gesehen, dass, wenn von solchen Wahrscheinlichkeiten abgesehen wird, Bitfolgen stärker komprimiert werden können. Dabei spielt die Struktur der Bitfolgen eine entscheidende Rolle. Allerdings ist das Vorgehen zur Beschreibung der Folgen nicht systematisch; es soll das Problembewusstsein dafür wecken, dass ein „universelles Maß“ zur Beschreibung der Informationskomplexität von Bitfolgen vonnöten ist. Ein solches Maß wird in den Kapiteln 7 und 8 eingeführt, analysiert und angewendet.

Einführungen in die Codierungs- und Informationstheorie sowie in Konzepte, Methoden und Verfahren zur Datenkompression findet man in [D06],[MS08], [Sa05], [Schu03] und [SM10]

Kapitel 2

Alphabete, Wörter, Sprachen

Im vorigen Kapitel haben wir nur Bitfolgen, d. h. nur Zeichenketten, die mit den Symbolen 0 und 1 gebildet werden, betrachtet. Diese beiden Symbole können als Buchstaben des Alphabets $\mathbb{B} = \{0, 1\}$ angesehen werden. Im Allgemeinen sind die Träger von Informationen nicht Bitfolgen, sondern Wörter über jeweils geeigneten endlichen Alphabeten. Das gilt für natürliche Sprachen wie z. B. für die deutsche Sprache als auch für formale Sprachen wie z. B. Programmiersprachen. Die Wörter der deutschen Sprache werden mit den bekannten 26 Klein- und Großbuchstaben, d. h. über dem Alphabet $\{a, b, c, \ldots, z, A, B, C, \ldots, Z\}$, gebildet. In der Schriftsprache kommen dann noch weitere Symbole hinzu, unter anderem der Punkt, das Komma und weitere Symbole zur Zeichensetzung sowie verschiedene Formen von Klammern. Programme von Programmiersprachen werden ebenfalls über einem vorgegebenen Alphabet gebildet. Dazu gehören Symbole aus dem ASCII-Zeichensatz[1] sowie Schlüsselwörter wie z. B. `read`, `write` und `while`.

Wörter einer Sprache werden – und das gilt sowohl für natürliche als auch für formale Sprachen – nach bestimmten Regeln über dem jeweils zugrunde liegenden Alphabet gebildet. So ist *Rechner* ein Wort der deutschen Sprache, *Urx* hingegen nicht, und `i++` ist eine zulässige Anweisung in der Programmiersprache C++, die Zeichenkette `i:=+-/` aber nicht.

Folgen von Wörtern bilden in natürlichen Sprachen Sätze. Wir wollen weder in natürlichen noch in formalen Sprachen zwischen Wörtern und Sätzen unterscheiden, da wir einen Satz, der aus aneinandergereihten durch Zwischenräume getrennten Wörtern besteht, als ein Wort betrachten wollen. Der Zwischenraum, auch *Blank* genannt, muss dann natürlich ein Symbol aus dem zugrunde liegenden Alphabet sein, sonst könnte es kein Buchstabe in einem solchen Wort sein.

Eine *Sprache* ist eine Menge von Wörtern. Für natürliche Sprachen und Programmiersprachen sollte festgelegt sein, welche Wörter über dem zugrunde liegenden Alphabet zu einer Sprache gehören. Für Programmiersprachen sollte es Programme – etwa als Bestandteil von Compilern – geben, die entscheiden, ob ein Wort zur Sprache gehört oder nicht. Um diese Problematik – in späteren Kapiteln – grundlegend zu

[1] *ASCII* ist die Abkürzung für *American Standard Code for Information Interchange*.

K.-U. Witt und M. E. Müller, *Algorithmische Informationstheorie*,
https://doi.org/10.1007/978-3-662-61694-9_2

untersuchen, werden wir am Ende dieses Kapitels Sprachklassen zur Behandlung von sogenannten Entscheidbarkeitsfragen einführen.

2.1 Alphabete

Das Alphabet einer Sprache ist die Menge der atomaren Grundsymbole, aus denen die Wörter der Sprache gebildet werden. Die Alphabete der deutschen Sprache bzw. der Programmiersprache C++ haben wir oben in der Einleitung erwähnt. Wir betrachten nur endliche Alphabete.

Im Allgemeinen bezeichnen wir ein Alphabet mit dem griechischen Buchstaben Σ. Falls wir kein konkretes, sondern allgemein ein Alphabet betrachten, dann benennen wir dessen Buchstaben in der Regel mit Buchstaben vom Anfang des deutschen Alphabets, also etwa mit a, b, c, d, ..., wie z. B. im Alphabet

$$\Sigma = \{a, b, c\}$$

Wenn die Anzahl der Buchstaben nicht genau bestimmt ist, benutzen wir diese Bezeichner indiziert, wie z. B. im folgenden Alphabet:

$$\Sigma = \{a_1, a_2, \ldots, a_n\}\,,\ n \geq 0$$

Der Fall $n = 0$ bedeutet, dass Σ leer ist: $\Sigma = \emptyset$.

Das Alphabet $\Sigma = \{0, 1\}$ zur Bildung von Bitfolgen kennen wir aus dem vorigen Kapitel. Wir haben es dort mit $\mathbb{B}$ bezeichnet und werden das auch weiterhin tun.[2]

Durch die Reihenfolge der Aufzählung der Buchstaben in Σ soll eine (**lexikografische** oder **alphabetische**) **Ordnung** festgelegt sein: $a_i < a_{i+1}$, $1 \leq i \leq n-1$. Wenn wir z. B. $\mathbb{B} = \{0, 1\}$ schreiben, bedeutet das also, dass $0 < 1$ gilt. Wenn wir möchten, dass $1 < 0$ gilt, müssen wir $\mathbb{B} = \{1, 0\}$ schreiben.

2.2 Wörter und Wortfunktionen

Die endlich langen Zeichenfolgen, die über einem Alphabet Σ gebildet werden können, heißen **Wörter** über Σ. Wörter entstehen, indem Symbole oder bereits erzeugte Wörter aneinandergereiht (miteinander verkettet, konkateniert) werden. Die Menge Σ^* aller Wörter, die über dem Alphabet Σ gebildet werden kann, ist wie folgt definiert (vgl. Definition von $\mathbb{B}^*$ im Abschnitt 1.2.1):

[2] Die Bezeichnung dieses Alphabets mit $\mathbb{B}$ steht für die *booleschen Werte* 0 und 1. Der britische Mathematiker und Logiker George Boole (1815–1864) gilt als Begründer der mathematischen Logik. Durch Formalisierung des mathematischen Denkens (*An investigation of the laws of thought* ist eine berühmte Schrift von Boole zu diesem Thema) begründete er eine Algebra der Logik, d. h. eine Logik, mit der man „rechnen“ kann. Dazu hat er die Zahlen 0 und 1 zur Repräsentation der Wahrheitswerte *falsch* bzw. *wahr* verwendet und für deren Verknüpfung Rechenregeln aufgestellt. Außerdem entwickelte Boole Ideen zur Konstruktion von Rechnern, die Konzepte enthalten, wie sie etwa hundert Jahre später zum Bau der ersten realen universellen Rechner verwendet wurden.

(1) Jeder Buchstabe $a \in \Sigma$ ist auch ein Wort über Σ; es gilt also $a \in \Sigma^*$ für alle $a \in \Sigma$.

(2) Werden bereits konstruierte Wörter hintereinandergeschrieben, entstehen neue Wörter, d. h. sind $v, w \in \Sigma^*$, dann ist auch ihre **Verkettung** (**Konkatenation**) vw ein Wort über Σ; es gilt also $vw \in \Sigma^*$ für alle $v, w \in \Sigma^*$.

(3) ε, das **leere Wort**, ist ein Wort über (jedem Alphabet) Σ, d. h. es gilt immer $\varepsilon \in \Sigma^*$. ε ist ein definiertes Wort ohne „Ausdehnung“. Es hat die Eigenschaft: $\varepsilon w = w\varepsilon = w$ für alle $w \in \Sigma^*$.

Wegen der Bedingung (3) ist das leere Wort in jedem Fall ein Element von Σ^*, auch dann, wenn Σ leer ist.

Beispiel 2.1 *Sei* $\Sigma = \{a, b\}$*, dann ist*

$$\Sigma^* = \{\varepsilon, a, b, aa, ab, ba, bb, aaa, aab, aba, abb, baa, bab, bba, bbb, \ldots\}$$

Mit Σ^+ bezeichnen wir die Menge aller Wörter über Σ ohne das leere Wort, d. h. $\Sigma^+ = \Sigma^* - \{\varepsilon\}$.

Folgerung 2.1 *Ist ein Alphabet* Σ *nicht leer, dann besitzen* Σ^* *und* Σ^+ *unendlich viele Wörter; ist dagegen* $\Sigma = \emptyset$*, dann ist* $\Sigma^* = \{\varepsilon\}$ *und* $\Sigma^+ = \emptyset$*.*

Bemerkung 2.1 *Im algebraischen Sinne bildet die Rechenstruktur* $(\Sigma^+, \circ)$ *für ein Alphabet* Σ *mit der (Konkatenations-) Operation*

$$\circ : \Sigma^* \times \Sigma^* \to \Sigma^*, \text{ definiert durch } v \circ w = vw$$

eine **Halbgruppe***, denn die Konkantenation ist eine assoziative Operation: Für alle Wörter* $u, v, w \in \Sigma^*$ *gilt* $u \circ (v \circ w) = (u \circ v) \circ w$*. Wegen der in der obigen Definition festgelegten Eigenschaft (3) bildet die Struktur* (Σ^*, ε) *ein* **Monoid***. Die Konkatenation von Wörtern ist über Alphabeten mit mehr als einem Buchstaben nicht kommutativ.*

Ist

$$w = \underbrace{v\, v \ldots v}_{n\text{-mal}},$$

dann schreiben wir abkürzend $w = v^n$. Wir nennen v^n die $\boldsymbol{n}$**-te Potenz** von v. Es ist $v^0 = \varepsilon$. In Beispiel 2.1 können wir Σ^* also auch wie folgt schreiben:

$$\Sigma^* = \left\{\varepsilon, a, b, a^2, ab, ba, b^2, a^3, a^2b, aba, ab^2, ba^2, bab, b^2a, b^3, \ldots\right\}$$

Übung 2.1 *Geben Sie eine rekursive Definition für Wortpotenzen an!*

Allgemein notieren wir Wörter in der Regel mit Buchstaben vom Ende des deutschen Alphabets: u, v, w, x, y, z. Wenn wir im Folgenden ein Wort w buchstabenweise notieren, $w = w_1 \dots w_k$, $k \geq 0$, sind die w_i Buchstaben, also $w_i \in \Sigma$, $1 \leq i \leq k$; $k = 0$ bedeutet, dass w das leere Wort ist: $w = \varepsilon$. Anstelle von $w = w_1 \dots w_k$ schreiben wir auch

$$w = \prod_{i=1}^{k} w_i.$$

Sei $w \in \Sigma^*$ ein Wort mit $k \geq 0$ Buchstaben, dann meinen wir mit $w[\,i\,]$ für $1 \leq i \leq k$ den i-ten Buchstaben von w; für $1 \leq i, j \leq k$ ist $w[\,i, j\,]$ das Teilwort von w, das beim i-ten Buchstaben beginnt und beim j-ten Buchstaben endet:

$$w[\,i, j\,] = \begin{cases} \prod_{s=i}^{j} w[\,s\,], & i \leq j \\ \varepsilon, & i > j \end{cases}$$

Offensichtlich gilt $w[\,i\,] = w[\,i, i\,]$.

Sei $w \in \Sigma^*$ ein Wort der Länge $k \geq 0$. Dann heißt jedes Teilwort $w[\,1, i\,]$ mit $0 \leq i \leq k$ ein **Präfix** von w. Entsprechend heißt jedes Teilwort $w[\,i, j\,]$ mit $1 \leq i, j \leq k$ ein **Infix** und jedes Teilwort $w[\,i, k\,]$ mit $1 \leq i \leq k + 1$ ein **Suffix** von w. Ist $w = xy$ mit $x, y \in \Sigma^*$, dann heißen das Präfix x und das Suffix y **einander zugehörig** in w. Das leere Wort ist Prä-, Suf- und Infix von jedem Wort $w \in \Sigma^*$, denn es gilt $w = \varepsilon w$, $w = w\varepsilon$ bzw. $w = x\varepsilon y$ für alle Präfixe x und ihre zugehörigen Suffixe y von w.

Man beachte, dass diese Definitionen Spezialfälle zulassen. So ist wegen

$$w = w\varepsilon = \varepsilon w\varepsilon = \varepsilon w$$

ein Wort w sowohl Präfix als auch Infix als auch Suffix von sich selbst, oder für $w = xy$ mit $x, y \in \Sigma^+$ ist x sowohl Präfix als auch Infix, und y ist sowohl Infix als auch Suffix von w.

Falls x ein Präfix von w ist und $x \neq w$ gilt, dann heißt x **echter Präfix** von w. Entsprechende Definitionen gelten für die Begriffe **echter Infix** bzw. **echter Suffix**.

Die Menge

$$\mathit{Pref}(w) = \{x \mid x \text{ ist Präfix von } w\} \tag{2.1}$$

ist die **Menge der Präfixe** von w. Entsprechend können die **Menge** $\mathit{Inf}(w)$ **der Infixe** bzw. die **Menge** $\mathit{Suf}(w)$ **der Suffixe** von w definiert werden.

Übung 2.2 *Es sei* $\Sigma = \{a, b\}$. *Bestimmen Sie* $\mathit{Pref}(aba)$, $\mathit{Inf}(aba)$, $\mathit{Suf}(aba)$!

Die **Länge eines Wortes** kann durch die Funktion $|\cdot| : \Sigma^* \to \mathbb{N}_0$, definiert durch

$$|\varepsilon| = 0,$$
$$|wa| = |w| + 1 \text{ für } w \in \Sigma^* \text{ und } a \in \Sigma,$$

berechnet werden, die einem Wort über Σ die Anzahl seiner Buchstaben als Länge zuordnet: Das leere Wort hat die Länge 0; die Länge eines Wortes, das mindestens einen Buchstaben a sowie ein – möglicherweise leeres – Präfix w enthält, wird berechnet, indem zur Länge des Präfixes eine 1 addiert wird.

Übung 2.3 *Es sei $\Sigma = \{a, b\}$. Berechnen Sie schrittweise die Länge des Wortes aba mit der Funktion $|\cdot|$!*

Es sei $k \in \mathbb{N}_0$ und Σ ein Alphabet. Dann ist

$$\Sigma^k = \{w \in \Sigma^* : |w| = k\} \tag{2.2}$$

die Menge aller Wörter über Σ mit der Länge k, und

$$\Sigma^{\leq k} = \{w \in \Sigma^* : |w| \leq k\} \tag{2.3}$$

ist die Menge aller Wörter über Σ mit einer Länge kleiner gleich k. Offensichtlich ist

$$\Sigma^{\leq k} = \bigcup_{i=0}^{k} \Sigma^i = \Sigma^0 \cup \Sigma^1 \cup \Sigma^2 \cup \ldots \cup \Sigma^k \tag{2.4}$$

sowie

$$\Sigma^* = \bigcup_{k \in \mathbb{N}_0} \Sigma^k = \Sigma^0 \cup \Sigma^1 \cup \Sigma^2 \cup \ldots \tag{2.5}$$

Übung 2.4 *Es sei $k \in \mathbb{N}_0$, $n \in \mathbb{N}$ und Σ ein Alphabet mit $|\Sigma| = n$. Bestimmen Sie $\left|\Sigma^k\right|$ und $\left|\Sigma^{\leq k}\right|$ (vgl. Übungen 1.2 und 1.3)!*

Die Funktion $|\cdot|_\cdot : \Sigma^* \times \Sigma \to \mathbb{N}_0$ soll definiert sein durch:

$|w|_a =$ Häufigkeit des Vorkommens des Buchstaben $a \in \Sigma$ im Wort $w \in \Sigma^*$

Übung 2.5 *Geben Sie in Anlehnung an die Definition der Funktion $|\cdot|$ eine Definition für die Funktion $|\cdot|_\cdot$ an!*

Eine mögliche Definition ist

$$|\varepsilon|_b = 0 \text{ für alle } b \in \Sigma,$$
$$|wa|_b = \begin{cases} |w|_b + 1, & a = b \\ |w|_b, & a \neq b \end{cases} \quad \text{für } a, b \in \Sigma,\ w \in \Sigma^*.$$

Im leeren Wort kommt kein Buchstabe vor. Stimmt der zu zählende Buchstabe mit dem letzten Buchstaben des zu untersuchenden Wortes überein, wird er gezählt, und die Anzahl muss noch für das Wort ohne den letzten Buchstaben bestimmt werden. Ist der zu zählende Buchstabe verschieden vom letzten Buchstaben, muss nur noch im Wort ohne den letzten Buchstaben gezählt werden.

Übung 2.6 *Es sei $\Sigma = \{a, b, c\}$. Berechnen Sie schrittweise $|cabcca|_c$!*

Für ein Wort $w \in \Sigma^*$ sei $\Sigma(w)$ die Menge der Buchstaben aus Σ, die in w enthalten sind.

Übung 2.7 *Geben sie eine formale Definition für $\Sigma(w)$ an!*

Für ein total geordnetes Alphabet Σ legt die Relation $\preccurlyeq \subseteq \Sigma^* \times \Sigma^*$, definiert durch

$$\begin{aligned} v \preccurlyeq w \text{ genau dann, wenn } & |v| < |w| \\ & \text{oder } v = w \\ & \text{oder } |v| = |w| \text{ und } v = xay \text{ und } w = xby \\ & \qquad \text{mit } a, b \in \Sigma \text{ und } a < b \text{ sowie } x, y \in \Sigma^*. \end{aligned}$$

eine Ordnung auf Σ^* fest. Diese Ordnung nennen wir **längenlexikografische** oder auch **kanonische (An-) Ordnung** von Σ^*. Die Ordnung wird zunächst durch die Länge bestimmt. Bei Wörtern mit gleicher Länge bestimmen die ersten Buchstaben, an denen sie sich unterscheiden, die Ordnung.

Beispiel 2.2 *Sei $\Sigma = \{a, b, c\}$ ein geordnetes Alphabet mit $a < b < c$, dann gilt $ac \preccurlyeq aba$, $cc \preccurlyeq aaa$, $bac \preccurlyeq bac$ und $aba \preccurlyeq aca$, während $aaa \preccurlyeq ab$ und $ac \preccurlyeq ab$ nicht zutreffen.*

Die Menge Σ^* wird durch die Relation $\preccurlyeq$ total geordnet. Deshalb besitzt jede nicht leere Teilmenge von Σ^* bezüglich dieser Ordnung genau ein kleinstes Element.

Am Ende dieses Abschnitts wollen wir uns zu Übungszwecken noch ein paar Wortfunktionen überlegen, die z. B. in Textverarbeitungssystemen Anwendung finden könnten. Wir beginnen mit der Teilwortsuche. Diese Problemstellung lässt sich formal beschreiben mit der Funktion[3]

$$substr : \Sigma^* \times \Sigma^* \to \mathbb{B},$$

definiert durch

$$substr(w, v) = \begin{cases} 1, & v \in Inf(w), \\ 0, & \text{sonst.} \end{cases}$$

Diese Funktion testet, ob v ein **Teilwort** von w ist. Es gilt z. B.

$$substr(ababba, abb) = 1 \text{ sowie } substr(ababba, aa) = 0.$$

Eine Variante von $substr$ ist die Funktion $substr' : \Sigma^* \times \mathbb{N}_0 \times \mathbb{N}_0 \to \Sigma^*$, definiert durch

$$substr'(w, i, l) = \begin{cases} w\,[\,i, i+l-1\,], & i+l-1 \leq n, \\ \perp, & \text{sonst.} \end{cases}$$

$substr'(w, i, l)$ liefert den Infix von w der Länge l ab dem i-ten Buchstaben von w, falls ein solcher existiert.

Die Anwendung $tausche(w, a, b)$ der Funktion $tausche : \Sigma^* \times \Sigma \times \Sigma \to \Sigma^*$ ersetzt jedes Vorkommen des Buchstabens a im Wort w durch den Buchstaben b.

Übung 2.8 **a)** *Geben Sie eine rekursive Definition für die Funktion* $tausche$ *an!*

b) *Es sei* $\Sigma = \{1, 2, 3\}$. *Berechnen Sie schrittweise* $tausche(12322, 2, 1)$*!*

Die Funktion $|\cdot|_{\cdot}^* : \Sigma^+ \times \Sigma^+ \to \mathbb{N}_0$ sei informell definiert durch

$$|x|_y^* = \text{Anzahl, mit der } y \text{ als Infix in } x \text{ vorkommt.}$$

Übung 2.9 *Geben Sie eine formale Definition für die Funktion* $|\cdot|_{\cdot}^*$ *an!*

Für $w \in \Sigma^*$ sei $\overleftarrow{w}$ die Spiegelung von w.

Übung 2.10 *Geben Sie eine formale Definition für die Spiegelung der Wörter in* Σ^* *an!*

[3] „1“ steht für die Antwort ja, „0“ steht für $nein$.

2.3 Homomorphismen

Seien Σ_1 und Σ_2 zwei Alphabete. Eine totale Abbildung $h : \Sigma_1^* \to \Sigma_2^*$ heißt **Homomorphismus** von Σ_1^* nach Σ_2^* genau dann, wenn

$$h(vw) = h(v)h(w) \text{ für alle } v, w \in \Sigma_1^* \tag{2.6}$$

gilt.

Folgerung 2.2 *Seien Σ_1 und Σ_2 Alphabete und h ein Homomorphismus von Σ_1^* nach Σ_2^*. Dann ist*

a) $h(\varepsilon) = \varepsilon$,

b) *h bereits durch $h_{|\Sigma_1}$ festgelegt.*

Beweis **a)** *Wir nehmen an, es sei $h(\varepsilon) = w$ und $w \in \Sigma_2^+$. Dann gilt*

$$w = h(\varepsilon) = h(\varepsilon\varepsilon) = h(\varepsilon)h(\varepsilon) = ww.$$

Für $w \neq \varepsilon$ ist aber $w \neq ww$. Die Annahme führt also zu einer falschen Aussage, womit sie widerlegt ist.

b) *Sei $w = w_1 \dots w_k$ mit $w_i \in \Sigma_1$, $1 \leq i \leq k$, dann gilt wegen (2.6)*

$$h(w) = h(w_1) \dots h(w_k).$$

Es reicht also, h für die Buchstaben von Σ_1 zu definieren, denn damit ist h für alle Wörter über Σ_1 festgelegt.

Beispiel 2.3 *Im Abschnitt 1.2.2 haben wir bei der Kompression von Bitfolgen die Abbildung $\eta : \{0, 1, [,]\} \to \mathbb{B}^2$, definiert durch*

$$\eta(0) = 00, \;\; \eta(1) = 11, \;\; \eta([) = 10, \;\; \eta(]) = 01,$$

verwendet. Die dort eingeführte Erweiterung $\eta^* : \{0, 1, [,]\}^* \to (\mathbb{B}^2)^*$, definiert durch

$$\begin{aligned} \eta^*(\varepsilon) &= \varepsilon, \\ \eta^*(bw) &= \eta(b) \circ \eta^*(w) \text{ für } b \in \{0, 1, [,]\} \text{ und } w \in \{0, 1, [,]\}^*, \end{aligned}$$

ist ein Homomorphismus.

Ist ein Homomorphismus h bijektiv, dann nennen wir ihn **Isomorphismus**. Die Abbildung η^* aus dem obigen Beispiel ist ein Isomorphismus von $\{0, 1, [,]\}^*$ nach $(\mathbb{B}^2)^*$.

2.4 Formale Sprachen

Sei Σ ein Alphabet, dann nennen wir jede Teilmenge $L \subseteq \Sigma^*$ eine **(formale) Sprache** über Σ. Die Potenzmenge 2^{Σ^*} von Σ ist die Menge aller Sprachen über Σ.

Sprachen sind also Mengen von Wörtern und können daher mit den üblichen Mengenoperationen wie Vereinigung, Durchschnitt und Differenz miteinander verknüpft werden. Wir wollen für Sprachen eine weitere Verknüpfung einführen, die wir schon von Wörtern kennen: die **Konkatenation**. Seien L_1 und L_2 zwei Sprachen über Σ, dann ist die Konkatenation $L_1 \circ L_2$ von L_1 und L_2 definiert durch

$$L_1 \circ L_2 = \{vw \mid v \in L_1, w \in L_2\}\,.$$

Es werden also alle Wörter aus L_1 mit allen Wörtern aus L_2 konkateniert. Gelegentlich lassen wir wie bei der Konkatenation von Wörtern auch bei der Konkatenation von Sprachen das Konkatenationssymbol $\circ$ weg, d. h., anstelle von $L_1 \circ L_2$ schreiben wir L_1L_2. Seien $L_1 = \{\varepsilon, ab, abb\}$ und $L_2 = \{b, ba\}$ zwei Sprachen über dem Alphabet $\Sigma = \{a, b\}$, dann ist

$$L_1 \circ L_2 = \{b, ba, abb, abba, abbb, abbba\}$$

sowie

$$L_2 \circ L_1 = \{b, bab, babb, ba, baab, baabb\}\,.$$

Folgerung 2.3 *Allgemein gilt: Falls $\varepsilon \in L_1$ ist, dann ist $L_2 \subseteq L_1 \circ L_2$, bzw. umgekehrt, falls $\varepsilon \in L_2$ ist, dann ist $L_1 \subseteq L_1 \circ L_2$.*

Die **n-te Potenz einer Sprache** $L \subseteq \Sigma^*$ ist festgelegt durch

$$\begin{aligned} L^0 &= \{\varepsilon\}, \\ L^{n+1} &= L^n \circ L \text{ für } n \geq 0. \end{aligned} \tag{2.7}$$

Sei $L = \{a, ab\}$, dann ist

$$\begin{aligned} L^0 &= \{\varepsilon\}, \\ L^1 &= L^0 \circ L = \{\varepsilon\} \circ \{a, ab\} = \{a, ab\} = L, \\ L^2 &= L^1 \circ L = \{a, ab\} \circ \{a, ab\} = \{a^2, a^2b, aba, (ab)^2\}, \\ L^3 &= L^2 \circ L = \{a^2, a^2b, aba, (ab)^2\} \circ \{a, ab\}, \\ &= \{a^3, a^3b, a^2ba, a^2bab, aba^2, aba^2b, (ab)^2a, (ab)^3\}, \\ &\vdots\ . \end{aligned}$$

Bemerkung 2.2 *Die Struktur $\left(2^{\Sigma^*}, \circ, \{\varepsilon\}\right)$, d. h., die Menge aller Sprachen über einem Alphabet Σ, bildet mit der Konkantenationsoperation für Sprachen ein Monoid (vgl. Bemerkung 2.1).*

Das **Kleene-Stern-Produkt**[4] (auch **Kleenesche Hülle**) L^* einer Sprache L ist die Vereinigung aller ihrer Potenzen L^n, $n \geq 0$:

$$L^* = \bigcup_{n \geq 0} L^n = L^0 \cup L^1 \cup L^2 \cup L^3 \cup \ldots$$

Die **positive Hülle** von L ist

$$L^+ = \bigcup_{n \geq 1} L^n = L^1 \cup L^2 \cup L^3 \cup \ldots \quad .$$

Übung 2.11 *Geben Sie $\emptyset^*$, $\emptyset^+$, $\{\varepsilon\}^*$ und $\{\varepsilon\}^+$ an!*

Sei Σ ein Alphabet und $f : \Sigma \to 2^{\Sigma^*}$ eine totale Abbildung, die jedem Buchstaben $a \in \Sigma$ eine formale Sprache $f(a) = L_a \subseteq \Sigma^*$ zuordnet. Wir definieren die **f-Substitution $sub_f(w)$ eines Wortes** $w = w_1 w_2 \ldots w_n$, $w_i \in \Sigma$, $1 \leq i \leq n$, $n \geq 0$, durch

$$sub_f(w) = \begin{cases} \emptyset, & n = 0, \\ \prod_{i=1}^{n} f(w_i), & n \geq 1. \end{cases} \tag{2.8}$$

Die f-Substitution $sub_f(w)$ ersetzt in einem nicht leeren Wort w jeden Buchstaben w_i durch die Sprache $f(w_i)$, $1 \leq i \leq n$, und konkateniert diese Sprachen.

Die **f-Substitution $SUB_f(L)$ einer Sprache** $L \subseteq \Sigma^*$ ist definiert durch

$$SUB_f(L) = \bigcup_{w \in L} sub_f(w) \tag{2.9}$$

Jedes Wort aus L wird also f-substituiert, und die entstehenden Sprachen werden vereinigt.

Beispiel 2.4 *Sei $\Sigma = \{0, 1, a, b, c\}$, $L = \{0^n c 1^n b \mid n \geq 0\}$ und $f : \Sigma \to 2^{\Sigma^*}$, definiert durch*

$$f(0) = L_0 = \{ab\}, \quad f(1) = L_1 = \left\{aab^k \mid k \geq 1\right\},$$
$$f(a) = L_a = \{\varepsilon\}, \quad f(b) = L_b = \{\varepsilon\}. \quad f(c) = L_c = \{c\}.$$

[4] Benannt nach Stephen C. Kleene (1909–1998), amerikanischer Mathematiker und Logiker, der fundamentale Beiträge zur Logik und zu theoretischen Grundlagen der Informatik geliefert hat.

Dann gilt z. B.

$$\begin{aligned} sub_f(0^2c1^2b) &= f(0) \circ f(0) \circ f(c) \circ f(1) \circ f(1) \circ f(b) \\ &= \{ab\} \circ \{ab\} \circ \{c\} \circ \left\{aab^k \mid k \geq 1\right\} \circ \left\{aab^k \mid k \geq 1\right\} \circ \{\varepsilon\} \\ &= \left\{(ab)^2c(aab^raab^s) \mid r, s \geq 1\right\}. \end{aligned}$$

Die f-Substitution von L ist

$$SUB_f(L) = \{(ab)^nc(aab^{r_1}aab^{r_2} \ldots aab^{r_n}) \mid n \geq 0,\ r_i \geq 1,\ 1 \leq i \leq n\}.$$

Mithilfe der Substitutionsoperation kann man Mengenverknüpfungen darstellen. Seien A, B, C Sprachen über dem Alphabet Σ mit $a, b \in \Sigma$.

Für die Vereinigung wählen wir die Funktion $f : \{a, b\} \to 2^{\Sigma^*}$, definiert durch $f(a) = L_a = A$ und $f(b) = L_b = B$, sowie die Sprache $L = \{a, b\}$. Damit gilt

$$SUB_f(L) = sub_f(a) \cup sub_f(b) = f(a) \cup f(b) = A \cup B.$$

Für die Konkatenation wählen wir ebenfalls die Funktion $f : \{a, b\} \to 2^{\Sigma^*}$, definiert durch $f(a) = L_a = A$ und $f(b) = L_b = B$, sowie die Sprache $L = \{ab\}$. Damit gilt

$$SUB_f(\{ab\}) = sub_f(ab) = f(a) \circ f(b) = A \circ B.$$

Für das Kleene-Stern-Produkt wählen wir die Funktion $f : \{a\} \to 2^{\Sigma^*}$, definiert durch $f(a) = L_a = C$, sowie die Sprache $L = \{a\}^* = \{a^n \mid n \in \mathbb{N}_0\}$. Damit gilt

$$\begin{aligned} SUB_f(\{a\}^*) &= \bigcup_{n \geq 0} sub_f(a^n) \\ &= sub_f(a^0) \cup sub_f(a^1) \cup sub_f(a^2) \cup \ldots \\ &= \{\varepsilon\} \cup f(a) \cup (f(a) \circ f(a)) \cup \ldots \\ &= C^0 \cup C^1 \cup (C^1 \circ C^1) \cup \ldots \\ &= C^0 \cup C^1 \cup C^2 \cup \ldots \\ &= \bigcup_{n \geq 0} C^n \\ &= C^*. \end{aligned}$$

Aus diesen Darstellungen der Sprachverknüpfungen durch geeignete Substitutionen kann man möglicherweise Beweise über Abschlusseigenschaften von Sprachklassen vereinfachen. Sei etwa $\mathcal{C}_\Sigma$ eine durch bestimmte Eigenschaften festgelegte Klasse von Sprachen über dem Alphabet Σ. Wenn man zeigen möchte, dass $\mathcal{C}_\Sigma$ abgeschlossen ist gegen Vereinigung, Konkatenation und Kleene-Stern-Produkt, reicht es, die Abgeschlossenheit gegenüber Substitution zu beweisen, denn die Abgeschlossenheit gegenüber den drei Operationen folgt dann daraus unmittelbar wegen der obigen Überlegungen.

2.5 Präfixfreie Sprachen

In späteren Kapiteln spielt die Präfixfreiheit von Sprachen eine wichtige Rolle. Eine Sprache $L \subseteq \Sigma^+$ heißt **präfixfrei** (hat die **Präfixeigenschaft**) genau dann, wenn für alle Wörter $w \in L$ gilt: Ist x ein echter Präfix von w, dann ist $x \notin L$. Von einem Wort $w \in L$ darf also kein echtes Präfix Element der Sprache sein.

Beispiel 2.5 *Betrachten wir als Beispiele die Sprachen*

$$L_1 = \{a^n b^n \mid n \geq 0\} \text{ und } L_2 = \left\{w \in \{a,b\}^* : |w|_a = |w|_b\right\}.$$

Die Sprache L_1 hat die Präfixeigenschaft, denn für jedes Wort $w = a^n b^n$ sind alle echten Präfixe $x = a^n b^i$ mit $n > i$ sowie $x = a^j$ für $j \geq 0$ keine Wörter von L_1. Die Sprache L_2 ist nicht präfixfrei, denn z. B. ist $w = abab \in L_2$ und $x = ab \in L_2$.

Der folgende Satz gibt eine hinreichende und notwendige Eigenschaft für die Präfixfreiheit von Sprachen an.

Satz 2.1 *Eine Sprache $L \subseteq \Sigma^*$ ist präfixfrei genau dann, wenn $L \cap (L \circ \Sigma^+) = \emptyset$ ist.*

Beweis *„⇒“: Wir nehmen an, es sei $L \cap (L \circ \Sigma^+) \neq \emptyset$. Dann gibt es ein Wort $w \in L \cap (L \circ \Sigma^+)$, d. h., es ist $w \in L$ und $w \in (L \circ \Sigma^+)$. Aus der zweiten Eigenschaft folgt, dass es ein $x \in L$ und ein $y \in \Sigma^+$ geben muss mit $w = xy$. Somit gibt es also einen echten Präfix x von w, der in L enthalten ist. Das bedeutet aber einen Widerspruch dazu, dass L präfixfrei ist. Damit ist unsere Annahme falsch, d. h., wenn L präfixfrei ist, dann ist $L \cap (L \circ \Sigma^+) = \emptyset$.*

„⇐“: Sei nun $L \cap (L \circ \Sigma^+) = \emptyset$. Wir nehmen nun an, dass L nicht präfixfrei ist. Es gibt also ein Wort $w \in L$ mit einem echten Präfix $x \in L$, d. h., es gibt ein $y \in \Sigma^+$ mit $w = xy$. Damit gilt, dass $w \in L \cap (L \circ \Sigma^+)$, d. h., dass $L \cap (L \circ \Sigma^+) \neq \emptyset$ ist, was ein Widerspruch zur Voraussetzung $L \cap (L \circ \Sigma^+) = \emptyset$ ist. Unsere Annahme ist also falsch, L muss also präfixfrei sein.

Damit haben wir insgesamt die Behauptung des Satzes bewiesen.

Bemerkung 2.3 **a)** *Sei $L \subseteq \Sigma^*$ eine Sprache und $\& \notin \Sigma$, dann ist die Sprache $L_\& = L \circ \{\&\}$ präfixfrei.*

b) *Eine präfixfreie Codierung der natürlichen Zahlen ist z. B. durch die Abbildung $\sigma : \mathbb{N}_0 \to \mathbb{B}^*$, definiert durch $\sigma(n) = 1^n 0$, gegeben.*

Übung 2.12 *Geben Sie eine präfixfreie Codierung von Bitfolgen ohne Verwendung eines Sonderzeichens an!*

Sei $\Sigma = \{a_1, a_2, \ldots, a_n\}$ ein geordnetes Alphabet, $\preccurlyeq$ die lexikografische Ordnung auf Σ^* und L eine nicht leere Sprache über Σ. Dann ist $\min(L)$ **das kleinste Wort in** L:

$$\min(L) = w \text{ genau dann, wenn } w \preccurlyeq x \text{ für alle } x \in L$$

Für alle $v \in \Sigma^*$ ist

$$SUC(v) = \{\, w \in \Sigma^* \mid v \preccurlyeq w \,\}$$

die Sprache, die alle Nachfolger von v enthält, und die **Nachfolgerfunktion**

$$suc(v) = \min(SUC(v))$$

bestimmt den Nachfolger von v.

2.6 Codierungen von Alphabeten und Wörtern über $\mathbb{N}_0$ und $\mathbb{B}$

Sei $\Sigma = \{a_1, \ldots, a_n\}$ ein nicht leeres, geordnetes Alphabet. Dann legt die Funktion $\tau_\Sigma : \Sigma^* \to \mathbb{N}_0$, definiert durch

$$\begin{aligned} \tau_\Sigma(\varepsilon) &= 0, \\ \tau_\Sigma(a_i) &= i,\ 1 \le i \le n, \\ \tau_\Sigma(wa) &= n \cdot \tau_\Sigma(w) + \tau_\Sigma(a),\ a \in \Sigma,\ w \in \Sigma^*, \end{aligned}$$

eine **Codierung** der Wörter von Σ^* durch natürliche Zahlen fest. Außerdem stellt τ_Σ eine **Abzählung** aller Wörter von Σ^* dar: Das Wort w kommt genau dann an der i-ten Stelle in der lexikografischen Anordnung der Wörter von Σ^* vor, wenn $\tau_\Sigma(w) = i$ ist.

Beispiel 2.6 *Sei* $\Sigma = \{a, b, c\}$. *Dann ist*

$$\begin{aligned} \tau_\Sigma(abbc) &= 3 \cdot \tau_\Sigma(abb) + \tau_\Sigma(c) \\ &= 3 \cdot (3 \cdot \tau_\Sigma(ab) + \tau_\Sigma(b)) + 3 \\ &= 3 \cdot (3 \cdot (3 \cdot \tau_\Sigma(a) + \tau_\Sigma(b)) + 2) + 3 \\ &= 3 \cdot (3 \cdot (3 \cdot (3 \cdot \tau_\Sigma(\varepsilon) + \tau_\Sigma(a)) + 2) + 2) + 3 \\ &= 3 \cdot (3 \cdot (3 \cdot 0 + 1) + 2) + 2) + 3 \\ &= 54. \end{aligned}$$

Das Wort $abbc$ *steht in der lexikografischen Anordnung der Wörter über* Σ *an der* 54*-ten Stelle.*

Übung 2.13 *Es sei $\Sigma = \{a_1, \ldots, a_n\}$ ein geordnetes Alphabet.*

a) *Die Funktion $\tau'_\Sigma : \Sigma^* \to \mathbb{N}_0$ sei für $w = w_1 w_2 \ldots w_k$ mit $w_i \in \Sigma$, $1 \leq i \leq k$, definiert durch*

$$\tau'_\Sigma(w) = \tau'_\Sigma(w_1 w_2 \ldots w_k) = \sum_{i=1}^{k} \tau_\Sigma(w_i) \cdot n^{k-i}.$$

Außerdem sei $\tau'_\Sigma(\varepsilon) = \tau_\Sigma(\varepsilon) = \varepsilon$.

Zeigen Sie, dass $\tau_\Sigma(w) = \tau'_\Sigma(w)$ für alle $w \in \Sigma^+$ gilt!

b) *Es sei $\Sigma = \mathbb{B}$. Zeigen Sie, dass dann*

$$\tau_\mathbb{B}(w_1 \ldots w_k) = wert(1 w_1 \ldots w_k) - 1$$

gilt!

Für jedes Alphabet Σ sind die Funktionen τ_Σ und ihre Umkehrfunktionen $\nu_\Sigma = \tau_\Sigma^{-1}$ bijektiv. Es ist $\nu_\Sigma(i) = w$ genau dann, wenn w das i-te Wort in der lexikografischen Anordnung von Σ^* ist. Für den Fall $\Sigma = \mathbb{B}$ ergibt sich die folgende lexikografische Anordnung der Bitfolgen als Paare $(w, \tau_\mathbb{B}(w)) \in \mathbb{B}^* \times \mathbb{N}_0$:

$$(\varepsilon, 0), (0, 1), (1, 2), (00, 3), (01, 4), (10, 5), (11, 6), (000, 7), \ldots \tag{2.10}$$

Es ist z. B. $\nu_\mathbb{B}(62) = 11111$ die 62.-Bitfolge in dieser Anordnung. Die Abbildung $\nu_\mathbb{B} : \mathbb{N}_0 \to \mathbb{B}^*$ stellt also neben der üblichen Dualcodierung $dual$ eine weitere Codierung natürlicher Zahlen dar. Ihr Zusammenhang ergibt sich aus der Übung 2.13 b).

In späteren Kapiteln werden wir nur Sprachen über dem Alphabet $\mathbb{B} = \{0, 1\}$ betrachten. Dass das keine prinzipielle Einschränkung ist, zeigt die folgende Überlegung: Jeder Buchstabe eines Alphabets $\Sigma = \{a_0, \ldots, a_{n-1}\}$ mit $n \geq 2$ Symbolen kann durch ein Wort der Länge

$$\ell(\Sigma) = \lceil \log n \rceil = \lceil \log |\Sigma| \rceil$$

über dem Alphabet $\mathbb{B}$ codiert werden, indem z. B. a_i durch die Dualdarstellung von i repräsentiert wird, die, falls nötig, nach links mit Nullen auf die Länge $\ell(\Sigma)$ aufgefüllt wird. Wir können dies formal durch die Funktion

$$bin_\Sigma : \Sigma \to \mathbb{B}^{\ell(\Sigma)},$$

definiert durch

$$bin_\Sigma(a_i) = x_{\ell(\Sigma)-1} \ldots x_0 \tag{2.11}$$

mit

$$i = \sum_{i=0}^{\ell(\Sigma)-1} x_i \cdot 2^i,$$

beschreiben. Wir nennen $bin_\Sigma(a)$ die **Binärdarstellung** von $a \in \Sigma$.

Beispiel 2.7 *Für das Alphabet $\Sigma = \{a, b, c\}$ ergibt sich z. B. $bin_\Sigma(a) = 00$, $bin_\Sigma(b) = 01$, $bin_\Sigma(c) = 10$ und damit das Alphabet $bin_\Sigma(\Sigma) = \{00, 01, 10\}$.*

Wir erweitern die Codierung bin_Σ von Symbolen auf Wörter. Dazu sei

$$bin_\Sigma{}^* : \Sigma^* \to \left(\mathbb{B}^{\ell(\Sigma)}\right)^*$$

definiert durch

$$\begin{aligned} bin_\Sigma{}^*(\varepsilon) &= \varepsilon, \\ bin_\Sigma{}^*(wa) &= bin_\Sigma{}^*(w) \circ bin_\Sigma(a) \text{ für } w \in \Sigma^* \text{ und } a \in \Sigma. \end{aligned} \tag{2.12}$$

Beispiel 2.8 *Für das Beispiel (2.6) ergibt sich mit der Festlegung von Beispiel 2.7*

$$\begin{aligned} bin_\Sigma{}^*(abbc) &= bin_\Sigma{}^*(abb) \circ bin_\Sigma(c) = bin_\Sigma{}^*(abb) \circ 10 \\ &= bin_\Sigma{}^*(ab) \circ bin_\Sigma(b) \circ 10 = bin_\Sigma{}^*(ab) \circ 01 \circ 10 \\ &= bin_\Sigma{}^*(a) \circ bin_\Sigma(b) \circ 0110 = bin_\Sigma{}^*(a) \circ 01 \circ 0110 \\ &= bin_\Sigma{}^*(\varepsilon) \circ bin_\Sigma(a) \circ 010110 = bin_\Sigma{}^*(\varepsilon) \circ 00 \circ 010110 \\ &= \varepsilon \circ 00010110 \\ &= 00010110. \end{aligned}$$

Das Wort $abbc \in \Sigma^*$ wird also durch das Wort $00010110 \in bin_\Sigma(\Sigma)^*$ binär codiert.

Da die Binärdarstellungen $bin_\Sigma(a)$ der Buchstaben $a \in \Sigma$ alle dieselbe Länge haben, folgt, dass $bin_\Sigma{}^*$ injektiv ist, womit die eindeutige Decodierung gewährleistet ist.

Wir wollen im Folgenden aus schreibtechnischen Gründen nicht mehr zwischen bin_Σ und $bin_\Sigma{}^*$ unterscheiden und beide Funktionen mit bin_Σ notieren.

Übung 2.14 *Codieren Sie das Alphabet $\Sigma = \{a, b, c, \ldots, z\}$ der deutschen Kleinbuchstaben binär!*

Für die binäre Codierung $bin_\Sigma(w)$ eines Wortes w über dem Alphabet Σ ergibt sich die Länge $|bin_\Sigma(w)| = |w| \cdot \ell(\Sigma)$. Die Dualcodierungen der Wörter w eines Alphabets mit n Eementen sind also um den Faktor $\lceil \log n \rceil$ länger als w selbst. Dieser Faktor ist konstant durch die Anzahl der Elemente von Σ gegeben und damit unabhängig von der Länge der Wörter w.

Übung 2.15 *Überlegen Sie, warum einelementige (unäre) Alphabete und damit die „Bierdeckelcodierung" von Zahlen ungeeignet sind!*

Die Menge $\mathbb{N}_0$ der natürlichen Zahlen kann als eine Sprache angesehen werden: Natürliche Zahlen sind Zeichenfolgen, die über dem Alphabet $N = \{0, 1, \ldots, 9\}$ gebildet werden. Wenden wir auf dieses Alphabet die Codierung bin_N an, erhalten wir

$$bin_N(N) = \{0000, 0001, 0010, 0011, 0100, 0101, 0110, 0111, 1000, 1001\}. \quad (2.13)$$

Es gilt aber nicht $N^+ = \mathbb{N}_0$, denn üblicherweise lassen wir führende Nullen weg. Wir schreiben beispielsweise nicht 00123, sondern 123. Die Menge der Wörter über N, die die natürlichen Zahlen ohne führende Nullen darstellen, ist $\{0\} \cup (\{1, 2, \ldots, 9\} \circ N^*)$.

Üblicherweise erfolgt die Codierung von natürlichen Zahlen durch Dualzahlen ebenfalls ohne führende Nullen (siehe Abschnitt 1.2.2): Jede Zahl $z \in \mathbb{N}_0$ lässt sich eindeutig darstellen durch eine Folge

$$\begin{aligned} dual(z) = x = x_{n-1} \ldots x_0 \in \mathbb{B}^n \text{ mit } n \in \mathbb{N}, \\ x_{n-1} \neq 0 \text{ für } n \geq 2, \\ z = wert(x) = \sum_{i=0}^{n-1} x_i \cdot 2^i \text{ und} \\ n = \lfloor \log z \rfloor + 1. \end{aligned} \quad (2.14)$$

Bemerkung 2.4 *Aus den obigen Überlegungen folgt:*

- *Mithilfe der Bijektionen τ_Σ können Betrachtungen über Zeichenketten und Sprachen über Alphabete Σ simuliert werden durch Betrachtungen über natürliche Zahlen und Teilmengen natürlicher Zahlen;*
- *mithilfe der Bijektionen $\nu = \tau_\Sigma^{-1}$ können Betrachtungen über natürliche Zahlen und Teilmengen natürlicher Zahlen simuliert werden durch Betrachtungen über Zeichenketten und Sprachen über Alphabete Σ;*
- *mithilfe der injektiven Codierung bin_Σ können Betrachtungen über Zeichenketten und Sprachen über Alphabete Σ simuliert werden durch Betrachtungen über Bitfolgen und Teilmengen von Bitfolgen.*

Wir können also im Folgenden „je nach Gusto" Betrachtungen über Elemente oder Teilmengen von Σ^*, $\mathbb{B}^*$ oder $\mathbb{N}_0$ anstellen, diese gelten gleichermaßen für die entsprechenden Codierungen in den jeweils anderen Mengen.

2.7 Entscheidbarkeit von Sprachen und Mengen

Welche Aufgaben hat ein Compiler einer Programmiersprache? Neben vielen weiteren Funktionalitäten hat ein Compiler die beiden folgenden wesentlichen Aufgaben:

(1) Überprüfung, ob ein eingegebenes Wort ein syntaktisch korrektes Programm ist,

(2) Übersetzung des syntaktisch korrekten Programms in eine auf dem vorliegenden Rechnersystem ausführbare Form.

Wir werden uns im Folgenden nur mit dem Aspekt (1) beschäftigen. Abstrakt betrachtet, muss der Compiler beim Syntaxcheck entscheiden, ob ein Wort w über einem Alphabet Σ zu einer Sprache L gehört oder nicht. Er muss die sogenannte charakteristische Funktion von L berechnen.

Für eine Sprache $L \subseteq \Sigma^*$ heißt die Funktion

$$\chi_L : \Sigma^* \to \mathbb{B},$$

definiert durch

$$\chi_L(w) = \begin{cases} 1, & w \in L, \\ 0, & w \notin L, \end{cases} \tag{2.15}$$

charakteristische Funktion von L.

Wegen der Bemerkung 2.4 können Sprachen als Teilmengen natürlicher Zahlen codiert werden. Deshalb werden wir anstelle der charakteristischen Funktionen von Sprachen charakteristische Funktionen von Mengen $A \subseteq \mathbb{N}_0$ betrachten: Für eine Menge $A \subseteq \mathbb{N}_0$ heißt die Funktion

$$\chi_A : \mathbb{N}_0 \to \mathbb{B},$$

definiert durch

$$\chi_A(x) = \begin{cases} 1, & x \in A, \\ 0, & x \notin A, \end{cases} \tag{2.16}$$

charakteristische Funktion von A.

Neben der charakteristischen Funktion benötigen wir noch eine Variante, die sogenannte semi-charakteristische Funktion, die wir nur für Mengen $A \subseteq \mathbb{N}_0$ angeben: Die Funktion

$$\chi'_A : \mathbb{N}_0 \to \mathbb{B},$$

definiert durch

$$\chi'_A(x) = \begin{cases} 1, & x \in A, \\ \perp, & x \notin A, \end{cases} \tag{2.17}$$

heißt **semi-charakteristische Funktion von A.**

Diese beiden Funktionen sind der Ausgangspunkt für die Definition von zwei Klassen von Mengen, mit denen wir uns im folgenden Kapitel noch näher beschäftigen werden: die Klasse der entscheidbaren sowie die Klasse der semi-entscheidbaren Mengen.

Definition 2.1 **a)** *Eine Menge $A \subseteq \mathbb{N}_0$ heißt* **entscheidbar** *genau dann, wenn ihre charakteristische Funktion χ_A berechenbar ist.*

R *sei die* **Klasse der entscheidbaren Mengen**.

b) *Eine Menge $A \subseteq \mathbb{N}_0$ heißt* **semi-entscheidbar** *genau dann, wenn ihre semi-charakteristische Funktion χ'_A berechenbar ist.*

RE *sei die* **Klasse der semi-entscheidbaren Mengen**.

Die Bezeichnung R steht für *recursive*, die englische Bezeichnung für berechenbar, und RE steht für *recursive enumerable*, die englische Bezeichnung für rekursiv aufzählbar. Warum semi-entscheidbare Mengen auch rekursiv aufzählbar genannt werden, wird sich im Abschnitt 3.4 herausstellen.

In Definition 2.1 wird die bisher nicht definierte Eigenschaft der Berechenbarkeit verwendet. Die beiden Funktionen χ und χ' sollen zunächst intuitiv als berechenbar bezeichnet werden, wenn es einen Algorithmus (ein Programm) gibt, der (das) sie berechnet. Die Begriffe *Algorithmus* und *Berechenbarkeit* werden wir im nächsten Kapitel formal präzisieren.

Die Frage nach der Berechenbarkeit von Funktionen $f : \mathbb{N}_0 \to \mathbb{N}_0$, die im folgenden Kapitel der Ausgangspunkt der Betrachtungen sein wird, kann mithilfe des Entscheidbarkeitsbegriffes von Mengen definiert werden. Dazu führen wir für eine Funktion $f : \mathbb{N}_0 \to \mathbb{N}_0$ die Menge

$$G_f = \{(x, f(x)) \mid x \in \mathbb{N}_0\} \tag{2.18}$$

ein und nennen diese den **Graph der Funktion f**.

Beispiel 2.9 *In (2.10) sind Elemente von $G_{\tau_{\mathbb{B}}}$ aufgelistet.*

Definition 2.2 *Eine Funktion $f : \mathbb{N}_0 \to \mathbb{N}_0$ heißt* **berechenbar** *genau dann, wenn G_f semi-entscheidbar, d. h., wenn χ'_{G_f} berechenbar ist.*

Es reicht also, den Begriff der Berechenbarkeit für charakteristische bzw.für semi-charakteristische Funktionen von Mengen formal zu definieren, weil damit implizit die Berechenbarkeit allgemein für Funktionen festgelegt wird.

2.8 Die Cantorsche k-Tupel-Funktion

Bei dem gerade angesprochenen Thema Berechenbarkeit geht es darum, eine mathematisch präzise Definition für die Berechenbarkeit von Funktionen $f : \mathbb{N}_0^k \to \mathbb{N}_0$, $k \geq 1$, und damit formale Definitionen für die oft nur informell festgelegten Begriffe *Algorithmus* und *Programm* anzugeben. Durch geeignete Codierung können wir uns auf Funktionen $f : \mathbb{N}_0 \to \mathbb{N}_0$ beschränken, denn die k-Tupel von $\mathbb{N}_0^k$ lassen sich eineindeutig durch natürliche Zahlen codieren. Eine Möglichkeit für solche Codierungen sind die Cantorschen k-Tupelfunktionen.[5]

Wir beginnen mit dem Fall $k = 2$, d. h., wir wollen die Menge $\mathbb{N}_0^2$ eineindeutig durch die Menge $\mathbb{N}_0$ codieren. Dazu betrachten wir die folgende Matrix:

	1	2	3	4	…	n	…
1	1	2	4	7			
		↙	↙	↙			
2	3	5	8				
		↙	↙				
3	6	9					
		↙					
4	10						
⋮							
m							
⋮							

Die Paare $(m, n) \in \mathbb{N}_0 \times \mathbb{N}_0$ werden in Pfeilrichtung nummeriert: Das Paar $(2, 3)$ bekommt z. B. die Nummer 8, und das Paar $(4, 1)$ bekommt die Nummer 10.

Wir wollen nun die bijektive Abbildung $c : \mathbb{N}_0 \times \mathbb{N}_0 \to \mathbb{N}_0$ angeben, die diese Matrix darstellt. In der ersten Spalte stehen die Nummern der Paare $(m, 1)$, $m \geq 1$. Diese ergeben sich durch

$$c(m, 1) = \sum_{i=1}^{m} i = \frac{m(m+1)}{2}. \tag{2.19}$$

Für $n \geq 2$ gilt, dass sich die Nummer an der Stelle (m, n) in der Matrix ergibt, indem man die Nummer an der Stelle $(m + 1, n - 1)$ (eine Zeile weiter und eine Spalte vorher) um 1 vermindert. Für c gilt also

$$c(m, n) = c(m + 1, n - 1) - 1, \ n \geq 2.$$

[5] Georg Cantor (1845–1918) war ein deutscher Mathematiker. Er gilt als Begründer der Mengenlehre und lieferte dadurch eine Basis für ein neues Verständnis der Mathematik. Insbesonders beschäftige er sich mit der Mächtigkeit und dem Vergleich der Mächtigkeiten von Mengen. Dabei verwendete er das Prinzip der Diagonalisierung (siehe oben sowie Kapitel 6).

Daraus ergibt sich

$$\begin{aligned} c(m,n) &= c(m+1,n-1)-1 \\ &= c(m+2,n-2)-2 \\ &\vdots \\ &= c(m+n-1,1)-(n-1),\ n \geq 2. \end{aligned}$$

Mit (2.19) folgt hieraus

$$c(m,n) = \sum_{i=1}^{m+n-1} i-(n-1) = \frac{(m+n-1)(m+n)}{2}-(n-1).$$

Man kann zeigen, dass c bijektiv ist; c stellt quasi eine Abzählung von $\mathbb{N}_0 \times \mathbb{N}_0$ dar. Abzählverfahren dieser Art werden **Cantors erstes Diagonalargument** genannt.

Die Funktion c ist der Ausgangspunkt für die rekursive Definition der sogenannten **Cantorschen k-Tupelfunktionen**

$$\langle\cdot\rangle_k : \mathbb{N}_0^k \to \mathbb{N}_0,\ k \geq 1,$$

die rekursiv wie folgt festgelegt werden können:

$$\begin{aligned} \langle i \rangle_1 &= i \\ \langle i,j \rangle_2 &= c(i,j+2) = \frac{(i+j+1)(i+j+2)}{2}-(j+1) \\ \langle i_1,\ldots,i_k,i_{k+1} \rangle_{k+1} &= \langle \langle i_1,\ldots,i_k \rangle_k, i_{k+1} \rangle_2,\ k \geq 2 \end{aligned}$$

Man kann zeigen, dass $\langle\cdot\rangle_k$ bijektiv ist für alle $k \geq 1$. Die Funktion $\langle\cdot\rangle_k$ stellt eine eineindeutige Nummerierung von $\mathbb{N}_0^k$ dar.

Mit $\left(\langle\cdot\rangle_k^{-1}\right)_i$, $1 \leq i \leq k$, notieren wir die k Umkehrfunktionen zu $\langle\cdot\rangle_k$, d. h., es ist

$$x_i = \left(\langle z \rangle_k^{-1}\right)_i \text{ genau dann, wenn } \langle x_1,\ldots,x_k \rangle_k = z \text{ ist.}$$

Wir schreiben im Folgenden, wenn die Stelligkeit k aus dem Zusammenhang klar ist, insbesondere im Fall $k=2$, $\langle\cdot\rangle$ anstelle von $\langle\cdot\rangle_k$.

