Einführung in die Informatik

nach der Vorlesung

von Prof. Dr. Marek Karpinksi

ausgearbeitet von

Mathias Hauptmann

Kai Werther

Thorsten Werther

Neue Auflage 2005.

Universität Bonn
Römerstraße 164
53117 Bonn

Vorwort

Das Skript "Einführung in die Informatik" ist im Herbst 1992 aus den Vorlesungen "Informatik I" (Wintersemester 91/92) und "Informatik II" (Sommersemester 92) von Herrn Prof. Dr. Marek Karpinski am Institut für Informatik der Universität Bonn entstanden.

Die Gliederung dieses Skriptes entspricht der inhaltlichen Struktur des Vorlesungsstoffes und weicht daher an einigen Stellen von dem didaktischen Aufbau der Vorlesung ab. Damit ist der Stoff auch ohne Kenntnis der eigentlichen Vorlesung zu erarbeiten. **Prüfungsrelevant bleibt aber die Vorlesung von Prof. Dr. Karpinski!**

Weiterhin werden teilweise geringfügig abweichende Notationen verwendet. So benutzen wir die in der Literatur übliche Bezeichnung für Zustandsübergangsfunktionen (δ statt M), während wir den Buchstaben M für die Bezeichnung von Turing-Maschinen reservieren. Desweiteren sei bemerkt, daß weitgehend auf die Unterstreichungen oder Großschreibungen der Vorlesung verzichtet wird, um die Lesbarkeit des Textes zu erhöhen.

Das Skript beginnt mit einem vorbereitenden Kapitel über mathematische Grundlagen, dessen Stoffauswahl weitgehend durch die Bedürfnisse der nachfolgenden Kapitel bestimmt ist.

Kapitel 2 gibt eine erste Einführung in die Theorie formaler Sprachen. Wir untersuchen die Struktur regulärer und ω–regulärer Mengen sowie den endlichen Automaten als das korrespondierende Maschinenmodell.

In den Kapiteln 3 und 4 werden wir durch die Einführung grundlegender Maschinenmodelle den Berechnungsbegriff präzisieren und einen ersten Einblick in die Komplexitätstheorie geben. Wir werden zunächst die klassischen Modelle der Turingmaschine und der Random Access Maschine behandeln und anschließend den neueren Entwicklungen in der Komplexitätstheorie durch die Diskussion paralleler und randomisierter Berechnungsmodelle Rechnung tragen.

Das Kalkül der Prädikatenlogik bildet den Grundstock für weite Teilgebiete der Informatik, so z.B. der künstlichen Intelligenz im Rahmen des maschinellen Beweisens oder auch der Semantik von Programmiersprachen. Wir werden daher in Kapitel 5 eine Einführung in die Prädikatenlogik geben.

In Kapitel 6 werden wir einen Abstecher in die Praxis unternehmen und einige grundlegende Datentypen und Algorithmen behandeln.

Schließlich diskutieren wir in Kapitel 7 die verschiedenen Ansatzpunkte zur Semantik von Programmiersprachen und werden dies anhand der einfachen Programmiersprache TINY verdeutlichen.

Letztlich möchten wir noch Herrn Udo Kalker für die Bereitstellung seiner ausgearbeiteten Vorlesungsmitschrift danken.

Bonn, im Dezember 1992

K. Werther
T. Werther

Vorwort zur neuen Auflage 2005

Das Skript wurde ausgehend von der Vorlesung "Informatik III" (Wintersemester 2003/04) von Herrn Prof. Dr. Marek Karpinski am Institut für Informatik der Universität Bonn um das Kapitel 2.4 "Probabilistische Endliche Automaten" erweitert.

Bonn, im Oktober 2005 M. Hauptmann

Inhaltsverzeichnis

1 Mathematische Grundlagen **1**

1.1 Grundlagen der Mengenlehre 1

1.1.1 Relationen und Funktionen 4

1.1.2 Die natürlichen Zahlen und vollständige Induktion 6

1.1.3 Diagonalisierungsverfahren 8

1.1.4 Das Russellsche Paradoxon in der Mengenlehre 9

1.2 Grundlagen der Graphentheorie 10

1.3 Aussagenlogik und Boolesche Funktionen 13

1.3.1 Simulation Boolescher Funktionen durch Schaltungen 15

1.4 Boolesche Algebra 17

1.5 Asymptotisches Wachstum von Funktionen 19

1.6 Elementare Wahrscheinlichkeitsrechnung 20

2 Automatentheorie **26**

2.1 Formale Sprachen 26

2.2 BNF: ein syntaktisches Beschreibungsmittel 27

2.2.1 Interpretation von Wörtern 29

2.2.2 Erweiterte BNF 30

2.3 Endliche Automaten und reguläre Ausdrücke 32

2.3.1 Reguläre Ausdrücke 32

2.3.2 Endliche Automaten 34

2.3.3 Kleenesche Sätze 38

2.3.4 Struktur von regulären Mengen 41
2.3.5 Struktur ω–regulärer Mengen 41
2.4 Probabilistische Endliche Automaten 51
2.4.1 Probabilistische Automaten 51
2.4.2 Randomisierte Automaten 53

3 Maschinenmodelle, Berechenbarkeit und Komplexität 56

3.1 Turing-Maschinen 56
3.2 Unentscheidbarkeit 60
3.2.1 Das Halteproblem 60
3.2.2 Weitere unentscheidbare Probleme 62
3.3 Elementare Komplexitätstheorie 63
3.3.1 Off-line Turing-Maschine 65
3.3.2 Komplexitätsklassen 66
3.4 Random Access Maschinen 69
3.4.1 Semantik von RAM-Programmen 72
3.4.2 Komplexitätsmaße der RAM 75
3.4.3 Vergleich zwischen TM und RAM 76

4 Parallele und Randomisierte Maschinenmodelle 79

4.1 Parallele Random Access Maschine 79
4.2 Schaltkreise 83
4.2.1 Uniforme Schaltkreisfamilien 85
4.3 Randomisierte Schaltkreise 87
4.4 Randomisierte Testalgorithmen 88
4.4.1 Nulltest für multivariate Polynome 88
4.4.2 Test für Matrizenmultiplikation 91

5 Prädikatenlogik 93

5.1 Syntax der Prädikatenlogik 93

5.2 Semantik der Prädikatenlogik 95
5.3 Erfüllbarkeit und Allgemeingültigkeit 97
5.4 Quantorengesetze und Substitution 100
5.4.1 Freie und gebundene Variablen 100
5.4.2 Substitution 101
5.4.3 Quantorengesetze 105
5.5 Logisches Schließen 106
5.6 Die Unentscheidbarkeit der Prädikatenlogik 108

6 Datenstrukturen und Algorithmen 111
6.1 Elementare abstrakte Datentypen 111
6.2 Lineare Datentypen 113
6.2.1 Listen 115
6.2.2 Keller 120
6.2.3 Schlange 121
6.3 Datentypen aus der Graphentheorie 122
6.3.1 Bäume 123

7 Semantik von Programmiersprachen und das λ-Kalkül 125
7.1 Die Sprache "TINY" 125
7.2 Semantik von TINY 126
7.3 Das λ - Kalkül 130
7.4 Denotationelle Semantik von TINY 134
7.5 Standard Semantik 136

Literaturverzeichnis 139

Kapitel 1

Mathematische Grundlagen

1.1 Grundlagen der Mengenlehre

Die Mengenlehre hat in der Mathematik eine grundlegende und zentrale Bedeutung, da sich jede mathematische Aussage in der Sprache der Mengenlehre formulieren läßt. In diesem Abschnitt möchten wir einige der grundlegenden Tatsachen der Mengenlehre beschreiben, verzichten allerdings auf eine formale axiomatische Konstruktion der Mengenlehre. Für uns ist die Mengenlehre vielmehr eine geignete Sprache, mit Hilfe derer wir Zusammenhänge ausdrücken werden.

G. Cantor (siehe z.B. [Fel78], dort auch weiteres über die Mengenlehre) folgend möchten wir informell eine *(abstrakte) Menge* wie folgt definieren:

> "Unter einer 'Menge' M verstehen wir jede Zusammenfassung von bestimmten wohlunterschiedenen Objekten m unserer Anschauung oder unseres Denkens (welche die 'Elemente' von M genannt werden) zu einem Ganzen"

So bezeichnen wir z.B. die Zusammenfassung aller Punkte auf einer Geraden, die Zusammenfassung aller Geraden in einer Ebene, oder auch die Zusammenfassung aller Hörer der Informatik I Vorlesung als Mengen.

Ein Objekt, das zu einer Menge A gehört, wird als ein *Element* von A bezeichnet. Falls x ein Element von A ist, so schreiben wir hierfür $x \in A$, anderenfalls schreiben wir $x \notin A$. Falls jedes Element einer Menge A in einer Menge B enthalten ist, so nennen wir A eine *Teilmenge* von B, bzw. B eine *Obermenge* von A und notieren dies durch $A \subseteq B$.

Wir benutzen die folgenden Notationen:

- Falls alle Objekte $x_1, x_2, \ldots, x_n$ Elemente der Menge A sind, so schreiben wir auch $x_1, x_2, \ldots, x_n \in A$ statt $x_1 \in A, x_2 \in A, \ldots, x_n \in A$.
- Falls keines der Objekte $x_1, x_2, \ldots, x_n$ ein Element der Menge A ist, so schreiben wir $x_1, x_2, \ldots, x_n \notin A$.
- Die leere Menge, d.h. die Menge, die kein Element enthält, bezeichnet man mit dem Symbol $\emptyset$.
- Für die folgenden, häufig benutzten Mengen führen wir die folgenden Bezeichnungen ein:

$\mathbb{N}$	Menge der natürlichen Zahlen.
$\mathbb{N}_0$	Menge der natürlichen Zahlen einschließlich der Null.
$\mathbb{Z}$	Menge der ganzen Zahlen.
$\mathbb{R}$	Menge der reellen Zahlen.
$\mathbb{B}$	Menge der Wahrheitswerte Wahr ($\underline{\mathrm{W}}$) und Falsch ($\underline{\mathrm{F}}$).

Wir werden die Menge der natürlichen Zahlen später axiomatisch einführen.

Es gibt mehrere Möglichkeiten eine Menge anzugeben:

- Falls eine Menge nur endlich viele Elemente umfaßt, so läßt sich die Menge durch die Auflistung sämtlicher Elemente angeben (*in extension*), wir schreiben dann z.B.

$$A = \{1, 2, 5, 7\}.$$

- Falls eine Menge unendlich viele Elemente umfaßt (und damit nicht vollständig aufgelistet werden kann), so kann man sich auf die Angabe einiger Elemente dieser Menge beschränken, falls die Beschaffenheit der fehlenden Elemente vom Kontext her klar ist. Wir können z.B die Menge U der ungeraden natürlichen Zahlen folgendermaßen angeben:

$$U = \{1, 3, 5, 7, 9, 11, \ldots\}.$$

- Falls sich die Elemente einer Menge A durch Transformation der Elemente einer zweiten Menge B beschreiben lassen, so ist A durch die Angabe von B und der Transformation festgelegt. So läßt sich z.B. die Menge G der geraden natürlichen Zahlen durch $f(n) = 2 \cdot n$ für $n \in \mathbb{N}$ charakterisieren. Hierfür schreiben wir abkürzend

$$G = \{2n \,|\, n \in \mathbb{N}\}.$$

- Falls sich die Elemente einer Menge A genau diejenigen Elemente einer Menge B sind, die eine gewisse Eigenschaft P erfüllen, so schreiben wir (*in intension*)

$$A = \{x \in B \,|\, P(x)\}$$

als die Menge aller $x \in B$ für die $P(x)$ wahr ist. Falls B sich aus dem Kontext ergibt, so verzichten wir auf die Angabe von B und schreiben kurz $A = \{x \,|\, P(x)\}$. So ist z.B.

$$P = \{k \in \mathbb{N} \,|\, k \text{ hat außer 1 und } k \text{ keine weiteren Teiler}\}$$

die Menge aller Primzahlen einschließlich der Eins.

Wir geben nun die grundlegende Terminologie der Mengenlehre an:

Definition 1.1 Seien A, B und X Mengen.

i) Falls A nur aus endlich vielen Elementen besteht, so heißt A *endlich*. Die Anzahl der Elemente einer Menge A heißt die *Kardinalität* oder auch *Mächtigkeit* von A und wird mit $|A|$ notiert. Die Kardinalität der leeren Menge ist Null ($|\emptyset| = 0$). Zwei endliche Mengen A und B sind von gleicher Kardinalität, falls $|A| = |B|$ gilt.

ii) Unter der *Vereinigungsmenge* $A \cup B$ von A und B versteht man die Zusammenfassung der Elemente von sowohl A als auch B, d.h

$$A \cup B := \{ x \,|\, x \in A \text{ oder } x \in B \}.$$

Für endliche Mengen gilt $|A \cup B| \leq |A| + |B|$.

iii) Unter der *Schnittmenge* $A \cap B$ von A und B versteht man die Menge der Elemente, die in beiden Mengen vorkommen, d.h.

$$A \cap B := \{ x \,|\, x \in A \text{ und } x \in B \}.$$

iv) Unter der *Differenz* $A \setminus B$ von A und B versteht man die Menge der Elemente, die nur in A vorkommen, d.h.

$$A \setminus B := \{ x \mid x \in A \text{ und } x \notin B \}.$$

v) Unter der *symmetrischen Differenz* $A \triangle B$ von A und B versteht man die Menge der Elemente, die entweder in A oder B vorkommen, nicht jedoch in beiden Mengen, d.h.

$$A \triangle B := (A \setminus B) \cup (B \setminus A) = (A \cup B) \setminus (A \cap B).$$

vi) Unter dem *Komplement* $\bar{A}$ einer Teilmenge A von X versteht man die Menge $B = \{a \in X \mid a \notin A\}$. Offensichtlich ist $\overline{(\bar{A})} = A$ (die Notation $\bar{A}$ darf nur benutzt werden, wenn die Menge X aus dem Kontext ersichtlich ist).

vii) A und B heißen *elementfremd* oder auch *disjunkt*, wenn sie kein gemeinsames Element besitzen, wenn also $A \cap B = \emptyset$ gilt. Falls A und B endlich sind, so gilt in diesem Fall $|A \cup B| = |A| + |B|$.

viii) A und B sind gleich ($A = B$), falls $A \subseteq B$ und $B \subseteq A$ gelten, d.h. alle Elemente, die in A enthalten sind, auch in B vorkommen und umgekehrt.

ix) Unter der *Potenzmenge* von A versteht man die Menge aller Teilmengen der Menge A. Wir notieren: $\mathfrak{P}(A) = \{X \mid X \subseteq A\}$. Für endliche Mengen A gilt $|\mathfrak{P}(A)| = 2^{|A|}$.

x) Seien $A_1, A_2, \ldots, A_n$ für $n \in \mathbb{N}$ beliebige Mengen. Dann heißt die Menge $A_1 \times A_2 \times \cdots \times A_n := \{(x_1, x_2, \ldots, x_n) \mid x_1 \in A_1, x_2 \in A_2, \ldots, x_n \in A_n\}$ (n-stelliges) *kartesisches Produkt* von $A_1, \ldots, A_n$. Ein Element $(x_1, x_2, \ldots, x_n)$ dieser Menge bezeichnen wir als n-Tupel. Falls die Mengen A_1 bis A_n endlich sind, so gilt $|A_1 \times \cdots \times A_n| = |A_1| \cdot \cdots \cdot |A_n|$, falls sie alle gleich sind, also $A_1 = A_2 = A_3 = \ldots = A_n = A$ gilt, so schreiben wir A^n statt $A_1 \times A_2 \times \ldots \times A_n$.

Es sei bemerkt, daß ein Tupel $(a_1, \ldots, a_n)$ nicht die Menge $\{a_1, \ldots, a_n\}$ ist. Die Elemente eines jeden Tupels sind im Gegensatz zu den Elementen einer Menge durch ihre Reihenfolge bestimmt. *2-Tupel* (a, b) bezeichnet man dabei auch als *geordnete Paare*, während *3-Tupel* (a, b, c) auch Tripel heißen. Formal kann ein n-Tupel $a = (a_1, \ldots, a_n)$ mit der Menge $A = \{\{a_1\}, \{a_1, a_2\}, \ldots, \{a_1, \ldots, a_n\}\}$ identifiziert werden. Die i-te Komponente a_i von a ist dann die Mengendifferenz zwischen dem (eindeutig bestimmten) Element $\{a_1, \ldots, a_i\} \in A$ der Kardinalität i und dem Element $\{a_1, \ldots, a_{i-1}\}$ der Kardinalität $i - 1$.

Die oben eingeführten Operationen auf Mengen gehorchen den folgenden Regeln.

Lemma 1.2 Seien A, B, C beliebige Mengen. Dann gelten:

i) $A \cup B = B \cup A$ (Kommutativgesetze)
$A \cap B = B \cap A$

ii) $A \cup (B \cup C) = (A \cup B) \cup C$ (Assoziativgesetze)
$A \cap (B \cap C) = (A \cap B) \cap C$

iii) $A \cup (A \cap B) = A$ (Absorptionsgesetze)
$A \cap (A \cup B) = A$

iv) $A \cup (B \cap C) = (A \cup B) \cap (A \cup C)$ (Distributivgesetze)
$A \cap (B \cup C) = (A \cap B) \cup (A \cap C)$

Für kartesische Produkte gelten die folgenden Rechenregeln. Dazu seien A_1, A_2, B_1, B_2 und B beliebige Mengen.

1. $(A_1 \cup A_2) \times B = (A_1 \times B) \cup (A_2 \times B)$
2. $(A_1 \setminus A_2) \times B = (A_1 \times B) \setminus (A_2 \times B)$
3. $(A_1 \cap A_2) \times (B_1 \cap B_2) = (A_1 \times B_1) \cap (A_2 \times B_2)$
4. Seien $A_1, A_2, B_1, B_2 \neq \emptyset$. Dann gilt $(A_1 \times B_1) = (A_2 \times B_2)$ genau dann, wenn $A_1 = A_2$ und $B_1 = B_2$ sind.

Die *Projektion* einer Teilmenge R des kartesischen Produktes $A \times B$ auf A ist die Menge $\{a \in A \mid \exists\, b \in B \text{ mit } (a,b) \in R\}$. Die *Bildmenge* $R(a)$ von $a \in A$ ist die Menge $\{b \in B \mid (a,b) \in R\}$. Analog ist die *Urbildmenge* $R^{-1}(b)$ von b die Menge $\{a \in A \mid (a,b) \in R\}$.

1.1.1 Relationen und Funktionen

Teilmengen des kartesischen Produktes $A \times B$ zweier Mengen A und B ermöglichen es uns, Beziehungen zwischen den Elementen der Mengen A und B auszudrücken. Formal ist eine *Relation* R über A und B eine beliebige Teilmenge $R \subseteq A \times B$. R heißt auch Relation auf A und B. Liegt ein Paar $(a,b) \in A \times B$ in der Menge R, so benutzen wir statt $(a,b) \in R$ auch die Notation $a\ R\ b$; andernfalls ist $(a,b) \notin R$ und wir schreiben $a \not R\, b$.

Der *Definitionsbereich* $DB(R)$ einer Relation $R \subset A \times B$ ist die Projektion von R auf A, der *Wertebereich* $WB(R)$ ist die Projektion von R auf B. Nun noch ein paar weitere Definitionen: Eine Relation R heißt

- *linkstotal*, falls $DB(R) = A$ gilt.
- *rechtstotal*, falls $WB(R) = B$ gilt.
- *linkseindeutig*, falls $R^{-1}(b)$ für alle $b \in B$ höchstens ein Element enthält.
- *rechtseindeutig*, falls $R(a)$ für alle $a \in A$ höchstens ein Element enthält.

Die Relation $\tilde{R} \subset B \times A$ mit $(b,a) \in \tilde{R}$ genau dann, wenn $(a,b) \in R$, heißt die zu R *inverse* Relation und wird mit R^{-1} bezeichnet.

Für Relation $R \subseteq A^2$ sind noch die folgenden Bezeichnungen üblich. R heißt

- *reflexiv*, falls für alle $a \in A$: $(a,a) \in R$ gilt,
- *nicht reflexiv*, falls für wenigstens ein $a \in A$: $(a,a) \notin R$ gilt,
- *irreflexiv*, falls für alle $a \in A$: $(a,a) \notin R$ gilt,
- *symmetrisch* , falls für alle $(a,b) \in R$ auch $(b,a) \in R$ gilt.
- *asymmetrisch*, falls aus $(a,b) \in R$ auch $(b,a) \notin R$ folgt.
- *antisymmetrisch*, falls aus $(a,b),(b,a) \in R$ die Gleichheit $a = b$ folgt.
- *transitiv* , falls aus $(a,b) \in R$ und $(b,c) \in R$ auch $(a,c) \in R$ folgt.
- *total*, falls für alle $a,b \in A$ gilt, daß $(a,b) \in R$ oder $(b,a) \in R$ ist.

Die kleinste Obermenge $R' \supseteq R$ von R, die transitiv ist, nennt man *transitive Hülle* von R. Sie wird auch mit R^T bezeichnet. Man kann R^T auch schreiben als

$$R^T = \{(a,b) \in A^2 \mid \exists\, n \geq 2, a = x_1, x_2, \ldots, x_{n-1}, x_n = b \,\forall\, 1 \leq i < n : (x_i, x_{i+1}) \in R\}$$

Relationen mit bestimmten Eigenschaften haben gesonderte Namen.

- Eine reflexive, symmetrische und transitive Relation $R \subseteq A^2$ heißt *Äquivalenzrelation.*
- Eine reflexive, antisymmetrische und transitive Relation $R \subseteq A^2$ heißt *Ordnung.*
- Eine reflexive, antisymmetrische, transitive und totale Relation heißt *Totalordnung* oder auch totale (lineare, konvexe) Ordnung. Ist die Ordnung nicht total, d.h. existieren $a,b \in A$ mit $(a,b),(b,a) \notin R$, so ist R eine *partielle* Ordnung.
- Eine linkstotale und rechtseindeutige Relation $R \subseteq A \times B$ heißt *Funktion* oder auch *Abbildung* von A nach B. Wir schreiben für $(a,b) \in R$ statt $R(a) = \{b\}$ dann auch $R(a) = b$ und statt $R \subseteq A \times B$ auch $R : A \to B$.

Funktionen sind besonders wichtige Objekte in der Mathematik. Daher haben sich für Funktionen gesonderte Bezeichnungen eingebürgert. Sei f eine Funktion von A nach B. Dann heißt f

- *surjektiv* (engl.: onto) oder auch *Surjektion* (von A auf B), falls f rechtstotal ist.
- *injektiv* oder auch *Injektion* (von A auf B), falls f linkseindeutig ist.
- *bijektiv* (engl.: one-one) oder auch *Bijektion* (von A auf B), falls f injektiv (linkseindeutig) und surjektiv (rechtstotal) ist.

Die Umkehrrelation f^{-1} einer bijektiven Funktion f von A nach B ist wieder eine Funktion (von B nach A). f^{-1} heißt in diesem Falle *Umkehrfunktion* von f (und ist ebenfalls bijektiv).

Eine nichtnotwendigerweise linkstotale, jedoch rechtseindeutige Funktion nennen wir *partielle Funktion*, da sie nur auf Teilen des möglichen Definitionsbereiches definiert ist. Zur Betonung des Unterschiedes nennen wir (linkstotale) Funktionen auch *totale Funktionen.*

Nun ist auch die folgende Definition möglich: zwei nicht notwendigerweise endliche Mengen A und B besitzen die gleiche Mächtigkeit, falls es eine Bijektion f von A nach B gibt (f^{-1} ist dann eine Bijektion von B nach A).

Wir werden diese Definition in abgewandelter Form in Abschnitt 1.1.3 benutzen um zu zeigen, daß die Mächtigkeit der natürlichen Zahlen mit der der rationalen Zahlen, nicht aber mit der der reellen Zahlen übereinstimmt.

1.1.2 Die natürlichen Zahlen und vollständige Induktion

In diesem Abschnitt werden wir die Menge $\mathbb{N}$ der natürlichen Zahlen axiomatisch einführen. Diese axiomatische Einführung geht auf PEANO zurück. Grundlegend hierfür ist die folgende Notation.

Sei $S \subset X$ eine Menge. Dann heißt die durch $^+$ notierte Funktion auf S *Nachfolgefunktion.* Zu $s \in S$ heißt $s^+ \in X$ Nachfolger von s.

Nun können wir die natürlichen Zahlen auch axiomatisch definieren.

Definition 1.3 Die eindeutig bestimmte Menge S, die die nachfolgenden fünf PEANO-Axiome erfüllt, heißt *Menge der natürlichen Zahlen* und wird mit $\mathbb{N}$ bezeichnet.

Axiom 1 $1 \in S$ ist eine natürliche Zahl.

Axiom 2 Wenn $n \in S$ eine natürliche Zahl ist, so ist auch $n^+ \in S$ eine natürliche Zahl, d.h. $^+$ ist eine Funktion von S nach S.

Axiom 3 Es gibt kein $n \in S$, dessen Nachfolger $n^+ = 1$ ist.

Axiom 4 Falls $n^+ = m^+$ gilt, so folgt $n = m$, d.h. die Funktion $^+$ ist eine Injektion.

Axiom 5 Falls T eine Menge ist, die 1 enthält ($1 \in T$) und mit n auch den Nachfolger n^+ ($n \in T \Rightarrow n^+ \in T$), so gilt $S \subseteq T$.

Es sei bemerkt, daß die Eindeutigkeit von $\mathbb{N}$ (und somit die Wohldefiniertheit!) aus dem Axiom 5 folgt. Denn, wenn es noch eine zweite Menge $S \neq \mathbb{N}$ gäbe, die die Peano-Axiome erfüllte, dann wäre nach Axiom 1 und Axiom 2 sowohl S als auch $\mathbb{N}$ eine erlaubte Menge für die Menge T aus Axiom 5. Somit folgt $S \subseteq \mathbb{N} \subseteq S$ und somit $S = \mathbb{N}$ im Widerspruch zur Annahme.

Außerdem sei darauf hingewiesen, daß sich Axiom 5 auch durch folgendes äquivalentes Axiom ersetzen läßt.

Axiom 5' Falls T eine Teilmenge von S ist, die 1 enthält ($1 \in T$) und mit n auch den Nachfolger n^+ ($n \in T \Rightarrow n^+ \in T$), so gilt $S = T$.

Man nennt das Axiom 5 bzw. 5' auch *Induktionsaxiom.*

Um eine mathematisch greifbare Definition der natürlichen Zahlen zu haben, werden wir uns jetzt einer Konstruktion für die Menge der natürlichen Zahlen zuwenden, die von VON NEUMANN stammt.

Sei X ein Menge. Dann ist der *Nachfolger* X^+ von X definiert durch $X^+ = X \cup \{X\}$. Falls X eine endliche Menge ist, ist X^+ offensichtlich von X verschieden, da $|X^+| > |X|$ ist.

Startend mit der leeren Menge $X = \emptyset$, können wir die Nachfolger $X^+ = X \cup \{X\} = \emptyset \cup \{\emptyset\} = \{\emptyset\}$, $(X^+)^+ = \{\emptyset\} \cup \{\{\emptyset\}\} = \{\emptyset, \{\emptyset\}\}$, usw. bilden. VON NEUMANN hat nun die natürlichen Zahlen

als diese Mengen definiert, bzw. diese Mengen mit den Symbolen für die natürlichen Zahlen belegt. Mit dieser Notation gilt: $0 := \emptyset$, $1 := 0^+ = \{\emptyset\} = \{0\}$, $2 := 1^+ = \{0, 1\}$, usw.. Alle so entstehenden Mengen sind endlich und daher alle verschieden.

Damit haben wir den folgenden Satz.

Satz 1.4 Die Menge N der oben konstruierten Mengen $(\{\emptyset\}, \{\emptyset, \{\emptyset\}\}, \ldots)$ ausschließlich der Menge $\emptyset$, bildet die Menge $\mathbb{N}$ der natürlichen Zahlen.

BEWEIS: Wir müssen die fünf PEANO-Axiome nachweisen. Axiome 1 und 2 folgen direkt aus der Definition. Für die obige Konstruktion folgen Axiom 3 und 4 schon aus den Axiomen 1 und 2 und $|X^+| > |X|$. Wir müssen jetzt noch Axiom 5' nachweisen. Dazu nehmen wir an, daß eine echte Teilmenge T von N existiert, die die Axiome 1 und 2 erfüllt. Sei $n \in N \setminus T$ die kleinste Menge aus N, die nicht in T enthalten ist. Da n in N ist, muß es ein $m \in N$ mit $m^+ = n$ geben. Da n kleinstmöglich gewählt war, folgt $m \in T$ und mit Axiom 2 für T auch $n \in T$. Dies ist ein Widerspruch und Axiom 5' bewiesen. □

Für die natürlichen Zahlen gilt auch das folgende Lemma.

Lemma 1.5 Sei $n \in \mathbb{N}$. Dann ist entweder $n = 1$ oder es gibt genau einen Vorgänger $m \in \mathbb{N}$ von n, der $m^+ = n$ erfüllt.

In Zukunft bezeichnen wir den Nachfolger n^+ von n mit $n+1$ bzw. $\mathrm{succ}\,(n)$ und den nach obigem Lemma eindeutigen Vorgänger von $n > 1$ mit $n - 1$ oder $\mathrm{pred}\,(n)$.

Nun wenden wir uns dem Beweisprinzip der *vollständigen Induktion* zu. Es dient dazu ganze Mengen von Aussagen $\{A_n\}_{n\in\mathbb{N}}$ zu beweisen. Die Korrektheit des Beweisprinzips beruht auf Axiom 5'. Demnach sind zwei Schritte durchzuführen:

1. *Induktionsverankerung* oder Basisschritt (engl.: base step): Zeige die Gültigkeit der Aussage A_1.

2. *Induktionsschritt* (engl.: induction step): Zeige, daß aus der Gültigkeit des Aussage A_n (die Annahme der Gültigkeit dieser Aussage nennt man *Induktionsannahme* (IA)) die Gültigkeit der Aussage A_{n+1} folgt.

Sei T die Menge der Indizes der gültigen Aussagen aus $\{A_n\}_{n\in\mathbb{N}}$. T erfüllt die Bedingungen des Axioms 5'. Daraus folgt $T = \mathbb{N}$ und die Gültigkeit jeder Aussage aus $\{A_n\}$.

Durch Transformierung der Indizes kann mit diesem Verfahren die Gültigkeit von Mengen von Aussagen der Form $\{A_{f(n)}\}_{n\in\mathbb{N}}$ gezeigt werden.

Wir wollen das Prinzip der vollständigen Induktion an zwei Beispielen veranschaulichen.

Beispiel 1.6 Für alle natürlichen Zahlen n gilt:

$$\sum_{i=1}^{n} i = n(n+1)/2. \tag{1.1}$$

Um diese Aussage (bzw. Menge von Aussagen) mittels vollständiger Induktion zu beweisen, müssen wir die Induktionsverankerung und den Induktionsschritt beweisen. Die Induktionsverankerung ist trivial ($1 = 1$); für den Induktionsschritt nehmen wir an, daß für ein $n \in \mathbb{N}$ die Gleichung (1.1) erfüllt ist (Induktionsannahme). Für $n+1$ gilt dann:

$$\begin{aligned} \sum_{i=0}^{n+1} i &= \sum_{i=0}^{n} i + (n+1) \\ &\overset{\text{(IA)}}{=} \frac{n(n+1)}{2} + (n+1) = \frac{n(n+1) + 2(n+1)}{2} = \frac{(n+1)(n+2)}{2}. \end{aligned}$$

Aus der Korrektheit von Induktionsverankerung und Induktionsschritt folgt, wie oben bereits erwähnt, die Gültigkeit von (1.1) für alle natürlichen Zahlen.

Beispiel 1.7 Für alle natürlichen Zahlen $n \geq 5$ gilt

$$n^2 < 2^n. \tag{1.2}$$

Die Induktionsverankerung ist wieder trivial, da $25 < 32$ gilt. Sei die Aussage (1.2) für $n \geq 5$ gültig, dann gilt für $n+1$ wegen

$$2^{n+1} = 2 \cdot 2^n \overset{\text{(IA)}}{>} 2n^2 \overset{n \geq 5}{\geq} n^2 + 2n + 1 = (n+1)^2$$

ebenfalls die Gültigkeit von (1.2) und somit ist die Aussage für alle natürlichen Zahlen $n \geq 5$ bewiesen.

1.1.3 Diagonalisierungsverfahren

Wir werden uns jetzt einem anderen wichtigen Beweisverfahren in der Mathematik zuwenden, das in der theoretischen Informatik eine hervorragende Rolle spielt. Dies ist das CANTOR*sche Diagonalisierungsverfahren*. Wir werden in diesem Abschnitt die folgenden beiden Sätze beweisen. Besonders der Beweis von Satz 1.9 wird uns in anderer Form noch einmal wieder begegnen.

Satz 1.8 Die Mächtigkeit der natürlichen Zahlen ist gleich der Mächtigkeit der rationalen Zahlen.

Satz 1.9 Die Mächtigkeit der natürlichen Zahlen ist kleiner als die Mächtigkeit der reellen Zahlen.

Wir beginnen mit dem Beweis von Satz 1.8. Zunächst wollen wir jedoch folgendes feststellen:

Zwei Mengen A und B haben die gleiche Mächtigkeit, falls es surjektive Abbildungen $f_A : A \to B$ und $f_B : B \to A$ gibt. Es gilt dann nämlich $|A| \leq |B| \leq |A|$ und damit Gleichheit.

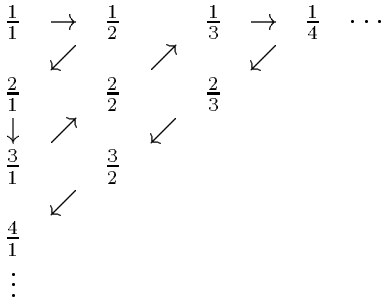
BEWEIS: (von Satz 1.8)
Wir geben zwei surjektive Abbildungen $f_N : \mathbb{N} \to \mathbb{Q}$ und $f_Q : \mathbb{Q} \to \mathbb{N}$ an. Wir definieren

$$f_Q(q) = \begin{cases} q & \text{falls } q \in \mathbb{N} \text{ ist} \\ 1 & \text{falls } q \notin \mathbb{N} \text{ ist} \end{cases}.$$

Offensichtlich ist f_Q surjektiv, da $\mathbb{N} \subset \mathbb{Q}$ ist. f_N setzen wir aus zwei surjektiven Funktionen zusammen (die Komposition surjektiver Funktionen ist natürlich wieder surjektiv). Dazu seien $g : \mathbb{N} \to \mathbb{Z}$ und $f_Z : \mathbb{Z} \to \mathbb{Q}$. Wir definieren g durch

$$g(n) = \begin{cases} (n-1)/2 & \text{falls } n \text{ ungerade} \\ -n/2 & \text{falls } n \text{ gerade} \end{cases} .$$

Offensichtlich ist g surjektiv. f_Z bildet 0 auf 0 ab, $\mathbb{Z}_{>0}$ auf $\mathbb{Q}_{>0}$ und $\mathbb{Z}_{<0}$ auf $\mathbb{Q}_{<0}$. Wir beschreiben die Abbildung von $\mathbb{Z}_{>0}$ auf $\mathbb{Q}_{>0}$, die Konstruktion des anderen Teils ist analog. Dazu schreiben wir die Menge $\mathbb{Q}_{>0}$ (mit Wiederholungen der Elemente) als zweidimensionale Tafel:



Die Pfeile reihen die rationalen Zahlen wie an einer Perlenkette auf. Dabei verlaufen die Pfeile immer diagonal von rechts oben nach links unten oder umgekehrt. Wir ordnen jetzt der Zahl 1 die Zahl 1 zu und induktiv der Zahl $n+1$ den durch die Pfeile angedeuteten Nachfolger des Bildes von n. Dadurch ist sicher gestellt, daß zu jeder rationalen Zahl ein Urbild existiert (genauer: zu $\frac{n}{m}$ gibt es ein Urbild k mit $k \leq (n+m)^2/2$). Daher ist auch diese Abbildung surjektiv und alles Behauptete bewiesen. □

Im Beweis von Satz 1.9 bezieht sich die Diagonalisierung auf die Hauptdiagonale. Dieses Verfahren wird auch als CANTORsches Diagonalverfahren bezeichnet. Wir haben die reellen Zahlen $\mathbb{R}$ nicht axiomatisch eingeführt. Daher wollen wir es hier bei unseren intuiven Vorstellung der reellen Zahlen als "unendliche Dezimalbrüche" belassen.

BEWEIS: (von Satz 1.9)
Nehmen wir an, der Satz würde nicht gelten und es gäbe eine surjektive Abbildung $g'_N : \mathbb{N} \to \mathbb{R}$. Dann gäbe es auch eine surjektive Abbildung $g_N : \mathbb{N} \to [0,1)$. Wir wollen diese Annahme zum Widerspruch führen; damit wäre die Aussage gezeigt. Wir werden eine reelle Zahl $x \in [0,1)$ konstruieren, die nicht im Wertebereich von g liegt. Dazu sei die n-te Dezimalstelle x_n von x hinter dem Komma definiert durch $x_n = (g(n)_n + 1) \bmod 10$. Wenn wir uns eine Liste der reellen Zahlen vorstellen, so ist x gerade durch die Diagonalelemente konstruiert. Die Konstruktion von x stellt jedoch sicher, daß sich x von jeder Zahl in der Liste $\{g(n)\}_{n\in\mathbb{N}}$ unterscheidet. Daher gilt $x \notin \{g(n)\}_{n\in\mathbb{N}}$ im Widerspruch zur Annahme. □

Man beachte, daß dieser Beweis nicht für die rationalen Zahlen funktioniert, da das konstruierte x in $\mathbb{R} \setminus \mathbb{Q}$ liegt. Mengen, die von der Mächtigkeit der natürlichen Zahlen sind, nennt man *abzählbar*, Mengen einer größeren Mächtigkeit nennt man *überabzählbar*. Für die Mächtigkeit der natürlichen Zahlen hat sich das Symbol χ_0 (Aleph 0) und für die Mächtigkeit der reellen Zahlen das Symbol χ_1 (Aleph 1) eingebürgert.

1.1.4 Das Russellsche Paradoxon in der Mengenlehre

In diesem Abschnitt werden wir zeigen, daß die Bildung von Mengen beliebiger Art zu Paradoxien führen kann. Das RUSSELLsche Paradoxon ist ähnlich der folgenden Paradoxie

"Diese Aussage ist falsch."

Denn wenn "Diese Aussage ist falsch." eine wahre Aussage ist, so folgt aus der Bedeutung der Aussage, daß "Diese Aussage ist falsch" eine falsche Aussage ist. In diesem Fall jedoch muß das Gegenteil der Aussage "Diese Aussage ist falsch." gelten, daß heißt die Aussage ist wahr. Paradoxon!

Ein Paradoxon benannt nach RICHARD ("Les principes de mathématiques et le problème des ensembles", Rev. Gen. des Sc. 16, 541) tritt bei der Betrachtung der folgenden Menge $A = \{r \in \mathbb{R} \mid$ es gibt eine endliche Beschreibung von r in der deutschen Sprache$\}$ auf. So sind z.B. die Zahlen mit den Beschreibungen "drei Komma sieben zwei", "Quadratwurzel aus neunundzwanzig" und "Limes der Folge Klammer auf eins plus eins durch n Klammer zu hoch n" in A enthalten. A ist sicherlich abzählbar, sei f eine Aufzählung (d.h. $f : \mathbb{N} \to A$). Analog zum CANTORschen Diagonalverfahren definieren wir die Zahl x als "diejenige Zahl zwischen Null und Eins, deren n-te Stelle hinter dem Komma gleich der Summe von eins und der n-ten Stelle der n-ten Zahl (bzgl. der Aufzählung f) der Menge A modulo zehn ist". Damit ist x nicht in A enthalten (siehe Beweis von Satz 1.9), besitzt jedoch eine endliche Beschreibung in der deutschen Sprache und ist damit in A enthalten. Paradoxon!

Nun kommen wir zum Paradoxon von RUSSELL. Dazu definieren wir die Menge M aller Mengen, die sich nicht selbst enthalten, d.h. $M = \{X \mid X \notin X\}$. Das Paradoxon entsteht nun dadurch, daß M weder sich selbst enthalten kann noch sich selbst nicht enthalten kann. Denn enthielte M sich selbst, dürfte M (nach Definition von M) sich nicht selbst enthalten. Falls M sich nicht selbst enthielte, müßte jedoch M in sich selbst enthalten sein. Paradoxon!

Diese Widersprüche in der Mengenlehre liegen an dem zu weit (und vorallem zu ungenau) gefaßten Mengenbegriff von CANTOR. Wir wollen jedoch auf eine axiomatische Einführung des Mengenbegriffs verzichten, im wesentlichen wird jedoch verboten, daß Mengen sich selbst enthalten dürfen. Stattdessen wollen wir darauf hinweisen, daß das Paradoxon von RUSSELL durch die *Selbstanwendung* eines *negierten* Prädikats auf ein Objekt entsteht ($X \in M$ genau dann, wenn *nicht* X *sich selbst* enthält).

1.2 Grundlagen der Graphentheorie

In diesem Abschnitt möchten wir einige Grundlagen aus der Graphentheorie erläutern, die wir später zur Definition und Darstellung algorithmischer Strukturen benötigen werden. Graphen treten in vielen möglichen Bereichen der Informatik auf; sie werden sich daher nicht nur in Kapitel 2 bei der Untersuchung von endlichen Automaten als ein nützliches Hilfswerkzeug erweisen.

Zunächst geben wir eine formale mathematische Definition eines (endlichen) Graphen.

Definition 1.10 Ein *Graph* G ist ein 2-Tupel (V, E), wobei V eine endliche Menge und E eine Teilmenge von $V \times V$ ist. Man nennt V die Menge der *Knoten* von G und E die Menge der (gerichteten) *Kanten* von G.

Oft wird V als $V = \{v_1, \ldots, v_n\}$ geschrieben und $E = \{e_1, \ldots, e_m\}$. In der Graphentheorie hat es sich eingebürgert, mit n die Anzahl der Knoten zu bezeichnen und mit m die Anzahl der Kanten. Desweiteren bezeichnen die (indizierten) Kleinbuchstaben u, v und w stets Knoten und e stets Kanten.

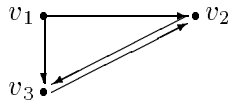


Abbildung 1.1: Graph mit drei Knoten

Der Knoten u heißt mit v *verbunden*, falls $(u,v) \in E$ gilt. Somit ist E gerade die Relation "verbunden". Falls diese Relation symmetrisch ist, d.h. aus $(u,v) \in E$ folgt auch $(v,u) \in E$, so nennen wir den Graphen $G = (V,E)$ *ungerichtet*, sonst heißt G *gerichtet*. Entsprechend gelten auch die anderen Bezeichnungen für Relationen (wie z.B. reflexiv, antisymmetrisch) auch für Graphen. Irreflexive Graphen nennt man auch *schlingenfrei*.

Ein 3-Tupel $G = (V,E,l_E)$ mit V und E wie oben und l_E eine Funktion mit Definitionsbereich E heißt *kantengelabelter* Graph. Dabei ist $l_E(e)$ die der Kante $e \in E$ zugeordnete Marke (Label). Den Wertebereich von l_E lassen wir offen, man kann sich z.B. reelle Zahlen, Symbole, Namen, Funktionen etc. dafür vorstellen.

Analog heißt ein 3-Tupel $G = (V,E,l_V)$, wobei l_V eine Funktion mit Definitionsbereich V eine *knotengelabelter* Graph. Ein 4-Tupel $G = (V,E,l_V,l_E)$ heißt folglich knoten- und kantengelabelter Graph.

Wir können einen Graphen G auch graphisch darstellen, indem wir die Knoten als Punkte (in der Ebene) und Kanten als Pfeile (im gerichteten Fall) oder Linien (im ungerichteten Fall) zwischen den Knoten entsprechenden Punkten darstellen.

Beispiel 1.11 Sei $V = \{v_1, v_2, v_3\}$ und $E = \{(v_1,v_2),(v_1,v_3),(v_2,v_3),(v_3,v_2)\}$. Die zu $G = (V,E)$ gehörige graphische Darstellung ist in Abbildung 1.1 gegeben.

Im folgenden werden wir die Begriffe "Graph" und "graphische Darstellung eines Graphen" miteinander identifizieren.

Eine endliche Folge $P = v_0 v_1 v_2 \cdots v_k$ von Knoten, so daß zwei aufeinander folgende Glieder v_i und v_{i+1} der Folge P durch eine Kante in E miteinander verbunden sind, nennt man einen (gerichteten) *Pfad* von v_0 nach v_k der *Länge* k. Falls $v_0 = v_k$ gilt, so nennt man den Pfad P *geschlossen* oder auch P einen *Zyklus* oder *Kreis*. Graphen, die keinen Zyklus enthalten, heißen *azyklisch*.

Ein *Teilgraph* oder auch *Untergraph* von G ist ein Graph $G' = (V',E')$ mit $V' \subseteq V$ und $E' = E \cap (V' \times V')$, d.h. G' ist auf einer Teilmenge der Knoten definiert und enthält alle Kanten aus G, die zwei Knoten dieser Teilmenge miteinander verbinden. Man nennt G' auch den durch V' induzierten Teilgraphen. Ein Graph $G = (V,E)$ heißt *zusammenhängend* , falls zu jedem $(u,v) \in V^2$ ein Pfad von u nach v oder umgekehrt gibt. Dies ist genau dann der Fall, falls die transitive und reflexive Hülle der Relation E' des ungerichteten Graphen $G' = (V,E')$ mit $E' = \{(u,v) \mid (u,v) \in E \text{ oder } (v,u) \in E\}$ die ganze Menge $V \times V$ ist. Man nennt den so entstehenden Graphen auch die ungerichtete transitive Hülle von G (die transitive Hülle von G ist (V,E^T)). Falls G nicht zusammenhängend ist, so nennt man die maximalen zusammenhängenden Teilgraphen die *Zusammenhangskomponenten* von G.

Eine Teilmenge von Knoten $V' \subseteq V$, so daß alle Knoten aus V' durch eine Kante $e \in E$ mitein-

ander verbunden sind, nennt man *Clique*. Eine Teilmenge $V' \subseteq V$, so daß keine Knoten aus V' miteinander verbunden sind, nennt man *unabhängige Menge*. Eine Clique V' maximaler Kardinalität (d.h. für alle Cliquen $W \subseteq V$ gilt $|W| \leq |V'|$) heißt auch maximale Clique, eine Clique V', die nicht durch Hinzunahme eines Knotens erweitert werden kann, nennt man hingegen nichterweiterbare Clique. Analoge Bezeichnungen gelten für unabhängige Mengen.

Sei u ein Knoten. Dann bezeichnen wir die Menge $\mathrm{IN}(u) = \{v \mid (v,u) \in E\}$ als die Menge der Eingangsknoten von u und die Menge $\mathrm{OUT}(u) = \{v \mid (u,v) \in E\}$ als die Menge der Ausgangsknoten von u. Die Menge $N(u) = \mathrm{IN}(u) \cup \mathrm{OUT}(u)$ ist die Menge der Nachbarn von u. Die Größen $\deg_{\mathrm{IN}}(u) = |\mathrm{IN}(u)|$, $\deg_{\mathrm{OUT}}(u) = |\mathrm{OUT}(u)|$ bzw. $\deg(u) = |N(u)|$ werden Eingangsgrad, Ausgangsgrad bzw. Grad von u genannt.

Mittels dieser Begriffe ist es uns möglich eine sehr wichtige Klasse von Graphen zu charakterisieren.

Definition 1.12 Ein *Baum* ist azyklischer zusammenhängender Graph.

Wir wollen nun einen Satz zitieren, der eine andere Charakterisierung von Bäumen erlaubt.

Satz 1.13 Sei $T = (V, E)$ ein zusammenhängender Graph. T ist genau dann ein Baum, wenn eine der folgenden Bedingungen erfüllt ist:

1. Für alle Knoten $u, v \in V$ gibt es genau einen Pfad von u nach v.
2. Das Hinzufügen einer Kante zu E erzeugt genau einen Zyklus.
3. $|E| = n - 1$.

Der Beweis kann in [Gib85] oder in jedem anderen Lehrbuch über Graphentheorie nachgelesen werden.

Ein gerichteter Baum ist ein gerichteter, azyklischer zusammenhängender Graph. Er kann auch folgendermaßen charakterisiert werden.

Satz 1.14 Ein gerichteter Graph T ist genau dann ein gerichteter Baum, falls die folgenden Eigenschaften erfüllt sind:

1. Es gibt genau einen Knoten w mit $\deg_{\mathrm{IN}}(w) = 0$, d.h. es führt keine Kante zu w.
2. Alle Knoten v außer w haben Eingangsgrad $\deg_{\mathrm{IN}}(v) = 1$.
3. Für alle Knoten v gibt es genau einen (gerichteten) Pfad von w nach v.

Ein Baum, in dem der Ausgangsgrad jeden Knotens durch k beschränkt ist, nennt man auch k-nären Baum. Der Knoten w aus Punkt 1. wird auch *Wurzel* des Baumes genannt. Die Knoten v mit Ausgangsgrad $\deg_{\mathrm{OUT}}(v) = 0$ nennt man auch die *Blätter* des Baumes T. Die Menge $\mathrm{OUT}(v)$ nennt man die Söhne von v. Ein Baum T heißt *balanciert*, falls sich die Größe der Teilbäume, die an den Söhnen eines (beliebigen) Knotens gewurzelt sind, um höchstens eins unterscheidet. Abbildung 1.2 zeigt den Graphen eines balancierten (binären) Baumes mit Wurzel v_1 und Blättern v_4, v_8, v_6, v_7. Man nennt die Länge eines längsten Pfades von der Wurzel zu einem Blatt in T auch die *Höhe* (manchmal auch Tiefe) des Baumes T.

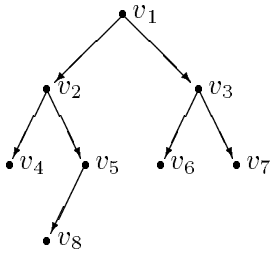


Abbildung 1.2: Ein binärer Baum

1.3 Aussagenlogik und Boolesche Funktionen

In diesem Abschnitt wollen wir untersuchen, wie sich logische Verknüpfungen auf den Wahrheitsgehalt von Aussagen auswirken. Formal kann man Aussagen als Funktionen auffassen, die Zeichenketten (aus mathematischen Symbolen) in die Menge $\mathbb{B}$ der Wahrheitswerte $\underline{\mathrm{W}}$ und $\underline{\mathrm{F}}$ abbilden. Da sich die Verknüpfungen von Aussagen nur auf ihren Wahrheitsgehalt beziehen, d.h. da die Verknüpfungen nur im Wertebereich stattfinden, können wir uns darauf beschränken, die Wirkungen der Verknüpfungen auf sogenannten *Boolesche Variablen*, d.h. Variablen über $\mathbb{B}$, zu betrachten. Der Einfachheit halber werden wir die Elemente von $\mathbb{B}$ auch mit 1 für $\underline{\mathrm{W}}$ und 0 für $\underline{\mathrm{F}}$ bezeichnen.

Wir werden in diesem Abschnitt daher *Boolesche Funktionen* studieren. Boolesche Funktionen sind Funktion $f : \mathbb{B}^n \to \mathbb{B}$. Eine Boolesche Funktion f verknüpft damit n Aussagen $A_1, \ldots, A_n$ zu der Aussage $A = f(A_1, \ldots, A_n)$. Die Menge der n-stelligen Booleschen Funktionen bezeichnen wir auch mit $\mathbb{B}_n = \{f \mid f : \mathbb{B}^n \to \mathbb{B}\}$.

Da Boolesche Funktionen einen endlichen Definitionsbereich haben, ist es möglich sie durch eine vollständige Tabelle zu beschreiben. Wir werden auf diese Art die folgenden Funktionen definieren:

1. die Negation (Verneinung) $\neg : \mathbb{B} \to \mathbb{B}$, synonym wird statt $\neg$ auch NOT geschrieben.

2. die Konjunktion (Und-Verknüpfung) $\wedge : \mathbb{B}^2 \to \mathbb{B}$ (auch AND).

3. die Disjunktion (Oder-Verknüpfung) $\vee : \mathbb{B}^2 \to \mathbb{B}$ (auch OR).

4. die Implikation (Folgerung) $\to : \mathbb{B}^2 \to \mathbb{B}$.

5. die Exclusiv-Oder-Verknüpfung $\oplus : \mathbb{B}^2 \to \mathbb{B}$ (auch XOR).

Die zweistelligen Funktionen werden oft auch als Operatoren zwischen den Argumenten geschrieben, d.h. anstatt $\vee(A, B)$ schreiben wir $A \vee B$.

Wir definieren jetzt die obigen Funktionen durch sogenannte *Wahrheitstabellen* oder *Wahrheitstafeln*. Seien A und B zwei Boolesche Variablen.

Negation

A	$\neg A$
W	F
F	W

Konjunktion

A	B	$A \wedge B$
W	W	W
W	F	F
F	W	F
F	F	F

Disjunktion

A	B	$A \vee B$
W	W	W
W	F	W
F	W	W
F	F	F

Implikation

A	B	$A \to B$
W	W	W
W	F	F
F	W	W
F	F	W

Exklusiv-Oder Verknüpfung

A	B	$A \oplus B$
W	W	F
W	F	W
F	W	W
F	F	F

Beispiel 1.15 Sei p eine natürliche Zahl. Sei A die Aussage "p ist eine Primzahl" und B die Aussage "p ist ein Quadrat". Dann ist $A \vee B$ die Aussage "p ist eine Primzahl oder ein Quadrat", $A \wedge B$ "p ist eine Primzahl und ein Quadrat" und $A \to B$ "Wenn p eine Primzahl ist, so ist p ein Quadrat". Die Aussage $\neg A$ ist "p ist keine Primzahl".

Die Tatsache, daß die Aussagen $A \wedge B$ und $A \to B$ aus obigen Beispiel falsche Aussagen sind auch wenn p eine *Variable* ist, kann nicht im Rahmen der Aussagenlogik bewiesen werden. Daher werden wir später in Kapitel 5 das aufwendigere Kalkül der Prädikatenlogik studieren müssen.

Für die Konjunktion und Disjunktion gelten die folgenden Distributivgesetze (dabei seien X, Y und Z Boolesche Variablen bzw. Aussagen):

1. $(X \wedge Y) \vee Z = (X \vee Z) \wedge (Y \vee Z)$
2. $(X \vee Y) \wedge Z = (X \wedge Z) \vee (Y \wedge Z)$

Außerdem sind alle obigen Funktionen außer der Implikation ("$\to$") kommutativ.

Eine Boolesche Funktion $f : \mathbb{B}^n \to \mathbb{B}$ ist eindeutig durch das Urbild $f^{-1}(1)$ festgelegt. f bildet nämlich gerade die Elemente von $f^{-1}(1)$ auf 1 und die Elemente aus $\mathbb{B}^n \setminus f^{-1}(1)$ auf 0 ab. Daher gibt es genauso viele n-stellige Boolesche Funktionen f wie Teilmengen von $\mathbb{B}^n$. Es ist also $|\mathbb{B}_n| = 2^{|\mathbb{B}^n|} = 2^{2^n}$.

Besonders wichtig sind die mehrstelligen Verallgemeinerungen der Konjunktion ($\wedge$) und Disjunktion ($\vee$) (die wir der Einfachheit halber ebenfalls mit $\wedge$ bzw. $\vee$ bezeichnen). Wir definieren induktiv $\wedge : \mathbb{B}^n \to \mathbb{B}$ durch $\wedge(x_1, \ldots, x_n) := (x_1 \wedge x_2) \wedge \cdots \wedge x_n$ und analog auch $\vee$.

Die oben eingeführten Funktionen genügen bereits, um alle Booleschen Funktionen als Verkettung dieser Funktionen darstellen zu können. Genauer gilt:

Satz 1.16 Jede Boolesche Funktion kann mittels Verkettung der Funktionen $\vee$, $\wedge$ und $\neg$ dargestellt werden.

BEWEIS: Sei $f \in \mathbb{B}_n$ und $\mathbf{x} = (x_1, \ldots, x_n)$ eine Variable über $\mathbb{B}^n$. Dann läßt f sich schreiben als

$$f = \bigvee_{a \in f^{-1}(1)} m_a,$$

wobei $m_a \in B_n$ die eindeutige Funktion mit

$$m_a(\mathbf{x}) = \begin{cases} 1 & \text{falls } x_i = a_i \text{ für } 1 \leq i \leq n \\ 0 & \text{sonst} \end{cases}$$

ist. m_a läßt sich schreiben als $\bigwedge_{i=1}^{n} x_i^{a_i}$ wobei x^1 als x und x^0 als $\neg x$ definiert sind (man nennt x^i auch *Literal* und m_a *Minterm*). Die so konstruierte Darstellung von f benutzt nur die Funktionen $\vee$, $\wedge$ und $\neg$. □

Die obige Darstellung einer Booleschen Funktion heißt *kanonische Disjunktive Normalform* (KDNF). Neben ihr existieren noch andere kanonische Darstellungen wie *kanonische Konjunktive Normalform* (KKNF) und die *Ringsummendarstellung* (RSE). Die KKNF ist definiert durch

$$f = \bigwedge_{a \in f^{-1}(0)} M_a,$$

wobei $M_a = \vee_{i=1}^{n} x_i^{\neg a_i}$ ein sog. *Maxterm* ist. Eine DNF D für f ist eine Disjunktion von Konjunktion von Literalen (dies müssen keine Minterme sein), so daß $f = D$ gilt. Analog ist eine KNF K für f als eine Konjunktion von Disjunktionen von Literalen mit $K = f$ definiert. Eine t-DNF ist eine DNF, in der jede Konjunktion höchstens t Literale enthält, analog ist eine t-KNF eine KNF, in der jede Disjunktion höchstens t Literale enthält. Im Gegensatz zur Darstellung einer Booleschen Funktion mittels KNF oder DNF ist die RSE-Darstellung eindeutig. Dabei wird f dargestellt als

$$f = \bigoplus_{a \in A} \bigwedge_{i, a_i = 1} x_i,$$

wobei A eine Teilmenge von $\{0,1\}^n$ ist.

1.3.1 Simulation Boolescher Funktionen durch Schaltungen

Wir wollen nun Boolesche Funktionen durch Schaltungen und Schaltnetze simulieren.

Eine Variable x stellen wir durch einen *Schalter* dar. Hat die Variable x den Wert 1, so ist der zugehörige Schalter A geschlossen, sonst ist er offen. Falls zwei Punkte P und Q über eine elektrische Leitung, in der sich ein Schalter befindet, miteinander verbunden sind, so kann nur dann Strom von P nach Q fließen, wenn der Schalter geschlossen ist. Der Schalter simuliert also die Aussage "Es kann Strom von P nach Q fließen". Abbildung 1.3 zeigt einen Schalter. Eine Kombination von mehreren Schaltern nennt man *Schaltung* oder auch *Schaltnetz*.