Bemerkung 2.5 *Mithilfe der Cantorschen k-Tupelfunktionen können wir uns bei Bedarf auf einstellige Funktionen beschränken, indem wir für eine eigentlich k-stellige Funktion $f : \mathbb{N}_0^k \to \mathbb{N}_0$ nicht $y = f(x_1,\ldots,x_k)$, sondern $y = f\langle x_1,\ldots,x_k \rangle$ schreiben.*

Mithilfe der Cantorschen k-Tupelfunktionen können wir auch den in Definition 2.2 eingeführten Begriff der Berechenbarkeit von Funktionen $f : \mathbb{N}_0 \to \mathbb{N}_0$ auf k-stellige Funktionen erweitern: Für eine Funktion $f : \mathbb{N}_0^k \to \mathbb{N}_0$ sei

$$G_f = \{\langle \langle x_1,\ldots,x_k \rangle, f(x_1,\ldots,x_k) \rangle \mid x_i \in \mathbb{N}_0,\ 1 \leq i \leq k\} \tag{2.20}$$

der **Graph von f**.

Definition 2.3 *Eine Funktion $f : \mathbb{N}_0^k \to \mathbb{N}_0$ heißt berechenbar genau dann, wenn G_f semi-entscheidbar ist.*

Die Cantorsche Paarungsfunktion kann verwendet werden, um die Menge $\mathbb{Z}$ der ganzen Zahlen sowie die Menge $\mathbb{Q}$ der rationalen Zahlen zu nummerieren.

Übung 2.16 **a)** *Geben Sie eine Nummerierung für $\mathbb{Z}$ an!*

b) *Geben Sie eine Nummerierung der Menge aller Brüche an!*

Im Unterschied zu der Menge $\mathbb{Q}$ der rationalen Zahlen kann die Menge $\mathbb{R}$ der reellen Zahlen nicht nummeriert werden. Mithilfe einer Diagonalisierung kann gezeigt werden, dass das offene Intervall $I = \{x \in \mathbb{R} \mid 0 < x < 1\}$ der reellen Zahlen zwischen Null und Eins nicht nummeriert werden kann, woraus folgt, dass auch $\mathbb{R}$ nicht nummeriert werden kann. Dazu nehmen wir an, dass I nummeriert werden kann. Jedes Element aus I lässt sich als unendlicher Dezimalbruch darstellen. Eine Nummerierung kann wie folgt als Tabelle dargestellt werden (dabei lassen wir die 0 vor dem Komma und das Komma weg):

$$\begin{aligned} x_1 &= x_{11}x_{12}\ldots x_{1i}\ldots \\ x_2 &= x_{21}x_{22}\ldots x_{2i}\ldots \\ &\vdots \\ x_i &= x_{i1}x_{i2}\ldots x_{ii}\ldots \\ &\vdots \end{aligned}$$

Dabei sind x_{ij}, $j \in \mathbb{N}_0$, die Dezimalziffern von x_i. Wir bilden mithilfe der Diagonalen der Tabelle die Zahl

$$y = y_1 y_2 \ldots y_i \ldots$$

durch

$$y_i = \begin{cases} 1, & x_{ii} \neq 1, \\ 2, & x_{ii} = 1. \end{cases} \tag{2.21}$$

Offensichtlich ist $y \in I$. Die Zahl y muss also in der obigen Tabelle vorkommen, d. h., es gibt eine Nummer k mit $y = x_k$. Daraus folgt, dass $y_k = x_{kk}$ sein muss. Gemäß Definition (2.21) ist aber $y_k \neq x_{kk}$. Die Annahme einer Nummerierung führt zu diesem Widerspruch und ist damit widerlegt.

2.9 Zusammenfassung und bibliografische Hinweise

In diesem Kapitel werden grundlegende Begriffe zu Alphabeten, Wörtern und Sprachen sowie zu Funktionen und Operationen darauf definiert.

Des Weiteren werden Möglichkeiten zur Nummerierung der Menge aller Wörter über einem Alphabet Σ sowie zur Codierung von natürlichen Zahlen durch Bitfolgen angegeben. Wegen dieser Möglichkeiten kann in den folgenden Kapiteln als Grundlage für die Betrachtungen eine dieser Mengen als Ausgangspunkt gewählt werden. Denn wegen der gegenseitigen Codierungen gelten die Betrachtungen gleichermaßen auch für die jeweils beiden anderen Mengen.

Analog ermöglicht die Codierung durch die Cantorschen k-Tupelfunktionen, anstelle von k-stelligen Funktionen nur einstellige Funktionen über $\mathbb{N}_0$ zu betrachten.

Die Darstellungen in diesem Kapitel sind teilweise an entsprechende Darstellungen in [VW16] angelehnt. Formale Sprachen sowie Codierungen und Nummerierungen von Zeichenketten und Sprachen werden in [AB02], [HMU13], [Hr14], [Koz97], [LP98], [Schö09] und [Si06] betrachtet.

Kapitel 3

Berechenbarkeit

Algorithmen werden schon seit alters her entwickelt und angewendet, um Probleme zu lösen. Araber, Chinesen, Inder und Griechen hatten schon vor langer Zeit für viele mathematische Problemstellungen Rechenverfahren entwickelt. Als Beispiel sei der *Euklidische Algorithmus*[1] zur Berechnung des größten gemeinsamen Teilers von zwei natürlichen Zahlen genannt. Allgemein geht es darum,

> *eine Funktion* $f : \mathbb{N}_0^k \to \mathbb{N}_0$ *als berechenbar anzusehen genau dann, wenn es einen Algorithmus gibt, der* f *berechnet.*

Der Begriff *Algorithmus* selbst wurde allerdings bis zu Beginn des vorigen Jahrhunderts eher informell beschrieben. Erst zu dieser Zeit entstand das Bedürfnis, diesen Begriff mathematisch präzise zu definieren.

Wegen der Definition 2.3 können wir die Berechenbarkeit von Funktionen auf die (Semi-) Entscheidbarkeit ihrer Graphen zurückführen. Graphen sind Mengen, und gemäß Definition 2.1 sind Mengen entscheidbar bzw. semi-entscheidbar, wenn es Algorithmen gibt, mit denen ihre charakteristischen bzw. ihre semi-charakteristischen Funktionen berechnet werden können. Wir müssen also letztendlich die Berechenbarkeit für charakteristische und semi-charakteristische Funktionen mathematisch präzisieren, dann ist die Entscheidbarkeit bzw. Semi-Entscheidbarkeit von Mengen und damit implizit die Berechenbarkeit von Funktionen formal definiert.

Es gibt eine Reihe von Ansätzen, den Begriff der Berechenbarkeit für Funktionen zu definieren. Wir wählen den Ansatz von Alan M. Turing[2], der das „menschli-

[1] Der griechische Mathematiker *Euklid* lebte wohl im 3. Jahrhundert vor Christus in Alexandria. In seinen *Elementen* stellte er die Erkenntnisse der damaligen griechischen Mathematik in einheitlicher, systematischer Weise zusammen. Sein methodisches Vorgehen und seine strenge, auf logischen Prinzipien beruhende Beweisführung waren beispielhaft und grundlegend für die Mathematik bis in die Neuzeit.

[2] Alan M. Turing (1912–1954) war ein britischer Logiker und Mathematiker. Er schuf mit dem Berechenbarkeitskonzept der Turingmaschine eine *der* wesentlichen Grundlagen sowohl für die Theoretische Informatik als auch für die Entwicklung realer universeller Rechner. Im Zweiten Weltkrieg war er maßgeblich an der Entschlüsselung des mit der „Wundermaschine“ Enigma verschlüsselten deutschen Funkverkehrs beteiligt. Turing lieferte außerdem wesentliche Beiträge zur Theoretischen Biologe, entwickelte Schachprogramme sowie den Turing-Test, mit dem festgestellt werden soll, ob eine Maschine menschliche

K.-U. Witt und M. E. Müller, *Algorithmische Informationstheorie*,
https://doi.org/10.1007/978-3-662-61694-9_3

che Rechnen“ als Ausgangspunkt für seine Formalisierung gewählt hat. Alle anderen Ansätze sind äquivalent zum Turingschen Ansatz und damit auch untereinander äquivalent. Die Äquivalenz ist die Begründung für die *Churchsche These* (siehe Abschnitt 3.3), die besagt, dass das Konzept der Turing-Berechenbarkeit genau mit dem intuitiven Verständnis von Berechenbarkeit übereinstimmt.

3.1 Turing-Berechenbarkeit

Angenommen, man möchte eine Formel ausrechnen oder eine Übungsaufgabe lösen, wie geht man vor? Man nimmt z. B. ein Blatt oder mehrere Blätter Papier zur Hand, einen Schreibstift und ein Löschwerkzeug (Radiergummi, Tintenkiller). Man kann dann Berechnungen auf das Papier notieren, kann zu jeder Stelle des Aufgeschriebenen gehen, um dieses zu verändern oder in Abhängigkeit des Notierten Veränderungen an anderen Stellen vornehmen.

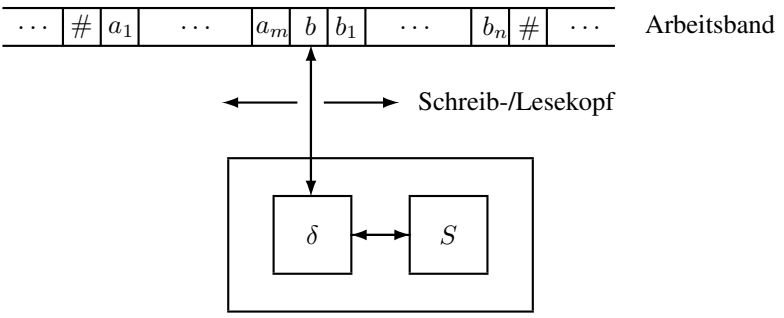

Abbildung 3.1: Arbeitsweise einer Turingmaschine

Diese Überlegung ist der Ausgangspunkt für Turings Ansatz, Berechenbarkeit formal, d. h. mathematisch, zu präzisieren: Berechnungen erfolgen mit der nach ihm benannten Turingmaschine (vgl. Abb. 3.1). Das nach beiden Seiten unendliche Arbeitsband entspricht dem Papier, der Schreib-/Lesekopf realisiert die Schreib- und Löschwerkzeuge. In einem endlichen internen Speicher S können Zustände für den Stand von Berechnungen gespeichert werden. Das auf dem Papier Geschriebene wird in Abhängigkeit vom jeweiligen Zustand nach einem bestimmten endlichen – durch

Intelligenz besitzt. Nach Turing ist der *Turing-Award* benannt, die höchste Auszeichnung für Wissenschaftlerinnen und Wissenschaftler in der Informatik. Wegen seiner Homosexualität wurde Turing 1952 zu einer zwangsweisen Hormonbehandlung verurteilt, in dessen Folge er an Depression erkrankte. Im Juni 1954 starb er durch Suizid. Anlässlich seines hundertsten Geburtstags gab es 2012 weltweit Veranstaltungen zu seinen Ehren. Erst Ende 2013 wurde er durch die britische Regierung und das Königshaus begnadigt und rehabilitiert.

die Zustandsüberführung δ festgelegten – Verfahren (Algorithmus, Programm) manipuliert. Hält das Verfahren in endlicher Zeit, folgt das Ergebnis aus dem erreichten Zustand der Berechnung.

Eingaben für die Berechnung von Turingmaschinen sind im Allgemeinen Wörter über einem Alphabet Σ. Wegen Bemerkung 2.4 können wir uns auf das Alphabet $\mathbb{B}$ beschränken. Das gilt auch im Hinblick auf die (Semi-) Entscheidung von Mengen $A \subseteq \mathbb{N}_0$ und für die Berechnung von Funktionen über $\mathbb{N}_0$.

Definition 3.1 *Eine* **deterministische Turingmaschine** T *ist gegeben durch*

$$T = (\Gamma, S, \delta, s_0, \#, t_a, t_r),$$

bestehend aus

- *dem Arbeitsalphabet (Bandalphabet)* Γ*, welches das Eingabealphabet* $\mathbb{B}$ *enthält, d. h., es ist* $\mathbb{B} \subset \Gamma$*,*
- *der endlichen Zustandsmenge* S*,*
  - *die den Startzustand* s_0*,*
  - *den akzeptierenden Zustand* t_a *(*a *steht für* accept*)*
  - *und den verwerfenden Zustand* t_r *(*r *steht für* reject*) enthält,*
- *dem Blanksymbol* $\# \in \Gamma$
- *und der totalen Funktion*

$$\delta : (S - \{t_a, t_r\}) \times \Gamma \to S \times \Gamma \times \{\leftarrow, \rightarrow, -\}, \tag{3.1}$$

  die die Zustandsüberführung von T *festlegt.*

Die Funktion δ kann als **Programm** der Maschine T aufgefasst werden. Die Anweisung $\delta(s, b) = (s', c, m)$, die wir auch in der Form $(s, b, s', c, m) \in \delta$ notieren, bedeutet: Ist T im Zustand s und befindet sich der Schreib-/Lesekopf (im Folgenden kurz S-/L-Kopf) unter dem Symbol b, dann geht T in den Zustand s' über, überschreibt b mit dem Symbol c und führt die Bewegung m aus. Ist $m = \leftarrow$, dann geht der S-/L-Kopf eine Position nach links, ist $m = \rightarrow$, geht er eine Position nach rechts, ist $m = -$, dann bewegt er sich nicht. Die Funktion δ beschreibt also das Bearbeiten des Arbeitsbandes sowie die Zustandsänderungen.

Der aktuelle Stand einer Bearbeitung wird beschrieben durch eine **Konfiguration**

$$k = \alpha s \beta \in \Gamma^* \circ S \circ \Gamma^+ \text{ mit } \alpha \in \Gamma^*,\ s \in S,\ \beta \in \Gamma^+,$$

die den aktuellen kompletten Bandinhalt sowie die Position des S-/L-Kopfes auf dem Arbeitswort festhält (siehe Abbildung 3.1). α ist der Teil des Arbeitswortes, der links

vor dem Kopf steht; in Abbildung 3.1 ist $\alpha = a_1 \dots a_m$, $a_i \in \Gamma$, $1 \leq i \leq m$, $m \geq 0$. $\beta[1]$, der erste Buchstabe von β, ist das Symbol, an dem sich der Kopf befindet (in Abbildung 3.1 ist $\beta[1] = b$), und $\beta[2, |\beta|]$ ist der Teil des Arbeitswortes rechts vom Kopf (in Abbildung 3.1 ist $\beta[j+1] = b_j$, $1 \leq j \leq n$, $n \geq 0$). Es sei

$$K_T = \Gamma^* \circ S \circ \Gamma^+$$

die **Menge der Konfigurationen von T**.

Wenn wir tatsächlich immer den kompletten Inhalt des beidseitig unendlichen Arbeitsbandes einer Turingmaschine betrachten würden, bestünden ein Präfix von α und ein Postfix von β jeweils aus unendlich vielen Blanksymbolen. Deshalb stellen wir nicht den kompletten Inhalt dar, sondern nur den endlichen Ausschnitt, der aktuell von Interesse ist. Das heißt in der Regel, dass α mit höchstens einem Blanksymbol beginnt, β mit höchstens einem oder zwei Blanksymbolen endet und die anderen Buchstaben von α und β – falls vorhanden – ungleich dem Blanksymbol sind.

Definition 3.2 *Die Arbeitsweise, d. h. die Berechnung, die eine Turingmaschine mit einer Eingabe durchführt, wird durch* **Konfigurationsübergänge** *beschrieben. Die Menge der möglichen Konfigurationsübergänge von T*

$$\vdash \subseteq K_T \times K_T$$

ist für $b, c \in \Gamma$ definiert durch

$$\alpha asb\beta \vdash \begin{cases} \alpha as'c\beta, & \text{falls } (s, b, s', c, -) \in \delta,\ \alpha a, \beta \in \Gamma^*, \\ \alpha acs'\beta, & \text{falls } (s, b, s', c, \rightarrow) \in \delta,\ \alpha a \in \Gamma^*\ \beta \in \Gamma^+, \\ \alpha s'ac\beta, & \text{falls } (s, b, s', c, \leftarrow) \in \delta,\ \alpha, \beta \in \Gamma^*,\ a \in \Gamma. \end{cases} \tag{3.2}$$

Im ersten Fall wird das Symbol b, das der Automat liest, durch c ersetzt, und der S-/L-Kopf bleibt unverändert. Im zweiten Fall wird b ersetzt durch c, und der Kopf geht nach rechts. Im dritten Fall wird b durch c ersetzt, und der Kopf geht eine Position nach links.

Die Berechnung einer Turingmaschine beginnt mit einer **Startkonfiguration** $s_0 w$: Der S-/L-Kopf befindet sich unter dem ersten Buchstaben der Eingabe $w \in \mathbb{B}^*$. Die Startkonfiguration $s_0\#$ bedeutet, dass die Eingabe die leere Bitfolge ε ist.

Konfigurationen $\alpha t_a \beta$ und $\alpha t_r \beta$ mit $\alpha \in \Gamma^*$, $\beta \in \Gamma^+$ heißen **akzeptierende** bzw. **verwerfende Konfigurationen**. Da δ total auf $(S - \{t_a, t_r\}) \times \Gamma$ definiert ist, sind akzeptierende und verwerfende Konfigurationen die einzigen Konfigurationen, bei denen die Bearbeitung anhält, weil für diese kein weiterer Konfigurationsübergang mehr möglich ist. Deswegen heißen diese Konfigurationen auch **Haltekonfigurationen**.

Definition 3.3 *Sei $T = (\Gamma, S, \delta, s_0, \#, t_a, t_r)$ eine Turingmaschine. Eine Eingabe $w \in \mathbb{B}^*$ wird von T* **akzeptiert**, *falls $\alpha, \beta \in \Gamma^*$ existieren mit $s_0 w \vdash^* \alpha t_a \beta$; w wird von T* **verworfen**, *falls $\alpha, \beta \in \Gamma^*$ existieren mit $s_0 w \vdash^* \alpha t_r \beta$.*

Konfigurationsfolgen müssen nicht endlich sein. Ist z. B. eine Konfiguration $\alpha s \beta$ erreicht und die Zustandsüberführung $(s, \beta[1], s, \beta[1], -) \in \delta$ gegeben, dann verharrt die Maschine in dieser Konfiguration und erreicht keine Haltekonfiguration.

Für eine Eingabe $w \in \mathbb{B}^*$ gibt es also drei Möglichkeiten: Nach endlich vielen Konfigurationsübergängen

- stoppt die Bearbeitung in einer akzeptierenden
- oder in einer verwerfenden Konfiguration,
- oder die Bearbeitung terminiert nicht.

Definition 3.4 *Eine deterministische Turingmaschine T definiert die Funktion*

$$\Phi_T : \mathbb{B}^* \to \mathbb{B}$$

wie folgt:

$$\Phi_T(w) = \begin{cases} 1, & w \text{ wird von } T \text{ akzeptiert} \\ 0, & w \text{ wird von } T \text{ verworfen} \\ \bot, & \text{sonst} \end{cases} \tag{3.3}$$

Die von T **akzeptierte Sprache** *ist die Menge*

$$L(T) = \{w \in \mathbb{B}^* \mid \Phi_T(w) = 1\} \tag{3.4}$$

der von T akzeptierten Bitfolgen. Eine Menge $L \subseteq \mathbb{B}^*$ heißt **Turing-akzeptierbar**, *falls eine Turingmaschine T existiert mit $L = L(T)$. T heißt* **Turing-Akzeptor** *für L.*

Wir bezeichnen mit DTA *die* **Klasse der von deterministischen Turingmaschinen akzeptierbaren Mengen**.

Eine Menge $L \subseteq \mathbb{B}^*$ heißt **Turing-entscheidbar**, *falls eine Turingmaschine T existiert mit $Def(\Phi_T) = \mathbb{B}^*$ und $L = L(T)$. T heißt* **Turing-Entscheider** *für L.*

Wir bezeichnen mit DTE *die* **Klasse der von deterministischen Turingmaschinen entscheidbaren Mengen**.

Turing-Entscheider terminieren also für jede Eingabe in einer Haltekonfiguration: Eine Eingabe wird entweder akzeptiert oder verworfen. Turing-Akzeptoren können bei einer Eingabe ebenfalls in eine Haltekonfiguration gelangen, sie können aber auch nicht terminieren.

Bemerkung 3.1 *Sei $L \subseteq \mathbb{B}^*$, $w \in \mathbb{B}^*$ und T eine Turingmaschine.*

a) *Gilt $\Phi_T(w) = 0$ oder $\Phi_T(w) = 1$, dann nennen wir 0 bzw. 1 auch die* **Ausgabe** *von T bei Eingabe w.*

b) *Gilt $\Phi_T(w) = \chi_L(w)$ oder $\Phi_T(w) = \chi'_L(w)$ für alle $w \in \mathbb{B}^*$, dann sagen wir, dass T die Funktion χ_L bzw. die Funktion χ'_L* **berechnet**.

Wir können Turing-Akzeptoren so umformen, dass sie auch in den Fällen, in denen sie in eine verwerfende Konfiguration gelangen, nicht terminieren: Dazu ergänzen wir die Zustandsüberführung durch die Anweisungen $\delta(t_r, a) = (t_r, a, -)$ für alle $a \in \Gamma$. Wir verstoßen damit allerdings gegen die Bedingung (3.1). Da Turing-Akzeptoren aber keinen verwerfenden Zustand benötigen, können wir deren Definition verändern in: $T = (\Gamma, S, \delta, s_0, \#, t_a)$ mit

$$\delta : (S - \{t_a\}) \times \Gamma \to S \times \Gamma \times \{\leftarrow, \rightarrow, -\}. \tag{3.5}$$

Mit der so veränderten Definition von Turing-Akzeptoren gilt:

Folgerung 3.1 *Sei $L \subseteq \mathbb{B}^*$, dann gilt:*

a) *$L \in \mathsf{DTA}$ genau dann, wenn ein Turing-Akzeptor T existiert mit*

$$\Phi_T(w) = \begin{cases} 1, & w \in L, \\ \perp, & w \notin L. \end{cases} \tag{3.6}$$

Wegen dieses Ausgabeverhaltens könnten Turing-Akzeptoren auch **Turing-Semi-Entscheider** *genannt werden.*

b) *$L \in \mathsf{DTE}$ genau dann, wenn ein Turing-Entscheider T existiert mit*

$$\Phi_T(w) = \begin{cases} 1, & w \in L, \\ 0, & w \notin L. \end{cases} \tag{3.7}$$

c) *Aus a) und b) folgt unmittelbar:* $\mathsf{DTE} \subseteq \mathsf{DTA}$.

Wir können jetzt insbesondere die Definition 2.1 vervollständigen, indem wir die natürlichen Zahlen für die Berechnung der charakteristischen und semi-charakteristischen Funktionen von Teilmengen $A \subseteq \mathbb{N}_0$ mithilfe der in Abschnitt 2.6 eingeführten Bijektion $\nu : \mathbb{N}_0 \to \mathbb{B}^*$ codieren.

Definition 3.5 **a)** *Eine Menge $A \subseteq \mathbb{N}_0$ ist* **entscheidbar** *genau dann, wenn ein Turing-Entscheider T existiert mit $\chi_A(x) = \Phi_T(\nu(x))$ für alle $x \in \mathbb{N}_0$.*

b) *Eine Menge $A \subseteq \mathbb{N}_0$ ist* **semi-entscheidbar** *genau dann, wenn ein Turing-Akzeptor T existiert mit $\chi'_A(x) = \Phi_T(\nu(x))$ für alle $x \in \mathbb{N}_0$.*

Mit der Umkehrfunktion $\tau_{\mathbb{B}} : \mathbb{B}^* \to \mathbb{N}_0$ von ν erhalten wir:

Folgerung 3.2 **a)** *Eine Menge $A \subseteq \mathbb{N}_0$ ist entscheidbar genau dann, wenn ein Turing-Entscheider T existiert mit $A = \tau_{\mathbb{B}}(L(T))$.*

b) *Eine Menge $A \subseteq \mathbb{N}_0$ ist semi-entscheidbar genau dann, wenn ein Turing-Akzeptor T existiert mit $A = \tau_{\mathbb{B}}(L(T))$.*

Im Folgenden vernachlässigen wir aus schreibtechnischen Gründen die Codierungen ν und τ und schreiben (1) anstelle von $\chi_A(x) = \Phi_T(\nu(x))$ nur $\chi_A(x) = \Phi_T(x)$ bzw. (2) anstelle von $\chi'_A(x) = \Phi_T(\nu(x))$ nur $\chi'_A(x) = \Phi_T(x)$. Analog schreiben wir anstelle von $A = \tau_{\mathbb{B}}(L(T))$ nur $A = L(T)$. Im Fall (1) nennen wir T einen **Entscheider für $\boldsymbol{A}$**, entsprechend nennen wir im Fall (2) T einen **Akzeptor für $\boldsymbol{A}$**.

Beispiel 3.1 *Wir wollen uns eine Turingmaschine überlegen, welche die charakteristische Funktion der Sprache $L = \{0^n1^n \mid n \in \mathbb{N}_0\}$ berechnet. Die Grundidee ist, dass die Maschine wiederholt von links nach rechts läuft, dabei überprüft, ob das erste Symbol eine 0 ist, gegebenenfalls diese mit einem Blank überschreibt, zum Ende läuft und überprüft, ob das letzte Symbol eine 1 ist. Falls ja, wird diese mit einem Blank überschrieben, und der S-/L-Kopf kehrt an den Anfang des aktuellen Wortes zurück. Dort wird mit einem neuen Durchlauf begonnen. Ist die Konfiguration vor einem Durchlauf $s0^k1^k$, dann ist die Konfiguration vor dem nächsten Durchlauf $s0^{k-1}1^{k-1}$. Ist die Eingabe 0^n1^n, also ein Wort aus L, dann stehen nach n solchen Durchläufen nur noch Blanks auf dem Band, was dem leeren Wort entspricht. Da das leere Wort zur Sprache gehört, geht die Maschine in den Zustand t_a. Ist die Eingabe kein Wort aus L, wird der beschriebene Ablauf unterbrochen, und die Maschine geht in den Zustand t_r.*
Das beschriebene Verfahren wird mit der folgenden Maschine realisiert:

$$T = (\{0, 1, \#\}, \{s, s_0, z, z_1, t_a, t_r\}, \delta, s, \#, t_a, t_r)$$

mit

$$\begin{aligned}
\delta = \{&(s, \#, t_a, \#, -), && \text{das leere Wort wird akzeptiert}\\
&(s, 0, s_0, \#, \rightarrow), && \text{die nächste } 0 \text{ wird überschrieben}\\
&(s_0, 0, s_0, 0, \rightarrow), && \text{zum Ende laufen, über Nullen}\\
&(s_0, 1, s_0, 1, \rightarrow), && \text{und Einsen hinweg}\\
&(s_0, \#, z_1, \#, \leftarrow), && \text{am Ende angekommen, ein Symbol zurück}\\
&(z_1, 1, z, \#, \leftarrow), && \text{prüfen, ob } 1\text{, dann zurück}
\end{aligned}$$

$(z_1, 0, t_r, 0, -)$, *falls* 0*, Eingabe verwerfen*
$(z_1, \#, t_r, \#, -)$, *falls Band leer, Eingabe verwerfen*
$(z, 1, z, 1, \leftarrow)$, *zurück über alle Einsen*
$(z, 0, z, 0, \leftarrow)$, *und Nullen*
$(z, \#, s, \#, \rightarrow)$, *am Wortanfang angekommen, neue Runde*
$(s, 1, t_r, 1, -)\}$ *Anfangssymbol ist* 1*, Eingabe verwerfen*

Wir testen das Programm mit den Eingaben 0011*,* 001 *und* 011*:*

$$s0011 \vdash s_0011 \vdash 0s_011 \vdash 01s_01 \vdash 011s_0\# \vdash 01z_11 \vdash 0z1 \vdash z01 \vdash z\#01$$
$$\vdash s01 \vdash s_01 \vdash 1s_0\# \vdash z_11 \vdash z\# \vdash s\# \vdash t_a\#$$

$$s001 \vdash s_001 \vdash 0s_01 \vdash 01s_0\# \vdash 0z_11 \vdash z0 \vdash z\#0 \vdash s0 \vdash s_0\# \vdash z_1\# \vdash t_r\#$$

$$s011 \vdash s_011 \vdash 1s_01 \vdash 11s_0\# \vdash 1z_11 \vdash z1 \vdash z\#1 \vdash s1 \vdash t_r1$$

Es gilt also $\Phi_T(0011) = 1$*,* $\Phi_T(001) = 0$ *sowie* $\Phi_T(011) = 0$*. Es kann gezeigt werden, dass* $\chi_L(x) = \Phi_T(x)$ *für alle* $x \in \mathbb{B}^*$ *gilt.*

Es ist $\tau_{\mathbb{B}}(0^n1^n) = 2^n(2^n + 1) - 1$*. Sei*

$$A = \{2^n(2^n + 1) - 1 \mid n \in \mathbb{N}_0\} = \{1, 5, 19, 71, 271, \ldots\}$$

dann gilt (mit der oben vereinbarten Schreibweise) $A = L(T)$*, und A ist entscheidbar.*

Übung 3.1 *Konstruieren Sie einen Turing-Entscheider für die Sprache*

$$L = \{w\overleftarrow{w} \mid w \in \mathbb{B}^*\}$$

Testen Sie diesen mit den Folgen 1001*,* 0111 *und* 111*!*

Bemerkung 3.2 **a)** *Aufgrund der Definitionen 3.5 und 2.3 haben wir mit den Turingmaschinen eine mathematische Definition für* **Algorithmen** *bzw. für* **Programme** *und damit auch für die Entscheidbarkeit und Semi-Entscheidbarkeit von Mengen bzw. für die* **Berechenbarkeit von Funktionen***. Bisher war die Definition „Eine Funktion ist berechenbar, wenn es einen Algorithmus gibt, der sie berechnet," eine informelle Beschreibung. Nun haben wir eine formale Definition für die Berechenbarkeit von Funktionen: Eine Funktion* $f : \mathbb{N}_0^k \to \mathbb{N}_0$ *ist berechenbar, wenn es eine Turingmaschine gibt, die* χ'_{G_f} *berechnet. Wir sehen ab jetzt ein Berechnungsverfahren als Algorithmus an, wenn dieser als eine Turingmaschine implementiert werden kann.*

b) *Sei die Funktion $f : \mathbb{N}_0^k \to \mathbb{N}_0$ berechenbar und T ein Semi-Entscheider von χ'_{G_f}. Gilt $\chi'_{G_f}(x1, \ldots, x_k, y) = 1$, dann nennen wir y die* **Ausgabe** *von T bei Eingabe von $\langle x_1, \ldots, x_k \rangle$ und schreiben dafür $\Phi_T \langle x_1, \ldots, x_k \rangle = y$.*

3.2 Varianten von Turingmaschinen

Turingmaschinen mit einseitig beschränktem Arbeitsband

In der Definition 3.1 ist das Arbeitsband beidseitig unbeschränkt. Man kann das Band einseitig – z. B. nach links – beschränken ohne Verlust von Berechnungskraft. Es ist klar, dass eine einseitig beschränkte Maschine durch eine beidseitig unbeschränkte unmittelbar simuliert werden kann. Will man eine unbeschränkte Maschine T durch eine einseitig beschränkte Maschine T' simulieren, dann muss T' immer dann, wenn T ein Symbol nach links über das Bandende hinausgehen möchte, in ein Unterprogramm verzweigen, das den kompletten Bandinhalt eine Position nach rechts verschiebt und dann wie T fortfährt.

Mehrband-Maschinen

Man kann Turingautomaten auch mit m Arbeitsbändern, $m \geq 1$, mit jeweils eigenem S-/L-Kopf definieren. In Abhängigkeit vom aktuellen Zustand und den Symbolen, unter denen sich die m S-/L-Köpfe auf ihren jeweiligen Bändern befinden, geht eine m-bändige Turingmaschine in einen neuen Zustand über, überschreibt die gelesenen Symbole mit neuen Symbolen und bewegt die einzelnen Köpfe unabhängig voneinander entweder eine Position nach links, eine Position nach rechts oder lässt sie stehen. Die Zustandsüberführung ist also eine Funktion der Art

$$\delta : (S - \{t_a, t_r\} \times \Gamma^m \to S \times \Gamma^m \times \{\leftarrow, \rightarrow, -\}^m.$$

Die Menge der Konfigurationen ist gegeben durch $K = S \times (\Gamma^* \circ \{\uparrow\} \circ \Gamma^+)^m$. Eine Konfiguration $(s, \alpha_1 \uparrow \beta_1, \ldots, \alpha_m \uparrow \beta_m)$ beschreibt, dass die Maschine sich im Zustand s befindet, der Inhalt von Band i gleich $\alpha_i \beta_i$ ist und der S-/L-Kopf von Band i unter dem Symbol $\beta_i[1]$ steht.

Mehrband-Maschinen sind oft vorteilhaft bei der Berechnung von mehrstelligen Funktionen $f : \mathbb{N}_0^k \to \mathbb{N}_0$, d. h. für die (Semi-) Entscheidung von G_f: Die k Komponenten der Eingabe werden auf k Eingabebänder geschrieben und die Ausgabe auf ein Ausgabeband. Weitere Bänder können als Hilfsbänder dienen. Eine solche Maschine hat dann $k + 1$ und mehr Arbeitsbänder. Die Verfügbarkeit von mehr als einem Arbeitsband erhöht die Mächtigkeit von Turingmaschinen nicht, denn jede Maschine mit mehr als einem Band kann durch eine mit nur einem Band simuliert werden.

Übung 3.2 *Konstruieren Sie eine Maschine mit zwei Bändern, die die charakteristische Funktion der Sprache $L = \{0^n 1^n \mid n \in \mathbb{N}_0\}$ berechnet!*

Mithilfe von Mehrbandmaschinen kann man zeigen, dass die Komposition berechenbarer Funktionen berechenbar ist.

Satz 3.1 *Es seien $f, g : \mathbb{N}_0 \to \mathbb{N}_0$ berechenbare Funktionen, dann ist ihre Komposition $h = f \circ g$, definiert durch $h(x) = f(g(x))$, berechenbar.*

Übung 3.3 *Beweisen Sie Satz 3.1!*

Bemerkung 3.3 *Ist T_h die Turingmaschine, die die Komposition $h = f \circ g$ durch geeignetes Zusammensetzen der Maschinen T_f und T_g berechnet (wie z. B. durch die in der Lösung von Übung 3.3 beschriebenen Konstruktion), dann schreiben wir $T_h = T_f \circ T_g$.*

Mehrere Schreib-/Leseköpfe

Eine weitere Variante lässt auf Arbeitsbändern mehrere S-/L-Köpfe zu. Auch solche Maschinen können durch Maschinen mit nur einem S-/L-Kopf pro Band simuliert werden.

k-dimensionales Arbeitsband

Ein k-dimensionales Arbeitsband kann man sich als k-dimensionales Array von Feldern vorstellen, das in alle $2k$ Richtungen unbeschränkt ist (im Fall $k = 2$ kann man sich das Array als eine in alle vier Richtungen unendliche Matrix vorstellen). In Abhängigkeit vom aktuellen Zustand und dem Feldinhalt der Position, an der sich der S-/L-Kopf befindet, geht die Maschine in einen Folgezustand über, überschreibt den Feldinhalt, und der S-/L-Kopf bewegt sich auf ein Nachbarfeld in einer der $2k$ Richtungen. Auch Maschinen dieser Art können durch Einband-Maschinen simuliert werden.

In den Abschnitten 3.5 und 3.7 werden wir noch zwei weitere Varianten von Turingmaschinen – sogenannte **Verifizierer** sowie **nicht deterministische Turingmaschinen** – kennenlernen und ausführlicher betrachten als die oben nur kurz erläuterten Varianten, denn diese beiden Varianten haben eine größere theoretische und praktische Bedeutung als die anderen.

Alle erwähnten Varianten können sich gegenseitig simulieren, berechnen somit dieselbe Klasse von Funktionen – sie sind berechnungsäquivalent.

Definition 3.6 *Zwei Turingmaschinen T_1 und T_2 (die nicht von derselben Variante sein müssen) heißen* **äquivalent** *genau dann, wenn $\Phi_{T_1}(w) = \Phi_{T_2}(w)$ für alle $w \in \mathbb{B}^*$ gilt, d. h., wenn sie dieselbe Funktion berechnen.*

3.3 Churchsche These

Neben dem Begriff der **Berechenbarkeit**, den wir in der obigen Definition mit Turing-Berechenbarkeit festgelegt haben, haben wir auch den Begriff **Algorithmus** mathematisch präzisiert: Ein Algorithmus ist als formales Berechnungsverfahren zum jetzigen Zeitpunkt durch den Begriff der Turingmaschine definiert. Eine Funktion ist algorithmisch berechenbar, falls es einen Algorithmus – hier eine Turingmaschine – gibt, der sie berechnet.

Neben der Turing-Berechenbarkeit gibt es eine Reihe **weiterer Berechenbarkeitskonzepte**, wie z. B.

- μ-rekursive Funktionen,
- Random Access-Maschinen,
- Markov-Algorithmen,
- λ-Kalkül,
- While-Programme,
- Goto-Programme,
- Programmiersprachen wie C++ und Java, deren Programme auf Rechnern mit beliebig großem Speicher ausgeführt werden.

Wir wollen jedes dieser Konzepte als Programmiersprache verstehen. Sei $\mathcal{M}$ der Name für eine Programmiersprache, z. B. $\mathcal{M} = \mathcal{T}$ für die durch die Menge aller Turingmaschinen $\mathcal{T}$ gegebene Programmiersprache. Ist eine Funktion $f : \mathbb{N}_0^k \to \mathbb{N}_0$ in der Sprache $\mathcal{M}$ durch ein Programm A berechenbar, dann schreiben wir: $f = \Phi_A^{(\mathcal{M})}$.

Es kann gezeigt werden: Sind $\mathcal{M}_1$ und $\mathcal{M}_2$ zwei der oben genannten Programmiersprachen, dann existiert zu jedem Programm $A \in \mathcal{M}_1$ ein Programm $B \in \mathcal{M}_2$ mit $\Phi_A^{(\mathcal{M}_1)} = \Phi_B^{(\mathcal{M}_2)}$, und zu jedem Programm $B \in \mathcal{M}_2$ existiert ein Programm $A \in \mathcal{M}_1$ mit $\Phi_B^{(\mathcal{M}_2)} = \Phi_A^{(\mathcal{M}_1)}$. Die Berechenbarkeitskonzepte sind also, obwohl sie sich (äußerlich) sehr voneinander unterscheiden, zueinander äquivalent, und damit sind alle äquivalent zur Turing-Berechenbarkeit. Diese Äquivalenz ist die Basis für

die Churchsche These.[3] Wir werden im Abschnitt 5.4.4 die Existenz von Übersetzern zwischen Programmiersprachen beweisen.

Churchsche These *Die Menge der Turing-berechenbaren Funktionen ist genau die Menge der im intuitiven Sinne berechenbaren Funktionen.*

Wir nennen die Berechenbarkeitskonzepte, die äquivalent zur Turing-Berechenbarkeit sind, **vollständig**. Beispiele für vollständige Berechenbarkeitskonzepte sind die oben aufgelisteten.

Gemäß der Churchschen These brauchen wir also nicht mehr zwischen unterschiedlichen vollständigen Berechenbarkeitskonzepten zu unterscheiden, z. B. zwischen Turing-Berechenbarkeit und While-Berechenbarkeit, sondern können unabhängig von dem im Einzelfall verwendeten Konzept allgemein von Berechenbarkeit sprechen. Wenn eine Funktion $f : \mathbb{N}_0^k \to \mathbb{N}_0$ Turing-berechenbar ist, d. h., wenn es ein Turingprogramm $T \in \mathcal{T}$ gibt, das f berechnet, müssen wir das nicht explizit angeben: Anstelle von $f = \Phi_T^{(\mathcal{T})}$ können wir lediglich $f = \Phi_T$ schreiben (wie wir das bisher auch gemacht haben).

Ebenso werden wir im Folgenden, wenn gezeigt werden soll, dass eine Funktion $f : \mathbb{N}_0^k \to \mathbb{N}_0$ berechenbar ist, dies nicht immer dadurch belegen, dass wir eine Turingmaschine für f programmieren, sondern dadurch, dass wir einen Algorithmus – mehr oder weniger informell – in einer Programmiersprachen ähnlichen Form angeben – wohl wissend, dass wir auch eine Turingmaschine oder ein Java-Programm zur Berechnung der Funktion angeben könnten. Pseudoformal notierte Programme sind oft verständlicher als Turingmaschinen.

Im Folgenden bezeichnen wir mit $\mathcal{R}^{(k)}$ die Menge der **total berechenbaren k-stelligen Funktionen** und mit $\mathcal{P}^{(k)}$ die **Menge der (partiell) berechenbaren k-stelligen Funktionen** und fassen diese mit

$$\mathcal{R} = \bigcup_{k \geq 0} \mathcal{R}^{(k)} \text{ und } \mathcal{P} = \bigcup_{k \geq 0} \mathcal{P}^{(k)} \tag{3.8}$$

zur **Menge aller total berechenbaren** bzw. zur **Menge aller berechenbaren Funktionen** zusammen.

Übung 3.4 *Zeigen Sie, dass $\mathcal{R} \subset \mathcal{P}$ gilt!*

[3] Diese These, auch Church-Turing-These genannt, wurde vorgeschlagen und begründet von Alonzo Church (1903–1995), amerikanischer Mathematiker und Logiker, der – wie sein Schüler Kleene – wesentliche Beiträge zur mathematischen Logik und Berechenbarkeitstheorie geleistet hat.

3.4 Entscheidbare, semi-entscheidbare und rekursiv-aufzählbare Mengen

In diesem Abschnitt werden wir grundlegende Eigenschaften von entscheidbaren und semi-entscheidbaren Mengen betrachten. Des Weiteren werden wir den Begriff der rekursiven Aufzählbarkeit von Mengen einführen und zeigen, dass die rekursiv-aufzählbaren Mengen genau die semi-entscheidbaren Mengen sind. Aus diesem Grund haben wir im Abschnitt 2.7 die Klasse der semi-entscheidbaren Mengen rekursiv-aufzählbar genannt und entsprechend mit RE bezeichnet.

Satz 3.2 **a)** *Ist die Menge $A \subseteq \mathbb{N}_0$ entscheidbar, dann ist auch ihr Komplement $\overline{A}$ entscheidbar.*

b) *Eine Menge $A \subseteq \mathbb{N}_0$ ist entscheidbar genau dann, wenn A und $\overline{A}$ semi-entscheidbar sind.*

Beweis **a)** *Sei T eine Turingmaschine, die χ_A berechnet. Daraus konstruieren wir die Maschine T' wie folgt: T und T' sind identisch bis auf das Ausgabeverhalten. Falls T eine 1 ausgibt, dann gibt T' eine 0 aus, und falls T eine 0 ausgibt, dann gibt T' eine 1 aus.*

b) *„⇒“: Sei T eine Turingmaschine, die χ_A berechnet. Daraus konstruieren wir die Maschine T' wie folgt: T und T' sind identisch bis auf das Ausgabeverhalten. Falls T eine 1 ausgibt, dann gibt T' eine 1 aus, und falls T eine 0 ausgibt, dann läuft T' unendlich weiter nach rechts (siehe Übung 3.4). Es folgt $\Phi_{T'} = \chi'_A$. Die Funktion χ'_A ist also berechenbar, und damit ist A semi-entscheidbar. Wir haben also gezeigt: Ist A entscheidbar, dann ist A auch semi-entscheidbar. Hieraus folgt mithilfe von a): Ist A entscheidbar, dann ist $\overline{A}$ semi-entscheidbar.*

„⇐“: Sei T_A eine Turingmaschine, die χ'_A berechnet, und sei $T_{\overline{A}}$ eine Turingmaschine, die $\chi'_{\overline{A}}$ berechnet. Wir konstruieren eine 2-Bandmaschine T, die auf dem ersten Band die Maschine T_A und auf dem zweiten Band die Maschine $T_{\overline{A}}$ simuliert. Dabei werden die Konfigurationsübergänge von T_A und $T_{\overline{A}}$ schrittweise parallel durchgeführt. Da eine Eingabe x entweder ein Element von A oder ein Element von $\overline{A}$ ist, wird im ersten Fall $\Phi_{T_A}(x) = 1$ und im anderen Fall $\Phi_{T_{\overline{A}}}(x) = 1$ sein. Die Maschine T, die quasi T_A und $T_{\overline{A}}$ parallel ausführt, terminiert also in jedem Fall. Ist $\Phi_{T_A}(x) = 1$, dann gibt T eine 1 aus, ist $\Phi_{T_{A'}}(x) = 1$, dann gibt T eine 0 aus. Es folgt $\Phi_T = \chi_A$. χ_A ist also berechenbar, und damit ist A entscheidbar.

Satz 3.3 **a)** *Ist $A \subset \mathbb{N}_0$ endlich, dann ist $A \in \mathsf{R}$.*

b) *Sind $A, B \in \mathsf{R}$, dann gilt $\overline{A}, A \cup B, A \cap B \in \mathsf{R}$.*

algorithm PN; // P(rim)N(ummerierung)
input: $i \in \mathbb{N}_0$;
output: i-te Primzahl $p(i)$;
$k := 0$; $n := 2$;
while $k < i$ **do**
if n prim **then** $k := k + 1$ **endif**;
$n := n + 1$;
endwhile;
return $n - 1$
endalgorithm

Abbildung 3.2: Algorithmus zur rekursiven Aufzählung der Primzahlen

Übung 3.5 *Beweisen Sie Satz 3.3!*

Definition 3.7 *Eine Menge $A \subseteq \mathbb{N}_0$ heißt* **rekursiv-aufzählbar** *genau dann, wenn $A = \emptyset$ ist oder wenn $A \neq \emptyset$ ist und eine total berechenbare Funktion $f : \mathbb{N}_0 \to \mathbb{N}_0$ mit $W(f) = A$ existiert.*

Wenn f eine rekursive Aufzählung einer nicht leeren Menge $A \subseteq \mathbb{N}_0$ ist, dann gilt $A = \{f(0), f(1), f(2), \ldots\}$. Die Aufzählung f nummeriert also genau die Elemente von A. Da f nicht injektiv sein muss, können Elemente von A mehr als eine Nummer, sogar unendlich viele Nummern haben. Da f total ist, wird jeder Nummer ein Wort aus L zugeordnet.

Beispiel 3.2 *Die Menge der Primzahlen $\mathbb{P}$ ist rekursiv-aufzählbar. Es gibt Algorithmen zum Test von natürlichen Zahlen auf Primalität, z. B. das* Sieb des Eratosthenes. *Der Algorithmus PN in Abb. 3.2 berechnet die totale Funktion $p : \mathbb{N}_0 \to \mathbb{N}_0$, definiert durch $p(i) = p_i$, und p_i ist die i-te Primzahl: $\Phi_{PN} = p$.*

Der folgende Satz zeigt die bereits in der Einleitung des Abschnitts angedeutete Äquivalenz der Begriffe Semi-Entscheidbarkeit und rekursive Aufzählbarkeit.

Satz 3.4 *Es sei $A \subseteq \mathbb{N}_0$, dann gilt:*

a) *A ist rekursiv-aufzählbar genau dann, wenn A semi-entscheidbar ist.*

b) *A ist genau dann rekursiv aufzählbar, wenn eine berechenbare Funktion $g : \mathbb{N}_0 \to \mathbb{N}_0$ existiert mit $Def(g) = A$.*

Beweis **a)** „⇒“: *Sei $A = \emptyset$. Dann ist $\chi'_A : \mathbb{N}_0 \to \mathbb{B}$, definiert durch $\chi'_A(x) = \perp$, für alle $x \in \mathbb{N}_0$ berechenbar (siehe Lösung zur Übung 3.4).*

Sei $A \neq \emptyset$ rekursiv aufzählbar. Dann gib es eine total berechenbare Funktion $f : \mathbb{N}_0 \to \mathbb{N}_0$ mit $W(f) = A$. Sei T eine Turingmaschine, die f berechnet: $\Phi_T = f$. Abb. 3.3 zeigt das Programm X, das χ'_A mithilfe der rekursiven Aufzählung Φ_T berechnet. Ist die Eingabe $x \in A$, dann ist $x \in W(f)$, und es existiert eine Nummer i mit $f(i) = x$. Das Programm X findet die kleinste Nummer mit dieser Eigenschaft und gibt eine 1 aus; es gilt also $\Phi_X(x) = 1$ in diesem Fall. Ist die Eingabe $x \notin A$, dann gibt es kein i mit $f(i) = x$, und das Programm terminiert nicht. Es gilt somit $\Phi_X(x) = \perp$. Insgesamt folgt $\Phi_X(x) = \chi'_A(x)$, damit ist A semi-entscheidbar.

„⇐“: Ist $A = \emptyset$, dann ist A per se rekursiv aufzählbar, und es ist nichts weiter zu zeigen.
Sei also $A \neq \emptyset$, dann enthält A mindestens ein Element. Wir wählen irgendein Element von A – etwa das kleinste – aus und nennen dieses j. Da A semi-entscheidbar ist, ist χ'_A berechenbar. Sei X ein Algorithmus, der χ'_A berechnet: $\Phi_X = \chi'_A$. Der Algorithmus F in Abb. 3.4 berechnet mithilfe von Φ_X eine rekursive Aufzählung von A. F benutzt die beiden Umkehrfunktionen der Cantorschen Paarungsfunktion um sicherzustellen, dass F für jede Eingabe $i \in \mathbb{N}_0$ terminiert, Φ_F also total ist (Φ_X ist nicht total definiert, wenn $A \subset \mathbb{N}_0$ ist).

Der Algorithmus F muss Folgendes leisten, damit Φ_F total sowie $A = W(\Phi_F)$ ist:

(1) Jedem $n \in \mathbb{N}_0$ muss ein $x \in A$ zugeordnet werden, d. h., Φ_F muss total und es muss $W(\Phi_F) \subseteq A$ sein.

(2) Allen $i \in A$ muss mindestens ein $n \in \mathbb{N}_0$ zugeordnet werden, d. h., es muss $A \subseteq W(\Phi_F)$ sein.

Zu (1): Zu $n \in \mathbb{N}_0$ existieren eineindeutig i und k mit $n = \langle i, k \rangle$. Ist $i \in A$ und stoppt X bei Eingabe i in k Schritten, dann wird n die Nummer $x = i$ zugeordnet.
Ist $i \notin A$, dann ist $\chi'_A(i) = \perp$. Deshalb gibt es keine Schrittzahl k, für die das Programm X bei Eingabe i anhält. Somit wird für dieses i den Elementen $n = \langle i, k \rangle$ für alle $k \in \mathbb{N}_0$ die Ausgabe $x = j$ zugeordnet.
In jedem Fall gilt also $\Phi_F(n) = x \in A$ für alle $n \in \mathbb{N}_0$. Die Funktion Φ_F ist also total und $W(\Phi_F) = \Phi_F(\mathbb{N}_0) \subseteq A$.

Zu (2): Sei $i \in A$, dann ist $\chi'_A(i) = 1$. Deshalb gibt es eine Schrittzahl k, bei der das Programm X bei Eingabe i stoppt. Damit ist $n = \langle i, k \rangle$ eindeutig durch i und k bestimmt. Zu $i \in A$ existiert also eine Nummer n, d. h., zu $i \in A$ existiert $n \in \mathbb{N}_0$ mit $\Phi_F(n) = i$, d. h., es ist $A \subseteq \Phi_F(\mathbb{N}_0) = W(\Phi_F)$.

b) *Wegen a) können wir die Behauptung ändern in: A ist semi-entscheidbar genau dann, wenn eine berechenbare Funktion $g : \mathbb{N}_0 \to \mathbb{N}_0$ existiert mit $Def(g) = A$.*

„⇒“: Ist A semi-entscheidbar, dann ist χ'_A berechenbar. Wir setzen $g(x) = \chi'_A(x)$ für alle $x \in \mathbb{N}_0$. Dann ist g berechenbar und $Def(g) = Def(\chi'_A) = A$.

„⇐“: Sei G ein Programm, das g berechnet: $\Phi_G = g$. Der Algorithmus X in Abb. 3.5 berechnet χ'_A: Für ein $x \in \mathbb{N}_0$ sucht X sukzessive einen Wert $y \in \mathbb{N}_0$ mit $g(x) = y$. Falls ein solcher existiert, d. h., wenn $x \in Def(g)$ ist, gibt X eine 1 aus; falls ein solcher nicht existiert, d. h., wenn $x \notin Def(g)$ ist, dann terminiert X nicht, es ist also $\Phi_X(x) = \perp$ für diese x. Insgesamt folgt: $\Phi_X = \chi'_A$. Damit ist χ'_A berechenbar, A ist also semi-entscheidbar.

---

algorithm X;
input: $x \in \mathbb{N}_0$;
output: 1 `falls` $x \in A$;
  $i := 0$;
  while $\Phi_T(i) \neq x$ **do**
    $i := i + 1$;
  endwhile;
  return 1
endalgorithm

Abbildung 3.3: Algorithmus X zur Berechnung von χ'_A mithilfe einer rekursiven Aufzählung von A

---

algorithm F;
input: $n \in \mathbb{N}_0$;
output: `ein Element` $x \in A$;
  $i := \left(\langle n \rangle_2^{-1}\right)_1$; $k := \left(\langle n \rangle_2^{-1}\right)_2$;
  if X bei Eingabe von i in k Schritten stoppt und 1 ausgibt
    then $x := i$
    else $x := j$
  endif;
  return x
endalgorithm

Abbildung 3.4: Algorithmus F zur Berechnung einer rekursiven Aufzählung von A mithilfe von χ'_A

Satz 3.5 *Sind $A, B \in$ RE, dann gilt $A \cup B, A \cap B \in$ RE.*

algorithm X;
input: $x \in \mathbb{N}_0$;
output: 1 `falls` $x \in A$;
$y := 0$;
while $\Phi_G(x) \neq y$ **do**
$y := y + 1$;
endwhile;
return 1
endalgorithm

Abbildung 3.5: Algorithmus X zur Berechnung von χ'_A mithilfe der Funktion g

Übung 3.6 *Beweisen Sie Satz 3.5!*

Der Satz 3.5 b) besagt, dass die Klasse R abgeschlossen gegen Komplement, Vereinigung und Durchschnitt ist, während der obige Satz besagt, dass die Klasse RE abgeschlossen gegen Vereinigung und Durchschnitt ist. RE ist nicht abgeschlossen gegen Komplement; das werden wir in Kapitel 6 (siehe Satz 6.5 b) zeigen.

Übung 3.7 *Sei $A \subseteq \mathbb{N}_0$. Zeigen Sie:*

a) *A ist entscheidbar genau dann, wenn eine Turingmaschine existiert, welche die Elemente von A der Größe nach auf ein Band schreibt.*

b) *A ist semi-entscheidbar genau dann, wenn eine Turingmaschine existiert, welche die Elemente von A auf ein Band schreibt.*

Es ist oftmals hilfreich, die Frage nach der Entscheidbarkeit einer Menge A auf die Entscheidbarkeit einer anderen Menge B zurückzuführen, und zwar in folgendem Sinne: Ist die Entscheidbarkeit von B geklärt, lässt sich daraus die Frage nach der Entscheidbarkeit von A beantworten.

Definition 3.8 *Die Menge $A \subseteq \mathbb{N}_0$ heißt* **reduzierbar** *auf die Menge $B \subseteq \mathbb{N}_0$, falls es eine total berechenbare Funktion $f : \mathbb{N}_0 \to \mathbb{N}_0$ gibt mit*

$$x \in A \text{ genau dann, wenn } f(x) \in B \text{ für alle } x \in \mathbb{N}_0$$

Ist A reduzierbar auf B (mittels der total berechenbaren Funktion f), so schreiben wir $A \leq B$ (oder $A \leq_f B$, falls die Funktion, mit der die Reduktion vorgenommen wird, genannt werden soll).

Gilt $A \leq B$, dann können wir die Entscheidbarkeit von A auf B (algorithmisch) transformieren: Ist B entscheidbar, d. h., gibt es ein Entscheidungsverfahren für B, dann ist auch A entscheidbar. Um zu entscheiden, ob $x \in A$ ist, berechnen wir $f(x)$ und wenden auf das Ergebnis das Entscheidungsverfahren für B an. Ist $f(x) \in B$, dann ist $x \in A$; ist $f(x) \notin B$, dann ist $x \notin A$. Der folgende Satz fasst diese Überlegung zusammen.

Satz 3.6 *Es sei $A, B \subseteq \mathbb{N}_0$. Ist $A \leq B$ und ist B entscheidbar oder semi-entscheidbar, dann ist auch A entscheidbar bzw. semi-entscheidbar.*

Beweis *Wir führen den Beweis nur für den Fall der Entscheidbarkeit, der Beweis für den Fall der Semi-Entscheidbarkeit ist analog.*

Es sei also $A \leq_f B$, $f : \mathbb{N}_0 \to \mathbb{N}_0$ eine total berechenbare Funktion, und B sei entscheidbar, d. h., χ_B, die charakteristische Funktion von B, ist berechenbar. Wir definieren die Funktion $g : \mathbb{N}_0 \to \mathbb{B}$ durch $g(x) = \chi_B(f(x))$. Für diese gilt:

- *g ist die charakteristische Funktion von A, denn es ist*

$$g(x) = \chi_B(f(x)) = \left\{ \begin{array}{ll} 1, & f(x) \in B \\ 0, & f(x) \notin B \end{array} \right\} = \left\{ \begin{array}{ll} 1, & x \in A \\ 0, & x \notin A \end{array} \right\} = \chi_A(x)$$

- *g ist berechenbar, denn f und χ_B sind berechenbar (siehe Satz 3.1).*

Mit der Reduktion f und der charakteristischen Funktion von B können wir also eine berechenbare charakteristische Funktion $\chi_A = g$ für A angeben. A ist also entscheidbar.

Wir werden Satz 3.6 des Öfteren im „negativen Sinne“ anwenden:

Folgerung 3.3 *Es sei $A, B \subseteq \mathbb{N}_0$. Ist $A \leq B$ und A nicht entscheidbar oder nicht semi-entscheidbar, dann ist auch B nicht entscheidbar bzw. nicht semi-entscheidbar.*

3.5 Turing-Verifizierer

In diesem Abschnitt führen wir eine weitere Variante von Turingmaschinen ein. Dazu betrachten wir als Erstes ein in der Theoretischen Informatik bedeutendes Entscheidungsproblem: *Das Erfüllbarkeitsproblem der Aussagenlogik*, das allgemein mit SAT (von englisch *Satisfiability*) bezeichnet wird.

SAT ist die Menge der erfüllbaren aussagenlogischen Formeln. Die Sprache $\mathcal{A}$ der aussagenlogischen Formeln wird gebildet aus Konstanten und Variablen, die mit aussagenlogischen Operatoren miteinander verknüpft werden.

Die Menge V der aussagenlogischen Variablen wird über dem Alphabet $\{v, |\}$ nach folgenden Regeln gebildet:

(1) v ist eine Variable: $v \in V$.

(2) Falls α eine Variable ist, dann auch $\alpha|$, d. h., ist $\alpha \in V$, dann ist auch $\alpha| \in V$.

Mithilfe dieser Regeln ergibt sich: $V = \{v|^n : n \in \mathbb{N}_0\}$. Wenn wir die Anzahl der Striche als Index notieren, ergibt sich als Variablenmenge: $V = \{v_0, v_1, v_2, \ldots\}$.

Die Menge der Operatorsymbole ist gegeben durch $O = \{0, 1, \neg, \wedge, \vee, (,)\}$. Über V und O werden die Elemente von $\mathcal{A}$, die **aussagenlogischen Formeln**, nach folgenden Regeln gebildet:

(i) Die aussagenlogischen Konstanten $0, 1 \in O$ sind Formeln: $0, 1 \in \mathcal{A}$.

(ii) Auch Variablen sind bereits Formeln: für jede Variable $\alpha \in V$ ist $\alpha \in \mathcal{A}$.

(iii) Sind $\alpha, \beta \in \mathcal{A}$, dann ist $(\alpha \wedge \beta)$, $(\alpha \vee \beta)$, $\neg\alpha \in \mathcal{A}$: Aus aussagenlogischen Formeln werden mithilfe von Operatorsymbolen und Klammern neue Formeln gebildet.

Beispiel 3.3 *Beispiele für aussagenlogische Formeln sind:*

$$v_1$$
$$\neg v_7$$
$$(v_5 \wedge v_3)$$
$$(((v_1 \vee (v_2 \wedge v_3)) \vee \neg(v_3 \wedge v_4)) \wedge \neg v_5)$$
$$((0 \vee v_1) \wedge 1)$$

Sei $\gamma \in \mathcal{A}$, dann bestimmen wir die Menge V_γ der in γ vorkommenden Variablen wie folgt:

(i) Ist $\gamma \in \{0, 1\}$, dann ist $V_\gamma = \{\}$.

(ii) Ist $\gamma \in V$, dann ist $V_\gamma = \{\gamma\}$.

(iii) Es ist $V_\gamma = V_\alpha$, falls $\gamma = \neg\alpha$ ist, sowie $V_\gamma = V_\alpha \cup V_\beta$, falls $\gamma = (\alpha \wedge \beta)$ oder $\gamma = (\alpha \vee \beta)$ ist.

Beispiel 3.4 *Mit diesen Regeln ergibt sich z. B. für die Formel*

$$\gamma = (((v_1 \vee (v_2 \wedge v_3)) \vee \neg(v_3 \wedge v_4)) \wedge \neg v_5)$$

die Variablenmenge

$$\begin{aligned} V_\gamma &= V_{((v_1 \vee (v_2 \wedge v_3)) \vee \neg(v_3 \wedge v_4))} \cup V_{\neg v_5} \\ &= V_{(v_1 \vee (v_2 \wedge v_3))} \cup V_{\neg(v_3 \wedge v_4)} \cup \{v_5\} \\ &= V_{v_1} \cup V_{(v_2 \wedge v_3)} \cup V_{(v_3 \wedge v_4)} \cup \{v_5\} \\ &= \{v_1\} \cup V_{v_2} \cup V_{v_3} \cup V_{v_3} \cup V_{v_4} \cup \{v_5\} \\ &= \{v_1\} \cup \{v_2\} \cup \{v_3\} \cup \{v_3\} \cup \{v_4\} \cup \{v_5\} \\ &= \{v_1, v_2, v_3, v_4, v_5\}. \end{aligned}$$

Die Bedeutung einer Formel, d. h. ihr Wahrheitswert, ergibt sich durch **Belegung der Variablen** mit Wahrheitswerten. Jeder Variablen v in γ, also jedem $v \in V_\gamma$, wird mit der Belegung $\mathcal{I}$ ($\mathcal{I}$ steht für *Interpretation*) genau ein Wahrheitswert zugewiesen: $\mathcal{I} : V_\gamma \to \mathbb{B}$. Für jede Variable $v \in V_\gamma$ gibt es zwei mögliche Belegungen: $\mathcal{I}(v) = 0$ oder $\mathcal{I}(v) = 1$. Ist n die Anzahl der Variablen in γ, also $|V_\gamma| = n$, dann gibt es 2^n mögliche Belegungen $\mathcal{I} : V_\gamma \to \mathbb{B}$. Diese fassen wir in der Menge

$$\mathcal{I}_\gamma = \mathbb{B}^{V_\gamma} = \{\mathcal{I} \mid \mathcal{I} : V_\gamma \to \mathbb{B}\}$$

zusammen. Mit einer gewählten Belegung $\mathcal{I} \in \mathcal{I}_\gamma$ wird die **Interpretation $\mathcal{I}^*(\gamma)$ der aussagenlogischen Formel $\gamma \in \mathcal{A}$** gemäß den folgenden Regeln berechnet:

(i) Für $\gamma \in \{0, 1\}$ sei $\mathcal{I}^*(0) = 0$ und $\mathcal{I}^*(1) = 1$: Die Konstanten werden unabhängig von der gegebenen Formel durch fest zugewiesene Wahrheitswerte interpretiert.

(ii) $\mathcal{I}^*(v) = \mathcal{I}(v)$: Die Variablen $v \in V_\gamma$ der Formel γ werden durch die gewählte Belegung $\mathcal{I}$ interpretiert.

(iii) Die Interpretation zusammengesetzter Formeln wird gemäß folgender Regeln für $\alpha, \beta \in \mathcal{A}$ berechnet:

Ist $\gamma = (\alpha \wedge \beta)$, dann ist $\mathcal{I}^*(\gamma) = \mathcal{I}^*(\alpha \wedge \beta) = \min\{\mathcal{I}^*(\alpha), \mathcal{I}^*(\beta)\}$.

Ist $\gamma = (\alpha \vee \beta)$, dann ist $\mathcal{I}^*(\gamma) = \mathcal{I}^*(\alpha \vee \beta) = \max\{\mathcal{I}^*(\alpha), \mathcal{I}^*(\beta)\}$.

Ist $\gamma = \neg\alpha$, dann ist $\mathcal{I}^*(\gamma) = \mathcal{I}^*(\neg\alpha) = 1 - \mathcal{I}^*(\alpha)$.

Beispiel 3.5 *Der besseren Lesbarkeit wegen nennen wir die Variablen in diesem Beispiel nicht v_1, v_2 und v_3, sondern p, q und r. Wir betrachten die Formel*

$$\gamma = (((p \vee (q \wedge r)) \wedge \neg(q \vee \neg r)) \vee 0).$$

Es ist $V_\gamma = \{p, q, r\}$. Wir wählen die Belegung $\mathcal{I}(p) = 1$, $\mathcal{I}(q) = 0$ sowie $\mathcal{I}(r) = 1$. Mit dieser Belegung ergibt sich gemäß den obigen Regeln folgende Interpretation: $\mathcal{I}^*(\gamma)$

$$
\begin{aligned}
&= \mathcal{I}^*(((p \vee (q \wedge r)) \wedge \neg(q \vee \neg r)) \vee 0) \\
&= \max\{\mathcal{I}^*((p \vee (q \wedge r)) \wedge \neg(q \vee \neg r)), \mathcal{I}^*(0)\} \\
&= \max\{\min\{\mathcal{I}^*(p \vee (q \wedge r)), \mathcal{I}^*(\neg(q \vee \neg r))\}, 0)\} \\
&= \max\{\min\{\max\{\mathcal{I}^*(p), \mathcal{I}^*(q \wedge r)\}, 1 - \mathcal{I}^*(q \vee \neg r)\}, 0\} \\
&= \max\{\min\{\max\{\mathcal{I}(p), \min\{\mathcal{I}^*(q), \mathcal{I}^*(r)\}\}, 1 - \max\{\mathcal{I}^*(q), \mathcal{I}^*(\neg r)\}\}, 0\} \\
&= \max\{\min\{\max\{1, \min\{\mathcal{I}(q), \mathcal{I}(r)\}\}, 1 - \max\{\mathcal{I}(q), 1 - \mathcal{I}^*(r)\}\}, 0\} \\
&= \max\{\min\{\max\{1, \min\{0, 1\}\}, 1 - \max\{0, 1 - \mathcal{I}(r)\}\}, 0\} \\
&= \max\{\min\{\max\{1, \min\{0, 1\}\}, 1 - \max\{0, 1 - 1\}\}, 0\} \\
&= \max\{\min\{\max\{1, 0\}, 1 - 0\}, 0\} \\
&= \max\{\min\{1, 1\}, 0\} \\
&= \max\{1, 0\} \\
&= 1
\end{aligned}
$$

Für die gewählte Belegung $\mathcal{I}$ ist der Wert der aussagenlogischen Formel also 1.

Übung 3.8 *Berechnen Sie den Wert der Formel γ für weitere der insgesamt $2^3 = 8$ möglichen Belegungen!*

Eine aussagenlogische Formel $\alpha \in \mathcal{A}$ heißt **erfüllbar**, falls es eine Belegung $\mathcal{I} \in \mathcal{I}_\alpha$ gibt mit $\mathcal{I}^*(\alpha) = 1$, d. h., wenn die Variablen von α so mit 0 oder 1 belegt werden können, dass die Auswertung von α insgesamt 1 ergibt. So ist die Formel γ aus obigem Beispiel erfüllbar, denn die dort gewählte Belegung macht die Formel wahr.

Die Formel $\alpha = (v \wedge \neg v)$ ist ein einfaches Beispiel für eine nicht erfüllbare Formel, denn sowohl für die Belegung $\mathcal{I}(v) = 1$ als auch für die Belegung $\mathcal{I}(v) = 0$ gilt $\mathcal{I}^*(\alpha) = 0$.

Die Menge

$$SAT = \{\alpha \in \mathcal{A} \mid \alpha \text{ ist erfüllbar}\}$$

ist die **Menge der erfüllbaren aussagenlogischen Formeln**. Nach dem derzeitigen Stand der Wissenschaft sind Entscheidungsverfahren, die feststellen, ob eine Formel erfüllbar ist oder nicht, sehr aufwändig: Im schlimmsten Fall müssen die Werte einer Formel mit n Variablen für alle 2^n möglichen Belegungen ausgerechnet werden. Führt mindestens eine Belegung zum Ergebnis 1, dann gehört die Formel zu SAT, ansonsten ist die Formel unerfüllbar.

Da in Anwendungen, z. B. in Steuerungen von Industrieanlagen, Flugzeugen oder medizinischen Geräten, logische Formeln mit 10 000 und mehr Variablen verwendet werden, müssten für ihren Test $2^{10\,000}$ Belegungen berechnet werden. Das würde selbst auf den aktuell schnellsten Rechnern mehrere Jahrhunderte dauern.

In der Berechenbarkeitstheorie hilft man sich bei Problemen dieser Art mit einem Konzept, das bereits Turing vorgeschlagen hat: Um die charakteristische Funktion einer Menge zu berechnen, kann eine Turingmaschine eine zusätzliche Information benutzen.

Definition 3.9 *Ein Turing-Akzeptor T heißt* **Verifizierer** *für $L \subseteq \mathbb{B}^*$, falls*

$$L = \{w \in \mathbb{B}^* \mid \textit{es existiert ein } x \in \mathbb{B}^* \textit{ mit } (w, x) \in L(T)\}. \tag{3.9}$$

x wird **Zertifikat** *(auch: Zeuge, Beweis) für w genannt.*

Wir nennen eine Menge, die von einem Verifizierer akzeptiert wird, **verifizierbar**. *Wir bezeichnen die* **Klasse der verifizierbaren Mengen mit** V.

Die Arbeitsweise eines Verifizierers T können wir uns so vorstellen: Zu einem gegebenen Wort w „rät“ T eine zusätzliche Information x (oder er befragt ein **„Orakel“** nach einem Hilfswort x), die auf ein zweites Arbeitsband geschrieben wird. Gibt es ein Wort x, mithilfe dessen T bei der Bearbeitung von (w, x) eine 1 ausgibt, dann gehört w zu L. Gibt es kein Zertifikat x, sodass T bei Eingabe (w, x) eine 1 ausgibt, dann ist $w \notin L$.

Ein Verifizierer T_{SAT} für SAT könnte wie folgt arbeiten: Zu einer (mit einer geeigneten Codierung) eingegebenen aussagenlogischen Formel α mit n Variablen $v_1, \ldots, v_n$ rät T_{SAT} eine Bitfolge $x \in \mathbb{B}^n$ und belegt für $1 \leq i \leq n$ die Variable v_i mit dem i-ten Bit $x[i]$ von x. Dann wertet T_{SAT} die Formel α mit dieser Belegung aus.

Bei Verifizierern wird davon ausgegangen, dass sie, wenn es für eine Eingabe w ein Zertifikat x gibt, ein solches auch als Erstes erraten. Ist eine aussagenlogische Formel α mit n Variablen erfüllbar, dann gibt es eine Belegung $x \in \mathbb{B}^n$, die α erfüllt, und T_{SAT} wird bei Eingabe α ein solches x „spontan erraten“ und nach der Überprüfung, ob x erfüllend ist, eine 1 ausgeben. Gibt es kein Zertifikat für α, d. h., es gibt keine erfüllende Belegung x für α, dann gibt T_{SAT} eine 0 aus.

3.6 Äquivalenz von Turing-Akzeptoren und Verifizierern

Es stellt sich nun die Frage, ob Verifizierer mächtiger sind als Akzeptoren. Es ist klar, dass jeder „normale“ Akzeptor auch ein Verifizierer ist; Akzeptoren benutzen kein Orakel. Wir können einen Akzeptor T in einen äquivalenten Verifizierer T_V transformieren, indem bei jeder Eingabe nur das leere Wort als Hilfswort zugelassen ist, und jede Zustandsüberführung $(s, a, s', b, m) \in \delta_T$ wird in die Überführung

$(s, a, \#, s', b, m, -) \in \delta_{T_V}$ transformiert. Der Verifizierer T_V ist eine 2-Bandmaschine, die auf dem ersten Band die Maschine T ausführt und auf dem zweiten nichts tut, da der S-/L-Kopf dort stehen bleibt. Dann gilt $w \in L(T)$ genau dann, wenn $(w, \varepsilon) \in L(T_V)$, und damit ist $L(T) = L(T_V)$. Es gilt also die nächste Folgerung.

Folgerung 3.4 $\mathsf{RE} \subseteq \mathsf{V}$.

Gilt auch die Umkehrung $\mathsf{V} \subseteq \mathsf{RE}$ und damit $\mathsf{RE} = \mathsf{V}$, oder gibt es verifizierbare Sprachen, die nicht rekursiv-aufzählbar sind, gilt also $\mathsf{RE} \subset \mathsf{V}$?

Satz 3.7 *Es gilt* $\mathsf{RE} = \mathsf{V}$.

Beweis *Aus Folgerung 3.4 wissen wir, dass* $\mathsf{RE} \subseteq \mathsf{V}$ *gilt. Wir müssen also noch überlegen, ob auch* $\mathsf{V} \subseteq \mathsf{RE}$ *gilt. Sei* $L \in \mathsf{V}$ *und* T_V *ein Verifizierer für* L*. Ein Akzeptor* T_A *für* L *könnte den Verifizierer* T_V *bei einer Eingabe* w *wie folgt simulieren:* T_A *erzeugt auf einem zweiten Arbeitsband sukzessive in längenlexikografischer Ordnung alle Wörter* x *über* $\mathbb{B}$*. Für jedes* x *führt* T_A *die Maschine* T_V *auf der Eingabe* (w, x) *aus. Akzeptiert* T_V *diese Eingabe, dann akzeptiert auch* T_A *und stoppt. Falls* T_V *die Eingabe nicht akzeptiert, bestimmt* T_A *den Nachfolger* $suc(x)$ *von* x *und führt* T_V *auf der Eingabe* $(w, suc(x))$ *aus. Falls es ein Zertifikat* $x \in \mathbb{B}^*$ *für* w *gibt, d. h., es ist* $(w, x) \in L(T_V)$ *und damit* $w \in L$*, dann findet* T_A *dieses und akzeptiert* w*, womit* $w \in L(T_A)$ *gilt. Gibt es kein Zertifikat für* w*, d. h., es ist* $w \notin L$*, dann sucht* T_A *immer weiter und stoppt nicht.* T_A *akzeptiert damit* w *nicht, es ist also* $w \notin L(T_A)$*. Insgesamt folgt* $L = L(T_A)$ *und damit* $L \in \mathsf{RE}$.

3.7 Nicht deterministische Turingmaschinen

Sei T ein Verifizierer mit der Überführungsfunktion δ. Für ein Eingabewort $w \in \mathbb{B}^*$ ist jedes Wort $x \in \mathbb{B}^*$ ein mögliches Zertifikat. Deshalb kann es zu einem Zustand s und einem Buchstaben $a \in \mathbb{B}$ zwei Elemente $b \in \mathbb{B}$ mit $(s, a, b) \in Def(\delta)$ geben. Wir definieren eine neue Zustandsüberführung δ' durch

$$\delta'(s, a) = \{\delta(s, a, b) \mid (s, a, b) \in Def(\delta)\}.$$

Sei T' die Turingmaschine, die entsteht, wenn wir δ durch δ' ersetzen. Die Funktion δ' ist eine mengenwertige Funktion. Deshalb können bei Abarbeitung eines Wortes w durch T' Konfigurationen jeweils bis zu zwei Folgekonfigurationen haben.

Turingmaschinen, bei denen die Zustandsüberführung eine mengenwertige Funktion ist, werden **nicht deterministisch** genannt. Eine nicht deterministische Turingmaschine akzeptiert ein Wort, falls es mindestens eine Konfigurationsfolge gibt, die mit einer Ausgabekonfiguration endet.

Allgemein ist also bei **nicht deterministischen Turingmaschinen** die Zustandsüberführung δ eine mengenwertige Funktion

$$\delta : (S - \{t_a, t_r\}) \times \Gamma \to 2^{S \times \Gamma \times \{\leftarrow, \rightarrow, -\}}.$$

Falls für ein Paar $(s, a) \in S \times \Gamma$ die Übergänge $(s, a, s_j, b_j, m_j) \in \delta$ für $1 \leq j \leq k$ mit $k \geq 0$ definiert sind, können wir dies auf folgende Weisen notieren:

$$(s, a, s_1, b_1, m_1), \ldots, (s, a, s_k, b_k, m_k)$$

oder

$$(s, a, \{(s_1, b_1, m_1), \ldots, (s_k, b_k, m_k)\})$$

oder

$$\delta(s, a) = \{(s_1, b_1, m_1), \ldots, (s_k, b_k, m_k)\}.$$

Es ist möglich, dass $\delta(s, a) = \emptyset$ ist; dies entspricht dem Fall $k = 0$. Das bedeutet, dass es für die Konfigurationen $\alpha s a \beta$ keine Folgekonfigurationen gibt, d. h., die Maschine hält an. Ist $\alpha s a \beta$ keine Ausgabekonfiguration, ist Φ_T für die Eingabe, die zu dieser Konfiguration geführt hat, nicht definiert.

Konfigurationen und Konfigurationsübergänge sind für nicht deterministische Maschinen genau so definiert wie für deterministische – wohl mit dem Unterschied, dass eine Konfiguration $\alpha s a \beta$ für den Fall, dass $|\delta(s, a)| > 1$ ist, mehrere Folgekonfigurationen haben kann.

Die Abarbeitung eines Wortes $w \in \mathbb{B}^*$ durch eine nicht deterministische Turingmaschine $T = (\Gamma, S, \delta, s_0, \#, t_a, t_r)$ kann durch einen **Konfigurationsbaum** (auch: **Berechnungsbaum**) $Tree_T(w)$ repräsentiert werden. Die Knoten dieses Baums stellen Konfigurationen dar; die Startkonfiguration $s_0 w$ ist die Wurzel von $Tree_T(w)$. Stellt ein Knoten die Konfiguration $\alpha s a \beta$ dar und ist $|\delta(s, a)| \geq 1$, dann stellen die Folgekonfigurationen die Nachfolger des Knotens $\alpha s a \beta$ dar. Ist $|\delta(s, a)| = 0$, dann stellt $\alpha s a \beta$ ein Blatt des Baums dar. Ist $s = t_a$, dann heißt dieses Blatt **akzeptierendes Blatt**, und der Weg von der Wurzel dorthin heißt **akzeptierender Pfad** von w. Ist $s = t_r$, dann heißen das Blatt und der Pfad **verwerfend**. Da eine Startkonfiguration $s_0 w$ zu nicht endenden Konfigurationsfolgen führen kann, kann $Tree_T(w)$ auch unendlich lange Pfade enthalten (die selbstverständlich keine Ausgabe berechnen).

Eine nicht deterministische Turingmaschine T akzeptiert das Wort $w \in \mathbb{B}^*$ genau dann, wenn $Tree_T(w)$ (mindestens) einen akzeptierenden Pfad enthält. Es ist

$$L(T) = \{w \in \mathbb{B}^* \mid T \text{ akzeptiert } w\}$$

die von T akzeptierte Sprache; T heißt **nicht deterministischer Turing-Akzeptor** für L. Eine Sprache $L \subseteq \mathbb{B}^*$ wird von T **akzeptiert**, falls $L = L(T)$ gilt.

NTA ist die **Klasse der Sprachen, die von nicht deterministischen Turingmaschinen akzeptiert werden**.

Eine nicht deterministische Turingmaschine heißt **nicht deterministischer Turing-Entscheider** für eine Sprache $L \subseteq \mathbb{B}^*$ genau dann, wenn $L = L(T)$ ist und die

Berechnungsbäume $Tree_T(w)$ für alle $w \in \mathbb{B}^*$ keine unendlichen Pfade enthalten. Es gilt also: Ist $w \in L(T)$, dann enthält $Tree_T(w)$ (mindestens) einen akzeptierenden Pfad; ist $w \notin L(T)$, dann sind alle Pfade von $Tree_T(w)$ verwerfend.

NTE ist die **Klasse der Sprachen, die von nicht deterministischen Turingmaschinen entschieden werden**.

Offensichtlich gelten die folgenden Aussagen:

Folgerung 3.5 **a)** NTE $\subseteq$ NTA.

b) DTE $\subseteq$ NTE *sowie* DTA $\subseteq$ NTA.

Eine Übungsaufgabe, die den Wunsch nach nicht deterministischen Turingmaschinen wecken kann, ist die Folgende.

Übung 3.9 *Konstruieren Sie eine Turingmaschine, welche die Menge*

$$L = \{ww \mid w \in \mathbb{B}^*\}$$

entscheidet!

Es ist klar, dass jede deterministische Turingmaschine T durch eine nicht deterministische Maschine T' simuliert werden kann. Denn deterministische Maschinen sind Spezialfälle von nicht deterministischen, bei denen $|\delta(s,a)| \leq 1$ für alle $s \in S$ und alle $a \in \Gamma$ gilt. Der folgende Satz besagt, dass auch die Umkehrung gilt. Nicht deterministische Maschinen sind also nicht mächtiger als deterministische; beide Varianten berechnen dieselbe Klasse von Funktionen.

Satz 3.8 *Zu jeder nicht deterministischen Turingmaschine existiert eine äquivalente deterministische Turingmaschine T'.*

Beweisskizze *Wir wollen die Transformation einer nicht deterministischen Turingmaschine T in eine äquivalente deterministische Maschine T' nicht formal angeben, sondern nur die Idee einer Transformationsmöglichkeit skizzieren. Im Kern geht es darum, dass T' für eine Eingabe $w \in \mathbb{B}^*$ den Konfigurationsbaum $Tree_T(w)$ in Breitensuche durchläuft und so lange T simuliert, bis eine Haltekonfiguration erreicht wird. Diese Strategie kann man wie folgt realisieren: Da T nicht deterministisch ist, gibt es für jedes Eingabesymbol möglicherweise mehrere Übergangsmöglichkeiten, d. h., für jedes Paar $(s,a) \in S \times \Gamma$ gibt es $|\delta(s,a)|$ viele Möglichkeiten; es sei $k = \max\{|\delta(s,a)| : (s,a) \in S \times \Gamma\}$ die maximale Anzahl solcher Möglichkeiten. Jede endliche Folge von Zahlen zwischen* 1 *und k stellt eine Folge von nicht deterministischen Auswahlentscheidungen dar.*

T' kann nun als 3-Bandmaschine konstruiert werden. Auf Band 1 wird die Eingabe geschrieben, auf Band 2 werden systematisch, d. h. geordnet nach der Länge und innerhalb gleicher Längen in nummerischer Ordnung, alle mit den Zahlen $1, \ldots, k$ bildbaren endlichen Folgen generiert. Für jede dieser Folgen kopiert T' die Eingabe von Band 1 auf Band 3. Anschließend simuliert T' die Maschine T auf Band 3. Dabei benutzt T' die Zahlenfolge auf Band 2, um in jedem Schritt die durch die entsprechende Zahl festgelegte (ursprünglich nicht deterministische) Übergangsmöglichkeit von T auszuführen.

Wenn T bei Eingabe von x eine Ausgabe y berechnet, dann gibt es eine Konfigurationsfolge von der Startkonfiguration in eine Ausgabekonfiguration mit Ausgabe y, bei der bei jedem Konfigurationsübergang von mehreren möglichen ein richtiger Zustandsübergang gewählt wird. Es gibt also eine endliche Folge von Zahlen, die die richtige Folge von Konfigurationsübergängen repräsentiert. Diese Folge wird auf jeden Fall von T' irgendwann auf Band 2 generiert. Dann simuliert T' diese Konfigurationsfolge von T und berechnet ebenfalls y.

Wenn T zu einer Eingabe x keine Ausgabe berechnet, gibt es keine Konfigurationsfolge, die bei einer Ausgabekonfiguration endet. Dann gibt es auch keine Folge von Zahlen, die eine Folge von Konfigurationsübergängen repräsentiert, die zu einer Ausgabekonfiguration führt, und damit berechnet auch T' bei Eingabe x keine Ausgabe.

Aus den Sätzen 3.7 und 3.8 folgt unmittelbar:

Folgerung 3.6 *Sei $L \subseteq \mathbb{B}^*$. Dann sind folgende Aussagen äquivalent:*

a) $L \in \mathsf{RE}$.

b) *Es existiert ein Turing-Akzeptor T mit $L = L(T)$.*

c) *Es existiert ein Verifizierer T mit $L = L(T)$.*

d) *Es existiert eine nicht deterministische Turingmaschine T mit $L = L(T)$.*

e) $\mathsf{RE} = \mathsf{DTA} = \mathsf{V} = \mathsf{NTA}$.

f) $\mathsf{DTE} = \mathsf{NTE}$.

3.8 Zusammenfassung und bibliografische Hinweise

In diesem Kapitel werden die Begriffe Algorithmus und Berechenbarkeit von Funktionen sowie Entscheidbarkeit, Semi-Entscheidbarkeit und rekursive Aufzählbarkeit von Mengen mathematisch präzisiert. Als Konzept für Berechenbarkeit wird die Turingmaschine gewählt. Die Churchsche These besagt, dass die Turing-Berechenbarkeit genau das intuitive Verständnis von Berechenbarkeit umfasst, da sich alle anderen Berechenbarkeitskonzepte als äquivalent zur Turing-Berechenbarkeit herausgestellt haben. Damit sind die Klassen $\mathcal{R}$ der total berechenbaren Funktionen sowie die Klasse $\mathcal{P}$ der (partiell) berechenbaren Funktionen unabhängig vom Berechenbarkeitskonzept festgelegt.

Als spezielle, aus theoretischen und praktischen Gesichtspunkten bedeutende Varianten werden deterministische und nicht deterministische Turingmaschinen sowie Turing-Verifizierer vorgestellt. Ihre Äquivalenz wird bewiesen.

Es werden wesentliche Eigenschaften der Klasse R der entscheidbaren Sprachen und der Klasse RE der rekursiv aufzählbaren Sprachen, die sich als identisch mit der Klasse der semi-entscheidbaren Sprachen herausstellt, bewiesen und ihre Zusammenhänge aufgezeigt.

Das Kapitel ist in Teilen an entsprechende Darstellungen in [VW16] angelehnt. Dort wird mehr oder weniger ausführlich erläutert, wie die aufgelisteten Varianten von Turingmaschinen in äquivalente mit nur einem Arbeitsband transformiert werden können. Des Weiteren werden dort neben der Turing-Berechenbarkeit weitere Berechenbarkeitskonzepte wie μ-rekursive Funktionen, While- und Goto-Programme ausführlich vorgestellt, und ihre Äquivalenz wird bewiesen. Außerdem werden die Klasse der Loop-berechenbaren und die dazu identische Klasse der primitiv-rekursiven Funktionen betrachtet, und es wird gezeigt, dass diese Klasse eine echte Teilklasse der total berechenbaren Funktionen bildet.

Einführungen in die Theorie formaler Sprachen und Berechenbarkeit geben auch [AB02], [BL74], [HS01], [HMU13], [Hr14], [Koz97], [LP98], [Ro67], [Schö09], [Si06] und [Wei87].

Kapitel 4

Laufzeit-Komplexität

Insbesondere für praktische Probleme, wie z. B. für SAT, ist von Bedeutung, welchen Aufwand ihre Berechnung kostet. Aufwand kann sein: Zeitaufwand, Speicheraufwand oder Kommunikationsaufwand, wenn die Berechnung auf verteilten Systemen durchgeführt wird. Wir wollen uns in diesem Abschnitt (einführend) mit dem Zeitaufwand von Entscheidungsalgorithmen beschäftigen bzw. mit dem zeitlichen Mindestaufwand, der notwendig ist, um ein Entscheidungsproblem zu lösen.

4.1 Die O-Notation

Dabei interessiert uns bei einem Entscheidungsproblemen nicht für jedes Wort w der exakte Aufwand für die Feststellung, ob es zur Sprache gehört, sondern die Größenordnung des Aufwands. Um diese auszudrücken, kann man die sogenannte O-Notation verwenden.

Definition 4.1 *Sei* $f : \mathbb{N}_0 \to \mathbb{R}_{\geq 0}$. *Dann ist*

$$\mathsf{O}(f) = \{g : \mathbb{N}_0 \to \mathbb{R}_{\geq 0} \mid \textit{es existieren ein } n_0 \in \mathbb{N}_0 \textit{ und ein } c \in \mathbb{N}, \textit{ sodass } g(n) \leq cf(n) \textit{ für alle } n \geq n_0 \textit{ gilt}\}$$

die Menge der Funktionen, die jeweils ab einem Argument nicht stärker wachsen als die Funktion f, *abgesehen von einer multiplikativen Konstante. Anstelle von* $g \in \mathsf{O}(f)$ *schreiben wir* $g = \mathsf{O}(f)$, *um auszudrücken, dass „g (höchstens) von der Ordnung* f *ist“.*

K.-U. Witt und M. E. Müller, *Algorithmische Informationstheorie*,
https://doi.org/10.1007/978-3-662-61694-9_4

Beispiel 4.1 *Es gilt $3n^2+7n+11 = \mathsf{O}(n^2)$, denn wir können wie folgt abschätzen:*

$$\begin{aligned} 3n^2 + 7n + 11 &\leq 3n^2 + 7n^2 + 11n^2 && \text{für alle } n \geq 1 \\ &\leq 11n^2 + 11n^2 + 11n^2 && \text{für alle } n \geq 1 \\ &= 33n^2 \end{aligned}$$

Es gibt also ein $c = 33 \geq 1$ und ein $n_0 = 1 \geq 0$, sodass $3n^2 + 7n + 11 \leq c \cdot n^2$ für alle $n \geq n_0$ gilt. Damit ist gezeigt, dass $3n^2 + 7n + 11 = \mathsf{O}(n^2)$ ist: Das Polynom zweiten Grades $3n^2 + 7n + 11$ wächst im Wesentlichen wie n^2.

Die Funktion $p_2(n) = 3n^2 + 7n + 11$ ist ein Beispiel für ein Polynom zweiten Grades über $\mathbb{N}_0$. Allgemein sind **Polynome k-ten Grades** über $\mathbb{N}_0$ Funktionen

$$p_k : \mathbb{N}_0 \to \mathbb{R}_{\geq 0},$$

definiert durch

$$\begin{aligned} p_k(n) &= a_k n^k + a_{k-1} n^{k-1} + \ldots + a_1 n + a_0 \\ &= \sum_{i=0}^{k} a_i n^i,\ a_i \in \mathbb{R}_{\geq 0},\ 0 \leq i \leq k,\ k \geq 0,\ a_k > 0. \end{aligned}$$

Satz 4.1 *Es gilt $p_k(n) = \mathsf{O}(n^k)$ für alle $k \in \mathbb{N}_0$.*

Beweis *Wir schätzen $p_k(n)$ ähnlich ab, wie wir die Funktion in Beispiel 4.1 abgeschätzt haben:*

$$\begin{aligned} p_k(n) &= a_k n^k + a_{k-1} n^{k-1} + \ldots + a_1 n + a_0 \\ &\leq a_k n^k + a_{k-1} n^k + \ldots + a_1 n^k + a_0 n^k && \text{für alle } n \geq 1 \\ &\leq \underbrace{a n^k + a n^k + \ldots + a n^k + a n^k}_{k+1\text{-mal}} && \text{mit } a = \max\{a_i \mid 0 \leq i \leq k\} \\ &= a(k+1) n^k \end{aligned}$$

Wir wählen $c = a(k+1)$ sowie $n_0 = 1$. Dann gilt $p_k(n) \leq c \cdot n^k$ für alle $n \geq n_0$ und damit $p_k(n) = \mathsf{O}(n^k)$.

Mithilfe des Satzes können wir unmittelbar die Aussage in Beispiel 4.1 folgern.

Gilt $f(n) = \mathsf{O}(p_k(n))$ für eine Funktion $f : \mathbb{N}_0 \to \mathbb{R}_{\geq 0}$, dann nennen wir f von **polynomieller Ordnung**. In späteren Betrachtungen interessiert bei polynomieller Ordnung einer Funktion f oft nicht ihre genaue Größenordnung, sprich der Grad des betreffenden Polynoms, sondern, dass es sich bei der Größenordnung überhaupt um

eine polynomielle handelt. In diesen Fällen schreiben wir $f = \mathsf{O}(\mathsf{poly})$ oder, wenn das Argument eine Rolle spielt, $f(\cdot) = \mathsf{O}(\mathsf{poly}(\cdot))$.

Satz 4.2 *Seien $d \in \mathbb{R}_{\geq 0}$ sowie $f, g, h : \mathbb{N}_0 \to \mathbb{R}_{\geq 0}$, dann gilt:*

(1) $d = \mathsf{O}(1)$,

(2) $f = \mathsf{O}(f)$,

(3) $d \cdot f = \mathsf{O}(f)$,

(4) $f + g = \mathsf{O}(g)$, *falls* $f = \mathsf{O}(g)$,

(5) $f + g = \mathsf{O}(\max\{f, g\})$,

(6) $f \cdot g = \mathsf{O}(f \cdot h)$, *falls* $g = \mathsf{O}(h)$.

Beweis *(1) folgt unmittelbar aus Satz 4.1, denn Konstanten sind Polynome vom Grad* 0.

(2) Aus $f(n) \leq 1 \cdot f(n)$ für alle $n \geq 0$ folgt sofort die Behauptung.

(3) Nach (2) gilt $f = \mathsf{O}(f)$, woraus folgt, dass eine Konstante $c > 0$ und ein $n_0 \in \mathbb{N}_0$ existieren mit $f(n) \leq c \cdot f(n)$ für alle $n \geq n_0$. Daraus folgt $d \cdot f(n) \leq d \cdot c \cdot f(n)$ für alle $n \geq n_0$. Es gibt also eine Konstante $c' = d \cdot c > 0$, sodass $d \cdot f(n) \leq c' \cdot f(n)$ für alle $n \geq n_0$ ist. Daraus folgt $d \cdot f = \mathsf{O}(f)$.

(4) Aus $f = \mathsf{O}(g)$ folgt, dass ein $c > 0$ und ein $n_0 \in \mathbb{N}_0$ existieren mit $f(n) \leq c \cdot g(n)$ für alle $n \geq n_0$. Daraus folgt

$$\begin{aligned} f(n) + g(n) &\leq c \cdot g(n) + g(n) && \text{für alle } n \geq n_0 \\ &= (c+1) \cdot g(n) && \text{für alle } n \geq n_0 \\ &= c' \cdot g(n) && \text{für alle } n \geq n_0 \text{ und } c' = c + 1. \end{aligned}$$

Somit gilt $f + g = \mathsf{O}(g)$.

(5) $\max\{f, g\}$ *ist definiert durch*

$$\max\{f, g\}(n) = \begin{cases} f(n), & \text{falls } f(n) \geq g(n), \\ g(n), & \text{sonst}. \end{cases}$$

Hieraus folgt

$$f(n) + g(n) \leq \max\{f, g\}(n) + \max\{f, g\}(n) = 2 \cdot \max\{f, g\}(n)$$

für alle $n \geq 0$. Somit gilt $f + g = \mathsf{O}(\max\{f, g\})$.

(6) Aus $g = \mathsf{O}(h)$ folgt, dass es ein $c > 0$ und ein $n_0 \in \mathbb{N}_0$ gibt mit $g(n) \leq c \cdot h(n)$ für alle $n \geq n_0$. Daraus folgt $f(n) \cdot g(n) \leq f(n) \cdot c \cdot h(n) = c \cdot f(n) \cdot h(n)$ für alle $n \geq n_0$. Somit gilt $f \cdot g = \mathsf{O}(f \cdot h)$.

Die O-Notation ermöglicht asymptotische Aussagen über obere Schranken. Dual dazu kann man auch eine Notation für untere Schranken, die sogenannte Ω-Notation, einführen. Stimmen für ein gegebenes Problem obere und untere Schranke überein, wird dies durch die sogenannte Θ-Notation ausgedrückt. Da wir im Folgenden diese beiden Notationen nicht benötigen, gehen wir hier nicht weiter darauf ein.

Neben der Komplexität (insbesondere der Laufzeit) von Algorithmen interessiert auch die Komplexität von Problemen, d. h. die Komplexität, die ein Lösungsalgorithmus für ein Problem mindestens hat. So kann man z. B. zeigen, dass das allgemeine Sortierproblem mit Verfahren, deren Grundoperation der paarweise Vergleich von Objekten ist, mindestens $\mathsf{O}(n \log n)$ Schritte benötigt (dabei ist n die Anzahl der zu sortierenden Objekte). Das allgemeine Sortierproblem kann nicht schneller gelöst werden. Da man Sortieralgorithmen mit solchen Laufzeiten kennt, wie z. B. Quicksort, Mergesort und Heapsort, braucht man sich im Prinzip nicht auf die Suche nach weiteren Algorithmen zu machen, da es keinen Sortieralgorithmus mit einer größenordnungsmäßig kleineren Laufzeit gibt. Hat ein Algorithmus eine Laufzeit, die größenordnungsmäßig mit der Komplexität des Problems übereinstimmt, so nennt man ihn **optimal** (für dieses Problem). Heapsort ist also ein optimaler Sortieralgorithmus.

Es ist im Allgemeinen schwierig, die Komplexität eines Problems zu bestimmen bzw. nachzuweisen, dass ein Algorithmus ein Problem optimal löst, da man über eine Klasse von Algorithmen als Ganzes argumentieren muss. Bei einem entsprechenden Nachweis muss man möglicherweise Laufzeitüberlegungen für Algorithmen anstellen, die man noch gar nicht kennt.

4.2 Die Komplexitätsklassen P und NP

Sei T ein Turing-Entscheider. Dann ist die Funktion

$$time_T : \mathbb{B}^* \to \mathbb{N}_0$$

definiert durch

$$time_T(w) = \text{Anzahl der Konfigurationsübergänge von } T \text{ bei Eingabe } w.$$

Wir fassen Mengen, deren Elemente innerhalb einer bestimmten Zeitschranke entschieden werden, zu einer **Komplexitätsklasse** zusammen.

Definition 4.2 *Für eine Funktion $f : \mathbb{N}_0 \to \mathbb{N}_0$ ist*

$$TIME(f) = \{L \subseteq \mathbb{B}^* \mid \textit{es existiert ein Turing-Entscheider } T \textit{ mit } L = L(T) \textit{ und } time_T(w) = \mathsf{O}(f(|w|)) \textit{ für alle } w \in L\}$$

die Klasse der Mengen, deren Elemente w mit einer Laufzeit der Größenordnung $f(|w|)$ entschieden werden.

Ist $L \in TIME(f)$, dann heißt L **in f-Zeit entscheidbar**.

In praktischen Anwendungen sind in der Regel nur Lösungsverfahren von Interesse, deren Laufzeit – gemessen in der Größe der Eingaben – polynomiell ist. Deshalb ist die **Klasse**

$$\mathsf{P} = \bigcup_{k \in \mathbb{N}_0} TIME(p_k) \tag{4.1}$$

der Mengen, die in Polynomzeit entschieden werden können, von Bedeutung.

Wir übertragen die obigen Definitionen auf nicht deterministische Entscheidungen: Sei T ein nicht deterministischer Turing-Entscheider. Dann sei die Funktion

$$ntime_T : \mathbb{B}^* \to \mathbb{N}_0$$

definiert durch

$$ntime_T(w) = \text{minimale Länge eines akzeptierenden Pfades in } Tree_T(w).$$

Definition 4.3 *Für eine Funktion $f : \mathbb{N}_0 \to \mathbb{N}_0$ ist*

$$NTIME(f) = \{L \subseteq \mathbb{B}^* \mid \text{es existiert ein nicht deterministischer Turing-Entscheider } T \text{ mit } L = L(T) \text{ und } ntime_T(w) = \mathsf{O}(f(|w|)) \text{ für alle } w \in L\}$$

die Klasse der Mengen, deren Elemente nicht deterministisch mit einer Laufzeit der Größenordnung $f(|w|)$ entschieden werden.

Ist $L \in NTIME(f)$, dann heißt L **nicht deterministisch in f-Zeit entscheidbar**.

Damit ergibt sich die **Klasse**

$$\mathsf{NP} = \bigcup_{k \in \mathbb{N}_0} NTIME(p_k) \tag{4.2}$$

der Mengen, die nicht deterministisch in Polynomzeit entschieden werden können.

Definition 4.4 *Ein Turing-Entscheider T für $L \subseteq \mathbb{B}^*$ heißt* **Polynomzeit-Verifizierer**, *falls*

$$L = \{w \in \mathbb{B}^* \mid \text{es existiert ein } x \in \mathbb{B}^* \text{ mit } (w, x) \in L(T), |x| = \mathsf{O}(\mathsf{poly}(|w|)),\ time_T(w, x) = \mathsf{O}(\mathsf{poly}(|w|))\}. \tag{4.3}$$

Eine Menge $L \subseteq \mathbb{B}^*$ heißt **Polynomzeit-verifizierbar**, *falls es einen Polynomzeit-Verifizierer für L gibt.*

Die Klasse

$$\mathsf{VP} = \{L \subseteq \mathbb{B}^* \mid L \text{ ist in Polynomzeit verifizierbar}\} \tag{4.4}$$

ist die Klasse aller Mengen, die von Polynomzeit-Verifizierern entschieden werden können.

Für die folgenden Betrachtungen benötigen wir noch die **Klasse**

$$\mathsf{EXPTIME} = \bigcup_{k \in \mathbb{N}} TIME(2^{p_k}) \tag{4.5}$$

der Mengen, die deterministisch in Exponentialzeit entschieden werden können.

Für $L \in \mathsf{EXPTIME}$ gilt also: Es gibt einen deterministischen Entscheider T für L mit

$$time_T(w) = \mathsf{O}\left(2^{\mathsf{poly}(|w|)}\right) \tag{4.6}$$

für alle $w \in L$.

Satz 4.3 *Es ist* $\mathsf{NP} = \mathsf{VP}$.

Beweis *Sei p ein Polynom, sodass $L \in NTIME(p)$ ist, T eine nicht deterministische Turingmaschine, die L in p-Zeit entscheidet, $w \in \mathbb{B}^*$ sowie g der maximale Verzweigungsgrad im Konfigurationsbaum $Tree_T(w)$. Es gilt:*

(1) Jeder innere Knoten k in $Tree_T(w)$ hat maximal g Nachfolger k_j, $1 \leq j \leq g$.

(2) Ein Pfad der Länge m in $Tree_T(w)$ kann codiert werden durch eine Folge $c_0, c_1, \ldots, c_{m-1}$ mit $1 \leq c_j \leq g$, wobei c_j angibt, dass vom Knoten k_j zu dessen c_j-ten Nachfolger $k_{j_{c_j}}$ gegangen werden soll.

(3) Ist $w \in L$ und sind $k_0, k_1, \ldots, k_n$, $n \geq 0$, die Knoten einer minimalen akzeptierenden Konfigurationsfolge für w in $Tree_T(w)$. Dann gilt $n = p(|w|)$.

Wir können einen Verifizierer V_T wie folgt konstruieren: Als Zertifikate für eine Eingabe $w \in \mathbb{B}^*$ werden Folgen $x = c_0, c_1, \ldots, c_{n-1}$ mit $1 \leq c_j \leq g$ gewählt. V_T führt auf w die Konfigurationsfolge aus, die sich aus der gewählten Folge x in $Tree_T(w)$ ergibt. Ist $w \notin L$, dann gibt es keine Folge, die einen akzeptierenden Pfad darstellt, d. h., es gibt kein Zertifikat für w. Ist $w \in L$, dann gibt es mindestens eine Folge, die einen akzeptierenden Pfad darstellt, d. h., es gibt ein Zertifikat und damit ein minimales Zertifikat x für w. Wegen (3) gilt $|x| = p(|w|)$. Insgesamt folgt, dass V_T ein Polynomzeit-Verifizierer für L ist. Damit gilt $L \in \mathsf{VP}$, und $\mathsf{NP} \subseteq \mathsf{VP}$ ist gezeigt.

Sei $L \in \mathsf{VP}$ und V ein Polynomzeit-Verifizierer für L. Wir konstruieren zu V eine nicht deterministische Turingmaschine T_V, indem wir für alle $a \in \Gamma_V$ die Zustandsüberführung von T_V definieren durch

$$\delta_{T_V}(s,a) = \{\delta_V(s,a,b) \mid b \in \Gamma_V\}. \tag{4.7}$$

Sei $w \in L$ und x ein Zertifikat für w, dann definieren die Buchstaben von x gemäß (4.7) einen akzeptierenden Pfad in $Tree_{T_V}(w)$. Da x durch die Länge von w polynomiell beschränkt ist, gilt dies auch für den akzeptierenden Pfad, d. h., w wird von T_V in Polynomzeit akzeptiert. Ist $w \notin L$, dann existiert auch kein Zertifikat für w, und damit gibt es in $Tree_{T_V}(w)$ keinen akzeptierenden Pfad. Es folgt $L = L(T_V)$ und damit $L \in \mathsf{NP}$, womit $\mathsf{VP} \subseteq \mathsf{NP}$ gezeigt ist.

Da wir $\mathsf{NP} = \mathsf{VP}$ gezeigt haben, werden wir im Folgenden die Klasse VP der Sprachen, die von Polynomzeit-Verifizierern entschieden werden, auch – wie es allgemein üblich ist – mit NP bezeichnen.

In Satz 3.8 haben wir gesehen, dass jede nicht deterministische Turingmaschine in eine äquivalente deterministische transformiert werden kann. Dabei spielt die Komplexität keine Rolle. Wenn wir diese berücksichtigen, dann gilt der folgende Satz.

Satz 4.4 $\mathsf{NP} \subseteq \mathsf{EXPTIME}$.

Beweis *Sei $L \in \mathsf{NP}$ und T ein Polynomzeit-Verifizierer, der L entscheidet. Sei p das Polynom, das die Länge der Zertifikate sowie die Laufzeit von T beschränkt. Wir können daraus einen deterministischen Entscheider T_d für L wie folgt konstruieren: T_d erzeugt bei Eingabe eines Wortes $w \in \Sigma^*$ (systematisch) alle möglichen Zertifikate x der Länge $|x| = p(|w|)$ und führt T jeweils auf das Paar (w,x) aus. Akzeptiert T dieses Paar, dann akzeptiert T_d die Eingabe w und stoppt. Akzeptiert T das Paar (w,x) nicht, dann erzeugt T_d das nächste Zertifikat x' und führt T auf das Paar (w,x') aus. Im schlimmsten Fall muss T auf alle möglichen Paare ausgeführt werden. Falls T keines dieser Paare akzeptiert, dann akzeptiert T_d letztendlich die Eingabe w nicht und stoppt.*

Sei $n = |w|$, dann gibt es $2^{p(n)}$ mögliche Zertifikate x mit $|x| = p(n)$. Die Anzahl der Runden, die T_d bei Eingabe von w möglicherweise durchführen muss, ist also von der Ordnung $2^{p(n)}$, und in jeder Runde führt T_d den Verifizierer T aus, der dafür jeweils $p(n)$ Schritte benötigt. Daraus folgt $time_{T_d}(w) = \mathsf{O}(2^{p(n)} \cdot p(n))$. Sei $grad(p) = k$. Es ist $n^k \leq 2^{n^k}$, damit gilt

$$time_{T_d}(w) = \mathsf{O}\left(2^{n^k} \cdot n^k\right) = \mathsf{O}\left(2^{n^k} \cdot 2^{n^k}\right) = \mathsf{O}\left(2^{2n^k}\right) = \mathsf{O}\left(2^{p_k(n)}\right).$$

Damit ist gemäß (4.5) $L \in \mathsf{EXPTIME}$.

Die Transformation einer nicht deterministischen Turingmaschine in eine äquivalente deterministische führt also im Allgemeinen dazu, dass die Laufzeiten auf der deterministischen Maschine im Vergleich zu denen auf der nicht deterministischen exponentiell anwachsen können.

Da offensichtlich $\mathsf{P} \subseteq \mathsf{NP}$ gilt, folgt mit dem obigen Satz:

Folgerung 4.1 $\mathsf{P} \subseteq \mathsf{NP} \subseteq \mathsf{EXPTIME}$.

Ob auch $\mathsf{NP} \subseteq \mathsf{P}$ und damit die Gleichheit $\mathsf{P} = \mathsf{NP}$ gilt, oder ob $\mathsf{P} \subset \mathsf{NP}$ und damit $\mathsf{P} \neq \mathsf{NP}$ gilt, ist *das* ungelöste Problem der Theoretischen Informatik. Es wird das **P-NP-Problem** genannt. Wir betrachten im Folgenden einige grundlegende Aspekte dieses Problems.

4.3 NP-Vollständigkeit

Im Abschnitt 3.4 haben wir den Begriff der Reduktion (siehe Definition 3.8) als ein Hilfsmittel kennengelernt, mit dem die Entscheidbarkeit einer Sprache mithilfe einer anderen Sprache gezeigt werden kann. Dabei spielt der Aufwand für die Reduktion keine Rolle. Da wir in diesem Kapitel an der Komplexität von Entscheidungen interessiert sind, müssen wir die Komplexität von Reduktionen beachten.

Definition 4.5 *Es sei $A, B \subseteq \mathbb{N}_0$. A heißt* **polynomiell reduzierbar** *auf B genau dann, wenn $A \leq_f B$ gilt und die Reduktion f in Polynomzeit berechenbar ist. Ist A auf B polynomiell reduzierbar, so schreiben wir $A \leq_{\mathsf{poly}} B$.*

Es muss also mindestens eine deterministische Turingmaschine T_f geben, die f in Polynomzeit berechnet, d. h., für die $\Phi_{T_f} = f$ gilt mit $time_{T_f}(n) = \mathsf{O}(\mathsf{poly}(n))$.

Folgerung 4.2 *Es seien $A, B, C \subseteq \mathbb{N}_0$.*

a) *Es gilt $A \leq_{\mathsf{poly}} B$ genau dann, wenn $\overline{A} \leq_{\mathsf{poly}} \overline{B}$ gilt.*

b) *Die Relation $\leq_{\mathsf{poly}}$ ist transitiv: Gilt $A \leq_{\mathsf{poly}} B$ und $B \leq_{\mathsf{poly}} C$, dann gilt auch $A \leq_{\mathsf{poly}} C$.*

Übung 4.1 *Beweisen Sie Folgerung 4.2!*

Folgender Satz besagt, dass ein Problem, das polynomiell auf ein in polynomieller Zeit lösbares Problem reduzierbar ist, selbst in polynomieller Zeit lösbar sein muss.

Satz 4.5 *Sei $A, B \subseteq \mathbb{N}_0$. Ist $A \leq_{\mathsf{poly}} B$ und ist $B \in \mathsf{P}$, dann ist auch $A \in \mathsf{P}$.*

Beweis *Aus $A \leq_{\mathsf{poly}} B$ folgt, dass es eine deterministisch in Polynomzeit berechenbare Funktion f geben muss mit $A \leq_f B$. Da $B \in \mathsf{P}$ ist, ist die charakteristische Funktion χ_B von B deterministisch in Polynomzeit berechenbar. Wir wissen aus dem Beweis von Satz 3.6, dass die Komposition von f und χ_B eine charakteristische Funktion für A ist: $\chi_A = \chi_B \circ f$. Da f und χ_B deterministisch in Polynomzeit berechenbar sind, ist auch χ_A deterministisch in Polynomzeit berechenbar, also ist $A \in \mathsf{P}$.*

Die Aussage von Satz 4.5 gilt entsprechend auch für die Klasse NP.

Satz 4.6 *Sei $A, B \subseteq \mathbb{N}_0$. Ist $A \leq_{\mathsf{poly}} B$ und ist $B \in \mathsf{NP}$, dann ist auch $A \in \mathsf{NP}$.*

Übung 4.2 *Beweisen Sie Satz 4.6!*

Die folgende Definition legt – auch im Hinblick auf die P-NP-Frage – eine wichtige Teilklasse von NP fest, die Klasse der NP-vollständigen Mengen.

Definition 4.6 **a)** *Eine Menge A heißt* **NP- schwierig** *genau dann, wenn für alle Mengen $B \in \mathsf{NP}$ gilt: $B \leq_{\mathsf{poly}} A$.*

b) *A heißt* **NP-vollständig** *genau dann, wenn $A \in \mathsf{NP}$ und A NP-schwierig ist.*

c) *Wir bezeichen mit* NPC *die* **Klasse der NP-vollständigen Mengen** (NPC *steht für* NP complete).

NPC enthält im Hinblick auf den zeitlichen Aufwand die schwierigsten Entscheidungsprobleme innerhalb von NP. Wenn man zeigen könnte, dass eine Menge aus NPC deterministisch polynomiell entschieden werden kann, dann wäre $\mathsf{P} = \mathsf{NP}$.

Satz 4.7 *Ist $A \in \mathsf{NPC}$, dann gilt $A \in \mathsf{P}$ genau dann, wenn $\mathsf{P} = \mathsf{NP}$ gilt.*

Beweis *„⇒“: Sei $A \in \mathsf{NPC}$ und $A \in \mathsf{P}$. Wir müssen zeigen, dass dann $\mathsf{P} = \mathsf{NP}$ gilt. Da $\mathsf{P} \subseteq \mathsf{NP}$ sowieso gilt, müssen wir also nur noch zeigen, dass auch $\mathsf{NP} \subseteq \mathsf{P}$ gilt.*

Sei also B eine beliebige Menge aus NP. Da A NP-vollständig ist, ist A auch NP-schwierig (siehe Definition 4.6 b), und deshalb gilt $B \leq_{\mathsf{poly}} A$ (siehe Definition 4.6 a).

Da $A \in \mathsf{P}$ ist, folgt mit Satz 4.5, dass auch $B \in \mathsf{P}$ ist. Da B beliebig aus NP gewählt ist, folgt somit $\mathsf{NP} \subseteq \mathsf{P}$ und damit insgesamt $\mathsf{P} = \mathsf{NP}$.

„⇐“: Wir müssen zeigen: Ist $A \in \mathsf{NPC}$ und ist $\mathsf{P} = \mathsf{NP}$, dann ist $A \in \mathsf{P}$. Das ist aber offensichtlich: Da A NP-vollständig ist, ist $A \in \mathsf{NP}$ (siehe Definition 4.6 b). Wenn $\mathsf{P} = \mathsf{NP}$ ist, ist damit auch $A \in \mathsf{P}$.

Satz 4.7 besagt: Um zu beweisen, dass $\mathsf{P} = \mathsf{NP}$ oder dass $\mathsf{P} \neq \mathsf{NP}$ ist, reicht es, für *irgendeine* NP-vollständige Menge zu zeigen, dass sie in P bzw. nicht in P liegt.

Es herrscht überwiegend die Ansicht, dass $\mathsf{P} \neq \mathsf{NP}$ und damit $\mathsf{P} \subset \mathsf{NP}$ gilt; es gibt aber auch Meinungen, die $\mathsf{P} = \mathsf{NP}$ für möglich halten. Wir gehen im folgenden Abschnitt 4.4 auf mögliche Bedeutungen und Auswirkungen der beiden möglichen Antworten ein.

Der folgende Satz ist die Grundlage für eine Standardtechnik zum Nachweis der NP-Vollständigkeit einer Menge.

Satz 4.8 *Sei $A \in \mathsf{NPC}$. Gilt dann für eine Menge B zum einen $B \in \mathsf{NP}$ und zum anderen $A \leq_{\mathsf{poly}} B$, so ist auch $B \in \mathsf{NPC}$.*

Beweis *Aufgrund unserer Voraussetzungen ist (nach Definition 4.6 a) nur noch zu zeigen, dass für alle Mengen $C \in \mathsf{NP}$ gilt: $C \leq_{\mathsf{poly}} B$. Sei also $C \in \mathsf{NP}$ beliebig, so gilt zunächst $C \leq_{\mathsf{poly}} A$, da $A \in \mathsf{NPC}$ ist. Damit gilt aber $C \leq_{\mathsf{poly}} A \leq_{\mathsf{poly}} B$ und somit aufgrund der Transitivität von $\leq_{\mathsf{poly}}$ (siehe Folgerung 4.2 b) auch $C \leq_{\mathsf{poly}} B$. Also ist auch $B \in \mathsf{NPC}$.*

4.4 Einige Beispiele für NP-vollständige Mengen

Die sich aus Satz 4.8 unmittelbar ergebende Technik zum Nachweis der NP-Vollständigkeit einer Menge B lautet also:

1. Zeige, dass $B \in \mathsf{NP}$ gilt.
2. Reduziere eine Sprache $A \in \mathsf{NPC}$ polynomiell auf B.

Man benötigt somit zumindest *ein* konkretes NP-vollständiges Problem als „Startpunkt“ für Ketten polynomieller Reduktionen.