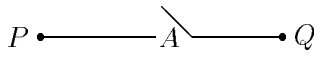


Abbildung 1.3: Schalter

Eine *Parallelschaltung* ist eine Schaltung mit Schaltern $A_1, \ldots, A_n$, so daß zwischen den Punkten P und Q genau dann Strom fließen kann, wenn mindestens einer der Schalter geschlossen ist. Eine *Serienschaltung* ist eine Schaltung mit Schaltern $A_1, \ldots, A_n$, so daß zwischen den Punkten P und Q genau dann Strom fließen kann, wenn alle Schalter geschlossen sind. Eine Parallelschaltung entspricht damit der Verkettung der Aussagen $A_1, \ldots, A_n$ mittels Disjunktion ($\vee$) und die Serienschaltung der Verkettung mittels Konjunktion ($\wedge$). Abbildungen 1.4 und 1.5 zeigen eine Parallel- bzw. Serienschaltung mit je drei Schaltern.

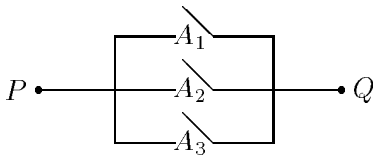


Abbildung 1.4: Parallelschaltung

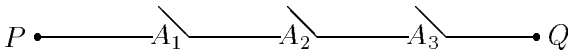


Abbildung 1.5: Serienschaltung

Beispiel 1.17 Abbildung 1.6 zeigt ein Schaltnetz für die Boolesche Funktion $f : \mathbb{B}^2 \to \mathbb{B}, (A, B) \mapsto (A \wedge B) \vee (\neg A \wedge B)$.

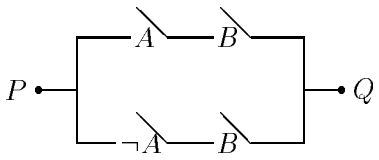


Abbildung 1.6: Schaltnetz für $(A \wedge B) \vee (\neg A \wedge B)$

Allgemeine Schaltnetze, in denen Teilschaltnetze auftauchen, die Parallel- oder Serienschaltungen sind, nennen wir *Serienparallelschaltungen* (SPS). Insbesondere ist ein Schalter eine SPS.

Formal kann ein Schaltnetz als ein gerichteter, kantengelabelter, zusammenhängender Graph $G = (V, E)$ definiert werden. Dabei ist E die Menge der Schalter und V die Menge der Punkte, an denen Leitungen (die genau einen Schalter enthalten) miteinander verbunden sind. Insbsondere sind P und Q in V. Die Kanten aus E sind mit Literalen gelabelt. P ist der einzige Knoten mit Eingangsgrad Null und Q der einzige Knoten mit Ausgangsgrad 0. Abbildung 1.7 zeigt das Schaltnetz aus Beispiel 1.17 als Graph.

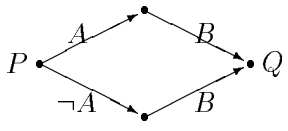


Abbildung 1.7: Schaltgraph für $(A \wedge B) \vee (\neg A \wedge B)$

Die Größe einer SPS definieren wir als die Anzahl der in ihr benutzten Schalter. Durch Vereinfachung der dargestellten Booleschen Funktion mittels der Distributivgesetze können Schaltnetze kleinerer Größe gewonnen werden.

Beispiel 1.18 Sei $Z = f(A, B, C) = (A \wedge B \wedge \neg C) \vee (\neg C \wedge \neg A)$. Die direkte Übersetzung dieser Funktion in ein Schaltnetz ergibt eine SPS der Größe 5. Unter Anwenden der Distributivgesetze

läßt sich f zu $f = \neg C \wedge (B \vee \neg A)$ vereinfachen. Diese Darstellung von f ermöglicht ein Schaltnetz der Größe 3.

Der Nachteil von Schaltnetzen ist die Tatsache, daß jedem Schalter eine Variable zugeordnet ist, und somit ist ein entsprechend großer Aufwand nötig. Daher wollen wir zusätzlich eine andere Darstellung Boolescher Funktion einführen. Dies sind *logische Schaltungen*, die aus (logischen) *Gattern* aufgebaut sind.

Genauer ist eine logische Schaltung ein knotengelabelter, azyklischer, gerichteter Graph $G = (V, E, k)$. V besteht aus der Menge von Knoten, die die logischen Gattern repräsentieren und n zusätzlichen Eingabeknoten für die Eingabevariablen. Diese Knoten sind mit den Variablen $x_1, \ldots, x_n$ gelabelt, die anderen Knoten mit einer Booleschen Funktion. Dabei muß der Eingangsgrad des Knoten der Stelligkeit der Funktion entsprechen. Ein Knoten aus V ist dadurch ausgezeichnet, daß er als einziger Ausgangsgrad 0 hat. Dieser Knoten ist der Ausgabeknoten y. Bei der Belegung der Variablen durch Werte erhält jeder Knoten in dem Graphen rekursiv (bei den den Eingangsvariablen entsprechenden Knoten anfangend) einen Wert zugewiesen. Dieser ergibt sich durch Anwenden der dem Knoten zugeordneten Funktion auf die Werte der Eingangsknoten. Desweiteren nehmen wir an, daß G keine überflüssigen Knoten enthält, d.h. daß jeder Knoten mit dem Ausgabeknoten y durch einen Pfad verbunden ist.

Eine logische Schaltung G mit n Eingabeknoten *berechnet* eine n-stellige Boolesche Funktion f, falls für jede Variablenbelegung der Funktionswert von f mit dem Wert des Ausgabeknotens y übereinstimmt.

Wir definieren nun noch die Größe und die Tiefe einer logischen Schaltung.

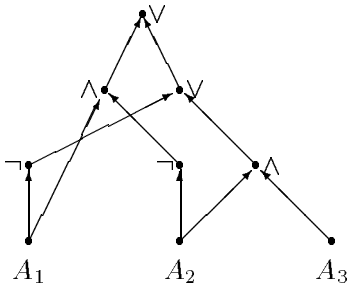
Definition 1.19 Sei $G = (V, E, k)$ eine logische Schaltung mit n Eingaben $x_1, \ldots, x_n$. Dann ist die *Größe*

$$\text{Size}(G) = |V \setminus \{k^{-1}(x_i) \mid 1 \leq i \leq n\}|$$

und die *Tiefe*

$$\text{Depth}(G) = \max\{|p| \mid p \text{ ist ein gerichteter Pfad in } V \setminus \{k^{-1}(x_i) \mid 1 \leq i \leq n\}\}.$$

Beispiel 1.20 Die folgende logische Schaltung berechnet die Funktion $f(A_1, A_2, A_3) = (A_1 \wedge \neg A_2) \vee (\neg A_1 \vee (A_2 \wedge A_3))$. Sie hat Größe 6 und Tiefe 3.



1.4 Boolesche Algebra

Wir hatten in Lemma 1.2 gesehen, daß die Operationen $\cup$ und $\cap$ gewisse Gesetze (Kommutativgesetz, Distributivgesetz usw.) erfüllen. In diesem Abschnitt werden wir allgemein die Struktur

von 4-Tupeln $(M, +, *, -)$ untersuchen, die der folgenden Definition genüge leisten.

Definition 1.21 Ein 4-Tupel $B = (M, +, *, -)$ bestehend aus einer Menge M, den zweistelligen Funktionen $+, * : M^2 \to M$ und der einstelligen Funktion $- : M \to M$ heißt *Boolesche Algebra*, falls B die folgenden Axiome erfüllt (dabei notieren wir $+, *$ als auch Operatoren zwischen den Argumenten):

Axiom 1 $+$ und $*$ sind kommutativ, d.h. für alle a und b gelten: $a+b = b+a$ und $a*b = b*a$.

Axiom 2 Es gibt eindeutig bestimmte Elemente 0 und 1 in M mit $0+a = a$ und $1*a = a$ für alle $a \in M$. 0 heißt *Nullelement* und 1 heißt *Einselement*.

Axiom 3 $+$ und $*$ sind distributiv, d.h. für alle a, b und c gelten: $(a+b)*c = (a*c)+(b*c)$ und $(a*b)+c = (a+c)*(b+c)$.

Axiom 4 Für alle $a \in M$ gilt: $a+(-a) = 1$ und $a*(-a) = 0$. $-a$ heißt das zu a *komplementäre* Element oder auch *Komplement* von a.

Aus diesen vier Axiomen lassen sich eine Reihe von Aussagen ableiten. Darunter befindet sich das *Dualitätsprinzip* (siehe auch [Kla83]), dessen Beweis sofort aus der Symmetrie der Axiome bezüglich der Operationen $*$ und $+$ und der Elemente 0 und 1 folgt.

Satz 1.22 Zu jeder Aussage, die sich aus den vier Axiomen ableiten läßt, existiert eine duale Aussage, die dadurch entsteht, daß man die Operatoren $+$ und $*$ und gleichzeitig die Elemente 0 und 1 vertauscht.

Die weiteren einfachen Folgerungen aus den Axiomen zitieren wir ohne Beweis.

1. Für alle $a \in M$ gilt: $a + 1 = 1$ und $a * 0 = 0$.
2. Für alle $a \in M$ gilt: $a + a = a$ und $a * a = a$.
3. Für alle $a, b \in M$ gelten: $a + (a * b) = a$ und $a * (a + b) = a$ (Absorbtionsgesetze).
4. Für alle $a, b \in M$ gelten: $a + (b * (-b)) = a$ und $a * (b + (-b)) = a$ (Negationsgesetze).
5. Für alle $a, b, c \in M$ gelten: $(a+b)+c = a+(b+c)$ und $(a*b)*c = a*(b*c)$ (Assoziativgesetze).
6. Für alle a gilt: $(-(-a)) = a$.
7. Das Null- und das Einselement sind zu einander komplementär, d.h. $-0 = 1$.
8. Für alle $a, b \in M$ gelten: $-(a+b) = (-a)*(-b)$ und $-(a*b) = (-a)+(-b)$ (Gesetze von De Morgan).

Wir haben, ohne es zu wissen, schon zwei Boolesche Algebren kennengelernt.

Beispiel 1.23 $(\mathbb{B}, \vee, \wedge, \neg)$ mit Einselement $\underline{\mathrm{W}}$ und Nullelement $\underline{\mathrm{F}}$ ist eine Boolesche Algebra.

Sei A eine Menge. Dann ist $(\mathfrak{P}(A), \cup, \cap, ^-)$ eine Boolesche Algebra mit Einselement A und Nullelement $\emptyset$ ($^-$ bezieht sich auf A). In diesem Fall nennt man die Boolesche Algebra auch *Mengenalgebra*.

1.5 Asymptotisches Wachstum von Funktionen

Dieser Abschnitt dient dazu, Klassen von Funktionen zu definieren, die ein ähnliches asymptotisches Wachstum aufweisen. Im folgenden seien f und g Funktionen von $\mathbb{N}$ nach $\mathbb{N}$. Dann definieren wir die folgenden Klassen von Funktionen:

- Die Klasse $O(f)$ (sprich: groß Oh) enthält alle Funktionen g, deren asymptotisches Wachstum von f majorisiert wird, d.h.

$$O(f) = \{g : \mathbb{N} \to \mathbb{N} \mid \exists\, c > 0, n_0\, \forall\, n \geq n_0 :\ g(n) \leq cf(n)\}.$$

- Die Klassen $\Omega(f)$ (sprich: groß Omega) bzw. $\Omega'(f)$ enthalten alle Funktionen g, deren asymptotisches Wachstum von f minorisiert wird. Dabei ist $\Omega(f)$ aus technischen Gründen nicht analog zu $O(f)$ definiert:

$$\begin{aligned} \Omega(f) &= \{g : \mathbb{N} \to \mathbb{N} \mid \exists\, c > 0 \ :\ g(n) \geq cf(n) \text{ für } \infty \text{ viele } n\}, \\ \Omega'(f) &= \{g : \mathbb{N} \to \mathbb{N} \mid \exists\, c > 0, n_0\, \forall\, n \geq n_0 :\ g(n) \geq cf(n)\}. \end{aligned}$$

- Die Klassen $\Theta(f)$ und $\Theta'(f)$ enthalten alle Funktionen g, deren asymptotisches Wachstum von f sowohl majorisiert als auch minorisiert wird:

$$\Theta(f) = O(f) \cap \Omega(f), \quad \Theta'(f) = O(f) \cap \Omega'(f).$$

- Die Klasse $o(f)$ (sprich: klein oh) enthält alle Funktionen g, deren asymptotisches Wachstum von f <u>echt</u> majorisiert wird, d.h.

$$o(f) = \{g : \mathbb{N} \to \mathbb{N} \mid \forall\, c > 0\, \exists\, n_0\, \forall\, n \geq n_0 :\ g(n) \leq cf(n)\}.$$

- Analog ist die Klasse $\omega(f)$ definiert. Sie enthält alle Funktionen g, deren asymptotisches Wachstum von f <u>echt</u> minorisiert wird, d.h.

$$\omega(f) = \{g : \mathbb{N} \to \mathbb{N} \mid \forall\, c > 0\, \exists\, n_0\, \forall\, n \geq n_0 :\ g(n) \geq cf(n)\}.$$

- Sei $\log : \mathbb{N}_0 \to \mathbb{N}$ definiert durch

$$\log(n) = \begin{cases} 1 & \text{falls } n \leq 1 \\ \lceil \log_2(n) \rceil & \text{falls } n \geq 2 \end{cases}$$

  und $\log^k$ durch $\log^k(n) = (\log(n))^k$. Dann definieren wir die Klasse $O^{\sim}(f)$ (sprich: soft Oh) durch

$$O^{\sim}(f) = \{g : \mathbb{N} \to \mathbb{N} \mid \exists\, k\ : g \in O(f \log^k(f))\}.$$

  Diese Klasse besteht also aus allen Funktionen, deren Wachstum unter Vernachlässigung logarithmischer Terme von f majorisiert wird.

Aus Gründen der einfacheren Notation schreiben wir auch $g = O(f)$ anstatt $g \in O(f)$. Dies ist jedoch nur eine Notation! Dabei ist auch zu beachten, daß $O(f)$ auf der rechten Seite der "Gleichung" steht. Analoge Konventionen gelten auch für die anderen asymptotischen Funktionenklassen.

Es sei bemerkt, daß die Funktionenklassen $o(f)$ und $\omega(f)$ in der Analysis oft durch

$$\begin{aligned} o(f) &= \{g \mid \lim_{h\to 0} \frac{g(h)}{f(h)} = 0\} \\ \omega(f) &= \{g \mid \lim_{h\to 0} \frac{f(h)}{g(h)} = 0\} \end{aligned}$$

definiert sind. Unsere Definition unterscheidet sich dadurch, daß wir den Grenzwert gegen ∞ anstatt gegen 0 betrachten. Die Symbole o und ω hat LANDAU in die Mathematik eingeführt; sie heißen daher auch oft *Landausche Symbole*.

Beispiel 1.24 Sei P ein Polynom, $P(x) = a_0 + a_1 x + \cdots + a_n x^n$. Dann ist

$$\begin{aligned} P(x) &\leq |a_0| + |a_1|x + \cdots + |a_n|x^n \\ &\leq (|a_0| + |a_1| + \cdots + |a_n|)x^n. \end{aligned}$$

Daher ist $P \in O(x^n)$ mit Wahl $c = |a_0| + \cdots |a_n|$ und $n_0 = 1$.

Wir nennen zwei Funktionen f und g *polynomiell verknüpft*, falls sowohl $f \in O(g^{O(1)})$ als auch $g \in O(f^{O(1)})$ gelten.

1.6 Elementare Wahrscheinlichkeitsrechnung

In diesem Abschnitt werden wir die Grundlagen der Wahrscheinlichkeitsrechnung legen, die wir unter anderem in Kapitel 4.3 brauchen werden. Sie sind im wesentlichen dem Buch von MORGENSTERN [Mor64] entnommen. Wir werden jedoch nur die *diskrete* Wahrscheinlichkeitsrechnung einführen, da die allgemeine Wahrscheinlichkeitstheorie wesentlich mehr mathematische Kenntnisse voraussetzt.

Zunächst definieren wir den Begriff des *Mengenkörpers*.

Definition 1.25 Sei Ω ein Menge. Dann heißt $R \subseteq \mathfrak{P}(\Omega)$ *Mengenkörper* über Ω, falls R die folgenden drei Axiome erfüllt:

1. $\Omega \in R$.
2. Mit A ist auch das Komplement von A (bzgl. Ω) in R enthalten, d.h. aus $A \in R$ folgt $\bar{A} \in R$.
3. Mit A und B ist auch die Vereinigung in R enthalten, d.h. aus $A, B \in R$ folgt $A \cup B \in R$.

Aus diesen Axiomen läßt sich unmittelbar folgern, daß $\emptyset \in R$ gilt und mit $A, B \in R$ auch die Schnittmenge $A \cap B$ in R enthalten ist.

Die Elemente eines Mengenkörpers bezeichnet man in der Wahrscheinlichkeitsrechnung als *Ereignisse*. Falls $R = \mathfrak{P}(\Omega)$ ist, so bezeichnet man die einelementigen Teilmengen von Ω auch als *Elementarereignisse*.

Beispiel 1.26 Modelliere Ω die Menge aller Ergebnisse eines Wurfes mit zwei (unterscheidbaren) Würfeln, d.h. sei $\Omega = \{(i,j) \mid 1 \leq i,j \leq 6\}$ und $R = \mathfrak{P}(\Omega)$. Dann ist z.B. $A = \{(1,3),(2,2),(3,1)\}$ das Ereignis, daß die Summe der Augen der Würfel 4 ergibt, $B = \{(1,1),(2,2),\ldots,(6,6)\}$ das Ereignis, daß ein Pasch geworfen wird und $C = \{(1,6)\}$ das Elementarereignis, daß der erste Würfel eine 1 und der zweite eine 6 zeigt.

Wir führen die folgenden Sprechweisen ein:

- Statt "$A \cap B$ tritt ein" sagen wir auch "A und B treten ein".
- Statt "$A \cup B$ tritt ein" sagen wir auch "A oder B tritt ein".
- Statt "$\bigcup A_i$ tritt ein" sagen wir auch "wenigstens ein A_i tritt ein".
- Statt "$\bigcap A_i$ tritt ein" sagen wir auch "alle A_i treten ein".

Nun werden wir eine Abbildung definieren, die jedem Ereignis einen Wert (Wahrscheinlichkeitswert, Wahrscheinlichkeit) zu ordnet.

Eine Abbildung $\mathbf{P} : R \to \mathbb{R}$ heißt *Wahrscheinlichkeitsmaß* auf R, falls $\mathbf{P}$ die folgenden drei Eigenschaften erfüllt:

1. Kein Ereignis hat eine negative Wahrscheinlichkeit, d.h. für alle $A \in R$ ist $\mathbf{P}(A) \geq 0$.
2. Die Abbildung $\mathbf{P}$ ist additiv auf disjunkten Ereignissen, d.h. für alle $A, B \in R$ mit $A \cap B = \emptyset$ gilt $\mathbf{P}(A \cup B) = \mathbf{P}(A) + \mathbf{P}(B)$.
3. $\mathbf{P}$ erfüllt die Normierungsbedingung $\mathbf{P}(\Omega) = 1$.

Man sagt auch $\mathbf{P}$ induziert eine *Verteilung* auf Ω.

Beispiel 1.27

1. Seien die Würfel aus obigen Beispiel ungezinkt, d.h. die Wahrscheinlichkeit für jedes Elementarereignis ist gleich. Dann gilt wegen der Additivität und Normierung $\mathbf{P}(\{i,j\}) = 1/36$ für alle (i,j) (uniforme Wahrscheinlichkeitsverteilung). Die Wahrscheinlichkeit für die Ereignisse A, B und C sind wie folgt: $\mathbf{P}(A) = 1/12$, $\mathbf{P}(B) = 1/6$ und $\mathbf{P}(C) = 1/36$.
2. Nun sei der zweite Würfel gezinkt, und zwar so, daß die Wahrscheinlichkeit für eine "Eins" $1/2$ beträgt und für die anderen Ereignisse jeweils $1/10$. Dann gelten für A, B und C:
$$\begin{aligned} \mathbf{P}(A) &= \frac{1}{60} + \frac{1}{60} + \frac{1}{12} = \frac{7}{60} \\ \mathbf{P}(B) &= \frac{1}{12} + \frac{1}{60} + \cdots \frac{1}{60} = \frac{1}{6} \\ \mathbf{P}(C) &= \frac{1}{60}. \end{aligned}$$
3. Nun sei der zweite Würfel so gezinkt, daß die Wahrscheinlichkeit für eine "Eins" 1 beträgt. Dann ergibt sich $\mathbf{P}(A) = 1/6$, $\mathbf{P}(B) = 1/6$ und $\mathbf{P}(C) = 0$.

Ein Wahrscheinlichkeitsmaß erfüllt auch noch die folgenden Eigenschaften:

- Für alle $A \in R$ gilt $\mathbf{P}(A) \leq 1$.
- $\mathbf{P}(\emptyset) = 0$.
- Für alle $A \in R$ gilt $\mathbf{P}(\bar{A}) = 1 - \mathbf{P}(A)$.
- Falls für $A, B \in R$ $A \supseteq B$ gilt, so folgt $\mathbf{P}(A) \geq \mathbf{P}(B)$ und $\mathbf{P}(A \setminus B) = \mathbf{P}(A) - \mathbf{P}(B)$.

Wir fassen zusammen: Das Tripel $(\Omega, R, \mathbf{P})$, wobei Ω eine endliche Menge, R eine Mengenkörper über Ω und $\mathbf{P}$ ein Wahrscheinlichkeitsmaß auf R sind, heißt *Wahrscheinlichkeitsraum* für Ω.

Wir wenden uns nun der Konstruktion von Wahrscheinlichkeitsräumen zu. Seien $(\Omega_1, R_1, \mathbf{P}_1)$ und $(\Omega_2, R_2, \mathbf{P}_2)$ zwei Wahrscheinlichkeitsräume. Dann ist der Produktraum $(\Omega_3, R_3, \mathbf{P}_3)$ definiert durch

$$\begin{aligned} \Omega_3 &= \Omega_1 \times \Omega_2, \\ R_3 &= R_1 \times R_2 \subseteq \mathfrak{P}(\Omega_1 \times \Omega_2) \\ \mathbf{P}_3 &: \mathbb{R}_3 \to \mathbb{R}, \quad (A, B) \mapsto \mathbf{P}_1(A) \cdot \mathbf{P}_2(B). \end{aligned}$$

Der Wahrscheinlichkeitsraum aus unserem Beispiel ist der Produktraum der den beiden Würfeln zugeordneten Wahrscheinlichkeitsräume. Eine andere Konstruktion entsteht durch eine Art Projektion.

Sei $(\Omega, R, \mathbf{P})$ ein Wahrscheinlichkeitsraum und $B \in R$ ein Ereignis mit $\mathbf{P}(B) > 0$. Dann ist die *bedingte Wahrscheinlichkeit* eines Ereignisses A unter der Bedingung B definiert durch

$$\mathbf{P}(A \mid B) = \frac{\mathbf{P}(A \cap B)}{\mathbf{P}(B)}.$$

Statt $\mathbf{P}(A \mid B)$ schreiben wir auch $\mathbf{P}_B(A)$. $\mathbf{P}_B$ ist das durch B *bedingte* Wahrscheinlichkeitsmaß (dies muß gezeigt werden), der durch $\mathbf{P}_B$ induzierte Wahrscheinlichkeitsraum $(\Omega, R, \mathbf{P}_B)$ heißt auch bedingter Wahrscheinlichkeitsraum.

Satz 1.28 Sei $(\Omega, R, \mathbf{P})$ und B wie oben. Dann ist $\mathbf{P}_B$ ein Wahrscheinlichkeitsmaß.

BEWEIS: Wir müssen die drei Axiome überprüfen. Es ist $\mathbf{P}_B(A) = \mathbf{P}(A \cap B)/\mathbf{P}(B) \geq 0$ und $\mathbf{P}_B(\Omega) = \mathbf{P}(B \cap \Omega)/\mathbf{P}(B) = 1$. Das zweite Axiom ergibt sich aus

$$\mathbf{P}_B(A \cup C) = \frac{\mathbf{P}((A \cup C) \cap B)}{\mathbf{P}(B)} = \frac{1}{\mathbf{P}(B)} \mathbf{P}((A \cap B) \cup (C \cap B)) = \mathbf{P}_B(A) + \mathbf{P}_B(C),$$

für zwei disjunkte Ereignisse A und C. □

BAYES hat die Beziehungen zwischen bedingten Wahrscheinlichkeiten untersucht. Ihm ist der folgende Satz zu verdanken.

Satz 1.29 Seien $A_1, \ldots, A_r$ und $B_1, \ldots, B_s$ jeweils disjunkte Ereignisse mit $\bigcup A_i = \Omega$ und $\bigcup B_k = \Omega$. Dann gilt die BAYESsche Formel

$$\mathbf{P}_{A_i}(B_k) = \frac{\mathbf{P}_{B_k}(A_i)\mathbf{P}(B_k)}{\sum_l \mathbf{P}_{B_l}(A_i)\mathbf{P}(B_l)}.$$

BEWEIS: Der Beweis ergibt sich durch elementare Umformungen und der Definition für bedingte Wahrscheinlichkeit. □

Es ist interessant zu untersuchen, unter welchen Voraussetzungen ein bedingter Wahrscheinlichkeitsraum gleich dem ursprünglichen Wahrscheinlichkeitsraum ist. Dies ist natürlich genau dann der Fall, wenn

$$\forall A \in R: \quad \mathbf{P}_B(A) := \mathbf{P}(A \mid B) = \mathbf{P}(A) \tag{1.3}$$

gilt. Diese Eigenschaft wird durch den Begriff *Unabhängigkeit* beschrieben. Genauer heißen zwei Ereignisse A und B *unabhängig*, falls

$$\mathbf{P}(A \cap B) = \mathbf{P}(A)\mathbf{P}(B) \tag{1.4}$$

gilt. Mittels der Definition der bedingten Wahrscheinlichkeit ist es nicht schwer die Äquivalenz von (1.3) und (1.4) zu zeigen.

Im folgenden sei $R = \mathfrak{P}(\Omega)$.

Eine auf Elementarereignissen definierte Funktion X nennen wir *Zufallsvariable*. Eine Zufallsvariable ordnet damit jedem Elementarereignis einen Wert zu. Im folgenden betrachten wir nur Zufallsvariablen $X : \Omega \to \mathbb{R}$.

Beispiel 1.30 Sei $(\Omega, R, \mathbf{P})$ wie in Beispiel 1.27.1. Dann ist z.B. "Die Summe der Würfelaugen" eine Zufallsvariable $X : \Omega \to \mathbb{R}, (i,j) \mapsto i + j$.

Der *Erwartungswert* einer Zufallsvariable X ist der "durchschnittliche" Wert von X. Formal ist der Erwartungswert ein Operator $\mathbf{E} : (\Omega \to \mathbb{R}) \to \mathbb{R}$ mit

$$\mathbf{E}(X) = \sum_{\omega \in \Omega} X(\omega) \cdot \mathbf{P}(\{\omega\})$$

Der Erwartungswert erfüllt die folgenden Eigenschaften:

- $\mathbf{E}$ ist linear, d.h. für $X, Y : \Omega \to \mathbb{R}$ gilt $\mathbf{E}(X+Y) = \mathbf{E}(X) + \mathbf{E}(Y)$ und $\mathbf{E}(\lambda X) = \lambda \mathbf{E}(X)$.
- $\mathbf{E}$ ist monoton, d.h. für $X < Y$ gilt $\mathbf{E}(X) < \mathbf{E}(Y)$.
- $\mathbf{E}$ ist normiert, d.h. $\mathbf{E}(1) = 1$ (die Zufallsvariable 1 bildet alle $\omega \in \Omega$ auf 1 ab).

Seien X und Y zwei Zufallvariablen und $a, b \in \mathbb{R}$. Dann notieren wir das Ereignis $\{\omega \in \Omega \mid X(\omega) = a\}$ mit $\{X = a\}$. Wir nennen X und Y *unabhängige* Zufallsvariablen, falls für alle $a, b \in \mathbb{R}$ die Ereignisse $\{X = a\}$ und $\{Y = b\}$ unabhängig sind, d.h.

$$\mathbf{P}(\{X = a\} \cap \{Y = b\}) = \mathbf{P}(\{X = a\})\mathbf{P}(\{Y = b\})$$

gilt.

Für unabhängige Zufallsvariablen ist der Erwartungswert multiplikativ, d.h. es gilt

$$\mathbf{E}(XY) = \mathbf{E}(X)\mathbf{E}(Y).$$

Neben dem Erwartungswert spielt auch die *Varianz* einer Zufallsvariable eine wichtige Rolle. Sie ist definiert durch $\mathrm{Var}\,(X) = \mathbf{E}((X - \mathbf{E}(X))^2)$. Es gilt auch $\mathrm{Var}\,(X) = \mathbf{E}(X^2) - (\mathbf{E}(X))^2$. Man nennt $\mathbf{E}(X^n)$ auch das n-te *Moment* von X. Damit ergeben sich die folgenden Regeln für $\mathrm{Var}\,(X)$:

1. $\mathrm{Var}\,(X+c) = \mathrm{Var}\,(X)$.
2. $\mathrm{Var}\,(cX) = c^2\mathrm{Var}\,(X)$.
3. Falls die Zufallsvariablen X_i unabhängig sind, so gilt

$$\mathrm{Var}\,(\sum_i X_i) = \sum_i \mathrm{Var}\,(X_i).$$

Beispiel 1.31 Wir berechnen den Erwartungswert und die Varianz der Zufallsvariable X aus Beispiel 1.30 unter der uniformen Wahrscheinlichkeitsverteilung. Da X eine Zufallsvariable ist, die sich als Summe zweier gleichverteilter Zufallsvariablen schreiben läßt, gilt $\mathbf{E}(X) = 2 \cdot \mathbf{E}(Y)$, da diese auch unabhängig sind, gilt $\mathrm{Var}\,(X) = 2\mathrm{Var}\,(Y) = 2(\mathbf{E}(Y^2) - (\mathbf{E}(Y))^2)$. Für $\mathbf{E}(Y)$ bzw. $(\mathbf{E}(Y^2))$ gelten

$$\begin{aligned}\mathbf{E}(Y) &= \sum_{\omega\in\Omega'} Y(\omega)\cdot\frac{1}{6} = \frac{1}{6}(1+2\cdots+6) = \frac{7}{2}\\ \mathbf{E}(Y^2) &= \sum_{\omega\in\Omega'} Y^2(\omega)\cdot\frac{1}{6} = \frac{1}{6}(1+2^2\cdots+6^2) = \frac{91}{6}\end{aligned}$$

Damit ergibt sich $\mathbf{E}(X) = 7$ und $\mathrm{Var}\,(X) = 35/6$.

Wir wollen jetzt noch eine wichtige Ungleichung in der Wahrscheinlichkeitsrechnung beweisen. Sie ist nach dem russischen Mathematiker TSCHEBYSCHEFF benannt.

Satz 1.32 Sei X eine Zufallsvariable mit Erwartungswert $\mathbf{E}(X)$. Dann ist

$$\mathbf{P}(|X - \mathbf{E}(X)| \geq a) \leq \frac{1}{a^2}\mathrm{Var}\,(X).$$

BEWEIS: Sei Y eine neue Zufallsvariable mit Wertebereich $\{0,1\}$ und $Y^{(2)}, Y^{(3)}, \ldots$ Zufallsvariablen mit reellen Wertebereich. Wir definieren

$$Y = \begin{cases} 1 & \text{falls } |X - \mathbf{E}(X)| \geq a \\ 0 & \text{sonst}\end{cases} \qquad \text{und} \quad Y^{(r)} = \left(\frac{|X - \mathbf{E}(X)|}{a}\right)^r \geq Y.$$

Damit ist

$$\mathbf{P}(|X - \mathbf{E}(X)| \geq a) = \mathbf{P}(Y = 1) = \mathbf{E}(Y).$$

Wegen $Y^{(r)} \geq Y$ folgt auch $\mathbf{E}(Y^{(r)}) \geq \mathbf{E}(Y)$ und damit

$$\mathbf{P}(|X - \mathbf{E}(X)| \geq a) \leq \mathbf{E}((Y^{(r)}) = \frac{1}{a^r}\mathbf{E}(|X - \mathbf{E}(X)|^r).$$

Man nennt diese Gleichung auch TSCHEBYSCHEFF*sche Ungleichung r-ter Art.* Für $r = 2$ ist

$$\mathbf{E}(Y^{(2)}) = \mathrm{Var}\,(X)/a^2.$$

□

Die Ungleichungen von TSCHEBYSCHEFF gelten für beliebige Zufallsvariablen. Für eine besondere Klasse von Zufallsvariablen, sogenannte binomialverteilte Zufallsgrößen gelten schärfere Abschätzungen.

Dazu sei A ein Ereignis in Ω mit Wahrscheinlichkeit p. Seien $X_1, \ldots, X_n$ unabhängige, gleichverteilte Zufallsvariablen definiert durch

$$X_i(B) = \begin{cases} 1 & \text{falls } B = A \\ 0 & \text{sonst} \end{cases}$$

und $S = \sum_{i=1}^{n} X_i$. Dann heißt S *binomialverteilte* Zufallsvariable (S ist eine Zufallsvariable über Ω^n) mit Parametern n und p. Es gilt $\mathbf{P}(S = k) = \binom{n}{k} p^k (1-p)^{n-k}$. Die Zufallsvariable X_i heißt auch Bernoulli-Variable, die Auswertung heißt Bernoulli-Versuch (z.B. Werfen einer Münze). Es gelten die folgenden leicht zu verifizierenden Formeln:

$$\begin{aligned} \mathbf{E}(X_i) &= p & \mathrm{Var}\,(X_i) &= p(1-p) \\ \mathbf{E}(S) &= np & \mathrm{Var}\,(S) &= np(1-p). \end{aligned}$$

Nach CHERNOFF (siehe z.B. [HaRu90]) sind die folgenden Ungleichungen für binomialverteilte Zufallsvariablen benannt.

Satz 1.33 Sei S eine binomialverteilte Zufallsvariable mit Parametern n und p. Dann gilt für jedes $0 \leq \alpha \leq 1$

$$\begin{aligned} \mathbf{P}(S \leq (1-\alpha)np) &\leq e^{-\alpha^2 np/2} \\ \mathbf{P}(S \geq (1+\alpha)np) &\leq e^{-\alpha^2 np/3}. \end{aligned}$$

Kapitel 2

Automatentheorie

Der Inhalt dieses Kapitels ist es, spezielle Mengen von Zeichenketten zu untersuchen. Dabei werden wir verschiedene mathematische Werkzeuge zur Untersuchung dieser Mengen entwikkeln. Zunächst werden wir den Begriff der sogenannten formalen Sprachen kennenlernen. In den weiteren Abschnitten dieses Kapitels werden wir einfach strukturierte Mengen, die die Eigenschaften einer Booleschen Algebra besitzen, und mehrere Mechanismen zur Berechnung bzw. Charakterisierung solcher Mengen kennenlernen.

2.1 Formale Sprachen

Das Rechnen mit Zahlen, z.B. die Addition von zwei natürlichen Zahlen oder ähnliches, kann auch als Manipulation der Zeichenketten verstanden werden, die diese Zahlen repräsentieren. Allgemein kann jede Berechnung auf die Manipulation von Zeichenketten, die gewisse Symbole (z.B. die Ziffern zwischen Null, Eins bis Neun) enthalten, zurückgeführt werden. Diese Manipulationen haben auf eine gewisse Teilmenge der Zeichenketten eine besondere Wirkung (z.B. die Zeichenketten, die gerade Zahlen im Binärsystem darstellen, enden mit dem Zeichen Null). Inhalt dieses Abschnittes ist es, formal solche Teilmengen zu definieren und einige wichtige Operationen auf diesen Zeichenketten einzuführen.

Eine Menge Σ von Symbolen σ nennen wir *Alphabet*.

Aus den Elementen des Alphabets lassen sich sowohl endliche als auch unendlich lange Zeichenketten bilden. Eine solche Zeichenkette bezeichnet man als *Wort* über Σ oder im Falle von unendlichlangen Zeichenketten ω-*Wort* (über Σ). Ein Wort v der Länge n ist dabei formal eine Funktion $v : \{0, \ldots, n-1\} \to \Sigma$, wobei $v(i)$ das $i+1$-te Zeichen von v ist (der Einfachheit halber bezeichen wir $\{0, \ldots, n-1\}$ mit $\underline{n}$, falls Zweideutigkeiten ausgeschlossen sind). Das Wort $v : \underline{0} \to \Sigma$, das kein Zeichen enthält, heißt auch Λ. Ein ω-Wort v ist eine Funktion $v : \mathbb{N}_0 \to \Sigma$. $v(n_1, n_2)$ bezeichnet das endliche Teilwort w der Länge $n_2 - n_1$, mit $w(i) = v(n_1 + i)$ für $0 \leq i < n_2 - n_1$. Im folgenden bezieht sich die Bezeichnung Wort nur auf endlich lange Wörter.

Wir bezeichnen die Menge aller Wörter der Länge n über Σ mit $\Sigma^n = \{v \mid v : \underline{n} \to \Sigma\}$; dabei identifizieren wir Σ mit $\Sigma^1 = \{v \mid v : \underline{1} \to \Sigma\}$. Alle endlich langen Wörter über Σ bilden die Menge $\Sigma^* = \bigcup_{n \geq 0} \Sigma^n$. Desweiteren ist $\Sigma^+ = \bigcup_{n>0} \Sigma^n$ die Menge aller endlichen Wörter außer dem leeren Wort Λ und $\Sigma^\omega = \{v \mid v : \mathbb{N}_0 \to \Sigma\}$ die Menge aller ω-Wörter über Σ. Die Länge eines Wortes v wird mit $|v|$ bezeichnet. Dabei gilt $|v| = n$ genau dann, wenn $v \in \Sigma^n$ ist, und

$|v| = \infty$, falls $v \in \Sigma^\omega$.

Es sei ausdrücklich darauf hingewiesen, daß ein Wort kein n-Tupel sondern eine Funktion ist.

Neben den Mengen Σ^n, $n \geq 0$, Σ^*, Σ^+ und Σ^ω interessieren wir uns noch für Teilmengen dieser Mengen.

Definition 2.1 Eine Teilmenge $L \subseteq \Sigma^*$ nennt man eine *Sprache* über Σ. L ist eine beliebige Menge von Wörtern über Σ. Eine Teilmenge $L' \subseteq \Sigma^\omega$ heißt ω-*Sprache*.

Da Sprachen Mengen von Wörtern sind, können alle mengentheoretischen Operationen auf Sprachen angewandt werden. Es gibt jedoch noch einige weitere interessante Operationen. Darunter befindet sich die *Konkatenation* von zwei Sprachen. Formal ist

$$\begin{aligned} \text{conc} : \Sigma^* \times \Sigma^* &\rightarrow \Sigma^* \\ (v, w) &\mapsto z := vw, \end{aligned}$$

wobei für $v \in \Sigma^n$, $w \in \Sigma^m$, $z \in \Sigma^{n+m}$ mit $z(i) = v(i)$ für $0 \leq i < n$ und $z(i) = w(i-n)$ für $n \leq i < n+m$ gilt. Die Konkatenation von zwei Sprachen ist dann definiert durch

$$\begin{aligned} \text{CONC} : \mathfrak{P}(\Sigma^*) \times \mathfrak{P}(\Sigma^*) &\rightarrow \mathfrak{P}(\Sigma^*) \\ (L_1, L_2) &\mapsto \{xy \mid x \in L_1, y \in L_2\} =: L_1 L_2. \end{aligned}$$

Mittels der Konkatenation können aus Sprachen $L_i \subseteq \Sigma^*$ weitere Sprachen definiert werden. Sei $L^0 = \{\Lambda\}$ und $L^n = LL^{n-1} = \text{CONC}\,(L, L^{n-1})$. Die *Kleenesche Hülle* von L ist die Menge aller Wörter v, so daß $v \in L^n$ für ein $n \geq 0$ ist. Formal gilt

$$L^* = \bigcup_{n \geq 0} L^n.$$

Der Operator * wird auch als *Kleenescher Sternoperator* bezeichnet. In der Menge L^* ist auch das leere Wort enthalten ($L^0 = \{\Lambda\}$). Analog kann auch

$$L^+ = \bigcup_{n > 0} L^n = LL^*$$

definiert werden. Man beachte, daß $L^+ = L^*$, falls $\Lambda \in L$, sonst gilt $L^+ = L^* \setminus \{\Lambda\}$.

Wir können auch die Konkatenation $L_1 L_2$ zwischen einer Sprache $L_1 \subseteq \Sigma^*$ und einer ω-Sprache $L_2 \subseteq \Sigma^\omega$ analog zu oben definieren. Man beachte, daß $L_1 L_2$ wieder eine ω-Sprache ist. Die Konkatenation $L_1 L_2$ zwischen einer ω-Sprache L_1 und einer anderen Sprache $L_2 \subseteq \Sigma^*$ oder $L_2 \subseteq \Sigma^\omega$ ist nicht definiert.

Der ω-Operator bildet eine Sprache auf ihre unendlichfache Konkatentation ab. Formal ist

$$L^\omega = \{v \in \Sigma^\omega \mid \exists\, (n_i)_{i \in \mathbb{N}_0} : \forall\, i \in \mathbb{N} : v(n_i, n_{i+1}) \in L\}.$$

2.2 BNF: ein syntaktisches Beschreibungsmittel

In diesem Abschnitt werden wir die sogenannte Backus-Naur-Form besprechen, die 1962 von John Backus und Peter Naur [BaNa62] eingeführt worden ist. Eine Backus-Naur-Form

(kurz: BNF) ist ein Regelsystem, mit Hilfe dessen Wörter über dem Alphabet Σ gebildet werden können. Die Menge aller Wörter, die mit einer BNF B abgeleitet werden können, ist eine Sprache. Die Mengen aller Sprachen, die durch BNFs beschrieben werden können, sind die Klasse der *kontextfreien Sprachen*. In diesem Zusammenhang nennt man das Regelsystem der BNF auch *kontextfreie Grammatik*. Grammatiken anderer Form führen zu anderen Klassen von Sprachen. Wir wollen hier jedoch nicht auf die allgemeine Theorie von Grammatiken eingehen, sondern uns auf die BNF beschränken.

Zunächst werden wir die BNF formal definieren.

Definition 2.2 Eine BNF B ist ein 4-Tupel $(\Sigma, V, \underline{S}, R)$, wobei

1. Σ ein Alphabet ist. Die von B beschriebene Sprache L ist eine Teilmenge von Σ^*. Die Elemente von Σ nennt man auch *Terminalsymbole*.
2. V eine endliche Menge von (syntaktischen) Variablen ist. Die Elemente von V nennt man auch *Nichtterminalsymbole* . Zur Unterscheidung von Terminal- und Nichtterminalsymbolen unterstreichen wir Nichtterminalsymbole und setzen sie ggf. in spitze Klammern.
3. $\underline{S} \in V$ das Startsymbol ist.
4. R eine endliche Menge von Regeln (Produktionen) der Form $\underline{T} ::= \alpha_1|\alpha_2|\cdots|\alpha_k$ ist, wobei $\underline{T} \in V$ eine Variable und $\alpha_1, \ldots, \alpha_k \in (V \cup \Sigma)^*$ beliebige endliche Zeichenketten aus Variablen und Terminalsymbolen sind ($k = 1$ ist erlaubt). Diese Zeichenketten nennen wir *Clausen*. Diese Regel ist äquivalent zu den k Regeln $\underline{T} ::= \alpha_i$ für $1 \leq i \leq k$.

Die Symbole "|" und "::=" sind sogenannte *Metasymbole*.

Mittels der Regeln R lassen sich Wörter $\alpha \in (V \cup \Sigma)^*$ modifizieren. Wörter $\alpha \in (V \cup \Sigma)^*$ nennen wir auch *Satzformen*. Sei $\alpha = \gamma\underline{T}\delta$ mit $\gamma, \delta \in (V \cup \Sigma)^*$ eine Satzform und $\underline{T} \in V$ und $\underline{T} ::= \beta_1|\beta_2|\cdots|\beta_k$ eine Regel aus R, dann lassen sich die Wörter $\alpha_1 = \gamma\beta_1\delta, \ldots, \alpha_k = \gamma\beta_k\delta$ aus α *ableiten*. Wir schreiben dann auch $\alpha \to \alpha_i$ für $1 \leq i \leq k$ und nennen den Übergang von α nach α_i *Ableitung*. Falls $\alpha \to \alpha' \to \cdots \to \beta$ gilt, schreiben wir auch $\alpha \stackrel{*}{\to} \beta$. $\stackrel{*}{\to}$ ist also die transitive Hülle von $\to$. Die Menge aller aus $\underline{S}$ ableitbaren Satzformen bezeichnen wir mit $S(B)$. Dies ist die kleinste Teilmenge von $(V \cup \Sigma)^*$ für die gilt:

1. $\underline{S} \in S(B)$
2. Aus $\gamma\underline{A}\delta \in S(B)$ und $\underline{A} ::= \beta_1|\beta_2|\cdots|\beta_k \in R$ folgt $\gamma\beta_i\delta \in S(B)$ für alle $0 \leq i \leq k$.

Nun können wir die von B beschriebene Sprache definieren.

Definition 2.3 Sei B eine BNF. Dann ist

$$L(B) := \{\alpha \in \Sigma^* \mid \underline{S} \stackrel{*}{\to} \alpha\} = S(B) \cap \Sigma^*$$

die durch B beschriebene Sprache.

Beispiel 2.4 Sei $B = (\Sigma, V, \underline{S}, R)$ mit $\Sigma = \{a, b\}$, $V = \{\underline{S}, \underline{A}, \underline{B}\}$ und $R = \{\underline{S} ::= a\underline{B}|b\underline{A}, \underline{A} ::= a|a\underline{S}|b\underline{A}\underline{A}, \underline{B} ::= b|b\underline{S}|a\underline{B}\underline{B}\}$. Dann ist $L(B) = \{ab, ba, abab, \ldots\}$ gerade die Menge aller Wörter über $\{a, b\}$, die dieselbe Anzahl von as und bs haben (Beweis siehe [HoUl79, Example 4.3]).

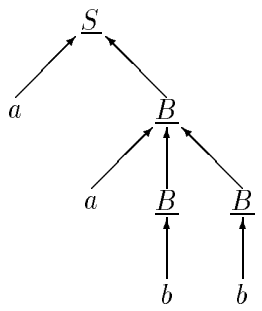


Abbildung 2.1: Ableitungsbaum für $\underline{S} \stackrel{*}{\rightarrow} aabb$

Wir bemerken, daß die Zusammenhangskomponenten des durch $\rightarrow$ induzierten (unendlichen) Graphen (mit Knotenmenge $(V \cup \Sigma)^*$) ein (unendlicher) Baum ist (unter der Voraussetzung, daß keine Produktionen der Form $\underline{A} ::= \underline{B}$ vorkommen). Hierauf wollen wir jedoch nicht näher eingehen. Wichtiger für uns ist der Begriff des *Ableitungsbaums*. Formal ist ein Ableitungsbaum für eine BNF $B = (\Sigma, V, \underline{S}, R)$ ein gelabelter gerichteter Baum, für den folgendes gilt:

1. Jeder Knoten ist mit einer Marke aus $V \cup \Sigma$ gelabelt.
2. Die Wurzel ist mit $\underline{S}$ gelabelt.
3. Innere Knoten, d.h. Knoten v mit $\deg_{\mathrm{OUT}}(v) > 0$, sind mit Marken aus V gelabelt.
4. Wenn ein Knoten v mit $\underline{\mathrm{A}}$ gelabelt ist, und die Söhne von links nach rechts (diese Anordnung kann durch gelabelte Kanten erreicht werden) mit $a_1, a_2, \cdots, a_k$ markiert sind, dann muß $A ::= a_1 a_2 \ldots a_k$ eine Produktion in R sein.

Ein Ableitungsbaum für eine Satzform α ist ein Ableitungsbaum, so daß die Blätter von links nach rechts gelesen α ergeben. Ein Ableitungsbaum für eine Satzform α ist im allgemeinen nicht eindeutig. Es gibt sogar Sprachen, so daß jede Grammatik, die diese Sprache erzeugt, mehrere Ableitungen für ein Wort der Sprache besitzt (siehe [HoUl79, Chapter 4.7]).

Abbildung 2.1 zeigt einen Ableitungsbaum für $\underline{S} \stackrel{*}{\rightarrow} aabb$ aus Beispiel 2.4.

2.2.1 Interpretation von Wörtern

Bisher haben wir nur die syntaktische Manipulation von Zeichenketten aus Σ^* untersucht. Nun möchten wir den syntaktischen Ausdrücken eine Bedeutung zuweisen und untersuchen, wie sich die Manipulation von Zeichenketten auf ihre Bedeutung auswirkt.

Beispiel 2.5 Sei $\Sigma = \{\underline{0}, \underline{1}, \ldots, \underline{9}\}$, so möchten wir z.B. dem Wort $\underline{0109}$ die natürliche Zahl 109 zuweisen. Die Manipulation von Wörtern entsprechen dann arithmetischen Operationen, Sprachen entsprechen Zahlenmengen mit gewissen Eigenschaften, etc.

Zu diesem Zweck führen wir eine semantische Funktion $\mathcal{N}_B$ ein, deren Definitionsbereich die Menge der Satzformen $S(B)$ ist. Dabei werden nur Elementen aus $L(B) \subseteq S(B)$ Werte zugewiesen. Der Wertebereich kann je nach Interpretation z.B. die Menge der natürlichen Zahlen, eine Menge von Funktionen, etc. sein.

Beispiel 2.6 Sei $B = (\Sigma = \{a_0, a_1\}, V = \{\underline{S}\}, \underline{S}, R = \{\underline{S} ::= a_0|a_1|\underline{S}a_0|\underline{S}a_1\})$. Die Sprache $L(B)$ ist Σ^*. Wir wollen die Elemente von $L(B)$ als natürliche Zahlen in Binärdarstellung interpretieren. Zu diesem Zweck definieren wir

$$\mathcal{N}(a_0) = 0 \quad \mathcal{N}(a_1) = 1$$

und

$$\mathcal{N}(\underline{S}a_0) = 2 \cdot \mathcal{N}(\underline{S}) \quad \mathcal{N}(\underline{S}a_1) = 2 \cdot \mathcal{N}(\underline{S}) + 1.$$

Man sieht leicht, daß diese Definition der semantische Funktion $\mathcal{N}$ das Gewünschte erreicht.

Wir werden später noch semantischen Funktionen im Zusammenhang mit regulären Ausdrücken und Programmiersprachen begegnen. Eine semantische Funktion φ ordnet dabei z.B. einem Ausdruck E in Abhängigkeit der Werte der Variablen einen Wert zu, oder, falls der Ausdruck nicht definiert ist, den speziellen Wert "error". Näheres dazu jedoch erst im Abschnitt 7.1.

2.2.2 Erweiterte BNF

In diesem Abschnitt wollen wir die Definition der Backus-Naur-Form ein wenig erweitern und das Metazeichen * analog zum Kleeneschen Sternoperator einführen. Dies erweitert jedoch nicht die Menge der so beschreibbaren Sprachen (siehe Satz 2.8), erleichtert jedoch die Notation.

Definition 2.7 Eine erweiterte BNF B (EBNF) ist ein 4-Tupel $B = (\Sigma, V, \underline{S}, R)$, wobei Σ, V und $\underline{S}$ wie in Definition 2.2 definiert sind (Σ darf nicht die Zeichen $(,)^*, \{, \}$ enthalten), und R eine endliche Menge von Regeln der Form $\underline{T} ::= \alpha_1|\alpha_2|\cdots|\alpha_k$ und $\alpha_i = \beta_{i,1}\beta_{i,2}\cdots\beta_{i,l}$ mit $\beta_{i,j} = a_1 \ldots a_s$ oder $\beta_{i,j} = (a_1 \ldots a_s)^*$ für $a_r \in V \cup \Sigma^*$ sind. Statt der Zeichen $()^*$ kann synomym auch $\{\}$ verwendet werden.

Eine Satzform, die eines der neuen Metasymbole enthält, kann wie folgt abgeleitet werden. Sei $\alpha = \gamma(\beta)^*\delta$. Dann gilt $\alpha \to \gamma\delta$, $\alpha \to \gamma\beta\delta$, $\alpha \to \gamma\beta\beta\delta, \ldots$.

Satz 2.8 Sei L eine Sprache, die durch eine EBNF B beschrieben werden kann. Dann ist L auch durch eine BNF B' beschreibbar.

BEWEIS: Seien $\alpha_1, \ldots, \alpha_k$ die Clausen, die die Metasymbole $(,)^*$ enthalten, und seien $\beta_1, \ldots, \beta_l$ die Zeichenketten in $\alpha_1, \ldots, \alpha_k$, die durch $(,)^*$ umklammert werden. Wir formen die BNF B' wie folgt. Zuerst erweitern wir die Menge der Variablen um die neuen Variablen $\underline{T_1}, \ldots, \underline{T_l}$, d.h. $V' = V \cup \{\underline{T_1}, \ldots, \underline{T_l}\}$ (disjunkte Vereinigung). Dann ersetzen wir jedes Vorkommnis von $(\beta_j)^*$ in einer Clause α_i durch $\underline{T_j}$ (auch für geschachtelte Clausen). Nun fügen wir die Regeln $\underline{T_j} ::= \Lambda|\beta_j\underline{T_j}$ zu R hinzu. Dies bildet die Menge R'. □

Wir wollen das Prinzip von BNFs noch einmal an Hand von Klammerausdrücken erläutern. Ein *Klammerausdruck* v ist ein Wort aus $\{(,)\}^*$, so daß die Anzahl von "(" gleich der Anzahl von ")" ist und für jedes $m < n$ das Teilwort $v(0, m)$ mindestens soviele "(" enthält wie ")".

Beispiel 2.9 Die BNF $B = (\{(,)\}, \{\underline{W}\}, \underline{W}, \{\underline{W} ::= \Lambda|(\underline{W}\underline{W})\})$ erzeugt Klammerausdrücke, die *Well Formed Forms* (WFF) genannt werden. So sind z.B. () und ((()())) Well Formed Forms, nicht jedoch der Klammerausdruck (()()()). Abbildung 2.2 zeigt einen Ableitungsbaum für den Ausdruck ((())).

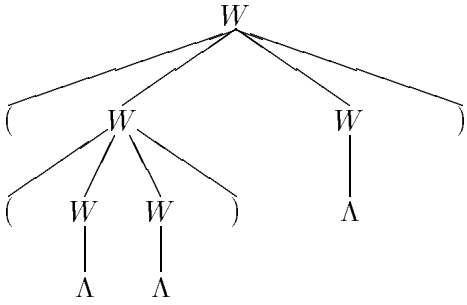


Abbildung 2.2: Ableitungsbaum für ((()))

Die Menge $T(\Sigma, V)$ sei die Menge aller Terme, die durch Benutzen von Symbolen, Variablen und Metasymbolen gebildet werden können. Die Terme aus $T(\Sigma, V)$ heißen *EBNF-Terme* und sind mögliche rechte Seiten für Produktionsregeln. Formal ist die Menge der EBNF-Terme die kleinste Menge, für die gilt:

1. $V \cup \Sigma^* \subset T(\Sigma, V)$.
2. Falls $\alpha \in T(\Sigma, V)$ ist, so gilt auch $(\alpha)^* \in T(\Sigma, V)$.
3. Falls $\alpha_1, \ldots, \alpha_k \in T(\Sigma, V)$ sind, so sind auch $\alpha_1|\alpha_2|\ldots|\alpha_k \in T(\Sigma, V)$ und $\alpha_1\alpha_2\ldots\alpha_k \in T(\Sigma, V)$.

Nun können wir eine zu Definition 2.7 äquivalente Definition geben.

Definition 2.10 Eine B (EBNF) ist ein 4-Tupel $B = (\Sigma, V, \underline{S}, R)$, wobei Σ, V und $\underline{S}$ wie in Definition 2.2 definiert sind, und R eine endliche Menge von Regeln der Form $\underline{T} ::= \alpha$ mit $\alpha \in T(\Sigma, V)$ ist.

Wir können einem Term $\alpha \in T(\Sigma, V)$ die Menge von Wörtern aus Σ^* zuordnen, die aus α ableitbar sind. Diese Zuordnung kann als eine "metasemantische" Funktion $\varphi : T(\Sigma, V) \to \mathfrak{P}(\Sigma^*)$ interpretiert werden. φ ist induktiv folgendermaßen definiert:

1. Für $\underline{S} \in V$ gilt $\varphi(\underline{S}) = \cup\varphi(\alpha)$, wobei die Vereinigung über alle α, für die eine Produktionsregel $\underline{S} ::= \alpha$ in R existiert, läuft.
2. Für $\alpha \in \Sigma^*$ gilt $\varphi(\alpha) = \{\alpha\}$.
3. Für $\alpha \in T(\Sigma, V)$ gilt $\varphi((\alpha)^*) = (\varphi(\alpha))^*$.
4. Für $\alpha_1, \ldots, \alpha_k \in T(\Sigma, V)$ gilt $\varphi(\alpha_1|\ldots|\alpha_k) = \varphi(\alpha_1) \cup \cdots \cup \varphi(\alpha_k)$.
5. Für $\alpha_1, \ldots, \alpha_k \in T(\Sigma, V)$ gilt $\varphi(\alpha_1\ldots\alpha_k) = \varphi(\alpha_1)\varphi(\alpha_2)\cdots\varphi(\alpha_k)$.

Mit Hilfe der Funktion φ läßt sich die durch B definierte Sprache als $L(B) = \varphi(\underline{S})$ schreiben.

Wir wollen diesen Abschnitt mit einem Beispiel abschliessen.

Beispiel 2.11 In Programmiersprachen ist ein Variablenname oft als eine Zeichenkette aus Buchstaben und Ziffern definiert, die mit einem Buchstaben anfangen muß. Wir möchten nun eine syntaktische Definition von Variablennamen mittels einer BNF angeben. Zu diesem Zweck sei LETTER $= \{a, \ldots, z, A, \ldots, Z\}$ die Menge der Buchstaben und DIGIT $= \{0, \ldots, 9\}$ die Menge der Ziffern (Ziffern sind Symbole, keine Zahlen!). Das Grundalphabet Σ sei LETTER $\cup$ DIGIT, die syntaktischen Variablen $\underline{I}$,$\underline{D}$,$\underline{L}$, das Startsymbol $\underline{I}$ und die BNF Regeln wie folgt: $\underline{D} ::= 0|1|\ldots|9$, $\underline{L} ::= a|\ldots|z|A|\ldots|Z$ und $\underline{I} ::= \underline{L}(\underline{L}|\underline{D})^*$. Mittels der semantischen Funktion φ ergeben sich $\varphi(\underline{L}) =$ LETTER, $\varphi(\underline{D}) =$ DIGIT, $\varphi((\underline{L}|\underline{D})^*) =$ (LETTER $\cup$ DIGIT)* und

$$\varphi(\underline{I}) = \text{LETTER}(\text{LETTER} \cup \text{DIGIT})^*.$$

2.3 Endliche Automaten und reguläre Ausdrücke

In diesem Abschnitt werden wir die Menge der regulären Sprachen kennenlernen, die eine zentrale Rolle in der Theorie formaler Sprachen spielt. Zur Definition der regulären Sprachen werden wir zwei unterschiedliche Vorgehensweisen betrachten, zum einen die durch die Grammatik der regulären Ausdrücke *erzeugten* Sprachen, zum anderen die durch endliche Automaten *erkannten* Sprachen. Beide Definitionen werden sich als äquivalent herausstellen.