Am Ende von Abschnitt 3.5 haben wir den Verifizierer T_{SAT} für SAT informell beschrieben. T_{SAT} ist ein Polynomzeit-Verifizierer: Enthält eine aussagenlogische Formel α die Variablen $v_1, \ldots, v_n$ und ist $x \in \mathbb{B}^n$, dann kann T_{SAT} in Polynomzeit überprüfen, ob x ein Zertifikat für α ist, d. h. überprüfen, ob die Belegung $\mathcal{I}(v_i) = x[i]$, $1 \leq i \leq n$, die Formel α erfüllt. Somit gilt $SAT \in \mathsf{NP}$. Wenn zudem gezeigt werden kann, dass $A \leq_{\mathsf{poly}} SAT$ für alle $A \in \mathsf{NP}$ gilt, d. h., dass SAT NP-schwierig ist,

dann gilt $SAT \in \mathsf{NPC}$. SAT ist dann die erste als NP-vollständig gezeigte Menge, die dann als Ausgangspunkt für Beweise gemäß Satz 4.8 dienen kann.

Dass SAT NP-schwierig – und damit NP-vollständig – ist, zeigt der folgende **Satz von Cook**.

Satz 4.9 $SAT \in \mathsf{NPC}$.

Beweisidee *Es muss gezeigt werden, dass jede Menge $A \in \mathsf{NP}$ in Polynomzeit auf SAT reduziert werden kann. Sei T eine nicht deterministische Turingmaschine, die A entscheidet. Die Abarbeitung eines Elementes $n \in \mathbb{N}_0$ durch T kann mithilfe einer aussagenlogischen Formel $\alpha_T(n)$ beschrieben werden, die genau dann erfüllbar ist, wenn n von T akzeptiert wird. Des Weiteren gilt: Ist $time_T(n) = \mathsf{O}(p_k(n))$, dann ist $|\alpha_T(n)| = \mathsf{O}(p_{3k}(n))$. Der Reduktionsalgorithmus transformiert also die in der Länge durch ein Polynom vom Grad k beschränkte Konfigurationsfolge für die Eingabe n in eine Formel, die in der Länge durch ein Polynom vom Grad $3k$ beschränkt ist. Es gilt somit $A \leq_{\mathsf{poly}} SAT$.*

Wenn man nachweist, dass ein Problem NP-vollständig ist, bedeutet dies – vorausgesetzt, es ist $\mathsf{P} \neq \mathsf{NP}$ –, dass das Problem zwar prinzipiell, aber praktisch nicht lösbar ist. Denn seine deterministische Berechnung – und Berechnungen auf realen Rechnern erfolgen aufgrund deterministischer Programme – benötigt eine Laufzeit von der Ordnung 2^{n^k}. Das bedeutet schon für $k = 1$ und relativ kleine Problemgrößen n auch für schnellste heute verfügbare Rechner Laufzeiten, die größer sind, als unser Universum alt ist.

NP-vollständige Probleme gelten also als praktisch nicht lösbar (nicht effizient, aber effektiv). Probleme in P gelten als praktisch lösbar (effizient), wobei die Laufzeiten ihrer Berechnungen de facto höchstens von der Ordnung $\mathsf{O}(n^3)$ sein sollten, denn bei Laufzeiten von $\mathsf{O}(n^k)$ mit $k \geq 4$ können die Laufzeiten auch schon für nicht allzu große n unakzeptabel werden.

Wir listen im Folgenden einige NP-vollständige Probleme auf, die als Abstraktionen von Problemen gelten können, wie sie in der täglichen Praxis häufig vorkommen, wie z. B. aussagenlogische Formeln in Normalform, Routenprobleme, Planungsprobleme, Zuordnungs- und Verteilungsprobleme.

kSAT

Aussagenlogische Formeln $\alpha \in \mathcal{A}$ sind in **konjunktiver Normalform**, falls sie aus Konjunktionen von Disjunktionen bestehen, also von der Gestalt

$$\alpha = \bigwedge_{i=1}^{n} \left(\bigvee_{j=1}^{k_i} v_{ij} \right) \tag{4.8}$$

sind; dabei sind die v_{ij} sogenannte Literale, das sind Variable oder negierte Variable. Die geklammerten Disjunktionen heißen Klauseln. Wir nennen Formeln in der Gestalt (4.8) KNF-Formeln. Man kann jede Formel β in eine äquivalente KNF-Formel β' transformieren. Äquivalent bedeutet, dass für jede Belegung $\mathcal{I}$ der Variablen, die dieselben in β und β' sind, $\mathcal{I}^*(\beta) = \mathcal{I}^*(\beta')$ ist.

Die Sprachen $kSAT \subset SAT$ bestehen aus genau den KNF-Formeln, deren Klauseln höchstens k Literale enthalten (in (4.8) ist dann also $k_i \leq k$, $1 \leq i \leq n$). Man kann zeigen, dass $2SAT \in \mathsf{P}$, aber $3SAT \in \mathsf{NPC}$ gilt; aus Letzterem folgt unmittelbar $kSAT \in \mathsf{NPC}$ für $k \geq 3$. Da SAT in NP liegt, liegt auch $3SAT$ in NP. Jede KNF-Formel kann mit polynomiellem Aufwand in eine äquivalente $3SAT$-Formel transformiert werden; es gilt also $SAT \leq_{\mathsf{poly}} 3SAT$, woraus $3SAT \in \mathsf{NPC}$ folgt.

HAM

HAM bezeichnet das Hamilton-Kreis-Problem: Gegeben sei ein ungerichteter Graph $G = (K, E)$ mit Knotenmenge $K = \{1, \ldots, n\}$ und Kantenmenge $E \subseteq K \times K$. Der Graph G gehört zu HAM genau dann, wenn G einen geschlossenen Pfad enthält, in dem alle Knoten von G genau ein Mal vorkommen (ein solcher Pfad wird Hamilton-Kreis genannt). Dass HAM in NP liegt, kann durch einen Polynom-Verifizierer gezeigt werden, der eine Permutation $k_1, \ldots, k_n$ von K „rät“ und überprüft, ob $(k_i, k_{i+1}) \in E$ für $1 \leq i \leq n-1$ sowie $(k_n, k_1) \in E$ gilt. Die Überprüfung kann in Polynomzeit erfolgen. Zum Nachweis der NP-Vollständigkeit kann eine polynomielle Reduktion von $3SAT$ auf HAM angegeben werden: $3SAT$-Formeln α können so in einen ungerichteten Graphen G_α transformiert werden, dass α erfüllbar ist genau dann, wenn G_α einen Hamilton-Kreis besitzt.

TSP

Angenommen, es sind n Orte zu besuchen. Dann ist für viele Anwendungen von Interesse, ob es eine Rundtour gibt, bei der man alle Städte besuchen kann, dabei zum Ausgangsort zurückkehrt und eine bestimmte Gesamtkilometergrenze b nicht überschreitet, weil sonst die Tour nicht in einer vorgegebenen Zeit (oder mit einer bestimmten elektrischen Ladung oder zu einem vorgegebenen Preis) zu schaffen wäre. Dieses Problem heißt Traveling-Salesman-Problem, abgekürzt mit TSP. Ein solches Problem lässt sich mithilfe von bewerteten ungerichteten Graphen $G = (K, E, c)$ mathematisch beschreiben. Dabei ist $K = \{1, \ldots, n\}$, $E \subseteq K \times K$ sowie $c : E \to \mathbb{R}_+$ eine Funktion, die jeder Kante $(i, j) \in E$ die Kosten $c(i, j)$ zuordnet.

Die Sprache TSP besteht aus allen Paaren (G, b), wobei G ein bewerteter Graph und $b \in \mathbb{N}$ ist, sodass G einen Hamilton-Kreis $k_1, \ldots, k_n$ besitzt, dessen Kosten höchstens b betragen, d. h., für den

$$\sum_{j=1}^{n-1} c(k_i, k_{i+1}) + c(k_n, k_1) \leq b \tag{4.9}$$

gilt. Es gilt $TSP \in \mathsf{NP}$, denn ein Verifizierer kann – wie bei HAM – überprüfen, ob eine Permutation von K ein Hamilton-Kreis ist, und falls ja, feststellen, ob dieser Kreis die Bedingung (4.9) erfüllt. Des Weiteren kann HAM wie folgt in polynomieller Zeit auf TSP reduziert werden: Sei $G = (K, E)$ ein (ungerichteter) Graph. Daraus konstruieren wir den bewerteten Graphen $G' = (K, E, c)$ mit

$$c(i,j) = \begin{cases} 1, & (i,j) \in E, \\ \infty, & \text{sonst.} \end{cases}$$

Durch Setzen der oberen Schranke auf $b = n$ zeigt sich, dass HAM ein Spezialfall von TSP ist: Wenn TSP in Polynomzeit entscheidbar wäre, dann auch HAM.

Wir haben jetzt eine erste Kette von Reduktionen gezeigt, nämlich

$$SAT \leq_{\mathsf{poly}} 3SAT \leq_{\mathsf{poly}} HAM \leq_{\mathsf{poly}} TSP.$$

Damit steht uns bereits eine Reihe von Kandidaten zur Verfügung, mit deren Hilfe sich weitere NP-Vollständigkeitsbeweise führen lassen. Wir geben im Folgenden einige weitere Probleme an, die auf diese Weise als schwierig, d. h. zu NPC gehörig, klassifiziert werden können.

RUCKSACK

Angenommen, man hat einen Behälter (z. B. einen Rucksack, eine Schachtel, einen Container, einen Lastwagen) einer bestimmten Größe $b \in \mathbb{N}_0$ sowie $k \geq 1$ Gegenstände mit den Größen $a_1, \ldots, a_k \in \mathbb{N}_0$. Gibt es dann eine Teilmenge der Gegenstände, die genau in den Behälter passen? Mathematisch ausgedrückt lautet die Fragestellung: Gibt es eine Teilmenge $I \subseteq \{1, \ldots, k\}$, sodass

$$\sum_{i \in I} a_i = b \tag{4.10}$$

gilt? Auch hier kann man sich einen Polynomzeit-Verifizierer überlegen, der überprüft, ob eine geratene Teilmenge $I \subseteq \{1, \ldots, k\}$ die Bedingung (4.10) erfüllt. Des Weiteren kann man $3SAT$ polynomiell auf $RUCKSACK$ transformieren. Somit gilt $RUCKSACK \in \mathsf{NPC}$.

PARTITION

Gegeben seien k Größen $a_1, \ldots, a_k \in \mathbb{N}$. Können diese in zwei gleich große Teilmengen aufgeteilt werden, d. h., gibt es eine Teilmenge $I \subseteq \{1, \ldots, k\}$, sodass

$$\sum_{i \in I} a_i = \sum_{i \notin I} a_i \tag{4.11}$$

ist? Es gilt $PARTITION \in \mathsf{NP}$, und $RUCKSACK$ kann mit polynomiellem Aufwand auf $PARTITION$ reduziert werden, woraus $PARTITION \in \mathsf{NPC}$ folgt.

BIN PACKING

Es seien n Behälter der gleichen Größe b gegeben sowie k Gegenstände $a_1, \ldots, a_k$. Können die Gegenstände so auf die Behälter verteilt werden, dass kein Behälter überläuft? Mathematisch formuliert lautet die Fragestellung: Gibt es eine Zuordnung

$$pack : \{1, \ldots, k\} \to \{1, \ldots, n\},$$

die jedem Gegenstand i, $1 \leq i \leq k$, einen Behälter $pack(i) = j$, $1 \leq j \leq n$, zuordnet, sodass für jeden Behälter j

$$\sum_{pack(i)=j} a_i \leq b \tag{4.12}$$

gilt? Zur Entscheidung von $BIN\ PACKING$ kann ein Polynomzeit-Verifizierer konstruiert und $PARTITION$ kann polynomiell auf $BIN\ PACKING$ reduziert werden, woraus $BIN\ PACKING \in \mathsf{NPC}$ folgt.

* * *

Mittlerweise ist von Tausenden von Problemen bewiesen worden, dass sie NP-vollständig sind, sodass man geneigt sein könnte anzunehmen, dass möglicherweise $\mathsf{NP} = \mathsf{NPC}$ ist, dass also alle Sprachen in NP auch NP-vollständig sind. Der folgende Satz, den wir ohne Beweis angeben, besagt, dass es für den Fall, dass $\mathsf{P} \neq \mathsf{NP}$ ist, Sprachen gibt, die weder in P noch in NPC liegen.

Satz 4.10 *Ist* $\mathsf{P} \neq \mathsf{NP}$*, dann ist* $\mathsf{NP} - (\mathsf{P} \cup \mathsf{NPC}) \neq \emptyset$.

Diese Sprachen werden auch NP-*unvollständig* genannt, und dementsprechend wird die Klasse dieser Sprachen mit NPI bezeichnet (NPI steht für NP *incomplete*).

Übung 4.3 *Für eine Komplexitätsklasse* C *sei* $\mathsf{coC} = \left\{\overline{A} \mid A \in \mathsf{C}\right\}$ *die komplementäre Klasse von* C*. Zeigen Sie, dass folgende Aussagen gelten:*

a) *Ist* C *eine deterministische Komplexitätsklasse, dann gilt* $\mathsf{C} = \mathsf{coC}$.

b) *Es ist* $\mathsf{P} = \mathsf{coP}$.

c) *Es gilt* $\mathsf{P} \subseteq \mathsf{NP} \cap \mathsf{coNP}$.

d) *Gilt* $\mathsf{P} = \mathsf{NP}$*, dann gilt auch* $\mathsf{NP} = \mathsf{coNP}$.

e) *Es gilt* $\mathsf{P} = \mathsf{NP}$ *genau dann, wenn* $\mathsf{NPC} \cap \mathsf{coNP} \neq \emptyset$.

Würde der Nachweis $\mathsf{NP} \neq \mathsf{coNP}$ gelingen, dann folgt aus Übung 4.3 d), dass $\mathsf{P} \neq \mathsf{NP}$ wäre, und das P-NP-Problem wäre gelöst. Die Frage, ob $\mathsf{NP} = \mathsf{coNP}$ ist, ist

ebenso offen wie die P-NP-Frage. Da bisher kein einziges NP-vollständiges Problem gefunden wurde, dessen Komplement in NP enthalten ist, wird dies als Indiz für $\mathsf{NP} \neq \mathsf{coNP}$ angesehen. Ob die Frage $\mathsf{NP} = \mathsf{coNP}$ äquivalent zur Frage $\mathsf{P} = \mathsf{NP}$ ist, also auch die Umkehrung der Aussage d) gilt, ist heutzutage ebenfalls noch ungelöst. Es könnte auch $\mathsf{NP} = \mathsf{coNP}$ sein und trotzdem $\mathsf{P} \neq \mathsf{NP}$ gelten.

Bild 4.1 illustriert den Zusammenhang zwischen den Klassen P, NP und NPC und ihren Komplementärklassen, soweit er sich nach heutigem Wissensstand darstellt.

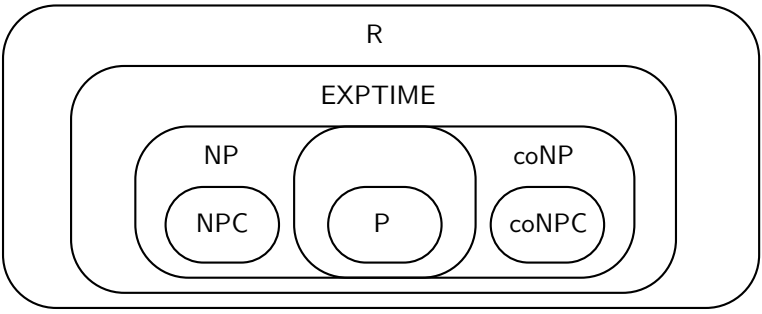

Abbildung 4.1: Beziehungen zwischen Zeit-Komplexitätsklassen

4.5 Bemerkungen zur P-NP-Frage

Einige der oben genannten Probleme, z. B. $RUCKSACK$, $BIN\ PACKING$ und TSP, sind mathematische Verallgemeinerungen von Problemen, die in der täglichen Praxis in vielfältigen Ausprägungen vorkommen. Zu diesen Problemen gehören insbesondere Scheduling-Probleme, d. h. Zuordnungs- und Planungsprobleme.

So müssen etwa Aufträge unterschiedlichen Umfangs zum Fertigen von Werkstücken zur Verfügung stehenden Maschinen unterschiedlicher Kapazität so zugeordnet werden, dass ihre Fertigung in einem vorgegebenen Kostenrahmen möglich ist. Diese Probleme müssen in der industriellen Fertigung in der Arbeitsvorbereitung gelöst werden. Transportunternehmen haben ähnliche Probleme: Transportaufträge unterschiedlicher Größe müssen möglichst kostengünstig auf die zur Verfügung stehenden Ladekapazitäten aufgeteilt werden.

Scheduling-Programme in Betriebssystemen haben die Aufgabe, zur Ausführung anstehende Anwendungsprogramme so den zur Verfügung stehenden Prozessoren zuzuordnen, dass ein optimaler Systemdurchsatz erreicht wird.

Das Aufstellen eines Stundenplans, bei dem Klassen von unterschiedlicher Größe, Klassenräume unterschiedlicher Größe, Fächer oder Fächerkombimationen in unterschiedlichem Umfang, Lehrpersonen mit unterschiedlichen Fachkombinationen usw.

so zugeordnet werden müssen, dass alle Schülerinnen und Schüler den vorgeschriebenen Unterricht bekommen, die Lehrpersonen ihr Lehrdeputat erfüllen, dabei möglicherweise ihren Wünschen (Zeiten, Fächern) Rechnung getragen wird, Klassenräume optimal ausgenutzt, auf keinen Fall überbelegt werden usw., gehört ebenfalls – in seiner allgemeinen Form – zu den NP-vollständigen Problemen.

Wenn $\mathsf{P} = \mathsf{NP}$ wäre, dann wären diese Probleme effizient lösbar, worüber sich viele Anwender freuen würden. Andererseits könnte aber die Schwierigkeit von Problemen nicht mehr benutzt werden, um bestimmte Eigenschaften zu sichern, wie etwa die Vertraulichkeit von Datenübertragungen mithilfe kryptografischer Protokolle, deren Sicherheit etwa auf der Existenz von Einwegfunktionen basiert, die es im Falle $\mathsf{P} = \mathsf{NP}$ nicht geben würde. In der Konsequenz wären derzeit wesentliche Internettechnologien unbrauchbar.

Was wäre, wenn z. B. bewiesen würde, dass ein NP-vollständiges Problem A mindestens von der Ordnung $n^{2^{2^{100}}}$ ist? Damit wäre $\mathsf{P} = \mathsf{NP}$ gezeigt, und alle Bücher der Theoretischen Informatik müssten umgeschrieben werden, aber aus praktischen Gesichtspunkten hätte sich die Situation nicht geändert. Es könnte auch sein, dass jemand beweist, dass eine Konstante d existiert, sodass A höchsten von der Ordnung n^d ist, ohne d konkret angeben zu können. Das P-NP-Problem wäre gelöst, aber weiter nicht viel gewonnen. Es könnte aber auch sein, dass A mindestens so aufwändig ist wie 1.2^n oder gar nur 1.0001^n, dann wäre $\mathsf{P} \neq \mathsf{NP}$ bewiesen, was aber für praktische Problemstellungen wenig relevant wäre (z. B. ist $1.2^{100} \approx 8.3 \cdot 10^7$). Möglich ist ebenso, dass A mindestens so aufwändig wie $n^{\log\log\log n}$ ist, die Laufzeit also superpolynomiell ist. Auch damit wäre $\mathsf{P} \neq \mathsf{NP}$ gezeigt, was auch in diesem Falle praktisch kaum von Bedeutung ist (z. B. für $n \leq 10^9$ ist $\log\log\log n < 2.3$). Prinzipiell wäre es auch möglich, dass A kein einheitliches Laufzeitverhalten hat: Für eine unendliche Teilmenge von A könnte die Laufzeit polynomiell, für eine andere Teilmenge exponentiell sein.

Unabhängig von diesen Spekulationen gibt es Möglichkeiten, schwierige Probleme zu lösen, z. B. mithilfe von Heuristiken und Randomisierung. Solche Ansätze werden heute schon weit verbreitet in der Praxis angewendet, denn diese kann nicht warten, bis das P-NP-Problem theoretisch gelöst ist, und obige Betrachtungen deuten an, das auch theoretische Lösungen praktisch nicht hilfreich sein müssen. Für einige Problemklassen liefern sogar deterministische Algorithmen in vielen Fällen in vertretbarer Zeit Lösungen. So wurden schon in den sechziger Jahren des vorigen Jahrhunderts SAT-Solver entwickelt, die das SAT-Problem in vielen Fällen effizient lösen. Mittlerweile sind in diesem Gebiet durch Weiterentwicklungen solcher Solver große Fortschritte erzielt worden. Das Erfüllbarkeitsproblem von aussagenlogischen Formeln ist in wichtigen Anwendungsbereichen von wesentlicher Bedeutung, wie z. B. bei Schaltwerken in Prozessoren oder Steuerungen von Fahr- und Flugzeugen. Hier können durchaus Schaltungen mit mehreren Hundert Variablen vorkommen, deren 2^{100x} Belegungen nicht alle getestet werden können. Hier werden in jüngerer Zeit randomisierte Ansätze zum Finden von geeigneten Zertifikaten untersucht, mit denen mit hoher Wahrscheinlichkeit fehlerhafte Schaltungen entdeckt werden können.

Mengen, die zu P bzw. zu NP gehören, besitzen folgende wesentliche Eigenschaften:

(1) Eine Menge B gehört zu P genau dann, wenn es einen deterministischen Algorithmus A gibt, der in Polynomzeit berechnet, ob ein Element w zu B gehört oder nicht. Das bedeutet, dass A die korrekte Antwort (ohne zusätzliche Information) aus der Eingabe w berechnet.

(2) Eine Menge B gehört zu NP genau dann, wenn es einen (deterministischen) Polynomzeit-Verifizierer V gibt, der mithilfe einer zusätzlichen Information (Beweis, Zertifikat) x in Polynomzeit überprüft, ob w zur Sprache gehört.

Was ist der wesentliche Unterschied? Der Unterschied ist, dass für die Entscheidung der Sprachen in P die Entscheider keiner Hilfe bedürfen, um die korrekte Antwort effizient, d. h. in polynomieller Zeit, zu berechnen. Ein Verifizierer hingegen berechnet die Antwort nicht alleine aus der Eingabe, sondern er überprüft (in polynomieller Zeit) die Zugehörigkeit der Eingabe zu einer Menge und benutzt dabei die Hilfe einer außerhalb stehenden mächtigen Instanz, d. h., er ist nicht in der Lage, selbst das Problem effizient zu lösen. Man kann sagen, dass der Verifizierer zu dumm ist, zu wenig Ideen hat, nicht kreativ genug ist, um die Lösung zu bestimmen. Man nennt diese Art der Entscheidung auch **Guess & Check-Prinzip**: Ein – woher auch kommender – Lösungsvorschlag kann effizient auf Korrektheit überprüft, aber nicht effizient kreiert werden.

Wäre $\mathsf{P} \neq \mathsf{NP}$, dann benötigte man also Kreativität, um schwierige Probleme zu lösen oder Behauptungen zu beweisen. Algorithmen alleine sind dann nicht in der Lage, solche Probleme (in akzeptabler Zeit) zu lösen; sondern es bedarf einer kreativen Instanz, etwa menschlicher Intelligenz, um Lösungen zu erarbeiten. Das bedeutet im Umkehrschluss: Wenn $\mathsf{P} = \mathsf{NP}$ wäre, dann könnte das Programmieren und das Beweisen von Sätzen automatisiert werden. Die Kreativität, einen Beweis zu entdecken, würde nicht mehr benötigt, sondern Beweise könnten von Algorithmen erzeugt werden. Alles, was maschinell überprüft werden kann, könnte auch maschinell erschaffen werden. Diese Konsequenz wird von manchen Leuten sogar auf die Kunst ausgedehnt: Finde ich z. B. ein Musikstück schön, d. h., ich überprüfe quasi seine Schönheit, dann kann ich es auch erschaffen.

4.6 Zusammenfassung und bibliografische Hinweise

Die Anzahl der Konfigurationsübergänge wird als Laufzeitmaß für das Akzeptieren von Sprachen zugrunde gelegt. Da man an der Größenordnung von Laufzeiten interessiert ist, wird die O-Notation eingeführt.

Sowohl aus theoretischer als auch aus praktischer Sicht sind die Klassen P und NP von besonderer Bedeutung. P ist die Klasse der von deterministischen Turingmaschinen in Polynomzeit entscheidbaren Mengen. NP ist die Klasse der von nicht deterministischen Turingmaschinen in Polynomzeit entscheidbaren Mengen. Es wird gezeigt, dass NP gleich der Klasse der von Polynomzeit-Verifizierern entscheidbaren Mengen

ist. Die Klasse NPC der NP-vollständigen Mengen enthält die am schwersten innerhalb von NP entscheidbaren Mengen. Die Menge SAT der erfüllbaren aussagenlogischen Formeln wurde als Erste als zugehörig zu dieser Klasse gezeigt. Mithilfe von NP-vollständigen Mengen kann durch polynomielle Reduktion die NP-Vollständigkeit weiterer Mengen gezeigt werden.

Die Frage, ob $\mathsf{P} = \mathsf{NP}$ ist, ist bis heute unbeantwortet. Die Entscheidungsprobleme in P gelten als effizient lösbar. Entscheidungsprobleme in NP sind lösbar, aber nicht effizient, denn – unter der Annahme, dass $\mathsf{P} \neq \mathsf{NP}$ ist – benötigt die Lösung auf deterministischen Maschinen exponentielle Zeit. Probleme dieser Art, die vielfältig in praktischen Anwendungen auftauchen, gelten als „praktisch nicht lösbar“. Mithilfe von Randomisierung oder Heuristiken können solche Probleme effizient gelöst werden, allerdings möglicherweise auf Kosten von Fehlern und Ungenauigkeiten.

Unter der Annahme, dass $\mathsf{P} \neq \mathsf{NP}$ ist, benötigt man für die Lösung von NP-Problemen Kreativität, die nicht durch Programme geleistet werden kann. Die Überprüfung, ob eine Lösung korrekt ist, kann effizient mit Programmen erfolgen, das Finden der Lösung nicht.

Die Darstellungen in diesem Kapitel sind entsprechenden Darstellungen in [VW16] angelehnt, insbesondere die Beispiele für NP-vollständige Sprachen. Dort werden auch Platz-Komplexitätsklassen, ihr Zusammenhang untereinander sowie ihr Zusammenhang mit den Zeit-Komplexitätsklassen betrachtet. Hier wird auch gezeigt, dass bei den Platzkomplexitätsklassen das Analogon zur P-NP-Frage geklärt ist. Die Klasse PSPACE der Mengen, die deterministisch mit polynomiellem Platzbedarf entschieden werden können, ist gleich der Klasse NPSPACE der Mengen, die nicht deterministisch mit polynomiellem Platzbedarf entschieden werden können. Der Grund ist, dass Speicherplatz im Unterschied zur Laufzeit wiederverwendet werden kann.

Einführungen in die Theorie der (Zeit- und Platz-) Komplexität findet man unter anderem auch in [HS01], [HMU13], [Hr14], [P94], [Si06] und [Weg05].

Eine Einführung in Randomisierung und Heuristiken wird in [VW16] gegeben. Umfassend und ausführlich mit deren Aspekten beschäftigt sich [Weg05].

Kapitel 5

Universelle Berechenbarkeit

Computer wie Großrechner, Personal Computer oder Laptops haben eine wesentliche Eigenschaft, die Turingmaschinen so, wie wir sie bisher kennen, nicht haben: Eine Turingmaschine berechnet ein einziges Programm, während die genannten praktisch verfügbaren Computersysteme alle Programme ausführen können. Ist z. B. auf einem Rechner die Programmiersprache JAVA verfügbar, dann können auf diesem alle (unendlich vielen) JAVA-Programme ausgeführt werden. Dazu sind auf dem Computersystem Programme installiert, die dieses leisten. Diese Systeme stellen **Universalrechner** dar, die, vorausgesetzt es steht die erforderliche Hard- und Softwareausstattung zur Verfügung, *alle* Programme ausführen können. Gehen wir von vollständigen Programmiersprachen aus, kann ein Universalrechner alle berechenbaren Funktionen berechnen.[1]

Eine in dieser Hinsicht sinnvolle Anforderung an Berechenbarkeitskonzepte, wie z. B. Turingmaschinen, ist also, dass in diesen ein *universelles* Programm konstruiert werden kann, das alle anderen Programme ausführen kann. Es sollte also eine universelle Turingmaschine U existieren, die jede andere Turingmaschine T ausführen kann. U erhält als Eingabe das (geeignet codierte) Programm δ_T von T sowie eine Eingabe x für T. Das Programm δ_U von U führt dann das Programm δ_T von T auf x aus, sodass

$$\Phi_U(\langle T \rangle, x) = \Phi_T(x)$$

gilt. Dabei sei $\langle T \rangle$ eine Codierung von T, die U verarbeiten kann. Eine universelle Maschine ist also *programmierbar* und somit ein theoretisches Konzept für die Existenz universeller Rechner, wie wir sie in der Praxis vorfinden. Ein universeller Rechner kann jedes Programm auf dessen Eingabe ausführen und damit jede berechenbare Funktion berechnen.

In diesem Kapitel beschäftigen wir uns ausführlich mit Eigenschaften universeller Berechenbarkeitskonzepte; im folgenden Kapitel werden deren Grenzen aufgezeigt.

[1] Reale Rechner sind – streng formal betrachtet – endliche Maschinen, denn ihnen steht nur ein endlicher Speicher zur Verfügung. Wir gehen aber davon aus, dass wie bei Turingmaschinen ein beliebig großer Speicher zur Verfügung steht. In der Praxis denken wir uns das dadurch gegeben, dass bei Bedarf hinreichend Speicher zur Verfügung gestellt werden kann.

K.-U. Witt und M. E. Müller, *Algorithmische Informationstheorie*,
https://doi.org/10.1007/978-3-662-61694-9_5

5.1 Codierung von Turingmaschinen

Damit eine Turingmaschine T von einer universellen Maschine U ausgeführt werden kann, muss T geeignet als Wort codiert werden, damit es als Eingabe von U dienen kann. In diesem Abschnitt entwickeln wir schrittweise eine Codierung für Turingmaschinen. Dabei gehen wir davon aus, dass eine Turingmaschine

$$T = (\Gamma, S, \delta, s, \#, t_a, t_r)$$

allgemein die Gestalt

$$T = (\{0, 1, A_0, \ldots, A_n\}, \{s_0, \ldots, s_k\}, \delta, s, \#, t_a, t_r) \tag{5.1}$$

hat mit $n \geq 0$, $k \geq 2$, $\# \in \{A_0, \ldots, A_n\}$ und $s, t_a, t_r \in \{s_0, \ldots, s_k\}$.

Des Weiteren kann man überlegen, dass man es – falls notwendig – durch entsprechendes Umnummerieren immer erreichen kann, dass $A_0 = \#$, $s_0 = s$ und $s_1 = t_a$ und $s_2 = t_r$ ist. Durch diese Maßnahmen kann jede Turingmaschine T, die zunächst in der Gestalt (5.1) gegeben ist, kürzer notiert werden in der Form

$$T = (\{A_0, \ldots, A_n\}, \{s_0, \ldots, s_k\}, \delta). \tag{5.2}$$

Startzustand, Haltezustände und Blanksymbol brauchen hierbei nicht mehr gesondert aufgeführt werden. Turingmaschinen dieser Gestalt nennen wir **normiert**.[2] Diese normierte Darstellung erlaubt es, jedem Symbol in der Darstellung (5.2) eine Zahl, etwa seinen Index, zuzuordnen. Ein Problem bei diesem Verfahren ist allerdings, dass die Anzahl von Symbolen und von Zuständen von Turingmaschine zu Turingmaschine variiert und somit auch die Größe der Zahlen, die zur Beschreibung einer Maschine benötigt werden. Dieses Problem kann gelöst werden, indem man die Symbole und Zustände sowie die Kopfbewegungen $\leftarrow$, $\rightarrow$ und $-$ einer Strichcodierung („Bierdeckelnotation“) τ unterzieht: Wir setzen $\tau(0) = 0$ und $\tau(1) = 1$ sowie für $0 \leq j \leq n$ und $0 \leq l \leq k$:

$$\tau(A_j) = A|^j, \;\; \tau(s_l) = s|^l, \;\; \tau(-) = m, \;\; \tau(\leftarrow) = m|, \;\; \tau(\rightarrow) = m|| \tag{5.3}$$

Und für eine Zustandsüberführung $\sigma = (s, a, s', b, m) \in \delta$ mit $m \in \{\leftarrow, \rightarrow, -\}$ setzen wir

$$\tau(\sigma) = \tau(s, a, s', b, m) = (\tau(s)\tau(a)\tau(s')\tau(b)\tau(m)) \tag{5.4}$$

sowie für $\delta = \{\sigma_1, \ldots, \sigma_t\}$

$$\tau(\delta) = \tau(\sigma_1) \ldots \tau(\sigma_t). \tag{5.5}$$

Insgesamt ergibt sich für eine gemäß (5.2) normierte Turingmaschine die Codierung

$$\langle\, T \,\rangle = (\tau(A_0) \ldots \tau(A_n)\tau(s_1) \ldots \tau(s_k)\tau(\delta)). \tag{5.6}$$

[2] Wenn klar ist, dass in der normierten Notation von Turingmaschinen die erste Menge die Arbeitssymbole und die zweite die Zustände darstellen, können die normierten Maschinen noch einfacher geschrieben werden: $T = (n, k, \delta)$.

Jede Turingmaschine T kann also mithilfe von τ als Wort $\langle T \rangle$ über dem achtelementigen Alphabet $\Omega = \{0, 1, A, s, m, |, (,)\}$ codiert werden.

Beispiel 5.1 *Für die normierte Turingmaschine*

$$T = (\{\#\}, \{s_0, t_a, t_r\}, \delta) \text{ oder kürzer } T = (0, 2, \delta)$$

mit

$$\delta = \{(s_0, 0, t_a, 0, -), (s_0, 1, t_r, 1, -), (s_0, \#, t_r, \#, -)\}$$

ergibt sich

$$\langle T \rangle = (Ass|s||(s0s|0m)(s1s||1m)(sAs||Am)).$$

T akzeptiert alle Wörter über $\mathbb{B}$, die mit 0 beginnen; es ist also $L(T) = \{0v \mid v \in \mathbb{B}^*\}$.

Der folgende Satz, das sogenannte **utm-Theorem** (utm steht für *universal turing machine*), postuliert die Existenz einer universellen Turingmaschine.

Satz 5.1 *Es existiert eine universelle Turingmaschine U, sodass für alle Turingmaschinen T und alle Wörter $w \in \mathbb{B}^*$ gilt*

$$\Phi_U(\langle T \rangle, w) = \Phi_T(w).$$

Beweisidee *U kann als Mehrbandmaschine die Maschine T wie folgt simulieren: Die Eingaben für U, die Codierung $\langle T \rangle$ der Maschine T sowie die Eingabe w für T werden auf Band 1 (Programmband) bzw. auf Band 2 (Arbeitsband) gespeichert. U merkt sich mithilfe weiterer Bänder den jeweils aktuellen Zustand und die Position des S-/L-Kopfes von T auf dem Arbeitsband. Damit hält U die jeweilige Konfiguration von T fest. Zu dieser Konfiguration sucht U dann auf dem Programmband eine passende Zustandsüberführung und führt diese (wie T es tun würde) auf dem Arbeitsband aus.*

Da möglicherweise auch die Eingabe w codiert werden muss, schreiben wir anstelle von $\Phi_U(\langle T \rangle, w)$ auch $\Phi_U(\langle T, w \rangle)$ oder – aus schreibtechnischen Gründen nur – $\Phi_U\langle T, w \rangle$. Im Kapitel 7 gehen wir näher darauf ein, wie eine Codierung $\langle T, w \rangle$ erfolgen kann.

5.2 Nummerierung von Turingmaschinen

Wir gehen jetzt noch einen Schritt weiter und ordnen jedem Buchstaben in Ω durch eine Abbildung $\rho : \Omega \to \{0, \ldots, 7\}$ eineindeutig eine Ziffer zu, etwa wie folgt: $\rho(0) = 0$, $\rho(1) = 1$, $\rho(A) = 2$, $\rho(s) = 3$, $\rho(m) = 4$, $\rho(|) = 5$, $\rho(() = 6$, $\rho()) = 7$. Die Nummer $\rho(x)$ eines Wortes $x = x_1 \ldots x_r$, $x_i \in \Omega$, $1 \leq i \leq r$, $r \geq 1$, ergibt sich durch $\rho(x) = \rho(x_1) \ldots \rho(x_r)$.

Mit den folgenden zwei Schritten kann nun jeder normierten Turingmaschine T eine Nummer zugeordnet werden:

1. Bestimme $\langle T \rangle$.
2. Bestimme $\rho(\langle T \rangle)$.

Beispiel 5.2 *Wir bestimmen zu der Codierung $\langle T \rangle$ der Turingmaschine T aus Beispiel 5.1 die Nummer*

$$\rho(\langle T \rangle) = 623353556303504763135514763235524 77.$$

Es ist natürlich nicht jede Zahl eine Nummer einer Turingmaschine, genauso wenig wie jedes beliebige Wort über Ω die Codierung einer Turingmaschine ist. Aber man kann, wenn man eine Zahl gegeben hat, feststellen, ob sie Nummer einer Turingmaschine ist. Ist eine Zahl Nummer einer Turingmaschine, lässt sich daraus die dadurch codierte Turingmaschine eindeutig rekonstruieren.

Beispiel 5.3 *Aus der Nummer*

$$\rho(\langle T \rangle) = 623353556323524763030455763135514 77$$

lässt sich die Darstellung

$$\langle T \rangle = (Ass|s||(sAs|Am)(s0s0m||)(s1s||1m))$$

rekonstruieren, welche die normierte Maschine

$$T = (\{\#\}, \{s_0, t_a, t_r\}, \delta)$$

mit

$$\delta = \{(s_0, \#, t_a, \#, -), (s_0, 0, s_0, 0, \rightarrow), (s_0, 1, t_r, 1, -)\}$$

repräsentiert. Es ist $L(T) = \{0^n \mid n \in \mathbb{N}_0\}$.

Mit **Gödelisierung**[3] oder **Gödelnummerierung** bezeichnet man eine effektive Codierung von Wörtern durch natürliche Zahlen. Im Allgemeinen ist für ein Alphabet Σ eine Gödelnummerierung gegeben durch eine Abbildung (**Gödelabbildung**)

$$g : \Sigma^* \rightarrow \mathbb{N}_0$$

[3] Benannt nach Kurt Gödel (1906–1978), einem österreichischen Mathematiker und Logiker (ab 1947 Staatsbürger der USA), der zu den größten Logikern der Neuzeit gerechnet wird. Er leistete fundamentale Beiträge zur Logik und zur Mengenlehre, insbesondere zur Widerspruchsfreiheit und Vollständigkeit axiomatischer Theorien. Diese Beiträge haben wesentliche Bedeutung für die Informatik.

mit den folgenden Eigenschaften:

(i) g ist injektiv, d. h., für $x, x' \in \Sigma^*$ mit $x \neq x'$ ist $g(x) \neq g(x')$.

(ii) g ist berechenbar.

(iii) Die Funktion $\chi_g : \mathbb{N}_0 \to \mathbb{B}$, definiert durch

$$\chi_g(n) = \begin{cases} 1, & \text{falls ein } x \in \Sigma^* \text{ existiert mit } g(x) = n, \\ 0, & \text{sonst.} \end{cases}$$

ist berechenbar.

(iv) g^{-1} ist berechenbar.

Es sei $\mathcal{T}$ die Menge aller (normierten) Turingmaschinen. Die oben beschriebene Codierung $\rho(\langle\ \rangle) : \mathcal{T} \to \mathbb{N}_0$ stellt eine Gödelisierung der Turingmaschinen dar.

Um allen Nummern eine Turingmaschine zuordnen zu können, benutzen wir die spezielle Turingmaschine $T_\omega = (\mathbb{B} \cup \{\#\}, \{s, t_a, t_r\}, \delta, s, \#, t_a, t_r)$ mit

$$\delta = \{(s, 0, s, 0, \rightarrow), (s, 1, s, 1, \rightarrow), (s, \#, s, \#, \rightarrow)\}.$$

T_ω berechnet die nirgends definierte Funktion ω (siehe Lösung der Übung 3.4).

Übung 5.1 *Bestimmen Sie $\rho(\langle T_\omega \rangle)$!*

Allen Zahlen $i \in \mathbb{N}_0$, denen durch $\rho(\langle\ \rangle)$ keine Maschine zugeordnet ist, ordnen wir die Maschine T_ω zu.

Wir betrachten nun die so vervollständigte (berechenbare) Umkehrung von $\rho(\langle\ \rangle)$: Die Abbildung $\xi : \mathbb{N}_0 \to \mathcal{T}$, definiert durch

$$\xi(i) = \begin{cases} T, & \text{falls } \rho(\langle T \rangle) = i, \\ T_\omega, & \text{sonst,} \end{cases} \tag{5.7}$$

stellt eine totale Abzählung von $\mathcal{T}$ dar. Falls $\xi(i) = T$ ist, nennen wir T auch die i-te Turingmaschine und kennzeichnen T mit dem Index i: $T_i = \xi(i)$. Wir können so die Elemente der Menge $\mathcal{T}$ aller Turingmaschinen entsprechend dieser Indizierung aufzählen:

$$\mathcal{T} = \{T_0, T_1, T_2, \ldots\} \tag{5.8}$$

Die Menge der Turingmaschinen ist also rekursiv-aufzählbar. Es folgt unmittelbar

Satz 5.2 *Die Klasse* RE *ist rekursiv aufzählbar.*

5.3 Nummerierung der berechenbaren Funktionen

Mithilfe der Abbildung ξ erhalten wir nun eine Abzählung $\varphi : \mathbb{N}_0 \to \mathcal{P}$ der Menge $\mathcal{P}$ aller berechenbaren Funktionen, indem wir

$$\varphi(i) = f \text{ genau dann, wenn } \Phi_{\xi(i)} = f$$

festlegen, d. h., wenn $\varphi(i) = \Phi_{\xi(i)}$ ist. φ heißt **Standardnummerierung von $\mathcal{P}$**. Abbildung 5.1 stellt die Schritte zur Nummerierung von $\mathcal{P}$ grafisch dar.

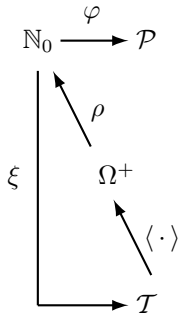

Abbildung 5.1: Standardnummerierung von $\mathcal{P}$

$\varphi(i)$ ist die Funktion $f \in \mathcal{P}$, die von der Turingmaschine $T = \xi(i)$ berechnet wird. Wir sprechen dabei von der i-ten berechenbaren Funktion sowie von der i-ten Turingmaschine. Im Folgenden schreiben wir in der Regel φ_i anstelle von $\varphi(i)$, um zu vermeiden, dass bei Anwendung der Funktion auf ein Argument j eine Reihung von Argumenten auftritt, d. h., wir schreiben $\varphi_i(j)$ anstelle von $\varphi(i)(j)$.

$\mathbb{N}_0$ enthält alle Programme als Nummern codiert: Jede Nummer $i \in \mathbb{N}_0$ stellt quasi den **Objektcode** eines Programms mit **Quellcode** T dar, und jedes Programm wird durch eine Nummer repräsentiert. Durch die Abbildung $\varphi : \mathbb{N}_0 \to \mathcal{P}$ ist die Semantik dieser Programme festgelegt: φ_i ist die Funktion, die vom Programm i berechnet wird.

Beispiel 5.4 *Die normierte T Turingmaschine im Beispiel 5.1 mit der Ω-Codierung*

$$\langle T \rangle = (Ass|s||(s0s|0m)(s1ss||1m)(sAs||Am))$$

berechnet die charakteristische Funktion der Sprache $L = \{0v \mid v \in \mathbb{B}^*\}$; es gilt also $\Phi_T = \chi_L$. Der in Beispiel 5.2 berechnete Objektcode von T ist

$$\rho(\langle T \rangle) = 623353556303504763135514763235524 77.$$

Damit gilt

$$\varphi_{623353556303504763135514763235524 77} = \chi_L.$$

Die Funktion χ_L hat also die Nummer $623353556303504763135514763235524 77$*, und es gilt*

$$\xi(623353556303504763135514763235524 77) = T.$$

Abbildung 5.2 stellt die Nummerierung von χ_L grafisch dar.

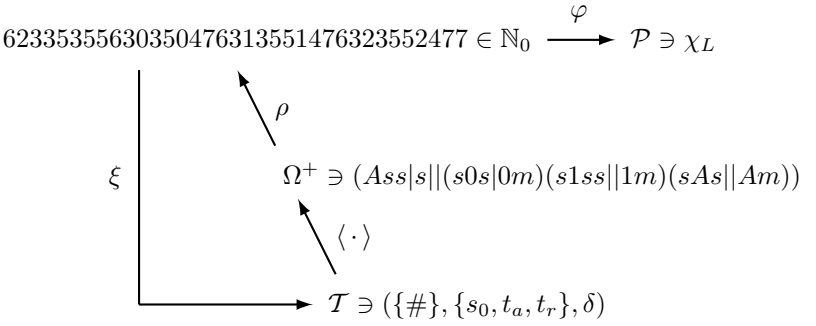

Abbildung 5.2: Standardnummerierung für die Funktion χ_L

Wenn man im Allgemeinen eine Programmiersprache als Tripel

$$(\mathcal{PROG}, \mathcal{F}, \Phi)$$

auffasst, wobei $\mathcal{PROG}$ die Menge aller (syntaktisch korrekten) Programme in einer Programmiersprache ist, $\mathcal{F}$ eine Menge von Funktionen festlegt, und die totale Abbildung $\Phi : \mathcal{PROG} \to \mathcal{F}$ die Semantik von $\mathcal{PROG}$ bestimmt, die jedem Programm $P \in \mathcal{PROG}$ als Bedeutung die Funktion $\Phi_P = \Phi(P) \in \mathcal{F}$ zuordnet, dann haben wir durch die Standardnummerierung aller Turingmaschinen die **abstrakte Programmiersprache**

$$(\mathbb{N}_0, \mathcal{P}, \varphi) \tag{5.9}$$

erhalten. $\mathbb{N}_0$ enthält alle Programme, $\mathcal{P}$ ist die Menge der berechenbaren Funktionen und φ ordnet jedem Programm $i \in \mathbb{N}_0$ seine Bedeutung zu, d. h. die von der Turingmaschine $\xi(i)$ berechnete Funktion $\varphi_i = f$.

5.4 Fundamentale Anforderungen an Programmiersprachen

Von einer Programmiersprache $(\mathcal{PROG}, \mathcal{F}, \Phi)$ kann man die folgenden beiden elementaren Eigenschaften fordern:

(U) Es sollte ein **universelles Programm** $U \in \mathcal{PROG}$ existieren, das alle Programme ausführen kann. U erhält als Eingabe ein Programm $P \in \mathcal{PROG}$ sowie eine Eingabe x für P und berechnet dazu den Wert $\Phi_U\langle P, x\rangle = \Phi_P(x)$.[4] Universelle Programme implementieren quasi universelle Maschinen.

(S) Die Sprache sollte **effektives Programmieren** ermöglichen, d. h., vorhandene Programme sollten „automatisch" zu neuen zusammengesetzt werden können. Es sollte also ein Programm $bind$ existieren, das Programme zu neuen Programmen „zusammenbindet". Ist z. B. $P \in \mathcal{PROG}$ ein zweistelliges Programm, das die Funktion $\Phi_P(x, y)$ berechnet, dann sollte $bind$ den Parameter x durch jedes Programm $P' \in \mathcal{PROG}$ ersetzen und aus P und P' ein Programm $P'' = bind(P, P')$ generieren können, sodass

$$\Phi_P(P', y) = \Phi_{bind(P,P')}(y) = \Phi_{P''}(y)$$

für alle Eingaben y gilt.

Wir nennen Programmiersprachen, die die Anforderungen (U) und (S) erfüllen, **vollständig**. In diesem Abschnitt zeigen wir, dass unsere abstrakte Programmiersprache $(\mathbb{N}_0, \mathcal{P}, \varphi)$ vollständig ist. Anschließend untersuchen wir wesentliche – auch für die praktische Anwendung bedeutsame – Eigenschaften von vollständigen Programmiersprachen.

5.4.1 Das utm-Theorem

Da unsere Standardnummerierung $(\mathbb{N}_0, \mathcal{P}, \varphi)$ auf der Nummerierung von Turingmaschinen basiert und für diese eine universelle Maschine existiert (siehe Satz 5.1), gibt es in $(\mathbb{N}_0, \mathcal{P}, \varphi)$ ein universelles Programm, genauer eine berechenbare **universelle Funktion**, die alle anderen berechenbaren Funktionen berechnen kann.

Satz 5.3 *Für die Nummerierung $(\mathbb{N}_0, \mathcal{P}, \varphi)$ ist die Funktion $u_\varphi : \mathbb{N}_0 \times \mathbb{N}_0 \to \mathbb{N}_0$, definiert durch*

$$u_\varphi(i, x) = \varphi_i(x) \text{ für alle } i, x \in \mathbb{N}_0, \tag{5.10}$$

berechenbar.

[4] $\langle P, x\rangle$ sei eine geeignete Codierung für die gesamte Eingabe für U, bestehend aus dem Programm P und der Eingabe x für P.

Die Funktion u_φ heißt universelle Funktion von $(\mathbb{N}_0, \mathcal{P}, \varphi)$: $u_\varphi(i, x)$ berechnet die i-te berechenbare Funktion für die Eingabe x. Wir werden des Öfteren u_φ als einstellige Funktion betrachten, indem wir die Cantorsche Paarungsfunktion verwenden und $u_\varphi : \mathbb{N}_0 \to \mathbb{N}_0$ durch $u_\varphi \langle i, x \rangle = \varphi_i(x)$ festlegen.

Eine Turingmaschine U mit $\Phi_U \langle \xi(i), x \rangle = u_\varphi \langle i, x \rangle$ heißt universelle Turingmaschine (siehe Satz 5.1). Sie kann als **Interpreter** der Programmiersprache $(\mathbb{N}_0, \mathcal{P}, \varphi)$ aufgefasst werden: U führt das i-te Turingprogramm auf die Eingabe x aus. Damit erfüllt die Standardnummerierung $(\mathbb{N}_0, \mathcal{P}, \varphi)$ also die Anforderung (U) aus der Einleitung dieses Kapitels.

Aus dem utm-Theorem folgt unmittelbar Folgerung 5.1.

Folgerung 5.1 *Es gilt $u_\varphi \in \mathcal{P}$. Somit gibt es ein $k \in \mathbb{N}_0$ mit $\varphi_k = u_\varphi$.*

5.4.2 Das smn-Theorem

Turingprogramme können hintereinander ausgeführt werden, und derart zusammengesetzte Programme sind wieder Turingprogramme. Mithilfe der Codierungen aus Abschnitt 5.1 lassen sich aus den Nummern von Turingprogrammen auch die Nummern der komponierten Programme bestimmen. Aufgrund dieser Überlegung kann der folgende Satz bewiesen werden.

Satz 5.4 *Sei $(\mathbb{N}_0, \mathcal{P}, \varphi)$ die Standardnummerierung sowie $i, j \in \mathbb{N}_0$. Dann gibt es eine total berechenbare Funktion $\mathrm{comp} : \mathbb{N}_0 \times \mathbb{N}_0 \to \mathbb{N}_0$ mit $\varphi_{\mathrm{comp}(i,j)} = \varphi_i \circ \varphi_j$. comp berechnet aus den Nummern der Programme i und j die Nummer der Komposition dieser beiden Programme.*

Beweis *Seien $\xi(i) = A$ und $\xi(j) = B$ die Turingprogramme mit den Nummern i bzw. j. Die Komposition $A \circ B$ der beiden Programme ist wohldefiniert durch (siehe Lösung zur Übung 3.3 und Bemerkung 3.3)*

$$\Phi_{A\circ B}(x) = (\Phi_A \circ \Phi_B)(x) = \begin{cases} \bot, & x \notin \mathit{Def}(\Phi_A), \\ \bot, & \Phi_A(x) \notin \mathit{Def}(\Phi_B), \\ \Phi_A(\Phi_B(x)), & \mathit{sonst}. \end{cases} \tag{5.11}$$

Die Nummer der Komposition $A \circ B$ von A und B ergibt sich durch Berechnung von $\rho(\langle \xi(i) \circ \xi(j) \rangle)$. Wenn wir nun die Funktion $\mathrm{comp} : \mathbb{N}_0 \times \mathbb{N}_0 \to \mathbb{N}_0$ durch

$$\mathrm{comp}(i, j) = \rho(\langle \xi(i) \circ \xi(j) \rangle) \tag{5.12}$$

definieren, gilt für die Standardnummerierung $(\mathbb{N}_0, \mathcal{P}, \varphi)$: Es existiert eine total berechenbare Funktion $\mathrm{comp} : \mathbb{N}_0 \times \mathbb{N}_0 \to \mathbb{N}_0$, sodass

$$\varphi_{\mathrm{comp}(i,j)} = \varphi_i \circ \varphi_j \tag{5.13}$$

für alle $i, j \in \mathbb{N}_0$ ist.

In $\mathcal{T}$ existiert also ein Programm $C = \xi(\mathrm{comp}(i, j))$ mit $\Phi_C = \Phi_{A\circ B}(x)$.

Dieser Satz dient als eine Grundlage für den Beweis des folgenden Satzes, der aussagt, dass die Programmiersprache $(\mathbb{N}_0, \mathcal{P}, \varphi)$ auch die Anforderung (S) aus der Einleitung des Kapitels erfüllt: Eingaben können auch als Programme interpretiert und – automatisch mithilfe von Bindungsprogrammen – zu neuen Programmen komponiert werden. Das folgende **smn-Theorem** hält diese Eigenschaft für $(\mathbb{N}_0, \mathcal{P}, \varphi)$ fest.

Satz 5.5 *In der Standardnummerierung $(\mathbb{N}_0, \mathcal{P}, \varphi)$ existiert eine total berechenbare Funktion $s \in \mathcal{R}$, sodass für alle $i, x, y \in \mathbb{N}_0$*

$$\varphi_i \langle x, y \rangle = \varphi_{s\langle i,x \rangle}(y) \tag{5.14}$$

gilt.

Eine allgemeine Formulierung des smn-Theorems lautet:[5] Es existiert eine total berechenbare Funktion $s \in \mathcal{R}$, sodass

$$\varphi_i \langle x_1, \ldots, x_m, y_1, \ldots, y_n \rangle = \varphi_{s\langle i,x_1,\ldots,x_m \rangle} \langle y_1, \ldots, y_n \rangle \tag{5.15}$$

für alle $i \in \mathbb{N}_0$ sowie für alle $(x_1, \ldots, x_m) \in \mathbb{N}_0^m$ und alle $(y_1, \ldots, y_n) \in \mathbb{N}_0^n$ gilt. Dies kann so interpretiert werden: Es gibt einen „Generator“ s, der aus dem Programm i und den Programmen $x_1, \ldots, x_m$ ein neues Programm $s \langle i, x_1, \ldots, x_m \rangle$ erzeugt.

5.4.3 Anwendungen von utm- und smn-Theorem

Seien $f, g : \mathbb{N}_0 \to \mathbb{N}_0$ berechenbare Funktionen, dann ist auch $h : \mathbb{N}_0 \times \mathbb{N}_0 \to \mathbb{N}_0$, definiert durch $h(x, y) = f(x) + g(y)$, berechenbar. Es gibt also $i, j \in \mathbb{N}_0$ mit $f = \varphi_i$ und $g = \varphi_j$ mit

$$h(x, y) = \varphi_i(x) + \varphi_j(y) = u_\varphi(i, x) + u_\varphi(j, y).$$

Wir verallgemeinern h zu $h' : \mathbb{N}_0^4 \to \mathbb{N}_0$, definiert durch

$$h'(i, j, x, y) = u_\varphi(i, x) + u_\varphi(j, y),$$

d. h., h' ist berechenbar. Es gibt also ein $m \in \mathbb{N}_0$ mit $h' = \varphi_m$, d. h. mit

$$h'(i, j, x, y) = \varphi_m \langle i, j, x, y \rangle.$$

Gemäß (5.15) gibt es eine totale berechenbare Funktion $s \in \mathcal{R}$ mit

$$\varphi_m \langle i, j, x, y \rangle = \varphi_{s\langle m,i,j \rangle} \langle x, y \rangle.$$

Es gibt also ein Programm s, das aus dem Additionsprogramm m und den Programmen für irgendwelche Funktionen f und g das Programm $s \langle m, i, j \rangle$ generiert, das

[5] Aus dieser Formulierung erkennt man, woher die Bezeichnung *smn-Theorem* stammt. Im Satz 5.5 wird die Variante des smn-Theorems für $m = 1$ und $n = 1$ formuliert. Wir wollen diese Variante s-1-1 nennen.

$f(x) + g(y)$ für alle berechenbaren Funktionen f und g berechnet. Die Anforderung (S) (siehe Einleitung von Abschnitt 5.4) ist also für diesen Fall erfüllt: Es gibt ein Programm, nämlich $s\langle m, i, j\rangle$, das die Summe der Ergebnisse der Programme von beliebigen Funktionen f und g berechnet.