2.3.1 Reguläre Ausdrücke

Definition 2.12 Sei $\Sigma = \{a_1, \ldots, a_n\}$ ein Alphabet. Die Menge der regulären Ausdrücke RE (regular expressions) über Σ wird durch die von der BNF

$$(\Sigma \cup \{+, (,), ^*\} \cup \{\emptyset, \Lambda\}, \{< \text{RE} >\}, < \text{RE} >, \{r\}) \quad \text{mit der Regel } r\text{:}$$

$$< \text{RE} > ::= a_1 \mid \ldots \mid a_n \mid (< \text{RE} > + < \text{RE} >) \mid (< \text{RE} >< \text{RE} >) \mid (< \text{RE} >)^* \mid \Lambda \mid \emptyset$$

beschriebenen Sprache gegeben.

Durch die semantische Funktion $\varphi : \text{RE} \to \mathfrak{P}(\Sigma^*)$ werden regulären Ausdrücken Sprachen über Σ zugeordnet:

i) $\varphi(a) = \{a\}$ für $a \in \Sigma$.

ii) $\varphi(R_1 + R_2) = \varphi(R_1) \cup \varphi(R_2)$ für $R_1, R_2 \in \text{RE}$

iii) $\varphi(R_1 R_2) = \varphi(R_1)\varphi(R_2)$ für $R_1, R_2 \in \text{RE}$

iv) $\varphi((R)^*) = (\varphi(R))^*$ für $R \in \text{RE}$

v) $\varphi(\Lambda) = \{\Lambda\}$

vi) $\varphi(\emptyset) = \emptyset$

Als Konvention schreiben wir $\varphi((R)) = \varphi(R)$ für $R \in \text{RE}$.

Man beachte, daß das Symbol $a \in \Sigma$ in einem regulären Ausdruck die Menge $\{a\}$ und nicht das Zeichen a bezeichnet. Um genau zwischen dem Alphabet Σ und dem Alphabet der BNF zu unterscheiden, werden die Symbole des regulären Ausdrucks gelegentlich durch Unterstreichung oder Fettdruck hervorgehoben.

Beispiel 2.13 Die durch den regulären Ausdruck $(0+1)^*$ gegebene Sprache ist

$$\varphi((0+1)^*) = (\varphi(0+1))^* = (\varphi(0) \cup \varphi(1))^* = (\{0\} \cup \{1\})^* = \{0,1\}^*.$$

Definition 2.14 Sei $\Sigma = \{a_1, \ldots, a_n\}$ ein Alphabet. Die Menge der ω–regulären Ausdrücke RE^ω über Σ ist durch die von der BNF

$$(\Sigma \cup \{+, (,), ^*, ^\omega\} \cup \{\emptyset\}, \{< \mathrm{RE}^\omega >, < \mathrm{RE}^+ >\}, < \mathrm{RE}^\omega >, R) \quad \text{mit den Regeln}$$

$$\begin{aligned} < \mathrm{RE}^\omega > ::= {} & (< \mathrm{RE}^\omega > + < \mathrm{RE}^\omega >) \mid (< \mathrm{RE} >< \mathrm{RE}^\omega >) \mid (< \mathrm{RE}^+ >)^\omega \\ < \mathrm{RE}^+ > ::= {} & a_1 \mid \ldots \mid a_n \mid (< \mathrm{RE}^+ > + < \mathrm{RE}^+ >) \mid (< \mathrm{RE}^+ >< \mathrm{RE} >) \mid \\ & ((< \mathrm{RE}^+ >)^* < \mathrm{RE}^+ >) \mid (< \mathrm{RE} >< \mathrm{RE}^+ >) \mid \emptyset \end{aligned}$$

beschriebenen Sprache gegeben.

Die Semantik ω–regulärer Ausdrücke ist durch die semantische Funktion $\varphi : \mathrm{RE}^\omega \to \mathfrak{P}(\Sigma^\omega)$ gegeben, die wie folgt definiert ist:

i) $\varphi((R)^\omega) = (\varphi(R))^\omega$ für $R \in \mathrm{RE}^+$

ii) $\varphi(R_1 R_2) = \varphi(R_1)\varphi(R_2)$ für $R_1 \in \mathrm{RE}$, $R_2 \in \mathrm{RE}^\omega$

iii) $\varphi(R_1 + R_2) = \varphi(R_1) \cup \varphi(R_2)$ für $R_1, R_2 \in \mathrm{RE}^\omega$

Wir wollen zwei reguläre (bzw. ω–reguläre) Ausdrücke R_1, R_2 als äquivalent ansehen, wenn sie die gleiche Sprache repräsentieren, d.h.

$$R_1 = R_2 \quad \Longleftrightarrow \quad \varphi(R_1) = \varphi(R_2)$$

Hiermit können wir die folgenden "Rechenregeln" für reguläre Ausdrücke formulieren.

Für alle $R_1, R_2, R_3 \in \mathrm{RE}$ bzw. $R_1, R_2, R_3 \in \mathrm{RE}^\omega$ gilt:

1. $R_1 + R_2 = R_2 + R_1$
   $R_1 + \emptyset = \emptyset + R_1$
   $R_1 + R_1 = R_1$
   $(R_1 + R_2) + R_3 = R_1 + (R_2 + R_3)$

2. $R_1 \Lambda = \Lambda R_1 = R_1$
   $R\emptyset = \emptyset R = \emptyset$
   $(R_1 R_2) R_3 = R_1 (R_2 R_3)$ *aber* $R_1 R_2 \neq R_2 R_1$

3. $R_1(R_2 + R_3) = R_1 R_2 + R_1 R_3$
   $(R_1 + R_2) R_3 = R_1 R_3 + R_2 R_3$

4. $R_1^* = R_1^* R_1^* = (R_1^*)^* = (\Lambda + R_1)^*$
   $\emptyset^* = \Lambda^* = \Lambda$

5. $R_1^* = \Lambda + R_1 + R_1^2 + R_1^3 + \ldots + R_1^* + R_1^{k+1} R_1^* \quad (k \geq 0)$
   $R_1^* = \Lambda + R_1 R_1^*$

6. $(R_1 + R_2)^* = (R_1^* + R_2^*)^* = (R_1^* R_2^*)^* = (R_1^* R_2)^* R_1^* = R_1^* (R_2 R_1^*)^*$

7. $R_1^* R_1 = R_1 R_1^*$
   $R_1(R_2 R_1)^* = (R_1 R_2)^* R_1$
8. $(R_1^* R_2)^* = \Lambda + (R_1 + R_2)^* R_2$
   $(R_1 R_2^*)^* = \Lambda + R_1(R_1 + R_2)^*$
9. $R_1 = R_2 R_1 + R_3 \Leftrightarrow R_1 = R_2^* R_3$
   $R_1 = R_1 R_2 + R_3 \Leftrightarrow R_1 = R_3 R_2^*$

9. heißt auch "Arden Regel".

2.3.2 Endliche Automaten

Wir werden nun das mathematische Modell des endlichen Automaten einführen. Man kann sich die Arbeitsweise eines endlichen Automaten folgenderweise vorstellen. Zunächst befindet sich der Automat in einem ausgezeichneten Zustand, dem sogenannten Startzustand. Bei Eingabe eines Wortes w arbeitet der Automat w zeichenweise ab, und verändert seinen internen Zustand in Abhängigkeit von dem soeben gelesenen Zeichen und seinem aktuellen Zustand. Die Menge der möglichen internen Zustände ist dabei endlich. Der wohldefinierte Zustand nach vollständigem Lesen von w dient dann als Entscheidung für die Akzeptanz von w. Wir geben nun die formale Definition an:

Definition 2.15 Ein *deterministischer endlicher Automat* (EA) über Σ ist ist ein Quadrupel $\mathcal{A} = (S, \delta, s_0, F)$, wobei

- S eine endliche Menge von Zuständen,
- $\delta : S \times \Sigma \to S$ die (totale) Zustandsübergangsfunktion,
- $s_0 \in S$ den Anfangszustand und
- $F \subseteq S$ die Menge der Endzustände (akzeptierenden Zustände) bezeichnet.

Sei $\mathcal{A}$ ein EA und $v : \underline{n} \to \Sigma$ ein Wort über Σ. Die Funktion $\rho_v : \underline{n+1} \to S$ mit

i) $\rho_v(0) = s_0$ und

ii) $\rho_v(i+1) = \delta(\rho_v(i), v(i))$ für $i \in \underline{n}$.

heißt die $\mathcal{A}$–Berechnung über v. ρ_v heißt akzeptierend, falls zusätzlich

iii) $\rho_v(n) \in F$.

Falls ρ_v akzeptierend ist, so sagen wir auch, daß der Automat $\mathcal{A}$ das Wort v akzeptiert (berechnet). Die Menge der von $\mathcal{A}$ akzeptierten Wörter bezeichnen wir als die von $\mathcal{A}$ akzeptierte (berechnete) Sprache und notieren sie durch $L(\mathcal{A})$.

Wir werden nun das Modell des endlichen Automaten erweitern, indem wir statt einem Folgezustand mehrere Folgezustände erlauben. Der Automat führt nun alle möglichen Zustandsübergänge aus, dupliziert sich dabei selber, sodaß für jeden Folgezustand eine eigene Instanz des Automaten existiert. Der Automat akzeptiert genau dann, wenn zumindest eine seiner nebeneinander existierenden Instanzen einen akzeptierenden Zustand erreicht hat.

Definition 2.16 Ein *nichtdeterministischer* endlicher Automat (NEA) über Σ ist ein Quadrupel $\mathcal{A} = (S, \delta, S_0, F)$, wobei

- S eine endliche Menge von Zuständen,
- $\delta : S \times \Sigma \to \mathfrak{P}(S)$ die Zustandsübergangsfunktion,
- $S_0 \subseteq S$ die Menge der Anfangszustände und
- $F \subseteq S$ die Menge der Endzustände (akzeptierenden Zustände) bezeichnet.

Sei $v : \underline{n} \to \Sigma$. Eine Funktion $\rho : \underline{n+1} \to S$ heißt $\mathcal{A}$–Berechnung über v falls

i) $\rho(0) \in S_0$ und

ii) $\rho(i+1) \in \delta(\rho(i), v(i))$ für $i \in \underline{n}$.

ρ heißt akzeptierend, falls zusätzlich

iii) $\rho(n) \in F$.

Der Automat $\mathcal{A}$ akzeptiert (berechnet) das Wort v genau dann, wenn eine akzeptierende $\mathcal{A}$–Berechnung über v existiert. Die von $\mathcal{A}$ akzeptierte (berechnete) Sprache $L(\mathcal{A})$ ist dann die Menge der von $\mathcal{A}$ akzeptierten Wörter.

Man beachte, daß ein NEA $\mathcal{A} = (S, \delta, S_0, F)$ genau dann deterministisch ist, falls $|S_0| = 1$ und $|\delta(s, \sigma)| = 1$ für jedes $s \in S$ und $\sigma \in \Sigma$ gilt. Gelegentlich werden wir sogar $|\delta(s, \sigma)| \leq 1$ zulassen. Dies bedeutet natürlich keine Einschränkung, da wir durch die Einführung eines "dummy"–Zustandes die Zustandsüberführungsfunktion total machen können.

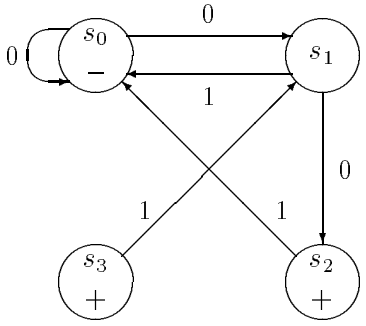
In der Regel ist es hilfreich sich einen endlichen Automaten durch seinen *Zustandsüberführungsgraphen* (state transition diagram STD) zu veranschaulichen. Sei $\mathcal{A} = (S, \delta, S_0, F)$ ein NEA. Das STD $G(\mathcal{A})$ von $\mathcal{A}$ ist ein gerichteter, kantengelabelter Graph mit Knotenmenge S. Falls ein Zustandsübergang von s_1 nach s_2 bei Lesen von σ definiert ist, so besitzt der Graph eine Kante von s_1 nach s_2 mit der Marke σ. Wir markieren Anfangszustände durch "–" und Endzustände durch "+".

Beispiel 2.17 Sei $\mathcal{A} = (S, \delta, S_0, F)$ mit $\Sigma = \{0, 1\}$, $S = (s_0, s_1, s_2, s_3)$, $S_0 = \{s_0\}$ und $F = \{s_2, s_3\}$. Die Zustandsübergangsfunktion δ sei durch die folgende Tabelle gegeben:

δ	0	1
s_0	$\{s_0, s_1\}$	$\emptyset$
s_1	$\{s_2\}$	$\{s_0\}$
s_2	$\emptyset$	$\{s_0\}$
s_3	$\emptyset$	$\{s_1\}$

$\mathcal{A}$ wird durch das folgende STD beschrieben:

$G(\mathcal{A})$: :



Definition 2.18 Eine Menge $A \subseteq \Sigma^*$ heißt *regulär*, falls ein NEA $\mathcal{A}$ mit $A = L(\mathcal{A})$ existiert. Die Menge aller regulären Sprachen über Σ bezeichnen wir mit REG_Σ, d.h.

$$\mathrm{REG}_\Sigma = \{\, A \mid \exists \text{NEA } \mathcal{A} \text{ über } \Sigma\, [\, A = L(\mathcal{A})\,]\, \} \subseteq \mathfrak{P}(\Sigma^*).$$

Überraschenderweise erweisen sich die Modelle des deterministischen endlichen Automaten und des nichtdeterministischen endlichen Automaten im Bezug auf Spracherkennung als äquivalent:

Satz 2.19 Sei $A \subseteq \Sigma^*$ eine reguläre Menge. Dann existiert ein EA $\mathcal{B}$ mit $L(\mathcal{B}) = A$.

BEWEIS: Sei $\mathcal{A} = (S, \delta, S_0, F)$ ein NEA mit$L(\mathcal{A}) = A$. Wir werden aus $\mathcal{A}$ den sogenannten *Potenzautomaten* $\mathcal{B} = (S', \delta', s_0', F')$ konstruieren. Dazu setzen wir

- $S' = \mathfrak{P}(S)$,
- $\delta'(A, \sigma) = \bigcup_{s \in A} \delta(s, \sigma)$ für $\sigma \in \Sigma$ und $A \in \mathfrak{P}(S)$,
- $s_0' = S_0$ und
- $F' = \{Z \in \mathfrak{P}(S) \mid Z \cap F \neq \emptyset\}$.

Man beachte, daß $\mathcal{B}$ deterministisch ist. Der aktuelle Zustand von $\mathcal{B}$ entspricht der Menge der möglichen aktuellen Zustände von $\mathcal{A}$. Wir haben nachzuweisen, daß $L(\mathcal{A}) = L(\mathcal{B})$ gilt.

Sei dazu zunächst $w \in L(\mathcal{A})$ mit $|w| = n$. Dann existiert eine akzeptierende $\mathcal{A}$–Berechnung $\rho : \underline{n+1} \to S$ über w. Sei $\rho' : \underline{n+1} \to \mathfrak{P}(S)$ die eindeutig bestimmte $\mathcal{B}$–Berechnung über w. Wir weisen durch Induktion über i nach, daß $\rho(i) \in \rho'(i)$ für jedes $i = 0, \ldots, n$ gilt.

Für $i = 0$ ist $\rho(i) \in S_0 = \rho'(0)$. Sei nun $\rho(i) \in \rho'(i)$ für ein $i < n$. Dann gilt

$$\rho(i+1) \in \delta(\rho(i), w(i)) \subseteq \bigcup_{s \in \rho'(i)} \delta(s, w(i)) = \delta'(\rho'(i), w(i)) = \rho'(i+1).$$

Da ρ eine akzeptierende $\mathcal{A}$–Berechnung ist, ist $\rho(n) \in F$, andererseits ist auch $\rho(n) \in \rho'(n)$. Damit ist $\rho'(n) \in F'$ und ρ' ist eine akzeptierende $\mathcal{B}$–Berechnung über w, d.h. $w \in L(\mathcal{B})$.

Sei nun umgekehrt $w \in L(\mathcal{B})$ mit $|w| = n$ und $\rho' : \underline{n+1} \to \mathfrak{P}(S)$ die akzeptierende $\mathcal{B}$–Berechnung über w. Nach Konstruktion von $\mathcal{B}$ ist $\rho'(i+1) = \bigcup_{s \in \rho'(i)} \delta(s, w(i))$ für $i = 0, \ldots, n-1$. Damit existiert für jedes $z \in \rho'(i+1)$ ein $s \in \rho'(i)$ mit $z \in \delta(s, w(i))$. Wir konstruieren eine akzeptierende $\mathcal{A}$–Berechnung über w wie folgt. Da $\rho'(n) \in F'$, existiert ein $q_n \in \rho'(n)$ mit $q_n \in F$. Aufgrund der obigen Bemerkung können wir induktiv eine Folge $(q_n, q_{n-1}, \ldots, q_0)$ konstruieren, so daß $q_i \in \rho'(i)$ für $i = 0, \ldots, n$ und $q_{i+1} \in \delta(q_i, w(i))$ für $i = 0, \ldots, n-1$ gilt. Weiterhin ist $q_0 \in S_0$ und $q_n \in F$, also ist $\rho : \underline{n+1} \to S$ mit $\rho(i) = q_i$ eine akzeptierende $\mathcal{A}$–Berechnung über w. □

Wir wollen nun unsere Betrachtungen auf ω–Sprachen ausdehnen. Dazu führen wir zunächst die "Unendlich-Oft" Funktion In ein: Sei $\mathbb{M}$ eine beliebige Menge und $f : \mathbb{N}_0 \to \mathbb{M}$. Wir sind an der Menge von Funktionswerten von f interessiert, die unendlich oft angenommen werden, d.h.

$$\mathrm{In}(f) = \{\, a \mid |f^{-1}(a)| = \infty \,\} \subseteq \mathbb{M},$$

wobei $f^{-1}(a) = \{\, i \mid i \in \mathbb{N}_0 \wedge f(i) = a \,\}$.

Definition 2.20 Sei $\mathcal{A} = (S, \delta, S_0, F)$ ein NEA über Σ und $v : \mathbb{N}_0 \to \Sigma$ ein ω–Wort über Σ.

Eine Funktion $\rho : \mathbb{N}_0 \to S$ heißt *$\mathcal{A}^\omega$–Berechnung* über v falls

i) $\rho(0) \in S_0$ und

ii) $\rho(i+1) \in \delta(\rho(i), v(i))$ für jedes $i \in \mathbb{N}_0$.

ρ heißt *akzeptierende* $\mathcal{A}^\omega$–Berechnung über v, falls zusätzlich

iii) $\mathrm{In}(\rho) \cap F \neq \emptyset$ gilt, d.h es existiert ein Endzustand, der in der $\mathcal{A}^\omega$–Berechnung ρ unendlich oft angenommen wird.

Der Automat $\mathcal{A}$ *ω–akzeptiert (ω–berechnet)* das Wort v genau dann, wenn eine akzeptierende $\mathcal{A}^\omega$–Berechnung über v existiert. Die von $\mathcal{A}$ ω–akzeptierte (ω–berechnete) Sprache ist gegeben durch

$$L^\omega(\mathcal{A}) = \{\, v \mid v : \mathbb{N}_0 \to \Sigma,\ \exists \rho\ : \rho \text{ ist akz. } \mathcal{A}^\omega\text{–Berechnung über } v \,\}.$$

Eine Menge $A \subseteq \Sigma^\omega$ heißt *ω–regulär*, falls ein NEA $\mathcal{A}$ mit $A = L^\omega(\mathcal{A})$ existiert. Die Menge aller ω–regulären Sprachen über Σ bezeichnen wir mit $\mathrm{REG}^\omega_\Sigma$, d.h.

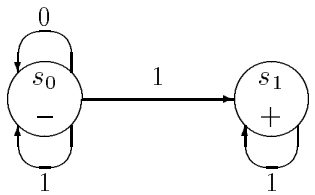
$$\mathrm{REG}^\omega_\Sigma = \{\, A \mid \exists \,\text{NEA } \mathcal{A} \text{ über } \Sigma\, [\, A = L^\omega(\mathcal{A}) \,] \,\} \subseteq \mathfrak{P}(\Sigma^\omega).$$

Leider läßt sich Satz 2.19 nicht auf ω–reguläre Mengen übertragen, vielmehr gilt das

Lemma 2.21 Es gibt ω–reguläre Mengen, die von keinem deterministischen EA ω–akzeptiert werden.

Beweis: Betrachte $A = \{0,1\}^*\{1\}^\omega \subseteq \{0,1\}^\omega$. Aus Satz 2.27 folgt, daß A ω–regulär ist, denn $A = A_0 B_0^\omega$ mit den regulären Mengen $A_0 = \{0,1\}^*$ und $B_0 = \{1\}$. A wird von dem NEA $\mathcal{A}$ ω–akzeptiert, der durch das folgende STD gegeben ist:

$G(\mathcal{A})$: :



Es ist $L^\omega(\mathcal{A}) = A$.

Nun haben wir noch nachzuweisen, daß kein EA existiert, der A ω–akzeptiert. Dazu nehmen wir an, daß der EA $\mathcal{B} = (S, \delta, s_0, F)$ A ω–akzeptiert. Wir können annehmen, daß für jedes $s \in F$ $L^\omega((S, \delta, s_0, \{s\})) \neq \emptyset$. Dann ist $L^\omega((S, \delta, s, F)) = \{1\}^\omega$. Sei $w \in A$. Da $\mathcal{B}$ deterministisch ist, existiert ein $n_0 \in \mathbb{N}$ mit $\rho(n_0) \in F$ für die $\mathcal{B}^\omega$–Berechnung über w. Betrachte nun $v \in \Sigma^\omega$ mit $v = w(0, n_0 - 1)01^\omega$. Es ist $v \in A$. Für die $\mathcal{B}^\omega$–Berechnung ρ' über v ist $\rho'(n_0) = \rho(n_0) \in F$, allerdings ist $v(n_0, \infty) = 01^\omega \notin \{1\}^\omega$ und v wird nicht von $\mathcal{B}$ ω–akzeptiert. □

2.3.3 Kleenesche Sätze

Wir werden nun zeigen, daß die Sprachen, die von endlichen Automaten erkannt werden, genau die Sprachen sind, die von regulären Ausdrücken beschrieben werden. Diese Äquivalenz rechtfertigt es, die von endlichen Automaten akzeptierten Sprachen regulär zu nennen.

Satz 2.22 Sei R ein regulärer Ausdruck. Dann existiert ein EA $\mathcal{A}$ mit $\varphi(R) = L(\mathcal{A})$.

BEWEIS: Aufgrund von Satz 2.19 genügt es zu zeigen, daß zu jedem $R \in \mathrm{RE}$ ein NEA $\mathcal{A}_R$ existiert mit $\varphi(R) = L(\mathcal{A}_R)$. Wir führen diesen Nachweis über den rekursiven Aufbau von R gemäß Definition 2.12, d.h. per Induktion über der Anzahl der regulären Operatoren in R.

i) (a) Sei $R = \emptyset$. Betrachte $\mathcal{A} = (S, \delta, S_0, \emptyset)$. Es ist $L(\mathcal{A}) = \emptyset = \varphi(R)$.

(b) Sei $R = \Lambda$. Konstruiere $\mathcal{A} = (\{s_0\}, \delta, \{s_0\}, \{s_0\})$ mit $\delta(s_0, \sigma) = \emptyset$ für jedes $\sigma \in \Sigma$. Dann ist $L(\mathcal{A}) = \{\Lambda\} = \varphi(R)$.

(c) Sei $R = \sigma$ mit $\sigma \in \Sigma$. Konstruiere $\mathcal{A} = (\{s_0, s_1\}, \delta, \{s_0\}, \{s_1\})$ und

$$\delta(s, a) = \begin{cases} s_1 & \text{falls } s = s_0 \text{ und } a = \sigma \\ \emptyset & \text{sonst.} \end{cases}$$

Damit ist $L(\mathcal{A}) = \{\sigma\} = \varphi(R)$.

ii) Sei $R = R_1 + R_2$. Nach Induktionsvoraussetzung existieren NEAs $\mathcal{A}_1 = (S_1, \delta_1, S_{0_1}, F_1)$ und $\mathcal{A}_2 = (S_2, \delta_2, S_{0_2}, F_2)$ mit $\varphi(R_1) = L(\mathcal{A}_1)$ und $\varphi(R_2) = L(\mathcal{A}_2)$. Wir können annehmen, daß $S_1 \cap S_2 = \emptyset$.

Aus $\mathcal{A}_1$ und $\mathcal{A}_2$ werden wir einen NEA $\mathcal{B} = (S, \delta, S_0, F)$ mit $L(\mathcal{B}) = L(\mathcal{A}_1) \cup L(\mathcal{A}_2) = \varphi(R)$ konstruieren. Dazu setzen wir $S = S_1 \cup S_2$, $S_0 = S_{0_1} \cup S_{0_2}$, $F = F_1 \cup F_2$ und definieren $\delta : (S_1 \cup S_2) \times \Sigma \to \mathfrak{P}(S_1 \cup S_2)$ durch

$$\delta(s, \sigma) = \begin{cases} \delta_1(s, \sigma) & \text{falls } s \in S_1 \\ \delta_2(s, \sigma) & \text{falls } s \in S_2. \end{cases}$$

iii) Sei $R = R_1 R_2$. Die NEAs $\mathcal{A}_1$ und $\mathcal{A}_2$ seien wie oben definiert. Sei $\mathcal{B} = (S_1 \cup S_2, \delta, S_{0_1}, F_2)$ mit

$$\delta(s, \sigma) = \begin{cases} \delta_1(s, \sigma) & \text{falls } s \in S_1 \setminus F_1 \\ \delta_1(s, \sigma) \cup \bigcup\limits_{s' \in S_{0_2}} \delta_2(s', \sigma) & \text{falls } s \in F_1 \\ \delta_2(s, \sigma) & \text{falls } s \in S_2. \end{cases}$$

Es ist leicht nachzuprüfen, daß $L(\mathcal{B}) = L(\mathcal{A}_1) L(\mathcal{A}_2)$ gilt.

iv) Sei $R = (R_1)^*$ und $\mathcal{A} = (S, \delta_{\mathcal{A}}, S_0, F)$ ein NEA mit $\varphi(R_1) = L(\mathcal{A})$. Sei $z \notin S$ und $\mathcal{B} = (S \cup \{z\}, \delta, S_0 \cup \{z\}, \{z\})$ mit

$$\delta(s, \sigma) = \begin{cases} \emptyset & \text{falls } s = z \\ \delta_{\mathcal{A}}(s, \sigma) \cup S_0 \cup \{z\} & \text{falls } \delta_{\mathcal{A}}(s, \sigma) \cap F \neq \emptyset \\ \delta_{\mathcal{A}}(s, \sigma) & \text{falls } \delta_{\mathcal{A}}(s, \sigma) \cap F = \emptyset. \end{cases}$$

Es ist $L(\mathcal{B}) = L(\mathcal{A})^*$. □

Es gilt auch die Umkehrung von Satz 2.22:

Satz 2.23 Sei $\mathcal{A}$ ein EA. Dann existiert ein regulärer Ausdruck R mit $\varphi(R) = L(\mathcal{A})$.

BEWEIS: Sei $\mathcal{A} = (S, \delta, s_0, F)$ mit $S = \{s_0, s_1, \ldots, s_{m-1}\}$.

Für $i, j = 0, \ldots, m-1$ definieren wir den EA $\mathcal{A}_{ij} = (S, \delta, s_i, \{s_j\})$ mit Startzustand s_i und Endzustand s_j. Damit ist

$$L(\mathcal{A}) = \bigcup_{s_j \in F} L(\mathcal{A}_{0j}).$$

Wir werden reguläre Ausdrücke R_j mit $\varphi(R_j) = L(\mathcal{A}_{0j})$ konstruieren. Somit gilt

$$L(\mathcal{A}) = \varphi(\mathop{+}_{s_j \in F} R_j)$$

Um uns die Konstruktion der reguläre Ausdrücke R_j weiter zu erleichtern, führen wir die folgenden Sprachen ein:

$$\overline{L}(\mathcal{A}_{ij}) = \{\, v \in L(\mathcal{A}_{ij}) \mid \text{für die } \mathcal{A}_{ij}\text{–Berechnung } \rho \text{ über } v \text{ gilt } \rho(i) \neq s_j \text{ für } 0 < i \leq |v| - 1 \,\}.$$

$\overline{L}(\mathcal{A}_{ij})$ entspricht der Menge aller Wörter, die von $\mathcal{A}_{ij}$ akzeptiert werden, wobei der Endzustand s_j nur einmal am Ende der Berechnung angenommen wird.

Mit dieser Vereinbarung gilt $L(\mathcal{A}_{0j}) = \overline{L}(\mathcal{A}_{0j})(\overline{L}(\mathcal{A}_{jj}))^*$. Damit haben wir unsere Aufgabe auf die Konstruktion von regulären Ausdrücken R_{ij} mit $\varphi(R_{ij}) = \overline{L}(\mathcal{A}_{ij})$ reduziert.

Für $s, z \in S$ führen wir zur leichteren Schreibweise die Menge

$$X_{s,z} = \{\, \sigma \in \Sigma \mid z = \delta(s, \sigma) \,\}$$

ein. Sei $X_{s,z} = \{\sigma_1, \sigma_2, \ldots, \sigma_s\}$, dann ist

$$S_{s,z} = \sigma_1 + \sigma_2 + \ldots + \sigma_s$$

ein regulärer Ausdruck mit $\varphi(S_{s,z}) = X_{s,z}$.

Nach diesen Vorbetrachtungen können wir die eigentliche Behauptung induktiv über die Anzahl m der Zustände von $\mathcal{A}$ nachweisen.

Zur Induktionsverankerung betrachten wir zunächst den Fall $m = 1$. Dann besteht S nur aus dem Zustand s_0 und wir haben lediglich für $\overline{L}(\mathcal{A}_{00})$ einen regulären Ausdruck zu konstruieren. Nun ist $\overline{L}(\mathcal{A}_{00})$ gerade X_{s_0,s_0} und S_{s_0,s_0} ist der gesuchte reguläre Ausdruck.

Als Induktionshypothese nehmen wir an, daß sich für jeden EA $\mathcal{B}$ mit weniger als m Zuständen ein regulärer Ausdruck $R_{\mathcal{B}}$ mit $\varphi(R_{\mathcal{B}}) = L(\mathcal{B})$ konstruieren läßt. Um den Induktionsschritt für $\mathcal{A}$ mit m Zuständen zu führen, müssen wir nach den obigen Bemerkungen für $\mathcal{A}_{ij}$ reguläre Ausdrücke R_{ij} mit $\varphi(R_{ij}) = \overline{L}(\mathcal{A}_{ij})$ konstruieren:

i) Wir betrachten zunächst den Fall $i \neq j$. Sei $Q \subseteq S \setminus \{s_j\}$ die Menge von Zuständen, von denen der Endzustand s_j in einem Schritt erreichbar ist, d.h

$$Q = \{\, s \in S \setminus \{s_j\} \mid \exists \sigma \in \Sigma \,:\, s_j = \delta(s, \sigma) \,\}.$$

Für $s \in Q$ bezeichne $\mathcal{A}_{ij}^s$ die Restriktion von $\mathcal{A}_{ij}$ auf $S \setminus \{s_j\}$ mit neuem Endzustand s, d.h.

$$\mathcal{A}_{ij}^s = (S \setminus \{s_j\}, \delta_j, s_i, \{s\})$$

mit $\delta_j(z,\sigma) = \delta(z,\sigma)$ für $z \in S \setminus \{s_j\}$. Da $\mathcal{A}_{ij}^s$ genau $m-1$ Zustände besitzt, existiert nach Induktionsannahme ein regulärer Ausdruck R_{ij}^s mit $\varphi(R_{ij}^s) = L(\mathcal{A}_{ij}^s)$. Sei

$$Q' = \{\, s \in Q \mid L(\mathcal{A}_{ij}^s) \neq \emptyset \vee s = s_i \,\}.$$

Dann ist

$$\overline{L}(\mathcal{A}_{ij}) = \bigcup_{s \in Q'} L(\mathcal{A}_{ij}^s) X_{s,s_j}$$

und $R_{ij} = \underset{s \in Q'}{+} R_{ij}^s S_{s,s_j}$ ist ein regulärer Ausdruck mit $\varphi(R_{ij}) = \overline{L}(\mathcal{A}_{ij})$.

ii) Sei nun $i = j$. Dann definieren wir die Mengen $Q_1, Q_2 \subseteq S \setminus \{s_i\}$ als die Menge von Zuständen, die vom Anfangszustand s_i im einem Schritt erreichbar sind, bzw. von denen aus der Endzustand s_i in einem Schritt erreichbar ist, d.h

$$Q_1 = \bigcup_{\sigma \in \Sigma} \delta(s_i,\sigma) \setminus \{s_i\} \text{ und}$$

$$Q_2 = \{\, s \in S \setminus \{s_i\} \mid \exists \sigma \in \Sigma \; : s_i = \delta(s,\sigma) \,\}.$$

Für $z_1 \in Q_1$ und $z_2 \in Q_2$ bezeichne $\mathcal{A}_{ij}^{z_1,z_2}$ die Restriktion von $\mathcal{A}_{ii}$ auf $S \setminus \{s_i\}$ mit Anfangszustand z_1 und Endzustand z_2, d.h.

$$\mathcal{A}_{ii}^{z_1,z_2} = (S \setminus \{s_i\}, \delta_i, z_1, \{z_2\})$$

mit $\delta_i(z,\sigma) = \delta(z,\sigma)$ für $z \in S \setminus \{s_i\}$. Da $\mathcal{A}_{ii}^{z_1,z_2}$ genau $m-1$ Zustände besitzt, existiert nach Induktionsannahme ein regulärer Ausdruck $R_{ii}^{z_1,z_2}$ mit $\varphi(R_{ii}^{z_1,z_2}) = L(\mathcal{A}_{ii}^{z_1,z_2})$. Sei

$$Q' = \{\, (z_1,z_2) \in Q_1 \times Q_2 \mid L(\mathcal{A}_{ij}^{z_1,z_2}) \neq \emptyset \,\}.$$

Dann ist

$$\overline{L}(\mathcal{A}_{ii}) = X_{s_i,s_i} \cup \bigcup_{z \in Q_1 \cap Q_2} X_{s_i,z} X_{z,s_i} \cup \bigcup_{(z_1,z_2) \in Q'} X_{s_i,z_1} L(\mathcal{A}_{ii}^{z_1,z_2}) X_{z_2,s_i}$$

und

$$R_{ii} = S_{s_i,s_i} + \underset{z \in Q_1 \cap Q_2}{+} S_{s_i,z} S_{z,s_i} + \underset{s \in Q'}{+} S_{s_i,z_1} R_{ii}^{z_1,z_2} S_{z_2,s_i}$$

ist ein regulärer Ausdruck mit $\varphi(R_{ii}) = \overline{L}(\mathcal{A}_{ii})$.

□

Die Sätze 2.22 und 2.23 implizieren das

Korollar 2.24 Die Menge der regulären Sprachen ist genau die Menge der durch reguläre Ausdrücke definierten Sprachen, d.h. $\mathrm{REG}_\Sigma = \varphi(\mathrm{RE}_\Sigma)$.

2.3.4 Struktur von regulären Mengen

In diesem Abschnitt weisen wir nach, daß die Menge der regulären Sprachen über Σ eine einfache mengentheoretische Struktur besitzt – sie bildet eine Boolesche Algebra.

Satz 2.25 Sei Σ ein Alphabet. Dann ist $(\mathrm{REG}_\Sigma, \cup, \cap, \bar{})$ mit $\bar{A} = \Sigma^* \setminus A$ für $A \in \mathrm{REG}_\Sigma$ eine Boolesche Algebra.

BEWEIS: Im Abschnitt 1.4 haben wir gesehen, daß $(\mathfrak{P}(\Sigma^*), \cup, \cap, \bar{})$ eine Boolesche Algebra mit Einselement Σ^* und Nullelement $\emptyset$ bildet.

Da $\mathrm{REG}_\Sigma \subseteq \mathfrak{P}(\Sigma^*)$ ist, haben wir lediglich nachzuweisen, daß REG_Σ abgeschlossen unter Vereinigung, Komplement und Durchschnitt ist:

i) Die Abgeschlossenheit unter Vereinigung wurde schon in Satz 2.22 gezeigt.

ii) Sei $A \in \mathrm{REG}_\Sigma$. Dann existiert nach Satz 2.19 ein EA $\mathcal{A} = (S, \delta, S_0, F)$ mit $L(\mathcal{A}) = A$. Sei $\mathcal{B} = (S, \delta, S_0, S \setminus F)$. Es ist leicht nachzuweisen, daß $L(\mathcal{B}) = \bar{A}$ gilt: Nach Konstruktion von $\mathcal{B}$ ist für $w \in \Sigma^*$ die $\mathcal{A}$–Berechnung ρ_w über w eine $\mathcal{B}$–Berechnung über w und umgekehrt. Sei nun $w \in L(\mathcal{A})$. Dann ist $\rho(|w|) \in F$, also $\rho(|w|) \notin S \setminus F$. Daher ist $w \notin L(\mathcal{B})$. Für $w \in L(\mathcal{B})$ ist $\rho(|w|) \in S \setminus F$, also $\rho(|w|) \notin F$, damit ist $w \notin L(\mathcal{A})$.

iii) Seien $A, B \in \mathrm{REG}_\Sigma$. Dann ist auch $A \cap B \in \mathrm{REG}_\Sigma$, denn $A \cap B = \overline{\bar{A} \cup \bar{B}}$ und die Abgeschlossenheit von REG_Σ unter Komplement und Vereinigung haben wir soeben nachgewiesen. □

Aufgrund der Kleeneschen Sätze können wir diese Struktur auch auf durch reguläre Ausdrücke definierte Sprachen übertragen.

Korollar 2.26 Seien $R_1, R_2 \in \mathrm{RE}$. Dann existieren $R_3, R_4 \in \mathrm{RE}$ mit

i) $\varphi(R_3) = \varphi(R_1) \cap \varphi(R_2)$.

ii) $\varphi(R_4) = \Sigma^* \setminus \varphi(R_1)$.

2.3.5 Struktur ω–regulärer Mengen

Die Menge der ω–regulären Sprachen bildet ebenfalls eine Boolesche Algebra. Der Nachweis erfordert jedoch wesentlich größere Anstrengungen als im Fall der regulären Sprachen.

Wir beginnen zunächst mit einer einfachen Charakterisierung ω–regulärer Mengen.

Satz 2.27 Sei $A \subseteq \Sigma^\omega$. A ist ω–regulär genau dann, wenn ein $n \in \mathbb{N}_0$ und reguläre Mengen $A_i, B_i \subseteq \Sigma^*$ für $i \in \underline{n}$ existieren, so daß

$$A = \bigcup_{0 \leq i < n} A_i B_i^\omega.$$

BEWEIS:

($\Rightarrow$) Sei A ω–regulär, dann existiert ein NEA $\mathcal{A} = (S, \delta, S_0, F)$ mit $A = L^\omega(\mathcal{A})$. Für $s_i, s_j \in S$ definieren wir die reguläre Menge

$$X(s_i, s_j) = L((S, \delta, \{s_i\}, \{s_j\})).$$

Sei $w \in \Sigma^\omega$. Dann ist $w \in A$ genau dann, wenn es eine $\mathcal{A}^\omega$–Berechnung $\rho : \mathbb{N}_0 \to S$ mit $\mathrm{In}(\rho) \cap F \neq \emptyset$ gibt, also wenn eine Folge $(n_i)_{i \in \mathbb{N}}$ und Zustände $s_0 \in S_0$, $s \in F$ existieren, so daß $\rho(0, n_1)$ eine $(S, \delta, \{s_0\}, \{s\})$–Berechnung für $w(0, n_1 - 1)$, und $\rho(n_i, n_{i+1})$ eine $(S, \delta, \{s\}, \{s\})$–Berechnung für $w(n_i, n_{i+1} - 1)$ für jedes $i \in \mathbb{N}$ ist, d.h. $w(0, n_1) \in X(s_0, s)$ und $w(n_i, n_{i+1}) \in X(s, s)$. Damit gilt

$$A = \bigcup_{\substack{s \in F \\ s_0 \in S_0}} X(s_0, s) X(s, s)^\omega.$$

($\Leftarrow$) Die Rückrichtung ergibt sich aus den beiden folgenden Lemmata 2.28 und 2.29. □

Lemma 2.28 Seien $A, B \subseteq \Sigma^*$ reguläre Mengen. Es existiert ein NEA $\mathcal{C}$ mit $L^\omega(\mathcal{C}) = AB^\omega$.

BEWEIS: Nach den Sätzen 2.19 und 2.22 existieren EAs $\mathcal{A} = (S_1, \delta_1, s_{0_1}, F_1)$ und $\mathcal{B} = (S_2, \delta_2, s_{0_2}, F_2)$, $S_1 \cap S_2 = \emptyset$ mit $L(\mathcal{A}) = A$ und $L(\mathcal{B}) = B^*$. Sei $\mathcal{C} = (S_1 \cup S_2, \delta, \{s_{0_1}\}, F_2)$ ein NEA mit

$$\delta(s, \sigma) = \begin{cases} \delta_1(s, \sigma) & \text{falls } \delta_1(s, \sigma) \in S_1 \setminus F_1 \\ \delta_1(s, \sigma) \cup \delta_2(s_{0_2}, \sigma) & \text{falls } \delta_1(s, \sigma) \in F_1 \\ \delta_2(s, \sigma) & \text{falls } s \in S_2. \end{cases}$$

Dann ist $L(\mathcal{C}) = AB^*$ und $L^\omega(\mathcal{C}) = AB^\omega$. □

Lemma 2.29 Seien $\mathcal{A}$ und $\mathcal{B}$ NEAs. Dann existiert ein NEA $\mathcal{C}$ mit $L^\omega(\mathcal{C}) = L^\omega(\mathcal{A}) \cup L^\omega(\mathcal{B})$.

BEWEIS: Die Konstruktion von $\mathcal{C}$ erfolgt analog zum Beweis des Satzes 2.22. □

Im folgenden möchten wir (vergl. Satz 2.25) zeigen, daß $(\mathrm{REG}^\omega_\Sigma, \cup, \cap, {}^-)$ ebenfalls eine Boolesche Algebra ist.

Nach Lemma 2.21 können wir die ω–regulären Sprachen nicht durch deterministische endliche Automaten charakterisieren. Daher können wir die Vorgehensweise des regulären Falles nicht auf den ω–regulären Fall übertragen.

Um ω–reguläre Mengen dennoch von *deterministischen* Automaten ω–berechnen zu können, werden wir ein zusätzliches Modell eines endlichen Automaten einführen, den sogenannten *Macro-Zustand Endlichen Automaten.*

Definition 2.30 Ein Macro-Zustand (Muller) Endlicher Automat (MEA) über Σ ist ein Quadrupel $\mathcal{A} = (S, \delta, s_0, \mathbb{F})$, wobei S die (endliche) Menge von Zuständen, $\delta : S \times \Sigma \to S$ die Zustandsübergangsfunktion und $s_0 \in S$ den Anfangszustand bezeichnet. $\mathbb{F} \subseteq \mathfrak{P}(S)$ ist eine Menge von akzeptierenden Macro-Zuständen.

Sei $\mathcal{A}$ ein MEA und $v \in \Sigma^\omega$. Die $\mathcal{A}^\omega$–Berechnung $\rho_v : \mathbb{N}_0 \to S$ über v (mit $\rho_v(0) = s_0$ und $\rho_v(i+1) = \delta(\rho_v(i), v(i))$ für jedes $i \in \mathbb{N}_0$) heißt *akzeptierend*, falls $\mathrm{In}(\rho_v) \in \mathbb{F}$ gilt, d.h. die

Menge der Zustände, die in der $\mathcal{A}^\omega$–Berechnung ρ unendlich oft angenommen werden, muß in $\mathbb{F}$ enthalten sein.

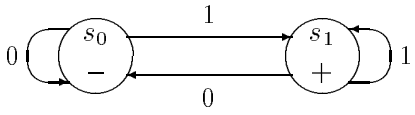
Falls ρ_v akzeptierend ist, so sagen wir auch, daß der MEA $\mathcal{A}$ das Wort v ω–akzeptiert (ω–berechnet). Die Menge der von $\mathcal{A}$ ω–akzeptierten Wörter bezeichnen wir als die von $\mathcal{A}$ ω–akzeptierte (ω–berechnete) Sprache und notieren sie durch $L^\omega(\mathcal{A})$.

Eine Menge $A \subseteq \Sigma^\omega$ heißt *ω–M–regulär*, falls ein MEA $\mathcal{A}$ mit $A = L^\omega(\mathcal{A})$ existiert. Die Menge aller ω–M–regulären Sprachen über Σ bezeichnen wir mit $\mathrm{REG}_\Sigma^{\omega M}$, d.h.

$$\mathrm{REG}_\Sigma^{\omega M} = \{\, A \subseteq \Sigma^\omega \mid \exists\, \text{MEA } \mathcal{A} \text{ über } \Sigma \,:\, A = L^\omega(\mathcal{A}) \,\}.$$

Als Beispiel geben wir für die ω–reguläre Menge $A = \{0,1\}^*\{1\}^\omega \subseteq \{0,1\}^\omega$, die nach Lemma 2.21 von keinem EA ω–berechnet wird, einen MEA $\mathcal{A} = (\{s_0, s_1\}, \delta, s_0, \{\{s_1\}\})$ mit $L^\omega(\mathcal{A}) = A$ an:

$G(\mathcal{A})$: :



Unser Ziel ist der Nachweis, daß $(\mathrm{REG}_\Sigma^\omega, \cup, \cap, {}^-)$ eine Boolesche Algebra ist. Dazu werden wir zunächst zeigen, daß $(\mathrm{REG}_\Sigma^{\omega M}, \cup, \cap, {}^-)$ eine Boolesche Algebra ist, und anschließend nachweisen, daß $\mathrm{REG}_\Sigma^\omega = \mathrm{REG}_\Sigma^{\omega M}$ gilt.

Um die Abgeschlossenheit von $\mathrm{REG}_\Sigma^{\omega M}$ unter Vereinigung nachzuweisen, führen wir noch ein weiteres Modell eines deterministischen endlichen Automaten ein, den *Tafel Endlichen Automaten.*

Definition 2.31 Ein Tafel (Table) Endlicher Automat (TEA) über Σ ist ein Quadrupel $\mathcal{A} = (S, \delta, s_0, \mathbb{T})$, wobei S die (endliche) Menge von Zuständen, $\delta : S \times \Sigma \to S$ die Zustandsübergangsfunktion und $s_0 \in S$ den Anfangszustand bezeichnet. $\mathbb{T} \subseteq (\mathfrak{P}(S) \times \mathfrak{P}(S))^m$ für ein $m \in \mathbb{N}_0$ heißt die Tafel von $\mathcal{A}$ der Größe m. Im folgenden werden wir die Notation $\mathbb{T} = \{(L_i, U_i) | i \in \underline{m}\}$ mit $L_i, U_i \subseteq S$ benutzen.

Sei $\mathcal{A}$ ein TEA und $v \in \Sigma^\omega$. Die $\mathcal{A}^\omega$–Berechnung $\rho_v : \mathbb{N}_0 \to S$ über v heißt *akzeptierend*, falls

$$\exists\, i \in \underline{m} \,:\, \mathrm{In}(\rho_v) \cap L_i = \emptyset \text{ und } \mathrm{In}(\rho_v) \cap U_i \neq \emptyset.$$

Falls ρ_v akzeptierend ist, so sagen wir auch, daß der TEA $\mathcal{A}$ das Wort v ω–akzeptiert (ω–berechnet). Die Menge der von $\mathcal{A}$ ω–akzeptierten Wörter bezeichnen wir als die von $\mathcal{A}$ ω–akzeptierte (ω–berechnete) Sprache und notieren sie durch $L^\omega(\mathcal{A})$.

Eine Menge $A \subseteq \Sigma^\omega$ heißt *ω–T–regulär*, falls ein TEA $\mathcal{A}$ mit $A = L^\omega(\mathcal{A})$ existiert. Die Menge aller ω–T–regulären Sprachen über Σ bezeichnen wir mit $\mathrm{REG}_\Sigma^{\omega T}$.

Wir werden nun die Äquivalenz von MEAs und TEAs nachweisen.

Lemma 2.32 Sei $A \in \Sigma^\omega$. A ist ω–M–regulär genau dann, wenn A ω–T–regulär ist, d.h.

$$\mathrm{REG}_\Sigma^{\omega M} = \mathrm{REG}_\Sigma^{\omega T}.$$

BEWEIS:

($\Rightarrow$) Sei $\mathcal{A} = (S, \delta, s_0, \mathbb{F})$ mit $\mathbb{F} = \{Q_0, \ldots, Q_{k-1}\}$, $\emptyset \notin \mathbb{F}$ ein MEA mit $A = L^\omega(\mathcal{A})$. Wir konstruieren einen TEA $\mathcal{B} = (S', \delta', s_0', \mathbb{T})$ mit $L^\omega(\mathcal{B}) = L^\omega(\mathcal{A})$. Sei dazu

$$S' = \mathfrak{P}(Q_0) \times \cdots \times \mathfrak{P}(Q_{k-1}) \times S.$$

Die ersten k Komponenten von S' dienen dazu, die Makrozustände von $\mathcal{A}$ zu simulieren. Für $s' = (A_0, \ldots, A_{k-1}, s) \in S'$ bezeichne $\pi_i(s')$ die Projektion von s' auf die i-te Komponente, d.h. $\pi_i(s') = A_i$ für $0 \leq i < k$ und $\pi_k(s') = s$.

Wir definieren $\delta : S' \times \Sigma \to S'$ durch $\delta(s', \sigma) = s''$ mit

$$\begin{aligned} \pi_i(s'') &= \begin{cases} \emptyset & \text{falls } \pi_i(s') = Q_i \\ Q_i \cap (\pi_i(s') \cup \{\pi_k(s')\}) & \text{falls } \pi_i(s') \subsetneq Q_i \end{cases} \quad \text{für } 0 \leq i < k \text{ und} \\ \pi_k(s'') &= \delta(\pi_k(s'), \sigma). \end{aligned}$$

Sei $s_0' = (\emptyset, \ldots, \emptyset, s_0)$.

Sei $w \in \Sigma^\omega$ und ρ_1 die $\mathcal{A}^\omega$–Berechnung über w. Für die $\mathcal{B}^\omega$–Berechnung ρ_2 über w gilt dann $\pi_k(\rho_2(i)) = \rho_1(i)$ für $i \in \mathbb{N}_0$. Nach Konstruktion von δ' ist weiterhin $\pi_j(\mathrm{In}(\rho_2)) = Q_j$ gleichbedeutend mit $Q_j \subseteq \mathrm{In}(\rho_1)$. Sei

$$U_j = \{\, s' \in S' \mid \pi_j(s') = Q_j \,\} \quad \text{und} \quad L_j = \{\, s' \in S' \mid \pi_k(s') \in S' \setminus Q_j \,\}.$$

Dann ist $\mathrm{In}(\rho_1) = Q_j$ äquivalent mit $(\mathrm{In}(\rho_2) \cap U_j \neq \emptyset) \wedge (\mathrm{In}(\rho_2) \cap L_j = \emptyset)$.

Mit $\mathbb{T} = \{(L_i, U_i) | 0 \leq i < k\}$ ist dann $w \in L^\omega(\mathcal{A}) \Leftrightarrow w \in L^\omega(\mathcal{B})$.

($\Leftarrow$) Sei $\mathcal{B} = (S, \delta, s_0, \mathbb{T})$ ein TEA mit $A = L^\omega(\mathcal{B})$. Sei $\mathbb{T} = \{(L_i, U_i) | 0 \leq i < m\}$. Wir konstruieren einen MEA $\mathcal{A} = (S, \delta, s_0, \mathbb{F})$ mit

$$\mathbb{F} = \{\, Q \subseteq S \mid \exists\, 0 \leq i < m \;:\; Q \cap U_i \neq \emptyset \wedge Q \cap L_i = \emptyset \,\}.$$

Offensichtlich ist damit $L^\omega(\mathcal{A}) = L^\omega(\mathcal{B})$. □

Wir werden nun die Abgeschlossenheit von $\mathrm{REG}_\Sigma^{\omega T}$ – und damit auch von $\mathrm{REG}_\Sigma^{\omega M}$ – unter Vereinigung nachweisen.

Lemma 2.33 Es seien $\mathcal{A}, \mathcal{B}$ TEAs. Dann existiert ein TEA $\mathcal{C}$ mit $L^\omega(\mathcal{C}) = L^\omega(\mathcal{A}) \cup L^\omega(\mathcal{B})$.

BEWEIS: Seien $\mathcal{A} = (S_1, \delta_1, s_{0_1}, \mathbb{T}_1)$ und $\mathcal{B} = (S_2, \delta_2, s_{0_2}, \mathbb{T}_2)$ TEAs mit den Tafeln $\mathbb{T}_1 = \{\, (L_{1_i}, U_{1_i}) \mid 0 \leq i < k_1 \,\}$ der Größe k_1 und $\mathbb{T}_2 = \{\, (L_{2_i}, U_{2_i}) \mid 0 \leq i < k_2 \,\}$ der Größe k_2.

Sei $\mathcal{C} = (S_1 \times S_2, \delta', (s_{0_1}, s_{0_2}), \mathbb{T}')$ mit $\delta'((s_1, s_2), \sigma) = (\delta_1(s_1, \sigma), \delta_2(s_2, \sigma))$ für $s_1 \in S_1$, $s_2 \in S_2$ und $\sigma \in \Sigma$. Damit simuliert $\mathcal{C}$ die Berechnungen von $\mathcal{A}$ und $\mathcal{B}$ nebeneinander. Wir spezifizieren die Tafel $\mathbb{T}'$ derart, daß $L^\omega(\mathcal{C}) = L^\omega(\mathcal{A}) \cup L^\omega(\mathcal{B})$ gilt. Sei

$$\mathbb{T}' = \{\, (L_{1_i} \times S_2, U_{1_i} \times S_2) \mid 0 \leq i < k_1 \,\} \cup \{\, (S_1 \times L_{2_j}, S_1 \times U_{2_j}) \mid 0 \leq j < k_2 \,\}.$$

Für $w \in \Sigma^\omega$ sei ρ_1 die $\mathcal{A}^\omega$–Berechnung und ρ_2 die $\mathcal{B}^\omega$–Berechnung über w. Für die $\mathcal{C}^\omega$–Berechnung ρ' über w gilt dann $\rho'(i) = (\rho_1(i), \rho_2(i))$ für $i \in \mathbb{N}_0$.

Nun ist $w \in L^\omega(\mathcal{A}) \cup L^\omega(\mathcal{B})$ genau dann, wenn i, j mit $0 \leq i < k_1$, $0 \leq j < k_2$ existieren, so daß

$$(\,\mathrm{In}(\rho_1) \cap L_{1_i} = \emptyset \;\wedge\; \mathrm{In}(\rho_1) \cap U_{1_i} \neq \emptyset\,) \;\vee\; (\,\mathrm{In}(\rho_2) \cap L_{2_j} = \emptyset \;\wedge\; \mathrm{In}(\rho_2) \cap U_{j_i} \neq \emptyset\,).$$

Nach der Definition von $\mathbb{T}'$ ist dies genau dann erfüllt, wenn ρ' akzeptierend ist, also wenn $w \in L^\omega(\mathcal{C})$. □

Im folgenden Lemma weisen wir die Abgeschlossenheit von $\mathrm{REG}_\Sigma^{\omega M}$ unter Komplementbildung nach:

Lemma 2.34 Sei $\mathcal{A}$ ein MEA über Σ. Dann existiert ein MEA $\mathcal{B}$ mit $L^\omega(\mathcal{B}) = \Sigma^\omega \setminus L^\omega(\mathcal{A})$.

Beweis: Sei $\mathcal{A} = (S, \delta, s_0, \mathbb{F})$ ein MEA. Wir konstruieren den komplementären MEA $\mathcal{B} = (S, \delta, s_0, \mathfrak{P}(S) \setminus \mathbb{F})$. Offensichtlich gilt $L^\omega(\mathcal{B}) = \Sigma^\omega \setminus L^\omega(\mathcal{A})$. □

Aus der Abgeschlossenheit von $\mathrm{REG}_\Sigma^{\omega M}$ unter Vereinigung und Komplementbildung folgt die Abgeschlossenheit unter Durchschnitt (vergl. Satz 2.25). Damit gilt der

Satz 2.35 Sei Σ ein Alphabet. Dann ist $(\mathrm{REG}_\Sigma^{\omega M}, \cup, \cap, {}^-)$ eine Boolesche Algebra.

Wir möchten schließlich noch nachweisen, daß die ω–M–regulären Mengen gerade die ω–regulären Mengen sind. Wir teilen den Beweis wieder in eine Reihe von Lemmata auf. Zunächst benötigen wir die folgenden Definitionen.

Sei $\mathcal{A} = (S, \delta, s_0, F)$ ein EA mit Zustandsübergangsfunktion $\delta : S \times \Sigma \to S$. Wir erweitern δ auf Wörter aus Σ^*, d.h. wir betrachten δ als eine Abbildung $S \times \Sigma^* \to S$. Für jedes $s \in S$ und jedes $v \in \Sigma^*$ setzen wir $\delta(s, v) = s'$ mit $s' = \rho(|v|)$ für eine (S, δ, s, F)–Berechnung ρ über v.

Definition 2.36 Sei $\mathcal{A} = (S, \delta, s_0, F)$ ein EA und $A = L(\mathcal{A})$. Sei $v \in \Sigma^\omega$ und $i \in \mathbb{N}_0$. i heißt *Flag–Punkt* von $\mathcal{A}$ in v, wenn ein $j \in \mathbb{N}_0$ mit $0 < j < i$ existiert, so daß die folgenden Bedingungen erfüllt sind:

i) $v(0, j) \in L(\mathcal{A})$

ii) $\delta(s_0, v(j, i)) = \delta(s_0, v(0, i))$

iii) Für jedes i' mit $j < i' < i$ gilt $\delta(s_0, v(j, i')) \neq \delta(s_0, v(0, i'))$

Wir nennen ein solches j einen *assoziierten Punkt* von i und bezeichnen ihn mit $a(i) = j$.

Weiterhin definieren wir die Sprache $A^\Delta \subseteq \mathfrak{P}(\Sigma^\omega)$ als Menge aller ω–Wörter, in denen $\mathcal{A}$ unendlich viele Flag–Punkte hat, d.h.

$$A^\Delta = \{\, v \mid v : \mathbb{N}_0 \to \Sigma,\ |\,\{\, i \mid i \text{ ist Flag–Punkt von } \mathcal{A} \text{ in } v \,\}\,| = \infty \,\}.$$

Die dritte Eigenschaft assoziierter Punkte garantiert uns, daß paarweise verschiedene Flag–Punkte eines ω–Wortes $v \in \Sigma^\omega$ verschiedene assoziierte Punkte besitzen, d.h. für Flag–Punkte i_1, i_2 von v mit $i_1 \neq i_2$ gilt $a(i_1) \neq a(i_2)$. Jeder assoziierte Punkt ist also eindeutig einem Flag–Punkt zugeordnet.

Lemma 2.37 Sei $\mathcal{A}$ ein EA mit $A = L(\mathcal{A})$. Dann ist $A^\Delta \subseteq A^\omega$.

BEWEIS: Sei $\mathcal{A} = (S, \delta, s_0, F)$ und $A = L(\mathcal{A})$. Sei $v \in A^\Delta$. Dann existiert eine (unendliche) Folge $I = (i_1, i_2, \ldots)$ von Flag–Punkten von $\mathcal{A}$ in v mit $i_k < i_{k+1}$ für jedes $k \in \mathbb{N}$. Da $|I| = \infty$ ist, existiert eine (unendliche) Teilfolge $J = (j_1, j_2, \ldots)$ von I mit $a(j_k) < j_k < a(j_{k+1}) < j_{k+1}$ für jedes $k \in \mathbb{N}$.

Um $v \in A^\omega$ nachzuweisen, genügt es zu zeigen, daß $v(a(j_n), a(j_{n+1})) \in A$ für alle $n \in \mathbb{N}$ gilt. Dies folgt jedoch direkt aus der Definition 2.36:

i) $v(0, a(j_{n+1})) \in A$, daher ist $\delta(s_0, v(0, a(j_{n+1}))) \in F$.

ii) $\delta(s_0, v(a(j_n), j_n)) = \delta(s_0, v(0, j_n))$.
Da $j_n < a(j_{n+1})$ ist, gilt dann auch $\delta(s_0, v(a(j_n), a(j_{n+1}))) = \delta(s_0, v(0, a(j_{n+1}))) \in F$.

Also ist $v(a(j_n), a(j_{n+1})) \in A$, damit ist $v \in A^\omega$ und $A^\Delta \subseteq A^\omega$. □

Lemma 2.38 Sei $\mathcal{A}$ ein EA mit $A = L(\mathcal{A})$. Es gelte $A^* = A$. Dann ist $AA^\Delta = A^\omega$.

BEWEIS: Sei $\mathcal{A} = (S, \delta, S_0, F)$.

($\subseteq$) Nach Lemma 2.37 ist $A^\Delta \subseteq A^\omega$, also ist $AA^\Delta \subseteq AA^\omega = A^\omega$.

($\supseteq$) Sei $v \in \Sigma^\omega$. Dann existiert eine Folge $J = (j_0, j_1, \ldots)$ mit $0 = j_0 < j_1 < \ldots < j_n < \ldots$, so daß für alle $n \in \mathbb{N}_0$ $v(j_n, j_{n+1}) \in A$ gilt. Da $A = A^*$ gilt, ist $v(0, j_n) \in A$ für jedes $n \in \mathbb{N}_0$.

Wir definieren eine Relation $\sim \subseteq \mathbb{N}_0^2$ durch

$$j \sim j' \iff \exists\, i \in \mathbb{N} : j, j' < i \wedge \delta\Big(s_0, v(j, i)\Big) = \delta\Big(s_0, v(j', i)\Big) \tag{2.1}$$

für alle $j, j' \in \mathbb{N}_0$.

Man überlegt sich leicht, daß $\sim$ eine Äquivalenzrelation ist:

i) $j \sim j$ für jedes $j \in \mathbb{N}_0$ ($\sim$ ist reflexiv)

ii) $j \sim j' \Longrightarrow j' \sim j$ für alle $j, j' \in \mathbb{N}_0$ ($\sim$ ist symmetrisch)

iii) $j \sim j' \wedge j' \sim j'' \Longrightarrow j \sim j''$ für alle $j, j', j'' \in \mathbb{N}_0$ ($\sim$ ist transitiv)

Die Äquivalenzklassen von $\sim$ wollen wir im folgenden mit $\sim[i] = \{j \mid j \in \mathbb{N}_0 \wedge i \sim j\}$ für $i \in \mathbb{N}_0$ bezeichnen. Die Quotientenmenge von $\mathbb{N}_0$ nach $\sim$ ist dann durch $\mathbb{N}_0/\sim = \{\sim[i] \mid i \in \mathbb{N}_0\}$ gegeben.

Für die Anzahl der Äquivalenzklassen von $\sim$ gilt $|\mathbb{N}_0/\sim| \leq |S|$. Um dies nachzuweisen nehmen wir $|\mathbb{N}_0/\sim| > |S|$ an. Dann existieren $k_1, k_2, \ldots, k_{|S|+1} \in \mathbb{N}_0$, die paarweise nicht äquivalent sind. Sei $j > \max\{k_1, k_2, \ldots, k_{|S|+1}\}$ und $s_i = \delta(s_0, v(k_i, j))$ für $i = 1, \ldots, |S|+1$. Da $k_1, k_2, \ldots, k_{|S|+1}$ paarweise nicht äquivalent sind, müssen nach (2.1) $s_1, s_2, \ldots, s_{|S|+1}$ paarweise verschieden sein. Dann wäre aber $|S| > |S|$ und damit ist die Annahme zum Widerspruch geführt.

Da es nur endlich viele Äquivalenzklassen gibt, muß es in der (unendlichen) Folge J unendlich viele paarweise äquivalente Folgenglieder geben. Sei also $\bar{J} = (\bar{j}_1, \bar{j}_2, \ldots)$ mit $\bar{j}_1 < \bar{j}_2 < \ldots < \bar{j}_n < \ldots$ und $\bar{j}_1 \sim \bar{j}_n$ für alle $n \in \mathbb{N}$.

Da $v(0, j_n) \in A$ für jedes $n \in \mathbb{N}_0$ ist, gilt auch $v(0, \bar{j}_1) \in A$. Mit $v = v(0, \bar{j}_1)v(\bar{j}_1, \infty)$ müssen wir noch nachweisen, daß $v(\bar{j}_1, \infty) \in A^\Delta$ ist.

Da $\bar{J}$ Teilfolge von J ist und $A = A^*$ gilt, ist $v(\bar{j}_1, \bar{j}_n) \in A$ für jedes $n \in \mathbb{N}$.

Weiterhin gibt es wegen $\bar{j}_1 \sim \bar{j}_n$ eine kleinste Zahl $i_n \in \mathbb{N}$ mit

$$\bar{j}_1 < \bar{j}_n < i_n \wedge \delta\Big(s_0, v(\bar{j}_1, i_n)\Big) = \delta\Big(s_0, v(\bar{j}_n, i_n)\Big). \tag{2.2}$$

Dann ist (vergl. Definition 2.36) $i_n - \bar{j}_1$ Flag–Punkt von $\mathcal{A}$ in $v(\bar{j}_1, \infty)$ und $a(i_n - \bar{j}_1) = \bar{j}_n - \bar{j}_1$, denn

i) $v(\bar{j}_1, \infty)(0, a(i_n - \bar{j}_1)) = v(\bar{j}_1, \infty)(0, \bar{j}_n - \bar{j}_1) = v(\bar{j}_1, \bar{j}_n) \in A$

ii) $\delta\Big(s_0, v(\bar{j}_1, \infty)(a(i_n - \bar{j}_1), i_n - \bar{j}_1)\Big) = \delta\Big(s_0, v(\bar{j}_n, i_n)\Big) \overset{(2.2)}{=} \delta\Big(s_0, v(\bar{j}_1, i_n)\Big)$
$= \delta\Big(s_0, v(\bar{j}_1, \infty)(0, i_n - \bar{j}_1)\Big)$

iii) Für jedes i' mit $a(i_n - \bar{j}_1) < i' < i_n - \bar{j}_1$ gilt $\delta\Big(s_0, v(\bar{j}_1, \infty)(a(i_n - \bar{j}_1), i')\Big) \neq \delta\Big(s_0, v(\bar{j}_1, \infty)(0, i')\Big)$, da i_n die kleinste Zahl mit der Eigenschaft (2.2) ist.

Sei nun $\bar{j}_m > i_n$. Wir wiederholen die obige Konstruktion und finden ein $i_m \in \mathbb{N}$ mit $\bar{j}_m < i_m$ (d.h. auch $i_n < i_m$), so daß $i_m - \bar{j}_1$ Flag–Punkt von $\mathcal{A}$ in $v(\bar{j}_1, \infty)$ ist. Dieses Vorgehen läßt sich beliebig oft wiederholen (da $\bar{J}$ eine unendliche Folge ist), also hat $\mathcal{A}$ beliebig viele Flag–Punkte in $v(\bar{j}_1, \infty)$ und damit ist $v(\bar{j}_1, \infty) \in A^\Delta$. □

Lemma 2.39 Sei $\mathcal{A}$ ein EA. Dann existiert ein MEA $\mathcal{B}$, so daß $L(\mathcal{A})^\Delta = L^\omega(\mathcal{B})$.

BEWEIS: Sei $\mathcal{A} = (S, \delta, s_0, F)$. Wir definieren eine Menge $C \subseteq \Sigma^*$ durch

$$C = \{\, x(0,i) \mid x \in \Sigma^\omega \text{ und } i \in \mathbb{N}_0 \text{ ist ein Flag–Punkt von } \mathcal{A} \text{ in } x \,\}. \tag{2.3}$$

Dann ist

$$L(\mathcal{A})^\Delta = \{\, x \mid x \in \Sigma^\omega \text{ und } |\,\{\, i \mid i \in \mathbb{N}_0 \wedge x(0,i) \in C \,\}\,| = \infty \,\}. \tag{2.4}$$

Wir zeigen zunächst, daß C regulär ist. Mit der Charakterisierung von $L(\mathcal{A})^\Delta$ in (2.4) können wir dann einen TEA angeben, der $L(\mathcal{A})^\Delta$ ω–akzeptiert.

Um einzusehen, daß C regulär ist, geben wir eine zu (2.3) äquivalente Definition der Menge C an (vergl. Definition 2.36):

$$C = \Big\{\, y \mid y \in \Sigma^*, \exists\, 0 < j < |y| \,:\, y(0,j) \in L(\mathcal{A}) \wedge \delta(s_0, y) = \delta(s_0, y(j, |y|)) \\ \wedge \Big(\, \forall\, j < i < |y| \,[\, \delta(s_0, y(0,i)) \neq \delta(s_0, y(j,i)) \,]\, \Big) \Big\}. \tag{2.5}$$

Ein NEA $\mathcal{C}$ mit $L(\mathcal{C}) = C$ kann wie folgt konstruiert werden. Für ein $y \in \Sigma^*$ verhält sich $\mathcal{C}$ zunächst wie $\mathcal{A}$. Falls $\mathcal{A}$ nach Lesen von $y(0,j)$ einen Endzustand von $\mathcal{A}$ erreicht, so kann $\mathcal{C}$ ($\mathcal{C}$ ist nichtdeterministisch) entweder mit der Simulation von $\mathcal{A}$ fortfahren, oder zusätzlich eine zweite Simulation von $\mathcal{A}$ mit Eingabe $y(j, |y|)$ beginnen. Die jeweils erreichten Zustände (von $\mathcal{A}$) der beiden Simulationen werden verglichen. Falls diese Zustände gleich sind, so ist ein Endzustand von $\mathcal{C}$ erreicht. Von diesen Endzuständen kann $\mathcal{C}$ in keine weiteren Zustände gelangen.

Wir definieren also $\mathcal{C} = (\bar{S}, \bar{\delta}, \bar{S}_0, \bar{F})$ mit $\bar{S} = S \times (S \cup \{\perp\})$ wobei $\perp \notin S$, $\bar{S}_0 = \{(s_0, \perp)\}$, $\bar{F} = \{\, (s,s) \mid s \in S \,\}$ und

$$\bar{\delta}((z_1,z_2),\sigma) = \begin{cases} \{\, (\delta(z_1,\sigma), \perp) \,\} & \text{falls } z_2 = \perp \text{ und } z_1 \in S \setminus F \\ \{\, (\delta(z_1,\sigma), \perp), (\delta(z_1,\sigma), \delta(s_0,\sigma)) \,\} & \text{falls } z_2 = \perp \text{ und } z_1 \in F \\ \{\, (\delta(z_1,\sigma), \delta(z_2,\sigma)) \,\} & \text{falls } z_1, z_2 \in S \text{ und } z_1 \neq z_2 \\ \emptyset & \text{falls } z_1, z_2 \in S \text{ und } z_1 = z_2 \end{cases}$$

für alle $\sigma \in \Sigma$.

Nach Satz 2.19 existiert dann ein EA $\mathcal{D} = (\tilde{S}, \tilde{\delta}, \tilde{s}_0, \tilde{F})$ mit $L(\mathcal{D}) = L(\mathcal{C}) = C$. Mit der Charakterisierung (2.4) ω–akzeptiert $\mathcal{D}$ genau $L(\mathcal{A})^\Delta$, d.h. $L^\omega(\mathcal{D}) = L(\mathcal{A})^\Delta$.

Also gilt für den TEA $\mathcal{E} = (\tilde{S}, \tilde{\delta}, \tilde{s}_0, \{(\emptyset, \tilde{F})\})$ ebenfalls $L^\omega(\mathcal{E}) = L(\mathcal{A})^\Delta$.

Nach Lemma 2.32 existiert dann auch ein MEA $\mathcal{B}$ mit $L^\omega(\mathcal{B}) = L^\omega(\mathcal{D}) = L(\mathcal{A})^\Delta$. □

Für den Beweis des folgenden Lemmas treffen wir noch die folgende Vereinbarung. Sei Γ eine Menge und $u \in \Gamma^*$ mit $u = u_0 u_1 \ldots u_{k-1}$. Wir bilden die *contraction* (Kurzform) $c(u)$ von u, indem wir Wiederholungen von Zeichen aus u eliminieren, d.h. mit

$$\{i_1, i_2, \ldots, i_m\} = \{\, i \in \underline{k} \mid u_j \neq u_i \text{ für } i \in \underline{j} \,\}$$

und $i_1 < i_2 < \cdots < i_m$ ist $c(u) = u_{i_1} u_{i_2} \ldots u_{i_m}$. Wir nennen u *contracted* (gekürzt), falls $c(u) = u$ gilt. Weiterhin definieren wir die Kontraktionszahl $\mu(u)$ von u als die Länge des längsten gekürzten Präfix von u. Offensichtlich ist $\mu(v) = |v|$ für ein gekürztes Wort v. Falls Γ endlich ist, so ist sicherlich $\mu(u) \leq |\Gamma|$ und insbesondere existieren nur endlich viele gekürzte Wörter über Γ.

Als Beispiel sei $\Gamma = \{a, \ldots, z\}$ und $u =$ eliminieren $\in \Gamma^*$. Dann ist $c(u) =$ elimnr und $\mu(u) = |\text{elim}| = 4$.

Lemma 2.40 Sei $\mathcal{A}$ ein EA und $\mathcal{B}$ ein MEA. Dann ist $L(\mathcal{A})L^\omega(\mathcal{B})$ ω–M–regulär.

BEWEIS: Sei $A = L(\mathcal{A})$ und $B = L^\omega(\mathcal{B})$. Sei $\mathcal{A} = (S_1, \delta_1, s_{0_1}, F)$. Nach Lemma 2.32 existiert ein TEA $\mathcal{D} = (S_2, \delta_2, s_{0_2}, \mathbb{T})$, $\mathbb{T} = \{(L_i, U_i) | 0 \leq i < m\}$, $|S_2| = n$, mit $L^\omega(\mathcal{D}) = L^\omega(\mathcal{B}) = B$. Wir werden im folgenden einen TEA $\mathcal{E} = (S', \delta', s'_0, \mathbb{T}')$ konstruieren mit $L^\omega(\mathcal{E}) = AB$.

Sei $x \in \Sigma^\omega$. Nun ist $x \in AB$ genau dann, wenn sich x in $x(0,i)$ und $x(i,\infty)$ aufspalten läßt mit $x(0,i) \in A$ und $x(i,\infty) \in B$. Sei $\rho_1 : \mathbb{N}_0 \to S_1$ die $\mathcal{A}^\omega$–Berechnung über x, dann ist $x(0,i) \in A$ gleichbedeutend mit $\rho_1(i) \in F$. Weiterhin bezeichne $\rho^i : \mathbb{N}_0 \to S_2$ die $\mathcal{D}^\omega$–Berechnung über $x(i,\infty)$, dann ist $x(i,\infty) \in B$, wenn $(\text{In}(\rho^i) \cap L_k = \emptyset) \wedge (\text{In}(\rho^i) \cap U_k \neq \emptyset)$ für ein $k \in \underline{m}$.

Wir wollen $i \in \mathbb{N}_0$ als (potentiellen) Trennpunkt bezeichnen, wenn $\rho_1(i) \in F$ gilt. Dann ist $x \in AB$, wenn es einen Trennpunkt i' gibt, so daß $\rho^{i'}$ akzeptierend ist. Leider können wir nicht alle möglichen $\mathcal{D}^\omega$–Berechnung $\rho^{i'}$ parallel simulieren, da es i.a. beliebig viele Trennpunkte geben kann. Die folgende Betrachtung löst dieses Problem:

Wir wollen zwei Trennpunkte $i_1, i_2 \in \mathbb{N}_0$ mit $i_1 < i_2$ äquivalent nennen, falls ein $l \geq i_2$ existiert mit $\rho^{i_1}(l - i_1) = \rho^{i_2}(l - i_2)$. Man beachte, daß hieraus $\rho^{i_1}(j - i_1) = \rho^{i_2}(j - i_2)$ für jedes $j \geq l$ folgt.

Also ist ρ^{i_1} akzeptierend (über $x(i_1,\infty)$) genau dann, wenn ρ^{i_2} (über $x(i_2,\infty)$) akzeptierend ist. Um zu entscheiden, ob $x \in AB$ gilt, müssen wir in der Simulation nur diejenigen Trennpunkte weiterverfolgen, für die bisher kein äquivalenter Trennpunkt aufgetaucht ist. Da $|S_2| = n$, gibt es stets höchstens n nichtäquivalente Berechnungen (von $\mathcal{D}$), die wir verfolgen müssen.

Wir werden nun die Menge der Zustände der nichtäquivalenten Berechnungen als ein Wort $u \in S_2^*$ betrachten. Daher erweitern wir die Zustandsübergangsfunktion δ_2 auf Wörter über S_2: Für $u = u_0 \ldots u_k \in S_2^*$ und $\sigma \in \Sigma$ definieren wir $\delta_2^*(u,\sigma) = \delta_2(u_0,\sigma)\ldots\delta_2(u_k,\sigma)$.

Wir definieren $\mathcal{E} = (S', \delta', s_0', \mathbb{T}')$ wie folgt:

- $S' = \{\, (s,u,i) \mid s \in S_1, u \in S_2^*, c(u) = u, 0 \le i \le n \,\}$,
- $\delta'(\,(s_1,u,i),\sigma\,) = \begin{cases} (\,\delta_1(s_1,\sigma), c(\delta_2^*(u,\sigma)), \mu(\delta_2^*(u,\sigma)))\,) & \text{falls } s_1 \notin F \\ (\,\delta_1(s_1,\sigma), c(\delta_2^*(us_{0_2},\sigma)), \mu(\delta_2^*(us_{0_2},\sigma)))\,) & \text{falls } s_1 \in F, \end{cases}$
- $s_0' = (s_{0_1}, \Lambda, 0)$,
- $\mathbb{T}' = \{\, (L_{i,k}, U_{i,k}) \mid 0 \le i < m, 0 \le k < n \,\}$ mit

$$L_{i,k} = \{\, (s_1,u,j) \in S' \mid u(k) \in L_i \text{ oder } j \le k \,\} \text{ und } U_{i,k} = \{\, (s_1,u,j) \in S' \mid u(k) \in U_i \,\}.$$

wobei c und μ wie oben definiert sind.

Wir müssen $L^\omega(\mathcal{C}) = AB$ nachweisen. Sei dazu $\rho : \mathbb{N}_0 \to S'$ die $\mathcal{C}^\omega$–Berechnung über $x \in \Sigma^\omega$ und $\varrho_i = \pi_i(\rho)$, $i = 1,2,3$. Dann entspricht ϱ_1 der $\mathcal{A}^\omega$–Berechnung ρ_1 über x und $\varrho_1(j) \in F$ g.d.w. $x(0,j) \in A$ für jedes $j \in \mathbb{N}_0$. Weiterhin sei $\varrho_2[k] : \mathbb{N}_0 \to S_2$ definiert durch

$$\varrho_2[k](n) = \begin{cases} \varrho_2(n)(k) & \text{falls } |\varrho_2(n)| > k \\ \Lambda & \text{sonst.} \end{cases}.$$

Falls $\varrho_1(j) \in F$ ist, dann gilt $\varrho_2(j+1) = c(\delta_2^*(\varrho_2(j)s_{0_2}, x(j)))$, also existiert ein $k_1 < n$ mit $\varrho_2(j+1)(k_1) = \rho^j(1)$. Aufgrund der contraction gilt dann

$$\forall\, l \ge 1\, \exists\, k_l < n\, [\, \varrho_2(j+l)(k_l) = \rho^j(l)\,]$$

mit $k_{l+1} \le k_l$ für $l \in \mathbb{N}$. Also existiert ein $l' \in \mathbb{N}$ mit $k_{l'} = k_{l'+1} = \ldots = k$. Für dieses k ist $\varrho_2(j+l'')(k) = \rho^j(l'')$ und $k < \varrho_3(j+l'')$ für alle $l'' > l'$, also ist $\mathrm{In}(\rho^j) = \mathrm{In}(\varrho_2[k])$. Damit gilt:

$$\begin{aligned} x \in AB &\iff \exists\, j\, [\, \rho_1(j) \in F \text{ und } \exists\, i\, [\, (\mathrm{In}(\rho^j) \cap L_i = \emptyset) \wedge (\mathrm{In}(\rho^j) \cap U_i \neq \emptyset)\,]\,] \\ &\iff \exists\, i\, \exists\, k\, [\, (\mathrm{In}(\varrho_2[k]) \cap L_{i,k} = \emptyset) \wedge (\mathrm{In}(\varrho_2[k]) \cap U_{i,k} \neq \emptyset)\,] \\ &\iff x \in L^\omega(\mathcal{C}). \end{aligned}$$

Damit ist AB ω–T–regulär und nach Lemma 2.32 auch ω–M–regulär. □

Nun können wir die Gleichheit der Menge der ω–regulären Mengen und der Menge der ω–M–regulären Mengen nachweisen.

Korollar 2.41 Sei $A \subseteq \Sigma^\omega$. A ist ω–regulär genau dann, wenn A ω–M–regulär ist.

BEWEIS:

($\Rightarrow$) Sei A ω–regulär. Dann existieren nach Satz 2.27 $n \in \mathbb{N}_0$ und reguläre Mengen $B_i, C_i \subseteq \Sigma^*$, $0 \leq i < n$, so daß $A = \bigcup_{0 \leq i < n} B_i C_i^\omega$.

Wir können für die Mengen C_i annehmen, daß $C_i^* = C_i$ gilt, denn $(C_i^*)^\omega = C_i^\omega$. Nun können wir Lemma 2.38 anwenden und folgern, daß $B_i C_i^\omega = B_i C_i C_i^\Delta$. Weiterhin ist $B_i C_i$ eine reguläre Menge (vergl. Satz 2.25) und nach Lemma 2.39 ist C_i^Δ ω–M–regulär, also ist auch $B_i C_i C_i^\Delta$ ω–M–regulär (Lemma 2.40). Nach Satz 2.35 ist $(\mathrm{REG}_\Sigma^{\omega M}, \cup, \cap, {}^-)$ eine Boolesche Algebra. Also ist auch $A = \bigcup_{0 \leq i < n} B_i C_i^\omega$ ω–M–regulär.

($\Leftarrow$) Sei A ω–M–regulär. Dann existiert nach Lemma 2.32 ein TEA $\mathcal{A}$ mit $L^\omega(\mathcal{A}) = A$. Sei $\mathcal{A} = (S, \delta, s_0, \mathbb{T})$ mit $\mathbb{T} = \{ (L_i, U_i) \mid 0 \leq i < m \}$ mit $S = \{s_0, \ldots s_k\}$. Wir müssen also nachweisen, daß wir den TEA $\mathcal{A}$ durch einen NEA $\mathcal{B}$ simulieren können. Wir werden zunächst für jedes $i \in \underline{m}$ den TEA $\mathcal{A}_i = (S, \delta, s_0, \mathbb{T}_i)$ mit $\mathbb{T}_i = \{(L_i, U_i)\}$ durch einen NEA $\mathcal{B}_i = (S_i, \delta_i, \{s_{0_i}\}, F_i)$ mit $L^\omega(\mathcal{A}_i) = L^\omega(\mathcal{B}_i)$ simulieren.

Sei $x \in \Sigma^\omega$ und ρ^i die $\mathcal{A}_i^\omega$–Berechnung über x. Dann ist $x \in \mathrm{L}^\omega(\mathcal{A}_i)$ genau dann, wenn $\mathrm{In}(\rho^i) \cap L_i = \emptyset$ und $\mathrm{In}(\rho^i) \cap U_i \neq \emptyset$, also wenn ein $n \in \mathbb{N}$ existiert, so daß $\mathrm{In}(\rho^i(n, \infty)) \cap L_i = \emptyset$ und $\mathrm{In}(\rho^i(n, \infty)) \cap U_i \neq \emptyset$.

Sei nun $S' = \{s'_0, \ldots, s'_k\}$, $S_i = S' \cup (S \setminus L_i)$, $s_{0_i} = s'_0$ und $F_i = U_i$. Wir definieren $\mathcal{B}_i$ derart, daß $\mathcal{B}_i$ zunächst $\mathcal{A}_i$ auf der Zustandsmenge S' simuliert, und dann nichtdeterministisch in die Simulation von $\mathcal{A}_i$ auf $S \setminus L_i$ wechseln kann. Also ist

$$\delta_i(z, \sigma) = \begin{cases} \{\delta(z, \sigma)\} & \text{falls } z \in S \setminus L_i, \\ \{(\delta(s, \sigma))'\} & \text{falls } z = s' \in S' \text{ und } \delta(s, \sigma) \in L_i, \\ \{(\delta(s, \sigma))', \delta(s, \sigma)\} & \text{falls } z = s' \in S' \text{ und } \delta(s, \sigma) \notin L_i. \end{cases}$$

Mit der obigen Bemerkung ist offensichtlich $L^\omega(\mathcal{A}_i) = L^\omega(\mathcal{B}_i)$.

Weiterhin ist

$$L^\omega(\mathcal{A}) = \bigcup_{i \in \underline{m}} L^\omega(\mathcal{A}_i) = \bigcup_{i \in \underline{m}} L^\omega(\mathcal{B}_i).$$

Wir definieren den NEA $\mathcal{B}$ durch

$$\mathcal{B} = (S_0 \times \cdots \times S_{m-1}, \delta', (s_{0_0}, \ldots, s_{0_{m-1}}), F')$$

mit

$$\delta'((z_0, \ldots, z_{m-1}), \sigma) = (\delta_0(z_0, \sigma), \ldots, \delta_{m-1}(z_{m-1}, \sigma))$$

für $(z_0, \ldots, z_{m-1}) \in S_0 \times \cdots \times S_{m-1}$, $\sigma \in \Sigma$ und

$$F' = \{ (q_0, \ldots, q_{m-1}) \mid \exists\, k \in \underline{m}\, [\, q_k \in F_k \,] \}.$$

Es gilt

$$L^\omega(\mathcal{B}) = \bigcup_{i \in \underline{m}} L^\omega(\mathcal{B}_i),$$

daher ist $L^\omega(\mathcal{B}) = L^\omega(\mathcal{A})$. □

Mit Satz 2.35 und Korollar 2.41 sind wir am Ziel:

Satz 2.42 Sei Σ ein Alphabet. Dann ist $(\mathrm{REG}_\Sigma^\omega, \cup, \cap, {}^-)$ eine Boolesche Algebra.

2.4 Probabilistische Endliche Automaten

Wir werden in diesem Abschnitt das Modell des Probabilistischen Endlichen Automaten kennenlernen. Bei einem Probabilistischen Endlichen Automaten $\mathcal{A}$ ist eine Wahrscheinlichkeitsverteilung auf der Menge der möglichen Zustandsübergänge fest gegeben, die zu jeder Eingabe v eine Wahrscheinlichkeitsverteilung auf der Menge aller $\mathcal{A}$-Berechnungen über v induziert. Die Eingabe v wird *probabilistisch* akzeptiert, falls mit Wahrscheinlichkeit größer als $1/2$ die Berechnung in einen akzeptierenden Endzustand gerät.

Probabilistische und Randomisierte Algorithmen spielen in der Informatik eine zentrale Rolle. Ausführlicher werden sie in Kapitel 4 betrachtet.

2.4.1 Probabilistische Automaten

Definition 2.43 Es sei Σ ein endliches Alphabet. Ein probabilistischer endlicher Automat (PEA, engl.: probabilistic finite automaton) über Σ ist ein 4-Tupel $\mathcal{A} = (S, M, \pi_0, F)$, wobei gilt:

- $S = \{s_0, \ldots, s_{|S|-1}\}$ ist eine endliche Zustandsmenge.
- $F \subseteq S$ ist die Menge der Endzustände.
- $\pi_0 : S \to [0,1]$ ist eine Wahrscheinlichkeitsverteilung (d.h. $\sum_{s \in S} \pi_0(s) = 1$), die sog. Anfangsverteilung (engl.: initial distribution).
- $M = (M_a : a \in \Sigma)$, wobei für jedes $a \in \Sigma$ $M_a \in [0,1]^{S \times S}$ eine stochastische Matrix ist, d.h. $\forall a \in \Sigma \, \forall s_i \in S \ \sum_{j=0}^{|S|-1} M_a(s_i, s_j) = 1$.

Berechnungen von PEAs sind etwas allgemeiner als im Fall der EA und NEA definiert, allerdings wird jeder Berechnung von A über der Eingabe $v : \underline{n} \to \Sigma$ durch die stochastischen Matrizen $M_{\sigma(i)}, 0 \leq i \leq n$ eine Wahrscheinlichkeit zugeordnet.

Definition 2.44 Eine Berechnung von $\mathcal{A}$ über $v : \underline{n} \to \Sigma$ ist eine Abbildung $\sigma : \underline{n+1} \to S$. Die Berechnung heißt akzeptierend, falls $\sigma(n) \in F$.

Definition 2.45 Die von $\mathcal{A}$ akzeptierte Sprache $L(\mathcal{A})$ ist definiert als

$$L(\mathcal{A}) = \left\{ v : \underline{n} \to \Sigma \,\middle|\, n \in \mathbb{N}_0, \, \pi_0 \cdot \prod_{i=0}^{n-1} M_{v(i)} \cdot \eta_F \geq \frac{1}{2} \right\},$$

wobei wir π_0 hier als Zeilenvektor $\pi_0 = (\pi_0(s_0), \ldots, \pi_0(s_{|S|-1}))$ auffassen und $\eta_F = (\eta_F(0), \ldots, \eta_F(|S|-1))^T$ ein Spaltenvektor ist mit

$$\eta_F(i) := \begin{cases} 1, & s_i \in F, \\ 0, & s_i \notin F. \end{cases}$$

Intuitiv gesprochen wird die Eingabe $v : \underline{n} \to \Sigma$ probabilistisch akzeptiert (d.h. liegt in $L(A)$) genau dann, wenn Pr(die Berechnung von A über v endet in einem $s_j \in F$) $> 1/2$ gilt. Wir wollen dies auch formal beweisen. Dazu werden wir zunächst zu gegebenem PEA $\mathcal{A}$ und Eingabe v den zugehörigen Wahrscheinlichkeitsraum $B(\mathcal{A}, v)$ aller $\mathcal{A}$-Berechnungen über v konstruieren.

Definition 2.46 Es seien $\mathcal{A} = (S, M, \pi_0, F)$ ein PEA über dem Alphabet Σ und $v: \underline{n} \to \Sigma$. Der Wahrscheinlichkeitsraum $B(\mathcal{A}, v) = (\Omega, P(\Omega), \Pr)$ aller $\mathcal{A}$-Berechnungen über v ist definiert durch

- $\Omega := \{\sigma \mid \sigma: \underline{n+1} \to S\}$
- $\Pr(\sigma) := \pi_0(\sigma(0)) \cdot \prod_{i=0}^{n-1} M_{v(i)}(\sigma(i), \sigma(i+1)) \quad (\sigma \in \Omega)$

Weiterhin sei $\Pr\{\mathcal{A} \text{ akzeptiert } v\} := \Pr\{\sigma \in \Omega | \sigma(n) \in F\}$.

Lemma 2.47 Für jedes $v: \underline{n} \to \Sigma$ gilt: $\Pr\{\mathcal{A} \text{ akzeptiert } v\} = \pi_0 \cdot \prod_{i=0}^{n-1} M_{v(i)} \cdot \eta_F$.

BEWEIS: Wir führen den Beweis durch Induktion nach n.
Induktionsanfang: Es sei $n = 0$. Dann gilt $v = \Lambda$, und da das leere Produkt von Matrizen als die Einheitsmatrix I definiert ist, erhalten wir

$$\Pr\{\sigma: \underline{n+1} \to S | \sigma(n) \in F\} = \pi_o \cdot \eta_F = \pi_0 \cdot \prod_{i=0}^{n-1} M_{v(i)} \cdot \eta_F.$$

Induktionsschritt: Es sei $v: \underline{n+1} \to \Sigma$. Wir definieren nun einen neuen PEA $\mathcal{A}' = (S, M, \pi_1, F)$ durch $\pi_1 := \pi_0 \cdot M_{v(0)}$. Es sei $v': \underline{n} \to \Sigma$ definiert durch $v'(i) := v(i+1), i = 0, \ldots, n-1$. Weiterhin sei zu jeder $\mathcal{A}$-Berechnung $\sigma: \underline{n+1} \to S$ die $\mathcal{A}$-Berechnung σ' über v' gegeben durch $\sigma'(i) := \sigma(i+1), 0 \leq i \leq n$. Dann ist offenbar

$$\begin{aligned}
\Pr\{\sigma: \underline{n+2} \to S | \sigma(n+1) \in F\} &= \Pr\{\sigma': \underline{n+1} \to S | \sigma'(n) \in F\} \\
&\quad \text{wobei die erste Wahrscheinlichkeit in } B(\mathcal{A}, v), \\
&\quad \text{die zweite in } B(\mathcal{A}', v') \text{ definiert ist} \\
&= \pi_1 \cdot \prod_{i=1}^{n} M_{v(i)} \cdot \eta_F \quad \text{(nach Induktionsannahme)} \\
&= \pi_0 \cdot M_{v(0)} \cdot \prod_{i=1}^{n} M_{v(i)} \cdot \eta_F
\end{aligned}$$

□

Beispiel 2.48 Der PEA $\mathcal{A} = (S, M, \pi_0, F)$ über dem Alphabet $\Sigma = \{0, 1\}$ sei gegeben durch $S = \{s_0, s_1\}$ mit $\pi_0(s_0) = 1$ und $\pi_0(s_1) = 0$, $F = \{s_1\}$ und

$$M_1 = \begin{pmatrix} 0 & 1 \\ 0 & 1 \end{pmatrix} \quad M_0 = \begin{pmatrix} 1/2 & 1/2 \\ 0 & 1 \end{pmatrix}$$

Wir möchten $L(\mathcal{A})$ bestimmen. Wir beobachten hierzu, daß nach Definition der Matrizen M_0, M_1 eine $\mathcal{A}$-Berechnung, die einmal den Zustand s_1 erreicht, diesen nicht mehr verlassen kann:

$$\Pr\{\sigma(j) = s_1 \mid \exists i \leq j \, [\, \sigma(i) = s_1 \,]\} = 1$$

Weiterhin gilt $M_1(s_0, s_1) = M_1(s_1, s_1) = 1$, somit also $\Pr\{\sigma(j) = s_1 \mid \sigma(j-1) = s_0, v(j-1) = 1\} = 1$. Wir erhalten $L(\mathcal{A}) = 0^\star \, 1 \, (0+1)^\star \cup \{0^k \mid \Pr(\mathcal{A} \text{ akzepiert } 0^k) \geq 1/2\}$. Wir bestimmen

nun das minimale k, für das dieses gilt. Es gilt

$$\begin{aligned}\Pr(\mathcal{A}\text{ akzepiert }0^k) &= \sum_{i=0}^{k-1}\Pr(\mathcal{A}\text{-Berechnung wechselt bei }i\text{-tem Symbol nach }s_1)\\ &= \sum_{i=1}^{k}\left(\frac{1}{2}\right)^{i-1}\cdot\frac{1}{2}=\frac{1-2^{-k}}{1-2^{-1}}\cdot\frac{1}{2}\end{aligned}$$

und somit $\Pr(\mathcal{A}\text{ akzepiert }0^k) > 1/2$ genau dann, wenn $k \geq 2$ ist. Wir erhalten also

$$L(\mathcal{A}) = 0^\star\,1\,(0+1)^\star \cup \{0^k \mid k \geq 2\} = \{0,1\}^* \setminus \{\Lambda, 0\}.$$

2.4.2 Randomisierte Automaten

Definition 2.49 Ein PEA $\mathcal{A} = (S, M, \pi_0, F)$ über dem endlichen Alphabet Σ heißt **randomisiert (REA)**, falls es ein $c > 0$ gibt, so daß für alle $v \in L(\mathcal{A})$ gilt: $\Pr(\mathcal{A}\text{ akz. }v) \geq 1/2 + c$. Wir nennen $\mathcal{A}$ in diesem Fall auch **Monte-Carlo EA**.

Theorem 2.50 (δ-Lemma für REA)
Es sei $\mathcal{A} = (S, M, \pi_0, F)$ ein REA über dem endlichen Alphabet Σ. Zu jedem $\delta > 0$ gibt es einen REA $\mathcal{A}_\delta$ über Σ mit $L(\mathcal{A}) = L(\mathcal{A}_\delta)$, so daß gilt: Für alle $v \in L(\mathcal{A})$ ist $\Pr(\mathcal{A}\text{ akzepiert }v) \geq 1 - \delta$.

Beweis: Zu jedem $k \in \mathbb{N}$ mit k ungerade definieren wir einen PEA $\mathcal{A}_k = (S^{(k)}, M^{(k)}, \pi_0^{(k)}, F^{(k)})$ mit Zustandsmenge $S^{(k)} = S^k$ vermöge

- $F^{(k)} = \{(s_{i_1}, \ldots, s_{i_k}) \in S^k \mid |\{j \mid 1 \leq j \leq k, s_{i_j} \in F\}| \geq k/2\}$.
- $\pi_0^{(k)}((s_{i_1}, \ldots, s_{i_k})) = \prod_{j=1}^{k} \pi_0(s_{i_j})$.
- $M_a^{(k)}((s_{i_1}, \ldots, s_{i_k}), (s_{j_1}, \ldots, s_{j_k})) = \prod_{l=1}^{k} M_a(s_{i_l}, s_{j_l})$.

Es gelte $\Pr(\mathcal{A}\text{ akzepiert }v) \geq 1/2 + c$ für ein festes $c > 0$ und alle $v \in L(\mathcal{A})$. Es sei nun $v \in L(\mathcal{A})$, $v\colon \underline{n} \to \Sigma$. Dann gilt also $p_v := \Pr(\mathcal{A}\text{ akzepiert }v) \geq 1/2 + c$. Wir möchten die Wahrscheinlichkeit $\Pr(\mathcal{A}^{(k)}\text{ akzepiert }v)$ nach unten abschätzen. Dazu definieren wir über dem Wahrscheinlichkeitsraum aller $\mathcal{A}^{(k)}$-Berechnungen $\sigma\colon \underline{n+1} \to S^{(k)}$ über v die Zufallsvariable X durch $X(\sigma) := |\{i \mid 1 \leq i \leq k, \sigma(n)_i \in F$. Mit $X_i(\sigma) := 1$, falls $\sigma(n)_i \in F$ und $X_i(\sigma) = 0$ sonst $(1 \leq i \leq k)$ gilt dann $X(\sigma) = \sum_{i=1}^{k} X_i(\sigma)$. Wegen der Linearität des Erwartungswertes erhalten wir $E[X] = \sum_{i=1}^{k} E[X_i] = k \cdot p_v$. Wir erhalten

$$\begin{aligned}\Pr(\mathcal{A}\text{ akzeptiert }v) &= \Pr\left(X \geq \frac{k}{2}\right) = 1 - \Pr\left(X \leq \frac{k}{2}\right)\\ &= 1 - \Pr\left(X \leq E(X) - (k \cdot p_v - k/2)\right)\\ &\geq 1 - \Pr\left(|X - E(X)| \geq k \cdot p_v - k/2\right)\\ &\geq 1 - \frac{1}{(k \cdot (p_v - 1/2))^r} \cdot E\left(|X - E(X)|^r\right),\end{aligned}$$

durch Anwenden der Tschebyscheff-Ungleichung. Wir wählen $r = 2$. Es gilt

$$E(X^2) = E(\sum_{i=1}^{k} X_i)^2) = \sum_{i=1}^{k} E(X_i^2) + \sum_{(i,j):i\neq j} E(X_i \cdot X_j) = k \cdot p_v + (k^2 - k) \cdot p_v^2.$$

Somit erhalten wir

$$E(X^2) - E(X)^2 = kp_v + (k^2 - k)p_v^2 - (kp_v)^2 = k(p_v - p_v^2) = k \cdot \left(1 - \left(\frac{1}{2} + c\right)^2\right).$$

Also gilt $\Pr(X \geq 1) \geq 1 - \frac{k\cdot(1-(1/2+c)^2)}{(k\cdot c)^2} \geq 1 - \delta$ für $k \geq \frac{1-(1/2+c)^2}{c^2\cdot\delta}$. □

Offenbar gibt es zu jedem EA $\mathcal{A}$ einen REA $\mathcal{B}$ mit $L(\mathcal{A}) = L(\mathcal{B})$. Das folgende Resultat besagt, daß die Umkehrung auch gilt.

Theorem 2.51 Es sei $\mathcal{B} = (S, \pi_0, M, F)$ ein REA. Dann gibt es einen EA $\mathcal{A} = (S', s_0, \delta, F')$ mit $L(\mathcal{B}) = L(\mathcal{A})$.

Wir benötigen zum Beweis noch folgende Charakterisierung regulärer Sprachen.

Lemma 2.52 Es sei $L \subseteq \Sigma^\star$ eine Sprache. Dann ist

$$R_L := \{(x, y) \in \Sigma^\star \times \Sigma^\star | \forall z \in \Sigma^\star \, (xz \in L \Leftrightarrow yz \in L) \}$$

eine Äquivalenzrelation über $\Sigma^\star$. Es gilt: L ist regulär genau dann, wenn R_L nur endlich viele verschiedene Äquivalenzklassen hat.

BEWEIS: Direkt aus der Definition folgt, daß R_L Äquivalenzrelation ist. Sei L regulär, $L = L(\mathcal{A})$ für einen EA $\mathcal{A} = (S, \delta, s_0, F)$. Wir betrachten die Menge $R_\mathcal{A}$ aller Paare $(x, y) \in \Sigma^\star \times \Sigma^\star$, so daß die $\mathcal{A}$-Berechnung über x im selben Zustand endet wie die über y. Offenbar ist $R_\mathcal{A}$ eine Äquivalenzrelation mit höchstens $|S|$ Äquivalenzklassen, und jede Äquivalenzklasse von R_L ist Vereinigung von Äquivalenzklassen von $R_\mathcal{A}$.

Habe nun umgekehrt R_L die Äquivalenzklassen $K_1, \ldots, K_k$. Es seien $x^i \in K_i, 1 \leq i \leq k$, ohne Einschränkung sei $x_1 = \Lambda$. Wir konstruieren einen EA $\mathcal{A} = (S, \delta, s_0, F)$ wie folgt: $S = \{s_1, \ldots, s_k\}$, der Startzustand sei s_1, $F = \{s_j | x_j \in L\}$ und die Übergangsfunktion δ sei definiert durch $\delta(s_i, a) = s_j$ genau dann, wenn $x^i a \in K_j$. Der Nachweis der Korrektheit der Konstruktion (d.h. $L(\mathcal{A}) = L$) bleibt zur Übung überlassen. □

BEWEIS: (von Theorem 2.51)
Es seien $\mathcal{B} = (S, \pi_0, M, F)$ ein REA mit $S = \{s_1, \ldots, s_m\}$ und $L = L(\mathcal{B})i$. Es sei $c > 0$ so, daß $\Pr(\mathcal{B}$ akzeptiert $v) \geq 1/2 + c$ für alle $x \in L$. Wir betrachten nur den Fall $|F| = 1, F = \{s_m\}$, der allgemeine Fall kann analog behandelt werden. Wir wollen zeigen, daß R_L nur endlich viele Äquivalenzklassen hat. Es seien $x_1, \ldots, x_k \in \Sigma^\star$ mit $(x_i, x_j) \notin R_L, 1 \leq i, j \leq k, i \neq j$. Dann gibt es Strings y_{ij} mit $x_i y_{ij} \in L, x_j y_{ij} \notin L$ oder $x_i y_{ij} \notin L, x_j y_{ij}$. Wir definieren zu $v: \underline{n} \to \Sigma$ $M(v) := \prod_{i=0}^{n-1} M_{v(i)}$. Die erste Zeile von $M(x_i)$ sei $\xi^i = (\xi_1^i, \ldots, \xi_m^i)$, die letzte Spalte von $M(y_{ij})$ sei $(\eta_1, \ldots, \eta_m)$. Es gilt $M(x_i y_{ij}) = M(x_i)M(y_{ij})$, $M(x_j y_{ij}) = M(x_j)M(y_{ij})$ und somit $\Pr(\mathcal{B}$ akzeptiert $x_i y_{ij}) = \xi^i \cdot \eta$, $\Pr(\mathcal{B}$ akzeptiert $x_j y_{ij}) = \xi^j \cdot \eta$. Wir betrachten den Fall $x_i y_{ij} \in L, x_j y_{ij} \notin L$. Dann gilt $\xi^i \eta \geq 1/2 + c, \xi^j \eta \leq 1/2$ und somit $2\delta \leq (\xi_i - \xi^j) \cdot \eta$ für $\delta = c/2$. Dann gilt wegen $0 \leq \eta_i \leq 1, 1 \leq i \leq m$ auch

$$2\delta \leq |\xi_1^i - \xi_1^j| + \ldots + |\xi_m^i - \xi_m^j| \quad (i \neq j, 1 \leq i, j \leq k)$$

Intuitiv ist nun schon klar, daß es nur endlich viele solche Äquivalenzklassen geben kann: Die Vektoren ξ^i haben L_1-Norm $\xi^i_1 + \ldots + \xi^i_m \leq 1$ und L_1-Abstand mindestens 2δ voneinander. Wir präzisieren dieses Argument nun. Dazu sei für $1 \leq i \leq k$

$$\sigma_i := \{(\xi_1, \ldots, \xi_m) \mid \xi^i_j \leq \xi_j, 1 \leq j \leq m, \sum_j (\xi_j - \xi^i_j) = \delta\}.$$

Jede dieser Mengen entsteht durch Translation des $(m-1)$-dimensionalen Simplex $\sigma = \{(\xi_1, \ldots, \xi_m) | \xi_j \geq 0, 1 \leq j \leq m, \sum_j \xi_j = \delta\}$. Das $(m-1)$-dimensionale Volumen von σ ist $V_{m-1}(\sigma) = b \cdot \delta^{m-1}$ für eine Konstante b, die nicht von δ abhängt. Wegen $\sum_j \xi^i_j = 1$ gilt für $\xi \in \sigma_i$ die Gleichheit $\sum_j \xi_j = 1 + \delta$. Also gilt $\sigma_i \subseteq \tau := \{(\xi_1, \ldots, \xi_m) | \sum_j \xi_j = 1 + \delta, \xi_j \geq 0, 1 \leq j \leq m\}$. Man rechnet leicht nach, daß für $i \neq j$ die Simplices σ_i und σ_j keine inneren Punkte gemeinsam haben. Daher gilt

$$k \cdot b \cdot \delta^{m-1} = V_{m-1}(\sigma_1) + \ldots + V_{m-1}(\sigma_k) \leq V_{m-1}(\tau) = b \cdot (1 + \delta)^{m-1}.$$

Also gilt $k \leq (1 + 1/\delta)^{m-1}$, d.h. R_L hat nur endlich viele Äquivalenzklassen, damit ist L regulär. □

Kapitel 3

Maschinenmodelle, Berechenbarkeit und Komplexität

In diesem Kapitel werden wir uns mit abstrakten mathematischen Maschinenmodellen auseinandersetzen, die es uns ermöglichen, den Begriff der *Berechenbarkeit* zu fassen. Wir werden uns eingehend mit dem Modell der Turing-Maschine beschäftigen, das neben anderen mathematischen Modellen als erstes dazu benutzt wurde, berechenbare Funktionen zu charakterisieren. Es hat sich jedoch herausgestellt, daß alle diese Modelle den gleichen Begriff von Berechenbarkeit beschreiben.

Daher stellt CHURCH die sogenannte *Church-Turing These* auf:

> Alles was intuitiv berechenbar ist, ist (Turing)-berechenbar.

Wir können diese These sicherlich nicht beweisen, so lange wir nicht präzisieren, was unter "intuitiv berechenbar" zu verstehen ist (ist dies überhaupt möglich?). Im weiteren Verlauf dieses Kapitels werden wir auch weitere Maschinenmodelle einführen, die als Mathematisierung moderner Rechenanlagen verstanden werden können. Hier wird sich die Church-Turing These bestätigen, denn wir werden zeigen, daß auch diese Modelle genau die Turing-berechenbaren Funktionen berechnen können.

3.1 Turing-Maschinen

In diesem Abschnitt werden wir die von A. M. TURING ("On Computable Numbers, with an Application to the Entscheidungsproblem", Proc. London Math. Soc. **42**, 230–265, 1937) eingeführte *Turing-Maschine* kennenlernen.

Operationell besteht eine Turing-Maschine aus einer endlichen Kontrolleinheit, die mittels eines Schreib/Lesekopfes mit einem beidseitig unendlichlangen, in Felder unterteilten Arbeitsband verbunden ist. Ein Schritt der Turing-Maschine besteht darin, in Abhängigkeit des Status in der endlichen Kontrolleinheit und dem Symbol unter dem Lesekopf, eine Aktion der folgenden Art auszuführen: ein (neues) Symbol auf das Arbeitsband unterhalb des Schreib/Lesekopfes zu schreiben, den Schreib/Lesekopf um maximal ein Feld nach rechts oder links zu bewegen, und den Zustand der endlichen Kontrolle zu ändern. Abbildung 3.1 zeigt eine solche Turing-Maschine.

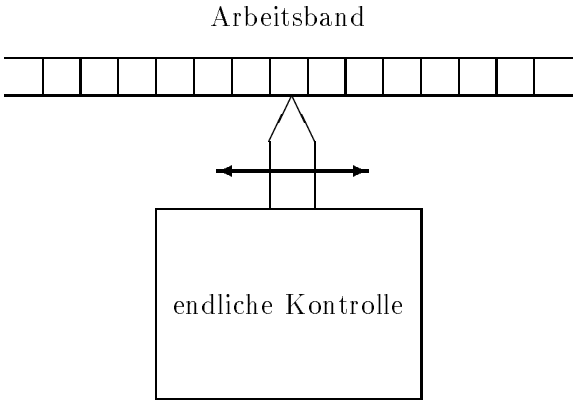


Abbildung 3.1: Einband Turing-Maschine

Wir haben schon den endlichen Automaten als endliche Kontrolleinheit kennengelernt. Das Kernstück der Turing-Maschine, die Kontrolleinheit, ist ein spezieller endlicher Automat mit Ausgabe nach jedem Schritt. Diese Ausgabe besteht aus Befehlen an den Schreib/Lesekopf (zu schreibendes Symbol und Kopfbewegung). Die Eingabe des endlichen Automaten ist das Symbol auf dem Arbeitsband, das sich direkt unterhalb des Schreib/Lesekopfes befindet. Nun können wir eine formale mathematische Definition einer Turing-Maschine geben. Wir unterscheiden auch hier, analog zu den endlichen Automaten, zwischen deterministischen und nichtdeterministischen Modellen. Wir werden zunächst das allgemeinere Modell der nichtdeterministischen Turing-Maschine definieren.

Definition 3.1 Eine *nichtdeterministische Turing-Maschine* (kurz: NTM) ist ein 5-Tupel $M = (S, \Gamma, \delta, S_0, F)$ wobei

- S die Menge der Zustände der endlichen Kontrolle ist,
- $\Gamma = \{a_1, a_2, \ldots, a_n\} \cup \{\#\}$ das endliche Bandalphabet (dabei symbolisiert $\#$ das Leerzeichen),
- $S_0 \subseteq S$ die Menge der Startzustände ist,
- $F \subseteq S$ die Menge der akzeptierenden Zustände ist,
- $\delta : S \times \Gamma \to \mathfrak{P}(S \times \Gamma \times \{-1, 0, 1\})$ die Zustandsübergangsfunktion von M ist. Dabei assoziieren wir die Kopfbewegungen links mit -1, rechts mit 1 und stehenbleiben mit 0.

Da die Zustandsübergangsfunktion nur einen endlichen Definitionsbereich hat, wird sie oft als Tabelle angegeben. Diese Tabelle heißt auch *Turing-Tafel*. Sie enthält Einträge der Form (s, a, s', b, τ) mit $\tau \in \{-1, 0, 1\}$ und $\delta(s, a) = (s', b, \tau)$.

Analog zur Automatentheorie können wir eine deterministische Turing-Maschine wie folgt definieren.

Definition 3.2 Eine *deterministische Turing-Maschine* (kurz: DTM oder TM) ist eine NTM $M = (S, \Gamma, \delta, S_0, F)$ mit den Einschränkungen, daß $|S_0| = 1$ und für alle $a \in \Gamma$ und $s \in S$ $|\delta(s, a)| \leq 1$ gelten.

Wenn wir allgemein von Turing-Maschinen sprechen, meinen wir immer nichtdeterministische Turing-Maschinen, da deterministische TMen nur ein Spezialfall sind.

Bei endlichen Automaten ergab sich eine vollständige Beschreibung des Systems durch Angabe des Zustands des endlichen Automaten. Bei Turing-Maschinen müssen wir den Inhalt des Arbeitsbandes und die Position des Schreib/Lesekopfes zusätzlich mit angeben.

Definition 3.3 Eine Zeichenfolge $c = (w, s, a, v) \in \Gamma^* \times S \times \Gamma \times \Gamma^*$ ist die *Zustandsbeschreibung* oder *Konfiguration* der TM M, in der wav die Bandinschrift und s der Zustand sind, und der Schreib/Lesekopf sich über dem Zeichen a befindet.

Definition 3.4 Die Konfiguration $c_2 = (w_2 s_2 a_2 v_2)$ heißt *Nachfolgekonfiguration* von $c_1 = (w_1 s_1 a_1 v_1)$, falls

i) $w_1 = w_2 a_2$ (oder $w_1 = \Lambda = w_2$, $a_2 = \#$), $v_2 = b v_1$ und $(s_2, b, -1) \in \delta(s_1, a_1)$ sind, oder

ii) $w_1 = w_2$, $v_1 = v_2$ und $(s_2, a_2, 0) \in \delta(s_1, a_1)$ sind, oder

iii) $w_2 = w_1 b$, $v_1 = a_2 v_2$ (oder $v_1 = \Lambda = v_2$, $a_2 = \#$) und $(s_2, b, 1) \in \delta(s_1, a_1)$ sind.

Abbildung 3.2 dient zur Veranschaulichung obiger Definition.

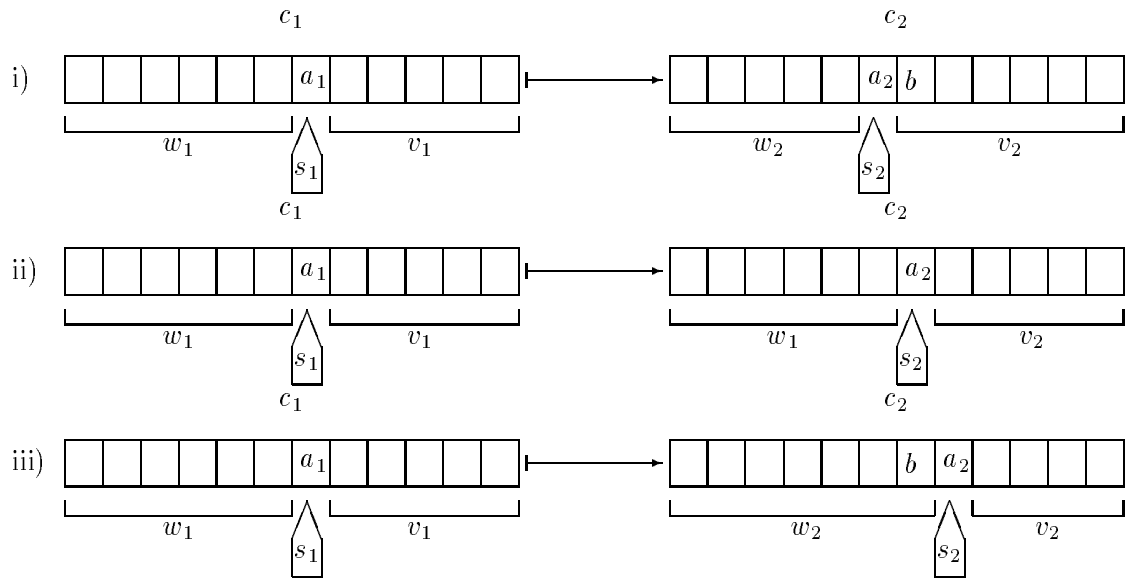


Abbildung 3.2: Mögliche Konfigurationsübergänge

Den Übergang von einer Konfiguration c in eine Nachfolgekonfiguration c' bezeichnen wir auch als einen *Rechenschritt* der TM M. Wir schreiben dann auch $c \vdash c'$.

Eine Konfiguration c heißt *Startkonfiguration*, falls $s \in S_0$ und $w = \Lambda$ sind. av bildet dann die Eingabe der Turing-Maschine. c heißt *Haltekonfiguration*, falls $\delta(s, a) = \emptyset$ und *akzeptierende Konfiguration*, falls c Haltekonfiguration mit $s \in F$ ist.

Definition 3.5 Sei M eine TM. Eine *Berechnung* von M (kurz: M-Berechnung) der Länge $k+1$ über v ist eine Funktion

$$\rho : \underline{k+1} \to \Gamma^* \times S \times \Gamma \times \Gamma^*$$

für die $\rho(0)$ eine Startkonfiguration mit Eingabe v ist und für alle $i \in \underline{k}$ $\rho(i+1)$ eine Nachfolgekonfiguration von $\rho(i)$ ist.

Eine M-Berechnung der Länge k heißt *haltend* , falls $\rho(k)$ eine Haltekonfiguration ist, und *akzeptierend*, falls $\rho(k)$ eine akzeptierende Konfiguration ist.

Wir können nun die von M erkannte Sprache $L(M)$ definieren als die Menge aller Zeichenketten $v \in \Gamma^*$, für die es eine akzeptierende Berechnung gibt:

$$L(M) = \{v \in \Gamma^* \mid \exists \text{ akzeptierende } M\text{-Berechnung über } v\}.$$

Wie bei den endlichen Automaten gilt der folgende Satz, den wir später beweisen werden (er ist nicht selbstverständlich!).

Satz 3.6 Sei M eine NTM. Dann gibt es eine DTM M' mit $L(M) = L(M')$.