Bemerkung 5.1 *Wir können das obige Beispiel für die Anwendung des smn-Theorems verallgemeinern. Seien dazu $f, g : \mathbb{N}_0 \to \mathbb{N}_0$ berechenbare Funktionen, dann ist auch $h : \mathbb{N}_0 \to \mathbb{N}_0$, definiert durch $h(y) = f(g(y))$, berechenbar (siehe Satz 3.1). Es gibt also $i, j \in \mathbb{N}_0$ mit $f = \varphi_i$ und $g = \varphi_j$ mit*

$$h(y) = \varphi_i(\varphi_j(y)) = u_\varphi(i, u_\varphi(j, y)).$$

Wir verallgemeinern h zu $h' : \mathbb{N}_0^3 \to \mathbb{N}_0$, definiert durch

$$h'(i, j, y) = u_\varphi(i, u_\varphi(j, y)),$$

d. h., h' ist berechenbar. Es gibt also ein $k \in \mathbb{N}_0$ mit $h' = \varphi_k$, d. h. mit

$$h'(i, j, y) = \varphi_k\langle i, j, y\rangle.$$

Gemäß (5.15) gibt es eine total berechenbare Funktion $s \in \mathcal{R}$ mit

$$\varphi_k\langle i, j, y\rangle = \varphi_{s\langle k,i,j\rangle}\langle y\rangle.$$

Es gibt also ein Programm s, das aus dem Programm k für die Komposition von Programmen und den Programmen für irgendwelche Funktionen f und g das Programm $s\langle k, i, j\rangle$ generiert, das $f(g(y))$ für alle berechenbaren Funktionen f und g berechnet.

Das smn-Theorem besagt also, dass die Anforderung (S) in dem Sinne erfüllt ist, dass Programme $x_1, \ldots, x_m$ zu neuen Programmen $s\langle i, x_1, \ldots, x_m\rangle$ verknüpft werden können, die die Ergebnisse von $x_1, \ldots, x_m$ angewendet auf $y_1, \ldots, y_n$ gemäß dem Programm i miteinander verknüpfen.

Eine wichtige Folgerung aus dem smn-Theorem ist das **Übersetzungslemma**.

Satz 5.6 *Sei $(\mathbb{N}_0, \mathcal{P}, \varphi)$ die Standardnummerierung und $(\mathbb{N}_0, \mathcal{P}, \psi)$ eine weitere Nummerierung, die das smn-Theorem erfüllt. Dann gibt es eine total berechenbare Funktion $t \in \mathcal{R}$ mit*

$$\varphi_i(x) = \psi_{t(i)}(x) \text{ für alle } x \in \mathbb{N}_0. \tag{5.16}$$

Beweis *Da ψ alle Elemente von $\mathcal{P}$ nummeriert, wird auch u_φ, die universelle Funktion von φ, nummeriert, denn es ist $u_\varphi \in \mathcal{P}$ (siehe Folgerung 5.1). Es gibt also ein*

$k \in \mathbb{N}_0$ mit $\psi_k = u_\varphi$ (auch wenn $\psi_k = u_\varphi$ ist, muss ψ_k nicht zwangsläufig eine universelle Funktion von ψ sein).

Sei $s \in \mathcal{R}$ die gemäß Voraussetzung für ψ existierende s-1-1-Funktion. Wir setzen $t(i) = s\langle k, i\rangle$ (da k als Nummer von u_φ fest gegeben ist, hängt t nur von i ab). Dann ist $t \in \mathcal{R}$, und es gilt mit Satz 5.5

$$\psi_{t(i)}(x) = \psi_{s\langle k,i\rangle}(x) = \psi_k\langle i, x\rangle = u_\varphi(i, x) = \varphi_i(x),$$

was zu zeigen war.

Zwischen der Standardnummerierung $(\mathbb{N}_0, \mathcal{P}, \varphi)$ und der Nummerierung $(\mathbb{N}_0, \mathcal{P}, \psi)$ gibt es immer ein **Übersetzerprogramm** t, welches jedes Programm i von $(\mathbb{N}_0, \mathcal{P}, \varphi)$ in ein äquivalentes Programm $t(i)$ von $(\mathbb{N}_0, \mathcal{P}, \psi)$ transformiert, vorausgesetzt, die Nummerierung $(\mathbb{N}_0, \mathcal{P}, \psi)$ erfüllt das smn-Theorem.

Folgerung 5.2 *Sei $f \in \mathcal{P}$. Dann gibt es eine Funktion $t \in \mathcal{R}$, sodass für alle $i, x \in \mathbb{N}_0$ gilt:*

$$f\langle i, x\rangle = \varphi_{t(i)}(x) \tag{5.17}$$

Beweis *Da $f \in \mathcal{P}$ ist, gibt es ein $k \in \mathbb{N}_0$ mit $f\langle i, x\rangle = \varphi_k\langle i, x\rangle$. Mit dem smn-Theorem folgt, dass es eine Funktion $s \in \mathcal{R}$ gibt mit $\varphi_k\langle i, x\rangle = \varphi_{s\langle k,i\rangle}(x)$. Wir setzen $t(i) = s\langle k, i\rangle$. Dann ist $t \in \mathcal{R}$ und*

$$f\langle i, x\rangle = \varphi_k\langle i, x\rangle = \varphi_{s\langle k,i\rangle}(x) = \varphi_{t(i)}(x),$$

womit die Behauptung gezeigt ist.

Analog zur Verallgemeinerung (5.15) des smn-Theorems kann die folgende Verallgemeinerung der obigen Folgerung gezeigt werden: Sei $f \in \mathcal{P}$, dann gibt es eine total berechenbare Funktion $t \in \mathcal{R}$, sodass

$$f\langle i, x_1, \ldots, x_m, y_1, \ldots, y_n\rangle = \varphi_{t\langle i,x_1,\ldots,x_m\rangle}\langle y_1, \ldots, y_n\rangle \tag{5.18}$$

gilt für alle $i \in \mathbb{N}_0$, $(x_1, \ldots, x_m) \in \mathbb{N}_0^m$ und $(y_1, \ldots, y_n) \in \mathbb{N}_0^n$.

Die folgenden Aussagen sind weitere interessante Folgerungen aus dem utm-Theorem und dem smn-Theorem. Der **Rekursionssatz**[6] besagt, dass jede total berechenbare Programmtransformation in $(\mathbb{N}_0, \mathcal{P}, \varphi)$ mindestens ein Programm in ein äquivalentes transformiert, und der Selbstreproduktionssatz besagt, dass es in $(\mathbb{N}_0, \mathcal{P}, \varphi)$ mindestens ein Programm gibt, das für jede Eingabe seinen eigenen Quelltext ausgibt.

[6] Der Rekursionssatz ist auch als *Fixpunktsatz von Kleene* bekannt.

Satz 5.7 *Zu jeder total berechenbaren Funktion $f \in \mathcal{R}$ existiert eine Zahl $n \in \mathbb{N}_0$ mit $\varphi_{f(n)} = \varphi_n$.*

Beweis *Die Funktion $d : \mathbb{N}_0 \times \mathbb{N}_0 \to \mathbb{N}_0$ sei definiert durch*

$$d(x, y) = u_\varphi(u_\varphi(x, x), y).$$

Mithilfe des utm-Theorems gilt

$$d(x, y) = u_\varphi(u_\varphi(x, x), y) = u_\varphi(\varphi_x(x), y) = \varphi_{\varphi_x(x)}(y). \tag{5.19}$$

Gemäß utm-Theorem ist d also berechenbar. Wegen Folgerung 5.2 gibt es zu d ein $t \in \mathcal{R}$ mit

$$d(x, y) = \varphi_{t(x)}(y). \tag{5.20}$$

Die Funktionen f und t sind total berechenbar, also ist auch $f \circ t$ total berechenbar. Somit gibt es ein $m \in \mathbb{N}_0$ mit

$$\varphi_m = f \circ t \tag{5.21}$$

und φ_m ist total. Des Weiteren setzen wir (für das feste m)

$$n = t(m). \tag{5.22}$$

Es gilt mit (5.22), (5.20), (5.19) und (5.21)

$$\varphi_n(y) = \varphi_{t(m)}(y) = d(m, y) = \varphi_{\varphi_m(m)}(y) = \varphi_{f(t(m))}(y) = \varphi_{f(n)}(y),$$

womit die Behauptung gezeigt ist.

Der Rekursionssatz gilt für alle total berechenbaren Funktionen, z. B. für die Funktionen $f(x) = x + 1$ und $g(x) = x^2$. Der Rekursionssatz besagt, dass ein $m \in \mathbb{N}_0$ existiert mit $\varphi_{m+1} = \varphi_m$ bzw. ein $n \in \mathbb{N}_0$ existiert mit $\varphi_{n^2} = \varphi_n$.

Den Rekursionssatz kann man so interpretieren, dass jede total berechenbare Programmtransformation f mindestens ein Programm n in sich selbst transformiert (n also „reproduziert“). Das gilt z. B. auch für den Fall, dass f ein „Computervirus“ ist, der alle Programme verändert. Der Rekursionssatz besagt, dass der Virus mindestens ein Programm unverändert lassen würde.

Eine Folgerung aus dem Rekursionssatz ist der **Selbstreproduktionssatz**.

Satz 5.8 *Es gibt eine Zahl $n \in \mathbb{N}_0$ mit $\varphi_n(x) = n$ für alle $x \in \mathbb{N}_0$.*

Beweis *Wir definieren $f : \mathbb{N}_0 \times \mathbb{N}_0 \to \mathbb{N}_0$ durch $f(i, x) = i$. Die Funktion f ist offensichtlich berechenbar. Gemäß Folgerung 5.2 gibt es somit eine total berechenbare*

Funktion $t \in \mathcal{R}$ mit $\varphi_{t(i)}(x) = f(i,x) = i$. Nach dem Rekursionssatz gibt es zu t ein $n \in \mathbb{N}_0$ mit $\varphi_n = \varphi_{t(n)}$. Insgesamt folgt $\varphi_n(x) = \varphi_{t(n)}(x) = f(n,x) = n$ für alle $x \in \mathbb{N}_0$, womit die Behauptung gezeigt ist.

Der Selbstreproduktionssatz besagt, dass es in einer Programmiersprache, die den Anforderungen (U) und (S) aus der Einleitung genügt, mindestens ein Programm n gibt, das unabhängig von den Eingaben x sich selbst („seinen eigenen Code“) ausgibt.

5.4.4 Der Äquivalenzsatz von Rogers

Im vorigen Abschnitt haben wir gesehen, dass die Tatsache, dass die Programmiersprache $(\mathbb{N}_0, \mathcal{P}, \varphi)$ das utm- und das smn-Theorem erfüllt, dazu führt, dass diese Sprache sowohl theoretisch als auch praktisch nützliche Eigenschaften besitzt. Das utm-Theorem und das smn-Theorem können als Präzisierungen der in der Einleitung des Kapitels geforderten elementaren Eigenschaften (U) bzw. (S) für Programmiersprachen betrachtet werden. Dass diese Eigenschaften tatsächlich fundamental sind, besagt der folgende Satz: Jede andere Nummerierung $(\mathbb{N}_0, \mathcal{P}, \psi)$ ist äquivalent zur Standardnummerierung $(\mathbb{N}_0, \mathcal{P}, \varphi)$ genau dann, wenn sie ebenfalls das utm-Theorem und das smn-Theorem erfüllt.

Satz 5.9 *Sei $(\mathbb{N}_0, \mathcal{P}, \varphi)$ die Standardnummerierung und $\psi : \mathbb{N}_0 \to \mathcal{P}$ eine weitere Standardnummerierung von $\mathcal{P}$, dann sind die beiden folgenden Aussagen äquivalent:*

(1) Es gibt total berechenbare Funktionen $t_{\varphi\to\psi}, t_{\psi\to\varphi} \in \mathcal{R}$ („Übersetzer“ für φ bzw. ψ) mit $\varphi_i = \psi_{t_{\varphi\to\psi}(i)}$ und $\psi_j = \varphi_{t_{\psi\to\varphi}(j)}$ (d. h., „φ und ψ sind äquivalent“).

(2) Die Standardnummerierung ψ erfüllt das utm-Theorem und das smn-Theorem:

(U) Die universelle Funktion $u_\psi : \mathbb{N}_0 \times \mathbb{N}_0 \to \mathbb{N}_0$ mit $u_\psi(i,x) = \psi_i(x)$ für alle $i, x \in \mathbb{N}_0$ ist berechenbar.

(S) Es gibt eine total berechenbare Funktion t mit $\psi_i \langle x, y \rangle = \psi_{t\langle i,x \rangle}(y)$ für alle $i, x, y \in \mathbb{N}_0$.

Beweis *„(1) ⇒ (2)“: Es gilt*

$$u_\psi(i,x) = \psi_i(x) = \varphi_{t_{\psi\to\varphi}(i)}(x) = u_\varphi(t_{\psi\to\varphi}(i), x). \tag{5.23}$$

Nach dem utm-Theorem für φ ist u_φ berechenbar, und nach Voraussetzung ist $t_{\psi\to\varphi}$ berechenbar. Wegen der Gleichung (5.23) ist dann auch u_ψ berechenbar. Damit haben wir (2 U) gezeigt.

Da für φ das smn-Theorem gilt, gibt es gemäß Folgerung 5.2 für die berechenbare Funktion $u_\psi(i, \langle x, y \rangle)$ eine total berechenbare Funktion $s \in \mathcal{R}$ mit

$$u_\psi(i, \langle x, y \rangle) = \varphi_{s\langle i,x \rangle}(y).$$

Es ist $\psi_i \langle x, y \rangle = u_\psi(i, \langle x, y \rangle)$ und nach Voraussetzung gilt $\varphi_z = \psi_{t_{\varphi\to\psi}(z)}$. Somit gilt also insgesamt

$$\psi_i(x, y) = u_\psi(i, \langle x, y \rangle) = \varphi_{s\langle i,x \rangle}(y) = \psi_{t_{\varphi\to\psi}(s\langle i,x \rangle)}(y).$$

Wir setzen $t \langle i, x \rangle = t_{\varphi\to\psi}(s \langle i, x \rangle)$, also $t = t_{\varphi\to\psi} \circ s$. Somit gibt es also eine total berechenbare Funktion $t \in \mathcal{R}$, für die

$$\psi_i \langle x, y \rangle = \psi_{t\langle i,x \rangle}(y)$$

für alle $i, x, y \in \mathbb{N}_0$ gilt.

Damit ist (2 S) gezeigt.

„(2) $\Rightarrow$ (1)“: Die Funktion u_ψ ist berechenbar, da für ψ das utm-Theorem erfüllt ist. Nach dem smn-Theorem für φ gibt es zu u_ψ eine total berechenbare Funktion $t_{\psi\to\varphi} \in \mathcal{R}$ mit $u_\psi(j, y) = \varphi_{t_{\psi\to\varphi}(j)}(y)$. Mit der Voraussetzung $u_\psi(j, y) = \psi_j(y)$ folgt $\psi_j(y) = \varphi_{t_{\psi\to\varphi}(j)}(y)$ und damit $\psi_j = \varphi_{t_{\psi\to\varphi}(j)}$.

Mit analoger Argumentation leitet man $\varphi_i = \psi_{t_{\varphi\to\psi}(i)}$ für ein $t_{\varphi\to\psi} \in \mathcal{R}$ her.

Der Satz von Rogers besagt, dass es bis auf gegenseitige Übersetzbarkeit nur eine einzige Programmiersprache, beispielsweise $(\mathbb{N}_0, \mathcal{P}, \varphi)$, gibt, die das utm-Theorem und das smn-Theorem erfüllt. $(\mathbb{N}_0, \mathcal{P}, \varphi)$ kann somit als „Referenz-Programmiersprache“ gelten. Das bedeutet insbesondere, dass alle Eigenschaften, die auf die Sprache $(\mathbb{N}_0, \mathcal{P}, \varphi)$ zutreffen, für alle vollständigen Programmiersprachen gelten.

Aus dem Satz von Rogers folgt unmittelbar folgende für die Praxis wichtige Aussage.

Folgerung 5.3 *Zu zwei vollständigen Programmiersprachen $\mathcal{M}_1$ und $\mathcal{M}_2$ existieren Programme $T_{1\to2} \in \mathcal{M}_2$ und $T_{2\to1} \in \mathcal{M}_1$ mit*

$$\Phi_A^{(\mathcal{M}_1)} = \Phi_{T_{1\to2}(A)}^{(\mathcal{M}_2)} \text{ für alle } A \in \mathcal{M}_1 \tag{5.24}$$

bzw. mit

$$\Phi_B^{(\mathcal{M}_2)} = \Phi_{T_{2\to1}(B)}^{(\mathcal{M}_1)} \text{ für alle } B \in \mathcal{M}_2 \tag{5.25}$$

Es gibt also für je zwei vollständige Programmiersprachen **Übersetzungsprogramme**, welche die Programme der einen Sprache in äquivalente Programme der anderen Sprache übersetzen.

5.5 Zusammenfassung und bibliografische Hinweise

Die Nummerierung von Turingmaschinen führt zur Standardnummerierung $(\mathbb{N}_0, \mathcal{P}, \varphi)$ der partiell berechenbaren Funktionen, die als abstrakte Programmiersprache angesehen werden kann, anhand der grundsätzliche Eigenschaften von Programmiersprachen analysiert werden können.

Fundamentale Anforderungen an Programmiersprachen sind die Existenz von universellen Programmen sowie die Möglichkeit zum effektiven Programmieren. Universelle Programme können alle Programme ausführen und damit alle berechenbaren Funktionen berechnen. Effektives Programmieren bedeutet, dass Programme selbst Eingabe von Programmen sein können und es Programme gibt, die daraus neue Programme generieren können. Das utm-Theorem und das smn-Theorem drücken aus, dass die Standardnummerierung diese Anforderungen erfüllt.

Programmiersprachen, die das utm-Theorem und das smn-Theorem erfüllen, werden vollständig genannt. Der Äquvalenzsatz von Rogers besagt, dass es zwischen vollständigen Programmiersprachen totale Übersetzungsprogramme gibt, die jedes Programm der einen in ein äquivalentes Programm der anderen Sprache transformieren.

Zu den vollständigen Programmiersprachen gehören die in der Praxis weit verbreiteten Sprachen JAVA und C++. Zur Ausführung von Programmen in diesen Sprachen müssen allerdings hinreichend große Speicher – wie bei Turingmaschinen das unendliche Arbeitsband – zur Verfügung stehen.

Unter dem Stichwort „Accidentally Turing Complete“ findet man überraschende und kuriose vollständige Programmiermöglichkeiten, wie z. B. Spiele und das Page fault handling des X86-Prozessors.

Das Kapitel folgt Darstellungen in [VW16]. Die Themen werden mehr oder weniger ausführlich in [AB02], [BL74], [HMU13], [Hr14], [Koz97], [LP98], [Ro67], [Schö09], [Si06] und [Wei87] behandelt.

Es gibt viele interessante Fragestellungen, die die Mächtigkeit von Turingmaschinen betreffen, insbesondere dahin gehend, welche Art von Funktionalität noch maximal mit einer möglichst kleinen Menge von Zuständen oder Bandsymbolen erreicht werden kann.

Dazu gehört das sogenannte Fleißige-Biber-Problem, das von Tibor Radó in [Ra62] erstmalig vorgestellt wurde. Radó betrachtet zwei Funktionen: (1) $\Sigma(n) =$ maximale Anzahl der Einsen, die eine anhaltende Turingmaschine mit n Zuständen auf ein anfänglich leeres Band schreiben kann, sowie (2) $S(n) =$ maximale Anzahl der Kopfbewegungen, die eine anhaltende Turingmaschine mit n Zuständen und zweielementigem Arbeitsalphabet durchführen kann. Es kann gezeigt werden, dass die Mengen $\{(n, b) \mid \Sigma(n) \leq b\}$ und $\{(n, c) \mid S(n) \leq c\}$ nicht rekursiv-aufzählbar sind, d. h., $\Sigma(n)$ und $S(n)$ sind nicht berechenbar; beide Funktionen wachsen schneller als jede berechenbare Funktion. Für $1 \leq n \leq 4$ sieht das Wachstum von $\Sigma(n)$ noch harmlos aus: $\Sigma(1) = 1$, $\Sigma(2) = 4$, $\Sigma(3) = 6$, $\Sigma(4) = 13$. Für $n = 5$ gilt aber bereits $\Sigma(5) \geq 2^{12}$, und für $n = 6$ gilt $\Sigma(6) > 2^{54\,000}$.

Eine weitere interessante Frage ist: Wie viele Zustände und wie viele Arbeitssymbole muss eine universelle Turingmaschine mindestens haben? Unter

en.wikipedia.org/wiki/Wolfram%27s_2-state_3-symbol_Turing_machine

findet man die Beschreibung einer universellen Maschine von Stephen Wolfram mit zwei Zuständen und fünf Symbolen. Wolfram vermutete, dass eine universelle Turingmaschine mit zwei Zuständen und drei Symbolen existiert, und setzte einen Preis in Höhe von $ 25.000 für denjenigen aus, der als Erster eine solche Maschine findet. Alex Smith gibt in

www.wolframscience.com/prizes/tm23/TM23Proof.pdf

eine universelle Maschine mit zwei Zuständen und drei Symbolen an, wofür er dann auch den Preis bekam.

Kapitel 6

Unentscheidbare Mengen

In (5.8) wird festgestellt, dass die Menge $\mathcal{T}$ der Turingmaschinen nummeriert werden kann. Daraus folgt, dass diese Menge abzählbar ist. Wir zeigen im Folgenden, dass die Menge

$$X = \{\chi : \mathbb{N}_0 \to \mathbb{B} \mid \chi \text{ ist total}\} \tag{6.1}$$

der charakteristischen Funktionen überabzählbar ist. Daraus folgt, dass es charakteristische Funktionen gibt, die nicht berechenbar sind. Hieraus folgt, dass prinzipiell unentscheidbare Mengen existieren. In den folgenden Abschnitten werden wir dann konkrete unentscheidbare Mengen kennenlernen.

Satz 6.1 *Die in (6.1) definierte Menge X ist überabzählbar.*

Beweis *Wir nehmen an, dass X abzählbar ist. Das bedeutet, dass die Elemente von X eindeutig nummeriert werden können: $X = \{\chi_0, \chi_1, \chi_2, \ldots\}$. Wir denken uns für die beiden abzählbaren Mengen $\mathbb{N}_0$ und X die folgende unendliche Matrix:*

	x_0	x_1	x_2	$\ldots$	x	$\ldots$
χ_0	$\chi_0(x_0)$	$\chi_0(x_1)$	$\chi_0(x_2)$	$\ldots$	$\chi_0(x)$	$\ldots$
χ_1	$\chi_1(x_0)$	$\chi_1(x_1)$	$\chi_1(x_2)$	$\ldots$	$\chi_1(x)$	$\ldots$
χ_2	$\chi_2(x_0)$	$\chi_2(x_1)$	$\chi_2(x_2)$	$\ldots$	$\chi_2(x)$	$\ldots$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$

Diese Matrix ist eine 0-1*-Matrix, die keine Lücken aufweist, weil alle χ_i total definiert sind.*

Mithilfe der Diagonalen dieser Matrix definieren wir die Funktion $\chi : \mathbb{N}_0 \to \mathbb{B}$ durch

$$\chi(x_i) = 1 - \chi_i(x_i). \tag{6.2}$$

K.-U. Witt und M. E. Müller, *Algorithmische Informationstheorie*,
https://doi.org/10.1007/978-3-662-61694-9_6

Die Funktion χ unterscheidet sich von allen χ_i mindestens beim Argument x_i:

$$\chi(x_i) = 1 - \chi_i(x_i) = \begin{cases} 1, & \chi_i(x_i) = 0 \\ 0, & \chi_i(x_i) = 1 \end{cases}$$

Die Funktion χ ist eine wohl definierte, totale Funktion von $\mathbb{N}_0$ nach $\mathbb{B}$. Sie muss also ein Element von X sein und deshalb eine Nummer besitzen; sei k diese Nummer:

$$\chi = \chi_k \tag{6.3}$$

Dann folgt für das Argument x_k:

$$\begin{aligned} \chi_k(x_k) &= \chi(x_k) && \text{wegen (6.3)} \\ &= 1 - \chi_k(x_k) && \text{wegen (6.2)} \end{aligned}$$

Es folgt $\chi_k(x_k) = 1 - \chi_k(x_k)$, was offensichtlich ein Widerspruch ist. Unsere Annahme, dass X abzählbar ist, ist also falsch, womit die Behauptung gezeigt ist.

Die im obigen Beweis verwendete Beweismethode heißt **Cantors zweites Diagonalargument**.[1] Dabei wird auf der Basis einer Annahme eine Matrix aufgestellt und mithilfe der Diagonale der Matrix eine Funktion, eine Menge oder eine Aussage definiert, die zu einem Widerspruch führt, woraus dann folgt, dass die ursprüngliche Annahme falsch sein muss.

Der Satz 6.1 zeigt, dass es prinzipiell nicht entscheidbare Mengen $A \subseteq \mathbb{N}_0$ geben muss. In den folgenden Abschnitten geben wir sowohl konkrete Beispiele von Mengen an, die nicht entscheidbar sind, als auch Beispiele von Mengen, die nicht semientscheidbar, also nicht rekusiv-aufzählbar sind. Grundlage der Betrachtungen dafür ist nach wie vor unsere Standardprogrammiersprache $(\mathbb{N}_0, \mathcal{P}, \varphi)$. Damit gelten die folgenden Aussagen auch für alle anderen vollständigen Programmiersprachen.

6.1 Das Halteproblem

Wir betrachten zunächst ein **spezielles Halteproblem**, auch **Selbstanwendbarkeitsproblem** genannt, und formulieren es mithilfe der folgenden Menge

$$K = \{i \in \mathbb{N}_0 \mid i \in \mathit{Def}(\varphi_i)\}.$$

Die Menge K enthält alle Nummern von Turingmaschinen, die, angewendet auf sich selbst, anhalten.

[1] Cantors erstes Diagonalargument haben wir in Kapitel 2.8 angewendet.

Satz 6.2 **a)** *K ist rekursiv-aufzählbar.*

b) *K ist nicht entscheidbar.*

Beweis **a)** *Wir definieren $f : \mathbb{N}_0 \to \mathbb{N}_0$ durch $f(i) = u_\varphi(i,i)$. Dann gilt: Die Funktion f ist berechenbar sowie*

$$\begin{aligned} i \in K \ &\text{genau dann, wenn } i \in Def(\varphi_i), \\ &\text{genau dann, wenn } (i,i) \in Def(u_\varphi), \\ &\text{genau dann, wenn } i \in Def(f). \end{aligned}$$

Es ist also $K = Def(f)$. Mit Satz 3.4 b) folgt, dass K rekursiv aufzählbar ist.

b) *Wir nehmen an, K sei entscheidbar. Dann ist χ_K berechenbar. Wir definieren die Funktion $g : \mathbb{N}_0 \to \mathbb{N}_0$ durch*

$$g(x) = \begin{cases} u_\varphi(x,x) + 1, & \text{falls } \chi_K(x) = 1, \\ 0, & \text{falls } \chi_K(x) = 0. \end{cases} \tag{6.4}$$

Die Funktion g ist total berechenbar, denn, wenn $\chi_K(x) = 1$ ist, ist $x \in K$ und damit $x \in Def(\varphi_x)$, und demzufolge ist $u_\varphi(x,x) + 1 = \varphi_x(x) + 1$ definiert.

Da g berechenbar ist, gibt es ein p mit $g = \varphi_p$, d. h., es ist

$$g(x) = \varphi_p(x) \text{ für alle } x \in \mathbb{N}_0. \tag{6.5}$$

Wir berechnen nun $g(p)$. Als Ergebnis gibt es zwei Möglichkeiten: (1) $g(p) = 0$ oder (2) $g(p) = u_\varphi(p,p) + 1$.

Zu (1): Es gilt

$$\begin{aligned} g(p) = 0 \ &\text{genau dann, wenn } \chi_K(p) = 0, && \text{wegen (6.4)} \\ &\text{genau dann, wenn } p \notin K, \\ &\text{genau dann, wenn } p \notin Def(\varphi_p), \\ &\text{genau dann, wenn } p \notin Def(g), && \text{wegen (6.5)} \\ &\text{genau dann, wenn } g(p) = \perp. \end{aligned}$$

Damit haben wir die widersprüchliche Aussage

$$g(p) = 0 \text{ genau dann, wenn } g(p) = \perp$$

hergeleitet, d. h., der Fall $g(p) = 0$ kann nicht auftreten.

Zu (2): Mit (6.4) und (6.5) erhalten wir

$$g(p) = u_\varphi(p,p) + 1 = \varphi_p(p) + 1 = g(p) + 1$$

und damit den Widerspruch $g(p) = g(p) + 1$.

Die Anwendung von g auf p, d. h. von g auf sich selbst, führt also in jedem Fall zum Widerspruch. Unsere Annahme, dass K entscheidbar und damit χ_K berechenbar ist, muss also falsch sein, womit die Behauptung gezeigt ist.

Möglicherweise erscheint das Selbstanwendbarkeitsproblem auf den ersten Blick als ein künstliches Problem. Im Hinblick auf Programme, die man auf sich selbst anwenden kann, denke man z. B. an ein Programm, das die Anzahl der Buchstaben in einer Zeichenkette zählt. Es kann durchaus sinnvoll sein, das Programm auf sich selbst anzuwenden um festzustellen, wie lang es ist.

Die Menge

$$H = \{(i,j) \in \mathbb{N}_0 \times \mathbb{N}_0 \mid j \in Def(\varphi_i)\} \tag{6.6}$$

beschreibt das (**allgemeine**) **Halteproblem**. Sie enthält alle Paare von Programmen i und Eingaben j, sodass i angewendet auf j ein Ergebnis liefert, d. h. insbesondere, dass i bei Eingabe j anhält. K ist eine Teilmenge von H,[2] und damit ist intuitiv klar, dass auch H nicht entscheidbar sein kann.

Satz 6.3 **a)** *H ist rekursiv-aufzählbar.*

b) *H ist nicht entscheidbar.*

Übung 6.1 *Beweisen Sie Satz 6.3!*

Die Unentscheidbarkeit des Halteproblems bedeutet, dass es keinen allgemeinen Terminierungsbeweiser geben kann. Ein solcher Beweiser könnte überprüfen, ob ein Programm angewendet auf eine Eingabe anhält. Wenn man also verhindern möchte, dass Ausführungen von Schleifen nicht terminieren, muss man dies schon bei der Programmierung durch Anwendung geeigneter Methoden verhindern, weil im Nachhinein das Terminierungsverhalten algorithmisch im Allgemeinen nicht mehr überprüft werden kann.

Das Halteproblem ist mindestens so „schwierig“ wie jedes andere rekursiv-aufzählbare Problem, denn alle rekursiv-aufzählbaren Probleme lassen sich auf das Halteproblem reduzieren. Andererseits lassen sich entscheidbare Probleme auf alle anderen (nicht trivialen) Probleme reduzieren; in diesem Sinne sind entscheidbare Probleme „besonders einfach“. Dies werden wir im Abschnitt 8.4 noch aus einem anderen Blickwinkel betrachtet bestätigt finden.

Satz 6.4 **a)** *Für jede rekursiv-aufzählbare Menge $A \subseteq \mathbb{N}_0$ gilt $A \leq H$.*

b) *Für jede entscheidbare Menge $A \subseteq \mathbb{N}_0$ und jede Menge $B \subseteq \mathbb{N}_0$ mit $B \neq \emptyset$ und $B \neq \mathbb{N}_0$ gilt $A \leq B$.*

[2] Damit das mathematisch präzise ausgedrückt werden kann, müsste K eigentlich wie folgt notiert werden: $K = \{(i,i) \in \mathbb{N}_0 \times \mathbb{N}_0 \mid i \in Def(i)\}$.

Beweis **a)** *A ist rekursiv-aufzählbar, also semi-entscheidbar, d. h., χ'_A ist berechenbar. Es gibt also ein $i \in \mathbb{N}_0$ mit $\chi'_A = \varphi_i$. Mithilfe dieses fest gegebenen i definieren wir $f : \mathbb{N}_0 \to \mathbb{N}_0 \times \mathbb{N}_0$ durch $f(j) = (i, j)$. Die Funktion f ist total berechenbar, und es gilt*

$$\begin{aligned} j \in A \ &\textit{genau dann, wenn } \chi'_A(j) = 1, \\ &\textit{genau dann, wenn } \varphi_i(j) = 1, \\ &\textit{genau dann, wenn } j \in \mathit{Def}(\varphi_i), \\ &\textit{genau dann, wenn } (i, j) \in H, \\ &\textit{genau dann, wenn } f(j) \in H. \end{aligned}$$

Damit ist die Behauptung $A \leq_f H$ gezeigt.

b) *Da $B \neq \emptyset$, gibt es mindestens ein $i \in B$, und da $B \neq \mathbb{N}_0$, gibt es mindestens ein $j \notin B$. Mit diesen beiden Elementen definieren wir die Funktion $f : \mathbb{N}_0 \to \mathbb{N}_0$ durch*

$$f(x) = \begin{cases} i, & \chi_A(x) = 1, \\ j, & \chi_A(x) = 0. \end{cases}$$

Da A entscheidbar ist, sind χ_A und damit auch f total berechenbar. Außerdem gilt $x \in A$ genau dann, wenn $f(x) \in B$. Damit gilt $A \leq_f B$, was zu zeigen war.

Aus Satz 3.2 b) und den Sätzen 6.2 und 6.3 folgt unmittelbar ein Satz über die Komplemente von K und H.

Satz 6.5 **a)** *Die Komplemente von K und H*

$$\overline{K} = \{i \in \mathbb{N}_0 \mid i \notin \mathit{Def}(\varphi_i)\} \ \textit{ bzw. } \overline{H} = \{(i, j) \in \mathbb{N}_0 \times \mathbb{N}_0 \mid j \notin \mathit{Def}(\varphi_i)\}$$

sind nicht rekursiv-aufzählbar.

b) RE *ist nicht abgeschlossen gegenüber Komplementbildung.*

$\overline{K}$ und $\overline{H}$ sind also Beispiele für Mengen, die nicht nur nicht entscheidbar, sondern sogar nicht semi-entscheidbar sind. Damit sind zudem $\chi'_{\overline{K}}$ und $\chi'_{\overline{H}}$ Beispiele für nicht berechenbare Funktionen.

Um das Terminierungsproblem grundsätzlich zu vermeiden, könnte man auf die Idee kommen, universelle Programmiersprachen von vornherein so zu gestalten, dass alle Programme nur total berechenbare Funktionen berechnen. Sei $(\mathbb{N}_0, \mathcal{R}, \psi)$ die Nummerierung einer solchen Sprache. Der folgende Satz besagt, dass es in dieser Nummerierung keine universelle Funktion geben kann.

Satz 6.6 *Sei $u_\psi : \mathbb{N}_0 \times \mathbb{N}_0 \to \mathbb{N}_0$, definiert durch $u_\psi(i,j) = \psi_i(j)$, eine universelle Funktion für $(\mathbb{N}_0, \mathcal{R}, \psi)$, dann ist $u_\psi \notin \mathcal{R}$.*

Beweis *Wir nehmen an, dass $u_\psi \in \mathcal{R}$ ist, und definieren die Funktion $f : \mathbb{N}_0 \to \mathbb{N}_0$ durch $f(x) = u_\psi(x,x) + 1$. Dann folgt, dass $f \in \mathcal{R}$ ist. Damit gibt es ein $k \in \mathbb{N}_0$ mit $\psi_k = f$, d. h., für alle $x \in \mathbb{N}_0$ gilt $\psi_k(x) = f(x) = u_\psi(x,x) + 1$. Für $x = k$ folgt daraus*

$$u_\psi(k,k) = \psi_k(k) = f(k) = u_\psi(k,k) + 1,$$

was offensichtlich einen Widerspruch darstellt, womit unsere Annahme $u_\psi \in \mathcal{R}$ widerlegt ist.

In einem weiteren Schritt, dass Terminierungsproblem in den Griff zu bekommen, könnte man versuchen, die Menge der Programme in $(\mathbb{N}_0, \mathcal{P}, \varphi)$ auf die zu beschränken, die totale Funktionen berechnen. Dass auch dieser Versuch zum Scheitern verurteilt ist, besagt der folgende Satz.

Satz 6.7 *Sei $(\mathbb{N}_0, \mathcal{P}, \varphi)$ die Standardprogrammierung und $A \subset \mathbb{N}_0$ so, dass $\varphi(A) = \mathcal{R}$ gilt. Dann ist A nicht rekursiv-aufzählbar und damit nicht entscheidbar.*

Beweis *Wir nehmen an, dass A rekursiv-aufzählbar ist. Da $\mathcal{R} \neq \emptyset$ gilt, gilt auch $A \neq \emptyset$. Dann gibt es eine total berechenbare Funktion $f : \mathbb{N}_0 \to \mathbb{N}_0$ mit $A = W(f)$. Wir definieren die Funktion $\psi : \mathbb{N}_0 \to \mathcal{R}$ durch $\psi = \varphi \circ f$. Dann ist ψ eine totale surjektive Funktion mit berechenbarer universeller Funktion $u_\psi(i,x) = \psi_i(x) = \varphi_{f(i)}(x) = u_\varphi(f(i),x)$. Dieses widerspricht aber Satz 6.6.*

Eine Menge $A \subset \mathbb{N}_0$ von Programmen, die nicht entscheidbar ist, kann keine Programmiersprache sein, da für ein $i \in \mathbb{N}_0$ nicht entschieden werden könnte, ob es ein Programm aus A ist oder nicht.

Ein weiterer Ansatz, dass Terminierungsproblem zu lösen, wäre, jede berechenbare Funktion $f \in \mathcal{P}$ zu einer total berechenbaren Funktion $f' \in \mathcal{R}$ mit $f(x) = f'(x)$ für alle $x \in Def(f)$ fortzusetzen. Der folgende Satz besagt, dass auch dieses Ansinnen zum Scheitern verurteilt ist.

Satz 6.8 *Sei $(\mathbb{N}_0, \mathcal{P}, \varphi)$ die Standardnummerierung. Dann besitzt die Funktion $f : \mathbb{N}_0 \to \mathbb{N}_0$, definiert durch $f\langle i,j \rangle = u_\varphi(i,j)$, keine Fortsetzung.*

Beweis *Sei $f' \in \mathcal{R}$ eine Fortsetzung von f. Damit definieren wir eine Nummerierung $\psi : \mathbb{N}_0 \to \mathcal{R}$ durch $\psi_i(j) = f'\langle i,j \rangle$. Dann wäre die Funktion $u'(i,j) = f'(i,j)$ eine universelle Funktion der Nummerierung $(\mathbb{N}_0, \mathcal{R}, \psi)$, was aber Satz 6.6 widerspricht.*

6.2 Der Satz von Rice

Das Terminierungsproblem ist innerhalb vollständiger Programmiersprachen allgemein algorithmisch nicht lösbar. Es stellt sich die Frage, ob andere Eigenschaften von Programmen algorithmisch überprüfbar sind.

Syntaktische Eigenschaften von Programmen, wie die Länge eines Programms, die Anzahl der Variablen oder die Anzahl der Zuweisungen, in denen eine Variable vorkommt, sind sicher berechenbar. Wie steht es um die Entscheidbarkeit von semantischen Eigenschaften wie z. B.

(1) $E_1 = \{f \in \mathcal{P} \mid f(x) = 3, \text{ für alle } x \in \mathbb{N}_0\}$ enthält alle berechenbaren Funktionen, die jeder Eingabe den Wert 3 zuweisen („Konstante 3“).

(2) $E_2 = \{f \in \mathcal{P} \mid f(4) = 17\}$ ist die Menge aller berechenbaren Funktionen, die der Eingabe 4 den Wert 17 zuweisen.

(3) $E_3 = \{f \in \mathcal{P} \mid f(x) = x, \text{ für alle } x \in \mathbb{N}_0\}$ enthält alle berechenbaren Funktionen, die äquivalent zur Identität sind.

(4) $E_4 = \{f \in \mathcal{P} \mid Def(f) = \mathbb{N}_0\}$ ist die Menge aller total berechenbaren Funktionen.

(5) $E_5 = \{f \in \mathcal{P} \mid f \equiv g\}$, $g \in \mathcal{P}$, enthält alle berechenbaren zu g äquivalenten Funktionen.

Der Satz von Rice, den wir im Folgenden vorstellen, besagt, dass alle nicht trivialen semantischen Eigenschaften von Programmen nicht entscheidbar sind.

Definition 6.1 *Es seien f und g Funktionen von $\mathbb{N}_0^k$ nach $\mathbb{N}_0$.*

a) *f heißt* **Teilfunktion** *von g genau dann, wenn $Def(f) \subseteq Def(g)$ und $f(x) = g(x)$ für alle $x \in Def(f)$ gilt. Wir notieren diese Beziehung mit $f \subseteq g$.*

b) *Ist f eine Teilfunktion von g und ist $Def(f) \neq Def(g)$, dann heißt f* **echte Teilfunktion** *von g, was wir mit $f \subset g$ notieren.*

c) *Gilt $f \subseteq g$ und $g \subseteq f$, dann heißen f und g (***funktional***) **äquivalent**, was wir mit $f \approx g$ notieren.*

d) *Es sei F eine Menge von Funktionen von $\mathbb{N}_0^k$ nach $\mathbb{N}_0$. F heißt* **funktional vollständig** *genau dann, wenn gilt: Ist $f \in F$ und $f \approx g$, dann ist auch $g \in F$.*

Wir übertragen die Begriffe der Definition 6.1 auf Nummerierungen $(\mathbb{N}_0, \mathcal{P}, \varphi)$, indem wir $i \subseteq j$, $i \subset j$ oder $i \approx j$ notieren, falls $\varphi_i \subseteq \varphi_j$, $\varphi_i \subset \varphi_j$ bzw. $\varphi_i \approx \varphi_j$ gilt.

Definition 6.2 *Es sei $(\mathbb{N}_0, \mathcal{P}, \varphi)$ eine Standardnummerierung.*

a) *Dann heißt $A \subseteq \mathbb{N}_0$* **funktional vollständig** *genau dann, wenn gilt: Ist $i \in A$ und $i \approx j$, dann ist auch $j \in A$.*

b) *Sei $E \subseteq \mathcal{P}$, dann ist $A_E = \{\, i \mid \varphi_i \in E \,\}$ die* **Indexmenge** *von E.*

c) *Die Indexmengen $A_E = \emptyset$ und $A_E = \mathbb{N}_0$ der Mengen $E = \emptyset$ bzw. $E = \mathcal{P}$ heißen* **trivial**.

Folgerung 6.1 *Sei $(\mathbb{N}_0, \mathcal{P}, \varphi)$ eine Standardnummerierung und $E \subseteq \mathcal{P}$, dann ist die Indexmenge A_E von E funktional vollständig.*

Übung 6.2 *Beweisen Sie Folgerung 6.1!*

Der folgende **Satz von Rice** besagt, dass alle nicht trivialen semantischen Eigenschaften von Programmen unentscheidbar sind.

Satz 6.9 *Sei $E \subset \mathcal{P}$ nicht trivial, dann ist die Indexmenge*

$$A_E = \{i \mid \varphi_i \in E\}$$

nicht entscheidbar.

Beweis *Sei $E \subseteq \mathcal{P}$ mit $E \neq \emptyset$ und $E \neq \mathcal{P}$. Dann ist die Indexmenge A_E von E nicht trivial. Es gibt also mindestens einen Index $p \in A_E$ und mindestens einen Index $q \notin A_E$.*

Wir nehmen an, dass A_E entscheidbar ist. Dann ist die charakteristische Funktion χ_{A_E} von A_E berechenbar, und damit ist die Funktion $f : \mathbb{N}_0 \to \mathbb{N}_0$, definiert durch

$$f(x) = \begin{cases} q, & \chi_{A_E}(x) = 1, \\ p, & \chi_{A_E}(x) = 0, \end{cases}$$

total berechenbar. Die Funktion f erfüllt somit die Voraussetzungen des Rekursionssatzes 5.7. Also gibt es ein $n \in \mathbb{N}_0$ mit $\varphi_n = \varphi_{f(n)}$. Für dieses n gilt

$$\varphi_n \in E \text{ genau dann, wenn } \varphi_{f(n)} \in E \tag{6.7}$$

gilt. Andererseits gilt wegen der Definition von f: $x \in A_E$ genau dann, wenn $f(x) = q \notin A_E$, d. h. $\varphi_x \in E$ genau dann, wenn $\varphi_{f(x)} \notin E$ ist. Dieses gilt natürlich auch für $x = n$, was einen Widerspruch zu (6.7) bedeutet. Deshalb muss unsere Annahme, dass A_E entscheidbar ist, falsch sein, d. h., für $E \neq \emptyset$ und $E \neq \mathcal{P}$ ist $A_E = \{\, i \mid \varphi_i \in E \,\}$ nicht entscheidbar.

Beispiel 6.1 *Wir betrachten die Menge E_4 vom Beginn des Abschnitts. Die Indexmenge ist $A_{E_4} = \{i \mid Def(\varphi_i) = \mathbb{N}_0\}$. Es gilt $E_4 \subseteq \mathcal{P}$, $E_4 \neq \emptyset$, denn es ist z. B. $id \in E_4$, und $E_4 \neq \mathcal{P}$, denn ω, die nirgends definierte Funktion ($Def(\omega) = \emptyset$), ist in $\mathcal{P}$, aber nicht in E_4 enthalten. Mit dem Satz von Rice folgt, dass A_{E_4} nicht entscheidbar ist. Dieses haben wir im Übrigen bereits im Beweis von Satz 6.7 auf andere Weise gezeigt.*

6.3 Das Korrektheitsproblem

Bei der Softwareentwicklung wäre es sehr hilfreich, wenn man die Korrektheit von beliebigen Programmen mithilfe von automatischen Programmbeweisern nachweisen könnte. Ein solcher Beweiser würde überprüfen, ob ein Programm P eine Funktion $f : \mathbb{N}_0^k \to \mathbb{N}_0$ berechnet, ob also $\Phi_P = f$ gilt oder nicht. Der folgende Satz besagt, dass es einen solchen universellen Programmbeweiser nicht geben kann.

Satz 6.10 *Sei $f \in \mathcal{P}$, dann ist die Indexmenge*

$$P_f = \{i \in \mathbb{N}_0 \mid \varphi_i = f\}$$

nicht entscheidbar.

Beweis *Wenn wir $E = \{f\}$ wählen, sind die beiden Voraussetzungen von Satz 6.9, $E \neq \emptyset$ und $E \neq \mathcal{P}$, erfüllt. Damit ist $A_E = \{i \in \mathbb{N}_0 \mid \varphi_i = f\} = \{i \in \mathbb{N}_0 \mid \varphi_i \in E\}$ nicht entscheidbar.*

Die Indexmenge P_f ist nicht entscheidbar, d. h., wenn ein Programm i gegeben ist, dann ist es nicht entscheidbar, ob i die Funktion f berechnet, das Programm i also korrekt ist, oder ob $i \notin P_f$ ist, d. h., ob f nicht von i berechnet wird, das Programm i also nicht korrekt ist.

Da also *im Nachhinein* ein (automatischer) Korrektheitsbeweis nicht möglich ist und das Testen von Programmen dem Finden von Fehlern dient und nicht die Korrektheit von Programmen garantiert, müssen Programmiererinnen und Programmierer *während* der Konstruktion von Programmen für Korrektheit sorgen. Mit Methoden und Verfahren zur Konstruktion von korrekten Programmen beschäftigt sich die *Programmverifikation*.

6.4 Das Äquivalenzproblem

Ein weiteres Problem mit praktischer Bedeutung ist das Äquivalenzproblem. Angenommen, es stehen zwei Programme zur Verfügung, z. B. unterschiedliche Releases

einer Software, die ein Problem lösen sollen. Eine interessante Frage ist die nach der Äquivalenz der Programme, d. h., ob sie dasselbe Problem berechnen. Gäbe es einen Äquivalenzbeweiser, dann könnte man ihn zu Hilfe nehmen, um diese Frage zu entscheiden. Im positiven Falle könnte man dann aufgrund weiterer Qualitätskriterien (u.a. Effizienz, Benutzerfreundlichkeit) eines der äquivalenten Programme auswählen. Der folgende Satz besagt, dass es keinen universellen Äquivalenzbeweiser geben kann.

Satz 6.11 *Die Menge*

$$A = \{(i,j) \in \mathbb{N}_0 \times \mathbb{N}_0 \mid \varphi_i = \varphi_j\}$$

ist nicht entscheidbar.

Beweis *Wir reduzieren für ein $f \in \mathcal{P}$ die bereits in Satz 6.10 als unentscheidbar gezeigte Indexmenge P_f auf die Indexmenge A, woraus die Behauptung folgt (P_f ist ein Spezialfall von A).*

Zu $f \in \mathcal{P}$ existiert ein $i_f \in \mathbb{N}_0$ mit $\varphi_{i_f} = f$. Wir definieren die Funktion $g : \mathbb{N}_0 \to \mathbb{N}_0$ durch $g(j) = (j, i_f)$. g ist offensichtlich total berechenbar. Es gilt

$$\begin{aligned} j \in \mathcal{P}_f \text{ genau dann, wenn } & \varphi_j = f, \\ \text{genau dann, wenn } & \varphi_j = \varphi_{i_f}, \\ \text{genau dann, wenn } & (j, i_f) \in A, \\ \text{genau dann, wenn } & g(j) \in A, \end{aligned}$$

woraus $P_f \leq_g A$ und damit die Behauptung folgt.

6.5 Zusammenfassung und bibliografische Hinweise

Da die Menge der charakteristischen Funktionen überabzählbar ist, müssen nicht entscheidbare Mengen existieren. Beispiele für solche Mengen sind das Selbstanwendbarkeitsproblem und dessen Verallgemeinerung, das Halteproblem. Da beide Mengen rekursiv aufzählbar sind, sind ihre Komplemente nicht rekursiv aufzählbar. Deren semi-charakteristische Funktionen sind somit Beispiele für nicht berechenbare Funktionen.

Der Satz von Rice besagt, dass alle nicht trivialen semantischen Eigenschaften von Programmen unentscheidbar sind. Dazu gehören das Korrektheitsproblem und das Äquivalenzproblem, die auch für die praktische Programmentwicklung von Bedeutung sind.

Die Darstellungen in diesem Kapitel folgen den entsprechenden Darstellungen in [VW16]. Dort wird zudem ein Beweis des **Erweiterten Satzes von Rice** geführt. Dieser Satz besagt: Ist A eine nicht triviale Indexmenge und gibt es $p, q \in \mathbb{N}_0$ mit $p \in A$ und $q \notin A$ sowie $p \subseteq q$, dann ist A nicht rekursiv aufzählbar. Sowohl das Korrektheitsproblem als auch das Äquivalenzproblem erfüllen die zusätzlichen Voraussetzungen des Erweiterten Satzes von Rice; sie sind also nicht rekursiv aufzählbar.

Die Themen dieses Kapitels werden in [BL74], [HS01], [Koz97], [Ro67] und [Wei87] weitaus ausführlicher behandelt als in diesem Buch.

Kapitel 7

Kolmogorov-Komplexität

Im Abschnitt 1.2 haben wir beispielhaft Möglichkeiten zur Codierung von Zeichenketten durch Bitfolgen sowie Beispiele für die Komprimierung von Bitfolgen kennengelernt. Jetzt betrachten wir zunächst Möglichkeiten zur Codierung von Bitfolgen-Sequenzen, die wir für die Definition der Kolmogorov-Komplexität im übernächsten Abschnitt benötigen. Dort spielen nämlich universelle Turingmaschinen (siehe Abschnitt 5.1) eine wesentliche Rolle, und diese bekommen als Eingaben Codierungen von Maschinen und Eingaben. Ist U eine universelle Turingmaschine, dann muss $\Phi_U\langle T, x\rangle = \Phi_T(x)$ gelten. Dabei muss $\langle T, x\rangle \in \mathbb{B}^*$ sein, d. h., die Maschine T, das Komma (oder irgendein anderes Trennsymbol zwischen Maschine und Eingabe) und die Eingabe x müssen so als *eine* Bitfolge codiert sein, dass die drei Einzelteile aus der Codierung rekonstruiert werden können.

Des Weiteren sollen die Codierungen präfixfrei sein, denn eine Codierung von T sollte kein echtes Präfix erhalten, das die Codierung einer Turingmaschine T' ist. Denn sonst würde U möglicherweise die Maschine T' ausführen und nicht die Maschine T.

7.1 Codierung von Bitfolgen-Sequenzen

Um eine solche Codierung zu erhalten, gehen wir von der im Abschnitt 5.2 eingeführten Nummerierung $\rho\langle\cdot\rangle : \mathcal{T} \to \mathbb{N}_0$ der Turingmaschinen aus. Ist $\rho\langle T\rangle = i$, dann besteht die Zahl $i = i_1 \ldots i_k$ aus den Ziffern $i_j \in \{0, \ldots, 7\}$, $1 \leq j \leq k$, mit $i_1 = 6$ und $i_{k-1} = i_k = 7$. Wir codieren die Nummern $\rho\langle T\rangle = i$ jetzt ziffernweise binär (siehe Definitionen (2.11) und (2.12)). Dazu bezeichnen wir die Menge der Ziffern $0, \ldots, 7$ mit Ω (in Abschnitt 5.2 werden die Symbole des Alphabets Ω mit den Ziffern $0, \ldots, 7$ codiert).

Daraus ergibt sich mit der Abbildung $bin_\Omega : \Omega \to \mathbb{B}^3$ (es ist $\ell(\Omega) = \lceil \log |\Omega| \rceil = 3$) die Binärcodierung aller Turingmaschinen

$$\mathcal{M}_\mathcal{T} = \{bin_\Omega(\rho\langle T\rangle) \mid T \in \mathcal{T}\}. \tag{7.1}$$

K.-U. Witt und M. E. Müller, *Algorithmische Informationstheorie*,
https://doi.org/10.1007/978-3-662-61694-9_7

Da die Zeichenkette „))“ in der Codierung $\langle T \rangle$ für jede Turingmaschine $T \in \mathcal{T}$ genau ein Mal und immer am Ende vorkommt, markiert die Sequenz 111 111 auch immer eindeutig das Ende von $bin_\Omega(\rho \langle T \rangle)$. Deshalb ist diese Codierung präfixfrei (siehe Bemerkung 2.3).

Diese Art der präfixfreien Binärcodierung kann für jede Programmiersprache (μ-rekursive Funktionen, WHILE-Programme, JAVA-Programme) vorgenommen werden. Wir gehen im Folgenden davon aus, dass eine vollständige Programmiersprache in präfixfreier Binärcodierung gegeben ist. Die Menge dieser Codierungen bezeichnen wir im Allgemeinen mit $\mathcal{M}$.

Die Eingabe für ein universelles Programm $U \in \mathcal{M}$ besteht aus einem Programm $A \in \mathcal{M}$ und einer Eingabe $x \in \mathbb{B}^*$ für A. Die gesamte Eingabe für U ist das Paar (A, x). Diese, d. h., das Programm A, das Komma und die Eingabe x, müssen als eine Bitfolge $\langle A, x \rangle \in \mathbb{B}^*$ codiert werden, sodass U erkennen kann, wo das Programm A aufhört und die Eingabe x beginnt.

Wir verwenden den „Trick“, den wir schon im Abschnitt 1.2 verwendet haben, um Sonderzeichen, hier das Komma zur Trennung von zwei Bitfolgen x und y, zu codieren: Die Bits der Folge x werden verdoppelt, das Komma wird durch 01 oder 10 codiert, und anschließend folgen die Bits von y. Eine solche Codierung bezeichnen wir im Folgenden mit $\langle x, y \rangle$. Wenn wir für das Komma 01 wählen, ist z. B.

$$\langle 01011, 11 \rangle = 0011001111\,01\,11.$$

Für die Länge dieser Codierung gilt

$$|\langle x, y \rangle| = 2\,|x| + 2 + |y| = 2(|x| + 1) + |y|\,. \tag{7.2}$$

Für den Fall, dass U auf eine Folge $A_i \in \mathcal{M}$, $1 \leq i \leq m$, ausgeführt werden soll, legen wir die Codierung wie folgt fest: Die Bits in den Folgen A_i werden verdoppelt, die Kommata zwischen den A_i werden durch 10 codiert, das Komma vor der Eingabe x wird durch 01 codiert, worauf x unverändert folgt. Für die formale Beschreibung dieser Codierung verwenden wir die Bitverdopplungsfunktion $d : \mathbb{B}^* \to \mathbb{B}^*$, definiert durch

$$\begin{aligned} d(\varepsilon) &= \varepsilon, \\ d(bw) &= bb \circ d(w) \text{ für } b \in \mathbb{B} \text{ und } w \in \mathbb{B}^*. \end{aligned} \tag{7.3}$$

Damit definieren wir

$$\langle A_1, \ldots, A_m, x \rangle = d(A_1)\,10\,\ldots\,10\,d(A_m)\,01\,x. \tag{7.4}$$

Die Länge dieser Codierung ist

$$|\langle A_1, \ldots, A_m, x \rangle| = 2 \sum_{i=1}^{m} (|A_i| + 1) + |x|\,. \tag{7.5}$$

7.2 Definitionen

Wir haben in Abschnitt 1.2 Möglichkeiten zur Codierung und Komprimierung von Bitfolgen sowie im obigen Abschnitt Möglichkeiten zur Codierung und Komprimierung von Bitfolgen-Sequenzen kennengelernt. Es stellen sich nun Fragen wie: Welche Auswirkung hat die Komplexität einer Folge auf die Komprimierung? Wie können die Folgen beschrieben werden, sodass die Beschreibungen vergleichbar sind? Mit welchem Verfahren erreicht man optimale Beschreibungen?

Ein Ansatz für die Beschreibung der Komplexität von Bitfolgen ist die *Kolmogorov-Komplexität*. Dafür legen wir eine vollständige, binär codierte Programiersprache $\mathcal{M}$ zugrunde, d. h., $\mathcal{M}$ erfüllt das utm-Theorem und das smn-Theorem. Wir können also z. B. Turingmaschinen, While-Programme oder μ-rekursive Funktionen wählen.

Sei $A \in \mathcal{M}$, dann ist die Funktion $\mathcal{K}_A : \mathbb{B}^* \to \mathbb{N}_0 \cup \{\infty\}$ definiert durch[1]

$$\mathcal{K}_A(x) = \begin{cases} \min\left\{|y| : y \in \Phi_A^{-1}(x)\right\}, & \text{falls } \Phi_A^{-1}(x) \neq \emptyset, \\ \infty, & \text{sonst.} \end{cases} \tag{7.6}$$

$\mathcal{K}_A(x)$ ist die Länge der kürzesten Bitfolge y, woraus das Programm A die Bitfolge x berechnet, falls ein solches y existiert.