Analog zu Automaten können wir auch ω–Berechnungen von M definieren. Dann ist ρ eine Funktion $\rho : \mathbb{N}_0 \to \Gamma^* \times S \times \Gamma \times \Gamma^*$, für die $\rho(0)$ eine Startkonfiguration ist und die $\rho(i) \vdash \rho(i+1)$ für alle $i \in \mathbb{N}_0$ erfüllt.

Wir bezeichnen die Menge aller haltenden (endlichen) M-Berechnungen über x mit $\mathrm{COMP}(M, x)$ und die Menge aller ω–Berechnungen von M mit $\mathrm{COMP}^\omega(M, x)$. $\mathrm{AKZ}(M, x)$ sei die Menge aller haltenden Berechnungen, falls $x \notin L(M)$, und die Menge aller akzeptierenden Berechnungen, falls $x \in L(M)$.

Falls M eine deterministische Turing-Maschine ist, so ist die Nachfolgekonfiguration jeweils eindeutig bestimmt, und es gibt höchstens eine haltende Berechnung. Anders ausgedrückt ist für alle $x \in \Gamma^*$ $|\mathrm{COMP}(M, x) \cup \mathrm{COMP}^\omega(M, x)| = 1$. Eine TM M heißt *total*, falls M für alle $x \in \Gamma^*$ hält, d.h. $\mathrm{COMP}(M, x) \neq \emptyset$ für alle $x \in \Gamma^*$ gilt.

Wir können nun den Begriff der Berechenbarkeit exakt fassen:

Definition 3.7 Eine Menge $A \subseteq \Gamma^*$ heißt *berechenbar* oder auch *rekursiv aufzählbar*, falls es eine TM M mit $A = L(M)$ gibt. A heißt *entscheidbar* oder auch *rekursiv*, falls es eine totale TM M mit $A = L(M)$ gibt.

Bisher haben wir die Bandinschrift einer haltenden Turing-Maschine außer Acht gelassen. Interpretieren wir die Bandinschrift einer Haltekonfiguration als Ausgabe der TM, so berechnet diese TM eine partielle Funktion $f : \Gamma^* \to \Gamma^*$.

Definition 3.8 Eine TM M *berechnet* eine Funktion $f : R \subset \Gamma^* \to \Gamma^*$ genau dann, wenn

i) für alle $x \in R$ $\mathrm{COMP}(M, x) \neq \emptyset$ gilt.

ii) für alle $x \in R$ und alle $\rho \in \mathrm{COMP}(M, x)$ die Bandinschrift der Endkonfiguration von ρ gleich $f(x)$ ist.

Sei $DB(f)$ die Menge aller $x \in \Gamma^*$, die 1. und 2. erfüllen. Dann berechnet M die partielle Funktion $f_M : \Gamma^* \to \Gamma^*$ mit

$$f_M = \begin{cases} f(x) & \text{falls } x \in DB(f) \\ \text{undefiniert} & \text{sonst} \end{cases} .$$

Eine partielle Funktion $f : \Gamma^* \to \Gamma^*$ heißt berechenbar, falls es ein TM M mit $f = f_M$ gibt.

Man kann den Begriff der Turing-Maschine erweitern, indem man mehrere Bänder oder auch mehrere Köpfe pro Band zuläßt. Ein wichtiges Modell einer mehrbändigen Turing-Maschine ist die Off-line Turing-Maschine, die wir später bei der Betrachtung der Komplexität von Sprachen einführen werden.

In den nächsten Abschnitten werden wir die erste sich hier aufdrängende Frage beantworten: "Ist jede berechenbare Menge auch entscheidbar?". Wie wir sehen werden, ist dies nicht der Fall.

3.2 Unentscheidbarkeit

3.2.1 Das Halteproblem

In diesem Abschnitt werden wir zeigen, daß nicht alle Sprachen berechenbar sind und vorallem, daß nicht alle berechenbaren Sprachen entscheidbar sind, d.h. für alle Γ mit $|\Gamma| \geq 1$ gilt

$$\{A \subseteq \Gamma^* \mid A \text{ entscheidbar}\} \subsetneq \{A \subseteq \Gamma^* \mid A \text{ berechenbar}\} \subsetneq \mathfrak{P}(\Gamma^*)$$

Die zweite Ungleichung ist leicht einzusehen. Die Menge aller Sprachen $A \subseteq \Gamma^*$ ist überabzählbar (wieso?), die Menge aller Turing-Maschinen, und damit auch die Menge alle berechenbaren Sprachen, ist dagegen nur abzählbar. Daher können die Mengen nicht gleich sein. Man kann leicht mittels eines Diagonalverfahrens eine nicht berechenbare Sprache L_d konstruieren (siehe [HoUl79, Chapter 8.3]).

Die erste Ungleichung wird schwerer zu zeigen sein. Wir werden für ein gewisses Problem, das sogenannte Halteproblem, zeigen, daß es zwar berechenbar aber nicht entscheidbar ist. Wir werden die Unentscheidbarkeit dieses Problems mit Hilfe des Russellschen Paradoxon (siehe Abschnitt 1.1.4) beweisen. Informell können wir das Halteproblem folgendermaßen definieren:

Probleminstanz: Turing-Maschine M mit Eingabe $w \in \Gamma^*$
Frage: Ist $\text{COMP}(M, w) \neq \emptyset$, d.h. hält M bei Eingabe w?

Folgendes Problem tritt jetzt auf: Wir haben den Begriff der Berechenbarkeit und Entscheidbarkeit nur für Mengen $A \subseteq \Gamma^*$ eingeführt. Wir müssen also jedem Problem eine zugehörige Sprache eindeutig zuordnen. Dazu kodieren wir jede Probleminstanz eineindeutig durch ein Wort in Γ^* und bilden die Sprache aller Wörter, die zu einer positiven Probleminstanz korrespondieren. Wir wollen diese Kodierung nun anhand von Turing-Maschinen besprechen. Sei $\mathcal{M}$ die Menge aller (o.B.d.A. deterministischen) Turing-Maschinen $M = (S, \Gamma, \delta, S_0, F)$ und sei Σ ein festes (aber beliebiges) Alphabet mit $\# \notin \Sigma$. Wir definieren die Kodierungsfunktion $c : \mathcal{M} \to (\Sigma \cup \{\#\})^*$ wie folgt:

$$c(M) = c(S)\#\#\#c(\Gamma)\#\#\#c(\delta)\#\#\#c(S_0)\#\#\#c(F).$$

Da alle Komponenten von M endlich sind, ist eine Beschreibung dieser Komponenten leicht möglich. Z.B. kann S durch Auflistung der Kodierungen der einzelnen Zustände, getrennt durch ein # Zeichen geschehen, genau so können Γ, S_0 und F kodiert werden. δ wird kodiert, indem alle Einträge (s, a, s', b, τ) der Turing-Tafel durch $s\#a\#s'\#b\#\tau$ kodiert werden, wobei einzelne Tupel durch ## getrennt werden.

Nun können wir eine Probleminstanz M, w durch $c(M, w) = c(M)\#\#\#\#c(w)$ kodieren. Die zugehörige Sprache ist nun

$$A = \{c(M, w) \mid \mathrm{COMP}(M, w) \neq \emptyset\}.$$

Lemma 3.9 A ist berechenbar.

BEWEIS: Wir konstruieren eine Turing-Maschine M' für A wie folgt. Zuerst überprüft M', ob die Eingabe eine korrekte Kodierung eines Paares (M, w) ist. Falls dies der Fall ist, simuliert M' M auf Eingabe w, sonst verwirft M'. Dies ist möglich, da M' die volle Spezifikation von M kennt. Falls M hält, so akzeptiert M'. Falls M nicht hält, so hält M' ebenfalls nicht, da die Simulation nicht abbricht. Daher akzeptiert M' in diesem Fall auch nicht. □

Man nennt die oben konstuierte Turing-Maschine auch *universelle* Turing-Maschine.

Satz 3.10 A ist nicht entscheidbar.

BEWEIS: Angenommen A wäre entscheidbar, dann gäbe es eine totale Turing-berechenbare Funktion (d.h. eine Turing-entscheidbare Funktion)

$$f_H : \mathcal{M} \times \Gamma^* \to \{0, 1\} \quad (M, w) \mapsto \begin{cases} 1 & M \text{ hält über } w \\ 0 & \text{sonst.} \end{cases}$$

Aus f_H konstruieren wir die Funktion $d : \mathcal{M} \to \{0, 1\}$ wie folgt:

$$d(M) = 1 - f_H(M, c(M)).$$

Da f als entscheidbar angenommen wurde, so ist auch d entscheidbar. Nach Konstruktion von d gilt:

$$d(M) = 1 \iff M \text{ hält bei Eingabe } c(M) \text{ nicht.}$$

Wir wollen das Russellsche Paradoxon anwenden. Es sei erinnert, daß es unter der Selbstanwendung eines negierten Prädikates zustande kam. Die Negierung steckt bereits in der Definition von d ($1 - a = \neg a$, falls wir $\{1, 0\}$ mit $\{\underline{W}, \underline{F}\}$ identifizieren). Die Selbstanwendung geschieht durch die Betrachtung der Turing-Maschine M_d, die d berechnet, aber, anstatt nicht akzeptierend zu halten, in eine Endlosschleife geht. Bei Eingabe $c(M)$ hält M_d also genau dann, wenn M auf Eingabe $c(M)$ nicht hält. Wählen wir $M = M_d$ so erhalten wir

$$M_d \text{ hält bei Eingabe } c(M_d) \iff M_d \text{ hält bei Eingabe } c(M_d) \text{ nicht.}$$

Dies ist sicherlich ein Widerspruch. Damit ist die Unentscheidbarkeit von A nachgewiesen. □

3.2.2 Weitere unentscheidbare Probleme

In diesem Kapitel werden wir weitere unentscheidbare Probleme kennenlernen. Das erste ist das sogenannte Selbstanwendungsproblem:

Probleminstanz: Turing-Maschine M
Frage: Hält M bei Eingabe $c(M)$?
zugehörige Sprache: $\{c(M) \mid M \text{ hält bei Eingabe } c(M)\}$.

Es ist leicht zu sehen, daß wir den Beweis des folgenden Satzes schon implizit geführt haben:

Satz 3.11 Das Selbstanwendungsproblem ist unentscheidbar.

Wir wollen nun den Begriff der Reduzierbarkeit einführen, der sich als sehr nützlich erweisen wird.

Definition 3.12 Seien $A, B \subseteq \Sigma^*$ zwei Sprachen. A heißt *reduzierbar* auf B (in Zeichen $A \leq B$), wenn es eine totale und berechenbare Funktion $f : \Sigma^* \to \Sigma^*$ gibt, so daß für alle $x \in \Sigma^*$, $x \in A$ genau dann gilt, wenn $f(x) \in B$ ist.

Anschaulich ist also A bzw. das Bild von A in B eingebettet. Wenn A also unentscheidbar ist, muß dieses auch B sein. Genauer gilt das folgende Lemma.

Lemma 3.13 Sei $A \leq B$. Dann gelten:
i) Wenn A unentscheidbar ist, so ist auch B unentscheidbar.
ii) Wenn B entscheidbar ist, so ist auch A entscheidbar.

BEWEIS: i) und ii) sind offensichtlich äquivalente Aussagen. Angenommen B sei entscheidbar, und M_B eine TM, die B entscheidet und M_f eine Turing-Maschine, die die Reduktion berechnet. Dann ist $M_A = M_B(M_f)$ eine totale Turing-Maschine, die A entscheidet (M_A simuliert zunächst M_f und dann M_B auf der Ausgabe von M_f). □

Wir werden jetzt einen Katalog unentscheidbarer Probleme angeben.

- Halteproblem
  Probleminstanz: Turing-Maschine M mit Eingabe w
  Frage: Hält M bei Eingabe w?
  zugehörige Sprache: $\text{HP} = \{c(M, w) \mid M \text{ hält bei Eingabe } w\}$.

- Selbstanwendungsproblem
  Probleminstanz: Turing-Maschine M
  Frage: Hält M bei Eingabe $c(M)$?
  zugehörige Sprache: $K = \{c(M) \mid M \text{ hält bei Eingabe } c(M)\}$.

- Halteproblem mit leerem Band
  Probleminstanz: Turing-Maschine M
  Frage: Hält M bei leerer Eingabe?
  zugehörige Sprache: $H_0 = \{c(M) \mid M \text{ hält bei leerer Eingabe}\}$.

- Postsches Korrespondenzproblem (PCP)
  Probleminstanz: Endliche Folge von Wortpaaren (x_i, y_i), $1 \leq i \leq k$ mit $x_i, y_i \in \Gamma^+$.
  Frage: Gibt es eine Folge von Indizes $i_1, \ldots, i_n \in \{1, \ldots, k\}$ mit $x_{i_1} \ldots x_{i_n} = y_{i_1} \ldots y_{i_n}$?
  zugehörige Sprache: $\text{PCP} = \{((x_1, y_1), \ldots, (x_k, y_k)) \mid \exists\, i_1, \ldots, i_n : x_{i_1} \ldots x_{i_n} = y_{i_1} \ldots y_{i_n}\}$.
- 0-1 Postsches Korrespondenzproblem
  Probleminstanz: wie oben mit $\Gamma = \{0, 1\}$
  Frage: wie oben
  zugehörige Sprache: 01PCP, sonst wie oben.
- Modifiziertes Postsches Korrespondenzproblem
  Probleminstanz: wie bei PCP
  Frage: wie bei PCP mit $i_1 = 1$.
  zugehörige Sprache: MPCP wie bei PCP mit $i_1 = 1$.

Satz 3.14 Die obigen Probleme HP, K, H_0, PCP, 01PCP und MPCP sind unentscheidbar. Die dazugehörigen Sprachen sind nicht rekursiv.

BEWEIS: siehe [HoUl79, Chapter 8] (dort auch noch zahlreiche weitere unentscheidbare Probleme). Exemplarisch wollen wir die Reduktion $\text{HP} \leq H_0$ zeigen, d.h. wir konstruieren eine Funktion f, so daß

$$\forall\, x \in \Sigma^* : \quad x \in HP \iff f(x) \in H_0.$$

Wir ordnen jedem $c(M, w) = x$ den Code derjenigen Turing-Maschine M' zu, die zunächst w auf das Band schreibt und sich dann wie M verhält. Denjenigen x, die keine Kodierung $c(M, w)$ darstellen, ordnen wir irgendein festes y zu, das keine Kodierung einer TM ist. Offensichtlich ist diese Funktion total und erfüllt die geforderten Bedingungen. □

3.3 Elementare Komplexitätstheorie

Im vorigen Abschnitt haben wir die Begriffe der Berechenbarkeit und Entscheidbarkeit kennengelernt. Es ist nun eine interessante Frage, was die *Komplexität* einer entscheidbaren Menge ist. Dazu muß jedoch zuerst der Begriff *Komplexität* definiert werden.

Wir starten mit der folgenden Definition.

Definition 3.15 Sei M eine totale TM und $x \in \Gamma^*$. Dann ist der *Zeitverbrauch* von M bei Eingabe x definiert als die Länge einer kürzesten korrekten Berechnung:

$$T_M(x) := \min\{|\rho| \mid \rho \in \text{AKZ}(M, x)\}.$$

Der *Speicherverbrauch* ist definiert als

$$S_M(x) := \min_{\rho} \max_{i} \{|(w, a, v)| \mid \rho \in \text{AKZ}(M, x), (w, s, a, v) = \rho(i), 0 \leq i < |\rho|\}.$$

Desweiteren seien

$$T_M(n) := \max\{T_M(x) \mid |x| = n\}$$

und

$$S_M(n) := \max\{S_M(x) \mid |x| = n\}$$

(wir verwenden die Funktionen T_M und S_M doppeldeutig als Funktionen mit Definitionsbereich Γ^* bzw. $\mathbb{N}$).

Es sei bemerkt, daß die Definitionen des Zeit- und Speicherverbrauches auch auf nichttotale Turing-Maschinen M erweitert werden können, indem nur Wörter $x \in L(M)$ betrachtet werden (dies war die Definition aus der Vorlesung auch für totale Maschinen), d.h.

$$T_M(n) := \max\{T_M(x) \mid, x \in L(M), |x| = n\}.$$

Bei einer nichttotalen TM M gibt es x mit $\mathrm{COMP}(M, x) = \emptyset$. Für solche x sind der Zeitverbrauch $T_M(x)$ und der Speicherverbrauch $S_M(x)$ nicht definiert.

Man nennt den Zeitverbrauch und den Speicherverbrauch auch *Komplexitätsmaße* (der TM). Wir werden später im Zusammenhang mit anderen Maschinenmodellen noch weiter Komplexitätsmaße kennenlernen.

Nun können wir die Komplexität von Sprachen definieren. Seien $T, S : \mathbb{N} \to \mathbb{N}$.

Definition 3.16 Eine Sprache $A \subseteq \Gamma^*$ hat (deterministische/nichtdeterministische) Zeitkomplexität T, falls es eine DTM/NTM M mit $A = L(M)$ und $T_M = O(T)$ gibt. Man sagt auch M sei $O(T)$ Zeit beschränkt. Eine Sprache $A \subseteq \Gamma^*$ hat (deterministische/nichtdeterministische) Speicherkomplexität S, falls es eine DTM/NTM M mit $A = L(M)$ und $S_M = O(S)$ gibt. Analog ist M $O(S)$ Speicherplatz beschränkt.

Beispiel 3.17 Die Menge $A = \{wxw \mid w \in \{0,1\}^*, x \notin \{0,1\}\}$ hat Zeitkomplexität $O(n^2)$ und Speicherkomplexität $O(n)$.

Analog zu Sprachen kann man auch Komplexitäten für Funktionen einführen: Eine Funktion $f : \Gamma^* \to \Gamma^*$ hat deterministische/nichtdeterministische Zeitkomplexität T (Speicherkomplexität S), falls es eine DTM/NTM M mit $f = f_M$ und $T_M = O(T)$ ($S_M = O(S)$) gibt.

Beispiel 3.18 Die binäre (oder auch dezimale) Addition hat eine deterministische Zeitkomplexität von $O(n)$. Die binäre Multiplikation hat eine deterministische Zeitkomplexität von $O(n^2)$ (ob die binäre Multiplikation auch eine deterministische Zeitkomplexität von $O(n)$ besitzt, ist ungeklärt!).

Wir werden jetzt den Satz 3.6 zunächst in einer "verschärften" Version beweisen:

Satz 3.19 Sei M eine NTM mit Zeitverbrauch $T_M = O(T)$. Dann gibt es eine DTM M' und ein $c \in \mathbb{N}$, so daß $L(M) = L(M')$ und $T_{M'}(n) = O(c^{T(n)})$.

BEWEIS: Sei $x \in \Gamma^*$ der Länge n gegeben und definiere $r := \max\{|\delta(s,a)| \mid s \in S, a \in \Gamma\}$ als die maximale Anzahl von möglichen Nachfolgekonfigurationen einer Konfiguration von M. Da M $O(T)$-Zeit beschränkt ist, endet jede mögliche Berechnung von M nach maximal $O(T(n))$ Schritten in einer Haltekonfiguration. Da jede Konfiguration höchstens r Nachfolgekonfigurationen hat, gibt es höchstens $r^{O(T(n))}$ viele verschiedene Berechnungen. Wir konstruieren eine DTM M', so daß M' jede der möglichen Berechnungen simuliert. Dies kann sie in Zeit $O(T(n))r^{O(T(n))}$ durchführen. Ist unter den simulierten Berechnungen eine akzeptierende Berechnung, so akzeptiert M'. Dies sichert $L(M) = L(M')$. Der Zeitverbrauch kann leicht zu $O(c^{T(n)})$ abgeschätzt werden. □

BEWEIS: (von Satz 3.6)
Sei $x \in \Gamma^*$. Wir konstruieren die DTM M' wie folgt. Für jedes $t = 1, 2, \ldots$ führt M' eine Simulation von M analog zu obiger Simulation durch, nur daß für jedes $x \in \Gamma^*$ die Simulation maximal t Schritte lang durchgeführt wird und ggf. abgebrochen wird. M' akzeptiert x, falls eine der Simulationen von M akzeptierend endet. Wir müssen $L(M) = L(M')$ zeigen. Falls $x \in L(M)$, so gibt es eine endliche akzeptierende Berechnung und M' wird diese Berechnung nach endlich vielen (erfolglosen) Simulationen finden und daher akzeptieren. Falls $x \notin L(M)$, so gibt es keine akzeptierende Berechnung und M' wird die Simulation nicht beenden, sondern endlos laufen. □

3.3.1 Off-line Turing-Maschine

Der Speicherverbrauch einer TM M ist stets mindestens so groß wie die Länge der Eingabe, d.h. $S_M(n) = \Omega(n)$. Daher werden wir jetzt ein abgewandeltes Turing-Maschinenmodell einführen, mit Hilfe dessen wir den Arbeitsspeicher vom Ein- und Ausgabespeicher trennen können.

Definition 3.20 Eine *Off-line Turing-Maschine* ist eine dreibändige Turing-Maschine. Jedes Band besitzt einen Kopf. Band 1 ist das Eingabeband, von dem nur gelesen, Band 2 ist das Arbeitsband, auf dem gelesen und geschrieben werden kann, und Band 3 ist das Ausgabeband, auf das nur geschrieben werden darf. Weiterhin darf sich der Schreibkopf des Ausgabebandes nicht nach links bewegen.

Das Verbot der Linksbewegung des Schreibkopfes des Ausgabebandes verhindert, daß einmal gemachte Ausgaben korrigiert werden können, wenn der Kopf sich danach bewegt hat. Ein Schaubild einer Off-line TM zeigt Abbildung 3.3.

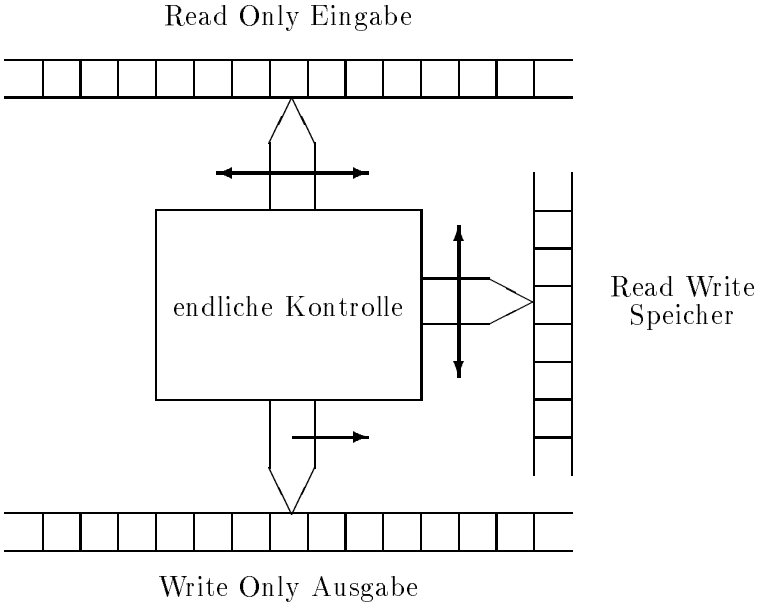


Abbildung 3.3: Off-line Turing-Maschine

Formal ist eine Off-line NTM M ein 5-Tupel $M = (S, \Gamma, \delta, S_0, F)$. Dabei sind S, Γ, S_0, F definiert wie im Fall gewöhnlicher NTMs und δ wie folgt:

$$\delta : S \times \Gamma \times \Gamma \to \mathfrak{P}(S \times \underbrace{\{-1,0,-1\}}_{\text{Eingabeband}} \times \overbrace{\Gamma \times \{-1,0,1\}}^{\text{Arbeitsband}} \times \underbrace{\Gamma \times \{0,1\}}_{\text{Ausgabeband}}).$$

Wir definieren eine Konfiguration c von M durch eine Zeichenkette

$$(w_1 s a v_1 \not\in w_2 \not\in b v_2 \not\in v_3) \in \Gamma^* \times S \times \Gamma \times \Gamma^* \times \{\not\in\} \times \Gamma^* \times \{\not\in\} \times \Gamma \times \Gamma^* \times \{\not\in\} \times (\Gamma \cup \{\#\})^*,$$

wobei $\not\in \notin (S \cup \Gamma)$ ein Trennzeichen ist. $w_1 a v_1$ ist die Bandinschrift des Arbeitsbandes, $w_2 b v_2$ ist die Bandinschrift des Eingabebandes, v_3 ist die Bandinschrift des Ausgabebandes, s ist der Zustand. Der Kopf des Arbeitsbandes befindet sich über dem Feld mit dem Symbol a, der Kopf des Eingabebandes auf dem Feld mit dem Symbol b und der Kopf des Ausgabebandes auf dem letzten Symbol von v_3. Diese Symbol ist #, falls sich der Kopf hinter der eigentlichen Ausgabe über einem Leerzeichen befindet. Dies ist die einzige Position in v_3, die aus einem Leerzeichen bestehen darf.

Analog zu den TMs definieren wir Off-line DTMs durch die Einschränkungen $|S_0| = 1$ und $|\delta(s,a,b)| \leq 1$ für alle $s \in S$ und $a, b \in \Gamma$. Ebenso definieren wir die Begriffe einer (ω-) Berechnung, einer haltenden und akzeptierenden Berechnung und der erkannten Sprache und den Zeitverbrauch, anders jedoch den Berechnungsbegriff für Funktionen und den Speicherverbrauch.

Eine Off-line TM berechnet eine Funktion $f : \Gamma^* \to \Gamma^*$, wenn für alle haltenden Berechnungen bei Eingabe x auf dem Eingabeband, der Bandinhalt des Ausgabebandes in der Haltekonfiguration $f(x)$ ist.

Den Speicherverbrauch definieren wir durch

$$S_M(x) = \min_{\rho} \max_{i} \{|(w_1, a, v_1)| \mid \rho \in \mathrm{AKZ}(M,x), (w_1, s, a, v_1, \ldots) = \rho(i), 0 \leq i < |\rho|\}$$

und

$$S_M(n) = \max\{S_M(x) \mid |x| = n\}.$$

Eine besonders wichtige Klasse von Funktionen sind die sogenannten *Speicherschrittfunktionen*. Eine Funktion $f : \mathbb{N} \to \mathbb{N}$ heißt speicherkonstruierbar, falls es eine TM M gibt, so daß für jedes $n \in \mathbb{N}$ stets $f(n) = S_M(n)$ gilt. f heißt voll speicherkonstruierbar, falls $f(n) = S_M(x)$ für alle x der Länge n gilt. Speicherschrittfunktionen sind voll speicherkonstruierbare Funktionen und können z.B. dazu benutzt werden, den Speicher, den eine Turing-Maschine verbrauchen darf, zu markieren. Somit erzwingt man, daß die Berechnung höchstens $f(n)$ Speicher benötigt. Diese Konstruktion wird bei der Untersuchung von Komplexitätsklassen oft angewandt. Die folgenden Funktionen sind Beispiele von Speicherschrittfunktionen: n, n^2, 2^n, $n!$, $\lceil\sqrt{n}\rceil$ und $\log n$.

3.3.2 Komplexitätsklassen

Die Menge aller Sprachen einer bestimmten Komplexität bilden eine sogenannte Komplexitätsklasse. Wir führen die folgenden Komplexitätsklassen ein:

- DTIME(T) und NTIME(T) sind die Menge aller Sprachen mit deterministischer / nichtdeterministischer Zeitkomplexität $O(T)$.

- DSPACE(S) und NSPACE(S) sind die Menge aller Sprachen mit deterministischer / nichtdeterministischer Speicherkomplexität $O(S)$ bzgl. einer Off-line-TM.
- $\mathcal{P} = \bigcup_{i\in\mathbb{N}} \mathrm{DTIME}(n^i)$.
- $\mathcal{NP} = \bigcup_{i\in\mathbb{N}} \mathrm{NTIME}(n^i)$.
- $\mathrm{EXP} = \bigcup_{i\in\mathbb{N}} \mathrm{DTIME}(2^{(n^i)})$.

In dieser Notation kann Satz 3.19 auch geschrieben werden als

$$\mathrm{NTIME}(T) \subseteq \bigcup_{c\in\mathbb{N}} \mathrm{DTIME}(c^T).$$

Die Beziehungen dieser und anderer Komplexitätsklassen sind in der Literatur ausgiebig untersucht worden. Die folgende Inklusionskette ist aus obigen klar:

$$\mathcal{P} \subseteq \mathcal{NP} \subseteq \mathrm{EXP}$$

Wir werden hier die Beziehung zwischen $\mathcal{P}$ und $\mathcal{NP}$ genauer studieren, d.h. die Frage angehen ob $\mathcal{P}$ gleich $\mathcal{NP}$ ist. Ähnlich wie beim Beweis der Unentscheidbarkeit von Problemen werden wir hier einen geeigneten Reduzierbarkeitsbegriff einführen.

Definition 3.21 Seien Σ und Γ zwei endliche Alphabete. Eine Sprache $A \subseteq \Sigma^*$ heißt *Polynomialzeit reduzierbar* auf eine Sprache $B \subseteq \Gamma^*$, falls es eine totale Funktion $f : \Sigma^* \to \Gamma^*$ mit polynomieller (det.) Zeitkomplexität gibt, so daß für alle $a \in \Sigma^*$ $a \in A$ äquivalent mit $f(a) \in B$ ist.

Falls A Polynomialzeit reduzierbar auf B ist, so schreiben wir auch $A \leq_P B$. Offensichtlich ist $\leq_P$ eine transitive Relation, d.h. aus $A \leq_P B$ und $B \leq_P C$ folgt $A \leq_P C$. Der Einfachheit halber nehmen wir an, daß wir es im folgenden nur noch mit dem Alphabet Σ zu tun haben.

Mit Hilfe der Polynomialzeit Reduzierbarkeit können wir eine Teilmenge hartnäckiger Sprachen von $\mathcal{NP}$ charakterisieren.

Definition 3.22 Eine Sprache $A \subseteq \Sigma^*$ heißt

- $\mathcal{NP}$-*einfach* (leicht), falls $A \in \mathcal{NP}$,.
- $\mathcal{NP}$-*schwer* (hart), falls für alle $B \in \mathcal{NP}$ gilt $B \leq_P A$,
- $\mathcal{NP}$-*vollständig*, falls A sowohl $\mathcal{NP}$-einfach als auch $\mathcal{NP}$-schwer ist.

Der nächste Satz zeigt, daß wir die Frage "$\mathcal{P} = \mathcal{NP}$?" auf die Frage "$A \in \mathcal{P}$?" für gewisse A reduziert haben.

Satz 3.23 Sei A $\mathcal{NP}$-vollständig. Dann ist $\mathcal{P} = \mathcal{NP}$ genau dann, wenn $A \in \mathcal{P}$.

BEWEIS: Falls $\mathcal{P} = \mathcal{NP}$ ist, so folgt sofort $A \in \mathcal{P}$. Für die Umkehrung sei $A \in \mathcal{P}$ angenommen und M eine DTM für A mit Zeitverbrauch p, wobei p ein geeignet gewähltes Polynom ist. Sei B eine beliebige Sprache aus $\mathcal{NP}$. Da A $\mathcal{NP}$-vollständig ist, gibt es eine Polynomialzeit berechenbare Funktion f, so daß für alle $x \in \Sigma^*$ $x \in B$ genau dann gilt, wenn $f(x) \in A$ (hier sind A und B umgekehrt zur Definition gebraucht). Sei f durch die DTM M' in Zeit q berechenbar, wobei q ein geeignet gewähltes Polynom ist. Wir konstruieren aus M und M' jetzt eine DTM M'', die B erkennt. Bei Eingabe x berechnet M'' durch Simulation von M' zunächst $f(x)$. Danach interpretiert M'' $f(x)$ als Eingabe von M und simuliert diese, nachdem M'' den Schreib/Lesekopf auf das erste Symbol von $f(x)$ gefahren hat. M'' akzeptiert genau dann, wenn M akzeptiert.

Nach Konstruktion ist $L(M'') = B$. Nun müssen wir den Zeitverbrauch von M'' abschätzen. Sei $n = |x|$. Die Simulation von M' benötigt maximal $q(n)$ Schritte. Da pro Schritt höchstens ein Feld beschrieben werden kann, ist die Länge von $f(x)$ durch $n+q(n)$ beschränkt. Dies beschränkt auch die Anzahl der Schritte, um den Anfang von $f(x)$ zu erreichen. Die Simulation von M bei Eingabe $f(x)$ benötigt daher höchstens $p(n + q(n))$ viele Schritte. Insgesamt hat M'' also einen Zeitverbrauch von höchstens $q(n) + (n + q(n)) + p(n + q(n))$, dies ist jedoch polynomiell. Daher liegt $B \in \mathcal{P}$. Da die Konstruktion für alle $B \in \mathcal{NP}$ gültig war, folgt $\mathcal{P} = \mathcal{NP}$. □

Aufgrund der Transitivität von $\leq_P$ gilt das folgende Korollar.

Korollar 3.24 Sei A $\mathcal{NP}$-vollständig und $B \in \mathcal{NP}$ mit $A \leq_P B$. Dann ist B auch $\mathcal{NP}$-vollständig.

BEWEIS: Da A $\mathcal{NP}$-vollständig ist, gilt für alle $C \in \mathcal{NP}$, daß $C \leq_P A$. Da $\leq_P$ transitiv ist, folgt aus $A \leq_P B$, daß $C \leq_P B$. Daher ist B $\mathcal{NP}$-schwer und folglich auch $\mathcal{NP}$-vollständig. □

Wir werden die Menge aller $\mathcal{NP}$ vollständigen Sprachen mit $\mathcal{NPC}$ bezeichnen. Im folgenden werden wir einige Beispiele von Sprachen in $\mathcal{NPC}$ angeben. Wir werden dies in der Notation von Entscheidungsproblemen tun. Ein Entscheidungsproblem korrespondiert in kanonischer Weise zu einer Sprache, nämlich zu derjenigen Sprache, die die Kodierungen von Probleminstanzen mit positiver Entscheidung enthalten. Da eine solche Kodierung nicht eindeutig ist, gibt es zu einem Entscheidungsproblem in der Regel eine Menge zugehöriger Sprachen.

Beispiel 3.25 Erfüllbarkeitsproblem Boolescher Formeln

Probleminstanz: n-stellige Boolesche Formel f in KNF.
Frage: Gibt es eine erfüllende Belegung der Variablen mit $a \in \{0,1\}^n$, d.h. ist $f(a) = 1$.
zugehörige Sprache: SAT $= \{f \mid f$ ist erfüllbare KNF$\}$.

Das obige Problem heißt auch SAT-Problem (satisfiablility). Die $\mathcal{NP}$-Vollständigkeit dieses Problems wurde zuerst von Cook [Coo71] nachgewiesen.

Satz 3.26 Das Erfüllbarkeitsproblem Boolescher Formeln ist $\mathcal{NP}$-vollständig.

BEWEIS: Ein Beweis kann z.B. in [HoUl79, Chapter 13.2] nachgelesen werden. □

Der Nachweis der $\mathcal{NP}$-Vollständigkeit anderer Probleme $B \in \mathcal{NP}$ kann jetzt durch Nachweis der Relation SAT $\leq_P B$ durchgeführt werden.

Die folgenden Probleme sind auch $\mathcal{NP}$-vollständig:

- Erfüllbarkeitsproblem für 3-KNF Formeln
  Probleminstanz: n-stellige Boolesche Formel f in 3-KNF, d.h. jede Klausel von f enthält höchstens 3 Literale.
  Frage: Gibt es eine erfüllende Belegung der Variablen mit $a \in \{0,1\}^n$, d.h. ist $f(a) = 1$?
  zugehörige Sprache: 3-SAT $= \{f \mid f$ ist erfüllbare 3-KNF$\}$.

- Cliquenproblem.
  Probleminstanz: ungerichteter Graph G, Parameter $k \in \mathbb{N}$.
  Frage: Gibt es eine Clique in G, die mindestens k Knoten enthält?
  zugehörige Sprache: CLIQUE $= \{(G,k) \mid G$ enthält eine Clique mit $\geq k$ Knoten$\}$.

- Problem des Hamiltonschen Kreises
  Probleminstanz: ungerichteter Graph G
  Frage: Gibt es einen Kreis in G, der jeden Knoten von G genau einmal enthält (ein solcher Kreis heißt hamiltonsch)?
  zugehörige Sprache: HK $= \{G \mid G$ enthält einen hamiltonschen Kreis$\}$.

- Problem des Handlungsreisenden
  Probleminstanz: ungerichteter, kantengewichteter Graph G, Parameter k.
  Frage: Gibt es einen hamiltonschen Kreis K in G, dessen Gesamtgewicht (Summe der Gewichte der Kanten in K) höchstens k ist?
  zugehörige Sprache: HR $= \{(G,k) \mid G$ enthält einen hamilt. Kreis vom Gewicht $\leq k\}$.

  Man kann die Knoten von G mit Städten identifizieren und die Gewichte $w(i,j)$ der Kanten (i,j) als die Distanz $dist(i,j) = w(i,j)$ zwischen den Städten i und j auffassen (die Gewichte der Kanten müssen dann einer Dreiecksungleichung $w(i,j) + w(j,l) \geq w(i,l)$ für alle i,j,l gehorchen). Gefragt wird jetzt, ob es eine Rundreise mit einer Gesamtlänge $\leq k$ gibt, in der alle Städte besucht werden.

Satz 3.27 Die Probleme 3-SAT, CLIQUE, HK und HR sind $\mathcal{NP}$-vollständig.

BEWEIS: Trivialerweise liegen die obigen Problem in $\mathcal{NP}$. Außerdem gilt

$$\text{SAT} \leq_P \text{3-SAT} \leq_P \text{CLIQUE} \leq_P \text{HK} \leq_P \text{HR}.$$

□

3.4 Random Access Maschinen

Nun wollen wir die grundlegenden traditionellen Rechnermodelle verlassen und uns den modernen Berechnungsmodellen zuwenden. Das grundlegende Modell ist das der Random Access Maschine, das in [ShSt63] eingeführt wurde. Es ist stark an das J. v. NEUMANNSCHE Rechnermodell angelehnt.
Das Modell von VON NEUMANN besteht aus einer Rechen- und Steuereinheit, einem Hauptspeicher, der sowohl die Instruktionen als auch die Daten beinhaltet, zusätzlich Ein- und Ausgabeeinheiten und einem Hintergrundspeicher. Die Steuereinheit kann wahlfrei auf die Speicherzellen im Hauptspeicher zugreifen, diese Daten manipulieren und wieder speichern. Dabei wird die Zeit für eine elementare Operation durch die Zeit, die für die Kommunikation zwischen Speicher und Steuereinheit notwendig ist, majorisiert. Das v. NEUMANNsche Rechnermodell ist in fast allen gängigen sequentiell arbeitenden Computern verwirklicht.

Definition 3.28 Random Access Maschine
Eine *Random Access Maschine* , kurz auch RAM genannt, ist ein theoretisches Rechnermodell. Eine RAM besteht aus einem *Akkumulator* α und einer beschränkten Anzahl von *Indexregistern* $\gamma_1, \gamma_2, \ldots, \gamma_g$, einer abzählbaren Menge von *Speicherzellen* ρ_i, $i \in \mathbb{N}$, auf die wahlfrei zugegriffen werden kann, und einem endlichen Programm mit einem Programmzähler LC. Der Inhalt der Speicherzellen, der Register und des Programmzählers sind beliebig große ganze Zahlen. Somit kann der Speicher auch als eine Funktion $\rho : \mathbb{N}_0 \to \mathbb{Z}$ aufgefaßt werden. Die Abbildung 3.4 zeigt ein Schaubild der RAM.

Wir wollen uns im weiteren auf $g = 3$ Indexregister beschränken. Dies stellt jedoch keine generelle Einschränkung dar.

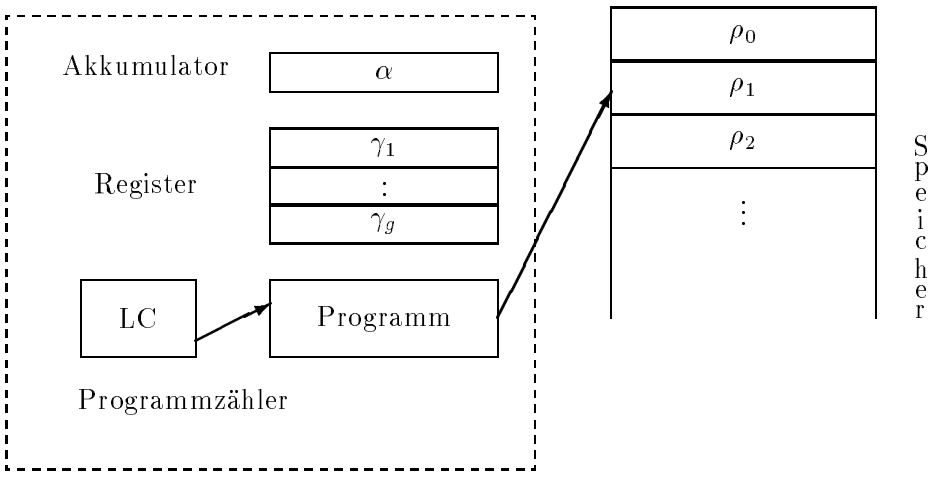


Abbildung 3.4: Random Access Maschine

Eine Random Access Maschine kann als eine Art 'springende' TM angesehen werden, da der wahlweise Zugriff auf den Speicher einem Sprung des Lese/Schreibkopfes an die entspechende Bandzelle entspricht (anstatt durch eine entsprechende Folge von Kopfbewegungen).

Nun wollen wir eine genaue Beschreibung des Maschinenmodells RAM geben. Um die syntaktische Definition zu geben, benutzen wir die Backus Naur Form.

Zunächst definieren wir die syntaktischen Kategorien:

Definition 3.29

- $<$ REG $>::= \alpha \mid \gamma_j$, $1 \leq j \leq 3$
- $<$ LOP $>::= \rho_i \mid <$ REG $>$
- $<$ OP $>::= i \mid \rho_i \mid <$ REG $>$
- $<$ MOP $>::= \rho_{i+\gamma_j}$ mit $i \in \mathbb{N}$ und $1 \leq j \leq 3$
- $<$ IND $>::= \gamma_j$ mit $1 \leq j \leq 3$
- $<$ AC $>::= \alpha$

- $< \underline{\text{NUM}} > ::= 0 \mid 1 \mid 2 \mid \ldots$

Dabei bedeutet ρ_i den Inhalt der Speicherzelle mit der Nummer i (statt ρ_i schreiben wir auch $\rho(i)$). Wie oben gesagt, kann ρ auch als Funktion $\rho : \mathbb{N}_0 \to \mathbb{Z} \supset \mathbb{N}$ interpretiert werden. Eine Adressenrechnung wird durch die Verwendung der indirekten Adressierung möglich. Sie erfolgt über die Benutzung modifizierter Operanden $< \underline{\text{MOP}} >$.

Wir können die Befehle der RAM in vier verschiedene Kategorien einteilen:

- Transportbefehle,
- Sprungbefehle,
- arithmetische Befehle und
- Indexbefehle.

Die Transportbefehle dienen zum Laden und Speichern des Akkumulatorinhaltes und der Registerinhalte von bzw. in die Speicherzellen. Für diese Aufgabe stehen 4 Befehle zur Verfügung:

$$< \underline{\text{TB}} > \quad ::= \quad < \underline{\text{REG}} > \leftarrow < \underline{\text{OP}} > \mid < \underline{\text{AC}} > \leftarrow < \underline{\text{MOP}} > \mid < \underline{\text{LOP}} > \leftarrow < \underline{\text{REG}} > \mid$$
$$< \underline{\text{MOP}} > \leftarrow < \underline{\text{AC}} > .$$

Die Sprungbefehle ermöglichen die Steuerung des Programmflusses:

$$< \underline{\text{SB}} > ::= \underline{\text{GOTO}} \ < \underline{\text{NUM}} > \mid \underline{\text{IF}} \ < \underline{\text{REG}} > < \underline{\text{REL}} > 0 \ \underline{\text{THEN}} \ \underline{\text{GOTO}} \ < \underline{\text{NUM}} >$$

wobei $< \underline{\text{REL}} >$ die Menge der Vergleichsoperatoren darstellt: $< \underline{\text{REL}} > ::= = \mid \neq \mid \leq \mid < \mid > \mid \geq$.

Die arithmetischen Befehle ermöglichen der RAM das Ausführen elementarer Rechenoperationen:

$$< \underline{\text{AB}} > ::= < \underline{\text{AC}} > \leftarrow < \underline{\text{AC}} > < \underline{\text{AO}} > \ < \underline{\text{OP}} > \mid < \underline{\text{AC}} > \leftarrow < \underline{\text{AC}} > < \underline{\text{AO}} > \ < \underline{\text{MOP}} >$$

mit den elementaren arithmetische Operationen $< \underline{\text{AO}} > ::= + \mid - \mid * \mid \underline{\text{DIV}} \mid \underline{\text{MOD}}$.

Zur Modifizierung der Indexregister stehen noch Indexbefehle zur Verfügung:

$$< \underline{\text{IB}} > ::= < \underline{\text{IND}} > \leftarrow < \underline{\text{IND}} > + < \underline{\text{NUM}} > \mid < \underline{\text{IND}} > \leftarrow < \underline{\text{IND}} > -NTNUM.$$

Damit sind alle Klassen von Befehlen definiert. Ein RAM-Programm ist nun eine (numerierte) Folge von RAM-Befehlen. Genauer:

Definition 3.30 Sei $< \underline{\text{B}} > ::= < \underline{\text{TB}} > \mid < \underline{\text{SB}} > \mid < \underline{\text{AB}} > \mid < \underline{\text{IB}} >$ die Menge aller RAM-Befehle. Dann ist die Kategorie der *RAM-Programme* als Menge von numerierten RAM-Befehlen durch $< \underline{\text{RP}} > ::= \{< \underline{\text{NUM}} > < \underline{\text{B}} >\}$ definiert.

Als Konvention nehmen wir an, daß die Befehle eines RAM-Programms fortlaufend von 0 an numeriert sind. Zum Start eines RAM-Programmes nehmen wir zusätzlich an, daß alle Speicherzellen mit 0 initialisiert sind und nur die Speicherzelle ρ_0 die Eingabe enthält. Nach Beendigung des RAM-Programms wird die Ausgabe in der Speicherzelle ρ_1 zur Verfügung gestellt.

Unsere Definition der RAM läßt folgende Vergleiche mit der Turing-Maschine zu:

- Bei der RAM ist ein wahlfreier Zugriff auf den Speicherraum möglich.
- Bei RAM und Turing-Maschine ist das Programm fest vorgegeben und getrennt von den zu verarbeitenden Daten.
- Die RAM kann im Gegensatz zur Turing-Maschine in einer Speicherzelle beliebig große Zahlen (und damit beliebig viel Information) speichern.
- Der Speicherraum von RAM und Turing-Maschine ist abzählbar unendlich.

3.4.1 Semantik von RAM-Programmen

Den Zustand einer Berechnung einer Turing-Maschine haben wir durch die Konfiguration der Turing-Maschine beschrieben. Dabei beinhaltet eine Konfiguration einer Turing-Maschine die vollen Information über den Zustand des eigentlichen Programmes (bei der Turing-Maschine ist dies der Zustand der endlichen Kontrolle) und den Zustand der Speichermedien (Bandinschriften und Kopfpositionen). Um eine Zustandsbeschreibung eines RAM-Programmes P angeben zu können, müssen wir also vorher eine geeignete Notation zur Beschreibung des Zustandes des Speichers und der Register einführen.

Formal ordnen wir jedem Register und jeder Speicherzelle einen Identifier zu, dies bildet die Kategorie $<$ RAM-IDE $>$:

$$< \underline{\text{RAM-IDE}} >::= a \mid g_1 \mid g_2 \mid g_3 \mid r_{<\underline{\text{NUM}}>}.$$

Dabei ordnen wir α den Identifier a, γ_j den Identifier g_j und ρ_i den Identifier r_i und umgekehrt zu. Dies liefert uns die *semantische Speicherfunktion*

$$\mathbb{C} : \underline{\text{RAM-IDE}} \rightarrow \underline{\text{NUM}}.$$

Aus Gründen der Einfachheit nehmen wir an, daß alle Speicherzellen außer den Eingabezellen mit 0 initialisiert sind (dann ist es nicht nötig den Wert unbound in den Wertebereich von $\mathbb{C}$ aufzunehmen).

Der Zustand der endlichen Kontrolle einer Turing-Maschine entspricht bei der RAM der Wert des Programmzählers LC. Damit haben wir eine vollständige Zustandsbeschreibung einer RAM mit RAM Programm P.

Definition 3.31 Eine *RAM Konfiguration* ist ein Paar $C = (p, \mathbb{C})$, wobei $p \in \underline{\text{NUM}}$ der Wert des Programmzählers und $\mathbb{C} : \underline{\text{RAM-IDE}} \rightarrow \underline{\text{NUM}}$ die semantische Speicherfunktion ist.

Ein Berechnungschritt eines RAM-Programms P ist analog zu den Turing-Maschinen als eine Konfigurationsänderung in einem Schritt definiert. Wir schreiben dafür auch $C \xrightarrow[P]{1} C'$. Die Konfigurationsänderung bei der Turing-Maschine ist durch die Zustandsübergangsfunktion eindeutig bestimmt; eine solche Zustandsübergangsfunktion entspricht der Angabe der Semantik von RAM-Befehlen und RAM-Programmen. Eine Zustandsübergangsfunktion gibt also die *semantische Interpretation* der syntaktischen Befehle.

Jede Befehlskategorie definiert eine Menge von Befehlen. So definiert z.B. die Kategorie

$$< \underline{\text{TB}} >::=< \underline{\text{AC}} >\leftarrow< \underline{\text{MOP}} >$$

Befehle A der Art $\alpha \leftarrow \rho(\gamma_j + i)$ mit $1 \leq j \leq 3$ und $i \in \mathbb{N}_0$. Wir nennen solche Befehle auch *semantische Klauseln*.

In Abhängigkeit der semantischen Klauseln werden wir nun einen Berechnungsschritt eines RAM-Programmes P definieren.

Definition 3.32 Sei P ein RAM-Programm und $C = (p, \mathbb{C})$ eine Konfiguration, d.h. p die Nummer eines RAM-Befehles A des RAM-Programms und $\mathbb{C}$ eine Speicherfunktion. Bezeichne p^+ die Nummer des nachfolgenden Befehls und sei $I \in \underline{\text{RAM-IDE}}$. Dann gilt

$$(p, \mathbb{C}) \overset{1}{\underset{P}{\rightarrow}} (p', \mathbb{C}')$$

genau dann, wenn

i) für $A = \gamma_j \leftarrow \rho(i)$ gelten: $p' = p^+$ und

$$\mathbb{C}'(I) = \begin{cases} \mathbb{C}(r_i) & \text{falls } I = g_j \\ \mathbb{C}(I) & \text{sonst.} \end{cases}$$

ii) für $A = \alpha \leftarrow \rho(i + \gamma_j)$ gelten: $p' = p^+$ und

$$\mathbb{C}'(I) = \begin{cases} \mathbb{C}(r_{i+k}) & \text{falls } I = a \text{ und } \mathbb{C}(g_j) = k \\ \mathbb{C}(I) & \text{sonst.} \end{cases}$$

iii) für $A = \rho(i) \leftarrow \gamma_j$ gelten: $p' = p^+$ und

$$\mathbb{C}'(I) = \begin{cases} \mathbb{C}(g_j) & \text{falls } I = r_i \\ \mathbb{C}(I) & \text{sonst.} \end{cases}$$

iv) für $A = \rho(\gamma_j + i) \leftarrow \alpha$ gelten: $p' = p^+$ und

$$\mathbb{C}'(I) = \begin{cases} \mathbb{C}(a) & \text{falls } I = r_{k+i} \text{ und } k = \mathbb{C}(g_j) \\ \mathbb{C}(I) & \text{sonst.} \end{cases}$$

v) für $A = \underline{\text{GOTO}}\ k$ gelten: $p' = k$ und $\mathbb{C}' = \mathbb{C}$.

vi) für $A = \underline{\text{IF}}\ \gamma_j = 0\ \underline{\text{THEN}}\ \underline{\text{GOTO}}\ k$ gelten: $\mathbb{C}' = \mathbb{C}$ und

$$p' = \begin{cases} k & \text{falls } \mathbb{C}(g_j) = 0 \\ p^+ & \text{sonst.} \end{cases}$$

vii) für $A = \alpha \leftarrow \alpha + \rho(\gamma_j + i)$ gelten: $p' = p^+$ und

$$\mathbb{C}'(I) = \begin{cases} \mathbb{C}(a) + \mathbb{C}(r_{k+i}) & \text{falls } I = a \text{ und } k = \mathbb{C}(g_j) \\ \mathbb{C}(I) & \text{sonst.} \end{cases}$$

viii) für $A = \gamma_j \leftarrow \gamma_k \pm n$ gelten: $p' = p^+$ und

$$\mathbb{C}'(I) = \begin{cases} \mathbb{C}(g_k) \pm n & \text{falls } I = g_j \\ \mathbb{C}(I) & \text{sonst.} \end{cases}$$

Um die Berechnung eines RAM-Programmes zu definieren, müssen wir noch Endkonfigurationen kennzeichnen. Dafür führen wir eine neue Befehlskategorie ein:

$$< \underline{\text{OUT}} > ::= \underline{\text{PRINT}} < \underline{\text{OP}} > .$$

Der PRINT-Befehl gibt den Wert einer Speicherzelle oder eines Registers aus. Jede Konfiguration $C = (m, \mathbb{C})$, wobei der RAM-Befehl mit der Nummer m ein PRINT-Befehl ist, ist eine *Endkonfiguration*. Ohne Einschränkung sei $\rho(1)$ die einzige Speicherzelle, die ausgegeben werden darf.

Wir schreiben:

i) $C \overset{k}{\underset{P}{\rightarrow}} C'$, wenn C in k Schritten in C' übergeht. Induktiv gilt $C \overset{0}{\underset{P}{\rightarrow}} C'$ genau dann, wenn $C = C'$ und $C \overset{k}{\underset{P}{\rightarrow}} C'$ genau dann, wenn es eine Konfiguration C'' mit $C \overset{k-1}{\underset{P}{\rightarrow}} C''$ und $C'' \overset{1}{\underset{P}{\rightarrow}} C'$ gibt.

ii) $C \underset{P}{\rightarrow} C'$, falls es ein k mit $C \overset{k}{\underset{P}{\rightarrow}} C'$ gibt, d.h. $\underset{P}{\rightarrow}$ ist die transitive und reflexive Hülle von $\overset{1}{\underset{P}{\rightarrow}}$.

iii) $C \overset{\bullet}{\underset{P}{\rightarrow}} C'$, falls $C \underset{P}{\rightarrow} C'$ und C' Endkonfiguration ist.

iv) $C \underset{P}{\rightarrow} \infty$, falls es kein C' mit $C \overset{\bullet}{\underset{P}{\rightarrow}} C'$ gibt.

Eine *Anfangskonfiguration* C mit Eingabe $x_0, \ldots, x_k$ schreiben wir als $C(x_0, \ldots, x_k) = (0, \mathbb{C})$ mit

$$\mathbb{C}(I) = \begin{cases} x_i & \text{falls } I = r_i,\ 0 \leq i \leq k \\ 0 & \text{sonst} \end{cases} .$$

Nun können wir den Berechenbarkeitsbegriff für RAMs definieren.

Definition 3.33 Ein RAM-Programm P berechnet die Funktion $f_P : \underline{\text{NUM}}^{k+1} \rightarrow \underline{\text{NUM}}$, falls es für alle $x_0, \ldots, x_k$ eine Endkonfiguration $C' = (m, \mathbb{C}')$ mit $C(x_0, \ldots, x_k) \overset{\bullet}{\underset{P}{\rightarrow}} (m, \mathbb{C}')$ und $f_P(x_0, \ldots, x_k) = \mathbb{C}'(r_1)$ gibt.
Eine Funktion $f : \mathbb{N}_0^{k+1} \rightarrow \mathbb{N}_0$ heißt *RAM-programmierbar*, falls es ein RAM-Programm P mit $f_P = f$ gibt (dabei seien $\mathbb{N}_0$ und $\underline{\text{NUM}}$ unter Anwendung der semantischen Funktion ϕ miteinander identifiziert).

Zum Erkennen von Sprachen, können wir auch die charakteristische Funktion

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

der Sprache A betrachten. Dann können wir auch abgeschwächt fordern, daß eine Endkonfiguration nur für $x \in A$ erreicht werden muß (dabei kodieren wir $x \in A$ in geeigneter Weise als Element von $\underline{\text{NUM}}$).

3.4.2 Komplexitätsmaße der RAM

Da eine RAM in jeder Speicherzelle beliebig große Zahlen speichern kann, und damit in einer Operation auch Daten beliebiger Größe verarbeiten kann, benötigen wir ein geeignetes Komplexitätsmaß, welches die Größe der Operanden berücksichtigt. Diese Notwendigkeit tritt bei den Turing-Maschinen nicht auf, da in einer Zeiteinheit nur eine Bandzelle pro Band bearbeitet werden kann, deren Größe nur von der (festen) Kardinalität des Bandalphabetes abhängt. Daher definieren wir nun für die RAM zwei Kostenmaße.

- *Einheitskostenmaß*, d.h. jeder Speicherzugriff und jeder Befehl kostet eine Zeiteinheit
- *Logarithmisches Kostenmaß* (auch Bit-Kostenmaß genannt), d.h. die Kosten jedes Speicherzugriffs und jedes Befehls sind proportional zur Länge der Operanden (in Binärdarstellung).

Wir wollen diese Definition noch ein wenig konkretisieren. Sei $L(x)$ die Länge der Dualdarstellung von $x \in \mathbb{N}$, d.h.

$$L(x) := \begin{cases} 1 & \text{falls } x = 0 \\ \lfloor \log x \rfloor + 1 & \text{sonst.} \end{cases}$$

Die folgenden Tabellen zeigen die Kosten für die Bereitstellung von Operanden,

Operand	Einheitskostenmaß	Log. Kostenmaß
i	0	0
REG	0	0
ρ_i	1	$L(i)$
$\rho_{i+\gamma_j}$	1	$L(i) + L(\gamma_j)$

und für die Kosten der Befehle

Befehlsart	Einheitskostenmaß	Log. Kostenmaß
Transportbefehl	1	$1 + L(m)$
Sprungbefehl	1	$1 + L(k)$
Arithm. Befehl	1	$1 + L(m_1) + L(m_2)$
Indexbefehl	1	$1 + L(i) + L(\gamma_j)$

wobei m, m_1 und m_2 Operanden und k der Sprunglabel sind.

Mit Hilfe dieser verschiedenen Kostenmaße können wir einem RAM-Programm auch verschiedene Komplexitäten zuordnen.

Definition 3.34 Gegeben sei ein RAM-Programm P. Dann ist die *Einheitszeitkomplexität* von P

$$\tilde{T}_P(n) := \max_x \{\text{Anzahl der ausgeführten Operationen bei Eingabe } x \text{ mit } |x| = n\}$$

und die *Einheitsspeicherkomplexität* von P

$$\tilde{S}_P(n) := \max_x \{ \text{ maximale Anzahl der benutzten Register und Zellen während der Berechnung bei Eingabe } x \text{ mit } |x| = n\}.$$

Definition 3.35 Gegeben sei ein RAM-Programm P. Dann ist die *logarithmische Zeitkomplexität* von P

$$T_P(n) := \max_x \{ \text{Summe der logarithmischen Kosten der Operationen während der Berechnung bei Eingabe } x \text{ mit } |x| = n \}$$

und die *logarithmische Speicherkomplexität* von P

$$S_P(n) := \max_x \{ \text{maximale Summe der Längen der Binärdarstellungen der Zelleninhalte, Adressen und Register während der Berechnung bei Eingabe } x \text{ mit } |x| = n \}.$$

Zur Erläuterung dieser Definition sei ein kleines Beispielprogramm zur Berechnung von 2^n gedacht. Dabei steht die Eingabe in unärer Darstellung in der Speicherzelle ρ_0.

```
0  α ← 1
1  γ1 ← ρ0
2  IF γ1 = 0 THEN GOTO 6
3  α ← α * 2
4  γ1 ← γ1 − 1
5  GOTO 2
6  ρ1 ← α
```

Eine kleine Analyse zeigt, daß dieses Programm eine Einheitszeitkomplexität von $\tilde{T}_P(n) = O(n)$ jedoch eine logarithmische Zeitkomplexität von $T_P(n) = O(n^2)$ besitzt. Die Einheitsspeicherkomplexität ist 4, während die logarithmische Speicherkomplexität $O(n)$ beträgt.

Es sei bemerkt, daß die Verwendung der *Square and Multiply* Technik zu einem Algorithmus zur Berechnung von 2^n führt, der eine Einheitszeitkomplexität von $\tilde{T}_P(n) = O(\log n)$ und eine logarithmische Zeitkomplexität von $T_P(n) = O(n \log n)$ besitzt.

Nun führen wir Klassen von Funktionen ein, die durch RAM-Programme definiert werden.

Definition 3.36 Sei $T : \mathbb{N} \to \mathbb{N}$. Dann gilt:

$$\text{RAM-TIME}(T) := \{g : \mathbb{N}_0 \to \mathbb{N}_0 \mid \exists\, P \text{ mit } T_P = O(T) \text{ und } f_P = g\}.$$

Sei $S : \mathbb{N} \to \mathbb{N}$. Dann gilt:

$$\text{RAM-SPACE}(S) := \{g : \mathbb{N}_0 \to \mathbb{N}_0 \mid \exists\, P \text{ mit } S_P = O(S) \text{ und } f_P = g\}.$$

3.4.3 Vergleich zwischen TM und RAM

In diesem Abschnitt werden wir zeigen, daß das Modell der deterministischen Turing-Maschine polynomiell zum Modell der RAM verknüpft ist, d.h. daß

$$\text{RAM-TIME}(n^{O(1)}) = \text{DTIME}(n^{O(1)})$$

und

$$\text{RAM-SPACE}(n^{O(1)}) = \text{DSPACE}(n^{O(1)}).$$

Dabei modifizieren wir die Definitionen von DTIME und DSPACE wie folgt, damit sie mit den Klassen RAM-TIME und RAM-SPACE vergleichbar sind. Dazu müssen wir für ein Alphabet $\Gamma = \{a_1, \ldots, a_k\}$ eine geeignete Kodierung von Elementen aus Γ^* durch natürliche Zahlen finden. Es ist leicht zu überprüfen, daß die Abbildung $c_k : \Gamma^* \to \mathbb{N}_0$ mit $c_k(\Lambda) = 0$ und $c_k(a_{i_1}, \ldots, a_{i_n}) = \sum_{j=1}^{n} i_j k^{n-j}$ bijektiv ist. Daher definieren wir

$$\mathrm{DTIME}_c(T) := \{g : \mathbb{N}_0 \to \mathbb{N}_0 \mid \exists \text{Offl.-TM } M \text{ mit } c_k(f_M(c_k^{-1}(n))) = g(n) \text{ und } T_M = O(T)\}$$

und

$$\mathrm{DSPACE}_c(S) := \{g : \mathbb{N}_0 \to \mathbb{N}_0 \mid \exists \text{Offl.-TM } M \text{ mit } c_k(f_M(c_k^{-1}(n))) = g(n) \text{ und } S_M = O(S)\}$$

Jetzt lassen sich die Komplexitätsklassen vergleichen. Es gilt der folgende Satz über die Simulation von RAMs und TMs.

Satz 3.37 Sei P ein RAM-Programm mit logarithmischer Zeitkomplexität $T_P(n)$ und logarithmischer Speicherplatzkomplexität $S_P(n)$. Dann existiert eine TM M mit Zeitkomplexität $O(T_P(n)S_P(n)^2) = O(T_P^3(n))$ und Speicherkomplexität $O(S_P(n))$.

BEWEIS: Sei $\# \notin \Gamma$ ein Trennzeichen. Wir konstruieren eine 7-spurige TM M, d.h. eine TM M über dem Alphabet $(\Gamma \cup \{\#\})^7$ (die Zelleninhalte sind dann 7-Tupel). Spur 1, d.h. jeweils die erste Komponente der Tupel, simuliert den Speicher der RAM wie folgt. Es werden alle benutzten Adressen, kodiert mittels der Codefunktion c, aufgeschrieben, gefolgt von dem aktuellen Zelleninhalt. Dabei trennen wir Adresse und Zelleninhalt durch ein #-Zeichen und Adreß-Inhalt Paare durch zwei #-Zeichen, d.h. Spur 1 hat folgendes Aussehen:

$$\#\#\text{Adresse}\#\text{Inhalt}\#\#\text{Adresse}\#\text{Inhalt}\#\#\cdots.$$

Spur 2 enthält die Kodierungen des Inhalts des Akkumulators und Spuren 3–5 die Kodierungen der Inhalte der Register. Wir werden die Spuren 6 und 7 für Neben- und Adreßrechnungen verwenden. Um die oben genannte Behauptung zu beweisen, werden wir zeigen, daß jeder RAM-Befehl in $O(S_P(n)^2)$ Zeit ausführbar ist.

Als erstes beobachten wir, daß auf Spur 1 nie mehr Speicherzellen benutzt werden, als die Gesamtlänge aller Adressen und Zelleninhalte, die während der Berechnung von P verwendet werden. Da die anderen Spuren auch nur Inhalte von Registern speichern, ist die Länge jeder Spur und damit auch der Speicherverbrauch von M durch $O(S_P(n))$ beschränkt. Ein RAM-Befehl besteht nun aus:

i) dem Lesen einer Speicherzelle

ii) dem Ausführen einer arithmetischen Operation

iii) dem Schreiben in eine Speicherzelle.

Um den ersten und dritten Punkt zu bewerkstelligen, muß

a) die Adresse auf Spur 1 gesucht werden,

b) der Inhalt auf eine Hilfsspur kopiert, bzw. von einer Hilfsspur auf Spur 1 kopiert werden.

Schritt a) kann in Zeitaufwand $O(S_P^2(n))$ bewerkstelligt werden (der Kopf der Maschine fährt immer hin und zurück und vergleicht dabei Zelle für Zelle). Da die Länge des Inhaltes einer Speicherzelle durch $O(S_P(n))$ beschränkt ist, kann das Kopieren in Schritt b) ebenfalls in $O(S_P^2(n))$ Zeit geschehen (nach dem gleichen Schema). Falls auf Spur 1 kopiert wird, muß entweder eine neue Adresse mit Inhalt angelegt werden (falls diese noch nicht vorhanden), oder der alte Inhalt muß überschrieben werden. Da die Längen des alten und des neuen Inhaltes nicht übereinstimmen müssen, kann es erforderlich sein, eine Justierung vorzunehmen. Dies geschieht dadurch, daß der ganze Inhalt rechts von den abgrenzenden ## Zeichen der Spur 1 um Länge(neuer Inhalt) - Länge(alter Inhalt) nach rechts verschoben wird. Jetzt ist der zur Verfügung stehende Platz gerade von der Länge des neuen Inhaltes (es wird nichts überschrieben und auch kein Speicher verschwendet). Da aber höchstens $O(S_P(n))$ viele Zelleninhalte um nicht mehr als $O(S_P(n))$ viele Felder bewegt werden müssen, ist auch dies in $O(S_P^2(n))$ Zeit möglich.

Aufgabe 2., d.h. arithmetische Operationen wie Addition, Multiplikation und Division mit Operanden der Länge m können in Zeit $O(m^2)$ ausgeführt werden. Da $m = O(S_P(n))$ gilt, hat auch dieser Schritt eine Zeitkomplexität von $O(S_P^2(n))$.

Insgesamt ergibt sich also, daß jeder RAM-Befehl mit $O(S_P^2(n))$ vielen TM Schritten simulierbar ist. Da P höchstens $O(T_P(n))$ viele Schritte ausführt, ergibt sich die Behauptung. □

Die andere Richtung gilt auch:

Lemma 3.38 Sei M eine deterministische Turing-Maschine mit Zeitkomplexität $T_M(n) = O(T(n))$ und Speicherkomplexität $S_M(n) = O(S(n))$ mit $S(n) \geq n$. Dann gibt es ein RAM-Progamm P mit logarithmischer Zeitkomplexität $O(T \log S)$ und logarithmischer Speicherkomplexität von $O(S \log S)$.

BEWEIS: Der Beweis ergibt sich durch die direkte Simulation, in dem jede Speicherzelle der RAM einer Speicherzelle der TM entspricht. Da alle Größen bis auf die Adressen von konstanter Länge sind, und die Länge der Adressen durch $O(\log S(n))$ abgeschätzt werden können, ergibt sich sofort die Behauptung. □

Kapitel 4

Parallele und Randomisierte Maschinenmodelle

In diesem Kapitel werden wir parallele und randomisierte Berechnungsmodelle kennenlernen. Dabei werden wir Probleme kennenlernen, die nur mit Hilfe von Randomisierung effizient gelöst werden können. Zwar erweitern die hier eingeführten Berechnungsmodelle die Menge der berechenbaren Funktionen nicht, aber wir erreichen schnellere Algorithmen.

In Hinblick auf den Fortschritt in der Herstellung von Rechenanlagen werden wir zwei parallele Berechnungsmodelle kennenlernen, die sich später als annähernd identisch herausstellen. Dies sind als erstes die Parallele Random Access Maschine, die auf einer theoretischen Ebene moderne Supercomputer mit einer Vielzahl von Recheneinheiten widerspiegelt, und zweitens Schaltkreise, die als Elementarbausteine von Computern verstanden werden können.

4.1 Parallele Random Access Maschine

Zunächst werden wir unser Modell der RAM erweitern. Dies führt zum parallelen Berechnungsmodell der Parallelen RAM.

Definition 4.1 Eine Folge von Tupeln $R_0, R_1, \ldots$ mit $R_j = (\alpha^j, \gamma_1^j, \gamma_2^j, \gamma_3^j, LC^j)$ und eine Menge von Speicherzellen ρ_i, $i \in \mathbb{N}_0$, heißt Parallele Random Access Maschine (kurz: PRAM), falls

- jedes Tupel $(\alpha^j, \gamma_1^j, \gamma_2^j, \gamma_3^j, LC^j)$ mit der Menge der Speicherzellen eine RAM bildet; (es gelten die im vorigen Kapitel definierten Befehle.)
- die Menge der Speicherzellen für alle RAMs dieselbe ist, d.h. die RAMs keinen lokalen Speicher besitzen;
- jede RAM durch eine für sie spezifische *RAM-Nummer* RNUM ausgezeichnet ist (diese Nummer ist fest und für jede RAM unterschiedlich);
- alle RAMs dasselbe RAM-Programm bearbeiten.
- die RAMs synchronisiert sind, d. h. alle RAMs beginnen ihre Anweisung zeitgleich. Die nächste Anweisung wird erst gemeinsam von allen RAMs begonnen, nachdem alle aktiven RAMs die vorherige Anweisung ausgeführt haben.

Eine RAM als Teil einer PRAM unterscheidet sich also grundsätzlich von einer normalen RAM dadurch, daß sie keinen unendlich großen privaten Speicher hat. Daher können wir eine RAM, die Bestandteil einer PRAM ist, auch als Prozessor, also als reines Rechen- und Steuerwerk ansehen.

Der Verzicht auf einen lokalen Speicher ist jedoch keine Einschränkung. Der globale Speicher kann mittels einer geeigneten Adreßrechnung so aufgeteilt werden, daß er in unendlich große lokale Speicher für jede RAM und einem unendlich großen gemeinsamen Speicher zerfällt. Diese Speicheraufteilung kann z.B. mittels Methoden ähnlich zum CANTORschen Diagonalverfahren konstruiert werden.

Um den Programmfluß zu steuern, so daß nicht jede RAM dasselbe berechnet, werden wir unseren Befehlssatz erweitern.

Zu den Sprungbefehlen wird der folgende Befehl hinzugefügt:

$$< \underline{\text{SB}} >::= \underline{\text{IF}}\ \underline{\text{RNUM}} < \underline{\text{REL}} >< \underline{\text{NUM}} >\ \ \underline{\text{THEN}}\ \underline{\text{GOTO}}\ \ < \underline{\text{NUM}} > .$$

Außerdem wird eine weitere Klasse von Befehlen definiert, die für die Parallelität sorgt:

$$< \underline{\text{PB}} >::= \underline{\text{FORK}}.$$

Wir beschreiben kurz informal die Semantik der neuen Befehle:

- Erreicht eine RAM während der Programmausführung einen bedingten Sprung, der in Abhängigkeit von seiner Nummer steht, so verzweigt er in Abhängigkeit **seiner** (festen) Nummer. Dadurch kann der Kontrollfluß so beeinflußt werden, daß nicht alle RAMs das gleiche berechnen.
- Erreicht eine RAM einen FORK Befehl, so wird eine neue RAM (diejenige mit der kleinsten Nummer unter allen bisher inaktiven RAMs) aktiviert. Dabei wird der lokale Inhalt der RAM $(\alpha, \gamma_1, \ldots, \gamma_g, LC)$ in die neu aktivierte RAM kopiert.

Es treten aber noch mehr Probleme auf. Zunächst muß definiert werden, wie eine Berechnung gestartet wird und wann eine Berechnung beendet ist.

Definition 4.2 Die Berechnung einer PRAM wird dadurch gestartet, daß die RAM mit der Nummer 0 die Berechnung beginnt. Die Berechnung einer PRAM ist beendet, wenn die RAM mit der Nummer 0 stoppt. Man kann also die RAM R_0 als eine *Master*–RAM ansehen.

Definition 4.3 Die Laufzeit TP_P eines PRAM Programmes P ist die Laufzeit von R_0 im logarithmischen Kostenmaß.
SP_P bezeichne die maximale Anzahl von RAMs, die während der Berechnung aktiviert werden.

Bei der Definition der Laufzeit ist zu beachten, daß die RAMs synchronisiert sind, das heißt die Laufzeit pro Schritt ist die maximale Laufzeit die eine RAM benötigt hat, da alle anderen auf diese RAM warten müssen.

Die Semantik eines PRAM-Programmes läßt sich unmittelbar aus der Semantik von RAM-Programmen mittels einer (etwas modifizierten) Speicherfunktion $\mathbb{C}$ ausdrücken. Analog zu den

RAMs sagen wir, daß eine Funktion $f_P : \mathbb{N}_0^k \to \mathbb{N}_0$ PRAM programmierbar ist, wenn es ein PRAM-Programm P gibt, welches f_P berechnet.

Wir können nun Komplexitätsklassen einführen, indem wir wieder die Resourcen beschränken.

Definition 4.4 Sei $T : \mathbb{N} \to \mathbb{N}$. Dann gilt:

$$\text{PRAM-TIME}(T(n)) := \{g : \mathbb{N}_0 \to \mathbb{N}_0 \mid \exists \text{PRAM-Prog.} P \text{ mit } TP_P = O(T) \text{ und } f_P = g\}$$

Sei $S : \mathbb{N} \to \mathbb{N}$. Dann gilt:

$$\text{PRAM-SIZE}(S(n)) := \{g : \mathbb{N}_0 \to \mathbb{N}_0 \mid \exists \text{PRAM-Prog.} P \text{ mit } SP_P = O(S) \text{ und } f_P = g\}$$

Desweiteren betrachten wir die Klasse der simultan beschränkt berechenbaren Funktionen.

$$\begin{aligned}\text{PRAM-TIME-SIZE}(T(n), S(n)) := \quad & \{g : \mathbb{N}_0 \to \mathbb{N}_0 \mid \exists \text{PRAM-Prog.} P \text{ mit } TP_P = O(T), \\ & SP_P = O(S) \text{ und } f_P = g\}\end{aligned}$$

Statt PRAM-TIME-SIZE(T, S) schreiben wir der Kürze halber auch einfach PRAM(T, S).

Die Tatsache, daß viele RAMs gleichzeitig auf einen gemeinsamen Speicher zugreifen dürfen, kann zu *Konflikten* führen.

1. Zwei RAMs wollen gleichzeitig aus einer Zelle lesen.
2. Zwei RAMs wollen in die gleiche Zelle schreiben.
3. Eine RAM will aus einer Zelle lesen, in die ein andere RAM schreiben will.

Der Fall 3. kann leicht mit Hilfe der durch die Unterteilung eines Rechenschrittes in drei zeitlich getrennte Phasen gelöst werden.

1. Lesephase: Jede RAM liest die Daten, die für den nächsten elementaren Befehl nötig sind, aus dem gemeinsamen Speicher.
2. Berechnungsphase: Jede RAM führt eine Operation aus, die nur von den lokalen und gerade gelesenen Daten abhängig ist.
3. Schreibphase: Jede RAM schreibt das Ergebnis seiner Berechnung in den gemeinsamen Speicher.

Somit findet der Lese- und der Schreibvorgang zu verschiedenen Zeiten statt; dadurch sind Lese- und Schreibkonflikte ausgeschlossen. Die anderen beiden Fälle lassen sich nicht allgemein lösen. Daher definieren wir je nach Lösung dieses Problems geeignete Modelle.

Definition 4.5 Wir unterscheiden die folgenden Modelle von PRAMs.

- EREW-PRAM (Exclusive Read Exclusive Write): gemeinsames Lesen und Schreiben sind verboten.
- CREW-PRAM (Concurrent Read Exclusive Write): nur gemeinsames Lesen ist erlaubt, Schreibzugriffe auf dieselbe Speicherzelle sind verboten.

- CRCW-PRAM (Concurrent Read Concurrent Write): sowohl gemeinsames Lesen als auch gemeinsames Schreiben ist erlaubt.

Im Falle der CRCW-PRAM müssen wir noch folgende Fälle betrachten:

- Common CRCW-PRAM: gemeinsames Schreiben ist nur erlaubt, falls *alle* RAMs, die in eine Speicherzelle schreiben wollen, den **gleichen Wert** schreiben.
- Arbitrary CRCW-PRAM: beim gemeinsamen Schreiben setzt sich einer der RAMs **willkürlich** durch.
- Priority CRCW-PRAM: beim gemeinsamen Schreiben setzt sich die RAM mit der **kleinsten Nummer** durch.

Das Modell der PRAM erweitert die Menge der berechenbaren Funktionen nicht, da PRAMs leicht durch RAMs simuliert werden können. Dabei simuliert die RAM in $SP(n)$ Schritten je einen Schritt der $SP(n)$ aktiven RAMs der PRAM. Dadurch erhalten wir ein RAM-Programm mit Laufzeit $SP(n) \cdot TP(n)$.
Ein PRAM-Programm P ist somit *optimal*, wenn das "Prozessor-Zeit-Produkt" $SP(n) \cdot TP(n)$ in der gleichen Größenordnung wie die Zeit des besten sequentiellen RAM-Programms P' liegt, das dieselbe Funktion berechnet ($f_P = f_{P'}$).

Satz 4.6 Zwischen den durch die verschiedenen Typen von PRAMs definierten Sprachklassen gelten die folgenden Beziehungen im Einheitskostenmaß:

1. Priority-CRCW-PRAM$(T(n), S(n)) \subseteq$ EREW-PRAM$(T(n) \cdot \log S(n), S(n))$
2. Priority-CRCW-PRAM$(T(n), S(n)) \subseteq$ Common-CRCW-PRAM$(T(n), S(n)^2)$
3. EREW-PRAM$(T(n), S(n)) \subseteq$ Common-CRCW-PRAM$(T(n), S(n))$
   $\subseteq$ Priority-CRCW-PRAM$(T(n), S(n))$

BEWEIS:

zu 1 Wir wollen ein Priority-CRCW-PRAM Programm durch ein EREW-PRAM Programm simulieren. Dabei kann es sowohl beim Schreiben als auch beim Lesen zu Schwierigkeiten kommen.

- Schreiben: Wir nehmen an, daß die RAMs $R_1, \ldots, R_k$ in die Speicherzellen $\rho_1, \ldots, \rho_r$ schreiben wollen. Jeder RAM R_i ordnen wir ein Tupel (j_i, i, x_i) zu, wobei x_i der Wert ist, den die RAM R_i in die Speicherzelle ρ_{j_i} schreiben möchte. Nun sortieren wir die Folge der Tupel $\{(j_i, i, x_i)\}_{1 \le i \le k}$ nach der lexikographischen Ordnung. Verwenden wir hierzu den parallelen Mergesort von COLE ([Col86] und [GiRy88]), so benötigen wir hierzu nur k RAMs und $O(\log k)$ parallele Zeit. Mit Hilfe der sortierten Folge können wir jetzt leicht festlegen, welche RAM schreiben darf. Für alle Elemente der Folge gilt: Schreibe x_i nach ρ_{j_i},
  - falls das Tupel (j_i, i, x_i) das erste Tupel in der sortierten Folge ist oder
  - falls für den Vorgänger (j_l, l, x_l) des Tupels (j_i, i, x_i) $\quad j_l < j_i$ gilt,

  da in diesem Fall i die kleinste Nummer einer RAM ist, die nach ρ_{j_i} schreiben möchte.

- Lesen: Sei $S := \{(j_i, i) \mid R_i \text{ liest aus } \rho_{j_i}\}$. Sortiere S aufsteigend nach der lexikographischen Ordnung mit $|S|$ RAMs und in $O(\log |S|)$ paralleler Zeit. Bezeichne $S = \{s_1, \ldots, s_k\}$ jetzt die sortierte Folge. Falls für ein i $s_i = (j_i, i)$ und $s_{i+1} = (j_{i+1}, i+1)$ mit $j_i \neq j_{i+1}$ gilt, so setze

$$\max(j_i) = i \text{ und } \min(j_{i+1}) = i + 1.$$

  Für jedes j fertige $\max(j) - \min(j)$ Kopien von M_j an und verteile diese an die entsprechenden RAMs.

Damit ist eine Simulation innerhalb der behaupteten Schranken erreicht, da maximal $S(n)$ RAMs parallel lesen und schreiben. □

zu 2 Wir wollen nun eine Priority-CRCW-PRAM durch eine Common-CRCW-PRAM simulieren. Dazu müssen wir unser Programm so abändern, daß sichergestellt ist, daß beim parallelen Schreiben in eine Speicherzelle nur die gleichen Werte geschrieben werden. Dazu nehmen wir an, daß die RAMs $R_1 \ldots R_k$ in die Speicherzellen $\rho_1 \ldots \rho_r$ schreiben wollen. Jeder RAM R_i ordnen wir jetzt ein Tupel (j_i, i) zu, wobei ρ_{j_i} die Speicherzelle ist, in die R_i schreiben will. N_i seien Hilfsspeicherzellen, die mit 0 initialisiert werden. Nun führe mit den HilfsRAMs $R_{i',i}$ $1 \leq i' < i \leq k$ folgende Zuweisung parallel aus:

$$N_i := 1 \text{ falls } j_i = j_{i'} .$$

Damit enthält N_i genau dann eine 1, falls R_i nicht schreiben darf. Dies ist genau dann der Fall, wenn es eine RAM mit kleinerer Nummer gibt, die auf die gleiche Zelle schreiben will. Da immer nur der Wert 1 geschrieben wird, ist dies ein Common-CRCW Algorithmus. Nun überprüft jede RAM mit Hilfe von N_i, ob er schreibberechtigt ist und schreibt gegebenenfalls. □

zu 3 Die verschiedenen Berechnungsmodelle sind Verallgemeinerungen bzw. Spezialfälle voneinander. Daher ist die Inklusionskette klar. □

Der folgende wichtige Satz ermöglicht eine Trade-Off zwischen PRAM-TIME und PRAM-SIZE. Er stammt von BRENT.

Satz 4.7 Sei P ein PRAM-Programm mit paralleler Laufzeit t, das insgesamt m RAM-Operationen benötigt. Dann kann P so implementiert werden, daß es mit p parallel arbeitenden RAMs $O(t + m/p)$ parallele Zeit benötigt.

BEWEIS: Bezeiche $m(i)$ die Anzahl der parallel ausgeführten Operationen zum Zeitpunkt $1 \leq i \leq t$. Diese können mit p RAMs in $\lfloor \frac{m(i)}{p} + 1 \rfloor$ Zeit ausgeführt werden. Wenn wir nun noch über t summieren erhalten wir für die Gesamtzeit

$$T_P(n) = \sum_{i=1}^{t} (\lfloor \frac{m(i)}{p} + 1 \rfloor) \leq m/p + t.$$

□

4.2 Schaltkreise

In diesem Abschnitt untersuchen wir das Berechnungsmodell des Schaltkreises. Wir hatten in Abschnitt 1.3.1 logische Schaltungen kennengelernt. Die Bezeichung *Schaltkreis* ist einfach ein anderer Name für "logische Schaltung".

Zwischen Schaltkreisen und den bisher betrachteten Berechnungsmodellen tritt nun ein wesentlicher Unterschied auf. Während sowohl die Turing-Maschine als auch die RAM eine Funktion $f : \mathbb{B}^* \to \mathbb{B}$ berechnen (dabei interpretieren wir ein $x \in \mathbb{N}$ als die zugehörige Binärdarstellung), so berechnet ein Schaltkreis eine Funktion $g : \mathbb{B}^n \to \mathbb{B}$ für ein festes, durch den Schaltkreis definiertes n. Ein Schaltkreis ist ein fester Hardware Chip und berechnet somit **genau eine** Boolesche Funktion und **nicht eine Familie** von Booleschen Funktionen wie die RAM oder die Turing-Maschine. Wenn wir also Schaltkreise mit den anderen Berechnungsmodellen vergleichen wollen, so müssen wir *Familien* von Schaltkreisen betrachten. Dabei ist für jede Eingabegröße ein Schaltkreis in der *Schaltkreisfamilie*.

Definition 4.8 Sei $\{c_n\}_{n\in\mathbb{N}}$ eine Familie von Schaltkreisen. $\{c_n\}$ berechnet eine Funktion $f : \{0,1\}^+ \to \{0,1\}$ genau dann, wenn für jedes $n \in \mathbb{N}$ der Schaltkreis c_n die Funktion $f_{c_n} : \{0,1\}^n \to \{0,1\}$ mit $f_{c_n} = f_{|\{0,1\}^n}$ berechnet.

Eine Schaltkreisfamilie $\{c_n\}$ hat beschränkten Eingangsgrad (Fan-In), falls es eine Konstante k gibt, so daß für alle Schaltkreise c_n der Eingangsgrad aller Gatter in c_n durch k beschränkt ist; sonst hat die Familie einen unbeschränkten Eingangsgrad. Analog können wir auch von Familien mit beschränkten und unbeschränkten Ausgangsgrad (Fan-Out) reden. Falls wir es mit Schaltkreisfamilien mit beschränktem Eingangsgrad zu tun haben, können wir o.B.d.A. annehmen, daß der Eingangsgrad durch 2 beschränkt ist.

Es gilt der folgende Satz, der die Mächtigkeit des Berechnungsmodells des Schaltkreises darlegt.

Satz 4.9

1. Jede Funktion $g : \{0,1\}^+ \to \{0,1\}$ ist durch eine Schaltkreisfamilie mit unbeschränktem Eingangsgrad in Tiefe 3 berechenbar.
2. Jede Funktion $g : \{0,1\}^+ \to \{0,1\}$ ist durch eine Schaltkreisfamilie mit beschränktem Eingangsgrad in Tiefe $n + \lceil \log_2 n \rceil + 1$ berechenbar.
3. Jede Funktion $g : \{0,1\}^+ \to \{0,1\}$ ist durch eine Schaltkreisfamilie mit Größe $O(2^n/n)$ berechenbar.

BEWEIS:

1. Der Beweis von 1. folgt aus der Darstellung von Booleschen Funktionen in kanonischer Disjunktiver Normalform (siehe Abschnitt 1.3). Der Schaltkreis c_n, der $f = g_{|\{0,1\}^n}$ berechnet, wird wie folgt aus der Darstellung

$$f = \bigvee_{a \in f^{-1}(1)} \bigwedge_i x_i^{a_i}$$

   gewonnen. Der Ausgabeknoten ist ein $\vee$–Gatter mit maximal 2^n vielen Eingängen. Diese Eingänge sind mit den Ausgängen von $\wedge$ Gatter verbunden, deren Eingangsgrad wiederrum durch n beschränkt ist. Die Eingänge dieser $\wedge$–Gatter sind die Literale $x_i^{a_i}$, also entweder Variablen oder negierte Variablen. Die Negation von Variablen wird mit einem $\neg$–Gatter erreicht. Insgesamt hat der Schaltkreis also eine Tiefe von maximal 3.

2. Wir simulieren die in 1. gewonnene Schaltkreisfamilie durch eine Schaltkreisfamilie mit Eingangsgrad 2. Jedes Gatter mit einem Eingangsgrad $k > 2$ wird durch einen balancierten

binären Baum simuliert, dessen Knoten Gatter mit Eingangsgrad 2 vom selben Typ sind. Dieser Baum hat dann höchstens die Tiefe $\lceil \log_2 k \rceil$. Hieraus ergibt sich eine Schaltkreisfamilie, in der die Tiefe des n-ten Schaltkreises durch $n + \lceil \log_2 n \rceil + 1$ beschränkt ist.

3. Dieses Resultat stammt von LUPANOV aus dem Jahre 1959. Auf einen Beweis werden wir hier verzichten. □

Wir sehen, daß uneingeschränkte Schaltkreisfamilien in ihrer Berechnungskraft viel zu mächtig sind, da sie *alle* Funktion "berechnen" können. Insbesondere gibt es also auch Schaltkreisfamilien, die nicht Turing-berechenbare Funktionen berechnen können. Dies macht einen Vergleich von Schaltkreisfamilien mit den anderen Berechnungsmodellen sehr schwierig. Wir werden daher im folgenden nur eine Unterklasse von Schaltkreisfamilien betrachten.

4.2.1 Uniforme Schaltkreisfamilien

Wie wir oben gesehen haben, ist die Berechnungskraft allgemeiner Schaltkreisfamilien zu stark. Daher werden wir sogenannte *uniforme Schaltkreisfamilien* definieren.

Definition 4.10 Eine Schaltkreisfamilie $\{c_n\}_{n\in\mathbb{N}}$ heißt *uniform*, falls es eine TM gibt, die die Abbildung (Compiler Funktion)

$$c : 1^n \to \bar{c}_n$$

berechnet. Dabei ist $\bar{c}_n$ eine Beschreibung des Schaltkreises c_n (etwa die Adjazenzmatrix des Graphen von c_n). Eine uniforme Schaltkreisfamilie heißt auch *UC-Algorithmus*. Eine Schaltkreisfamilie heißt *log-space*-uniform, falls die Abbildung $c : 1^n \to \bar{c}_n$ mit logarithmisch beschränkten Speicher (auf einer Offline TM) berechenbar ist ($c \in \mathrm{DSPACE}(\log n)$).

Es ist offensichtlich, daß UC-Algorithmen nur Turing-berechenbare Funktionen berechnen können: eine Turing-Maschine, die f_A berechnet, errechnet bei einer n-stelligen Eingabe zuerst den Schaltplan $\bar{c}_n$ des n-ten Schaltkreises und simuliert diesen dann. Im folgenden verwenden wir die Begriffe uniform und UC-Algorithmus nur im Zusammenhang mit log-space Uniformität. Die von einem UC-Algorithmus A berechnete Funktion bezeichnen wir mit f_A.

Durch UC-Algorithmen sind wieder Komplexitätsklassen gegeben.

Definition 4.11 Seien $S, T : \mathbb{N} \to \mathbb{N}$ und haben alle UC-Algorithmen einen beschränkten Eingangsgrad. Dann sind

$$\begin{aligned}
\mathrm{SIZE}(S) &= \{g : \{0,1\}^+ \to \{0,1\} \mid \exists \text{UC-Alg. } A = \{c_n\} \text{ mit } g = f_A \\
&\qquad \text{und } \mathrm{Size}(c_n) = O(S(n))\} \\
\mathrm{DEPTH}(T) &= \{g : \{0,1\}^+ \to \{0,1\} \mid \exists \text{UC-Alg. } A = \{c_n\} \text{ mit } g = f_A \\
&\qquad \text{und } \mathrm{Depth}(c_n) = O(T(n))\} \\
\mathrm{DEPTH\text{-}SIZE}(T,S) &= \{g : \{0,1\}^+ \to \{0,1\} \mid \exists \text{UC-Alg. } A = \{c_n\} \text{ mit } g = f_A, \\
&\qquad \mathrm{Depth}(c_n) = O(T(n)) \text{ und } \mathrm{Size}(c_n) = O(S(n))\}.
\end{aligned}$$

Die Klasse $\mathcal{NC}$ enthält die Funktionen, die "superschnelle und hardware-effiziente" Schaltkreisimplementationen besitzen. Sie ist nach dem amerikanischen Mathematiker NICK PIPPENGER benannt (Nick's Class).

Definition 4.12 Sei $k \in \mathbb{N}_0$. Dann ist

$$\mathcal{NC}^k = \bigcup_{l \in \mathbb{N}} \text{DEPTH-SIZE}(\log^k(n), n^l)$$

die Klasse aller Funktionen, die durch uniforme Schaltkreisfamilien mit beschränktem Eingangsgrad in polynomieller Größe und $\log^k$ Tiefe berechnet werden können, und

$$\mathcal{NC} = \bigcup_{k \in \mathbb{N}_0} \mathcal{NC}^k$$

die Klasse aller Funktionen, die durch uniforme Schaltkreisfamilien mit beschränktem Eingangsgrad in polynomieller Größe und polylogarithmischer Tiefe berechnet werden können.

Es gelten analoge Definitionen für $\mathcal{AC}^k$ und $\mathcal{AC}$, wenn statt uniformer Schaltkreisfamilien mit beschränktem Eingangsgrad auch uniforme Schaltkreisfamile mit unbeschränktem Eingangsgrad betrachtet werden.

Nun wollen wir diese Komplexitätsklassen mit anderen Komplexitätsklassen vergleichen. Wir werden die folgenden Sätze nur angeben und die Beweise weglassen.

Satz 4.13 Mit den Abkürzungen X-PRAM^k für X-PRAM$(\log^k n, n^{O(1)})$ gilt:

$$\text{CRCW-PRAM}(T, S) \subseteq \text{DEPTH-SIZE}(T \log S, S^{O(1)})$$

und auch

$$\mathcal{NC}^k \subseteq \text{EREW-PRAM}^k \subseteq \text{CREW-PRAM}^k \subseteq \text{CRCW-PRAM}^k = \mathcal{AC}^k \subseteq \mathcal{NC}^{k+1}.$$

Korollar 4.14

$$\mathcal{NC} = \bigcup_{k \in \mathbb{N}_0} \text{CRCW-PRAM}(\log^k n, n^{O(1)}).$$

Wir können die parallelen Komplexitätsklassen auch mit den sequentiellen Komplexitätsklassen vergleichen. Der folgende Satz ist eine Folgerung aus einer Arbeit von Borodin ([Bor77]).

Satz 4.15 Sei $S : \mathbb{N} \to \mathbb{N}$, so daß $\log S$ eine Speicherschrittfunktion ist. Dann ist

$$\text{SIZE}(S^{O(1)}) = \text{DTIME}(S^{O(1)}) \cap \{g \mid g : \{0,1\}^+ \to \{0,1\}\}$$

Sei $T : \mathbb{N} \to \mathbb{N}$ eine Speicherschrittfunktion. Dann ist

$$\text{DEPTH}(T^{O(1)}) = \text{SPACE}(T^{O(1)}) \cap \{g \mid g : \{0,1\}^+ \to \{0,1\}\}$$

Damit gilt

Korollar 4.16

- Die Tiefe von Schaltkreisfamilien, die parallele Zeit von PRAMs und der Speicherverbrauch von TMs sind polynomiell miteinander verknüpt.
- Die Größe von Schaltkreisfamilien, die Anzahl der parallel arbeitenden RAMs einer PRAM und der Zeitverbrauch von TMs sind polynomiell miteinander verknüpt.

4.3 Randomisierte Schaltkreise

In diesem Abschnitt betrachten wir *probabilistische Schaltkreise*. Probabilistische Schaltkreise $c_{n,m}$ sind Schaltkreise, die zu den n Eingängen $x_1, \ldots, x_n$ noch zusätzlich $m = m(n)$ Eingänge $\rho_1, \ldots, \rho_m$ besitzen. Diese Eingänge werden mit Zufallswerten aus $\{0,1\}$ belegt. Wir ordnen dem Schaltkreis $c_{n,m}$ den Schaltkreis c_{n+m} zu, der dadurch entsteht, daß wir die Zufallseingänge als normale Eingänge interpretieren. Damit berechnet der Schaltkreis c_{n+m} eine Funktion von $n+m$ Booleschen Variablen.

Durch die Interpretation der Eingänge $\rho_1, \ldots, \rho_m$ als Zufallswerte wird der Wert $y = f_{c_{n,m}}(x)$ des Ausgabeknotens bei Eingabe $x = (x_1, \ldots, x_n)$ zu einer Zufallsvariablen über $\{0,1\}^m$. Für $a \in \{0,1\}$ gilt

$$\mathbf{P}(\{f_{c_{n,m}}(x) = a\}) = \frac{|\{\rho \mid f_{c_{n+m}}(x,\rho) = a\}|}{2^m}.$$

Damit berechnet ein probabilistischer Schaltkreis die partielle Funktion

$$\begin{array}{rcl} \varphi_{c_{n,m}} : \{0,1\}^n & \to & \{0,1\} \\ x & \mapsto & \begin{cases} 1 & \text{falls } \mathbf{P}(\{f_{c_{n,m}}(x) = 1\}) > 1/2 \\ 0 & \text{falls } \mathbf{P}(\{f_{c_{n,m}}(x) = 0\}) > 1/2 \\ \text{undefiniert} & \text{sonst} \end{cases} . \end{array}$$

Wir nennen einen probabilistischen Schaltkreis $c_{n,m}$ *randomisiert* oder *Monte-Carlo Schaltkreis*, falls $\varphi_{c_{n,m}}$ überall definiert ist, und für alle $x \in \{0,1\}^n$

$$\mathbf{P}(f_{c_{n,m}}(x) = 1) \notin (\frac{1}{4}, \frac{3}{4})$$

gilt.

Hieraus folgt, daß für $x \in \{0,1\}^n$ entweder $\mathbf{P}(f_{c_{n,m}}(x) = 1) \geq 3/4$ oder $\mathbf{P}(f_{c_{n,m}}(x) = 0) \geq 3/4$ gilt.

Für Funktionen, die durch randomisierte Schaltkreise berechnet werden können, gilt eine Art Komplementeigenschaft.

Lemma 4.17 Sei $c_{n,m}$ ein randomisierter Schaltkreis. Dann gibt es einen randomisierten Schaltkreis $c'_{n,m}$ mit

$$\varphi_{c'_{n,m}} = \neg\varphi_{c_{n,m}},$$

$\text{Size}(c'_{n,m}) = \text{Size}(c_{n,m}) + 1$ und $\text{Depth}(c'_{n,m}) = \text{Depth}(c_{n,m}) + 1$.

BEWEIS: Der Schaltkreis $c'_{n,m}$ entsteht durch Negation des Ausgabegatters y von $c_{n,m}$. Dann gilt nämlich $\mathbf{P}(f_{c'_{n,m}}(x) = 1) = 1 - \mathbf{P}(f_{c_{n,m}}(x) = 1) \notin (1/4, 3/4)$. □

Wir können randomisierte Schaltkreise auch durch deterministische Schaltkreise simulieren. Jedoch ist diese Simulation nichtuniform.

Lemma 4.18 Sei $c_{n,m}$ ein randomisierter Schaltkreis mit $\mathbf{P}(f_{c_{n,m}}(x) = 1) \notin [2^{-n-1}, 1 - 2^{-n-1}]$. Dann gibt es einen deterministischen Schaltkreis c_n mit $f_{c_{n,m}} = f_{c_n}$ mit $\text{Size}(c_n) \leq \text{Size}(c_{n,m}) + 3$ und $\text{Depth}(c_n) \leq \text{Depth}(c_{n,m}) + 2$.

BEWEIS: Wir zeigen, daß es ein festes $a \in \{0,1\}^m$ gibt, so daß $f_{c_{n+m}}(x,a) = \varphi_{c_{n,m}}(x)$ für alle $x \in \{0,1\}^n$ ist. Wenn wir dies gezeigt haben, folgt die Behauptung, da die Konstanten 0 und 1 in Tiefe 2 mit 3 Gattern konstruiert werden können ($0 = x_1 \wedge \neg x_1$, $1 = x_1 \vee \neg x_1$). Da für $x \in \{0,1\}^n$ nach Voraussetzung $\mathbf{P}(f_{c_{n,m}}(x) = 1) \notin [2^{-n-1}, 1-2^{-n-1}]$ ist, ist die Anzahl der $\rho \in \{0,1\}^m$ mit $\varphi_{c_{n,m}}(x) \neq f_{c_{n+m}}(x,\rho)$ kleiner als 2^{m-n}. Wenn wir dies über alle $x \in \{0,1\}^n$ summieren, erhalten wir, daß die Anzahl der $\rho \in \{0,1\}^m$, für die ein $x \in \{0,1\}^n$ mit $\varphi_{c_{n,m}}(x) \neq f_{c_{n+m}}(x,\rho)$ existiert, kleiner als $2^n 2^{m-n} = 2^m$ ist. Daher muß es mindestens ein $a \in \{0,1\}^m$ geben, für das

$$\forall\, x \in \{0,1\}^n \quad \varphi_{c_{n,m}}(x) = f_{c_{n+m}}(x,a)$$

gilt. Dieses a ist gerade unser gesuchtes a. □

Zu dem obigen Lemma muß jedoch gesagt werden, daß kein effizientes Verfahren existiert, ein solches a zu konstruieren (wir haben oben nur die Existenz nachgewiesen).

Analog zu den nicht randomisierten Schaltkreisen können wir auch hier uniforme Schaltkreise einführen. Die Definitionen gehen aus den entsprechenden Definitionen für nicht randomisierte Schaltkreise durch Hinzufügen des Zusatzes "randomisiert" hervor. Dadurch erhalten wir den Begriff des randomisierten UC-Algorithmus (RUC-Algorithmus) und die Komplexitätsklassen RSIZE(S) und RDEPTH(T). Entsprechend definieren sich auch die Klassen $\mathcal{RNC}^k$ und $\mathcal{RNC}$.

Die Beziehung randomisierter Komplexitätsklassen untereinander oder zu anderen Komplexitätsklassen ist bisher weitgehend unbekannt. Neben der Frage "$\mathcal{NC} = \mathcal{P}$?" treten nun die Frage "$\mathcal{RNC} = \mathcal{NC}$?" und die Beziehung zwischen $\mathcal{RNC}$ und $\mathcal{P}$ in den Vordergrund ("$\mathcal{P} \subset \mathcal{RNC}$?", "$\mathcal{P} \supset \mathcal{RNC}$?").

4.4 Randomisierte Testalgorithmen

In diesem Kapitel werden wir einige *Testalgorithmen* vorstellen, mit deren Hilfe algebraische Gleichheiten überprüft werden können. Diese Algorithmen sind randomisiert und haben den gleichen einfachen Aufbau: Falls eine algebraische Gleichung nicht allgemeingültig erfüllt ist, so gibt es nur *relativ wenige* Belegungen der Variablen, die diese Gleichung zufällig erfüllen. Ist diese algebraische Gleichung z.B. eine Polynomgleichung, so sind diese ungünstigen Belegungen der Variablen gerade die Nullstellen des Polynoms.

Wir müssen zunächst ein geeignetes Berechnungsmodell definieren.

Definition 4.19 Ein (randomisierter) *Black Box Schaltkreis* (Orakelschaltkreis) ist ein (randomisierter) Schaltkreis, der außer den logischen Gattern noch Orakelknoten (mit beliebigen Eingangsgraden) enthalten darf. Bei der Berechnung der Größe bzw. Tiefe des Schaltkreises geht ein Orakelknoten mit k Eingängen mit k in die Größe und $\log k$ in die Tiefe ein.

Analog zu den Klassen $\mathcal{NC}^k$ und $\mathcal{NC}$ definieren wir die Klassen $\blacksquare\mathcal{NC}^k$, $\blacksquare\mathcal{RNC}^k$,$\blacksquare\mathcal{NC}$ und $\blacksquare\mathcal{RNC}$.

4.4.1 Nulltest für multivariate Polynome

Unsere erste Anwendung ist ein Nulltest für multivariate Polynome. Die Schaltkreise enthalten also Orakelknoten (Black Box), die ein (festes) n-variates Polynom P mit ganzzahligen Koeffi-

zienten vom Grad D auswerten. Ziel ist es, zu bestimmen, ob dieses Polynom P identisch dem Nullpolynom ist. Wir bezeichnen dieses Problem im folgenden mit NTMP.

Es gilt der folgende Satz.

Satz 4.20

$$\text{NTMP} \notin \blacksquare\mathcal{NC} \quad (\notin \blacksquare\mathcal{P})$$

BEWEIS: Wir wollen den Beweis nur kurz anreißen. Die Koeffizienten eines multivariaten Polynoms $P = \sum c_i x_1^{i_1} \dots x_n^{i_n}$ sind durch ein lineares Gleichungssystem mit den Auswertungen von P an den Stellen $a^1, \dots, a^m$ gekoppelt (die dazugehörige $m \times (D+1)^n$ Matrix hängt natürlich von $a^1, \dots, a^m$ ab). Da ein n-variates Polynom p mit $\deg_{x_i} p = D$ bis zu $(D+1)^n$ viele Koeffizienten $c_i \neq 0$ hat, finden wir für jede Wahl von $m < (D+1)^n$ vielen Auswertungsstellen ein Polynom $q \neq 0$ (und dazu den Vektor von Koeffizienten), so daß q an allen Auswertungsstellen 0 ergibt. Daher sind wenigstens $(D+1)^n$ Auswertungen nötig und die Größe eines Black Box Schaltkreises für NTMP ist nicht polynomiell. □

Erstaunlicherweise gilt jedoch der

Satz 4.21

$$\text{NTMP} \in \blacksquare\mathcal{RNC}$$

Bevor wir einen Algorithmus angeben, werden wir ein Lemma beweisen, das Auskunft über die Anzahl von Nullstellen multivariater Polynome gibt. Es ist von SCHWARTZ in [Sch80] bewiesen worden.

Lemma 4.22 Sei $P = P_1(x_1, \dots, x_n) \in \mathbb{Z}[x_1, \dots, x_n]$ und $P \not\equiv 0$. Sei d_1 der Grad von x_1 in P_1 und P_2 das Koeffizientenpolynom von $x_1^{d_1}$:

$$P_1(x_1, \dots, x_n) = x_1^{d_1} \cdot P_2(x_2, \dots, x_n) + q_1(x_1, \dots, x_n).$$

Seien P_i und d_i für $2 \leq i \leq n$ entsprechend induktiv definiert. Dann hat P höchstens

$$|I_1 \times \cdots \times I_n| \left(\frac{d_1}{|I_1|} + \cdots + \frac{d_n}{|I_n|} \right)$$

Nullstellen in $I_1 \times \cdots \times I_n$.

BEWEIS: Wir führen den Beweis über Induktion nach der Anzahl der Variablen.

- Für den Fall eines univariaten Polynoms P gilt, daß die Anzahl der Nullstellen beschränkt ist durch
$$\deg(P) = d_1 = |I_1| \frac{d_1}{|I_1|}.$$
- Induktionsschritt:
P_1 hat die Darstellung
$$P_1(x_1, \dots, x_n) = x_1^{d_1} \cdot P_2(x_2, \dots, x_n) + q_1(x_1, \dots, x_n).$$
Nun gibt es zwei Möglichkeiten:

1. $(z_2, \ldots, z_n)$ ist Nullstelle von P_2. Dann kann eventuell $P(x_1, z_2, \ldots, z_n) = 0$ für alle $x_1 \in I_1$ gelten. Da die Anzahl der Nullstellen von P_2 nach Induktionsannahme durch

$$|I_2 \times \cdots \times I_n| \left(\frac{d_2}{|I_2|} + \ldots + \frac{d_n}{|I_n|} \right)$$

beschränkt ist, kann die Anzahl der Nullstellen von P für diesen Fall durch

$$|I_1| \cdot |I_2 \times \cdots \times I_n| \left(\frac{d_2}{|I_2|} + \ldots + \frac{d_n}{|I_n|} \right)$$

abgeschätzt werden.

2. $(z_2, \ldots, z_n)$ ist keine Nullstelle von P_2. Dann ist $P(x_1, z_2, \ldots, z_n)$ ein nicht verschwindendes Polynom in x_1 vom Grade d_1. Eine Schranke von

$$d_1 \cdot |I_2 \times \cdots \times I_n|.$$

für die Anzahl der Nullstellen von P in diesem Fall ergibt sich daher aus dem Induktionsanfang und der Tatsache, daß es höchstens $|I_2 \times \cdots \times I_n|$ viele $(z_2, \ldots, z_n)$ Vektoren gibt.

Damit kann die Gesamtanzahl von Nullstellen von P durch

$$\begin{aligned} |I_1| \cdot |I_2 \times \cdots \times I_n| \left(\frac{d_2}{|I_2|} + \ldots + \frac{d_n}{|I_n|} \right) + d_1 |I_2 \times \cdots \times I_n| \\ = |I_1 \times \cdots \times I_n| \left(\frac{d_1}{|I_1|} + \ldots + \frac{d_n}{|I_n|} \right) \end{aligned}$$

beschränkt werden. □

Korollar 4.23 Sei $P = P(x_1, \ldots, x_n) \in \mathbb{Z}[x_1, \ldots, x_n]$, $P \not\equiv 0$ und $I = I_1 = \cdots = I_n$ mit $|I| \geq c \cdot \deg(P)$. Dann ist die Anzahl der Nullstellen in I^n durch $|I|^n/c$ beschränkt.

BEWEIS: Wir wenden das Lemma 4.22 an. In diesem Fall gilt

$$\begin{aligned} |I|^n \cdot \sum_{j=1}^{n} d_j/|I| &= |I|^{n-1} \cdot (\sum_{j=1}^{n} d_j) \\ &\leq |I|^{n-1} \cdot \deg(P) \leq |I|^n/c, \end{aligned}$$

und somit ist die Anzahl der Nullstellen durch $|I|^n/c$ beschränkt. □

Nun können wir den folgenden Algorithmus angeben:

Eingabe:	Eine Black-Box für ein Polynom $P(x_1, \ldots, x_n)$ über $\mathbb{Z}$ und eine Schranke D für den Grad des Polynoms.
Schritt 1:	Konstruiere die Menge $E = \{1, 2, \ldots, 4D\}$
Schritt 2:	Wähle zufälliges $\bar{x} = (x_1, \ldots, x_n) \in E^n$
Ausgabe:	$\begin{cases} P \equiv 0 & \text{falls } P(\bar{x}) = 0 \\ P \not\equiv 0 & \text{sonst} \end{cases}$

Satz 4.24 Obiger Algorithmus ist ein korrekter Monte-Carlo Algorithmus.

BEWEIS: Mit Hilfe von Lemma 4.23 gilt, daß die Anzahl der Nullstellen des Polynoms P durch $|E|^n/4$ beschränkt ist. Für einen zufällig gewählten Vektor $\bar{x}$ aus E^n und $P \not\equiv 0$ gilt damit

$$\mathbf{P}\{P(\bar{x}) = 0\} \leq \frac{|E|^n/4}{|E|^n} = 1/4.$$

Damit ist die Ausgabe $P \equiv 0$ mit Wahrscheinlichkeit $\geq 3/4$ korrekt. Die Ausgabe $P \not\equiv 0$ ist immer korrekt, da für $P \equiv 0$ kein $\bar{x}$ mit $P(\bar{x}) \neq 0$ existiert. □

Eine direkte Übersetzung dieses Algorithmus in einen randomisierter Black Box UC–Algorithmus liefert Schaltkreise $c_{n,m}$ mit $m = n(\log D + 2)$ und Tiefe $O(\log(n \log D))$.

4.4.2 Test für Matrizenmultiplikation

Als nächstes wenden wir uns dem Problem der *Matrizenmultiplikation* zu. Dabei ist zu entscheiden, ob das Produkt zweier Matrizen eine dritte gegebene Matrix ergibt. Ein möglicher deterministischer Algorithmus zur Lösung dieses Problems besteht darin, die Matrizen A und B miteinander zu multiplizieren und das Ergebnis mit C zu vergleichen. Dies liefert auf naive Weise einen Algorithmus, der in DTIME(n^3) und in DEPTH-SIZE($\log n, n^3$) liegt. Mit Hilfe der neuesten Algorithmen im Bereich der Matrizenmultiplikation — aufbauend auf den Ideen von Strassen — ist dies sogar in DTIME($n^{2.376}$) und DEPTH-SIZE($\log n, n^{2.376}$) möglich. Eine untere Schranke für einen solchen Algorithmus wäre in jedem Fall $\Omega(n^2)$.
Der nachfolgende Algorithmus ist ein optimaler in $\mathcal{RNC}$ der Größe $O(n^2)$ für das Problem des Tests für Matrizenmultiplikation (TMM).

Eingabe:	Boolesche $n \times n$ Matrizen A,B und C.
Schritt 1:	Erzeuge zufällig 2 Vektoren $X_1, X_2 \in \{-1,1\}^n$: $X_1 = (x_{11}, \ldots, x_{1n}), \quad X_2 = (x_{21}, \ldots, x_{2n}).$
Schritt 2:	Berechne: $A(BX_1), A(BX_2), CX_1$ und CX_2.
Ausgabe:	$\begin{cases} 1 & \text{falls } ABX_1 = CX_1 \text{ und } ABX_2 = CX_2 \\ 0 & \text{sonst} \end{cases}$

Satz 4.25 Obiger Algorithmus arbeitet korrekt. Er liefert eine randomisierte Schaltkreisfamilie $c_{3n^2,m}$ der Tiefe $O(\log n)$ und der Größe $O(n^2)$ für TMM, d.h. TMM $\in \mathcal{RNC}^1$.

BEWEIS: Sei $D = (d_{ij}) = A \cdot B$ und $C = (c_{ij})$. Sei nun $\bar{x} = (x_1, \ldots, x_n)$ mit $x_i \in \{-1, 1\}$. Falls nun für $\bar{x}$

$$(A \cdot B - C) \cdot \bar{x} = 0$$

gilt, ist dies äquivalent zu

$$\forall i \quad (d_{i1} - c_{i1}, \ldots, d_{in} - c_{in}) \perp (x_1, \ldots, x_n). \tag{4.1}$$

Falls $A \cdot B \neq C$, könnten wir bei unserer zufälligen Wahl der Vektoren X_1 und X_2 schlechte Vektoren bekommen haben, für die trotzdem die Orthogonalitätsbeziehung (4.1) gilt. Es können nach Lemma 4.26 jedoch höchstens 2^{n-1} der 2^n vielen $\{-1, 1\}$ Vektoren orthogonal zu den n Vektoren $(d_{i1} - c_{i1}, \ldots, d_{in} - c_{in})$ sein, da wenigstens einer dieser Vektoren ungleich dem Nullvektor ist. Somit ist die Wahrscheinlichkeit, zweimal einen schlechten Vektor zu wählen, kleiner als $1/4$. □

Lemma 4.26 Sei $v \neq (0, \ldots, 0)$. Dann sind höchstens 2^{n-1} viele $\{-1, 1\}$ Vektoren der Länge n orthogonal zu v.

BEWEIS: Da $v = (v_1, \ldots, v_n)$ ungleich dem Nullvektor ist, ist wenigstens ein $v_{i_0} \neq 0$. Sei i_0 o.B.d.A gleich n, und seien $x = (x_1, \ldots, x_{n-1}, -1)$ und $y = (x_1, \ldots, x_{n-1}, 1)$. Somit gilt

$$< x, v >= \sum_{i=1}^{n} x_i \cdot v_i \neq < x, v > +2v_n = < y, v > .$$

Daher können nicht sowohl $< x, v >$ als auch $< y, v >$ gleich 0 sein. Die Zuordnung $x \mapsto y$ ordnet also einem zu v orthogonalen Vektor eineindeutig einen nicht orthogonalen Vektor zu. Somit können höchstens die Hälfte der 2^n Vektoren orthogonal zu v sein. □

Kapitel 5

Prädikatenlogik

In diesem Kapitel beschäftigen wir uns mit der sogenannten Prädikatenlogik. Sie handelt von der Untersuchung der Gültigkeit und Erfüllbarkeit von Formeln, die Variablen über beliebigen Mengen enthalten dürfen. Besonderes Gewicht werden wir dabei auf die Beziehung zwischen dem Wahrheitsgehalt verschiedener Formeln legen. Wir beschränken uns hier jedoch auf die Prädikatenlogik erster Stufe, in der nur Quantifizierungen über Variablen und nicht über Prädikate erlaubt sind. Eine Übersicht über die gesamte Prädikatenlogik und deren Anwendung (nicht nur der in diesem Kapitel dargestellten Sachverhalte) gibt das Buch von BERGMANN und NOLL [BeNo77]

5.1 Syntax der Prädikatenlogik

Zunächst möchten wir eine syntaktische Definition der Objekte der Prädikatenlogik geben. Die grundlegenden Objekte sind Funktionssymbole und Prädikatensymbole.

Definition 5.1 Mit $\underline{\mathrm{FS}}$ bezeichnen wir eine (höchstens) abzählbar unendliche Menge von *Funktionssymbolen*, mit $\underline{\mathrm{PS}}$ ebenfalls eine (höchstens) abzählbar unendliche Menge von *Prädikatensymbolen*. Die n-stelligen Funktionssymbole aus $\underline{\mathrm{FS}}$ bezeichnen wir mit $\underline{\mathrm{FS}}_n$, genauso die n-stelligen Prädikatensymbole mit $\underline{\mathrm{PS}}_n$. Es ist damit $\underline{\mathrm{FS}} = \bigcup_{n\geq 0} \underline{\mathrm{FS}}_n$ und $\underline{\mathrm{PS}} = \bigcup_{n\geq 0} \underline{\mathrm{PS}}_n$. Das Paar $\underline{B} = (\underline{\mathrm{FS}}, \underline{\mathrm{PS}})$ bezeichnen wir als *syntaktische Basis*.

Beispiel 5.2 Nullstellige Funktionssymbole sind z.B. Symbole für Zahlen oder andere Objekte, mehrstellige Funktionssymbole können z.B. die Addition oder Multiplikation sein, einstellige Funktionssymbole sind z.B. die succ Funktion $x \mapsto x + 1$. Prädikatensymbole sind z.B. $=$, $\leq$, oder auch < 0 als einstelliges Prädikatensymbol. Bezeichnen wir mit g und h zwei zweistellige Funktionssymbole (die wir als Addition bzw. Multiplikation interpretieren werden), so ist

$$\underline{B}_1 = (\{\underbrace{\ldots, \underline{-1}, \underline{0}, \underline{1}, \ldots}_{\underline{\mathrm{FS}}_0}, \underbrace{g, h}_{\underline{\mathrm{FS}}_2}\}, \{\underbrace{=}_{\underline{\mathrm{PS}}_2}\})$$

eine syntaktische Basis.