In dieser Definition hängt $\mathcal{K}_A(x)$ nicht nur von x, sondern auch von dem gewählten Programm A ab. Der folgende Satz besagt, dass wir von dieser Abhängigkeit absehen können.

Satz 7.1 *Sei U ein universelles Programm der vollständigen Programmiersprache $\mathcal{M}$. Dann existiert zu jedem Programm $A \in \mathcal{M}$ eine Konstante c_A, sodass*

$$\mathcal{K}_U(x) \leq \mathcal{K}_A(x) + c_A = \mathcal{K}_A(x) + \mathsf{O}(1)$$

für alle $x \in \mathbb{B}^*$ gilt.

Beweis *Es sei $\Phi_A^{-1}(x) \neq \emptyset$. Für alle $y \in \Phi_A^{-1}(x)$ gilt $\Phi_A(y) = x$ und damit $\Phi_U\langle A, y\rangle = x$. Hieraus folgt $\mathcal{K}_U(x) \leq 2\,(\,|\langle A\rangle| + 1\,) + |y|$. Dieses gilt auch für das lexikografisch kleinste $y \in \Phi_A^{-1}(x)$, d. h., es gilt*

$$\mathcal{K}_U(x) \leq \min\left\{|y| : y \in \Phi_A^{-1}(x)\right\} + 2\,(\,|\langle A\rangle| + 1\,).$$

Das Programm A ist unabhängig von x, d. h., wir können $c_A = 2\,(\,|\langle A\rangle| + 1\,)$ als Konstante annehmen und erhalten damit die Behauptung.

Falls $\Phi_A^{-1}(x) = \emptyset$ ist, folgt mit analoger Argumentation $\mathcal{K}_U(x) \leq \infty + c_A = \infty$.

Wir können also $\mathcal{K}_A(x)$ für jedes $A \in \mathcal{M}$ ohne wesentliche Beeinträchtigung der folgenden Betrachtungen durch $\mathcal{K}_U(x)$ ersetzen. Es bleibt jetzt noch die Abhängigkeit

[1] Wir benutzen im Folgenden das Symbol ∞ mit der Bedeutung $\infty > n$ für alle $n \in \mathbb{N}_0$ sowie mit $n + \infty = \infty$ bzw. $n \cdot \infty = \infty$ für alle $n \in \mathbb{N}_0$.

von der gewählten Programmiersprache $\mathcal{M}$. Der folgende Satz besagt, dass wir auch diese Abhängigkeit vernachlässigen können. Als Vorbereitung auf diesen Satz ziehen wir zunächst zwei Folgerungen aus Satz 7.1.

Folgerung 7.1 *Es seien $\mathcal{M}_1$ und $\mathcal{M}_2$ vollständige Programmiersprachen.*

a) *U_1 sei ein universelles Programm von $\mathcal{M}_1$. Dann existiert zu jedem Programm $B \in \mathcal{M}_2$ eine Konstante c mit $\mathcal{K}_{U_1}(x) \leq \mathcal{K}_B(x) + c$ für alle $x \in \mathbb{B}^*$.*

b) *Seien U_1 und U_2 universelle Programme von $\mathcal{M}_1$ bzw. von $\mathcal{M}_2$. Dann existiert eine Konstante c, sodass für alle $x \in \mathbb{B}^*$ gilt: $\mathcal{K}_{U_1}(x) \leq \mathcal{K}_{U_2}(x) + c$.*

Beweis **a)** *Gemäß Satz 5.9, dem Äquvalenzsatz von Rogers, existiert ein Programm $T_{2\to1} \in \mathcal{M}_1$ mit*

$$\Phi_B^{(\mathcal{M}_2)}(x) = \Phi_{T_{2\to1}(B)}^{(\mathcal{M}_1)}(x) \tag{7.7}$$

für alle $x \in \mathbb{B}^*$. Da $T_{2\to1}(B) \in \mathcal{M}_1$ ist, gibt es gemäß Satz 7.1 eine Konstante c mit $\mathcal{K}_{U_1}(x) \leq \mathcal{K}_{T_{2\to1}(B)}(x) + c$. Hieraus folgt mit (7.7) die Behauptung.

b) *Folgt unmittelbar aus a), wenn wir U_2 für B einsetzen.*

Satz 7.2 *Es seien $\mathcal{M}_1$ und $\mathcal{M}_2$ zwei vollständige Programmiersprachen, und U_1 und U_2 seien universelle Programme von $\mathcal{M}_1$ bzw. $\mathcal{M}_2$. Dann existiert eine Konstante c, so dass für alle $x \in \mathbb{B}^*$ gilt:*

$$|\mathcal{K}_{U_1}(x) - \mathcal{K}_{U_2}(x)| \leq c = \mathsf{O}(1)$$

Beweis *Gemäß Folgerung 7.1 b) gibt es zwei Konstanten c_1 und c_2, sodass $\mathcal{K}_{U_1}(x) \leq \mathcal{K}_{U_2}(x) + c_1$ bzw. $\mathcal{K}_{U_2}(x) \leq \mathcal{K}_{U_1}(x) + c_2$ für alle $x \in \mathbb{B}^*$ gelten. Damit ist $\mathcal{K}_{U_1}(x) - \mathcal{K}_{U_2}(x) \leq c_1$ bzw. $\mathcal{K}_{U_2}(x) - \mathcal{K}_{U_1}(x) \leq c_2$. Für $c = \max\{c_1, c_2\}$ gilt dann die Behauptung.*

Die Konstante c misst quasi den Aufwand für den (größeren der beiden) Übersetzer zwischen $\mathcal{M}_1$ und $\mathcal{M}_2$.

Aufgrund der Sätze 7.1 und 7.2 unterscheiden sich die Längenmaße $\mathcal{K}$ für Programme und universelle Programme aus einer und auch aus verschiedenen Programmiersprachen jeweils nur um eine additive Konstante. Diesen Unterschied wollen wir als unwesentlich betrachten (obwohl die Konstante, die die Größe eines Übersetzers angibt, sehr groß sein kann). Insofern sind die folgenden Definitionen unabhängig von der gewählten Programmiersprache und unabhängig von dem in dieser Sprache gewählten universellen Programm. Deshalb wechseln wir im Folgenden je nachdem, wie es für die jeweilige Argumentation oder Darstellung verständlicher erscheint, das Be-

rechnungsmodell; so verwenden wir etwa im Einzelfall Turingmaschinen, While-Programme oder μ-rekursive Funktionen. Mit $\mathcal{M}$ bezeichnen wir im Folgenden unabhängig vom Berechnungsmodell die Menge aller Programme, also z. B. die Menge aller Turingmaschinen, die Menge aller While-Programme bzw. die Menge aller μ-rekursiven Funktionen. Mit U bezeichnen wir universelle Programme in diesen Berechnungsmodellen, also universelle Turingmaschinen, universelle While-Programme bzw. universelle Funktionen.

Definition 7.1 *Sei A ein Pogramm von $\mathcal{M}$ sowie $y \in \mathbb{B}^*$ mit $y \in Def(\Phi_A(y))$ und $\Phi_A(y) = x$, dann heißt $b(x) = \langle A, y \rangle$* **Beschreibung** *von x. Sei $\mathcal{B}(x)$ die Menge aller Beschreibungen von x, dann ist $b^*(x) = \min\{|y| : y \in \mathcal{B}(x)\}$ die lexikografisch kürzeste Beschreibung von x. Wir nennen die Funktion $\mathcal{K} : \mathbb{B}^* \to \mathbb{N}_0$, definiert durch*

$$\mathcal{K}(x) = |b^*(x)|,$$

die **Beschreibungs-** *oder* **Kolmogorov-Komplexität** *von x. Die Kolmogorov-Komplexität des Wortes x ist die Länge der lexikografisch kürzesten Bitfolge $\langle A, y \rangle$, sodass das Programm A bei Eingabe y das Wort x ausgibt.*

Wir können $\mathcal{K}(x)$ als den Umfang an Informationen verstehen, die durch die Bitfolge x dargestellt wird.

Bemerkung 7.1 *Da die lexikografische Ordnung von Bitfolgen total ist, ist die kürzeste Beschreibung $b^*(x) = \langle A, y \rangle$ einer Folge x eindeutig. Außerdem gilt $\Phi_U\langle A, y \rangle = \Phi_A(y) = x$ für jedes universelle Programm U. Deshalb können wir das Programm A und die Eingabe y zu einem Programm verschmelzen, d. h., wir betrachten y als Teil des Programms A (y ist quasi „fest in A verdrahtet"). Diese Verschmelzung bezeichnen wir ebenfalls mit A. Wenn wir nun für diese Programme als Default-Eingabe das leere Wort vorsehen (das natürlich nicht codiert werden muss), gilt $\Phi_U\langle A, \varepsilon \rangle = x$. Da ε die Default-Eingabe ist, können wir diese auch weglassen, d. h., wir schreiben $\Phi_U\langle A \rangle$ anstelle von $\Phi_U\langle A, \varepsilon \rangle$. Mit dieser Sichtweise kann die obige Definition unwesentlich abgeändert werden zu:*

Gilt $\Phi_U\langle A \rangle = x$, dann heißt $b(x) = \langle A \rangle$ eine **Beschreibung** *von x. Sei $\mathcal{B}(x)$ die Menge aller Beschreibungen von x, dann ist $b^*(x) = \min\{|A| : A \in \mathcal{B}(x)\}$ die lexikografisch kürzeste Beschreibung von x. Wir nennen die Funktion $\mathcal{K} : \mathbb{B}^* \to \mathbb{N}_0$, definiert durch*

$$\mathcal{K}(x) = \min\{|A| : A \in \mathcal{B}(x)\},$$

die **Beschreibungs-** *oder* **Kolmogorov-Komplexität** *von x.*

Für natürliche Zahlen $n \in \mathbb{N}_0$ legen wir die Kolmogorov-Komplexität fest als

$$\mathcal{K}(n) = \mathcal{K}(dual(n)). \tag{7.8}$$

7.3 Eigenschaften

Als Erstes können wir überlegen, dass $\mathcal{K}(x)$ im Wesentlichen nach oben durch $|x|$ beschränkt ist, denn es gibt sicherlich ein Programm, das nichts anderes leistet als x zu erzeugen. Ein solches Programm enthält Anweisungen und bekommt x als Eingabe oder enthält x „fest verdrahtet“, woraus dann – in beiden Fällen – x erzeugt wird. Somit ergibt sich die Länge dieses Programms aus der Länge von x und der Länge der Anweisungen, die unabhängig von der Länge von x ist. Ein kürzestes Programm zur Erzeugung von x ist höchstens so lang wie x selbst.

Satz 7.3 *Es existiert eine Konstante c, sodass für alle $x \in \mathbb{B}^*$*

$$\mathcal{K}(x) \leq |x| + c = |x| + \mathsf{O}(1) \tag{7.9}$$

gilt.

Beweis *Die Turingmaschine T aus der Lösung von Aufgabe 3.9 entscheidet den Graphen der identischen Funktion $id_{\mathbb{B}^*}$, d. h., T berechnet diese. Es gilt also $\Phi_T(x) = x$ für alle $x \in \mathbb{B}^*$. Es ist also $b(x) = \langle T, x \rangle$ eine Beschreibung für x. T ist unabhängig von x; wir können also $c = |\langle T \rangle|$ setzen. Damit haben wir $|b(x)| = c + |x|$. Mit Definition 7.1 folgt unmittelbar*

$$\mathcal{K}(x) = |b^*(x)| \leq |b(x)| = |x| + c,$$

womit die Behauptung gezeigt ist.

Die Information, die eine Bitfolge x enthält, ist also in keinem Fall wesentlich länger als x selbst – was sicherlich auch intuitiv einsichtig ist.

Bemerkung 7.2 *Im Beweis von Satz 7.3 haben wir eine feste Maschine gewählt, die bei Eingabe $x \in \mathbb{B}^*$ die Ausgabe x berechnet. Gemäß Bemerkung 7.1 können wir aber für jedes x eine eigene Maschine T_x konstruieren, in der x fest verdrahtet ist. Sei $x = x_1 \ldots x_l$, $x_i \in \mathbb{B}$, $1 \leq i \leq l$, dann gilt z. B. für den 2-Bandakzeptor*

$$T_x = (\{0, 1, \#\}, \{s_0, \ldots, s_l\}, \delta, s_0, \#, s_l) \tag{7.10}$$

mit $\delta = \{(s_i, \#, \#, s_{i+1}, x_{i+1}, x_{i+1}, \rightarrow) \mid 0 \leq i \leq l-1\}$: $\Phi_{T_x}(\varepsilon) = x$. Es ist also $b(x) = \langle T_x \rangle$ eine Beschreibung von x. Die Länge von T_x hängt allerdings von der Länge von x ab. Es gilt $|b(x)| = |x| + c$ und damit auch hier

$$\mathcal{K}(x) = |b^*(x)| \leq |b(x)| = |x| + c.$$

Dabei ist c die Länge des „Overheads“, d. h., c gibt die Länge des Anteils von T_x an, der unabhängig von x ist. Dieser ist für alle x identisch, wie z. B. das Arbeitsalphabet und das Blanksymbol sowie Sonderzeichen wie Klammern und Kommata, siehe (7.10).

Wie die durch Satz 7.3 bestätigte Vermutung scheint ebenfalls die Vermutung einsichtig, dass die Folge xx nicht wesentlich mehr Information enthält als x alleine. Das folgende Lemma bestätigt diese Vermutung.

Lemma 7.1 *Es existiert eine Konstante c, sodass für alle $x \in \mathbb{B}^*$*

$$\mathcal{K}(xx) \leq \mathcal{K}(x) + c \leq |x| + \mathsf{O}(1) \tag{7.11}$$

gilt.

Beweis *Sei $b^*(x) = \langle B \rangle \in \mathcal{M}$ die kürzeste Beschreibung von x, d. h., es ist $\Phi_U\langle B \rangle = x$. Sei nun $A \in \mathcal{M}$ eine Maschine, welche die Funktion $f : \mathbb{B}^* \to \mathbb{B}^*$, definiert durch $f(x) = xx$, berechnet: $\Phi_A(x) = f(x) = xx$, $x \in \mathbb{B}^*$. Damit gilt*

$$\Phi_A\langle \Phi_U\langle B \rangle \rangle = f(\Phi_U\langle B \rangle) = \Phi_U\langle B \rangle \Phi_U\langle B \rangle = xx. \tag{7.12}$$

Seien i ein Index von A und u ein Index von U, dann gilt wegen (7.12)

$$\varphi_i \circ \varphi_u \langle B \rangle = \varphi_u \langle B \rangle \varphi_u \langle B \rangle. \tag{7.13}$$

Wegen Bemerkung 5.1 gibt es dann einen Index k und eine total berechenbare Funktion s mit

$$\varphi_{s\langle k,i,u \rangle} \langle B \rangle = \varphi_u \langle B \rangle \varphi_u \langle B \rangle. \tag{7.14}$$

Sei nun $D = \xi(s\langle k, i, u\rangle)$, d. h., D ist die Turingmaschine mit der Nummer $s\langle k, i, u\rangle$, dann haben wir mit (7.12), (7.13) und (7.14)

$$\Phi_D\langle B \rangle = xx \tag{7.15}$$

und damit $\Phi_U\langle D, B \rangle = xx$. Wegen $b^*(x) = \langle B \rangle$ gilt dann

$$\Phi_U\langle D, b^*(x) \rangle = xx, \tag{7.16}$$

woraus sich ergibt, dass $b(xx) = \langle D, b(x^*) \rangle$ eine Beschreibung von xx ist. Es folgt $|b(xx)| = c + |b^*(x)|$ mit $c = 2(|\langle D \rangle| + 1)$ (siehe (7.5)); c ist konstant, da D unabhängig von x ist. Damit sowie mit Satz 7.3 erhalten wir

$$\mathcal{K}(xx) = |b^*(xx)| \leq |b(xx)| = |b^*(x)| + c = \mathcal{K}(x) + c \leq |x| + \mathsf{O}(1)$$

womit die Behauptung gezeigt ist.

Wie bei der Verdopplung kann man vermuten, dass auch der Informationsgehalt der Spiegelung $\overleftarrow{x}$ von $x \in \mathbb{B}^*$ sich unwesentlich von dem von x unterscheidet.

Lemma 7.2 *Es existiert eine Konstante c, sodass für alle $x \in \mathbb{B}^*$*

$$\mathcal{K}(\overleftarrow{x}) \leq \mathcal{K}(x) + c = \mathcal{K}(x) + \mathsf{O}(1) \leq |x| + \mathsf{O}(1) \tag{7.17}$$

gilt.

Beweis *Sei $b^*(x) = \langle B \rangle \in \mathcal{M}$ die kürzeste Beschreibung von x, d. h., es ist $\Phi_U\langle B \rangle = x$. Sei nun $A \in \mathcal{M}$ ein Programm, das die Funktion $g : \mathbb{B}^* \to \mathbb{B}^*$, definiert durch $g(x) = \overleftarrow{x}$, berechnet: $\Phi_A(x) = g(x) = \overleftarrow{x}$, $x \in \mathbb{B}^*$. Dann gilt*

$$\Phi_A\langle \Phi_U\langle B \rangle \rangle = \overleftarrow{(\Phi_U\langle B \rangle)} = \overleftarrow{x}\,. \tag{7.18}$$

Seien i ein Index von A und u ein Index von U, dann gilt wegen (7.18)

$$\varphi_i \circ \varphi_u \langle B \rangle = \overleftarrow{(\varphi_u \langle B \rangle)}. \tag{7.19}$$

Wegen Bemerkung 5.1 gibt es dann einen Index k und eine total berechenbare Funktion s mit

$$\varphi_{s\langle k,i,u \rangle}\langle B \rangle = \overleftarrow{(\varphi_u \langle B \rangle)}. \tag{7.20}$$

Sei nun $R = \xi(s \langle k, i, u \rangle)$, dann haben wir mit (7.18), (7.19) und (7.20)

$$\Phi_R\langle B \rangle = \overleftarrow{x} \tag{7.21}$$

und damit $\Phi_U\langle R, B \rangle = \overleftarrow{x}$. Wegen $b^*(x) = \langle B \rangle$ gilt dann

$$\Phi_U\langle R, b^*(x) \rangle = \overleftarrow{x}\,, \tag{7.22}$$

woraus sich ergibt, dass $b(\overleftarrow{x}) = \langle R, b(x^*) \rangle$ eine Beschreibung von $\overleftarrow{x}$ ist. Damit gilt $|b(\overleftarrow{x})| = c + |b^*(x)|$ mit $c = 2(|\langle R \rangle| + 1)$ (siehe (7.5)); c ist konstant, da R unabhängig von x ist. Damit sowie mit Satz 7.3 erhalten wir

$$\mathcal{K}(\overleftarrow{x}) = |b^*(\overleftarrow{x})| \leq |b(\overleftarrow{x})| = |b^*(x)| + c = \mathcal{K}(x) + c \leq |x| + \mathsf{O}(1)$$

womit die Behauptung gezeigt ist.

Aus den beiden Lemmata ergibt sich unmittelbar die nachfolgende Folgerung.

Folgerung 7.2 *Es existieren Konstanten c bzw. c', sodass für alle $x \in \mathbb{B}^*$*

a) $|\mathcal{K}(xx) - \mathcal{K}(x)| \leq c$ *sowie*

b) $|\mathcal{K}(\overleftarrow{x}) - \mathcal{K}(x)| \leq c'$

gilt. Die von x unabhängigen Konstanten c und c' messen den Aufwand für das Verdoppeln bzw. für das Spiegeln.

Die beiden Funktionen $f, g : \mathbb{B}^* \to \mathbb{B}^*$, definiert durch $f(x) = xx$ bzw. $g(x) = \overleftarrow{x}$, sind total und berechenbar. Ihre Werte tragen unwesentlich mehr Informationen in sich als ihre Argumente. Der folgende Satz besagt, dass diese Eigenschaft für alle total berechenbaren Funktionen gilt.

Satz 7.4 *Sei $f : \mathbb{B}^* \to \mathbb{B}^*$ eine berechenbare Funktion, dann existiert eine Konstante c, sodass für alle $x \in Def(f)$ gilt:*

$$\mathcal{K}(f(x)) \leq \mathcal{K}(x) + c = \mathcal{K}(x) + \mathsf{O}(1) \leq |x| + \mathsf{O}(1)$$

Beweis *Sei $b^*(x) = \langle A \rangle$ die kürzeste Beschreibung von x, d. h., es ist $\Phi_U \langle A \rangle = x$. Sei F ein Programm, das f berechnet. Dann gilt*

$$\Phi_F \langle \Phi_U \langle A \rangle \rangle = f(\Phi_U \langle A \rangle) = f(x). \tag{7.23}$$

Seien ι ein Index von F und u ein Index von U, dann gilt wegen (7.23)

$$\varphi_\iota \circ \varphi_u \langle A \rangle = f(\varphi_u \langle A \rangle). \tag{7.24}$$

Wegen Bemerkung 5.1 gibt es dann einen Index k und eine total berechenbare Funktion s mit

$$\varphi_{s\langle k,\iota,u \rangle} \langle A \rangle = f(\varphi_u \langle A \rangle). \tag{7.25}$$

Sei nun $\mathcal{F} = \xi(s \langle k, \iota, u \rangle)$, dann haben wir mit (7.23), (7.24) und (7.25)

$$\Phi_{\mathcal{F}} \langle A \rangle = f(x) \tag{7.26}$$

und damit $\Phi_U \langle \mathcal{F}, B \rangle = f(x)$. Wegen $b^*(x) = \langle A \rangle$ gilt dann

$$\Phi_U \langle \mathcal{F}, b^*(x) \rangle = f(x), \tag{7.27}$$

woraus folgt, dass $b(f(x)) = \langle \mathcal{F}, b(x^*) \rangle$ eine Beschreibung von $f(x)$ ist. Damit gilt $|b(f(x))| = c + |b^*(x)|$ mit $c = 2(|\langle \mathcal{F} \rangle| + 1)$ (siehe (7.5)); c ist konstant, da $\mathcal{F}$ unabhängig von x ist. Damit sowie mit Satz 7.3 erhalten wir

$$\mathcal{K}(f(x)) = |b^*(f(x))| \leq |b(f(x)| = |b^*(x)| + c = \mathcal{K}(x) + c \leq |x| + \mathsf{O}(1),$$

womit die Behauptung gezeigt ist.

Bemerkung 7.3 *Die Lemmata 7.1 und 7.2 sind Beispiele für die allgemeine Aussage in Satz 7.4.*

Mithilfe der obigen Sätze und Lemmata ergibt sich die nächste Folgerung.

Folgerung 7.3 **a)** *Sind $f, g : \mathbb{B}^* \to \mathbb{B}^*$ total berechenbare Funktionen. Dann existiert eine Konstante c, sodass für alle $x \in \mathbb{B}^*$*

$$\mathcal{K}(g(f(x))) \leq \mathcal{K}(x) + c \leq |x| + \mathsf{O}(1)$$

gilt.

b) *Für jede monoton wachsende, bijektive, berechenbare Funktion $f : \mathbb{N}_0 \to \mathbb{N}_0$ gibt es unendlich viele $x \in \mathbb{B}^*$ mit $\mathcal{K}(x) \leq f^{-1}(|x|)$.*

c) *Es gibt unendlich viele $x \in \mathbb{B}^*$ mit $\mathcal{K}(x) \ll |x|$.*

Beweis **a)** *Diese Aussage folgt unmittelbar aus Satz 7.4.*

b) *Zunächst halten wir fest, dass es zu jeder Konstanten $c \in \mathbb{N}_0$ ein $n_c \in \mathbb{N}_0$ gibt, sodass $\lfloor \log n \rfloor + c \leq n$ für alle $n \geq n_c$ ist. Dann überlegen wir uns ein Programm A, das bei Eingabe von $dual(n)$ die 1-Folge $x_n = 1^{f(n)}$ berechnet; es ist also $A(dual(n)) = 1^{f(n)}$ und damit $b(x_n) = \langle\, A, dual(n) \,\rangle$ eine Beschreibung für x_n. Es folgt*

$$|b(x_n)| = |\langle\, A, dual(n) \,\rangle| = 2(|\langle\, A \,\rangle| + 1) + |dual(n)| = c + \lfloor \log n \rfloor$$

mit $c = 2\,|\langle\, A \,\rangle| + 3$. Damit gilt für alle $n \geq n_c$

$$\mathcal{K}(x_n) = |b^*(x_n)| \leq |b(x_n)| = \lfloor \log n \rfloor + c \leq n = f^{-1}(f(n)) = f^{-1}(|x_n|),$$

womit die Behauptung gezeigt ist.

c) *Diese Aussage folgt unmittelbar aus b), wenn wir $f(n) = n$ setzen und damit $x_n = 1^n$ wählen.*

Nach den Vermutungen über die Komplexität von xx bzw. von $\overleftarrow{x}$, die durch die Lemmata 7.1 und 7.2 bestätigt werden, könnte man auch vermuten, dass die Kolmogorov-Komplexität $\mathcal{K}(xx')$ einer Konkatenation von zwei Folgen x und x' sich als Summe von $\mathcal{K}(x)$ und $\mathcal{K}(x')$ ergibt, d. h., dass es ein c gibt, sodass für alle $x, x' \in \mathbb{B}^*$ gilt: $\mathcal{K}(xx') \leq \mathcal{K}(x) + \mathcal{K}(x') + c$. Das ist aber nicht der Fall: Die Kolmogorov-Komplexität ist nicht subadditiv. Der Grund dafür ist, dass eine Information benötigt wird, die x und x' bzw. deren Beschreibungen voneinander abgrenzt. Dazu legen wir zunächst $\mathcal{K}(xx') = \mathcal{K}(\langle\, x, x' \,\rangle)$ fest und schreiben auch hierfür nur $\mathcal{K}\,\langle\, x, x' \,\rangle$. Somit benötigen wir für x die doppelte Anzahl von Bits. Damit können wir mithilfe von Beweisschritten analog zu den in den obigen Beweisen zeigen, dass eine Konstante c existiert, sodass für alle $x, x' \in \mathbb{B}^*$

$$\mathcal{K}(xx') \leq 2\mathcal{K}(x) + \mathcal{K}(x') + c$$

gilt. Wir können allerdings die Codierung von x kürzer gestalten, als es durch die

Verdopplung der Bits geschieht, indem wir vor xx' bzw. vor $b^*(x)b^*(x')$ die Länge von x bzw. die Länge von $b^*(x)$ – dual codiert – schreiben: $\langle\, dual(|x|), xx' \,\rangle$. Wir legen also $\mathcal{K}(xx') = \mathcal{K}(\langle\, dual(|x|), xx' \,\rangle)$ fest. Da die Bits von $dual(|x|)$ verdoppelt werden, ergibt sich (siehe (7.2))

$$|\langle\, dual(|x|), xx' \,\rangle| \leq 2(\lfloor \log |x| \rfloor + 1) + |x| + |x'|\,.$$

Falls x' kürzer als x sein sollte, kann man auch die Länge von x' vor xx' schreiben, um eine noch bessere Komprimierung zu erhalten. Diese Überlegungen führen zu folgendem Satz.

Satz 7.5 *Es existiert eine Konstante c, sodass für alle $x, x' \in \mathbb{B}^*$ gilt:*

$$\mathcal{K}(xx') \leq \mathcal{K}(x) + \mathcal{K}(x') + 2\log(\min\{\mathcal{K}(x), \mathcal{K}(x')\}) + c.$$

Beweis *Wir betrachten den Fall, dass $\mathcal{K}(x) \leq \mathcal{K}(x')$ ist, und zeigen*

$$\mathcal{K}(xx') \leq \mathcal{K}(x) + \mathcal{K}(x') + 2\log \mathcal{K}(x) + c' \tag{7.28}$$

Analog kann für den Fall, dass $\mathcal{K}(x') \leq \mathcal{K}(x)$ ist,

$$\mathcal{K}(xx') \leq \mathcal{K}(x) + \mathcal{K}(x') + 2\log \mathcal{K}(x') + c'' \tag{7.29}$$

gezeigt werden. Aus (7.28) und (7.29) folgt dann unmittelbar mit $c = \max\{c', c''\}$ die Behauptung des Satzes.

Sei also $\mathcal{K}(x) \leq \mathcal{K}(x')$, $b^*(x) = \langle\, A \,\rangle$ und $b^*(x') = \langle\, B \,\rangle$ und damit $\Phi_U\langle\, A \,\rangle = x$ bzw. $\Phi_U\langle\, B \,\rangle = x'$. Sei D ein Programm, das zunächst testet, ob eine Bitfolge die Struktur $v\,01\,w$ mit $v \in \{00, 11\}^*$ und $w \in \mathbb{B}^*$ hat. Falls das zutrifft, testet D, ob $wert(d^{-1}(v)) \leq |w|$ ist, wobei d die in (7.3) definierte Bit-Verdoppelungsfunktion ist. Falls auch das zutrifft, überprüft D, ob der Präfix der Länge $wert(d^{-1}(v))$ von w sowie der verbleibende Suffix Codierungen von Programmen sind. Gegebenenfalls führt D beide Programme aus und konkateniert ihre Ausgaben; diese Konkatenation ist die Ausgabe von D. In allen anderen Fällen ist D für die eingegebene Bitfolge nicht definiert. Für das so definierte Programm D gilt

$$\Phi_D\langle\, dual(|\langle\, A \,\rangle|), AB \,\rangle = \Phi_U\langle\, A \,\rangle\, \Phi_U\langle\, B \,\rangle$$

und damit

$$\Phi_U\langle\, D, dual(|\langle\, A \,\rangle|, AB \,\rangle = xx'.$$

Die Funktion $b(xx') = \langle\, D, dual(|\langle\, A \,\rangle|), AB \,\rangle$ ist also eine Beschreibung von xx'. Es folgt

$$\begin{aligned} |b(xx')| &= 2(|\langle\, D \,\rangle| + |dual(|\langle\, A \,\rangle|)| + 2) + |\langle\, AB \,\rangle| \\ &\leq 2(|\langle\, D \,\rangle| + 2) + 2(\log\lfloor |b^*(x)| \rfloor + 1) + |b^*(x)| + |b^*(x')|\,. \end{aligned} \tag{7.30}$$

Das Programm D ist unabhängig von x und x'. So setzen wir $c' = 2(|\langle D \rangle| + 3)$ und erhalten aus (7.30)

$$\mathcal{K}(xx') = |b^*(xx')| \leq |b(xx')| \leq \mathcal{K}(x) + \mathcal{K}(x') + 2\log\mathcal{K}(x) + c',$$

womit (7.28) gezeigt ist.

Bemerkung 7.4 *Auch wenn noch Verbesserungen möglich sein sollten, gilt jedoch, dass sich diese logarithmische Differenz zur intuitiven Vermutung generell nicht vermeiden lässt, d. h., dass $\mathcal{K}(xx') \leq \mathcal{K}(x) + \mathcal{K}(x') + c$ im Allgemeinen nicht erreichbar ist. Dieses werden wir in Satz 7.10 beweisen.*

Aus dem Satz 7.1 und dessen Beweis lässt sich leicht folgern, dass

$$\mathcal{K}(x^n) \leq \mathcal{K}(x) + \mathsf{O}(1)$$

für jedes fest gewählte n ist.

Ist n variabel, dann kann n in den Programmen zur Erzeugung von x^n nicht „fest verdrahtet“ sein, sondern n muss eine Eingabe für die Programme sein. Es geht also um die Berechnung der Funktion $f : \mathbb{N}_0 \times \mathbb{B}^* \to \mathbb{B}^*$, definiert durch $f(n, x) = x^n$. Diese können wir dual codieren durch $f : \mathbb{B}^* \to \mathbb{B}^*$ mit $f\langle\, dual(n), x \,\rangle = x^n$. Sei $b^*(x) = \langle\, A \,\rangle$ die kürzeste Beschreibung für x, womit $\Phi_U\langle\, A \,\rangle = x$ gilt. Sei F ein Programm, das – wie das im Beweis von Satz 7.5 beschriebene Programm D – zunächst testet, ob eine Bitfolge die Struktur $v\,01\,w$ mit $v \in \{00, 11\}^*$ und $w \in \mathbb{B}^*$ hat. Falls das zutrifft, testet F, ob $wert(d^{-1}(v)) = |w|$ ist, wobei d die in (7.3) definierte Bit-Verdoppelungsfunktion ist. Falls auch das zutrifft, überprüft F, ob w Codierung eines Programms ist. Gegebenenfalls führt F dieses aus und kopiert dessen Ausgabe $(wert(d^{-1}(v)) - 1)$-Mal; die Konkatenation der insgesamt $wert(d^{-1}(v))$ Kopien ist die Ausgabe von F. In allen anderen Fällen ist F für die eingegebene Bitfolge nicht definiert. Für das so definierte Programm F gilt

$$\Phi_F\langle\, dual(n), A \,\rangle = (\Phi_U\langle\, A \,\rangle)^n.$$

Damit gilt

$$\Phi_U\langle\, F, dual(n), A \,\rangle = x^n,$$

und $b(x^n) = \langle\, F, dual(n), b^*(x) \,\rangle$ ist eine Beschreibung von x^n. Es folgt

$$\begin{aligned} |b(x^n)| &= 2(|\langle\, F \,\rangle| + |dual(n)| + 2) + |\langle\, A \,\rangle| \\ &\leq 2(|\langle\, F \,\rangle| + \lfloor \log n \rfloor + 3) + |b^*(x)| \\ &= 2(|\langle\, F \,\rangle| + 3) + 2\,\lfloor \log n \rfloor + |b^*(x)|\,. \end{aligned}$$

Es folgt mit $c = 2(|\langle F \rangle| + 3)$

$$\begin{aligned}\mathcal{K}(x^n) &= |b^*(x^n)| \\ &\leq |b(x^n)| \\ &\leq |b^*(x)| + 2\lfloor \log n \rfloor + c \\ &= \mathcal{K}(x) + \mathsf{O}(\log n). \end{aligned} \tag{7.31}$$

Diese Aussage kann auch mithilfe von Satz 7.3 hergeleitet werden:

$$\begin{aligned}\mathcal{K}\langle dual(n), x \rangle &\leq |\langle dual(n), x \rangle| + \mathsf{O}(1) \\ &\leq 2(|dual(n)| + 1) + \mathcal{K}(x) + \mathsf{O}(1) \\ &\leq 2(\lfloor \log n \rfloor + 2) + \mathcal{K}(x) + \mathsf{O}(1) \\ &= \mathcal{K}(x) + \mathsf{O}(\log n)\end{aligned}$$

Wir betrachten noch den Spezialfall, dass n eine Zweierpotenz ist: $n = 2^k$, $k \in \mathbb{N}$. Dann benötigen wir zur Erzeugung von x^n nicht den Parameter n, sondern den Exponenten $k = \log n$. Dann ergibt sich aus den obigen Überlegungen

$$\begin{aligned}\mathcal{K}\langle dual(k), x \rangle &= \mathcal{K}\langle dual(\log n), x \rangle \\ &\leq |\langle dual(\log n), x \rangle| + \mathsf{O}(1) \\ &\leq 2(|dual(\log n)| + 1) + \mathcal{K}(x) + \mathsf{O}(1) \\ &\leq 2(\lfloor \log \log n \rfloor + 2) + \mathcal{K}(x) + \mathsf{O}(1) \\ &= \mathcal{K}(x) + \mathsf{O}(\log \log n). \end{aligned} \tag{7.32}$$

Als weiteren Spezialfall betrachten wir die Folgen $x = 1^l$ und $x = 0^l$ für ein beliebiges, aber festes $l \in \mathbb{N}$. Abbildung 7.1 zeigt ein Programm, welches 1^l erzeugt.

```
read();
write(1^l).
```

Abbildung 7.1: Programm zur Erzeugung des Wortes 1^l

Es folgt, dass für jedes l eine Konstante c_l existiert mit

$$\mathcal{K}(1^l) \leq c_l. \tag{7.33}$$

Entsprechendes gilt für $x = 0^l$.

Aus (7.32) und (7.33) folgt, dass für $x \in \{0, 1\}$ und $n = 2^k$ gilt:

$$\mathcal{K}(x^n) \leq \log \log n + \mathsf{O}(1) \tag{7.34}$$

Dass die Art der Codierung von Zahlen keinen wesentlichen Einfluss auf ihre Kolmogorov-Komplexität hat, besagt der folgende Satz.

Satz 7.6 *Es existiert eine Konstante c, sodass für alle $n \in \mathbb{N}_0$*

$$|\mathcal{K}(n) - \mathcal{K}(1^n)| \leq c \tag{7.35}$$

gilt.

Beweis *Wir halten zunächst fest, dass wir in (7.8) $\mathcal{K}(n) = \mathcal{K}(dual(n))$ festgelegt haben.*

Es sei $b^*(1^n) = \langle A \rangle$, also $\Phi_U\langle A \rangle = 1^n$. Sei B ein Programm, das bei Eingabe $w \in \mathbb{B}^*$ überprüft, ob w die Codierung eines Prgramms ist und gegebenenfalls das Programm ausführt. Besteht die Ausgabe aus einer Folge von Einsen, dann zählt B diese und gibt die Anzahl als Dualzahl aus. In allen anderen Fällen liefert B keine Ausgabe.

Es gilt dann $\Phi_B\langle A \rangle = dual(n)$ und demzufolge $\Phi_U\langle B, A \rangle = dual(n)$. Damit ist $b(dual(n)) = \langle B, A \rangle$ eine Beschreibung von $dual(n)$.

Es folgt

$$\begin{aligned}
\mathcal{K}(n) &= \mathcal{K}(dual(n)) \\
&= |b^*(dual(n))| \\
&\leq |b(dual(n))| \\
&= |\langle B, A \rangle| \\
&= 2(|\langle B \rangle| + 1) + |\langle A \rangle| \\
&= |b^*(1^n)| + c' \\
&= \mathcal{K}(1^n) + c'
\end{aligned} \tag{7.36}$$

für $c' = 2(|\langle B \rangle| + 1)$. Es existiert also eine Konstante c', sodass

$$\mathcal{K}(n) - \mathcal{K}(1^n) \leq c' \tag{7.37}$$

gilt.

Sei nun $b^*(dual(n)) = \langle A \rangle$, also $\Phi_U\langle A \rangle = dual(n)$. Sei B ein Programm, das bei Eingabe $w \in \mathbb{B}^*$ überprüft, ob w die Codierung eines Prgramms ist und gegebenenfalls die Ausgabe $1^{wert(\Phi_U(w))}$ erzeugt, anderenfalls liefert B keine Ausgabe.

Es gilt dann

$$\Phi_B\langle A \rangle = 1^{wert(\Phi_U\langle A \rangle)} = 1^{wert(dual(n))} = 1^n$$

und damit $\Phi_U\langle B, A \rangle = 1^n$. Es ist also $b(1^n) = \langle B, A \rangle$ eine Beschreibung von 1^n.

Es folgt

$$\begin{aligned}\mathcal{K}(1^n) &= |b^*(1^n)| \\ &\leq |b(1^n)| \\ &= |\langle B, A \rangle| \\ &= 2(|\langle B \rangle| + 1) + |\langle A \rangle| \\ &= |b^*(dual(n))| + c'' \\ &= \mathcal{K}(dual(n)) + c'' \\ &= \mathcal{K}(n) + c''\end{aligned}$$

für $c'' = 2(|\langle B \rangle| + 1)$. Es existiert also eine Konstante c'', sodass

$$\mathcal{K}(1^n) - \mathcal{K}(n) \leq c'' \tag{7.38}$$

gilt.

Mit $c = \max\{c', c''\}$ folgt aus (7.37) und (7.38) die Behauptung (7.35)

Zum Schluss dieses Abschnitts betrachten wir noch die Differenz der Kolmogorov-Komplexität $\mathcal{K}(x)$ und $\mathcal{K}(x')$ von Bitfolgen $x, x' \in \mathbb{B}^*$.

Satz 7.7 *Es existiert eine Konstante c, sodass für alle $x, h \in \mathbb{B}^*$ gilt:*

$$|\mathcal{K}(x+h) - \mathcal{K}(x)| \leq 2\,|h| + c \tag{7.39}$$

Beweis *Sei $b^*(x) = \langle A \rangle$ die kürzeste Beschreibung von x; es ist also $\Phi_U\langle A \rangle = x$. Sei F ein Programm, das zunächst testet, ob eine eingegebene Bitfolge u die Struktur $u = v01w$ mit $v \in \{00, 11\}^*$ und $w \in \mathbb{B}^*$ hat und w die Codierung eines Programms von $\mathcal{M}$ ist. Falls das nicht der Fall ist, liefert F keine Ausgabe, anderenfalls berechnet F die Summe von $d^{-1}(v)$ und der Ausgabe der Ausführung von w (falls die Ausführung w nicht terminiert, liefert F natürlich ebenfalls keine Ausgabe). Dabei ist d die in (7.3) definierte Bit-Verdopplungsfunktion. Für F gilt*

$$\Phi_F\langle h, A \rangle = h + \Phi_U\langle A \rangle = h + x$$

und damit

$$\Phi_U\langle F, h, A \rangle = x + h.$$

Es folgt, dass $b(x+h) = \langle F, h, A \rangle$ eine Beschreibung von $x + h$ ist. Damit gilt

$$\begin{aligned}\mathcal{K}(x+h) = |b^*(x+h)| \leq |b(x+h)| &= |\langle F, h, A \rangle| \\ &= 2(|\langle F \rangle| + |h| + 2) + |\langle A \rangle| \\ &= 2\,|\langle F \rangle| + 4 + 2\,|h| + |b^*(x)| \\ &= \mathcal{K}(x) + 2\,|h| + c'\end{aligned}$$

mit $c' = 2(|\langle F \rangle| + 2)$. Es existiert also eine Konstante c' mit

$$\mathcal{K}(x+h) - \mathcal{K}(x) \leq 2\,|h| + c'. \tag{7.40}$$

Sei nun $b^*(x+h) = \langle A \rangle$ die kürzeste Beschreibung von $x+h$; es ist also $\Phi_U\langle A \rangle = x+h$. Sei G ein Programm, das zunächst testet, ob eine eingegebene Bitfolge u die Struktur $u = v01w$ mit $v \in \{00,11\}^*$ und $w \in \mathbb{B}^*$ hat und w die Codierung eines Programms von $\mathcal{M}$ ist. Falls das nicht der Fall ist, liefert G keine Ausgabe, anderenfalls führt G das Programm aus und bestimmt, falls die Ausführung von G eine Ausgabe liefert, die Differenz $\Phi_U(w) - d^{-1}(v)$ (falls der Subtrahend größer als der Minuend sein sollte, ist die Ausgabe 0). Für G gilt

$$\Phi_G\langle h, A \rangle = \Phi_U\langle A \rangle - h = x + h - h$$

und damit

$$\Phi_U\langle G, h, A \rangle = x.$$

Es folgt, dass $b(x) = \langle G, h, A \rangle$ eine Beschreibung von x ist. Damit gilt

$$\begin{aligned}\mathcal{K}(x) = |b^*(x)| \leq |b(x)| &= |\langle G, h, A \rangle| \\ &= 2(|\langle G \rangle| + |h| + 2) + |\langle A \rangle| \\ &= 2\,|\langle G \rangle| + 4 + 2\,|h| + |b^*(x+h)| \\ &= \mathcal{K}(x+h) + 2\,|h| + c''\end{aligned}$$

mit $c'' = 2(|\langle G \rangle| + 2)$. Es existiert also eine Konstante c'' mit

$$\mathcal{K}(x) - \mathcal{K}(x+h) \leq 2\,|h| + c''. \tag{7.41}$$

Mit $c = \max\{c', c''\}$ folgt dann aus (7.40) und (7.41)

$$|\mathcal{K}(x+h) - \mathcal{K}(x)| \leq 2\,|h| + c,$$

womit die Behauptung gezeigt ist.

7.4 (Nicht-) Komprimierbarkeit und Zufälligkeit

Satz 7.1 zeigt, dass eine minimale Beschreibung eines Wortes niemals viel länger als das Wort selbst ist. Sicherlich gibt es Wörter, deren minimale Beschreibung sehr viel kürzer als das Wort selbst ist, etwa wenn es Redundanzen enthält, wie z. B. das Wort xx (siehe Lemma 7.1). Es stellt sich die Frage, ob es Wöter gibt, deren minimale Beschreibungen länger als sie selber sind. Wir zeigen in diesem Kapitel, dass solche Wörter existieren. Eine kurze Beschreibung eines solchen Wortes besteht dann aus einem Programm, das im Wesentlichen nichts anderes tut als dieses auszudrucken.

Zunächst führen wir den Begriff der Komprimierbarkeit von Wörtern ein und werden diesen verwenden, um einen Zufälligkeitsbegriff einzuführen.

Definition 7.2 *Sei $c \in \mathbb{N}$.*

a) *Ein Wort $x \in \mathbb{B}^*$ heißt* **c-komprimierbar** *genau dann, wenn $\mathcal{K}(x) \leq |x| - c$ gilt. Falls es ein c gibt, sodass x c-komprimierbar ist, dann nennen wir x auch* **regelmäßig**.

b) *Falls x nicht 1-komprimierbar ist, d. h., wenn $\mathcal{K}(x) \geq |x|$ ist, dann nennen wir x* **nicht komprimierbar** *oder* **zufällig**.

Wörter sind also regelmäßig, falls sie Beschreibungen besitzen, die deutlich kürzer als sie selbst sind. Diese Eigenschaft trifft insbesondere auf periodisch aufgebaute Wörter zu. Wörter sind dementsprechend zufällig, wenn ihr Aufbau keine Regelmäßigkeiten zeigt. Diese können quasi nur durch sich selbst beschrieben werden.

Der folgende Satz besagt, dass es Wörter gibt, die nicht komprimierbar im Sinne der Kolmogorov-Komplexität sind, und zwar gibt es sogar für jede Zahl n mindestens ein Wort mit der Länge n, dass nicht komprimierbar ist.

Satz 7.8 *Zu jeder Zahl $n \in \mathbb{N}$ gibt es mindestens ein Wort $x \in \mathbb{B}^*$, sodass*

$$\mathcal{K}(x) \geq |x| = n$$

ist.

Beweis *Es gilt $|\mathbb{B}^n| = 2^n$. Es seien x_i, $1 \leq i \leq 2^n$, die Elemente von $\mathbb{B}^n$ und $b^*(x_i)$ die kürzeste Beschreibung von x_i. Somit gilt*

$$\mathcal{K}(x_i) = |b^*(x_i)|\,. \tag{7.42}$$

Für $i \neq j$ ist $b^*(x_i) \neq b^*(x_j)$. Es gibt also 2^n kürzeste Beschreibungen $b^*(x_i)$, $1 \leq i \leq 2^n$. Es gilt $b^*(x_i) \in \mathbb{B}^*$ sowie $\left|\mathbb{B}^i\right| = 2^i$. Die Anzahl der nicht leeren Wörter mit einer Länge von höchstens $n-1$ ist

$$\sum_{i=1}^{n-1} \left|\mathbb{B}^i\right| = \sum_{i=1}^{n-1} 2^i = 2^n - 2 < 2^n.$$

Unter den 2^n Wörtern $b^*(x_i)$, $1 \leq i \leq 2^n$, muss es also mindestens eines mit einer Länge von mindestens n geben. Sei k mit $|b^*(x_k)| \geq n$. Es folgt mit (7.42)

$$\mathcal{K}(x_k) = |b^*(x_k)| \geq n$$

und damit die Behauptung. Das Wort $x = x_k$ ist also nicht komprimierbar.

Aus dem Beweis des Satzes folgt: Für $c \in \mathbb{N}$ kann es nur höchstens $2^{n-c+1} - 1$ Elemente $x \in \mathbb{B}^*$ geben mit $\mathcal{K}(x) \leq n - c$. Der Anteil komprimierbarer Bitfolgen beträgt somit

$$\frac{2^{n-c+1}}{2^n} = 2^{-c+1}.$$

Es gibt also z. B. weniger als 2^{n-7} Bitfolgen der Länge n mit einer Kolmogorov-Komplexität kleiner oder gleich $n-8$. Der Anteil 8-komprimierbarer Bitfolgen beträgt

$$2^{-7} < 0{,}8\,\%.$$

Das heißt, dass mehr als 99 % Prozent aller Bitfolgen der Länge n eine Kolmogorov-Komplexität größer als $n - 8$ haben. Es ist also sehr unwahrscheinlich, dass eine zufällig erzeugte Bitfolge c-komprimierbar ist.

Aus dem Satz folgt unmittelbar die nächste Folgerung.

Folgerung 7.4 *Es gibt unendlich viele Wörter $x \in \mathbb{B}^*$ mit $\mathcal{K}(x) \geq |x|$, d. h., es gibt unendlich viele Wörter, die nicht komprimierbar sind.*

Des Weiteren können wir die Mindestanzahl von Zeichenketten der Länge n angeben, die nicht c-komprimierbar sind.

Folgerung 7.5 *Seien $c, n \in \mathbb{N}$. Es gibt mindesten $2^n(1 - 2^{-c+1}) + 1$ Elemente $x \in \mathbb{B}^n$, die nicht c-komprimierbar sind.*

Beweis *Wie im Beweis von Satz 7.8 können wir überlegen, dass höchstens 2^{n-c+1} Zeichenketten der Länge n c-komprimierbar sind, weil höchstens so viele minimale Beschreibungen der Länge $n - c$ existieren. Also sind die restlichen $2^n - (2^{n-c+1} - 1) = 2^n(1 - 2^{-c+1}) + 1$ Zeichenketten nicht c-komprimierbar.*

Bemerkung 7.5 *Die Folgerung 7.5 kann wie folgt umformuliert werden: Seien $c, n \in \mathbb{N}$. Dann gibt es mindesten $2^n(1 - 2^{-c+1}) + 1$ Elemente $x \in \mathbb{B}^n$, für die*

$$\mathcal{K}(x) > n - c = \log 2^n - c = \log |\mathbb{B}^n| - c$$

gilt.

Man könnte vermuten, dass die Kolmogorov-Komplexität Präfix-monoton ist, d. h., dass $\mathcal{K}(x) \leq \mathcal{K}(xy)$ für alle $x, y \in \mathbb{B}^*$ gilt. Das trifft allerdings im Allgemeinen nicht zu; die Komplexität eines Präfixes kann größer sein als die der gesamten Bitfolge.

Dazu betrachten wir folgendes Beispiel: Sei $xy = 1^n$ mit $n = 2^k$, also $k = \log n$, dann gibt es wegen (7.34) ein c mit

$$\mathcal{K}(xy) = \mathcal{K}(1^n) \leq \log\log n + c. \tag{7.43}$$

Laut obiger Bemerkung existieren zu c und k mindestens $2^k(1-2^{-c+1})+1$ Elemente $z \in \mathbb{B}^k$ mit

$$\mathcal{K}(z) > k - c = \log n - c. \tag{7.44}$$

Für $z \in \mathbb{B}^k$ gilt $wert(z) < 2^k = n$. Damit ist $x = 1^{wert(z)}$ ein Präfix von $xy = 1^n$. Hieraus folgt mit Satz 7.6 (siehe Teil 1 des Beweises), (7.8) und (7.44), dass eine Konstante c' existiert mit

$$\begin{aligned}\mathcal{K}(x) &= \mathcal{K}\left(1^{wert(z)}\right)\\ &= \mathcal{K}(dual(wert(z))) - c'\\ &= \mathcal{K}(z) - c'\\ &> \log n - c - c'.\end{aligned} \tag{7.45}$$

Aus (7.43) und (7.45) folgt dann für hinreichend große n: $\mathcal{K}(x) > \mathcal{K}(xy)$. Komprimierbare Zeichenketten können also nicht komprimierbare Präfixe enthalten.

Andererseits gilt der nchfolgende Satz.

Satz 7.9 *Sei $d \in \mathbb{N}$ gegeben. Ein hinreichend langes Wort $x \in \mathbb{B}^*$ besitzt immer ein Präfix w, das d-komprimierbar ist, d. h., für das $\mathcal{K}(w) \leq |w| - d$ gilt.*

Beweis *Sei v ein Präfix von x mit $\tau_{\mathbb{B}}(v) = i$, d. h. mit $\nu(i) = v$. Es ist also $x = v\beta$, und v ist das i-te Wort in der lexikografischen Aufzählung der Wörter von $\mathbb{B}^*$. Sei nun w' das Infix von x der Länge i, also mit $|w'| = i$, das auf v folgt. Es ist also $x = vw'\beta'$ mit $|w'| = i$. Wir setzen $w = vw'$, womit $x = w\beta'$ ist. Das Wort w kann aus dem Wort w' erzeugt werden, denn es ist $w = \nu(|w'|)w'$. Die Funktionen $\tau_{\mathbb{B}}$, $|\cdot|$ sowie die Konkatenation von Wörtern sind total berechenbare Funktionen, mit denen aus einem Wort w' das Wort $w = vw'$ berechnet werden kann. Wegen Folgerung 7.3 gibt es eine Konstante c, sodass $\mathcal{K}(w) \leq |w'| + c$ ist, wobei c weder von x noch von v abhängt. Des Weiteren ist $|w| = |v| + |w'|$. Wenn wir also das Präfix v von x so wählen, dass $|v| \geq c + d$ ist, dann gilt*

$$\mathcal{K}(w) \leq |w'| + c = |w| - |v| + c \leq |w| - (c + d) + c = |w| - d,$$

womit die Behauptung gezeigt ist.

Mithilfe der obigen Überlegungen können wir nun auch zeigen, dass die Kolmogorov-Komplexität nicht subadditiv ist (siehe Bemerkung 7.4).

Satz 7.10 *Es existiert eine Konstante b, sodass für alle $x, x' \in \mathbb{B}^*$ mit $|x|, |x'| \leq n$*

$$\mathcal{K}(xx') > \mathcal{K}(x) + \mathcal{K}(x') + \log n - b$$

gilt.

Beweis *Es sei $B = \{(x, x') \in \mathbb{B}^* \times \mathbb{B}^* : |x| + |x'| = n\}$. Es gilt $|B| = 2^n(n+1)$. Wegen Folgerung 7.5 gibt es mindestens ein Paar $xx' \in B$, das nicht 1-komprimierbar ist. Für dieses Paar gilt wegen Bemerkung 7.5*

$$\mathcal{K}(xx') \geq \log |B| - 1 = \log (2^n(n+1)) - 1 \geq n + \log n - 1. \tag{7.46}$$

Wegen Satz 7.3 gibt es eine von x und x' unabhängige Konstante b, sodass

$$\mathcal{K}(x) + \mathcal{K}(x') \leq |x| + |x'| + b \tag{7.47}$$

gilt. Mit (7.46) und (7.47) gilt

$$\begin{aligned} \mathcal{K}(xx') &\geq n + \log n - 1 \\ &= |x| + |x'| + \log n - 1 \\ &\geq \mathcal{K}(x) + \mathcal{K}(x') - b + \log n - 1 \\ &> \mathcal{K}(x) + \mathcal{K}(x') + \log n - b, \end{aligned}$$

womit die Behauptung gezeigt ist.

Die obigen Überlegungen können wir verwenden, um die wesentliche Frage zu beantworten, ob es ein Verfahren gibt, mit dem die Kolmogorov-Komplexität $\mathcal{K}(x)$ für jedes $x \in \mathbb{B}^*$ bestimmt werden kann. Der folgende Satz beantwortet diese Frage negativ.

Satz 7.11 *Die Kolmogorov-Komplexität $\mathcal{K}$ ist nicht berechenbar.*

Beweis *Wir nehmen an, dass es ein Programm C gibt, dass die Funktion $\mathcal{K}$ für alle $x \in \mathbb{B}^*$ berechnet. Sei x_n das erste Wort in der lexikografischen Ordnung von $\mathbb{B}^*$ mit $\mathcal{K}(x_n) \geq n$ (ein solches Wort existiert gemäß Satz 7.8 und Folgerung 7.4). Die in Abbildung 7.2 dargestellte Familie W_n, $n \in \mathbb{N}$, von Programmen berechnet die Folge dieser x_n mit $\mathcal{K}(x_n) \geq n$. Da alle Programme W_n bis auf die („fest verdrahtete") Zahl n identisch sind, sind die Binärcodierungen aller W_n ohne n identisch und damit unabhängig von n konstant lang. Diese Länge sei c. Es folgt*

$$\mathcal{K}(x_n) \leq \lfloor \log n \rfloor + 1 + c.$$

Für x_n gilt aber $\mathcal{K}(x_n) \geq n$. Es folgt, dass

$$n \leq \mathcal{K}(x_n) \leq \lfloor \log n \rfloor + 1 + c$$

sein muss. Diese Ungleichung kann aber nur für endlich viele $n \in \mathbb{N}$ zutreffen. Unsere Annahme, dass die Funktion $\mathcal{K}$ berechenbar ist, ist also falsch.

```
read();
    x := ε;
    while Φ_C(x) < n do
        x := suc(x)
    endwhile;
write(x).
```

Abbildung 7.2: Programme W_n zur Erzeugung des kürzesten Wortes x_n mit $\mathcal{K}(x_n) \geq n$

Leider ist die Kolmogorov-Komplexität einer Zeichenkette nicht berechenbar. Allerdings gibt es eine berechenbare Funktion, mit der $\mathcal{K}(x)$ approximiert werden kann.

Satz 7.12 *Es gibt eine total berechenbare Funktion* $\Gamma : \mathbb{N}_0 \times \mathbb{N}_0 \to \mathbb{N}_0$ *mit*

$$\lim_{t\to\infty} \Gamma(t,x) = \mathcal{K}(x).$$

Beweis Γ *sei durch das in Abbildung 7.3 dargestellte Programm definiert. Wegen Satz 7.3 existiert eine Konstante* c *mit* $\mathcal{K}(x) \leq |x| + c$ *für alle* $x \in \mathbb{B}^*$. *Der Beweis dieses Satzes zeigt, dass sich ein solches* c *bestimmen lässt. Deswegen kann in dem in Abbildung 7.3 dargestellten Programm* $|x| + c$ *als Schleifengrenze verwendet werden. Dieses Programm benutzt außerdem das Programm* C, *das die total berechenbare Funktion* $\gamma : \mathbb{N}_0 \times \mathbb{N}_0 \times \mathbb{N}_0 \to \mathbb{N}_0$, *definiert durch*

$$\gamma(U,A,t) = \begin{cases} 1, & \textit{falls } U \textit{ angewendet auf } A \textit{ innerhalb von } t \textit{ Schritten hält}, \\ 0, & \textit{sonst}, \end{cases}$$

berechnet. Das Programm probiert in lexikografischer Ordnung alle Programme mit einer Länge kleiner gleich $|x| + c$ *durch, ob diese in höchstens* t *Schritten* x *erzeugen. Falls es solche Programme gibt, wird die Länge des kürzesten dieser Programme ausgegeben. Falls es ein solches Programm nicht gibt, wird die obere Schranke* $|x| + c$ *von* $\mathcal{K}(x)$ *ausgegeben. Es ist offensichtlich, dass das Programm für alle Eingaben* (t,x) *anhält. Die Funktion* Γ *ist also eine total berechenbare Funktion. Des Weiteren ist offensichtlich, dass* Γ *monoton fallend in* t *ist, d. h., es ist* $\Gamma(t,x) \geq \Gamma(t',x)$ *für* $t' > t$, *und ebenso offensichtlich ist, dass* $\mathcal{K}(x) \leq \Gamma(t,x)$ *für alle* t *gilt. Der Grenzwert* $\lim_{t\to\infty} \Gamma(t,x)$ *existiert für jedes* x, *denn für jedes* x *gibt es ein* t, *sodass* U *in* t *Schritten mit Ausgabe* x *anhält, nämlich wenn* U *die Eingabe eines Programms* A *mit* $\langle A \rangle = b^*(x)$, *d. h. mit* $|\langle A \rangle| = \mathcal{K}(x)$, *erhält.*

Da für jedes x *der Grenzwert* $\lim_{t\to\infty} \Gamma(t,x)$ *existiert,* $\Gamma(t,x)$ *eine monoton fallende Folge ist, die nach unten durch* $\mathcal{K}(x)$ *beschränkt ist, folgt die Behauptung.*

```
read(t, x);
    A := ε;
    M := ∅;
    while |⟨ A ⟩| < |x| + c do
        if Φ_C⟨U, A, t⟩ = 1 and Φ_U⟨ A ⟩ = x then M := M ∪ {A}
        else A := suc(A) endif
    endwhile;
    if M ≠ ∅ then return min{|⟨ A ⟩| : A ∈ M}
    else return(|x| + c)
    endif.
```

Abbildung 7.3: Programm zur Berechnung der Approximation $\Gamma(t,x)$ von $\mathcal{K}(x)$

Bemerkung 7.6 *$\mathcal{K}$ ist keine berechenbare Funktion, deswegen kann algorithmisch nicht entschieden werden, ob $\Gamma(t,x) = \mathcal{K}(x)$ ist. $\mathcal{K}(x)$ kann aber durch die total berechenbare Funktion Γ (von oben) approximiert werden; $\mathcal{K}$ ist quasi „von oben berechenbar“.*

Satz 7.13 *Es existieren berechenbare Funktionen $g, h : \mathbb{B}^* \to \mathbb{B}^*$ mit*

$$g(b^*(x)) = \langle\, x, \mathcal{K}(x) \,\rangle \quad \text{bzw. mit} \quad h\,\langle\, x, \mathcal{K}(x) \,\rangle = b^*(x).$$

Beweis *Die Funktion g kann durch das folgende Programm $A \in \mathcal{M}$ berechnet werden: A testet, ob eine Eingabe w die Codierung eines Programms ist. Falls ja, bestimmt A die Länge von w und führt w aus. Die Ausgabe von A ist dann die Ausgabe des Programms w sowie $|w|$. Es gilt also*

$$\Phi_A(w) = \begin{cases} \langle\, \Phi_U(w), |w| \,\rangle, & w \in \mathcal{M}, \\ \bot, & \textit{sonst}. \end{cases}$$

Daraus folgt

$$\Phi_A(b^*(x)) = \langle\, \Phi_U(b^*(x)), |b^*(x)| \,\rangle = \langle\, x, \mathcal{K}(x) \,\rangle$$

und damit $\Phi_A(b^*(x)) = g(b^*(x))$, d. h., g wird von A berechnet.

Die Funktion h kann durch das folgende Programm A berechnet werden: A erhält als Eingabe x sowie $\mathcal{K}(x)$, die Länge des kürzesten Programms, das x berechnet. A erzeugt der Reihe nach alle Programme mit der Länge $\mathcal{K}(x)$ und führt diese jeweils

aus. Wenn ein erzeugtes Programm B die Ausgabe x hat, dann gibt A dieses aus, es ist dann nämlich $b^*(x) = \langle B \rangle$ die kürzeste Beschreibung von x. Da $\mathcal{K}(x)$ als Eingabe existiert, muss A ein Programm dieser Länge x erzeugen.

Bemerkung 7.7 *Der Satz 7.13 besagt, dass $b^*(x)$ und $\langle x, \mathcal{K}(x) \rangle$ im Wesentlichen denselben Informationsgehalt haben. Denn mit Satz 7.4 folgt aus dem Teil 1 des obigen Satzes*

$$\mathcal{K}\langle x, \mathcal{K}(x) \rangle = \mathcal{K}(g(b^*(x))) \leq \mathcal{K}(b^*(x)) + c.$$

Wir wollen am Ende dieses Kapitels noch die Frage beantworten, ob die kürzeste Beschreibung eines Wortes komprimierbar ist. Die Vermutung, dass das nicht der Fall ist, bestätigt der folgende Satz.

Satz 7.14 *Es existiert eine Konstante $c \in \mathbb{N}$, sodass $b^*(x)$ für alle $x \in \mathbb{B}^*$ nicht c-komprimierbar ist.*

Beweis *Sei $b^*(x) = \langle B \rangle$ die kürzeste Beschreibung von x; es gilt also $\Phi_U\langle B \rangle = x$. Wir überlegen uns das folgendes Programm A mit der Eingabe y: A überprüft, ob y die Codierung eines Programms ist. Gegebenenfalls führt A dieses Programm aus. Dann überprüft A, ob die resultierende Ausgabe ebenfalls die Codierung eines Programms ist. Ist das der Fall, dann führt A auf dieses Programm das universelle Programm aus. Das Ergebnis dieser Anwendung ist dann die Ausgabe von A. In allen anderen Fällen ist A nicht definiert. Mit dieser Festlegung von A gilt*

$$\Phi_A(b^*(b^*(x))) = x, \tag{7.48}$$

denn $b^*(b^*(x))$ ist Codierung eines Programms, und $b^*(b^*(x)) = b^*\langle B \rangle$ ist das kürzeste Programm, das $\langle B \rangle$ erzeugt. Das heißt: $b^*\langle B \rangle$ ist die Codierung eines Programms, dessen Ausführung die Codierung $\langle B \rangle$ erzeugt. Die Anwendung des universellen Programms hierauf liefert die Ausgabe x. Es gilt also

$$\Phi_U\langle A, b^*(b^*(x)) \rangle = x, \tag{7.49}$$

und damit ist

$$b(x) = \langle A, b^*(b^*(x)) \rangle \tag{7.50}$$

eine Beschreibung von x. Wir setzen

$$c = 2\,|\langle A \rangle| + 3 \tag{7.51}$$

und zeigen im Folgenden, dass dieses c die Behauptung des Satzes erfüllt.

Dazu nehmen wir an, dass $b^*(x)$ c-komprimierbar für ein $x \in \mathbb{B}^*$ ist, d. h., dass für dieses x

$$\mathcal{K}(b^*(x)) \leq |b^*(x)| - c$$

und damit

$$|b^*(b^*(x))| \leq |b^*(x)| - c \tag{7.52}$$

ist. Mit (7.50), (7.51) und (7.52) folgt

$$\begin{aligned} |b^*(x)| &\leq |b(x)| \\ &= 2(|\langle A \rangle| + 1) + |b^*(b^*(x))| \\ &= c - 1 + |b^*(b^*(x))| \\ &\leq c - 1 + |b^*(x)| - c \\ &= |b^*(x)| - 1, \end{aligned}$$

was offensichtlich einen Widerspruch darstellt. Damit ist unsere Annahme widerlegt, und das gewählte c erfüllt die Behauptung.

7.5 Zusammenfassung und bibliografische Hinweise

Der Binärcode eines Programmes, das eine Bitfolge x erzeugt, kann als eine Beschreibung dieser Folge angesehen werden. Das lexikografisch kürzeste Programm, das x erzeugt, ist die Kolmogorov-Komplexität von x. Diese ist im Wesentlichen unabhängig von der gewählten Programmiersprache.

Die Anwendung von berechenbaren Funktionen auf Bitfolgen ändert ihre Kolmogorov-Komplexität nur unwesentlich. Beispiele hierfür sind die Spiegelung und die Wiederholung von Bitfolgen.

Die Kolmogorov-Komplexität der Konkatenation von zwei verschiedenen Bitfolgen ist nicht additiv. Das liegt daran, dass die Länge einer der beiden Folgen Teil der Beschreibung sein muss. Es wird gezeigt, dass es keine wesentlich kürzere Beschreibung als eine solche geben kann.

Bitfolgen, deren Kolmogorov-Komplexität wesentlich kürzer als sie selbst ist, gelten als regelmäßig und komprimierbar. Die Folgen, deren Beschreibungen größer als ihre Länge sind, gelten als zufällig und nicht komprimierbar. Es ist sehr unwahrscheinlich, dass eine zufällig erzeugte Bitfolge komprimierbar ist.

Einerseits existieren Bitfolgen, die Präfixe besitzen, deren Kolmogorov-Komplexität größer als sie selbst ist. Andererseits besitzen hinreichend lange Bitfolgen komprimierbare Präfixe.

Die Kolmogorov-Komplexität ist nicht berechenbar. Sie ist „von oben berechenbar“, d. h., sie kann durch eine total berechenbare, monoton fallende Funktion appro-

ximiert werden.

Am Ende von Abschnitt 1.2.2 wird erwähnt, dass G. Chaitin, A. Kolmogorov und R. J. Solomonoff etwa zur gleichen Zeit unabhängig voneinander, teilweise unterschiedlich motiviert ähnliche Ansätze zur Einführung und Untersuchung von Beschreibungskomplexitäten für Zeichenketten entwickelt haben. Entsprechende Literaturstellen sind für Chaitin [Ch66], für Kolmogorov [Kol65], [Kol68] und [Kol69] (in englischer Übersetzung) sowie für Solomonoff [So64a] und [So64b].

In [M66] wird begründet, dass der Zufälligkeitsbegriff von Kolmogorov allen Anforderungen der Zufälligkeit genügt.

Eine kurze Einführung in die Beschreibungskomplexität von Bitfolgen gibt [Si06]. Ein umfangreiches und umfassendes Lehrbuch zur Kolmogorov-Komplexität, das weit über die in diesem Kapitel dargestellten Aspekte hinausgeht, ist [LV93]. Wesentliche Aspekte, erläuternd dargestellt, findend man in [Ho11].

Kapitel 8

Anwendungen der Kolmogorov-Komplexität

In diesem Kapitel werden einige Problemstellungen der Theoretischen Informatik mithilfe der Kolmogorov-Komplexität beleuchtet. Wir werden beispielhaft sehen, dass bekannte Aussagen der Theoretischen Informatik, die üblicherweise mit solchen Methoden und Techniken wie der Diagonalisierung oder dem Pumping-Lemma für reguläre Sprachen gezeigt werden, auch mithilfe der Kolmogorov-Komplexität bewiesen werden können. Diese Beweise basieren auf der folgenden Idee: Zufällige Bitfolgen x, d. h., solche, bei denen $\mathcal{K}(x) \geq |x|$ ist, können nicht komprimiert werden. Sei nun das Prädikat $P(y)$ zu beweisen. Für einen Widerspruchsbeweis nimmt man an, dass $\neg P(x)$ gilt für ein nicht komprimierbares x. Folgt nun aus der Annahme, dass x komprimiert werden kann, ist ein Widerspruch hergeleitet, und die Annahme muss falsch sein.

Des Weiteren werden wir sehen, wie alle Entscheidbarkeitsfragen mithilfe einer (unendlichen) Bitfolge, der sogenannten Haltesequenz, codiert werden können.

8.1 Unentscheidbarkeit des Halteproblems

Die Unentscheidbarkeit des Halteproblems haben wir bereits im Abschnitt 6.1 bewiesen. Ein Beweis kann auch mithilfe der Unberechenbarkeit der Kolmogorov-Komplexität erfolgen. Zu diesem Zweck definieren wir das **Halteproblem** durch die Sprache

$$\mathcal{H} = \{\langle A, w \rangle \in \mathbb{B}^* \mid w \in \mathit{Def}(\Phi_A)\}$$

Satz 8.1 *Die charakteristische Funktion $\chi_{\mathcal{H}}$ der Sprache $\mathcal{H}$ ist nicht berechenbar.*

Beweis *Wir nehmen an, dass $\chi_{\mathcal{H}}$ berechenbar ist. Es sei $H \in \mathcal{M}$ ein Programm, das $\chi_{\mathcal{H}}$ berechnet: $\Phi_H = \chi_{\mathcal{H}}$. Mithilfe von H konstruieren wir das in Abbildung 8.1*

K.-U. Witt und M. E. Müller, *Algorithmische Informationstheorie*,
https://doi.org/10.1007/978-3-662-61694-9_8

```
read(x);
   i := 0;
   gefunden := false;
   while not gefunden do
      A := ν(i);
      if A ∈ M then
         if Φ_H ⟨A, ε⟩ = 1 then
            if Φ_U ⟨A⟩ = x then
               k := |⟨A⟩|;
               gefunden := true
            else i := i + 1
            endif;
         endif;
      endif;
   endwhile;
write(k).
```

Abbildung 8.1: Programm C zur Berechnung der Kolmogorov-Komplexität unter der Voraussetzung, dass das Halteproblem entscheidbar ist

```
read();
write(x).
```

Abbildung 8.2: Naives Programm zur Erzeugung des Wortes x

dargestellte Programm C. Dabei ist $\nu(i) = w$ das i-te Wort in der lexikografischen Anordnung der Wörter von $\mathbb{B}^*$ (siehe Abschnitt 2.6). Das Programm durchläuft die Wörter von $\mathbb{B}^*$ der Größe nach. Es überprüft, ob das aktuelle Wort ein Programm $A \in \mathcal{M}$ darstellt. Falls dies zutrifft, wird überprüft, ob A, angewendet auf das leere Wort, anhält. Im gegebenen Fall wird überprüft, ob A dabei die Eingabe x erzeugt. Falls ja, dann ist die Länge dieses Programms die Kolmogorov-Komplexität von x: $|b^*(x)| = |\langle A \rangle|$. Das Programm terminiert und gibt die Kolmogorov-Komplexität

$$k = |\langle A \rangle| = \mathcal{K}(x)$$

von x aus. In allen anderen Fällen wird das nächste Wort in der lexikografischen Ordnung entsprechend überprüft.

Da alle Wörter von $\mathbb{B}^*$ durchlaufen werden, werden auch alle Programme von $\mathcal{M}$ durchlaufen. Darunter ist auf jeden Fall eines, welches (ohne Eingabe) das eingegebene Wort x erzeugt (mindestens das in Abbildung 8.2 dargestellte Programm). Damit

ist für jede Eingabe die Terminierung des Programms C gesichert. Da die Programme der Größe nach durchlaufen werden, ist auch gesichert, dass C das kürzeste Programm, das x erzeugt, findet.

Damit berechnet das Programm C die Kolmogorov-Komplexität für jedes $x \in \mathbb{B}^*$. Daraus folgt, dass $\mathcal{K}$ eine berechenbare Funktion ist, ein Widerspruch zu Satz 7.11. Unsere eingangs des Beweises gemachte und im Programm C verwendete Annahme, dass $\chi_{\mathcal{H}}$ berechenbar ist, muss also falsch sein. Damit ist die Behauptung des Satzes gezeigt.

Bemerkung 8.1 *Der Beweis von Satz 8.1 gibt quasi eine Reduktion der Berechnung von $\mathcal{K}$ auf die Berechnung von $\chi_{\mathcal{H}}$ an.*

8.2 Die Menge der Primzahlen ist unendlich

Ein bekannter Beweis für die Unendlichkeit der Menge $\mathbb{P}$ der Primzahlen geht auf Euklid zurück und argumentiert wie folgt: Es wird angenommen, dass es nur endliche viele Primzahlen $p_1, \ldots, p_k$, $k \geq 1$, gibt. Damit bildet man die Zahl $n = p_1 \cdot \ldots \cdot p_k + 1$. Es folgt, dass keine der Primzahlen p_i, $1 \leq i \leq k$, ein Teiler von n sein kann. Dies ist ein Widerspruch zu der Tatsache, dass jede Zahl $n \in \mathbb{N}_0$, $n \geq 2$, mindestens einen Primteiler besitzt (so ist z. B. der kleinste Teiler immer eine Primzahl). Die Annahme, dass $\mathbb{P}$ endlich ist, führt also zu einem Widerspruch.

Ein Beweis der Unendlichkeit von $\mathbb{P}$ kann wie folgt mithilfe der Kolmogorov-Komplexität geführt werden. Für die Kolmogorov-Komplexität natürlicher Zahlen (siehe (7.8)) gilt gemäß Folgerung 7.4, dass es unendlich viele natürliche Zahlen $n \in \mathbb{N}_0$ mit

$$\mathcal{K}(n) = \mathcal{K}(dual(n)) \geq |dual(n)| = \lfloor \log n \rfloor + 1 \tag{8.1}$$

gibt. Wir nehmen ein solches n und nehmen an, dass es nur endlich viele Primzahlen $p_1, \ldots, p_k$, $k \geq 1$, gibt. Sei

$$n = p_1^{\alpha_1} \cdot p_2^{\alpha_k} \cdot \ldots \cdot p_k^{\alpha_k} \tag{8.2}$$

die eindeutige Faktorisierung von n; n ist also eindeutig durch die Exponenten α_i, $1 \leq i \leq k$, erzeugbar, d. h., $b(n) = \langle\, \alpha_1, \ldots, \alpha_k \,\rangle$ ist eine Beschreibung von n. Aus (8.2) folgt

$$n \geq 2^{\alpha_1 + \ldots + \alpha_k}$$

und daraus

$$\log n \geq \alpha_1 + \ldots + \alpha_k \geq \alpha_i,\ 1 \leq i \leq k$$

und hieraus

$$|dual(\alpha_i)| \leq \lfloor \log \log n \rfloor + 1,\ 1 \leq i \leq k.$$

Insgesamt folgt nun

$$\begin{aligned}\mathcal{K}(n) = |b^*(n)| \leq |b(n)| &= |\langle\, \alpha_1, \ldots, \alpha_k \,\rangle| \\ &= 2\sum_{i=1}^{k}(|dual(\alpha_i)| + 1) \\ &\leq k \cdot 2(\lfloor \log\log n \rfloor + 2).\end{aligned}$$

Dies ist aber ein Widerspruch zu (8.1), denn für jedes beliebige, aber feste $k \geq 1$ gilt

$$\lfloor \log n \rfloor + 1 > k \cdot 2(\lfloor \log\log n \rfloor + 2)$$

für fast alle n. Damit haben wir auch auf diese Art und Weise die Annahme, dass $\mathbb{P}$ endlich ist, widerlegt.

8.3 Reguläre Sprachen

Wir schränken Turing-Entscheider wie folgt ein:

$$A = (\mathbb{B}, S, \delta, s_0, \, , t_a, t_r)$$

mit

$$\delta : (S - \{t_a, t_r\}) \times \Sigma \to S \times \Sigma \times \{\rightarrow, -\}$$

Das Arbeitsband ist ein reines Eingabeband, das nur von links nach rechts gelesen werden kann, ohne dabei ein Symbol zu verändern. Deshalb benötigen diese Maschinen keine zusätzlichen Hilfssymbole, auch kein Blanksymbol, und der S-/L-Kopf, der nur ein Lesekopf ist, kann stehen bleiben oder nach rechts gehen. Das Stehenbleiben kann durch einen ε-Übergang realisiert werden, d. h., die Maschine liest das Symbol nicht und führt nur einen Zustandswechsel durch. Wir können die Definition solcher Maschinen vereinfachen zu

$$A = (S, \delta, s_0, t_a, t_r)$$

mit

$$\delta : (S - \{t_a, t_r\}) \times (\mathbb{B} \cup \{\varepsilon\}) \to S.$$

Wir nennen Maschinen dieser Art **endliche Automaten**. Die Menge der Konfigurationen ist gegeben durch $K = S \circ \Sigma^*$, die Konfigurationsübergänge $\vdash \subseteq K \times K$ sind definiert durch

$$sav \vdash s'v \;\text{ genau dann, wenn }\; \delta(s, a) = s' \;\text{ mit }\; a \in \Sigma \cup \{\varepsilon\},\; v \in \Sigma^*,$$

und

$$L(A) = \{w \in \Sigma^* \mid s_0 w \vdash^* t_a \varepsilon\}$$

ist die von A akzeptierte Sprache. Eine Sprache heißt **regulär** genau dann, wenn ein endlicher Automat existiert, der sie akzeptiert. Wir bezeichnen mit REG die **Klasse der regulären Sprachen**.

Folgerung 8.1 $\mathsf{REG} \subseteq TIME(n)$.

Beweis *Sei $L \in \mathsf{REG}$. Dann existiert ein endlicher Automat $A = (S, \delta, s_0, t_a, t_r)$ mit $L = L(A)$. Sei $w \in L$ mit $|w| = n \in \mathbb{N}$. Dann existieren Zustände $t_j \in S$, $1 \leq j \leq n+1$, mit $t_1 = s_0$, $t_{n+1} = t_a$ und $t_j w\,[\,j, n\,] \vdash t_{j+1} w\,[\,j+1, n\,]$, $1 \leq j \leq n$. Daraus folgt unmittelbar $time_A(w) = n$ und damit $L \in TIME(n)$, und die Behauptung ist gezeigt.*

Hat ein endlicher Automat A n Zustände und ist $w \in L(A)$ mit $|w| \geq n$, dann wird in der akzeptierenden Konfigurationsfolge $s_0 w \vdash^* t_a \varepsilon$ mindestens ein Zustand s zweimal durchlaufen. w lässt sich deshalb aufteilen in $w = xyz$, sodass

$$s_0 x \vdash syz \vdash sz \vdash t_a \varepsilon \tag{8.3}$$

gilt. Dabei ist $|y| \geq 1$, und xy kann immer so gewählt werden, dass $|xy| < n$ ist. Wenn nämlich $|xy| \geq n$ ist, kann bereits xy in drei Teile mit den oben genannten Eigenschaften aufgeteilt werden.

Aus (8.3) folgt, dass auch folgende akzeptierende Konfiguratinsfolgen existieren

$$\begin{aligned} &s_0 x \vdash sz \vdash sz \vdash t_a \varepsilon, \\ &s_0 x \vdash syz \vdash syz \vdash sz \vdash t_a \varepsilon, \\ &s_0 x \vdash syz \vdash syz \vdash syz \vdash sz \vdash t_a \varepsilon, \\ &\quad\vdots \end{aligned}$$

weil der Zyklus $syz \vdash sz$ beliebig oft durchlaufen werden kann.

Aus den obigen Überlegungen folgt das sogenannte **Pumping-Lemma** für reguläre Sprachen.

Satz 8.2 *Sei $L \in \mathsf{REG}$, dann existiert eine Zahl $n \in \mathbb{N}$, sodass sich alle Wörter $w \in L$ mit $|w| \geq n$ aufteilen lassen in $w = xyz$ und für eine solche Aufteilung die folgenden drei Eigenschaften gelten:*

(i) $|y| \geq 1$,

(ii) $|xy| < n$ *und*

(iii) $xy^i z \in L$ *für alle* $i \in \mathbb{N}_0$.

Das Pumping-Lemma gibt eine notwendige, keine hinreichende Bedingung für die Regularität einer Sprache an, denn man kann nicht reguläre Sprachen angeben, die das Pumping-Lemma erfüllen. Man kann das Lemma also nur im „negativen Sinne“

anwenden, d. h. um zu zeigen, dass eine Sprache nicht regulär ist. Wenn man vermutet, dass eine Sprache nicht regulär ist, nimmt man an, dass sie es doch sei. Sie muss dann das Pumping-Lemma erfüllen. Wenn man dann einen Widerspruch gegen eine der Bedingungen des Lemmas herleiten kann, weiß man, dass die Annahme falsch ist.

Beispiel 8.1 *Die Sprache* $L = \{0^l 1^l \mid l > 0\}$ *ist nicht regulär. Zum Beweis nehmen wir an, dass* L *regulär ist. Dann muss gemäß Pumping-Lemma eine Zahl* $n \in \mathbb{N}$ *existieren, sodass die Bedingungen des Lemmas erfüllt sind. Wir wählen das Wort* $w = 0^n 1^n \in L$*. Es ist* $|w| = 2n > n$*. Also kann* w *zerlegt werden in* $w = xyz$*, sodass die Bedingungen (i) – (iii) des Pumping-Lemmas erfüllt sind. Da* $|xy| < n$ *ist, ist* xy *ein Präfix von* 0^n*. Die Aufteilung von* w *hat deshalb Gestalt*

$$w = \underbrace{0^p}_{x} \underbrace{0^q}_{y} \underbrace{0^r 1^n}_{z}.$$

Dabei ist

(0) $p + q + r = n$*,*

(1) $q \geq 1$*,*

(2) $p + q < n$*,*

(3) $xy^i z \in L$ *für alle* $i \in \mathbb{N}_0$*.*

Wir wählen $i = 0$*, dann muss* $xy^0 z = xz \in L$ *sein. Es gilt aber*

$$xy^0 z = 0^p (0^q)^0 0^r 1^n = 0^p 0^r 1^n = 0^{p+r} 1^n.$$

Wegen der Bedingungen (0) und (1) ist

$$p + r < p + q + r = n,$$

woraus folgt, dass $0^{p+r} 1^n \notin L$ *ist, ein Widerspruch zu (3). Damit ist die Annahme, dass* L *regulär ist, falsch.*

Die Automaten heißen endlich, weil sie ein endliches Gedächtnis haben, nämlich nur eine endliche Anzahl von Zuständen, mit denen sie sich den Stand der Abarbeitung einer Eingabe merken können. Deshalb sind sie nicht in der Lage, Sprachen wie die im Beispiel zu entscheiden. Dazu benötigt ein Automat einen beliebig großen Speicher, um die Präfixe 0^l, $l > 1$, zu speichern, sodass dann die anschließend folgenden Einsen damit abgeglichen werden können. Wenn die Anzahl der Speicherplätze zu gering ist, führt das unweigerlich dazu, dass Zustände mehrfach durchlaufen werden können, wodurch „aufgepumpte“ Wörter akzeptiert werden, die nicht zu der zur Entscheidung anstehenden Sprache gehören. Mit einem unendlichen, veränderbaren Speicher, wie sie Turingmaschinen mit dem Arbeitsband zur Verfügung stehen, kann der Abgleich

von beliebig langen Wortteilen erfolgen. In Beispiel 3.1 haben wir eine Turingmaschine konstruiert, welche die Sprache $L = \left\{0^l 1^l \mid l > 0\right\}$ akzeptiert.

Folgerung 8.2 $\mathsf{REG} \subset \mathsf{R}$.

Das Pumping-Lemma sowie Beispiele zu seiner Anwendung gehören zur „Folklore" der Theoretischen Informatik. Wir demonstrieren im Folgenden an zwei Beispielen, wie mithilfe der Kolmogorov-Komplexität gezeigt werden kann, dass eine Sprache nicht regulär ist.

Als Erstes betrachten wir die Sprache $L = \left\{0^l 1^l \mid l > 0\right\}$ aus dem obigen Beispiel und nehmen wieder an, dass L regulär ist, dass also ein endlicher deterministischer Automat A mit der Zustandsmenge S und dem akzeptierenden Zustand t_a existiert, der L akzeptiert. Zu jedem k gibt es (eindeutig) einen Zustand $s_k \neq t_a$, der nach Abarbeitung von 0^k erreicht wird; von dort gelangt der Automat nach Abarbeitung von 1^k (erstmals) in den Zustand t_a. Der Automat A und der Zustand s_k können als eine Beschreibung von k angesehen werden, denn wir können uns ein Programm P überlegen, das A als („fest verdrahteten") Bestandteil enthält, und das, wenn ihm s_k übergeben wird, eine Folge von Einsen produziert und sich damit auf den Weg zum Endzustand t_a macht. Dieser Zustand wird nach genau k Einsen erreicht, womit P die Zahl k ausgeben kann.

Da A die Wörter $0^k 1^k$ für alle $k \geq 1$ akzeptiert, muss für jedes k ein (Nicht-End-) Zustand s_k existieren. Das heißt, wir können dem Programm P alle möglichen (Nicht-End-) Zustände s übergeben. Das sind endlich viele Eingaben, und P geht für jedes s jeweils wie oben beschrieben vor und kann so alle $k \in \mathbb{N}$ erzeugen.

Das bedeutet, dass wir eine geeignete Codierung $\langle P, s_k \rangle$ von P (inklusive des „fest verdrahteten" Automaten A) und s_k als Beschreibung der Zahl k ansehen können: $b(k) = \langle P, s_k \rangle$. Ihre Länge $|b(k)|$ ist unabhängig von k, also konstant. Damit gilt $\mathcal{K}(k) \leq |b(k)| = |\langle P, s_k \rangle|$. Sei nun

$$c = \max \{|\langle P, s \rangle| : s \in S - \{t_a, t_r\}\} .$$

Dann gilt

$$\mathcal{K}(k) \leq c \tag{8.4}$$

für alle k.

Aus (7.8) und Folgerung 7.4 folgt, dass es Zahlen $n \in \mathbb{N}$ mit

$$\mathcal{K}(n) = \mathcal{K}(dual(n)) \geq |dual(n)| = \lfloor \log n \rfloor + 1$$

gibt. Sei $n_c \in \mathbb{N}$ eine solche Zahl, d. h. eine Zahl mit

$$\mathcal{K}(n_c) \geq \lfloor \log n_c \rfloor + 1 > c. \tag{8.5}$$

Wenn wir nun das Wort $0^{n_c}1^{n_c}$, also $k = n_c$, wählen, folgt mit (8.4) $\mathcal{K}(n_c) \leq c$, was offensichtlich ein Widerspruch zu (8.5) ist. Damit ist unsere Annahme, dass L regulär ist, widerlegt.

Für das zweite Beispiel verwenden wir das sogenannte **Lemma der KC-Regularität.**

Lemma 8.1 *Es sei $L \subseteq \mathbb{B}^*$ eine reguläre Sprache. Für die Wörter $x \in \mathbb{B}^*$ sei $L_x = \{y \mid xy \in L\}$. Des Weiteren sei $f : \mathbb{N}_0 \to \mathbb{B}^*$ eine rekursive Aufzählung von L_x (siehe Abschnitt 3.4). Dann existiert eine nur von L und f abhängende Konstante c, sodass für jedes x und $y \in L_x$ mit $f(n) = y$ gilt: $\mathcal{K}(y) \leq \mathcal{K}(n) + c$.*

Beweis *Sei A der endliche deterministische Automat, der L akzeptiert, und s der Zustand, den A nach Abarbeitung von x erreicht. Die Zeichenkette y mit $xy \in L$ und $f(n) = y$ kann nun mithilfe von A, s und dem Programm P_f, das f berechnet, bestimmt werden. Geeignete Codierungen von A, s, P_f und n können somit als Beschreibung von y angesehen werden. Dabei sind die Codierungen von A, s und P_f unabhängig von y. Ihre Gesamtlänge fassen wir unter der Konstanten c zusammen. Damit folgt dann $\mathcal{K}(y) \leq \mathcal{K}(n) + c$.*

Wir wenden das Lemma an, um zu zeigen, dass die Sprache $L = \{1^p \mid p \in \mathbb{P}\}$ nicht regulär ist. Wir nehmen an, dass L regulär ist, und betrachten das Wort $xy = 1^p$ mit $x = 1^{p'}$, wobei p die $k+1$-te Primzahl und p' die k-te Primzahl ist. Des Weiteren sei f die lexikografische Aufzählung von L_x. Dann gilt $f(1) = y = 1^{p-p'}$. Mit dem Lemma folgt dann: $\mathcal{K}(y) \leq \mathcal{K}(1) + \mathsf{O}(1) = \mathsf{O}(1)$. Das bedeutet, dass die Kolmolgorov-Komplexität der Differenz zwischen allen benachbarten Primzahlen jeweils durch eine von diesen unabhängige Konstante beschränkt ist. Das ist ein Widerspruch zur Tatsache, dass zu jeder Differenz $d \in \mathbb{N}$ zwei benachbarte Primzahlen gefunden werden können, die mindestens den Abstand d haben.

8.4 Unvollständigkeit formaler Systeme

Kurt Gödel (siehe Fußnote Seite 86) hat gezeigt, dass die Konsistenz eines hinreichend mächtigen formalen Systems (z. B. ein die Arithmetik natürlicher Zahlen enthaltendes System) nicht innerhalb dieses Systems bewiesen werden kann. Auch diese Unvollständigkeit formaler Systeme, die üblicherweise mit einem Diagonalisierungsbeweis gezeigt wird, kann mithilfe der Kolmogorov-Komplexität gezeigt werden.

Satz 8.3 *Sei $\mathcal{F}$ ein formales System, das folgende Eigenschaften erfüllt:*

(1) Falls es einen korrekten Beweis für eine Aussage α gibt, dann ist diese Aussage auch wahr.

(2) Für eine Aussage α und einen Beweis p kann algorithmisch entschieden werden, ob p ein Beweis für α ist.

```
read(n);
   k := 1;
   while true do
      for all x ∈ B* with |x| ≤ k do
         for all p ∈ B* with |p| ≤ k do
            if p ein korrekter Beweis für α(x,n) ist
               then return x
            endif
         endfor
      endfor;
      k := k + 1
   endwhile;
```

Abbildung 8.3: Programm A, das Beweise für die Aussagen $\alpha(x, n)$ findet.

(3) Für jedes $x \in \mathbb{B}^*$ und jedes $n \in \mathbb{N}_0$ kann eine Aussage $\alpha(n, x)$ formuliert werden, die äquivalent zur Aussage „$\mathcal{K}(x) \geq n$“ ist.

Dann existiert ein $t \in \mathbb{N}_0$, sodass alle Aussagen $\alpha(x, n)$ für $n > t$ in $\mathcal{F}$ nicht beweisbar sind.

Beweis *Wir betrachten zunächst das Programm A in Abbildung 8.3: Falls es für eine Eingabe n ein x gibt, für das die Aussage $\alpha(x, n)$ beweisbar ist, dann findet A ein solches x. Wir nehmen nun an, dass für alle n ein x existiert, sodass $\alpha(x, n)$ beweisbar ist. Dann findet das Programm A ein solches x. Die Beweisbarkeit von $\alpha(x, n)$ bedeutet, dass $\mathcal{K}(x) \geq n$ ist. Des Weiteren ist $b(x) = \langle A, n \rangle$ eine Beschreibung von x, denn das Programm A erzeugt x bei Eingabe von n. Es folgt mit $c = 2\,|\langle A \rangle| + 3$*

$$\mathcal{K}(x) = |b^*(x)| \leq |b(x)| = |\langle A, n \rangle| = 2(|\langle A \rangle| + 1) + |\mathit{dual}(n)| = \lfloor \log n \rfloor + c.$$

Insgesamt erhalten wir also wieder die Ungleichungen $n \leq \mathcal{K}(x) \leq \lfloor \log n \rfloor + c$, die nur für endlich viele und insbesondere nicht für hinreichend große n erfüllt sind. Damit erhalten wir einen Widerspruch, womit unsere Annahme widerlegt ist.

Bemerkung 8.2 *Der obige Satz zeigt uns eine Möglichkeit auf, in einem hinreichend mächtigen formalen System $\mathcal{F}$, das die Bedingungen (1)–(3) des Satzes erfüllt, Aussagen zu generieren, die wahr aber innerhalb des Systems nicht beweisbar sind. Wir benutzen das Programm A, das die von n und x unabhängige konstante Länge c hat. Für $n \in \mathbb{N}_0$ mit $n > \lfloor \log n \rfloor + c$ ist keine Aussage der Art $\mathcal{K}(x) \geq n$ beweisbar. Wählen wir zufällig eine Bitfolge x der Länge $n+20$, dann gilt die Aussage $\mathcal{K}(x) \geq x$ mit einer Wahrscheinlichkeit $\geq 1 - 2^{-20}$, aber diese Aussage ist in $\mathcal{F}$ nicht beweisbar.*

8.5 Die Kolmogorov-Komplexität entscheidbarer Sprachen

Der folgende Satz besagt, dass entscheidbare Sprachen nur eine (sehr) geringe Kolmogorov-Komplexität besitzen.

Satz 8.4 *Sei $L \subseteq \mathbb{B}^*$ eine entscheidbare Sprache. Des Weiteren seien die Wörter von L lexikografisch angeordnet, und x_n sei das n-te Wort in dieser Anordnung. Dann gilt*

$$\mathcal{K}(x_n) \leq \lceil \log n \rceil + \mathsf{O}(1).$$

Beweis *Da L entscheidbar ist, ist χ_L berechenbar. Es sei $C_{\chi_L} \in \mathcal{M}$ das Programm, das χ_L berechnet, und suc sei die total berechenbare Funktion, die zu einem Wort in der lexikografischen Anordnung der Wörter von $\mathbb{B}^*$ den Nachfolger bestimmt. Die in Abbildung 8.4 dargestellten Programme X_n generieren jeweils das n-te Wort $x_n \in L$. Wie in anderen Beispielen in den vorherigen Abschnitten sind alle Programme bis auf die Zahl n identisch, unabhängig von n hat der identische Anteil eine konstante Länge. Hieraus folgt unmittelbar die Behauptung.*

```
read();
   i := 0;
   z := ε;
   while i ≤ n do
      if Φ_{C_{χ_L}}(z) = 1 then
         x_n := z;
         i := i + 1;
      endif;
      z := suc(z);
   endwhile;
write(x_n).
```

Abbildung 8.4: Programm X_n zur Generierung des n-ten Wortes der entscheidbaren Sprache L

Die Wörter entscheidbarer Sprachen haben also eine geringe Kolmogorov-Komplexität. Diese Aussage bestätigt die Aussage von Satz 6.4 b), dass entscheidbare Sprachen „besonders einfach“ sind.

Satz 8.5 *Die Sprache der zufälligen, d. h. nicht komprimierbaren, Bitfolgen*

$$L = \{x \in \mathbb{B}^* \mid \mathcal{K}(x) \geq |x|\}$$

ist nicht entscheidbar.

Beweis *Wir nehmen an, dass L entscheidbar ist. Dann ist χ_L berechenbar. Es sei $C_{\chi_L} \in \mathcal{M}$ das Programm, das χ_L berechnet. Sei $n \in \mathbb{N}_0$ beliebig, aber fest gewählt. In Abbildung 8.5 ist das Programm A_n dargestellt, das die erste zufällige Bitfolge x in der lexikografischen Anordnung von $\mathbb{B}^*$ mit $|x| > n$ berechnet (wegen Satz 7.8 existiert eine solche Folge). suc sei wieder die total berechenbare Funktion, die zu jeder Folge x ihren Nachfolger in der lexikografischen Anordnung von $\mathbb{B}^*$ bestimmt. Für die Ausgabe x des Programms A_n gilt*

$$|x| > n \text{ und } \mathcal{K}(x) \leq \lceil \log n \rceil + c. \tag{8.6}$$

Da x zufällig ist, gilt

$$\mathcal{K}(x) \geq |x|. \tag{8.7}$$

Aus (8.6) und (8.7) erhalten wir

$$n < |x| \leq \mathcal{K}(x) \leq \lceil \log n \rceil + c,$$

was für hinreichend große n offensichtlich einen Widerspruch darstellt.

```
read();
   gefunden := false;
   x := ε;
   while not gefunden do
      if |x| > n and Φ_{C_{χL}}(x) = 1 then
         gefunden := true;
      else x := suc(x);
      endif;
   endwhile;
write(x).
```

Abbildung 8.5: Programm A_n zur Berechnung der ersten zufälligen Bitfolge x mit $|x| > n$

8.6 Die Chaitin-Konstante

Wir übertragen den Begriff *zufällig* auf unendliche Bitfolgen, d. h., wir betrachten jetzt die Elemente von $\mathbb{B}^\omega$.

Definition 8.1 *Eine Bitfolge $x \in \mathbb{B}^\omega$ heißt* **zufällig** *genau dann, wenn eine Konstante $c \in \mathbb{N}$ existiert, sodass*

$$\mathcal{K}(x[1,n]) > n - c$$

für alle $n \in \mathbb{N}$ gilt.

Den folgenden Betrachtungen legen wir wieder eine Gödelisierung $(\mathbb{N}_0, \mathcal{P}, \varphi)$ einer Programmiersprache $\mathcal{M}$ zugrunde.

Definition 8.2 *Die Bitfolge $H = h_1 h_2 h_3 \ldots \in \mathbb{B}^\omega$, definiert durch*

$$h_i = \begin{cases} 1, & \varepsilon \in \mathit{Def}(\varphi_i), \\ 0, & \mathit{sonst}, \end{cases}$$

heißt **Haltesequenz**.

Die Haltesequenz H ist also eine unendliche Bitfolge, deren i-tes Bit gleich 1 ist, wenn das i-te Programm (bei Eingabe des leeren Wortes) anhält, ansonsten ist das Bit gleich 0. Es ist klar, dass H von der Gödelisierung abhängt. Wir halten fest, dass aus der Nummer i das Programm $\nu(i) = A$ zwar rekonstruiert, aber die Sequenz H wegen der Unentscheidbarkeit des Halteproblems nicht (vollständig) berechnet werden kann.

Bemerkung 8.3 **a)** *Die Sequenz $H[1,k]$ gibt an,*

(1) wie viele der ersten k Progamme anhalten und

(2) welche Programme das sind.

Dabei könnten wir die Information (2) aus der Information (1) gewinnen: Sei $l \leq k$ die Anzahl der terminierenden Programme unter den k ersten. Wir lassen alle k Programme parallel laufen, markieren die haltenden Programme und zählen diese. Falls l Programme gestoppt haben, können wir alle anderen Programme auch anhalten.

b) *Wenn wir wüssten, wie groß die Wahrscheinlichkeit ist, mit der sich unter den k ersten Gödelnummern solche von terminierenden Programmen befinden, dann könnten*

wir die Anzahl l der terminierenden Programme durch Multiplikation mit k bestimmen, denn es gilt

$$Prob\,[\,H(i) = 1\,] = \frac{l}{k}$$

woraus sich bei gegebenem k und bekannter Wahrscheinlichkeit $Prob\,[\,H(i) = 1\,]$

$$l = k \cdot Prob\,[\,H(i) = 1\,]$$

berechnen lässt.

c) *Im Folgenden geht es im Wesentlichen darum, wie $Prob\,[\,H(i) = 1\,]$ bestimmt werden kann. Daraus könnten l und damit gemäß a) die terminierenden Programme A_i, $1 \leq i \leq k$, bestimmt werden.*

Hilfreich dafür ist die nachfolgende Definition.

Definition 8.3 *Es sei $x \in \mathbb{B}^n$. Dann heißt*

$\Omega_n = Prob[x$ beginnt mit der Gödelnummer eines terminierenden Programms$]$

Haltewahrscheinlichkeit.

Beispiel 8.2 *Wir verdeutlichen die Definition anhand des folgenden fiktiven Beispiels. Sei etwa*

$$H = 00100000110000001010\ldots .$$

Es ist also $h_i = 1$ für $i \in \{3, 9, 10, 17, 19\}$. Die entsprechenden Gödelnummern seien $a_3 = 10$, $a_9 = 11$, $a_{10} = 101$, $a_{17} = 0101$, $a_{19} = 1010$. Es ergeben sich folgende Haltewahrscheinlichkeiten:

1. $\Omega_1 = 0$, *denn es gibt kein Programm mit einstelliger Gödelnummer.*
2. $\Omega_2 = \frac{1}{2}$, *denn $10 \in \mathbb{B}^2$ ist die Gödelnummer $a_3 = 10$, und $11 \in \mathbb{B}^2$ ist die Gödelnummer $a_9 = 11$*
3. $\Omega_3 = \frac{5}{8}$, *denn 100 und 101 beginnen mit der Gödelnummer $a_3 = 10$; und 110 und 111 beginnen mit der Gödelnummer $a_9 = 11$.*
4. $\Omega_4 = \frac{12}{16}$, *denn $1000, 1001, 1010, 1011 \in \mathbb{B}^4$ beginnen mit der Gödelnummer $a_3 = 10$; $1100, 1101, 1110, 1111 \in \mathbb{B}^4$ beginnen mit der Gödelnummer $a_9 = 11$; $1010, 1011 \in \mathbb{B}^4$ beginnen mit der Gödelnummer $a_{10} = 101$; und $0101, 1010 \in \mathbb{B}^4$ sind die Gödelnummern $a_{17} = 0101$ bzw. $a_{19} = 1010$.*

Wir können uns vorstellen, eine Gödelnummer durch Münzwurf zu generieren. Sobald die so erzeugte Bitfolge eine Gödelnummer ist, können wir aufhören zu würfeln, denn wegen der Präfixfreiheit kann keine weitere Gödelnummer entstehen. Es kann vorkommen, dass keine Gödelnummer gewürfelt wird (das Würfeln stoppt nicht).

Im Beispiel 8.2 werden die Programme mit den Gödelnummern a_3 und a_9 jeweils mit der Wahrscheinlichkeit $\frac{1}{4}$ erzeugt, das Programm a_9 mit der Wahrscheinlichkeit $\frac{1}{8}$ und die Programme a_{17} und a_{19} jeweils mit der Wahrscheinlichkeit $\frac{1}{16}$. Daraus ergibt sich

$$\begin{aligned}
\Omega_1 &= 0 \\
\Omega_2 &= \frac{1}{4} + \frac{1}{4} = \frac{1}{2} \\
\Omega_3 &= \frac{1}{4} + \frac{1}{4} + \frac{1}{8} = \frac{5}{8} \\
\Omega_4 &= \frac{1}{4} + \frac{1}{4} + \frac{1}{8} + \frac{1}{16} + \frac{1}{16} = \frac{12}{16}.
\end{aligned}$$

Es gilt

$$\Omega_n = \sum_{\substack{A \text{ terminiert} \\ |\langle A \rangle| \leq n}} 2^{-|\langle A \rangle|}. \tag{8.8}$$

Bemerkung 8.4 *In der binären Darstellung von Ω_n sind die Antworten zu allen mathematischen Fragestellungen codiert, die sich über die Terminierungseigenschaft eines Programms der Länge kleiner oder gleich n berechnen lassen.*

Betrachten wir als Beispiel die Goldbachsche Vermutung: „Jede gerade natürliche Zahl größer zwei lässt sich als Summe zweier Primzahlen darstellen.“ Diese Vermutung ist für eine große Anzahl von Fällen bestätigt, aber nicht endgültig bewiesen. Wir können uns ein Programm $G \in \mathcal{M}$ überlegen, das die erste Zahl größer zwei sucht, die sich nicht als Summe zweier Primzahlen darstellen lässt. Sei $|\langle G \rangle| = n$. Die Anzahl der Gödelnummern von Programmen mit einer Länge kleiner oder gleich n lässt sich bestimmen. Wenn wir Ω_n kennen würden, dann könnten wir mit diesen beiden Angaben die Anzahl der Bitfolgen bestimmen, die mit der Gödelnummer eines terminierenden Programms beginnen. Aus der Bemerkung 8.3 folgt, dass wir mithilfe dieser Kenntnisse herausfinden können, ob G terminiert oder nicht. Falls G terminiert, wäre die Goldbachsche Vermutung falsch, falls G nicht terminiert, wäre sie wahr.

Dieses Beispiel verdeutlicht die in Bemerkung 8.3 c) geäußerte Bedeutung der Haltesequenz Ω_n für die Lösung von Entscheidungsfragen, insbesondere auch von derzeit offenen Entscheidungsfragen.

Folgerung 8.3 **a)** *Die Folge $\{\Omega_n\}_{n\geq 1}$ wächst monoton, d. h., es ist $\Omega_n \leq \Omega_{n+1}$.*

b) *Die Folge $\{\Omega_n\}_{n\geq 1}$ ist nach oben beschränkt, denn es gilt $\Omega_n \leq 1$ für alle $n \in \mathbb{N}$.*

c) *Die Folge* $\{\Omega_n\}_{n\geq 1}$ *ist konvergent.*

Beweis **a)** *Die Folgerung ist offensichtlich.*

b) *Es gilt*

$$\Omega_n = \sum_{\substack{A \text{ terminiert} \\ |\langle A \rangle| \leq n}} 2^{-|\langle A \rangle|} \leq \sum_{i=1}^{n} \left(\frac{1}{2}\right)^i = \frac{1 - \left(\frac{1}{2}\right)^{n+1}}{1 - \frac{1}{2}} - 1 = 1 - \left(\frac{1}{2}\right)^n < 1.$$

c) *Die Aussage folgt unmittelbar aus a) und b).*

Definition 8.4 *Der Grenzwert*

$$\Omega = \lim_{n \to \infty} \Omega_n = \sum_{A \text{ terminiert}} 2^{-|\langle A \rangle|}$$

heißt **Chaitin-Konstante**.

Wir wollen uns mit der Frage befassen, ob sich Ω bestimmen lässt. Dazu variieren wir die Definition 8.3 wie folgt.

Definition 8.5 *Es sei* $x \in \mathbb{B}^n$. *Dann heißt*

Ω_n^k =*Prob*[x beginnt mit der Gödelnummer eines terminierenden Programms, das in k Schritten stoppt]

Haltewahrscheinlichkeit *(der in* k *Schritten terminierenden Programme).*

Offensichtlich gilt Folgendes.

Folgerung 8.4 **a)** $\Omega_n^n \leq \Omega_n \leq \Omega$.

b) $\lim_{n\to\infty} \Omega_n^n = \lim_{n\to\infty} \Omega_n = \Omega$.

Trotz dieser gleichen Verhaltensweisen sind die Folgen $\{\Omega_n\}_{n\geq 1}$ und $\{\Omega_n^n\}_{n\geq 1}$ wesentlich verschieden: Im Gegensatz zur ersten Folge sind alle Glieder der zweiten Folge berechenbar. Die Haltewahrscheinlichkeit Ω_n^n kann bestimmt werden, indem jedes Programm A mit $|\langle A \rangle| \leq n$ maximal n Schritte ausgeführt wird und dabei gezählt wird, wie viele terminieren. Je größer n gewählt wird, umso genauer kann so der Grenzwert Ω bestimmt werden: Die Nachkommabits von Ω_n^n müssen sich bei

wachsendem n von links nach rechts stabilisieren. Allerdings weiß man nie, wann ein Bit sich nicht mehr ändert – die Änderung weit rechts stehender Bits kann durch Überträge Einfluss auf links stehende Bits haben.

```
read(Ω[1,n]);
   t := 1;
   while Ω_t^t[1,n] ≠ Ω[1,n] do
      t := t + 1;
   endwhile;
 // Jedes terminierende Programm A mit |⟨A⟩| ≤ n
 // hält in höchstens t Schritten;
   A_n^t := Menge aller Bitfolgen
         mit einer Länge ≤ n,
         die mit der Gödelnummer
         eines Programms beginnen,
         das in höchstens t Schritten stoppt;

   Ω_n := |A_n^t| / 2^n;
write(Ω_n).
```

Abbildung 8.6: Verfahren zur Konstruktion von Ω_n aus $\Omega[n]$

Satz 8.6 *Die Haltewahrscheinlichkeit Ω_n kann aus $\Omega[1,n]$, d. h. aus den ersten n Bits der Chaitin-Konstante, bestimmt werden.*

Beweis *Abbildung 8.6 stellt ein Verfahren dar, mit dem Ω_n aus $\Omega[1,n]$ konstruiert werden kann. Wir bestimmen $\Omega_1^1, \Omega_2^2, \Omega_3^3, \ldots$ Die Werte der Folgenglieder nähern sich immer mehr an Ω an. Irgendwann wird ein t erreicht, sodass Ω_t^t und Ω in den ersten n Bits übereinstimmen.*

Für $s > t$ können sich die ersten n Bits von Ω_s^s nicht mehr ändern, denn dann wäre $\Omega_s^s > \Omega$, was ein Widerspruch zur Folgerung 8.4 a) ist. Da jedes Programm A mit $\frac{1}{2^{|\langle A\rangle|}}$ in die Berechnung von Ω_s^s eingeht und sich die ersten n Bits nicht mehr ändern können, muss jedes Programm, das nach mehr als t Schritten anhält, eine Länge größer n haben.

Es folgt, dass es kein Programm A mit $|\langle A\rangle| \leq n$ geben kann, das nach mehr als t Schritten anhält. Deshalb können wir Ω_n wie folgt bestimmen: Es werden alle Programme A mit $|\langle A\rangle| \leq n$ t Schritte ausgeführt. Terminiert ein Programm A innerhalb dieser Schrittzahl, dann geht $\frac{1}{2^{|\langle A\rangle|}}$ in die Berechnung von Ω_n ein. Hält ein Programm nicht innerhalb dieser Schrittzahl, dann terminiert es nie, und $\frac{1}{2^{|\langle A\rangle|}}$ wird nicht berücksichtigt.

Bemerkung 8.5 **a)** *Das in Abbildung 8.6 dargestellte Verfahren ist nicht berechenbar. Seine Berechenbarkeit würde voraussetzen, dass das kleinste t, für das $\Omega_t^t[1,n] = \Omega[1,n]$ gilt, effektiv berechnet werden kann, d. h., dass die Funktion*

$$f(n) = \min\left\{t \mid \Omega_t^t[1,n] = \Omega[1,n]\right\}$$

berechenbar ist. Man kann zeigen, dass f schneller wächst als jede berechenbare Funktion, woraus folgt, dass f nicht berechenbar ist.

b) *Die Chaitin-Konstante Ω enthält das Wissen über alle Haltewahrscheinlichkeiten. Diese Konstante wird in einigen Literaturstellen auch „Chaitins Zufallszahl der Weisheit“ genannt. In ihr versteckt sind die Antworten auf alle mathematischen Entscheidungsfragen, wie die Entscheidbarkeit des Halteproblems für alle möglichen Programme. Allerdings ist Ω nicht berechenbar, d. h., wir sind nicht in der Lage, das in Ω verborgene Wissen zu erkennen.*

c) *Die Haltesequenz H lässt sich komprimieren. Die ersten 2^n Bits von H lassen sich aus Ω_n bestimmen, und Ω_n lässt sich aus $\Omega[1,n]$ rekonstruieren. Damit kann jedes Anfangsstück der Länge m von H mit einem Programm generiert werden, dessen Länge logarithmisch in m wächst. Somit ist H keine Zufallszahl.*

d) *Ω ist eine rekursiv aufzählbare Zahl.*

```
read();
   Berechne M aus Ω[1,n];
   i := 1;
   x := ν(i);
   while x ∈ M do
      i := i + 1;
      x := ν(i);
   endwhile;
write(x).
```

Abbildung 8.7: Das Programm A_n gibt die erste Bitfolge in der lexikografischen Anordnung von $\mathbb{B}^*$ aus, die von keinem terminierenden Programm mit einer Länge kleiner oder gleich n ausgegeben wird.

Satz 8.7 *Die Chaitin-Konstante ist zufällig.*

Beweis *Gemäß Satz 8.6 können wir aus den ersten n Bits von Ω die Menge T_n aller terminierenden Programme mit einer Länge kleiner gleich n bestimmen. Wir führen diese aus und sammeln deren Ausgaben in der Menge M.*

Nun betrachten wir das in Abbildung 8.7 dargestellte Programm A_n. Dabei sei $\nu(i)$ wieder die total berechenbare Funktion, welche die i-te Bitfolge in der lexikografischen Anordnung von $\mathbb{B}^*$ bestimmt.

Das Programm gibt die erste Bitfolge x in der lexikografischen Anordnung von $\mathbb{B}^*$ aus, die von keinem Programm in T_n erzeugt wird. Es gilt

$$n < \mathcal{K}(x) \leq |\langle A_n \rangle|. \tag{8.9}$$

Abgesehen von der Berechnung von $\Omega[n]$ ist die Länge des Programms A_n fest. Wir setzen

$$c = |\langle A_n \rangle| - \mathcal{K}(\Omega[n]).$$

Dieses eingesetzt in (8.9) liefert

$$n < \mathcal{K}(x) \leq c + \mathcal{K}(\Omega[n]),$$

woraus

$$\mathcal{K}(\Omega[n]) > n - c$$

folgt, womit die Behauptung gezeigt ist.

Die Chaitin-Konstante ist also nicht komprimierbar. Das Wissen über alle Haltewahrscheinlichkeiten Ω_n kann also im Wesentlichen nicht kürzer angegeben werden als durch die Chaitin-Konstante selbst.

8.7 Praktische Anwendungen

Dieses Buch ist eine Einführung in die Algorithmische Informationstheorie, die lange Zeit ein Spezialthema gewesen ist, mit dem sich nur wenige Experten beschäftigt haben und das bis auf ganz wenige Ausnahmen keine Rolle in mathematischen oder informatischen Studiengängen gespielt hat. Seit einigen Jahren ist eine starke Belebung zu beobachten, was sich unter anderem durch die Veröffentlichung von Lehrbüchern widerspiegelt, die dieses Thema mehr oder weniger ausführlich behandeln und in Zusammenhang mit anderen Themen setzen. Des Weiteren ist zu beobachten, dass Konzepte und Methoden der Theorie Eingang in praktische Anwendungen finden. Dabei spielt oft eine Variante der Kolmogorov-Komplexität, die sogenannte **bedingte Kolmogorov-Komplexität**, eine Rolle, die wir in dieser Ausarbeitung nicht betrachtet haben:

$$\mathcal{K}(x|y) = \min\{|A| : A \in \mathcal{M} \text{ und } \Phi_U\langle A, y \rangle = x\}$$

ist die minimale Länge eines Programms, das x aus y berechnet; zum Erzeugen der Bitfolge x können die Programme $A \in \mathcal{M}$ die Hilfsinformation y benutzen.[1] Offensichtlich ist $\mathcal{K}(x) = \mathcal{K}(x|\varepsilon)$. Des Weiteren ist einsichtig, dass $\mathcal{K}(x|y) \leq \mathcal{K}(x) + \mathsf{O}(1)$

[1] Falls kein A existiert mit $\Phi_U\langle A, y \rangle = x$, dann setzen wir $\mathcal{K}(x|y) = \infty$.

ist, und je „ähnlicher“ sich x und y sind, d. h., je einfacher ein Programm ist, das x aus y berechnet, umso mehr werden sich $\mathcal{K}(x)$ und $\mathcal{K}(x|y)$ voneinander unterscheiden. So misst $I(y:x) = \mathcal{K}(x) - \mathcal{K}(x|y)$ die Information, die y von x enthält; $I(y:x)$ ist quasi die Bedeutung, die y für x hat.