Die Menge der *Variablen* $\underline{\mathrm{VA}}$ ist eine nicht leere und höchstens abzählbare Menge. Z.B. können wir $\underline{\mathrm{VA}} = \{x, y, z, x_1, y_1, z_1 \ldots\}$ setzen. Übersetzt in die Schreibweise der BNF hieße dies $< \underline{\mathrm{VA}} > ::= x|y|z|x_1|y_1|z_1| \cdots$.

Zu einer syntaktischen Basis definieren wir nun die zugehörige Prädikatenlogik. Neben den Symbolen aus der syntaktischen Sprache besitzt jede Prädikatenlogik noch die *logischen Zeichen*

$$\neg \text{ (Negation)} \quad \vee \text{ (ODER/Disjunktion)} \quad \exists \text{ (Existenz-Quantor)}$$

zur Verknüpfung von Formelteilen und Quantifizierung von Variablen. Aus diesen logischen Zeichen können wir als Abkürzungen noch die folgenden Zeichen konstruieren: $\wedge$ (Konjunktion), $\rightarrow$ (Implikation), $\leftrightarrow$ (Äquivalenz) und $\forall$ (All-Quantor). Der All-Quantor ist definiert durch $\forall x\ A \equiv \neg\exists x\ \neg A$. Ferner seien "="das zweistellige Prädikatensymbol, welches als Gleichheit gedeutet wird, und die nullstelligen Prädikatensymbole $\underline{W}$ und $\underline{F}$ (oder $\underline{1}$ und $\underline{0}$) die syntaktischen Zeichen für Wahrheit oder Falschheit.

Definition 5.3 Die Menge der *Terme* bezüglich einer Basis $\underline{B}$ bezeichnen wir mit $\underline{\text{TE}}^{\underline{B}}$. Sie ist die kleinste Menge, für die gilt:

1. Jede Variable gehört zu dieser Menge.
2. Gehören $t_1, \ldots, t_n$ zu dieser Menge und ist g ein beliebiges n-stelliges Funktionssymbol $(n \geq 0)$, so gehört auch $gt_1 \ldots t_n$ $(= g(t_1, \ldots, t_n))$ zu dieser Menge.

Die Menge $\underline{\text{TE}}$ läßt sich durch die BNF auch schreiben als

$$< \underline{\text{TE}} > ::= < \underline{\text{VA}} > | < \underline{\text{FS}_n} > (< \underline{\text{TE}} >)^n |_{n=0}^{\infty}.$$

Als zweite syntaktische Klasse führen wir die prädikatenlogischen Formeln ein. Sie dienen als sprachlicher Ausdruck von Aussagen und werden als Wahrheitswerte interpretiert. Die einfachsten Formeln sind die nullstelligen Prädikatensymbole. Die nächst komplexeren Formeln sind frei von logischen Zeichen und werden als atomare Formeln bezeichnet.

Definition 5.4 Die Menge der *atomaren Formeln* bezüglich einer Basis $\underline{B}$ bezeichnen wir mit $\underline{\text{AF}}^{\underline{B}}$. Sie ist die kleinste Menge, die alle nullstelligen Prädikatensymbole enthält und für Terme $t_1, \ldots, t_n$ und n-stellige Prädikatensymbole p auch $pt_1 \ldots t_n$ $(= p(t_1, \ldots, t_n))$.

Enthält eine Formel ein logisches Zeichen, so ist sie keine atomare Formel. Genauer wird die Menge der Formeln durch die folgende Definition beschrieben.

Definition 5.5 Die Menge der *Formeln* bezüglich einer Basis $\underline{B}$ bezeichnen wir mit $\underline{\text{FO}}^{\underline{B}}$. Sie ist die kleinste Menge X für die gilt:

1. $\underline{\text{AF}}^{\underline{B}} \subset X$, d.h. jede atomare Formel ist eine Formel.
2. Aus $A, B \in X$ folgt $\neg A, A \vee B \in X$, d.h. die Negation und Disjunktion zweier Formeln ist wieder eine Formel.
3. Wenn $x \in \underline{\text{VA}}$ eine Variable und $A \in X$ eine Formel sind, so ist auch $\exists\, x\ A \in X$ eine Formel.

Wieder können wir dies durch eine BNF ausdrücken

$$\begin{aligned} < \underline{\text{FO}} > \quad ::= \quad & < \underline{\text{PS}_n} > (< \underline{\text{TE}} >)^n |_{n=0}^{\infty} \mid \neg < \underline{\text{FO}} > | < \underline{\text{FO}} > \vee < \underline{\text{FO}} > | \\ & \exists\ < \underline{\text{VA}} > < \underline{\text{FO}} > | \ (< \underline{\text{FO}} >) \end{aligned}$$

Wenn wir gewisse Eigenschaften für Terme oder Formeln beweisen wollen, werden wir dies über Induktion über den Aufbau der Terme bzw. Formeln tun. Man kann diese Induktion als eine Induktion über die Länge der Terme verstehen. Wir nennen diese Art von Induktion kurz auch *Netzinduktion.*

Bei der Induktion über den Aufbau der Terme ist die Induktionsverankerung der Beweis der Aussage E für alle Variablen $x \in \underline{\mathrm{VA}}$. Der Induktionsschritt besteht im Beweis der Aussage E für $g(t_1, \ldots, t_n)$ (dabei sei g ein n-stelliges Funktionssymbol) unter der Annahme, daß die Aussage E für $t_1, \ldots, t_n$ gilt.

Bei der Induktion über den Aufbau der Formeln ist die Induktionsverankerung der Beweis der Aussage E für alle atomaren Formeln $A \in \underline{\mathrm{AF}}^{\underline{B}}$. Der Induktionsschritt besteht in den Beweisen der Aussage E für $\neg A$, $A \vee B$ und $\exists\, x\, A$ unter der Annahme, daß die Aussage E für A und B gelten.

5.2 Semantik der Prädikatenlogik

Nach dem wir nun syntaktisch eine Menge von Formeln durch eine BNF definiert haben, versuchen wir jetzt den Formeln Wahrheitswerte zuzuordnen. Diese Zuordnung geschieht durch eine Interpretation der Funktions- und Prädikatensymbole durch Funktionen und Prädikate und der Angabe einer Grundmenge, auf der die den Funktions- und Prädikatensymbolen zugeordneten Funktionen oder Prädikate operieren.

Definition 5.6 Das Paar $\Sigma = (I, \omega)$ heißt eine *Struktur* für die Basis $\underline{B}$, wenn folgende Eigenschaften erfüllt sind:

1. I ist eine nichtleere Menge, der sogenannte *Individuenbereich.*
2. ω ist für jedes n-stellige Funktionssymbol g eine Abbildung $\omega(g) : I^n \to I$.
3. ω ist für jedes n-stellige Prädikatensymbol p eine Abbildung $\omega(p) : I^n \to \{W, F\}$.
4. Sind $\underline{W}$ und $\underline{F}$ nullstellige Prädikatensymbole, so ist $\omega(\underline{\mathrm{W}}) = W$ und $\omega(\underline{\mathrm{F}}) = F$. Das Prädikatensymbol "=" werten wir als Gleichheit.

Beispiel 5.7 Sei $\underline{B}_1 = (\{\ldots, \underline{-1}, \underline{0}, \underline{1}, \ldots, g, h\}, \{=\})$ eine Basis und $\Sigma_1 = (\mathbb{Z}, \omega)$ eine Struktur für diese Basis, dann interpretieren wir für alle $n, n' \in \mathbb{Z}$ die Symbole wie folgt: $\omega(\underline{n}) = n$ für alle $\underline{n} \in \underline{\mathrm{FS}}_0$, $\omega(g)(\underline{n}, \underline{n'}) = n + n'$ und $\omega(h)(\underline{n}, \underline{n'}) = nn'$.

Variablen haben keine feste Beziehung zu einem bestimmten Individuum, sondern können ein beliebiges Individuum aus dem Individuenbereich meinen.

Definition 5.8 Eine Abbildung $f : \underline{\mathrm{VA}} \to I$, die jeder Variablen ein Individuum zuordnet, heißt ein *Zustand der Variablen.*

Um eine Zustandsänderung einer Variablen ausdrücken zu können, benutzten wir die folgende Schreibweise.

Definition 5.9 Seien $f : \underline{\text{VA}} \to I$ ein Zustand, $x \in \underline{VA}$ eine Variable und $\xi \in I$ ein Individuum. Dann sei der Zustand $f\left\langle \begin{smallmatrix} x \\ \xi \end{smallmatrix} \right\rangle : \underline{\text{VA}} \to I$, der aus f dadurch hervorgeht, daß in f der Wert von x in ξ abgeändert wird, wie folgt definiert:

$$\left(f\left\langle \begin{smallmatrix} x \\ \xi \end{smallmatrix} \right\rangle\right)(y) = (\texttt{IF } x = y \texttt{ THEN } \xi \texttt{ ELSE } f(y)) \quad (\forall\, y \in \underline{\text{VA}})$$

Wir werden einige Eigenschaften von Zustandsänderungen aufzählen.

Lemma 5.10

- Wenn man den Wert einer Variablen in den schon dort vorhandenen Wert abändert, so ändert sich der Zustand nicht, d.h.

$$f\left\langle \begin{smallmatrix} x \\ f(x) \end{smallmatrix} \right\rangle = f.$$

- Wird zuerst eine Variable x in ξ und dann in η abgeändert, so bleibt der zweite Wert η erhalten und ξ wird überschrieben, d.h.

$$\forall\, \xi, \eta \in I \quad \left(f\left\langle \begin{smallmatrix} x \\ \xi \end{smallmatrix} \right\rangle\right)\left\langle \begin{smallmatrix} x \\ \eta \end{smallmatrix} \right\rangle = f\left\langle \begin{smallmatrix} x \\ \eta \end{smallmatrix} \right\rangle.$$

- Es spielt keine Rolle in welcher Reihenfolge Abänderungen von zwei verschiedenen ungleichen Variablen vorgenommen werden, d.h.

$$\forall\, x, y \in \underline{\text{VA}}, x \neq y,\ \forall\, \xi, \eta \in I \quad \left(f\left\langle \begin{smallmatrix} x \\ \xi \end{smallmatrix} \right\rangle\right)\left\langle \begin{smallmatrix} y \\ \eta \end{smallmatrix} \right\rangle = \left(f\left\langle \begin{smallmatrix} y \\ \eta \end{smallmatrix} \right\rangle\right)\left\langle \begin{smallmatrix} x \\ \xi \end{smallmatrix} \right\rangle.$$

Die Auswertung von Funktions- bzw. Prädikatensymbolen, in denen keine Variablen enthalten sind, erfolgt durch die oben eingeführte Funktion ω. Wir werden diese semantische Auswertung nun auf beliebige Funktions- bzw. Prädikatensymbole erweitern. Da Variablen auftreten, hängt der Wert sicherlich vom Zustand der Variablen ab. Genauer definieren wir die Funktionen $\underline{\text{wert}}_\Sigma$ und $\underline{\text{Wert}}_\Sigma$ wie folgt.

Definition 5.11 Wir definieren eine Abbildung

$$\underline{\text{wert}}_\Sigma : \underline{\text{TE}} \times (\underline{\text{VA}} \to I) \to I$$

induktiv über den Aufbau der Terme: für $x \in \underline{\text{VA}}$ gilt $\underline{\text{wert}}_\Sigma(x, f) = f(x)$ und für alle Terme $t_i \in \underline{\text{TE}}$ und für alle n-stelligen Funktionssymbole $g \in \underline{\text{FS}}_n$ gilt

$$\underline{\text{wert}}_\Sigma(g(t_1, \ldots, t_n), f) = \omega(g)(\underline{\text{wert}}_\Sigma(t_1, f), \ldots, \underline{\text{wert}}_\Sigma(t_n, f)).$$

Analog definieren wir eine Abbildung

$$\underline{\text{Wert}}_\Sigma : \underline{\text{FO}} \times (\underline{\text{VA}} \to I) \to \{W, F\}$$

induktiv über den Aufbau der Formeln: für atomare Formeln $p(t_1, \ldots, t_n) \in \underline{\text{AF}}$ gilt

$$\underline{\text{Wert}}_\Sigma(p(t_1, \ldots, t_n), f) = \omega(p)(\underline{\text{wert}}_\Sigma(t_1, f), \ldots, \underline{\text{wert}}_\Sigma(t_n, f)),$$

für beliebige Formeln A, B gelten

$$\begin{aligned} \underline{\text{Wert}}_\Sigma(\neg A, f) &= \neg(\underline{\text{Wert}}_\Sigma(A, f)) \\ \underline{\text{Wert}}_\Sigma(A \vee B, f) &= (\underline{\text{Wert}}_\Sigma(A, f)) \vee (\underline{\text{Wert}}_\Sigma(B, f)) \end{aligned}$$

und

$$\underline{\text{Wert}}_\Sigma(\exists\, x\ A, f) = W \Leftrightarrow \text{ Es gibt ein } \xi \in I, \text{ so daß } \underline{\text{Wert}}_\Sigma\left(A, f\left\langle \begin{smallmatrix} x \\ \xi \end{smallmatrix} \right\rangle\right) = W.$$

Die $\underline{\text{Wert}}_\Sigma$ Funktion ist auch für Formeln, in denen andere logische Zeichen vorkommen, auf kanonische Weise definiert.

Beispiel 5.12 Gegeben sei dieselbe Basis und Struktur wie in Beispiel 5.7. Wir wollen nun die Formel "$\forall\, x\ \exists\, y\ (g(x,y) = \underline{0})$" auswerten. Es gilt

$$\begin{aligned}\underline{\text{Wert}}_\Sigma(\forall\, x\ \exists\, y\ (g(x,y) = \underline{0}), f) = W &\Leftrightarrow \forall \xi \in \mathbb{Z}\ \left[\underline{\text{Wert}}_\Sigma\left(\exists y\ (g(x,y) = \underline{0}), f\left\langle \begin{smallmatrix} x \\ \xi \end{smallmatrix} \right\rangle\right) = W\right] \\ &\Leftrightarrow \forall \xi \in \mathbb{Z}\ \exists \eta \in \mathbb{Z}\left[\underline{\text{Wert}}_\Sigma\left(g(x,y) = \underline{0}, f\left\langle \begin{smallmatrix} x \\ \xi \end{smallmatrix} \right\rangle\left\langle \begin{smallmatrix} y \\ \eta \end{smallmatrix} \right\rangle\right) = W\right]\end{aligned}$$

Als Zwischenrechung werten wir nun die Formel $g(x,y) = \underline{0}$ in dem durch die Quantifizierung geändertern Zustand $f\left\langle \begin{smallmatrix} x \\ \xi \end{smallmatrix} \right\rangle\left\langle \begin{smallmatrix} y \\ \eta \end{smallmatrix} \right\rangle$ aus:

$$\begin{aligned}&\underline{\text{Wert}}_\Sigma\left(g(x,y) = \underline{0}, f\left\langle \begin{smallmatrix} x \\ \xi \end{smallmatrix} \right\rangle\left\langle \begin{smallmatrix} y \\ \eta \end{smallmatrix} \right\rangle\right) = W \\ \Leftrightarrow\ &\underline{\text{wert}}_\Sigma\left(g(x,y), f\left\langle \begin{smallmatrix} x \\ \xi \end{smallmatrix} \right\rangle\left\langle \begin{smallmatrix} y \\ \eta \end{smallmatrix} \right\rangle\right) = \underline{\text{wert}}_\Sigma\left(\underline{0}, f\left\langle \begin{smallmatrix} x \\ \xi \end{smallmatrix} \right\rangle\left\langle \begin{smallmatrix} y \\ \eta \end{smallmatrix} \right\rangle\right) \\ \Leftrightarrow\ &\omega(g)\left(f\left\langle \begin{smallmatrix} x \\ \xi \end{smallmatrix} \right\rangle\left\langle \begin{smallmatrix} y \\ \eta \end{smallmatrix} \right\rangle(x), f\left\langle \begin{smallmatrix} x \\ \xi \end{smallmatrix} \right\rangle\left\langle \begin{smallmatrix} y \\ \eta \end{smallmatrix} \right\rangle(y)\right) = \omega(\underline{0}) \\ \Leftrightarrow\ &\omega(g)\left(f\left\langle \begin{smallmatrix} x \\ \xi \end{smallmatrix} \right\rangle(x), \eta\right) = 0 \\ \Leftrightarrow\ &\omega(g)(\xi, \eta) = 0 \\ \Leftrightarrow\ &\xi + \eta = 0\end{aligned}$$

Setzen wir dies ein, so erhalten wir daß die Interpretation der Formel $\forall\, x\ \exists\, y\ (g(x,y) = \underline{0})$ durch

$$\forall \xi \in \mathbb{Z}\ \exists \eta \in \mathbb{Z} \quad \xi + \eta = 0$$

gegeben ist. Da diese gültig ist (zu jeder ganzen Zahl ξ gibt es ein additives Inverses), folgt auch die Gültigkeit der Formel $\forall\, x\ \exists\, y\ (g(x,y) = \underline{0})$ in der Struktur Σ_1 bei jedem Zustand f.

5.3 Erfüllbarkeit und Allgemeingültigkeit

Mit Hilfe von semantischen Objekten, den Strukturen $\Sigma = (I, \omega)$, haben wir den syntaktischen Klassen $\underline{\text{TE}}$ und $\underline{\text{FO}}$ Bedeutungen zuordnen können. Wir möchten uns jetzt jedoch die Frage stellen, wann eine Formel erfüllbar oder allgemeingültig ist.

Die folgenden Definitionen entsprechen der intuitiven Vorstellung der Erfüllbarkeit und Allgemeingültigkeit einer Formel.

Definition 5.13 Sei $\underline{B}$ eine Basis.

- Eine Formel $A \in \underline{\text{FO}}^{\underline{B}}$ ist *erfüllbar in der Struktur* $\Sigma = (I, \omega)$, wenn es einen Zustand $f : \underline{\text{VA}} \to I$ gibt, so daß $\underline{\text{Wert}}_\Sigma(A, f) = W$ gilt, d.h. für wenigstens eine Variablenbelegung ist die Formel A erfüllt. Die Menge aller in der Struktur Σ erfüllbaren Formeln bezeichnen wir $\underline{\text{SAT}}_\Sigma$.
- Eine Formel $A \in \underline{\text{FO}}^{\underline{B}}$ heißt *erfüllbar*, wenn es eine Struktur $\Sigma = (I, \omega)$ für $\underline{B}$ gibt, in der A erfüllbar ist. Die Menge aller erfüllbaren Formeln bezeichnen wir $\underline{\text{SAT}}$.

- Eine Formel $A \in \underline{\mathrm{FO}}^{\underline{B}}$ ist *gültig in der Struktur* $\Sigma = (I, \omega)$, wenn $\underline{\mathrm{Wert}}_\Sigma(A, f) = W$ für alle Zustände $f : \underline{\mathrm{VA}} \to I$ gilt, d.h. alle Variablenbelegungen die Formel A erfüllen. Die Menge aller in der Struktur Σ gültigen Formeln bezeichnen wir $\underline{\mathrm{VAL}}_\Sigma$.
- Eine Formel $A \in \underline{\mathrm{FO}}^{\underline{B}}$ heißt *allgemeingültig*, wenn A in allen Strukturen Σ für $\underline{B}$ gültig ist. Die Menge aller allgemeingültigen Formeln bezeichnen wir $\underline{\mathrm{VAL}}$.

Allgemeingültige Formeln heißen auch prädikatenlogische Gesetze, Identitäten, Tautologien oder (prädikatenlogisch) wahre Formeln. Ein Beispiel ist die Formel $\neg B \vee B$, wobei B eine beliebige Formel ist.

Definition 5.14 Sei $\underline{B}$ eine Basis, dann sei die Ähnlichkeitsklasse von Strukturen für $\underline{B}$ definiert durch:

$$\underline{\mathrm{ST}}^{\underline{B}} = \{\Sigma \mid \Sigma \text{ ist eine Struktur für die Basis } \underline{B}\}.$$

Aus der Definition der Klassen $\underline{\mathrm{SAT}}_\Sigma$, $\underline{\mathrm{SAT}}$, $\underline{\mathrm{VAL}}_\Sigma$ und $\underline{\mathrm{VAL}}$ ergibt sich sofort, daß

$$\begin{aligned} \underline{\mathrm{VAL}} &= \bigcap_{\Sigma \in \mathrm{ST}^{\underline{B}}} \underline{\mathrm{VAL}}_\Sigma \\ \underline{\mathrm{SAT}} &= \bigcup_{\Sigma \in \mathrm{ST}^{\underline{B}}} \underline{\mathrm{SAT}}_\Sigma. \end{aligned}$$

Der Beweis des folgenden Lemmas ergibt sich durch elementare Umformungen.

Lemma 5.15 Es gilt:

1. $A \in \underline{\mathrm{SAT}}_\Sigma \quad \Leftrightarrow \quad \neg A \notin \underline{\mathrm{VAL}}_\Sigma$.
2. $A \in \underline{\mathrm{VAL}}_\Sigma \quad \Leftrightarrow \quad \neg A \notin \underline{\mathrm{SAT}}_\Sigma$.
3. $A \in \underline{\mathrm{SAT}} \quad \Leftrightarrow \quad \neg A \notin \underline{\mathrm{VAL}}$.
4. $A \in \underline{\mathrm{VAL}} \quad \Leftrightarrow \quad \neg A \notin \underline{\mathrm{SAT}}$.

BEWEIS: Wir werden die erste Aussage beweisen:

$$\begin{aligned} A \in \underline{\mathrm{SAT}}_\Sigma \quad &\Leftrightarrow \quad \exists\, f : \underline{\mathrm{VA}} \to I : \ \underline{\mathrm{Wert}}_\Sigma(A, f) = W \\ &\Leftrightarrow \quad \exists\, f : \underline{\mathrm{VA}} \to I : \ \underline{\mathrm{Wert}}_\Sigma(\neg A, f) = F \ \Leftrightarrow \ \neg A \notin \underline{\mathrm{VAL}}_\Sigma. \end{aligned}$$

□

Im folgenden sei $\underline{B}$ eine beliebige, aber feste Basis. Wir werden daher den Index $\underline{B}$ nicht mehr mitführen.

Definition 5.16

- Zwei Formeln $A, B \in \underline{\mathrm{FO}}$ sind *(logisch) äquivalent*, wenn $(A \leftrightarrow B) \in \underline{\mathrm{VAL}}$ gilt.
- Zwei Formeln $A, B \in \underline{\mathrm{FO}}$ sind *erfüllbarkeitsgleich*, wenn $A \in \underline{\mathrm{SAT}} \ \Leftrightarrow \ B \in \underline{\mathrm{SAT}}$ gilt.

- Eine Formelmenge $X \subseteq \underline{\text{FO}}$ heißt *simultan erfüllbar in der Struktur* Σ, wenn es einen Zustand $f : \underline{\text{VA}} \to I$ gibt, so daß für alle Formeln $A \in X$ $\underline{\text{Wert}}_\Sigma(A, f) = W$ gilt. Die Menge aller in der Struktur Σ simultan erfüllbaren Formelmengen bezeichnen wir mit $\underline{\text{SAT}}^{\text{sim}}_\Sigma$.
- Eine Formelmenge $X \subseteq \underline{\text{FO}}$ heißt *simultan erfüllbar*, wenn es eine Struktur Σ gibt, in der X simultan erfüllbar ist. Die Menge aller simultan erfüllbaren Formelmengen bezeichnen wir mit $\underline{\text{SAT}}^{\text{sim}}$[208$z$.

Ein Beispiel einer nicht simultan erfüllbaren Menge ist die Formelmenge $X = \{p(x), \neg p(x)\}$.

In manchen Fällen können wir einer Formel ansehen, ob sie allgemeingültig ist, oder ob eine Formelmenge simultan erfüllbar ist. Dies geschieht allein aus dem aussgagenlogischen Aufbau der Formeln heraus, ohne die Quantifizierung von Variablen zu beachten. Daher nennt man diesen Teil der Prädikatenlogik auch die *Aussagenlogik im Rahmen der Prädikatenlogik* (engl. propositional calculus). Das folgende Lemma macht dies deutlich.

Lemma 5.17 Für alle Formeln A ist die Formel $\neg A \vee A$ allgemeingültig, d.h.

$$\forall\, A \in \underline{\text{FO}} \quad (\neg A \vee A) \in \underline{\text{VAL}}.$$

BEWEIS: Wir haben zu zeigen, daß $\neg A \vee A$ in allen Strukturen $\underline{\Sigma}$ gültig ist, d.h. wir müssen folgendes zeigen:

$$\forall\, \Sigma \in \underline{\text{ST}}^B\ \forall\, f : \underline{\text{VA}} \to I : \quad \underline{\text{Wert}}_\Sigma(\neg A \vee A, f) = W.$$

Seien also Σ eine beliebige Struktur und $f : \underline{\text{VA}} \to I$ ein beliebiger Zustand. Aufgrund der Definition der $\underline{\text{Wert}}$-Funktion ist $\underline{\text{Wert}}_\Sigma(A, f) = \neg \underline{\text{Wert}}_\Sigma(\neg A, f)$. Damit ist aber

$$\underline{\text{Wert}}_\Sigma(\neg A \vee A, f) = W \vee F = W$$

. □

Im Beweis haben wir nur die Eigenschaften von $\neg$ und $\vee$ benutzt. Dabei zerteilten $\neg$ und $\vee$ die Formel in Teilformeln. Eine Zerteilung einer prädikatenlogischen Formel in Teilformeln, die mittels $\vee$, $\neg$ und anderer aussagenlogischer Symbole verknüpft sind, nennen wir eine *aussagenlogische Zerlegung* der Formel, den so erkennbaren Aufbau auch *aussagenlogischen Aufbau.*

Beispielsweise hat die Formel $\forall\, x A \ \vee\ \exists\, x \neg A$ den aussagenlogischen Aufbau aus den Teilformen $\forall\, x A$ und $\exists\, x \neg A$. Die Formel $\forall\, x (A \vee B)$ kann jedoch bzgl. des aussagenlogischen Aufbaus nicht weiter zerlegt werden.

Wir werden nun die semantische Definitionen weiterer logischer Symbole angeben.

- Es gilt $A \wedge B \in \underline{\text{VAL}}_\Sigma$ genau dann, wenn $\underline{\text{Wert}}_\Sigma(A, f) = \underline{\text{Wert}}_\Sigma(B, f) = W$ für alle $f : \underline{\text{VA}} \to I$ gilt.
- Es gilt $A \to B \in \underline{\text{VAL}}_\Sigma$ genau dann, wenn $(\underline{\text{Wert}}_\Sigma(A, f) = F) \vee (\underline{\text{Wert}}_\Sigma(B, f) = W)$ für alle $f : \underline{\text{VA}} \to I$ gilt.
- Es gilt $A \leftrightarrow B \in \underline{\text{VAL}}_\Sigma$ genau dann, wenn $\underline{\text{Wert}}_\Sigma(A, f) = \underline{\text{Wert}}_\Sigma(B, f)$ für alle $f : \underline{\text{VA}} \to I$ gilt.
- Es gilt $A \vee B \in \underline{\text{VAL}}_\Sigma$ genau dann, wenn $(\underline{\text{Wert}}_\Sigma(A, f) = W) \vee (\underline{\text{Wert}}_\Sigma(B, f) = W)$ für alle $f : \underline{\text{VA}} \to I$ gilt.

Aus diesen Definitionen ergibt sich sofort:

Lemma 5.18 Für alle $A, B, C \in \underline{\mathrm{FO}}$ sind $(A \leftrightarrow A) \in \underline{\mathrm{VAL}}$, $(A \leftrightarrow \neg(\neg A)) \in \underline{\mathrm{VAL}}$, $(A \leftrightarrow (A * A)) \in \underline{\mathrm{VAL}}$, $((A * B) \leftrightarrow (B * A)) \in \underline{\mathrm{VAL}}$ und $(((A * B) * C) \leftrightarrow (A * (B * C))) \in \underline{\mathrm{VAL}}$, wobei $*$ ein beliebiges logisches Zeichen aus $\{\vee, \wedge, \leftrightarrow\}$ ist.

Besonders wichtig ist nun noch die Abtrennungsregel.

Lemma 5.19 Für alle Strukturen $\Sigma \in \underline{\mathrm{ST}}^B$ und alle Formeln A, B gilt:

$$(A \to B) \in \underline{\mathrm{VAL}}_\Sigma \;\Rightarrow\; (A \in \underline{\mathrm{VAL}}_\Sigma \;\Rightarrow\; B \in \underline{\mathrm{VAL}}_\Sigma).$$

5.4 Quantorengesetze und Substitution

5.4.1 Freie und gebundene Variablen

Betrachten wir den Aufbau einer Formel wie z.B. $\exists\ x\ p(x, y)$, wobei p ein zweistelliges Prädikatensymbol sei, stellen wir fest, daß die Variable x eine Sonderrolle gegenüber der Variablen y spielt. Der Existenzquantor bindet den Wert von x, so daß der Wert der Formel unabhängig von dem Wert $f(x)$ ist, an den der Zustand das x bindet (siehe die Definition der $\underline{\mathrm{Wert}}_\Sigma$-Funktion). Wir bezeichnen daher x als eine in $\exists\ x\ p(x, y)$ durch den Existenzquantor *gebundene Variable* und y als eine in $\exists x\ p(x, y)$ *freie Variable*. Ein Analogon finden wir in der Analysis beispielsweise bei dem Ausdruck $\int_0^y f(x)\ dx$, in dem y frei und x gebunden ist.

Wir definieren nun induktiv über den Aufbau der Terme und Formeln die Menge der freien Variablen.

Definition 5.20 Die Menge der freien Variablen eines Terms bzw. Formel ist durch die Funktion

$$\underline{\mathrm{FR}} : \underline{\mathrm{TE}} \cup \underline{\mathrm{FO}} \to \mathfrak{P}(\underline{\mathrm{VA}})$$

gegeben, die jedem Term bzw. Formel die Menge der in ihr enthaltenden *freien Variablen* zuordnet. Wir definieren $\underline{\mathrm{FR}}$ induktiv wie folgt:

1. Für alle Variablen $x \in \underline{\mathrm{VA}}$ gilt $\underline{\mathrm{FR}}(x) = \{x\}$.
2. Für alle n-stelligen Funktionssymbole $g \in \underline{\mathrm{FS}}_n$ und alle Terme $t_1, \ldots, t_n$ gilt
$$\underline{\mathrm{FR}}(g(t_1, \ldots, t_n)) = \bigcup_{j=1}^{n} \underline{\mathrm{FR}}(t_j).$$
3. Für alle n-stelligen Prädikatensymbole $p \in \underline{\mathrm{PS}}_n$ und alle Terme $t_1, \ldots, t_n$ gilt
$$\underline{\mathrm{FR}}(p(t_1, \ldots, t_n)) = \bigcup_{j=1}^{n} \underline{\mathrm{FR}}(t_j).$$
4. Für alle Formeln A gilt $\underline{\mathrm{FR}}(\neg A) = \underline{\mathrm{FR}}(A)$.
5. Für alle Formeln A, B gilt $\underline{\mathrm{FR}}(A \vee B) = \underline{\mathrm{FR}}(A) \cup \underline{\mathrm{FR}}(B)$.

6. Für alle Formeln A gilt $\underline{\mathrm{FR}}(\exists\, x\, A) = \underline{\mathrm{FR}}(A) \setminus \{x\}$.

Für Mengen von Formeln und Termen $X \subset \underline{FO} \cup \underline{TE}$ definieren wir die Menge der freien Variablen von X durch $\underline{\mathrm{FR}}(X) = \bigcup_{a \in X} \underline{\mathrm{FR}}(a)$.

Für die weiteren logischen Zeichen gilt das folgende Lemma, das sich unmittelbar aus der Definition von $\underline{\mathrm{FR}}$ und des Ersetzens der logischen Zeichen durch Kombinationen von $\vee$ und $\neg$ ergibt.

Lemma 5.21

1. Für alle Formeln A, B gilt $\underline{\mathrm{FR}}(A \leftrightarrow B) = \underline{\mathrm{FR}}(A) \cup \underline{\mathrm{FR}}(B)$.
2. Für alle Formeln A, B gilt $\underline{\mathrm{FR}}(A \wedge B) = \underline{\mathrm{FR}}(A) \cup \underline{\mathrm{FR}}(B)$.
3. Für alle Formeln A gilt $\underline{\mathrm{FR}}(\forall\, x\, A) = \underline{\mathrm{FR}}(A) \setminus \{x\}$.

Um den Begriff der gebundenen Variablen einführen zu können, möchten wir formalisieren, was wir unter der Bindung einer Variablen an einen Quantor verstehen.

Seien $x, y \in \Sigma^*$ zwei Zeichenketten, dann ist x ein *Teilwort* von y wenn es Zeichenketten $u, v \in \Sigma^*$ mit $y = uxv$ gibt. x heißt echtes Teilwort von y, wenn $u \neq \Lambda$ oder $v \neq \Lambda$ gilt. Ein Teilwort T' eines Terms bzw. einer Formel T heißt *Teilterm* bzw. *Teilformel*, wenn T' ebenfalls ein Term bzw. eine Formel ist.

Definition 5.22 Eine Variable $x \in \underline{\mathrm{VA}}$ heißt *gebunden* in einer Formel $A \in \underline{\mathrm{FO}}$, wenn es eine Teilformel A' von A der Form $\exists\, x\, B$ bzw. $\forall\, x\, B$ gibt.

Man beachte, daß es Formeln geben kann, in denen eine Variable sowohl frei als gebunden vorkommt. Als Beispiel sei die Formel $p(x) \vee \forall\, x B(x)$ angegeben. Im nächsten Abschnitt werden wir sehen, daß wir diese Formeln aber in andere äquivalente Formeln umwandeln können, in denen die Mengen der freien und gebundenen Variablen disjunkt sind.

5.4.2 Substitution

Wir untersuchen jetzt, welche Beziehung zwischen dem Zustand, den Variablen und dem Wert eines Terms oder einer prädikatenlogischen Formel bestehen. Der erste Satz, das sogenannte *Koinzidenztheorem* besagt, daß der Wert eines Terms oder einer Formel nur vom Zustand der freien Variablen abhängt.

Satz 5.23 Seien Σ eine Struktur, $t \in \underline{\mathrm{TE}}$ ein Term, $A \in \underline{\mathrm{FO}}$ eine Formel und $f, f' : \underline{VA} \to I$ Zustände, dann gelten:

1. Stimmen f und f' auf den freien Variablen von t überein, d.h. gilt $f(x) = f'(x)$ für alle $x \in \underline{\mathrm{FR}}(t)$, so sind die Werte von t in den Zuständen f und f' identisch, d.h. $\underline{\mathrm{wert}}_\Sigma(t, f) = \underline{\mathrm{wert}}_\Sigma(t, f')$.

2. Stimmen f und f' auf den freien Variablen von A überein, d.h. gilt $f(x) = f'(x)$ für alle $x \in \underline{\mathrm{FR}}(A)$, so sind die Werte von A in den Zuständen f und f' identisch, d.h. $\underline{\mathrm{Wert}}_\Sigma(A, f) = \underline{\mathrm{Wert}}_\Sigma(A, f')$.

BEWEIS: Wir beweisen diesen Satz durch Induktion über den Aufbau der Terme bzw. der Formeln (Netzinduktion). Für alle Variablen $x \in \underline{\mathrm{VA}}$ gilt $\underline{\mathrm{FR}}(x) = \{x\}$ und damit für f, f' mit $f(x) = f'(x)$

$$\underline{\mathrm{wert}}_\Sigma(x, f) = f(x) = f'(x) = \underline{\mathrm{wert}}_\Sigma(x, f').$$

Für Terme $g(t_1, \ldots, t_n)$ ist die Menge der freien Variablen $F = \underline{\mathrm{FR}}(g(t_1, \ldots, t_n)) = \bigcup_{i=1}^{n} \underline{\mathrm{FR}}(t_i)$. Wenn also $f(x) = f'(x)$ für alle $x \in F$ ist, so gilt $f(x) = f'(x)$ auch für alle $x \in \underline{\mathrm{FR}}(t_i)$ für alle t_i und damit nach Induktionsannahme $\underline{\mathrm{wert}}_\Sigma(t_i, f) = \underline{\mathrm{wert}}_\Sigma(t_i, f')$. Daher ist auch

$$\begin{aligned} \underline{\mathrm{wert}}_\Sigma(g(t_1, \ldots, t_n), f) &= \omega(g)(\underline{\mathrm{wert}}_\Sigma(t_1, f), \ldots, \underline{\mathrm{wert}}_\Sigma(t_n, f)) \\ &= \omega(g)(\underline{\mathrm{wert}}_\Sigma(t_1, f'), \ldots, \underline{\mathrm{wert}}_\Sigma(t_n, f')) \\ &= \underline{\mathrm{wert}}_\Sigma(g(t_1, \ldots, t_n), f'). \end{aligned}$$

Im Falle von Formeln beweisen wir nur den Induktionsschritt $(B \to \exists\, x\, B)$. Die Menge der freien Variablen von $\exists\, x\, B$ ist durch $F = \underline{\mathrm{FR}}(\exists\, x\, B) = \underline{\mathrm{FR}}(B) \setminus \{x\}$ gegeben. Stimmen f und f' auf F überein, gilt nach Induktionsannahme für jedes ξ, daß $\underline{\mathrm{Wert}}_\Sigma(B, f\left\langle \begin{smallmatrix} x \\ \xi \end{smallmatrix} \right\rangle) = \underline{\mathrm{Wert}}_\Sigma(B, f'\left\langle \begin{smallmatrix} x \\ \xi \end{smallmatrix} \right\rangle)$ ist. Nach Definition von $\underline{\mathrm{Wert}}_\Sigma(\exists\, x\, B, f)$ ist damit auch

$$\underline{\mathrm{Wert}}_\Sigma(\exists\, x\, B, f) = \underline{\mathrm{Wert}}_\Sigma(\exists\, x\, B, f').$$

□

Für Terme und Formeln ohne freie Variablen ergibt sich das folgende Korollar.

Korollar 5.24 Seien f und f' beliebige Zustände. Dann gilt für Terme $t \in \underline{\mathrm{TE}}$ mit $\underline{\mathrm{FR}}(t) = \emptyset$, daß $\underline{\mathrm{wert}}_\Sigma(t, f) = \underline{\mathrm{wert}}_\Sigma(t, f')$ ist, und für Formeln $A \in \underline{\mathrm{FO}}$ mit $\underline{\mathrm{FR}}(A) = \emptyset$, daß $\underline{\mathrm{Wert}}_\Sigma(A, f) = \underline{\mathrm{Wert}}_\Sigma(A, f')$ ist.

BEWEIS: Da t bzw. A keine freien Variablen haben, sind die Voraussetzungen des Koinzidenztheorems (Satz 5.23) erfüllt. □

Terme ohne freie Variablen spezifizieren in jedem Zustand das gleiche Individuum. Daher nennen wir solche Terme auch *Konstanten*, *konstante Terme* oder *variablenfreie Terme*. Die Menge aller variablenfreien Terme bezeichnen wir mit $\underline{\mathrm{TE}}^-$.

Formeln ohne freie Variablen sind für alle Zustände entweder wahr oder für alle Zustände falsch. Daher nennen wir Formeln ohne freie Variablen auch *Aussagen* oder *abgeschlossene Formeln*. Die Menge aller Aussagen bezeichnen wir mit $\underline{\mathrm{FO}}^-$.

Für Aussagen gilt das folgende einfach zu beweisende Lemma.

Lemma 5.25 Es sei $A \in \underline{\mathrm{FO}}^-$ eine Aussage, dann gilt:

1. $A \in \underline{\mathrm{VAL}}_\Sigma \;\Leftrightarrow\; A \in \underline{\mathrm{SAT}}_\Sigma$.
2. $\neg A \notin \underline{\mathrm{VAL}}_\Sigma \;\Leftrightarrow A \in \underline{\mathrm{VAL}}_\Sigma$.

Lemma 5.26

1. Sei $t \in \underline{\text{TE}}$ ein Term mit $x \notin \underline{\text{FR}}(t)$, dann gilt:

$$\forall f : \underline{\text{VA}} \to I,\ \forall \xi \in I :\ \underline{\text{wert}}_\Sigma(t, f) = \underline{\text{wert}}_\Sigma(t, f \left\langle \begin{smallmatrix} x \\ \xi \end{smallmatrix} \right\rangle).$$

2. Sei $A \in \underline{\text{FO}}$ eine Formel mit $x \notin \underline{\text{FR}}(A)$, dann gilt:

$$\forall f : \underline{\text{VA}} \to I,\ \forall \xi \in I :\ \underline{\text{Wert}}_\Sigma(A, f) = \underline{\text{Wert}}_\Sigma(A, f \left\langle \begin{smallmatrix} x \\ \xi \end{smallmatrix} \right\rangle).$$

BEWEIS: Wegen $x \notin \underline{\text{FR}}(t)$ bzw. $x \notin \underline{\text{FR}}(A)$ stimmen f und $f \left\langle \begin{smallmatrix} x \\ \xi \end{smallmatrix} \right\rangle$ auf den freien Variablen von t bzw. A überein. Daher können wir das Koinzidenztheorem (Satz 5.23) anwenden. □

Nun kommen wir zum Begriff der Substitution. Dabei ersetzen wir in einem Term t oder in einer Formel A eine Variable x durch einen Term t'. Den so entstandenen Term oder Formel bezeichnen wir mit $t_x[t']$ bzw. $A_x[t']$. Nun wird man von einer Substitution sicherlich verlangen, daß die Formel A das gleiche über x aussagt, wie die Formel $A_x[t]$ über t. Bevor wir eine formale Definition der Substitution geben, überlegen wir zunächst, welche Probleme dabei auftauchen können.

Seien $x, y \in \underline{\text{VA}}$, $x \neq y$, p ein dreistelliges Prädikatensymbol und $A = p(x, y, y)$ eine Formel. Die freien Variablen von A sind $\{x, y\}$ und $A_x[t] = p(t, y, y)$ und $A_y[t] = p(x, t, t)$. Sei q ein einstelliges Prädikatensymbol und $B = q(x)$ eine Formel. Die freien Variablen von B sind $\{x\}$ und daher ist $B_y[t] = B$, da y nicht in B vorkommt. Sei r ein zweistelliges Prädikatensymbol und $C = r(x, y) \vee \exists\, x\, r(x, x)$ eine Formel. Die freien Variablen von C sind $\{x, y\}$ und daher ist $C_x[t] = r(t, y) \vee \exists\, x\, r(x, x)$. Dabei wird das gebundene x nicht substituiert. Sei $D = \exists\, x\, r(x, y)$ und $t = g(z)$. Die freien Variablen von D sind $\{y\}$ und daher ist $D_y[t] = \exists\, x\, r(x, g(z))$.

Bisher traten keine Probleme auf. Nun sei aber $t = g(x)$. Dann würde eine Substitution von y durch t in D zu

$$D_y[t] = \exists\, x\, r(x, g(x))$$

führen. Diese Formel sagt jedoch nicht mehr aus das gleiche über $g(x)$ aus wie D über y. Dies geschieht dadurch, daß eine in t freie Variable in $D_y[t]$ gebunden wird. Für die Gültigkeit einer Substitution wird man daher fordern, daß freie Variablen in t nicht in $D_y[t]$ gebunden werden, oder anders ausgedrückt, daß $\underline{\text{FR}}(t) \subseteq \underline{\text{FR}}(D_y[t])$ gilt. Nun können wir formal die Substitution einführen.

Definition 5.27 Die syntaktische Operation *Substitution* definiert induktiv über den Aufbau der Terme und Formeln für jeden Term $t' \in \underline{\text{TE}}$, jede Formel $A \in \underline{\text{FO}}$, jede Variable $x \in \underline{\text{VA}}$ und jeden Term $t \in \underline{\text{TE}}$ eine Zeichenreihe $t'_x[t]$ bzw. $A_x[t]$ wie folgt:

1. $y_x[t] = \texttt{IF}\ x = y\ \texttt{THEN}\ t\ \texttt{ELSE}\ y$ für alle $y \in \underline{\text{VA}}$
2. $(g(t_1, \ldots, t_n))_x[t] = g((t_1)_x[t], \ldots, (t_n)_x[t])$
3. $(p(t_1, \ldots, t_n))_x[t] = p((t_1)_x[t], \ldots, (t_n)_x[t])$
4. $(\neg B)_x[t] = \neg(B_x[t])$
5. $(B \vee C)_x[t] = B_x[t] \vee C_x[t]$

6. $(\exists y\ B)_x[t] = \begin{cases} \underline{\text{undefiniert}} & \text{falls } x \in \underline{\text{FR}}(\exists y\ B) \text{ und } y \in \underline{\text{FR}}(t) \\ \exists y\ B & \text{falls } x \notin \underline{\text{FR}}(\exists y\ B) \\ \exists y\ (B_x[t]) & \text{falls } x \in \underline{\text{FR}}(\exists y\ B) \text{ und } y \notin \underline{\text{FR}}(t) \end{cases}$

Im folgenden werden wir nur $A_x[t]$ schreiben, falls t für x in A substituierbar ist, d.h. wenn $\underline{\text{FR}}(t) \subseteq \underline{\text{FR}}(A_x[t])$ gilt. Wir werden später sehen, daß die Konfliktsituationen bei der Substitution immer durch eine sogenannte gebundene Umbenennung gelöst werden können, wie dies aus der Analysis bekannt ist ($\int_a^b f(x)dx = \int_a^b f(y)dy$). Bevor wir dies jedoch zeigen können, wollen wir untersuchen, welche Auswirkungen eine Substitution auf den Wert von Termen und Formeln hat. Der folgende Satz (Überführungstheorem) ist intuitiv sofort verständlich.

Satz 5.28 Sei $\Sigma = (I, \omega)$ eine Struktur, A eine Formel, t, t' Terme und $f : \underline{\text{VA}} \to I$ ein Zustand. Dann gelten:

1. $\underline{\text{wert}}_\Sigma(t'_x[t], f) = \underline{\text{wert}}_\Sigma(t', f \left\langle \begin{smallmatrix} x \\ \underline{\text{wert}}_\Sigma(t, f) \end{smallmatrix} \right\rangle)$.

2. $\underline{\text{Wert}}_\Sigma(A_x[t], f) = \underline{\text{Wert}}_\Sigma(A, f \left\langle \begin{smallmatrix} x \\ \underline{\text{wert}}_\Sigma(t, f) \end{smallmatrix} \right\rangle)$.

BEWEIS: Wir beweisen das Überführungstheorem induktiv über den Aufbau von Formeln und Termen. Der einzige schwierige Fall sind Formeln von der Form $\exists\, y\, B$. Hier führen wir eine Fallunterscheidung durch.

1. Sei $x \notin \underline{\text{FR}}(\exists\, y\, B)$. Dann ist

$$\underline{\text{Wert}}_\Sigma((\exists\, y\, B)_x[t], f) = \underline{\text{Wert}}_\Sigma((\exists\, y\, B)_x, f \left\langle \begin{smallmatrix} x \\ \underline{\text{wert}}_\Sigma(t, f) \end{smallmatrix} \right\rangle)$$

   trivialerweise erfüllt.

2. Sei $x \in \underline{\text{FR}}(\exists\, y\, B)$ und $y \notin \underline{\text{FR}}(t)$ (dies ist die zweite Voraussetzung für die Gültigkeit einer Substitution). Dann ist

$$\begin{aligned} &\underline{\text{Wert}}_\Sigma((\exists\, y\, B)_x[t], f) = W \\ \Leftrightarrow\quad & \exists\, \eta \in I :\ \underline{\text{Wert}}_\Sigma(B_x[t], f \left\langle \begin{smallmatrix} y \\ \eta \end{smallmatrix} \right\rangle) = W \\ \Leftrightarrow\quad & \exists\, \eta \in I :\ \underline{\text{Wert}}_\Sigma(B, f \left\langle \begin{smallmatrix} y \\ \eta \end{smallmatrix} \right\rangle \left\langle \begin{smallmatrix} x \\ \underline{\text{wert}}_\Sigma(t, f) \end{smallmatrix} \right\rangle) = W \quad \text{(Induktionsannahme)} \\ \Leftrightarrow\quad & \exists\, \eta \in I :\ \underline{\text{Wert}}_\Sigma(B, f \left\langle \begin{smallmatrix} x \\ \underline{\text{wert}}_\Sigma(t, f) \end{smallmatrix} \right\rangle \left\langle \begin{smallmatrix} y \\ \eta \end{smallmatrix} \right\rangle) = W \quad \text{(Lemma 5.10)} \\ \Leftrightarrow\quad & \underline{\text{Wert}}_\Sigma(\exists\, y\, B, f \left\langle \begin{smallmatrix} x \\ \underline{\text{wert}}_\Sigma(t, f) \end{smallmatrix} \right\rangle) = W. \end{aligned}$$

Die verbleibenden Fälle sind einfach nachzuweisen. Damit folgt die Behauptung. □

Die obigen Beobachtungen ermöglichen es uns jetzt, Umbenennung von gebundenen Variablen vorzunehmen. Dabei ist jedoch darauf zu achten, daß keine freien Variablen dadurch gebunden werden. Desweiteren ist eine sogenannte *Variablenkollision* zu vermeiden, d.h. wir können die Substitution $\exists\, x\, B_x[y]$ nicht durchführen, wenn y gebunden in B vorkommt, da dadurch y zweifach gebunden würde. Das folgende Lemma (Lemma der gebundenen Umbenennung) präzisiert unsere Intuition.

Lemma 5.29 Für alle $B \in \underline{\mathrm{FO}}$ und alle $y \notin \underline{\mathrm{FR}}(B)$, die nicht auch gebunden in B auftreten, gelten

$$(\exists\, x\, B) \;\leftrightarrow\; (\exists\, y\, B_x[y]) \in \underline{\mathrm{VAL}}$$
$$(\forall\, x\, B) \;\leftrightarrow\; (\forall\, y\, B_x[y]) \in \underline{\mathrm{VAL}}.$$

Falls y gebunden in B auftritt, kann man das Lemma rekursiv auf B bzw. Teilformen von B anwenden, und damit erreichen, daß y auch in B nicht gebunden ist. Daher ist dies keine Einschränkung.

Beweis:

$\underline{\mathrm{Wert}}_\Sigma((\exists\, y\, B_x[y]), f) = W$

$\Leftrightarrow \quad \exists\, \xi \in I : \underline{\mathrm{Wert}}_\Sigma(B_x[y], f\left\langle \begin{smallmatrix} y \\ \xi \end{smallmatrix} \right\rangle) = W$

$\Leftrightarrow \quad \exists\, \xi \in I : \underline{\mathrm{Wert}}_\Sigma(B, f\left\langle \begin{smallmatrix} y \\ \xi \end{smallmatrix} \right\rangle \left\langle \begin{smallmatrix} x \\ \underline{\mathrm{wert}}_\Sigma(y, f\left\langle \begin{smallmatrix} y \\ \xi \end{smallmatrix} \right\rangle) \end{smallmatrix} \right\rangle) = W$ (Überführungstheroem 5.28)

$\Leftrightarrow \quad \exists\, \xi \in I : \underline{\mathrm{Wert}}_\Sigma(B, f\left\langle \begin{smallmatrix} y \\ \xi \end{smallmatrix} \right\rangle \left\langle \begin{smallmatrix} x \\ \xi \end{smallmatrix} \right\rangle) = W$

$\Leftrightarrow \quad \exists\, \xi \in I : \underline{\mathrm{Wert}}_\Sigma(B, f\left\langle \begin{smallmatrix} x \\ \xi \end{smallmatrix} \right\rangle \left\langle \begin{smallmatrix} y \\ \xi \end{smallmatrix} \right\rangle) = W$ (Lemma 5.10)

$\Leftrightarrow \quad \exists\, \xi \in I : \underline{\mathrm{Wert}}_\Sigma(B, f\left\langle \begin{smallmatrix} x \\ \xi \end{smallmatrix} \right\rangle) = W$ da $y \notin \underline{\mathrm{FR}}(B)$

$\Leftrightarrow \quad \underline{\mathrm{Wert}}_\Sigma(\exists\, x\, B, f) = W.$

Für den All-Quantor gilt eine analoge Rechnung. □

5.4.3 Quantorengesetze

Aus dem Überführungstheorem 5.28 lassen sich die folgenden Gesetze über Quantoren schließen.

Lemma 5.30

1. Für alle $t \in \underline{\mathrm{TE}}$ gilt: $(\forall\, x\, A) \to (A_x[t]) \in \underline{\mathrm{VAL}}$.
2. Für alle $t \in \underline{\mathrm{TE}}$ gilt: $(A_x[t]) \to (\exists\, x\, A) \in \underline{\mathrm{VAL}}$.
3. $(\forall\, x\, A) \to A \in \underline{\mathrm{VAL}}$ und $A \to (\exists\, x\, A) \in \underline{\mathrm{VAL}}$
4. $A \in \underline{\mathrm{VAL}}_\Sigma \;\Leftrightarrow\; \forall\, x\, A \in \underline{\mathrm{VAL}}_\Sigma$

Beweis:

1. Nach Definition gilt $\underline{\mathrm{Wert}}_\Sigma(\forall\, x\, A, f) = W \;\Leftrightarrow\; \forall\, \xi \in I : \underline{\mathrm{Wert}}_\Sigma(A, f\left\langle \begin{smallmatrix} x \\ \xi \end{smallmatrix} \right\rangle) = W.$ Insbesondere können wir auch $\xi = \underline{\mathrm{wert}}_\Sigma(t, f)$ wählen. Dann ist aber
$$\underline{\mathrm{Wert}}_\Sigma(A, f\left\langle \begin{smallmatrix} x \\ \xi \end{smallmatrix} \right\rangle) = \underline{\mathrm{Wert}}_\Sigma(A, f\left\langle \begin{smallmatrix} x \\ \underline{\mathrm{wert}}_\Sigma(t, f) \end{smallmatrix} \right\rangle) = \underline{\mathrm{Wert}}_\Sigma(A_x[t], f).$$
2. folgt aus 1. unter Verwendung der Gleichheit $\exists\, x\, A \;=\; \neg\forall\, x\, \neg A$.
3. Setze $t = x$.

4. Die Richtung ($\Leftarrow$) folgt aus 3. Die Richtung ($\Rightarrow$) kann man wie folgt einsehen:

$$\begin{aligned} A \in \underline{\mathrm{VAL}}_\Sigma &\Leftrightarrow \forall f : \underline{\mathrm{VA}} \to I : \underline{\mathrm{Wert}}_\Sigma(A, f) = W \\ &\Rightarrow \forall f : \underline{\mathrm{VA}} \to I \, \forall \xi \in I : \underline{\mathrm{Wert}}_\Sigma(A, f\left\langle \begin{smallmatrix} x \\ \xi \end{smallmatrix} \right\rangle) = W \\ &\Leftrightarrow \forall f : \underline{\mathrm{VA}} \to I : \underline{\mathrm{Wert}}_\Sigma(\forall x\, A, f) = W \Leftrightarrow (\forall x\, A) \in \underline{\mathrm{VAL}}_\Sigma. \end{aligned}$$

□

Korollar 5.31 Sei $A \in \underline{\mathrm{FO}}$ eine Formel und π eine Permutation der Zahlen $\{1, \ldots, n\}$, dann gilt

$$A \in \underline{\mathrm{VAL}}_\Sigma \Leftrightarrow \forall x_{\pi(1)} \ldots \forall x_{\pi(n)} A \in \underline{\mathrm{VAL}}_\Sigma$$

BEWEIS: Der Beweis erfolgt durch Induktion über n unter Benutzung von Lemma 5.30.4. □

5.5 Logisches Schließen

Wir möchten uns jetzt mit der Frage beschäftigen, wann eine Aussage aus gegebenen Voraussetzung logisch *folgt*. Bezeichnungsweisen hierfür sind *logisches Schließen* oder *semantisches Folgern*. Mit den bisher bekannten Notationen können wir den Folgerungsbegriff wie folgt präzisieren.

Definition 5.32 Eine Formel A *folgt logisch* aus einer Formelmenge X in einer Struktur Σ genau dann, wenn für alle Zustände $f : \underline{\mathrm{VA}} \to I$ gilt:

$$\forall B \in X : \underline{\mathrm{Wert}}_\Sigma(B, f) = W \Rightarrow \underline{\mathrm{Wert}}_\Sigma(A, f) = W.$$

Eine Formel A *folgt logisch* aus einer Formelmenge X genau dann, wenn A in allen Strukturen Σ logisch aus X folgt.

Zur Formulierung des Folgerungsbegriff der Logik führen wir geeignete Bezeichnungen ein.

Definition 5.33 Eine Struktur Σ heißt ein *Modell* einer Formelmenge X, wenn die Formelmenge X in Σ gültig ist, d.h. $X \subseteq \underline{\mathrm{VAL}}_\Sigma$. Die Menge aller Modelle für eine Formelmenge X bezeichnen wir durch $\underline{\mathrm{MOD}}(X)$.

Für Formeln A schreiben wir statt $\underline{\mathrm{MOD}}(\{A\})$ kurz auch $\underline{\mathrm{MOD}}(A)$. Offensichtlich gilt:

- $\underline{\mathrm{MOD}}(X) \subseteq \underline{\mathrm{ST}}^{\underline{B}}$.
- $\underline{\mathrm{MOD}} : \mathfrak{P}(\underline{\mathrm{FO}}^{\underline{B}}) \to \mathfrak{P}(\underline{\mathrm{ST}}^{\underline{B}})$.
- A folgt logisch aus X genau dann, wenn $\underline{\mathrm{MOD}}(X) \subseteq \underline{\mathrm{MOD}}(A)$.

Definition 5.34 Eine Formel A heißt eine *logische Folgerung* aus einer Formelmenge X, wenn alle Modelle von X auch Modelle von A sind. Die Menge aller logischen Folgerungen einer Formelmenge X bezeichnen wir mit $\underline{\mathrm{FL}}(X)$. Sie heißt auch *Folgerungsmenge*. Statt $A \in \underline{\mathrm{FL}}(X)$ wird in der Literatur auch häufig die Notation $X \models A$ verwendet.

Für Formeln A schreiben wir statt $\underline{\mathrm{FL}}(\{A\})$ kurz auch $\underline{\mathrm{FL}}(A)$. Für die Menge der Modelle gilt:

- Die Folgerungsmenge $\underline{\mathrm{FL}}$ ist eine Abbildung:

$$\underline{\mathrm{FL}} : \mathfrak{P}(\underline{\mathrm{FO}}) \to \mathfrak{P}(\underline{\mathrm{FO}}).$$

- $Y \subseteq \underline{\mathrm{FL}}(X) \Leftrightarrow \underline{\mathrm{MOD}}(X) \subseteq \underline{\mathrm{MOD}}(Y)$
- Jede Struktur ist Modell der leeren Menge, d.h. $\underline{\mathrm{ST}}^{\underline{B}} = \underline{\mathrm{MOD}}(\emptyset)$.
- Für jede Struktur Σ ist Σ ein Modell der in Σ gültigen Formeln, d.h.

$$\forall \Sigma : \Sigma \in \underline{\mathrm{MOD}}(\underline{\mathrm{VAL}}_{\Sigma}).$$

- Eine Formel ist genau dann allgemeingültig, wenn jede Struktur ein Modell dieser Formel ist, d.h. $A \in \underline{\mathrm{VAL}} \Leftrightarrow \underline{\mathrm{MOD}}(A) = \underline{\mathrm{ST}}^{\underline{B}}$.