Als **Informationsunterschied** zwischen x und y kann man etwa

$$E(x,y) = \min\{|A| : \Phi_U\langle A, x\rangle = y \text{ und } \Phi_U\langle A, y\rangle = x\}$$

festlegen. Man kann dann zeigen, dass $E(x,y) \approx \max\{\mathcal{K}(x|y), \mathcal{K}(y|x)\}$ gilt, d. h. genauer, dass

$$E(x,y) = \max\{\mathcal{K}(x|y), \mathcal{K}(y|x)\} + \mathsf{O}(\log\max\{\mathcal{K}(x|y), \mathcal{K}(y|x\})$$

gilt. Man kann zeigen, dass die Abstandsfunktion E in einem gewissen Sinne minimal ist. Das heißt, dass für alle anderen möglichen Abstandsfunktionen D für alle $x, y \in \mathbb{B}^*$ gilt: $E(x,y) \leq D(x,y)$ (abgesehen von einer additiven Konstante). Durch die Normalisierung von E durch

$$e(x,y) = \frac{\max\{\mathcal{K}(x|y), \mathcal{K}(y|x)\}}{\max\{\mathcal{K}(x), \mathcal{K}(y)\}}$$

erhält man eine relative Distanz zwischen x und y mit Werten zwischen 0 und 1, die eine Metrik darstellt.

Es besteht damit die Möglichkeit, auf dieser Basis ein Ähnlichkeitsmaß zwischen Bitfolgen einzuführen, um Probleme im Bereich der Datenanalyse zu lösen, wie z. B. den Vergleich von Gensequenzen oder das Entdecken von Malware (SPAM). In der praktischen Anwendung kann dabei natürlich nicht die Kolmogorov-Komplexität $\mathcal{K}$ verwendet werden, denn diese ist ja nicht berechenbar (siehe Satz 7.11). Hier muss man in der Praxis verwendete Kompressionsverfahren einsetzen, wie z. B. Lempel-Ziv- oder ZIP-Verfahren. Wenn wir die Kompressionsfunktionen dieser Verfahren mit κ bezeichnen, gilt natürlich $\mathcal{K}(x) \leq \kappa(x)$ für alle $x \in \mathbb{B}^*$, denn die Kolmogorov-Komplexiät $\mathcal{K}(x)$ ist ja die kürzeste Komprimierung von x.

8.8 Zusammenfassung und bibliografische Hinweise

In diesem Kapitel werden zunächst eine Reihe von Anwendungsmöglichkeiten der Kolmogorov-Komplexität beim Beweis von mathematischen und informatischen Aussagen vorgestellt. Des Weiteren wird die Chaitin-Konstante Ω vorgestellt. In ihr ist das gesamte mathematische Wissen versteckt – inklusive der Antworten auf derzeit offenen Fragen –, ohne dass es umfassend genutzt werden kann. Aus dieses Gründen wird Ω auch „Chaitins Zufallszahl der Weisheit“ genannt.

Am Ende des Kapitels wird angedeutet, wie die Kolmogorov-Komplexität im Bereich der Datenanalyse, zum Beispiel zur Clusterung von Datenmengen, genutzt werden kann. Da die Kolmogorov-Komplexität nicht berechenbar ist, muss sie in solchen Anwendungen durch praktisch verfügbare Kompressionsverfahren ersetzt werden.

Die Themen dieses Kapitels werden mehr oder weniger ausführlich, teilweise in Zusammenhängen mit anderen mathematischen und informatischen Aspekten, in [Ca94], [Ho11], [Hr14] und [LV93] dargestellt und diskutiert. In [LV08] werden weitere Anwendungsmöglichkeiten der Kolmogorov-Komplexität in der Theorie Formaler Sprachen vorgestellt.

Die Chaitin-Konstante wird in diesen Werken ebenfalls mehr oder weniger ausführlich behandelt. Im Literaturverzeichnis gibt es eine umfangreiche Liste von Chaitin-Originalarbeiten, die sich um diese Konstante drehen und Konsequenzen erörtern, die sich daraus in wissenschaftstheoretischen und philosophischen Hinsichten ergeben.

In [CDS02] und [CD07] werden Päfixe von Ω berechnet.

In [LV93] werden auf der Basis der Kolomogorov-Komplexität Distanzmaße für Bitfolgen eingeführt und diskutiert sowie praktische Anwendungen bei der Klassifikation von Daten vorgestellt.

Lösungen zu den Aufgaben

Aufgabe 1.7

a) Es ist

$$x_{21} = [\,10\,]\,1\underbrace{00\ldots0}_{\text{32-mal}},$$

woraus sich

$$x_{31} = \eta(x_{21}) = 10\,1100\,01\,11\underbrace{00\ldots0}_{\text{64-mal}}$$

ergibt. Somit gilt

$$|x_{31}| = 74 < 8\,589\,934\,392 = |x|\,.$$

b) Im Hinblick auf eine lesbare Notation von iterierten Zweierpotenzen führen wir die Funktion $iter_2 : \mathbb{N} \times \mathbb{N}_0 \to \mathbb{N}$, definiert durch

$$iter_2(k,n) = \begin{cases} 2^n, & k = 1, \\ 2^{iter_2(k-1,n)}, & k \geq 2, \end{cases}$$

ein. Es gilt also z. B.

$$iter_2(2,5) = 2^{iter_2(1,5)} = 2^{2^5}.$$

Wir betrachten nun mithilfe der Funktion $iter_2$ allgemein für $w \in \mathbb{B}^*$ die k-fach iterierte Zweierpotenz

$$y = \underbrace{ww\ldots w}_{iter_2(k,n)\text{-mal}}.$$

In der Aufgabe a) ist $w = 10$, $k = 2$ und $n = 5$.

Die Potenzdarstellung von y ist

$$y_1 = [\,w\,]^{iter_2(k,n)}.$$

Wir können nun z. B. als Zwischencodierung für y_1 die Darstellung

$$y_2 = [\,w\,]\;dual(k)\;]\;dual(n)$$

K.-U. Witt und M. E. Müller, *Algorithmische Informationstheorie*,
https://doi.org/10.1007/978-3-662-61694-9

wählen. Dabei ist $dual(m)$ die Dualdarstellung von $m \in \mathbb{N}_0$. Darauf wird dann η angewendet:

$$y_3 = \eta(y_2)$$

c) Mit dem Verfahren aus b) ergibt sich für x_{12} zunächst

$$x_{22} = [\,10\,]\,1\,]\,100000$$

und daraus dann

$$x_{32} = \eta(x_{22}) = 10\,1100\,01\,11\,01\,110000000000.$$

Damit gilt

$$|x_{32}| = 24 < 8\,589\,934\,392 = |x|\,.$$

Analog erhalten wir für x_{13} zunächst

$$x_{23} = [\,10\,]\,10\,]\,101$$

und dann

$$x_{33} = \eta(x_{23}) = 10\,1100\,01\,1100\,01\,110011$$

sowie

$$|x_{33}| = 20 < 8\,589\,934\,392 = |x|\,.$$

d) Erwartungsgemäß erhalten wir für den Fall, dass die Bitfolgen aus Wiederholungen bestehen, deren Anzahl eine iterierte Zweierpotenz ist, sehr gute Komprimierungen.

Aufgabe 2.1

Wir setzen $v^0 = \varepsilon$ sowie $v^{n+1} = v^n v$ für $n \geq 0$.

Aufgabe 2.2

Es ist

$$\begin{aligned} \mathit{Pref}(aba) &= \{\varepsilon, a, ab, aba\} \\ \mathit{Inf}(aba) &= \{\varepsilon, a, b, ab, ba, aba\} \\ \mathit{Suf}(aba) &= \{\varepsilon, a, ba, aba\}\,. \end{aligned}$$

Aufgabe 2.3

Es ergibt sich folgende schrittweise Berechnung:

$$\begin{aligned} |aba| &= |ab| + 1 \\ &= |a| + 1 + 1 \\ &= |\varepsilon| + 1 + 1 + 1 \\ &= 0 + 1 + 1 + 1 \\ &= 3 \end{aligned}$$

Aufgabe 2.4

Zur Bildung eines Wortes $w = w_1 \dots w_k$ mit k Buchstaben über einem Alphabet mit n Buchstaben gibt es für jeden Buchstaben w_i n Möglichkeiten. Damit haben wir

$$\left|\Sigma^k\right| = \underbrace{n \cdot \ldots \cdot n}_{k \text{ -mal}} = n^k.$$

Damit folgt für $n \geq 2$

$$\left|\Sigma^{\leq k}\right| = \sum_{i=0}^{k} \left|\Sigma^{\leq i}\right| = \sum_{i=0}^{k} n^i = \frac{n^{k+1} - 1}{n - 1}.$$

Für $n = 1$ gilt

$$\left|\Sigma^{\leq k}\right| = \left|\left\{\varepsilon, a, a^2, \dots, a^k\right\}\right| = k + 1.$$

Aufgabe 2.6

Es ergibt sich

$$\begin{aligned} |cabcca|_c &= |cabcc|_c = |cabc|_c + 1 = |cab|_c + 1 + 1 \\ &= |ca|_c + 2 = |c|_c + 2 = |\varepsilon|_c + 1 + 2 = 0 + 3 \\ &= 3. \end{aligned}$$

Aufgabe 2.7

$\Sigma(w) = \{a \in \Sigma : |w|_a \geq 1\}$

Aufgabe 2.8

a) Formal kann $tausche$ definiert werden durch

$$tausche(\varepsilon, a, b) = \varepsilon \text{ für alle } a, b \in \Sigma$$

und

$$tausche(cw, a, b) = \begin{cases} b \circ tausche(w, a, b), & \text{falls } c = a, \\ c \circ tausche(w, a, b), & \text{falls } c \neq a, \end{cases} \text{ für } a, b, c \in \Sigma \text{ und } w \in \Sigma^*.$$

b) Es ist

$$\begin{aligned} tausche(12322, 2, 1) &= 1 \circ tausche(2322, 2, 1) = 1 \circ 1 \circ tausche(322, 2, 1) \\ &= 11 \circ 3 \circ tausche(22, 2, 1) = 113 \circ 1 \circ tausche(2, 2, 1) \\ &= 1131 \circ 1 \circ tausche(\varepsilon, 2, 1) = 11311 \circ \varepsilon \\ &= 11311. \end{aligned}$$

Aufgabe 2.9

Eine Möglichkeit ist

$$|x|_y^* = \begin{cases} 0, & |x| < |y|\,, \\ |x[\,2, |x|\,]|_y^* + 1, & y \in \mathit{Pref}(x), \\ |x[\,2, |x|\,]|_y^*\,, & y \notin \mathit{Pref}(x). \end{cases}$$

Falls y länger als x ist, kann y kein Infix von x sein. Anderenfalls wird geprüft, ob y Präfix von x. Ist das der Fall, dann wird berechnet, wie oft y Infix im Wort x ohne den ersten Buchstaben ist, und dazu eine 1 addiert. Ist das nicht der Fall, dann wird nur berechnet, wie oft y Infix von x ohne den ersten Buchstaben ist.

Aufgabe 2.10

Wir definieren $\overleftarrow{\cdot} : \Sigma^* \to \Sigma^*$ durch

$$\begin{aligned} \overleftarrow{\varepsilon} &= \varepsilon, \\ \overleftarrow{wa} &= a\overleftarrow{w} \text{ für } a \in \Sigma,\, w \in \Sigma^*. \end{aligned}$$

Aufgabe 2.11

Gemäß (2.7) gilt $L^0 = \{\varepsilon\}$ für alle Sprachen L, also ist auch $\emptyset^0 = \{\varepsilon\}$ und damit $\emptyset^* = \{\varepsilon\}$.

Es ist $\emptyset^1 = \emptyset$ und damit $\emptyset^+ = \emptyset$.

Da $\varepsilon^n = \varepsilon$ für alle $n \in \mathbb{N}_0$ ist, gilt $\{\varepsilon\}^* = \{\varepsilon\}$ sowie $\{\varepsilon\}^+ = \{\varepsilon\}$.

Aufgabe 2.12

Wir können die Codierung η^* aus Abschnitt 1.2.1, welche die Bits in einer Bitfolge verdoppelt, wählen.

Aufgabe 2.13

a) Wir beweisen die Behauptung mit vollständiger Induktion über die Länge k der Wörter w.

Induktionsanfang: Für $k = 1$ gilt einerseits

$$\tau_\Sigma(a_i) = i \text{ für } 1 \le i \le n$$

und andererseits

$$\tau'_\Sigma(a_i) = \sum_{i=1}^{1} \tau(a_i) \cdot n^{1-i} = \tau(a_i).$$

womit der Induktionsanfang gezeigt ist.

Induktionsschritt: Unter Verwendung der Annahme, dass die Behauptung für ein $k \in \mathbb{N}$ gilt, zeigen wir, dass dann die Behauptung auch für $k+1$ gilt, denn es ist

$$\begin{aligned}\tau'_\Sigma(w_1 \dots w_k w_{k+1}) &= \sum_{i=1}^{k+1} \tau_\Sigma(w_i) \cdot n^{k+1-i} \\ &= n \cdot \sum_{i=1}^{k} \tau_\Sigma(w_i) \cdot n^{k-i} + \tau_\Sigma(w_{k+1}) \\ &= n \cdot \tau_\Sigma(w_1 \dots w_k) + \tau_\Sigma(w_{k+1}) \text{ mit Induktionsannahme} \\ &= \tau_\Sigma(w_1 \dots w_k w_{k+1}) \text{ mit Definition von } \tau_\Sigma.\end{aligned}$$

b) Auch diese Behauptung zeigen wir mit vollständiger Induktion über die Länge k der Wörter w.

Induktionsanfang: Für $k = 1$ gilt

$$\begin{aligned}\tau_\mathbb{B}(0) &= 1 = 2 - 1 = wert(10) - 1, \\ \tau_\mathbb{B}(1) &= 2 = 3 - 1 = wert(11) - 1.\end{aligned}$$

Induktionsschritt: Unter Verwendung der Annahme, dass die Behauptung für ein $k \in \mathbb{N}$ gilt, zeigen wir, dass dann die Behauptung auch für $k+1$ gilt, denn es ist

$$\begin{aligned}&\tau_\mathbb{B}(w_1 \dots w_k w_{k+1}) \\ &\quad= 2 \cdot \tau_\mathbb{B}(w_1 \dots w_k) + \tau_\mathbb{B}(w_{k+1}) \text{ Definition von } \tau \\ &\quad= 2 \cdot wert(1 w_1 \dots w_k) - 1 + \tau_\mathbb{B}(w_{k+1}) \text{ Induktionsannahme} \\ &\quad= wert(1 w_1 \dots w_k 0) + w_{k+1} - 1 \\ &\quad= 1 \cdot 2^{k+1} + w_1 \cdot 2^k + \dots + w_k \cdot 2^1 + 0 \cdot 2^0 + w_{k+1} \cdot 2^0 - 1 \\ &\quad= 1 \cdot 2^{k+1} + w_1 \cdot 2^k + \dots + w_k \cdot 2^1 + w_{k+1} \cdot 2^0 - 1 \\ &\quad= wert(1 w_1 \dots w_k w_{k+1}) - 1.\end{aligned}$$

Aufgabe 2.14

Mit $\lceil \log |\Sigma| \rceil = \lceil \log 26 \rceil = 5$ ergibt sich $bin_\Sigma(a) = 00000$, $bin_\Sigma(b) = 00001$, $bin_\Sigma(c) = 00010$, …, $bin_\Sigma(z) = 11001$.

Aufgabe 2.15

Die Codierung der Zahl $n \in \mathbb{N}_0$ durch $|^n$ hat die Länge n. Die Codierung von n als Dualzahl hat die Länge $\lfloor \log n \rfloor + 1$. Die unäre Codierung ist also exponentiell länger als die Dualcodierung.

Aufgabe 2.16

a) Wir definieren $g : \mathbb{Z} \to \mathbb{N}_0$ durch

$$g(z) = \begin{cases} 0, & \text{falls } z = 0, \\ 2z, & \text{falls } z > 0, \\ -(2z+1), & \text{falls } z < 0. \end{cases}$$

g ist eine bijektive Abbildung, die den positiven ganzen Zahlen die geraden Zahlen und den negativen ganzen Zahlen die ungeraden Zahlen zuordnet. g nummeriert die ganzen Zahlen wie folgt:

$$\begin{array}{cccccccccc} \mathbb{Z} & 0 & -1 & 1 & -2 & 2 & -3 & 3 & \dots \\ g & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \dots \\ \mathbb{N}_0 & 0 & 1 & 2 & 3 & 4 & 5 & 6 & \dots \end{array}$$

b) Wir können die Menge der Brüche wie folgt notieren als

$$\mathbb{Q} = \Big\{ \frac{m}{n} \mid m \in \mathbb{Z}, n \in \mathbb{N} \Big\}.$$

Wenn wir die Brüche $\frac{m}{n}$ als Paare (m, n) schreiben, ergibt sich

$$\begin{aligned} \mathbb{Q} &= \{(m, n) \mid m \in \mathbb{Z}, n \in \mathbb{N}\} \\ &= \mathbb{Z} \times \mathbb{N} \\ &= (-\mathbb{N} \times \mathbb{N}) \cup \{0\} \cup (\mathbb{N} \times \mathbb{N}). \end{aligned}$$

Wir kennen bereits die Nummerierung c von $\mathbb{N} \times \mathbb{N}$. Wir setzen $c_-(m, n) = -c(m, n)$ und erhalten damit eine Nummerierung $c_- : -\mathbb{N} \times \mathbb{N} \to -\mathbb{N}$. Damit definieren wir

$$h : (-\mathbb{N} \times \mathbb{N}) \cup \{0\} \cup (\mathbb{N} \times \mathbb{N}) \to \mathbb{Z}$$

durch

$$\begin{aligned} h(0) &= 0, \\ h(m, n) &= \begin{cases} c(m, n), & (m, n) \in \mathbb{N} \times \mathbb{N}, \\ c_-(m, n), & (-m, n) \in -\mathbb{N} \times \mathbb{N}. \end{cases} \end{aligned}$$

In a) haben wir die Nummerierung g von $\mathbb{Z}$ definiert. Wir führen nun h und g hintereinander aus und erhalten damit die Nummerierung $q : \mathbb{Q} \to \mathbb{N}_0$ von $\mathbb{Q}$, definiert durch $q = g \circ h$.

Bemerkenswert ist, dass sich die Menge $\mathbb{Q}$ nummerieren lässt, obwohl sie dicht ist. Dicht bedeutet, dass zwischen je zwei verschiedenen Brüchen wieder ein Bruch, z. B. ihr arithmetisches Mittel, liegt (daraus folgt im Übrigen, dass zwischen zwei verschiedenen Brüchen unendlich viele weitere Brüche liegen).

Aufgabe 3.1

Die Lösungsidee ist genau wie im Beispiel 3.1. Es werden in jeder Runde wieder der erste und letzte Buchstabe abgeglichen. Im Beispiel 3.1 muss jeweils der erste Buchstabe eine 0 und der letzte eine 1 sein; jetzt müssen jeweils Anfangs- und Endbuchstabe identisch sein.

Eine Maschine, die dieses Verfahren realisiert, ist

$$T = (\mathbb{B} \cup \{\#\}, \{s, s_0, s_1, z, z_0, z_1, t_a, t_r\}, \delta, s, \#, t_a, t_r)$$

mit

$$\begin{aligned}
\delta = \{&(s, \#, t_a, \#, -), && \text{leeres Wort akzeptieren}\\
&(s, 0, s_0, \#, \rightarrow), && \text{die nächste 0 wird überschrieben}\\
&(s_0, 0, s_0, 0, \rightarrow), && \text{zum Ende laufen, über Nullen}\\
&(s_0, 1, s_0, 1, \rightarrow), && \text{und Einsen hinweg}\\
&(s, 1, s_1, \#, \rightarrow), && \text{die nächste 1 wird überschrieben}\\
&(s_1, 0, s_1, 0, \rightarrow), && \text{zum Ende laufen, über Nullen}\\
&(s_1, 1, s_0, 1, \rightarrow), && \text{und Einsen hinweg}\\
&(s_0, \#, z_0, \#, \leftarrow), && \text{am Ende angekommen, ein Symbol zurück}\\
&(s_1, \#, z_1, \#, \leftarrow), && \text{am Ende angekommen, ein Symbol zurück}\\
&(z_0, 0, z, \#, \leftarrow), && \text{prüfen, ob 0, dann zurück}\\
&(z_1, 1, z, \#, \leftarrow), && \text{prüfen, ob 1, dann zurück}\\
&(z_0, 1, t_r, 1, -), && \text{falls keine 0, Eingabe verwerfen}\\
&(z_0, \#, t_r, \#, -), && \text{falls Band leer, Eingabe verwerfen}\\
&(z_1, 0, t_a, 0, -), && \text{falls keine 1, Eingabe verwerfen}\\
&(z_1, \#, t_r, \#, -), && \text{falls Band leer, Eingabe verwerfen}\\
&(z, 1, z, 1, \leftarrow), && \text{zurück über alle Einsen}\\
&(z, 0, z, 0, \leftarrow), && \text{und Nullen}\\
&(z, \#, s, \#, \rightarrow)\} && \text{am Wortanfang angekommen, neue Runde.}
\end{aligned}$$

Mit dieser Maschine ergeben sich die folgenden Tests:

$$\begin{aligned}
s1001 &\vdash s_1001 \vdash 0s_101 \vdash 00s_11 \vdash 001s_1\# \vdash 00z_11 \vdash 0z0 \vdash z00 \vdash z\#00\\
&\vdash s00 \vdash s_00 \vdash 0s_0\# \vdash z_00 \vdash z\# \vdash s\# \vdash t_a\#\\
s0111 &\vdash s_0111 \vdash 1s_011 \vdash 11s_01 \vdash 111s_0\# \vdash 11z_01 \vdash 11t_r1\\
s111 &\vdash s_111 \vdash 1s_11 \vdash 11s_1\# \vdash 1z_11 \vdash z1 \vdash z\#1 \vdash s1 \vdash s_1\# \vdash z_1\# \vdash t_r\#
\end{aligned}$$

Es gilt also $\Phi_T(1001) = 1$, $\Phi_T(0111) = 0$, $\Phi_T(111) = 0$.

Aufgabe 3.2

Grundidee für die Konstruktion: Das Eingabewort steht auf Band 1. Das Präfix des Eingabewortes, das aus Nullen besteht, wird auf Band 2 kopiert und auf Band 1 mit Blanks überschrieben. Dann wird geprüft, ob der Rest des Eingabewortes aus genau so viel Einsen besteht, wie Nullen auf Band 2 kopiert wurden. Stimmt die Bilanz, akzeptiert die Maschine; in allen anderen Fällen verwirft die Maschine die Eingabe.

Die folgende 2-bändige Maschine realisiert dieses Verfahren:

$$T = (\mathbb{B} \cup \{\#\}, \{s, s_0, k, t_a, t_r\}, \delta, s, \#, t_a, t_r)$$

δ ist definiert durch

$$\begin{aligned}
\delta = \{&(s, \#, \#, t_a, \#, \#, -, -), && \text{leeres Wort akzeptieren}\\
&(s, 1, \#, t_r, 1, \#, -, -), && \text{Eingabe beginnt mit 1, verwerfen}\\
&(s, 0, \#, s_0, \#, 0, \rightarrow, \rightarrow), && \text{Nullen auf Band 2 schreiben}\\
&(s_0, 0, \#, s_0, \#, 0, \rightarrow, \rightarrow), && \text{Eingabe besteht nur aus Nullen,}\\
&(s_0, \#, \#, t_r, \#, \#, -, -), && \text{verwerfen}\\
&(s_0, 1, \#, k, 1, \#, -, \leftarrow), && \text{erste 1}\\
&(k, 1, 0, k, \#, \#, \rightarrow, \leftarrow), && \text{Abgleich Nullen und Einsen}\\
&(k, \#, \#, t_a, \#, \#, -, -), && \text{Bilanz ausgeglichen, akzeptieren}\\
&(k, 0, 0, t_r, 0, 0, -, -), && 0 \text{ nach } 1, \text{ verwerfen}\\
&(k, 0, \#, t_r, 0, \#, -, -), &&\\
&(k, \#, 0, t_r, \#, 0, -, -), && \text{zu wenig Einsen, verwerfen}\\
&(k, 1, \#, t_r, 1, \#, -, -)\} && \text{zu wenig Nullen, verwerfen.}
\end{aligned}$$

Wir testen das Programm mit den Eingaben 0011, 0111 und 00110:

$$\begin{aligned}
(s, \uparrow 0011, \uparrow\#) &\vdash (s_0, \#\uparrow 011, \#0\uparrow\#) \vdash (s_0, \#\uparrow 11, \#00\uparrow\#) \vdash (k, \#\uparrow 11, \#0\uparrow 0\#)\\
&\vdash (k, \#\uparrow 1, \#\uparrow 0\#) \vdash (k, \#\uparrow\#, \#\uparrow\#) \vdash (t_a, \#\uparrow\#, \#\uparrow\#)\\
(s, \uparrow 0111, \uparrow\#) &\vdash (s_0, \uparrow 111, \#0\uparrow\#) \vdash (k, \uparrow 111, \#\uparrow 0\#)\\
&\vdash (k, \uparrow 11, \#\uparrow\#) \vdash (t_r, \uparrow 11, \#\uparrow\#)\\
(s, \uparrow 00110, \uparrow\#) &\vdash (s_0, \#\uparrow 0110, \#0\uparrow\#) \vdash (s_0, \#\uparrow 110, \#00\uparrow\#)\\
&\vdash (k, \#\uparrow 110, \#0\uparrow 0\#) \vdash (k, \#\uparrow 10, \#\uparrow 0\#)\\
&\vdash (k, \#\uparrow 0, \#\uparrow\#) \vdash (t_r, \#\uparrow 0, \#\uparrow\#)
\end{aligned}$$

Aufgabe 3.3

Sei T_f eine 2-Bandmaschine, die f berechnet, und T_g eine 2-Bandmaschine, die g berechnet. Wir konstruieren daraus eine 3-Bandmaschine T_h, die bei Eingabe x auf das erste Band $y = \Phi_{T_g}(x) = g(x)$ berechnet und y auf Band 2 schreibt. Dann führt T_h die Maschine T_f mit Eingabe y auf Band 2 aus und schreibt das Ergebnis $z = \Phi_{T_f}(y) = f(y)$ auf Band 3. Es gilt

$$\Phi_{T_h}(x) = \begin{cases} \bot, & x \notin \mathit{Def}(\Phi_{T_g}),\\ \bot, & \Phi_{T_g}(x) \notin \mathit{Def}(\Phi_{T_f}),\\ \Phi_{T_f}(\Phi_{T_g}(x)), & \text{sonst.} \end{cases}$$

Es folgt $\Phi_{T_h}(x) = \Phi_{T_f}(\Phi_{T_g}(x)) = (\Phi_{T_f} \circ \Phi_{T_g})(x) = (f \circ g)(x) = h(x)$; h ist also berechenbar.

Aufgabe 3.4

Die Funktion $\omega : \mathbb{B}^* \to \mathbb{B}^*$ sei definiert durch $\omega(x) = \perp$ für alle $x \in \mathbb{B}^*$. ω ist die „nirgends definierte Funktion“; es gilt also $Def(\omega) = \emptyset$.

Die Turingmaschine $T = (\mathbb{B} \cup \{\#\}, \{s, t_a, t_r\}, \delta, s, \#, t_a, t_r)$ mit

$$\delta = \{(s, 0, s, 0, \rightarrow), (s, 1, s, 1, \rightarrow), (s, \#, s, \#, \rightarrow)\}$$

wandert bei jeder Eingabe über diese nach rechts hinweg und weiter über alle Blanks, hält also nie an. Damit gilt $\Phi_T(x) = \perp$ für alle $x \in \mathbb{B}^*$. Somit ist also $\Phi_T = \omega$ und $\omega \in \mathcal{P} - \mathcal{R}$ sowie $\mathcal{R} \subset \mathcal{P}$.

Aufgabe 3.5

a) Wir setzen $\chi_A(x) = 1 - \mathit{sign}\left(\prod_{a \in A} |x - a|\right)$. Diese Funktion ist eine charakteristische Funktion für A und berechenbar.

b) Wir setzen

$$\begin{aligned}\chi_{\overline{A}}(x) &= 1 - \chi_A(x),\\ \chi_{A \cup B} &= \max\{\chi_A(x), \chi_B(x)\},\\ \chi_{A \cap B} &= \min\{\chi_A(x), \chi_B(x)\}\end{aligned}$$

und erhalten damit berechenbare charakteristische Funktionen für $\overline{A}$, $A \cup B$ und $A \cap B$.

Aufgabe 3.6

Es sei T_A ein Turingprogramm, das χ'_A berechnet, und T_B ein Turingprogramm, das χ'_B berechnet. Wir konstruieren zwei 2-Bandmaschinen $T_{A \cup B}$ und $T_{A \cap B}$. Beide führen T_A auf dem ersten Band und T_B auf dem zweiten Band wechselweise – also quasi parallel – aus. $T_{A \cup B}$ stoppt, falls T_A oder T_B stoppt, und gibt eine 1 aus. $T_{A \cap B}$ stoppt, wenn beide angehalten haben und gibt dann eine 1 aus. Offensichtlich gilt $\Phi_{T_{A \cup B}} = \chi'_{A \cup B}$ bzw. $\Phi_{T_{A \cap B}} = \chi'_{A \cap B}$.

Aufgabe 3.7

a) „$\Rightarrow$“: Sei A entscheidbar. Dann ist χ_A berechenbar. Sei X das Programm, das χ_A berechnet: $\Phi_X = \chi_A$. Das Programm B in der folgenden Abbildung durchläuft alle natürlichen Zahlen $i \in \mathbb{N}_0$ in aufsteigender Reihenfolge und überprüft jeweils mithilfe des Programms X, ob $i \in A$ ist. Falls ja, wird i ausgegeben und weitergezählt; falls nein, wird nur weitergezählt.

„$\Leftarrow$“: Sei T eine Turingmaschine mit zwei Bändern. Band 1 sei das Eingabeband, auf Band 2 stehen die Elemente von A in aufsteigender Reihenfolge von links nach rechts. Bei Eingabe i vergleicht T das i mit den Elementen auf Band 2 von links nach rechts. T stoppt, falls i auf dem Band steht, und gibt eine 1 aus. Falls i nicht auf dem Band steht, stoppt T bei Erreichen des ersten Elements, das größer als i ist, und gibt eine 0 aus. Da die Elemente auf Band 2 aufsteigend sortiert sind, kann i nicht nach einem

```
algorithm B;
  input:
  output: alle Elemente von A aufsteigend;
    i := 0;
    while true do
        if Φ_X(i) = 1 then return i endif;
        i := i + 1
    endwhile
endalgorithm
```

Algorithmus B, der die Elemente der entscheidbaren Menge A der Größe nach ausgibt.

bereits größeren Element auf dem Band stehen. Es gilt offensichtlich $\Phi_T = \chi_A \cdot \chi_A$ ist also berechenbar; damit ist A entscheidbar.

b) „⇒“: Ist $A = \emptyset$, dann konstruieren wir die 2-Bandmaschine T wie folgt: Band 1 ist das Eingabeband, Band 2 ist das Ausgabeband, und T berechnet auf Band 1 die Funktion ω (siehe Lösung der Aufgabe 3.4).

Sei $A \neq \emptyset$ semi-entscheidbar, d. h., A ist rekursiv-aufzählbar. Sei f eine rekursive Aufzählung von f und F das Programm, das f berechnet: $\Phi_F = f$. Das folgende Programm C durchläuft alle natürlichen Zahlen $i \in \mathbb{N}_0$ in aufsteigender Reihenfolge und gibt $f(i)$ aus.[2]

```
algorithm C;
  input:
  output: alle Elemente von A;
    i := 0;
    while true do
        return Φ_F(i);
        i := i + 1
    endwhile
endalgorithm
```

Algorithmus C, der die Elemente der semi-entscheidbaren Menge A ausgibt.

„⇐“: Sei T eine Turingmaschine mit zwei Bändern. Band 1 sei das Eingabeband, auf Band 2 stehen die Elemente von A (in irgendeiner Reihenfolge von links nach rechts).

[2] Da f nicht injektiv sein muss, können Elemente von A mehrfach ausgegeben werden. Wenn man das vermeiden möchte, muss man bei jedem i die bereits ausgegebenen Elemente mit $f(i)$ vergleichen. Existiert $f(i)$ bereits, dann braucht es nicht noch einmal ausgegeben zu werden, anderefalls wird es erstmalig – und nur einmalig – ausgegeben.

Bei Eingabe i vergleicht T diese mit den Elementen auf Band 2 von links nach rechts. T stoppt, falls i auf dem Band steht, und gibt eine 1 aus. Falls i nicht auf dem Band steht, stoppt T nicht. Es gibt kein Kriterium, das T benutzen kann, um nicht weiter zu suchen und eine 0 auszugeben. Es gilt offensichtlich $\Phi_T = \chi'_A$. Die Funktion χ'_A ist also berechenbar; damit ist A semi-entscheidbar.

Mithilfe von T kann auch eine rekursive Aufzählung von A konstruiert werden. Da $A = \emptyset$ per se rekursiv aufzählbar ist, ist in diesem Fall nichts zu tun.

Sei $A \neq \emptyset$ unendlich. Bei Eingabe $i \in \mathbb{N}_0$ durchläuft T die Elemente auf Band 2 von links nach rechts bis zum i-ten Element und gibt dieses aus.

Sei $A \neq \emptyset$ endlich mit $|A| = k$, und x sei irgendein Element von A. Bei Eingabe $i \in \mathbb{N}_0$ mit $i \leq k-1$ durchläuft T die Elemente auf Band 2 von links nach rechts bis zum i-ten Element und gibt dieses aus. Ist $i \geq k$, dann gibt T das Element x aus.

In beiden Fällen ist T für alle $i \in \mathbb{N}_0$ definiert, d. h., Φ_T ist total definiert, und es gilt $W(\Phi_T) = A$. Φ_T ist somit eine rekursive Aufzählung von A.

Aufgabe 3.9

Der Wunsch nach einer nicht deterministischen Maschine kommt daher, dass man beim Scannen einer Eingabe nicht weiß, wo das Wort w endet und seine Wiederholung beginnt.

Ein Verifizierer könnte bei Eingabe eines Wortes $x \in \mathbb{B}^*$ als Zertifikat eine Zahl k raten und dann deterministisch überprüfen, ob $x[i] = x[k+i]$ für $1 \leq i \leq k$ gilt. Falls x zur Menge L gehört, gibt es ein solches k, falls x nicht zu L gehört, gibt es kein k.

Eine nicht deterministische Maschine muss beim Scannen der Eingabe davon ausgehen, dass bei jedem Buchstaben die Wiederholung des ersten Teils beginnen könnte. Dementsprechend gibt es bei jedem Buchstaben zwei Möglickeiten: weiter lesen oder annehmen, dass die Wiederholung beginnt.

Die folgende 2-bändige Maschine T schreibt die Buchstaben auf das Band 2. Sie nimmt bei jedem Buchstaben außerdem an, dass die Wiederholung beginnen könnte. Der S-/L-Kopf von Band 1 bleibt dann stehen, der von Band 2 läuft an den Anfang des Bandes. Dann beginnt der Abgleich von Band 1 und Band 2. Falls die Bilanz stimmt, akzeptiert T die Eingabe. In allen anderen Fällen verwirft T das Eingabewort:

$$T = (\mathbb{B} \cup \{\#\}, \{s, z, k, t_a, t_r\}, \delta, s, \#, t_a, t_r)$$

δ ist definiert durch

$\delta = \{(s, \#, \#, t_a, \#, \#, -, -),$	leeres Wort akzeptieren
$(s, 0, \#, s, \#, 0, \rightarrow, \rightarrow),$	Nullen und
$(s, 1, \#, s, \#, 1, \rightarrow, \rightarrow),$	Einsen auf Band 2 schreiben
$(s, 0, \#, z, 0, \#, -, \leftarrow),$	Annahme, Wiederholung erreicht,
$(s, 1, \#, z, 0, \#, -, \leftarrow),$	
$(z, 0, 0, z, 0, 0, -, \leftarrow),$	zum Anfang von Band 2

$(z, 0, 1, z, 0, 1, -, \leftarrow)$,	zurück
$(z, 1, 0, z, 1, 0, -, \leftarrow)$,	
$(z, 1, 1, z, 1, 1, -, \leftarrow)$,	
$(z, \#, 0, k, \#, 0, -, \rightarrow)$,	Bandanfang erreicht
$(z, \#, 1, k, \#, 1, -, \rightarrow)$,	
$(k, 0, 0, k, \#, \#, \rightarrow, \rightarrow)$,	Abgleich Band 1 und 2
$(k, 1, 1, k, \#, \#, \rightarrow, \rightarrow)$,	
$(k, \#, \#, t_a, \#, \#, -, -)$,	Bilanz stimmt, akzeptieren
$(k, 0, 1, t_r, \#, \#, -, -)$,	Bilanz stimmt nicht, verwerfen
$(k, 1, 0, t_r, \#, \#, -, -)$,	alle anderen Fälle verwerfen
$(k, 0, \#, t_r, \#, \#, -, -)$,	
$(k, 1, \#, t_r, \#, \#, -, -)$,	
$(k, \#, 0, t_r, \#, \#, -, -)$,	
$(k, \#, 1, t_r, \#, \#, -, -)\}$.	

Aufgabe 4.1

a) Sei f eine polynomielle Reduktion von A auf B. Dann gilt $x \in A$ genau dann, wenn $f(x) \in B$ ist, was äquivalent ist zur Aussage $x \in \overline{A}$ genau dann, wenn $f(x) \in \overline{B}$ ist.

b) Sei f eine polynomielle Reduktion von A auf B und g eine polynomielle Reduktion von B auf C. Dann gilt $x \in A$ genau dann, wenn $f(x) \in B$ ist, sowie $y \in B$ genau dann, wenn $g(y) \in C$ ist. Es folgt $x \in A$ genau dann, wenn $f(g(x)) \in C$ ist, und $f \circ g$ ist polynomiell berechenbar.

Aufgabe 4.2

Der Beweis folgt analog zum Beweis von Satz 4.6: Aus $A \leq_{\mathsf{poly}} B$ folgt, dass es eine deterministisch in Polynomzeit berechenbare Funktion f geben muss mit $A \leq_f B$. Da $B \in \mathsf{NP}$ ist, ist die charakteristische Funktion χ_B von B nicht deterministisch in Polynomzeit berechenbar. Die Komposition von f und χ_B ist eine charakteristische Funktion für A: $\chi_A = \chi_B \circ f$. Die Funktion f ist deterministisch in Polynomzeit berechenbar, und χ_B ist nicht deterministisch in Polynomzeit berechenbar, damit ist auch χ_A nicht deterministisch in Polynomzeit berechenbar, also ist $A \in \mathsf{NP}$.

Aufgabe 4.3

a) Sei $A \in \mathsf{C}$ und $T = (\Gamma, \delta, s, \#, t_a, t_r)$ ein Entscheider für A. Dann ist die Maschine $\overline{T} = (\Gamma, \delta, s, \#, t_r, t_a)$ ein Entscheider für $\overline{A}$, und es gilt $time_T(w) = time_{\overline{T}}(w)$ für alle $w \in \mathbb{B}^*$. Es folgt $A \in \mathsf{C}$ genau dann, wenn $\overline{A} \in \mathsf{C}$, womit die Behauptung gezeigt ist.

b) Folgt aus a), da P eine deterministische Komplexitätsklasse ist.

c) Aus $\mathsf{P} \subseteq \mathsf{NP}$ folgt unmittelbar $\mathsf{coP} \subseteq \mathsf{coNP}$ und daraus mit b) $\mathsf{P} \subseteq \mathsf{coNP}$. Also gilt $\mathsf{P} \subseteq \mathsf{NP} \cap \mathsf{coNP}$.

d) Aus $\mathsf{P} = \mathsf{NP}$ folgt $\mathsf{coP} = \mathsf{coNP}$. Daraus folgt mit b) $\mathsf{NP} = \mathsf{P} = \mathsf{coP} = \mathsf{coNP}$.

e) „⇒“: Sei $A \in \mathsf{NPC}$, dann gilt wegen der Voraussetzung $\mathsf{NP} = \mathsf{coNP}$ und wegen $\mathsf{NPC} \subseteq \mathsf{NP}$ auch $A \in \mathsf{coNP}$, woraus insgesamt $A \in \mathsf{NPC} \cap \mathsf{coNP}$ folgt. Damit ist $\mathsf{NPC} \cap \mathsf{coNP} \neq \emptyset$ gezeigt.

„⇐“: Gemäß Voraussetzung existiert eine Menge $A \in \mathsf{NPC}$ mit $A \in \mathsf{coNP}$.

Sei $B \in \mathsf{NP}$, dann gilt $B \leq_{\mathsf{poly}} A$ und damit $\overline{B} \leq_{\mathsf{poly}} \overline{A}$. Hieraus folgt, da $\overline{A} \in \mathsf{NP}$ ist, dass auch $\overline{B} \in \mathsf{NP}$ ist, womit $B \in \mathsf{coNP}$ ist. Aus $B \in \mathsf{NP}$ folgt also $B \in \mathsf{coNP}$ und damit $\mathsf{NP} \subseteq \mathsf{coNP}$.

Sei nun $B \in \mathsf{coNP}$, dann ist $\overline{B} \in \mathsf{NP}$. Es folgt $\overline{B} \leq_{\mathsf{poly}} A$, da $A \in \mathsf{NPC}$ ist. Es folgt $B \leq_{\mathsf{poly}} \overline{A}$ und daraus $B \in \mathsf{NP}$, da $\overline{A} \in \mathsf{NP}$ ist. Aus $B \in \mathsf{coNP}$ folgt also $B \in \mathsf{NP}$ und damit $\mathsf{coNP} \subseteq \mathsf{NP}$.

Damit ist insgesamt $\mathsf{NP} = \mathsf{coNP}$ gezeigt.

Aufgabe 5.1

Es ist

$$\langle\, T_\omega \,\rangle = (Ass|s||(s0s0m||)(s1s1m||)(sAsAm||)),$$

woraus sich

$$\rho(\langle\, T_\omega \,\rangle) = 623353556303455763131455763232455 77$$

ergibt.

Aufgabe 6.1

a) Wir definieren $f : \mathbb{N}_0 \times \mathbb{N}_0 \to \mathbb{N}_0$ durch $f(i,j) = u_\varphi(i,j)$. Dann gilt: Die Funktion f ist berechenbar sowie

$$\begin{aligned}(i,j) \in H \;&\text{ genau dann, wenn } j \in \mathit{Def}(\varphi_i),\\ &\text{genau dann, wenn } (i,j) \in \mathit{Def}(u_\varphi),\\ &\text{genau dann, wenn } (i,j) \in \mathit{Def}(f).\end{aligned}$$

Es ist also $H = \mathit{Def}(f)$. Mit Satz 3.4 b) folgt, dass H rekursiv aufzählbar ist.

b) Wir definieren $g : \mathbb{N}_0 \to \mathbb{N}_0$ durch $g(i) = (i,i)$. Die Funktion g ist offensichtlich total berechenbar, und es gilt

$$\begin{aligned}i \in K \;&\text{ genau dann, wenn } (i,i) \in H,\\ &\text{genau dann, wenn } g(i) \in H.\end{aligned}$$

Damit ist $K \leq_g H$ gezeigt. Da K nicht entscheidbar ist, ist wegen Folgerung 3.3 H ebenfalls nicht entscheidbar.

Aufgabe 6.2

Sei $i \in A_E$ mit $\varphi_i = f \in E$ sowie $i \approx j$, also $\varphi_i \approx \varphi_j$. Es folgt $\varphi_j = f$ und damit $\varphi_j \in E$ und damit $j \in A_E$.

Anhang: Mathematische Grundbegriffe

Wenn M eine Menge und a ein Element dieser Menge ist, schreiben wir $a \in M$; ist a kein Element von M, dann schreiben wir $a \notin M$. Eine Menge M definieren wir im Allgemeinen auf folgende Arten:

$$M = \{x \mid p(x)\} \quad \text{oder} \quad M = \{x : p(x)\}$$
$$M = \{x \in A \mid p(x)\} \quad \text{oder} \quad M = \{x \in A : p(x)\}$$

p ist ein Prädikat, das – zumeist halbformal notiert – eine Eigenschaft angibt, die bestimmt, ob ein Element a zu M gehört: Gilt ($a \in A$ und) $p(a)$, d. h., trifft die Eigenschaft p auf das Element a zu, dann gilt $a \in M$.

Ein Prädikat p trifft auf fast alle Elemente einer Menge M zu, falls es auf alle bis auf endlich viele Elemente von M zutrifft.

Ist M eine endliche Menge mit m Elementen, dann notieren wir das mit $|M| = m$. Ist M eine unendliche Menge, dann schreiben wir $|M| = \infty$.

Folgende Bezeichner stehen für gängige Zahlenmengen:

$\mathbb{N}_0 = \{0, 1, 2, \ldots\}$	Menge der natürlichen Zahlen
$\mathbb{N} = \mathbb{N}_0 - \{0\}$	Menge der natürlichen Zahlen ohne 0
$-\mathbb{N} = \{-n \mid n \in \mathbb{N}\}$	Menge der negativen ganzen Zahlen
$\mathbb{Z} = \mathbb{N}_0 \cup -\mathbb{N}$	Menge der ganzen Zahlen
$\mathbb{P} = \{x \mid x \text{ besitzt keinen echten Teiler}\}$	Menge der Primzahlen
$\mathbb{Q} = \left\{ \frac{p}{q} \mid p \in \mathbb{Z}, q \in \mathbb{N} \right\}$	Menge der rationalen Zahlen
$\mathbb{Q}_+ = \{x \in \mathbb{Q} \mid x \geq 0\}$	Menge der positiven rationalen Zahlen
$\mathbb{R}$	Menge der reellen Zahlen
$\mathbb{R}_+ = \{x \in \mathbb{R} \mid x \geq 0\}$	Menge der positiven reellen Zahlen

Ist jedes Element einer Menge A auch Element der Menge B, dann ist A eine Teilmenge von B: $A \subseteq B$. Es gilt $A = B$, falls $A \subseteq B$ und $B \subseteq A$ zutreffen. Die Menge A ist eine echte Teilmenge von B, $A \subset B$, falls $A \subseteq B$ und $A \neq B$ gilt.

K.-U. Witt und M. E. Müller, *Algorithmische Informationstheorie*,
https://doi.org/10.1007/978-3-662-61694-9

Die Potenzmenge einer Menge M ist die Menge $2^M = \{A \mid A \subseteq M\}$ aller Teilmengen von M. Ist $|M| = m$, dann ist $\left|2^M\right| = 2^m$.

Sind A und B Mengen, dann ist $A \cup B = \{x \mid x \in A \text{ oder } x \in B\}$ die Vereinigung von A und B; $A \cap B = \{x \mid x \in A \text{ und } x \in B\}$ ist der Durchschnitt von A und B; und $A - B = \{x \mid x \in A \text{ und } x \notin B\}$ ist die Differenz von A und B. Ist $A \subseteq B$, dann heißt $B - A$ das Komplement von A bezüglich B.

Seien A_i, $1 \leq i \leq k$, Mengen, dann heißt

$$A_1 \times A_2 \times \ldots \times A_k = \{(a_1, a_2, \ldots, a_k) \mid a_i \in A_i,\ 1 \leq i \leq k\}$$

das kartesische Produkt der Mengen A_i. $a = (a_1, a_2, \ldots, a_k)$ heißt k-Tupel (im Fall $k = 2$ Paar, im Fall $k = 3$ Tripel, ...). a_i ist die i-te Komponente von a.

Eine Menge $R \subseteq A_1 \times A_2 \times \ldots \times A_k$ heißt k-stellige Relation über $A_1, \ldots, A_k$.

Für $R \subseteq A \times B$ heißt $R^{-1} \subseteq B \times A$, definiert durch $R^{-1} = \{(y, x) \mid (x, y) \in R\}$, Umkehrrelation oder inverse Relation von R.

Seien $R \subseteq A \times B$ und $S \subseteq B \times C$, dann heißt die Relation $R \circ S \subseteq A \times C$, definiert durch

$$R \circ S = \{(x, z) \mid \text{es existiert ein } y \in B \text{ mit } (x, y) \in R \text{ und } (y, z) \in S\},$$

Komposition von R und S.

Sei $R \subseteq A \times A$, dann ist $R^0 = \{(x, x) \mid x \in A\} = A \times A$ die identische Relation auf A, und $R^n \subseteq A \times A$ ist für $n \geq 1$ definiert durch $R^n = R^{n-1} \circ R$.

Ist $R^0 \subseteq R$, dann heißt R reflexiv; ist $R = R^{-1}$, dann heißt R symmetrisch; ist $R \circ R \subseteq R$, dann heißt R transitiv.

Die Relation

$$R^+ = \bigcup_{n \geq 1} R^n$$

heißt transitive Hülle oder transitiver Abschluss von R, und

$$R^* = \bigcup_{n \geq 0} R^n$$

heißt refelxiv-transitive Hülle oder reflexiv-transitiver Abschluss von R.

Eine Relation $f \subseteq A \times B$ heißt rechtseindeutig, falls gilt: Ist $(x, y), (x', y') \in f$ und $y \neq y'$, dann muss auch $x \neq x'$ sein. Rechtseindeutige Relationen werden Funktionen genannt. Funktionen notiert man in der Regel in der Form $f : A \to B$ anstelle von $f \subseteq A \times B$ um auszudrücken, dass die Funktion f Elementen x aus der Ausgangsmenge A jeweils höchstens ein Element $y = f(x)$ aus der Zielmenge B zuordnet.

Die Menge

$$\mathit{Def}(f) = \{x \in A \mid \text{es existiert ein } y \in B \text{ mit } f(x) = y\}$$

heißt Definitionsbereich von f.

Die Menge

$$W(f) = \{y \in B \mid \text{es existiert ein } x \in A \text{ mit } f(x) = y\}$$

heißt Wertebereich oder Wertemenge von f. Anstelle von $W(f)$ schreibt man auch $f(A)$.

Für eine Funktion f schreiben wir $f(x) = \perp$, falls f für das Argument x nicht definiert ist, d. h., falls x nicht zum Definitionsbereich von f gehört: $x \notin \mathit{Def}(f)$.

Gilt $\mathit{Def}(f) = A$, dann heißt f total; gilt $W(f) = B$, dann heißt f surjektiv.

Die Menge $B^A = \{f : A \to B \mid f \text{ total}\}$ ist die Menge aller totalen Funktionen von A nach B. Sind A und B endlich, dann ist

$$\left|B^A\right| = |B|^{|A|}.$$

Gilt für $x, x' \in \mathit{Def}(f)$ mit $x \neq x'$ auch $f(x) \neq f(x')$, dann heißt f linkseindeutig oder injektiv. Ist f total, injektiv und surjektiv, dann heißt f bijektiv.

Ist f injektiv, dann ist $f^{-1} : B \to A$, definiert durch $f^{-1}(y) = x$, genau dann, wenn $f(x) = y$ ist, die Umkehrfunktion von f.

Sei $A' \subseteq A$, dann heißt die Funktion $f_{|A'}$, definiert durch $f_{|A'}(x) = f(x)$ für alle $x \in A'$, die Einschränkung von f auf A'.

Die Signum-Funktion $\mathit{sign} : \mathbb{R} \to \{0, 1\}$ ist definiert durch

$$\mathit{sign}(x) = \begin{cases} 1, & x > 0, \\ 0, & x \leq 0. \end{cases}$$

Die Dualdarstellung $z = \mathit{dual}(n)$ einer Zahl $n \in \mathbb{N}_0$ ist gegeben durch

$$z = z_{n-1} \ldots z_1 z_0$$

mit $z_i \in \{0, 1\}$, $0 \leq i \leq n-1$ und $z_{n-1} = 1$, falls $n \geq 2$, sowie

$$n = \mathit{wert}(z) = \sum_{i=0}^{n-1} z_i \cdot 2^i.$$

log ist der Logarithmus zur Basis 2. Für $a > 0$ ist also $\log a = b$ genau dann, wenn $2^b = a$ ist.

Literaturverzeichnis

[AB02] Asteroth, A., Baier, C.: *Theoretische Informatik.* Pearson Studium, München, 2002

[BL74] Brainerd, W.S., Landweber, L.H.: *Theory of Computation.* J. Wiley & Sons, New York, 1974

[Ca94] Calude, C. S.: *Information and Randomness, 2nd Edition.* Springer, Berlin, 2010

[CDS02] Calude, C. S., Dinneen, M. J., Shu, C.-K.: Computing a Glimpse of Randomness, *Exper. Math.* **11**, 2002, 361–370

[CD07] Calude, C. S., Dinneen, M. J.: Exact Approximations of Omega Numbers, *Int. J. Bifur. Chaos* **17**, 2007, 1937–1954

[Ch66] Chaitin, G.: On the length of programs for computing binary sequences, *Journal of the ACM* **13**, 1966, 547–569

[Ch691] Chaitin, G.: On the length of programs for computing binary sequences: Statistical considerations, *Journal of the ACM* **16**, 1969, 145–159

[Ch692] Chaitin, G.: On the simplicity and speed of programs for computing infinite sets of natural numbers, *Journal of the ACM* **16**, 1969, 407–412

[Ch693] Chaitin, G.: On the difficulty of computations, *IEEE Transactions on Information Theory* **16**, 1969, 5–9

[Ch74a] Chaitin, G.: Information-theoretic computational complexity, *IEEE Transactions on Information Theory* **20**, 1974, 10–15

[Ch74b] Chaitin, G.: Information-theoretic limitations of formal systems, *Journal of the ACM* **21**, 1974, 403–424

[Ch75] Chaitin, G.: A theory of program size formally identical to information theory, *Journal of the ACM* **22**, 1975, 329–340

[Ch98] Chaitin, G.: *The Limits of Mathematics.* Springer, Heidelberg, 1998

[Ch99] Chaitin, G.: *The Unknowable.* Springer, Heidelberg, 1999

K.-U. Witt und M. E. Müller, *Algorithmische Informationstheorie*,
https://doi.org/10.1007/978-3-662-61694-9

[Ch02] Chaitin, G.: *Conversations with a Mathematician.* Springer, Heidelberg, 2002

[Ch07] Chaitin, G.: *Meta Maths: The Quest of Omega.* Atlantic Books, London, 2007

[D06] Dankmeier, W.: *Grundkurs Codierung, 3. Auflage;* Vieweg, Wiesbaden, 2006

[Ho11] Hoffmann, D. W.: *Grenzen der Mathematik.* Spektrum Akademischer Verlag, Heidelberg, 2011

[HS01] Homer, S., Selman, A.L.: *Computability and Complexity Theory.* Springer, New York, 2001

[HMU13] Hopcroft, J. E., Motwani, R., Ullman, J. D.: *Introduction to Automata Theory, Languages, and Computation, 3rd Edition.* Pearson International Edition, 2013

[Hr14] Hromkrovič, J.: *Theoretische Informatik–Formale Sprachen, Berechenbarkeit, Komplexittstheorie, Algorithmik, Kommunikation und Kryptografie, 5. Auflage.* Springer Vieweg, Wiesbaden, 2014

[Kol65] Kolmogorov, A.: Three approaches for defining the concept of information quantity, *Problems of Information Transmission* **1**, 1965, 1–7

[Kol68] Kolmogorov, A.: Logical basis for information theory and probabilistic theory, *IEEE Transactions on Information Theory* **14**, 1968, 662–664

[Kol69] Kolmogorov, A.: On the logical foundations of information theory and probability theory, *Problems of Information Transmission* **5**, 1969, 1–4

[Koz97] Kozen, D. C.: *Automata and Computability.* Springer-Verlag, New York, 1997

[LP98] Lewis, H. R., Papadimitriou, C. H.: *Elements of the Theory of Computation, 2nd Edition.* Prentice-Hall, Upper Saddle River, NJ, 1998

[LV93] Li, M., Vitànyi, P.: *Introduction to Kolmogorov Complexity and Its Applications, 3rd Edition.* Springer, New York, 2008

[LV08] Li, M., Vitànyi, P.: A New Approach to Formal Language Theory by Kolmogorov Complexity, *arXiv:cs/0110040v1* [$cs.CC$] *18 oct 2001*

[MS08] Mackie, I., Salomon, D.: *A Concise Introduction to Data Compression.* Springer, London, 2008

[M66] Martin-Löf, P.: The Definition of Random Sequences, *Information and Control* **9**, 602–619, 1966

[P94] Papadimitriou, C. H.: *Computational Complexity.* Addison-Wesley, Reading, MA, 1994

[Ra62] Radó, T.: On non-computable functions, *The Bell System Technical Journal*, **41**, 1962, 877–884

[Ro67] Rogers, H., Jr.: *The Theory of Recursive Functions and Effective Computability.* McGraw-Hill, New York, 1967

[Sa05] Salomon, D.: *Coding for Data and Computer Communications.* Springer, New York, NY, 2005

[SM10] Salomon, D., Motta, G.: *Handbook of Data Compression.* Springer, London, 2010

[Schö09] Schöning, U.: *Theoretische Informatik kurzgefasst, 5. Auflage.* Spektrum Akademischer Verlag, Heidelberg, 2009

[Schu03] Schulz, R.-H.: *Codierungstheorie–Eine Einführung, 2. Auflage.* Vieweg, Braunschweig/Wiesbaden 2003

[Si06] Sipser, M.: *Introduction to the Theory of Computation, second Edition, International Edition.* Thomson, Boston, MA, 2006

[So64a] Solomonoff, R. J.: A formal theory of inductive Inference, Part I, *Information and Control* **7**, 1964, 1–22

[So64b] Solomonoff, R. J.: A formal theory of inductive Inference, Part II, *Information and Control* **7**, 1964, 224–254

[VW16] Vossen, G., Witt, K.-U..: *Grundkurs Theoretische Informatik, 6. Auflage.* Springer Vieweg, Wiesbaden, 2016

[Weg05] Wegener, I.: *Complexity Theory.* Springer Verlag, Berlin, 2005

[Wei87] Weihrauch, K.: *Computability.* Springer-Verlag, Berlin, 1997

Index

K.-U. Witt und M. E. Müller, *Algorithmische Informationstheorie*,
https://doi.org/10.1007/978-3-662-61694-9