Die obigen Behauptungen können durch elementare Umformungen unter Benutzung der Definition von $\underline{\mathrm{MOD}}$ and $\underline{\mathrm{FL}}$ bewiesen werden. Das gleiche gilt für das nachfolgende Lemma.

Lemma 5.35

1. Für Formeln $A, B \in \underline{\mathrm{FO}}$ gilt $\underline{\mathrm{MOD}}(A \wedge B) = \underline{\mathrm{MOD}}(A) \cap \underline{\mathrm{MOD}}(B)$.
2. Für Formeln $A, B \in \underline{\mathrm{FO}}^-$ gilt $\underline{\mathrm{MOD}}(A \vee B) = \underline{\mathrm{MOD}}(A) \cup \underline{\mathrm{MOD}}(B)$.
3. Für eine Aussage $A \in \underline{\mathrm{FO}}^-$ gilt $\underline{\mathrm{MOD}}(\neg A) = \underline{\mathrm{ST}}^{\underline{B}} \setminus \underline{\mathrm{MOD}}(A)$ und damit auch $\underline{\mathrm{MOD}}(A) \cap \underline{\mathrm{MOD}}(\neg A) = \emptyset$ und $\underline{\mathrm{MOD}}(A) \cup \underline{\mathrm{MOD}}(\neg A) = \underline{\mathrm{ST}}^{\underline{B}}$.
4. Für Formelmengen $X, Y \subset \underline{\mathrm{FO}}$ gilt $\underline{\mathrm{MOD}}(X \cup Y) = \underline{\mathrm{MOD}}(X) \cap \underline{\mathrm{MOD}}(Y)$.
5. Für Formelmengen $X \subseteq Y$ gilt $\underline{\mathrm{MOD}}(Y) \subseteq \underline{\mathrm{MOD}}(X)$. d.h. die kleinere Menge von Formeln hat die größere Menge von Modellen.
6. $\underline{\mathrm{VAL}} = \underline{\mathrm{FL}}(\emptyset)$, d.h. die allgemeingültigen Formeln sind genau die Formeln, die voraussetzungslos gefolgert werden können.
7. $X \subset \underline{\mathrm{FL}}(X)$, d.h. alle Voraussetzungen können gefolgert werden.
8. Für Formelmengen $X \subseteq Y$ gilt $\underline{\mathrm{FL}}(X) \subseteq \underline{\mathrm{FL}}(Y)$, d.h. die größere Menge an Voraussetzungen hat die größere Menge an Folgerungen.
9. Für jede Formelmenge X gilt $\underline{\mathrm{MOD}}(X) = \underline{\mathrm{MOD}}(\underline{\mathrm{FL}}(X))$.
10. $\underline{\mathrm{FL}}(X) = \underline{\mathrm{FL}}(\underline{\mathrm{FL}}(X))$, d.h. aus der Menge der Folgerungen können keine neuen Folgerungen gezogen werden.
11. Für Formeln A, B gilt: $(A \to B) \in \underline{\mathrm{FL}}(X) \Rightarrow B \in \underline{\mathrm{FL}}(X \cup \{A\})$.

Zu einer Menge von Strukturen $\mathfrak{A} \subset \underline{\mathrm{ST}}^{\underline{B}}$ sei X eine Formelmenge, die gerade $\mathfrak{A}$ als Modellmenge hat, also $\mathfrak{A} = \underline{\mathrm{MOD}}(X)$. Aus obigen Lemma folgt, daß dann auch $\mathfrak{A} = \underline{\mathrm{MOD}}(\underline{\mathrm{FL}}(X))$ gilt, so daß aus X alle in $\mathfrak{A}$ gültigen Formeln folgerbar sind.

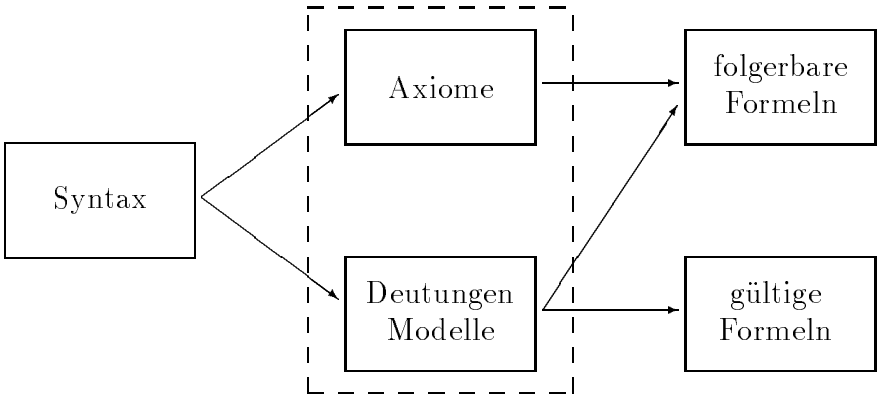
Definition 5.36 Eine entscheidbare (total TM-berechenbare) Menge von Formeln $X \subset \underline{\mathrm{FO}}$ heißt ein *Axiomensystem* für eine Klasse $\mathfrak{A} \subseteq \underline{\mathrm{ST}}^{\underline{B}}$ von Strukturen, wenn $\underline{\mathrm{MOD}}(X) = \mathfrak{A}$ gilt.

Falls X ein Axiomensystem für $\mathfrak{A} \subseteq \underline{\mathrm{ST}^B}$ ist, so gilt

$$\underline{\mathrm{FL}}(X) = \bigcap_{\Sigma \in \mathfrak{A}} \underline{\mathrm{VAL}}_\Sigma .$$

X ist natürlich gerade das Axiomensystem für $\underline{\mathrm{MOD}}(X)$.

Zusammenfassend sei das Verhältnis der eingeführten Begriffe zueinander in der folgenden Skizze verdeutlicht, wobei $\boxed{\mathrm{A}} \longrightarrow \boxed{\mathrm{B}}$ bedeutet, daß A zur Definition von B benötigt wird.



5.6 Die Unentscheidbarkeit der Prädikatenlogik

In diesem Abschnitt werden wir zeigen, daß es kein Verfahren gibt, das für eine beliebige Formelmenge und eine beliebige Formel A entscheidet, ob A eine logische Folgerung aus X ist. Wir bezeichnen dieses Problem als *Allgemeingültigkeitsproblem* der Prädikatenlogik (AGPL):

Probleminstanz: Formelmenge X, Formel A.
Frage: Ist $A \in \underline{\mathrm{FL}}(X)$?
zugehörige Sprache: AGPL:= $\{(X, A) \mid A \in \underline{\mathrm{FL}}(X)\}$.

Im Jahre 1936 hat CHURCH die Hoffnung der Mathematiker, daß die Prädikatenlogik entscheidbar sei, zerstört.

Satz 5.37 AGPL ist unentscheibar.

BEWEIS: Wir werden zeigen, daß das AGPL schon für $X = \emptyset$ unentscheidbar ist. Wir werden das 01PCP aus Abschnitt 3.2.2, von dem wir wissen, daß es unentscheidbar ist, auf das AGPL reduzieren. Aus Lemma 3.13 folgt dann die Unentscheidbarkeit von AGPL. Dazu konstuieren wir aus der Instanz $K = \{(x_1, y_1), (x_2, y_2), \ldots, (x_k, y_k)\}$ eines 01PCP eine prädikatenlogische Formel A_K, die genau dann gültig ist ($A_K \in \underline{\mathrm{FL}}(\emptyset) = \underline{\mathrm{VAL}}$), wenn das 01PCP eine Lösung besitzt. Wir wählen die Basis $\underline{B} = (\{c, f_0, f_1\}, P)$, wobei c ein nullstelliges und f_0, f_1 einstellige Funktionsymbole und P ein zweistelliges Prädikatensymbol ist. Sei $f_{j_1 j_2 \ldots j_m}(x) = f_{j_m}(f_{j_{m-1}}(\cdots(f_{j_1}(x))\cdots))$ für $j_i \in \{0, 1\}$. Dann sei $F = A_K$ gegeben durch

$$F = ((F_1 \wedge F_2) \rightarrow F_3),$$

wobei

$$\begin{aligned} F_1 &= \bigwedge_{i=1}^{k} P(f_{x_i}(c), f_{y_i}(c)) \\ F_2 &= \forall u \forall v \, (P(u,v) \rightarrow \bigwedge_{i=1}^{k} P(f_{x_i}(u), f_{y_i}(v))) \\ F_3 &= \exists y \, P(y,y). \end{aligned}$$

Offensichtlich ist F durch eine totale Turing-Maschine aus K berechenbar. Wir müssen zeigen, daß $F \in \underline{\mathrm{VAL}}$ genau dann, wenn K eine Lösung besitzt.

Sei also $F \in \underline{\mathrm{VAL}}$ angenommen. Dann ist $F \in \underline{\mathrm{VAL}}_\Sigma$ für alle Strukturen Σ für $\underline{B}$, insbesondere auch für die spezielle Struktur $\Sigma = (\{0,1\}^*, \omega)$ mit

- $\omega(c) = \Lambda$,
- $\omega(f_0)(z) = z0$ und $\omega(f_1)(z) = z1$ für $z \in \{0,1\}^*$.
- $\omega(P)(z_1, z_2) = W$ genau dann, wenn $z_1, z_2 \in \{0,1\}^+$ und es Indizes $i_1, \ldots, i_t$ mit $z_1 = x_{i_1} \ldots x_{i_t}$ und $z_2 = y_{i_1} \ldots y_{i_t}$ gibt.

Die Struktur ist gerade so gewählt, daß $F \in \underline{\mathrm{VAL}}_\Sigma$ ist und Σ das 01PCP simuliert. Der Formelteil F_1 stellt sicher, daß x_i und y_i ein korrespondierendes Paar ist, F_2 besagt folgendes: wenn zwei Zeichenketten u, v zueinander korrespondieren, so korrespondieren auch die Zeichenketten ux_i und vy_i für alle i miteinander. F_1 und F_2 modellieren also gerade das 01PCP Problem K. F_3 besagt, daß K eine Lösung besitzt. F ist genau dann in $\underline{\mathrm{VAL}}_\Sigma$, wenn aus F_1 und F_2 die Formel F_3 folgt. Da aber F_1 und F_2 wahr sind, muß damit auch $F_3 \in \underline{\mathrm{VAL}}_\Sigma$ gelten, d.h. K besitzt eine Lösung.

Nun zeigen wir die Rückrichtung. Wir nehmen an K habe eine Lösung. Dann müssen wir zeigen, daß F in allen Strukturen Σ für $\underline{B}$ gültig ist, d.h. anders ausgedrückt, daß für alle Strukturen $\Sigma \in \underline{\mathrm{ST}}^B$ $\Sigma \in \underline{\mathrm{MOD}}(F)$ gilt. Falls $\Sigma \notin \underline{\mathrm{MOD}}(F_1)$ oder $\Sigma \notin \underline{\mathrm{MOD}}(F_2)$ so folgt direkt durch Umformung der Formel $F = ((F_1 \wedge F_2) \rightarrow F_3)$ und Benutzen der Regeln aus Lemma 5.35, daß $\Sigma \in \underline{\mathrm{MOD}}(F)$ gilt. Sei also $\Sigma = (I, \omega) \in \underline{\mathrm{MOD}}(F_1) \cap \underline{\mathrm{MOD}}(F_2)$. Wir definieren eine Einbettung $\alpha : \{0,1\}^* \rightarrow I$ durch $\alpha(\Lambda) = \omega(c)$, $\alpha(x0) = \omega(f_0)(\alpha(x))$ und $\alpha(x1) = \omega(f_1)(\alpha(x))$. Damit ist für $x = x_1 \ldots x_n$

$$\alpha(x) = \omega(f_{x_n})(\omega(f_{x_{n-1}})(\cdots(\omega(f_{x_1})(\omega(c)))\cdots)).$$

Da $\Sigma \in \underline{\mathrm{MOD}}(F_1)$ ist, gilt für alle $1 \leq i \leq k$

$$\omega(P)(\alpha(x_i), \alpha(y_i)) = W.$$

Da auch $\Sigma \in \underline{\mathrm{MOD}}(F_2)$ ist, gilt induktiv für alle $1 \leq i_j \leq k$

$$\omega(P)(\alpha(x_{i_1} \ldots x_{i_n}), \alpha(y_{i_1} \ldots y_{i_n})) = W.$$

Da K eine Lösung hat, z.B. $j_1, \ldots, j_t$, mit $x_{j_1} \ldots x_{j_t} = \xi = y_{j_1} \ldots y_{j_t}$ ist auch $F_3 \in \underline{\mathrm{VAL}}_\Sigma$. Hieraus folgt jedoch sofort $F \in \underline{\mathrm{VAL}}_\Sigma$ oder äquivalent $\Sigma \in \underline{\mathrm{MOD}}(F)$. □

Als Korollar ergibt sich sofort die Unentscheidbarkeit des Erfüllbarkeitsproblems der Prädikatenlogik (EPP):

Probleminstanz: Formel A.
Frage: Ist $A \in \underline{\mathrm{SAT}}$?
zugehörige Sprache: EPP:= $\{A \mid A \in \underline{\mathrm{SAT}}\}$.

Korollar 5.38 Das Erfüllbarkeitsproblem der Prädikatenlogik ist unentscheidbar.

BEWEIS: Unmittelbar aus der Tatsache: $A \in \underline{\text{SAT}} \Leftrightarrow (\neg A) \notin \underline{\text{VAL}}$. □

Definition 5.39 Eine Formel $B \in \underline{\text{FO}}$ heißt eine *Pränexe Normalformel*, wenn B die folgenden Eigenschaften erfüllt:
Alle Quantoren mit den dazugehörigen an diese Quantoren gebundenen Variablen stehen am Anfang von B, d.h. $B = Q^1 x_1 \ldots Q^n x_n M$, wobei $Q^i \in \{\forall, \exists\}$ Quantoren und M eine quantorenfreie Formel (eine sogenannte Matrix) mit $\{x_1, \ldots x_n\} \subseteq \underline{\text{FR}}(M)$ sind. $Q^1 x_1 \ldots Q^n x_n$ nennt man auch das *Präfix* von B.

Es gibt jedoch einige Typen von Formeln, die entscheidbar sind:

- Quantorenfreie Formeln (Aussagenlogik) (POST, LUKASIEWICZ, WITTGENSTEIN, BEHMAN, 1921).
- Pränexe Normalformeln mit Präfixtyp $\forall^r \exists^s$ (BERNAYS, SCHÖNFINKEL, 1928).
- Pränexe Normalformeln mit Präfixtyp $\forall^r \exists \forall^s$ (ACKERMANN, SKOLEM, HERBRANDT, 1928).
- Pränexe Normalformeln mit Präfixtyp $\forall^r \exists^2 \forall^s$ (GÖDEL, KALMÁR, SCHÜTTE, 1934).

Kapitel 6

Datenstrukturen und Algorithmen

In diesem Kapitel werden wir die grundlegensten Datenstrukturen, mit denen Algorithmen arbeiten, kennenlernen. Dabei werden wir einerseits dieses Gebiet von der mathematischen Seite beleuchten, indem wir abstrakte mathematische Definition von Datentypen und der auf ihnen wirkenden Operation geben, auf der anderen Seite jedoch mögliche Implementationen in der Praxis nicht außer Sicht lassen. Für weiterführende Algorithmen verweisen wir auf das Buch [AhHoUl83] von AHO, HOPCROFT und ULLMAN.

Definition 6.1 Ein *abstrakter Datentyp* $G = (D, F, R)$ ist ein 3-Tupel, wobei D eine nichtleere Menge von Datenobjekten, F eine Menge von Funktionen und R eine Menge von Restriktionen und Bedingungen für Funktionen $f \in F$ ist.

6.1 Elementare abstrakte Datentypen

In diesem Abschnitt werden wir die elementaren Datentypen `Bit`, `Integer`, `Real`, `Boolean` und `Char` einführen, die in den meisten Rechenanlagen schon durch die Hardware implementiert sind.

Der Datentyp `Bit`

Der Datentyp `Bit` $G = (D, F, R)$ ist (exemplarisch ausführlich) wie folgt definiert:

- $D = \{0, 1\}$
- $F = \{\underline{\text{NULL}}, \underline{\text{EINS}}, \underline{\text{NOT}}, \underline{\text{AND}}, \underline{\text{OR}}\}$ mit $\underline{\text{NULL}}, \underline{\text{EINS}} : \{\} \rightarrow$D (nullstellig), $\underline{\text{NOT}} : D \rightarrow D$ und $\underline{\text{AND}}, \underline{\text{OR}} : D^2 \rightarrow D$.
- $R = \{R_i \mid 1 \leq i \leq 7\}$ mit

$$R_1 = (\underline{\text{NOT}}(\underline{\text{NULL}}) = \underline{\text{EINS}}),\ R_2 = (\underline{\text{NOT}}(\underline{\text{EINS}}) = \underline{\text{NULL}}),$$

$$R_3 = (\underline{\text{AND}}(\underline{\text{NULL}}, \underline{\text{NULL}}) = \underline{\text{NULL}}),\ R_4 = (\underline{\text{AND}}(\underline{\text{NULL}}, \underline{\text{EINS}}) = \underline{\text{NULL}}),$$

$$R_5 = (\underline{\text{AND}}(\underline{\text{EINS}}, \underline{\text{NULL}}) = \underline{\text{NULL}}),\ R_6 = (\underline{\text{AND}}(\underline{\text{EINS}}, \underline{\text{EINS}}) = \underline{\text{EINS}}),$$

$$R_7 = (\forall\, x, y \in D : \underline{\text{OR}}(x, y) = \underline{\text{NOT}}(\underline{\text{AND}}(\underline{\text{NOT}}(x), \underline{\text{NOT}}(y)))).$$

Der Datentyp `Integer`

Die Menge der Datenobjekte des Datentyp `Integer` $G = (D, F, R)$ ist nicht auf allen Rechenmaschinen die gleiche, sondern hängt von der jeweiligen Wortlänge ab. Normalerweise ist $D = \{-2^n, \ldots, 2^n - 1\}$ für ein n (normalerweise $n = 16, 32$). Die Menge der Funktionen beinhaltet:

- $+ : D^2 \to D$ (Addition)
- $- : D^2 \to D$ (Subtraktion)
- $* : D^2 \to D$ (Multiplikation)
- DIV $: D^2 \to D$ (ganzzahlige Division)
- MOD $: D^2 \to D$ (Rest bei ganzzahliger Division)

Neben diesen Funktion stehen gegebenfalls noch zur Verfügung:

- ABS $: D \to D$ (Betragsfunktion)
- SQR $: D \to D$ (Quadratfunktion)
- SUCC $: D \to D$ (Nachfolgerfunktion)
- PRED $: D \to D$ (Vorgängerfunktion)
- $- : D \to D$ (Vorzeichenwechsel)

Auf eine explizite Angabe der Regeln werden wir hier verzichten. Es sei nur kurz angegeben, daß SUCC$(x) := x + 1$ und PRED$(x) := x - 1$ definiert sind, falls $x + 1$ bzw. $x - 1$ in D liegen.

Der Datentyp `Real`

Der Datentyp `Real` wird verwendet, um Rechnungen mit reellen Zahlen durchführen zu können. Aufgrund ihrer Eigenschaft ist es jedoch nicht möglich, alle reellen Zahlen darzustellen. Daher beinhaltet der Datentyp `Real` nur eine endliche Teilmenge der reellen Zahlen und die auf ihm definierten Funktionen sind demzufolge auch nur Approximationen der entsprechenden arithmetischen Operationen in den reellen Zahlen. Neben den Grundoperationen $+$, $-$, $*$, $/$ sind oft auch noch folgende Operationen implementiert:

- SQRT $: D \to D$ (Quadratwurzel)
- POWER $: D^2 \to D$ (Potenzfunktion, $(x, y) \mapsto x^y$)
- LOG $: D \to D$ (natürlicher Logarithmus)
- EXP $: D \to D$ (Exponentialfunktion)
- SIN, COS, TAN, ARCSIN, ARCCOS, ARCTAN $: D \to D$ (Winkelfunktion und deren Umkehrung)

Der Datentyp `Boolean`

Der Datentyp Boolean ist ähnlich zum Datentyp `Bit`. Es gibt nur die Datenobjekte TRUE und FALSE. Es stehen die gängigen Booleschen ein- bzw. zweistelligen Funktionen AND, OR und NOT zur Verfügung.

Der Unterschied zwischen dem Datentyp `Bit` und `Boolean` besteht darin, daß sich der Wahrheitsgehalt von Vergleichen zwischen beliebigen Datentypen (durch sogenannte relationale Operatoren) durch den Datentyp `Boolean` ausdrücken läßt,d.h. es gilt:

$$=, <>, <, >, >= : D^2 \rightarrow \texttt{Boolean},$$

wobei D ein belieger Datentyp ist (dabei sei den Operatoren die offensichtlichen Bedeutungen zugewiesen).

Der Datentyp `Char`

Der Datentyp `Char` (Charakter) dient hauptsächlich zur Kommunikation zwischen Rechenmaschine und (menschlichem) Anwender. Die Datenmenge des Datentyps `Char` enthält neben den 26 (lateinischen) Großbuchstaben bzw. Kleinbuchstaben, den 10 (arabischen) Ziffern noch eine Anzahl von Satz-, Sonder- und Steuerzeichen. Auf dieser Datenmenge ist eine Ordnung mittels der Funktion ORD : `Char` $\rightarrow$ `Integer` definiert. Dabei gilt $A < B < \ldots < Z < a < \ldots < z$ und $0 < 1 < \ldots < 9$. Desweiteren gibt es noch die Funktionen

- CHR : `Integer` $\rightarrow$ `Char` definiert als Umkehrfunktion von ORD,
- SUCC : `Char` $\rightarrow$ `Char` definiert durch $\mathrm{SUCC}(c) = \mathrm{CHR}(\mathrm{ORD}(c) + 1)$, und
- PRED : `Char` $\rightarrow$ `Char` definiert durch $\mathrm{PRED}(c) = \mathrm{CHR}(\mathrm{ORD}(c) - 1)$.

6.2 Lineare Datentypen

Bevor wir komplexere Datentypen studieren können, werden wir Programmierkonstrukte kennen lernen, die uns die Implementation dieser Datentypen ermöglichen. Sie sind z.B. in der Programmiersprache PASCAL verfügbar.

Diese Programmierkonstrukte sind die Konstrukte des *Feldes*, des *Verbundes* und des *Zeigers*. Im folgenden sei `Type-Identifier` irgendein Datentyp z.B. ein unten definierter komplexer Datentyp oder ein elementarer Datentyp.

Definition 6.2 Ein Feld (Array) A von n Elementen eines Datentyps D ist vom Datentyp D^n. Es ist durch einen zusammenhängenden Speicherbereich realisiert, in den die Werte von n Variablen $A[1], \ldots, A[n]$ *gleichen Typs* gespeichert sind.

Eine Deklaration von Array-Typen in PASCAL sieht wie folgt aus:

`Array-Type = ARRAY[`$n_1..n_2$`,`$m_1..m_2$`,... ,`$r_1..r_2$`] OF Type-Identifier;`

Definition 6.3 Ein Verbund (Record) R ist eine Zusammenfassung einer festen Anzahl von Variablen $R.X, R.Y, \ldots$ möglicherweise unterschiedlichen Typs.

Eine Deklaration von Verbund-Typen in PASCAL sieht wie folgt aus:

```
Record-Type = RECORD
                var1 : Type-Identifier1;
                       ⋮
                varn : Type-Identifiern;
              END;
```

Definition 6.4 Ein Zeiger (Pointer) P ist eine Referenz auf eine Variable eines bestimmten Datentyps D, d.h. $P\uparrow$ ist eine Variable vom Datentyp D.

Eine Deklaration von Pointer-Typen in PASCAL sieht wie folgt aus:

```
Pointer-Type = ↑ Type-Identifier;
```

Durch die Funktion `new` wird einer Zeigervariablen eine neue Variable zugewiesen, die die Zeigervariable referenziert. Durch `dispose` wird diese Variable wieder entfernt. Ein Zeiger, dem noch keine Variable zugewiesen worden ist, zeigt auf den Wert `nil`. Durch Verwenden von Pointer können sogenannte *dynamische Variablen* während der Laufzeit des Programms erzeugt werden. Die Wirkungsweise von Pointer (Referenzen auf Variablen) im Gegensatz zu Variablen soll das folgende Beispiel erläutern.

Beispiel 6.5

```
TYPE PointInt : ↑Integer;
VAR p,q : PointInt;
BEGIN
  new(p); new(q);
  p↑:=4; q↑:=5;
  p↑:=q↑;   (* Nun ist p↑=q↑=5 *)
  q↑:=2;   (* Nun ist p↑=5, q↑=2 *)
  p:=q;   (* Nun ist p↑=q↑=2 *)
  q↑:=5;   (* Nun ist p↑=q↑=5 ! *)
END;
```

Wir werden im folgenden sogenannte *verkettete Listen* kennenlernen. Die Komponenten (Zellen) dieser Listen sind Verbunde, die Zeiger enthalten, die wieder auf Verbunde des gleichen Typs zeigen:

```
Celltype = RECORD
             element : Elementtyp;
             next : ↑Celltype
           END;
```

Im folgende werden wir eine erweiterte Version der Programmiersprache PASCAL verwenden. Dabei fügen wir die Kommandos `GOTO Output` und `Return(`e`)` hinzu. `GOTO Output` springt an einen *nichtnumerischen* Label `Output` und `Return(`e`)` gibt den Wert des Ausdruckes e zurück und verläßt die gerade aktive Prozedur bzw. Funktion. Beide Befehle können leicht in konstanter Zeit durch Standardbefehle simuliert werden. Anstatt Funktionen mit nicht veränderbaren Parametern zu benutzen, werden wir manchmal Funktionen mit variablen Parametern verwenden, d.h. Änderungen dieser Parameter wirken sich auch außerhalb der Funktion aus. Man nennt solche Parameterübergabe auch *call by reference* im Gegensatz zum normalen *call by value*, bei dem der Wert des Parameters in eine lokale Variable kopiert wird.

6.2.1 Listen

Wir werden jetzt den (abstrakten) Datentyp `Liste` einführen. Eine Liste L ist ein komplexer Datentyp und besteht aus einer endlichen Kette von Zeichen eines anderen Datentyps Σ. Eine Liste L der Länge n ist somit darstellbar als $L = a_1 \ldots a_n$, wobei a_i Elemente des gleichen Datentyps sind. Wir bezeichnen mit Λ das leere Wort, mit $\{\}$ die leere Liste und mit $\mathsf{End}(L)$ die Position, die a_n folgt. $\mathsf{End}(L)$ gibt das Ende der Liste L an. Wir definieren die folgende Menge an Operationen (Funktionen):

1. Insert: Die Funktion Insert ermöglicht das Einfügen eines Zeichens x in eine Liste L an einer beliebige Stelle p. Operationell ist Insert wie folgt definiert:

$$\begin{array}{rcl} \mathsf{Insert} : \Sigma \times \mathbb{N} \times \texttt{Liste} & \to & \texttt{Liste} \\ (x,p,L) & \mapsto & \begin{cases} a_1 \ldots a_{p-1} x a_p \ldots a_n & \text{falls } 1 \le p \le n \\ a_1 \ldots a_n x & \text{falls } p = \mathsf{End}(L) \\ \text{undefiniert} & \text{sonst} \end{cases} . \end{array}$$

2. Locate: Die Funktion Locate berechnet die erste Position p in der Liste L, so daß das p-te Listenelememt gleich x ist. Operationell ist Locate wie folgt definiert:

$$\begin{array}{rcl} \mathsf{Locate} : \Sigma \times \texttt{Liste} & \to & \mathbb{N} \\ (x,L) & \mapsto & \begin{cases} \min\{i \mid a_i = x\} & \text{falls } \{i \mid a_i = x\} \neq \emptyset \\ \mathsf{End}(L) & \text{sonst} \end{cases} . \end{array}$$

3. Retrieve: Die Funktion Retrieve liefert das p-te Element der Liste L. Operationell ist Retrieve wie folgt definiert:

$$\begin{array}{rcl} \mathsf{Retrieve} : \mathbb{N} \times \texttt{Liste} & \to & \Sigma \\ (p,L) & \mapsto & \begin{cases} a_p & \text{falls } 1 \le p \le n \\ \text{undefiniert} & \text{sonst} \end{cases} . \end{array}$$

4. Delete: Die Funktion Delete entfernt das p-te Element aus der Liste L. Operationell ist Delete wie folgt definiert:

$$\begin{array}{rcl} \mathsf{Delete} : \mathbb{N} \times \texttt{Liste} & \to & \texttt{Liste} \\ (p,L) & \mapsto & \begin{cases} a_1 a_2 \ldots a_{p-1} a_{p+1} \ldots a_n & \text{falls } 1 \le p \le n \\ \text{undefiniert} & \text{sonst} \end{cases} . \end{array}$$

5. Next: Die Funktion Next liefert den Index des dem p-ten Element folgenden Elementes der Liste L. Operationell ist Next wie folgt definiert:

$$\begin{array}{rcl} \mathsf{Next} : \mathbb{N} \times \texttt{Liste} & \to & \mathbb{N} \\ (p,L) & \mapsto & \begin{cases} p+1 & \text{falls } 1 \le p < n \\ \mathsf{End}(L) & \text{falls } p = n \\ \text{undefiniert} & \text{sonst} \end{cases} . \end{array}$$

6. Previous: Die Funktion Previous liefert den Index des dem p-ten Element vorausgehenden Elementes der Liste L. Operationell ist Previous wie folgt definiert:

$$\begin{array}{rcl} \mathsf{Previous} : \mathbb{N} \times \mathtt{Liste} & \to & \mathbb{N} \\ (p, L) & \mapsto & \begin{cases} p-1 & \text{falls } 1 < p \leq n \\ \text{undefiniert} & \text{sonst} \end{cases} \end{array} .$$

7. Makenull: Die Funktion Makenull liefert eine leere Liste und den Index auf das Ende der Liste. Operationell ist Makenull wie folgt definiert:

$$\begin{array}{rcl} \mathsf{Makenull} : \mathtt{Liste} & \to & \mathtt{Liste} \times \mathbb{N} \\ L & \mapsto & (\{\}, \mathsf{End}(\{\})) \end{array}$$

8. Printlist: Die Funktion Printlist gibt die Liste L aus: Diese Funktion ist eine Funktion mit Seiteneffekt, d.h. ihre operationelle Funktion ist uninteressant. Als Seiteneffekt wird die Liste ausgegeben.

9. First: Die Funktion First liefert einen Verweis auf das erste Element einer Liste. Operationell ist First wie folgt definiert:

$$\begin{array}{rcl} \mathsf{First} : \mathtt{Liste} & \to & \mathbb{N} \\ L & \mapsto & \begin{cases} \text{“erste Position in } L\text{”} & \text{falls } L \neq \{\} \\ \text{undefiniert} & \text{sonst} \end{cases} \end{array} .$$

Implementation mittels Arrays

Als erstes werden wir PASCAL Prozeduren angeben, die eine Liste mittels des Konstruktes ARRAY implementiert. Als Abkürzung sei `Elementtype` der Datentyp der Elemente der Liste (früher Σ). Dieser selbst kann beliebig kompliziert sein (z.B. durch Verbunde oder Arrays geschachtelt definiert).

In unser Implementation ist eine Liste dann dargestellt durch:

```
Liste = RECORD
          elements : ARRAY[1..Maxlength] of Elementtype;
          last : Position;
        END;
```

wobei `Maxlength` eine geeignet definierte Konstante ist und Position der Indextyp ist (hier Position = Integer).

Bevor wir zu den Implementationen der Liste mittels Arrays oder verketteter Listen kommen, möchten wir zeigen, daß wir Listenmanipulation ganz losgelöst von der Implementation betrachten können, d.h. auf einem abstrakten Niveau. Wir wollen dies mit der Funktion Purge tun, die die Liste von doppelt vorkommenden Elementen “reinigt”. Dazu sei Same eine Funktion, die feststellt, ob zwei Elemente vom Type `Elementtype` gleich sind.

```
PROCEDURE Purge(VAR L:Liste);
  VAR p,q:Position;
  BEGIN
```

```
    p:=First(L);
    WHILE (p<>End(L)) DO BEGIN
      q:=Next(p,L);
      WHILE (q<>End(L)) DO
        IF Same(Retrieve(p,L),Retrieve(q,L)) THEN Delete(q,L)
          ELSE q:=Next(q,L);
    END;p:=Next(p,L);
  END;
```

Dabei ist

```
FUNCTION End(VAR L:Liste);
  BEGIN
    Return(L.last+1)
  END;
```

Die restlichen Funktionen sind wie folgt implementiert:

1. ```
   PROCEDURE Insert(x:Elementtype;p:Position;VAR L:Liste);
     VAR q:Position;
     BEGIN
       IF L.last>=Maxlength THEN Error
         ELSE IF (p>L.last+1) OR (p<1) THEN Error
           ELSE BEGIN
             FOR q:=L.last DOWNTO p DO
               L.elements[q+1]:=L.elements[q];
             L.last:=L.last+1;
             L.elements[p]:=x;
           END;
     END;
   ```
   Die Zeitkomplexität von Insert beträgt $O(|L|)$.

2. ```
   FUNCTION Locate(x:Elementtype;VAR L:Liste):Position;
     VAR q:Position;
     BEGIN
       FOR q:=1 TO L.last DO
         IF L.elements[q]=x THEN Return(q);
       Return(L.last+1);
     END;
   ```
   Die Zeitkomplexität von Locate beträgt $O(|L|)$.

3. ```
   FUNCTION Retrieve(p:Position;VAR L:Liste):Elementtype;
     BEGIN
       IF (p<1) OR (p>L.last) THEN Error
       Return(L.elements[p]);
     END;
   ```
   Die Zeitkomplexität von Retrieve beträgt $O(1)$.

4. ```
   PROCEDURE Delete(p:Position; VAR L:List);
   ```

```
VAR q:Position;
BEGIN
  IF (p>L.last) OR (p<1) THEN Error
    ELSE BEGIN
      L.last:=L.last-1;
      FOR q:=p TO L.last DO
        L.elements[q]:=L.elements[q+1]
    END;
END;
```

Die Zeitkomplexität von Delete beträgt $O(|L|)$.

5.
```
FUNCTION Next(p:Position;VAR L:Liste):Position;
  BEGIN
    IF (p<1) OR (p>L.last) THEN Error;
    IF p=L.last THEN Return(L.last+1);
    Return(p+1);
  END;
```

Die Zeitkomplexität von Next beträgt $O(1)$.

6.
```
FUNCTION Previous(p:Position;VAR L:Liste):Position;
  BEGIN
    IF (p<=1) OR (p>L.last) THEN Error;
    Return(p-1);
  END;
```

Die Zeitkomplexität von Previous beträgt $O(1)$.

7.
```
PROCEDURE Makenull(VAR L:Liste);
  BEGIN
    L.last:=0;
  END;
```

Die Zeitkomplexität von Makenull beträgt $O(1)$.

8.
```
PROCEDURE Printlist(VAR L:Liste);
  VAR q:Postion;
  BEGIN
    FOR q=1 TO L.last DO
      Print(L.Element[q]);
  END;
```

Die Zeitkomplexität von Printlist beträgt $O(n)$.

9.
```
FUNCTION First(VAR L:Liste):Position;
  VAR q:Postion;
  BEGIN
    IF L.last=0 THEN Return(0);
    Return(1);
  END;
```

Die Zeitkomplexität von First beträgt $O(1)$.

Damit hat die Funktion Purge eine Zeitkomplexität von $O(n^2)$.

Implementation mittels verketteter Listen

Zunächst müssen wir den Zeigertypen und die Listenkomponenten deklarieren:

```
Listcomponent = RECORD
                  element: Elementtype;
                  next: List;
                END;
List = ↑ Listcomponent;
Position = ↑ Listcomponent;
```

Eine Liste L ist jetzt durch einen Zeiger auf den Anfang einer verketteten Liste gegeben. Dabei ist das erste Element ein dummy-Element, so daß das erste Element der Liste durch `L↑.next↑.element` referenziert wird. Entsprechend zeigt ein Zeiger statt auf das eigentliche Element auf den Vorgänger. Dies erleichtert die Implementation, wie wir gleich sehen werden.

Wir werden uns auf die Implementationen der Funktionen End, Insert, Locate und Delete beschränken.

- ```
  FUNCTION End(L:List):Position;
    VAR q:Position;
    BEGIN
      q:=L;
      WHILE q↑.next<>nil DO
        q:=q↑.next;
      Return(q);
    END;
  ```

- ```
  PROCEDURE Insert(x:Elementtype;p:Position;VAR L:List)
    VAR q:Position;
    BEGIN
      q:=p↑.next;
      new(p↑.next);
      p↑.next↑.element:=x;
      p↑.next↑.next:=q;
    END;
  ```

- ```
  FUNCTION Locate(x:Elementtype;VAR L:List):Position;
    VAR p:Position;
    BEGIN
      p:=L;
      WHILE p↑.next<>nil DO
        IF p↑.next↑.element=x THEN Return(p)
          ELSE p:=p↑.next;
      Return(p);
    END;
  ```

- ```
  PROCEDURE Delete(p:Position;VAR L:List);
    BEGIN
      IF p↑.next=nil THEN Error;
  ```

```
    p↑.next:=p↑.next↑.next;
  END;
```

In der folgenden Tabelle haben wir die Laufzeiten der wichtigsten Funktionen in der Array- und der Zeigerimplementation gegenüber gestellt.

	End	Insert	Locate	Delete
Array	$O(1)$	$O(n)$	$O(n)$	$O(n)$
Zeiger	$O(n)$	$O(1)$	$O(n)$	$O(1)$

6.2.2 Keller

In diesem Abschnitt werden wir den Datentyp des Kellers oder Stapels (engl. Stack) einführen. Es handelt sich hierbei um eine Liste, bei der Einfügungen und Löschungen nur an einer (derselben) Seite möglich sind. Diese Liste korrespondiert dann zu einem Stapel, auf den nur ein Element oben dazu gelegt werden kann, oder von dem nur das oberste Element entfernt werden darf. Daher nennt man einen Keller auch eine LIFO (Last In First Out) Struktur.

Wie bei dem Datentyp `Liste` können wir sowohl eine Array als auch eine Zeigerimplementation angeben. Wir werden uns jedoch auf die Arrayimplementation beschränken. Analog zur Implementation des Datentyps `Liste` definieren wir

```
Stack := RECORD
           elements : ARRAY[1..Maxlength] of Elementtype;
           top : Position;
         END;
```

wobei `Maxlength` eine geeignet definierte Konstante ist und Position der Indextyp ist (hier Position = Integer).

Operationell werden wir die folgenden Funktionen einführen, von denen wir direkt eine Array-Implementation angeben. Dabei "wächst" unser Stack S nach unten, d.h. es gilt `S.top=Maxlength+1` genau dann, wenn S leer ist.

- Makenull löscht den Stack S.

```
PROCEDURE Makenull(VAR S:Stack);
  BEGIN
    S.top:=Maxlength+1;
  END;
```

- Empty testet, ob der Stack S leer ist.

```
FUNCTION Empty(VAR S:Stack):Boolean;
  BEGIN
    IF S.top>Maxlength THEN Return(TRUE)
      ELSE Return(FALSE);
  END;
```

- Top liefert das oberste Element des Stacks S.

```
FUNCTION Top(VAR S:Stack):Elementtype;
  BEGIN
    IF Empty(S) THEN Error
      ELSE Return(S.elements[S.top]);
  END;
```

- Pop löscht das oberste Element des Stacks S.

```
PROCEDURE Pop(VAR S:Stack);
  BEGIN
    IF Empty(S) THEN Error
      ELSE S.top:=S.top+1;
  END;
```

- Push legt ein neues Element oben auf den Stack S.

```
PROCEDURE Push(x:Elementtype;VAR S:Stack);
  BEGIN
    IF S.top=1 THEN Error;
    S.top:=S.top-1;
    S.elements[S.top]:=x;
  END;
```

6.2.3 Schlange

Nun kommen wir zum Gegenstück des Kellers, zum Datentyp Schlange. Eine Schlange (engl:queue) ist eine FIFO (First In First Out) Struktur. Einfügen neuer Elemente ist nur am Ende einer Schlange erlaubt, während das Löschen nur am Anfang passiert (man vergleiche dies mit einer Warteschlange beispielsweise am Fahrkartenschalter). Wir werden eine Pointerimplementation angeben. Dazu seien

```
Celltype = RECORD
             element : Elementtype;
             next : ↑Celltype
           END;
```

und

```
Queue = RECORD
          front, rear:↑Celltype;
        END;
```

- Makenull erstellt eine leere Schlange.

```
PROCEDURE Makenull (VAR Q:Queue);
  BEGIN
```

```
    new(Q.front);
    Q.front↑.next:=nil;
    Q.rear:=Q.front;
  END;
```

- Empty testet ob eine Schlange leer ist.

```
FUNCTION Empty (VAR Q:Queue): Boolean;
  BEGIN
    IF Q.front=Q.rear THEN Return(TRUE)
      ELSE Return(FALSE)
  END;
```

- Front liefert das vorderste Element der Schlange.

```
FUNCTION Front (VAR Q:Queue):Elementtype;
  BEGIN
    IF Empty(Q) THEN Error
      ELSE Return(Q.front↑.next↑.element)
  END;
```

- Enqueue fügt eine neues Element am Ende der Schlange an.

```
PROCEDURE Enqueue (x:Elementtype;VAR Q:Queue);
  BEGIN
    new(Q.rear↑.next);
    Q.rear:=Q.rear↑.next;
    Q.rear↑.element:=x;
    Q.rear↑.next:=nil;
  END;
```

- Dequeue löscht das vorderste Element der Schlange.

```
PROCEDURE Dequeue (VAR Q:Queue);
  BEGIN
    IF Empty(Q) THEN Error
      ELSE Q.front↑.next:=Q.front↑.next↑.next;
  END;
```

6.3 Datentypen aus der Graphentheorie

In diesem Kapitel werden wir den Datentyp des gerichteten Graphen kennenlernen. Der Begriff des Graphen ist schon in Abschnitt 1.2 eingeführt worden. Wir geben die folgenden Darstellungsarten für Graphen an:

- *Adjazenzmatrix.* Ein Graph $G = (V, E)$ ist durch eine Boolesche $|V| \times |V|$ Matrix A mit

$$A_{i,j} = \begin{cases} \text{TRUE} & \text{falls } (i,j) \in E \\ \text{FALSE} & \text{sonst} \end{cases}$$

dargestellt. Ein symmetrischer Graph korrespondiert dann zu einer symmetrischen Matrix. Der Größenaufwand dieser Darstellung beträgt $|V|^2$, kann also ggf. größer sein als die Größe von G $(= |V| + |E|)$.

- *beschriftete Adjazenzmatrix.* Ein kantengelabelter Graph $G = (V, E, l)$ ist durch folgende $|V| \times |V|$ Matrix dargestellt:

$$A_{i,j} = \begin{cases} l((i,j)) & \text{falls } (i,j) \in E \\ \# & \text{sonst} \end{cases},$$

  wobei # eine leere Markierung darstellt. Es gilt sinngemäß dasselbe wie für Adjazenzmatrizen.

- *Adjazenzlisten.* Ein Graph $G = (V, E)$ wird durch $|V|$ Listen dargestellt (diese Listen bilden wieder eine Liste L), wobei für jedes v eine Liste die Menge der Knoten repräsentiert, die mit v verbunden sind, d.h. $L_{i,j}$ ist der j-te Knoten, der mit dem Knoten i verbunden ist. Vorteil dieser Darstellung ist der optimale Speicherverbrauch von $\Theta(|V| + |E|)$.

Nun können wir den Datentyp `Graph` definieren. Wir stellen hier die gleichen Operationen wie bei Datentyp `Liste` zur Verfügung (eine Liste der Knoten und eine Liste der Kanten). Zusätzlich definieren wir die folgenden Funktionen, die die Knoten- und Kantenliste miteinander in Verbindung bringt. Dabei nehmen wir an, daß die Knoten numeriert sind.

- First berechnet zu einem Knoten v den Knoten w mit kleinster Nummer, der mit v verbunden ist. Falls kein solcher Knoten existiert, ist die Funktion undefiniert.

- Next berechnet zu einem Knoten v und einem mit v verbundenen Knoten w den nächstgrößeren Knoten u, der mit v verbunden ist. Falls kein solcher Knoten existiert, ist die Funktion undefiniert.

- Vertex berechnet zu einem Knoten v und einer Zahl i den i-t kleinsten Knoten, der mit v verbunden ist. Falls kein solcher Knoten existiert, ist die Funktion undefiniert.

Eine geeignete Implementation mittels der Darstellung der Adjazenzmatrizen benötigt für alle 3 Funktionen $O(n)$ Zeit (dabei ist der Test, ob zwei Knoten miteinander verbunden sind, in konstanter Zeit ausführbar). Die Implementation mittels der Darstellung der Adjazenzliste benötigt für alle Funktionen (auch Test auf Verbundenheit) $O(n)$ Zeit.

6.3.1 Bäume

Wie wir schon im Abschnitt 1.2 gesehen haben, sind Bäume sehr wichtige und sehr einfache Graphen. Da ein Baum mit n Knoten genau $n - 1$ Kanten enthält, ist die Darstellung Adjazenzmatrix derart ungeeignet, daß wir sie hier ganz außer Acht lassen. Im folgenden gehen wir davon aus, daß wir einen Baum mit Wurzel 1 haben.

Neben der Adjazenzlistenrepräsentation verfügen wir noch über die sehr kompakte "Vaterrepräsentation". Diese beruht auf der Tatsache, daß in einem Baum mit einer Wurzel jeder Knoten einen eindeutigen Vaterknoten besitzt. Die Vaterrepräsentation ist also durch ein Array A realisiert, in dem der Eintrag $A[i]$ gerade der Vater von i ist. Die Wurzel, der einzige Knoten, der keinen Vater besitzt, soll mittels der Vaterrepräsentation der Wert 0 zugewiesen werden.

Beispiel 6.6 Der Baum aus Abbildung 1.2 aus Abschnitt 1.2 hat die Vaterrepräsentation $A = (0, 1, 1, 2, 2, 3, 3, 5)$.

Die Numerierung der Knoten impliziert direkt eine Ordnung auf der Menge der Söhne eines Knoten. Der linkeste Sohn sei dabei derjenige Sohn mit der kleinsten Nummer, und der rechte Bruder eines Knotens derjenige Bruder mit der nächstgrößeren Nummer. Wir definieren die folgenden Operationen:

- Parent(n, T) gibt den Vater des Knotens n im Baum T an (undefiniert, falls n die Wurzel von T ist).
- Leftmost-Child(n, T) gibt den linkesten Sohn von n in T an (undefiniert, falls n ein Blatt ist).
- Right-Sibling(n, T) gibt den rechten Bruder von n in T an (undefiniert, falls es keinen rechten Bruder von n gibt).
- Label(n, T) gibt die Beschriftung des Knotens n in T an.
- Create i $(v, T_1, \ldots, T_i)$ liefert einen Baum mit Wurzel r, die mit v beschriftet ist, und die als Söhne die Wurzeln der Bäume $T_1, \ldots, T_i$ hat.
- Root(T) gibt die Wurzel des Baums T an.
- Makenull(T) liefert einen leeren Baum.

Das Problem, eine systematische Auflistung der Knoten eines Graphen zu erhalten, kann im Spezialfall von Bäumen leicht gelöst werden. Der Trick hierbei ist die rekursive Struktur eines Baumes: ein Baum T besteht aus einer Wurzel r und den durch die Söhne von r gewurzelten Unterbäumen von T. Diese Struktur legt die folgenden sogenannten Durchmusterungsverfahren nahe:

- Preorder-Verfahren: durchmustere zuerst die Wurzel und dann die Teilbäume von links nach rechts.
- *Inorder-Verfahren:* durchmustere zuerst den linkesten Teilbaum, dann die Wurzel und dann die restlichen Teilbäume (wenn vorhanden).
- *Postorder-Verfahren:* durchmustere zuerst die Teilbäume von links nach rechts und dann die Wurzel.

Beispiel 6.7 Die Verfahren angewandt auf den Baum in Abbildung 1.2 erzeugen die folgenden Auflistungen der Knoten:

- Preorder: $(1, 2, 4, 5, 8, 3, 6, 7)$
- Inorder: $(4, 2, 8, 5, 1, 6, 3, 7)$
- Postorder: $(4, 8, 5, 2, 6, 7, 3, 1)$

Kapitel 7

Semantik von Programmiersprachen und das λ-Kalkül

Wir wollen untersuchen, welche Komplikationen bei der Betrachtung der Semantik von Programmiersprachen auftreten, und wie diese zu lösen sind. Wir werden dies an Hand einer sehr übersichtlichen Programmiersprache mit wenigen Konstrukten erläutern. Hierbei werden schon viele wesentliche Aspekte berücksichtigt werden, die bei der semantischen Analyse von Programmiersprachen auftauchen können. Um einen ersten Eindruck zu vermitteln, werden wir neben der Syntax auch eine intuitive Semantik angeben. Im Verlauf dieses Kapitels werden wir dann die Semantik immer weiter verfeinern, Lücken schließen und mathematisch präziser werden. Der Inhalt dieses Kapitels ist im wesentlichen dem Buch von Gordon [Gor79] entnommen.

7.1 Die Sprache "TINY"

Die Programmiersprache TINY stellt zwei wesentliche Konstrukte zur Verfügung: Ausdrücke und Befehle. Beide können Identifier, die in Beispiel 2.11 eingeführt wurden, beinhalten. Damit haben wir schon die drei wichtigsten syntaktischen Kategorien (Domains):

$$\begin{aligned} \underline{\mathrm{I}} &= \{I \mid I \text{ ist ein Identifier}\} \\ \underline{\mathrm{E}} &= \{E \mid E \text{ ist ein Ausdruck}\} \\ \underline{\mathrm{C}} &= \{C \mid C \text{ ist ein Befehl}\}. \end{aligned}$$

Wir können die erlaubten Ausdrücke und Kommandos mit einer BNF beschreiben:

$$\begin{aligned} <\underline{\mathrm{E}}> \quad ::= \quad & \underline{0} \mid \underline{1} \mid \text{true} \mid \text{false} \mid \text{read} \mid <\underline{\mathrm{I}}> \mid \text{not} \ <\underline{\mathrm{E}}> \mid <\underline{\mathrm{E}}> = <\underline{\mathrm{E}}> \mid \\ & <\underline{\mathrm{E}}> + <\underline{\mathrm{E}}> \mid (<\underline{\mathrm{E}}>) \\ <\underline{\mathrm{C}}> \quad ::= \quad & <\underline{\mathrm{I}}> ::= <\underline{\mathrm{E}}> \mid \text{output} <\underline{\mathrm{E}}> \mid \text{if} \ <\underline{\mathrm{E}}> \ \text{then} \ <\underline{\mathrm{C}}> \ \text{else} \ <\underline{\mathrm{C}}> \mid \\ & \text{while} \ <\underline{\mathrm{E}}> \ \text{do} \ <\underline{\mathrm{C}}> \mid <\underline{\mathrm{C}}>; <\underline{\mathrm{C}}> \mid (<\underline{\mathrm{C}}>) \end{aligned}$$

Wie wir bei der Betrachtung von Turing-Maschinen und RAMs gesehen haben, bezieht sich die Semantik auf die Beschreibung einer Konfigurationsänderung. Daher müssen wir zunächst darauf eingehen, aus welchen Komponenten eine Konfigurationsbeschreibung besteht:

- der Speicher, d.h. die Belegung der Variabeln mittels der Speicherfunktion m.

- die Eingabe (eine Folge von Eingabesymbolen)
- die Ausgabe (eine Folge von Ausgabesymbolen)

Desweiteren müssen wir noch einen Fehlerzustand einfügen, falls Befehle oder Ausdrücke auftauchen, die zwar syntaktisch korrekt aber semantisch nicht verarbeitbar sind (z.B.: true+3). Wir werden die Symbole $\mathbb{E}$ und $\mathbb{C}$ für die semantische Funktionen für Ausdrücke bzw. Befehle verwenden. Anhand der folgenden naiven Beschreibung der semantischen Funktionen sehen wir schon die Notwendigkeit zu einer Formalisierung, die dann eine Präzisierung der Semantik ermöglicht.

(E1) $\mathbb{E}[\underline{0}] = 0$, $\mathbb{E}[\underline{1}] = 1$;

(E2) $\mathbb{E}[\text{true}] = \text{TRUE}$, $\mathbb{E}[\text{false}] = \text{FALSE}$;

(E3) $\mathbb{E}[\text{read}]$ ist das nächste Zeichen der Eingabefolge, falls diese nicht leer ist. Falls die Eingabefolge leer ist, so erzeugt dies einen Fehler.

(E4) $\mathbb{E}[I]$ ist der an I gebundene Wert. Falls I ungebunden ist, so erzeugt dies einen Fehler.

(E5) $\mathbb{E}[\text{not } E] = \neg\mathbb{E}[E]$, falls $\mathbb{E}[E] \in \{\text{TRUE}, \text{FALSE}\}$, sonst wird ein Fehler erzeugt.

(E6) $\mathbb{E}[E_1 = E_2]$ ist TRUE, falls $\mathbb{E}[E_1] = \mathbb{E}[E_2]$ und beide keinen Fehler erzeugt haben. Sonst ist der Ausdruck FALSE.

(E7) $\mathbb{E}[E_1 + E_2] = \mathbb{E}[E_1] + \mathbb{E}[E_2]$, falls $\mathbb{E}[E_1], \mathbb{E}[E_2] \in \mathbb{N}_0$, sonst erzeugt dies einen Fehler.

(C1) $\mathbb{C}[I := E] = (m(I) := \mathbb{E}[E])$, falls $\mathbb{E}[E]$ keinen Fehler erzeugt hat, d.h. der Identifier I wird an den Wert von E gebunden. Ein vorher gebundener Wert wird überschrieben.

(C2) $\mathbb{C}[\text{output}(E)]$ bewirkt, daß an die Ausgabefolge der Wert $\mathbb{E}[E]$ angefügt wird.

(C3) $\mathbb{C}[\text{if } E \text{ then } C_1 \text{ else } C_2]$ ist $\mathbb{C}[C_1]$, falls $\mathbb{E}[E] = \text{TRUE}$, $\mathbb{C}[C_2]$, falls $\mathbb{E}[E] = \text{FALSE}$, sonst wird ein Fehler erzeugt.

(C4) $\mathbb{C}[\text{while } E \text{ do } C]$: falls $\mathbb{E}[E] = \text{TRUE}$ so wird C ausgeführt gefolgt von while E do C. Falls $\mathbb{E}[E] = \text{FALSE}$, so wird nichts ausgeführt. Falls $\mathbb{E}[E] \notin \{\text{TRUE}, \text{FALSE}\}$ wird ein Fehler erzeugt.

(C5) $\mathbb{C}[C_1; C_2]$ bewirkt, daß zunächst C_1 und dann C_2 ausgeführt wird.

7.2 Semantik von TINY

Wir werden jetzt die oben aufgeführten Interpretationen von Ausdrücken und Befehlen präzisieren. Wir hatten schon gesehen, daß jeder Schritt eines Programmes eine Veränderung des Speichers sowie der Ein- und Ausgabe hervorruft. Eine Konfiguration bzw. ein Zustand ist daher (zumindests) von diesen drei Komponenten abhängig. Im folgenden sei NUM eine Menge von Symbolen für die natürlichen Zahlen und BOOL eine Menge von Symbolen für die Wahrheitswerte. Wir können die folgenden sogenannten Bereichsgleichungen aufstellen:

$$\begin{aligned} \text{VALUE} &= \text{NUM} \cup \text{BOOL} \\ \text{INPUT} &= \text{VALUE}^* \end{aligned}$$

$$\begin{aligned} \text{OUTPUT} &= \text{VALUE}^* \\ \text{MEMORY} &= \underline{\text{I}} \to \text{VALUE} \cup \{\text{unbound}\} \\ \text{STATE} &= \text{MEMORY} \times \text{INPUT} \times \text{OUTPUT}, \end{aligned}$$

wobei unbound $\notin$ VALUE ein spezieller Wert ist, d.h. $mI = \text{unbound}$, falls der Identifier I an keinen Wert gebunden ist. Ein Zustand (STATE) ist somit also ein Tripel (m, i, o) mit $m \in$ MEMORY, $i \in$ INPUT und $o \in$ OUTPUT.

Als abkürzende Notation werden wir $[A \to B]$ für $\{f \mid f : A \to B\}$ und $[A + B]$ für die *disjunkte* Vereinigung von A und B verwenden.

Nun können wir die semantischen Funktionen $\mathbb{E}$ und $\mathbb{C}$ angeben. Der Wertebereich von $\mathbb{E}$ ist die Menge der Bedeutungen von Ausdrücken (Denotations of Expressions (DOE)), der von $\mathbb{C}$ ist die Menge der Bedeutungen von Befehlen (Denotations of Commands (DOC)). Damit sind

$$\begin{aligned} \mathbb{E} : \underline{\text{E}} &\to \text{DOE} \\ \mathbb{C} : \underline{\text{C}} &\to \text{DOC}. \end{aligned}$$

Wir müssen nun noch DOE und DOC genauer spezifizieren. Um alle möglichen Fälle abzudecken, definieren wir wie folgt:

$$\begin{aligned} \text{DOE} &= [\text{STATE} \to [[\text{VALUE} \times \text{STATE}] + \{\text{error}\}]] \\ \text{DOC} &= [\text{STATE} \to [\text{STATE} + \{\text{error}\}]] \end{aligned}$$

Damit sind $\mathbb{E}$ und $\mathbb{C}$ eigentlich mehrstellige Funktionen. Wir schreiben aber auch statt $\mathbb{E}(E, s)$ einfach $\mathbb{E}[E]s$ und statt $\mathbb{C}(C, s)$ einfach $\mathbb{C}[C]s$.

Da Fehler auftreten können und sich diese fortpflanzen, wird jeder semantische Ausdruck durch viele Bedingung und Fallunterscheidungen geprägt sein. Daher führen wir die sogenannten McCarthyschen bedingten Ausdrücke ein, die die folgende Form haben:

$$(p_1 \to e_1, p_2 \to e_2, \ldots, p_n \to e_n)$$

Dieser Ausdruck ist äquivalent zu der folgenden `if-then` Struktur.

```
if p1 then e1
   else if p2 then e2
      ⋮      ⋮
           else if pn then en
```

Falls das letzte p_n fehlt, so tritt der letzte Fall e_n ein, falls alle p_i falsch sind, d.h. die letzte Zeile lautet dann: `else` e_n. Dabei lassen wir auch der Übersichthalber die äußeren Klammern weg.

Mit diesen mathematischen Hilfsmitteln können wir die semantischen Funktionen $\mathbb{E}$ und $\mathbb{C}$ wie folgt spezifizieren.

(E1) $\mathbb{E}[\underline{0}]s = (0, s)$, $\mathbb{E}[\underline{1}]s = (1, s)$.

(E2) $\mathbb{E}[\text{true}]s = (\text{TRUE}, s)$, $\mathbb{E}[\text{false}]s = (\text{FALSE}, s)$.

(E3) $\mathbb{E}[\text{read}](m,i,o) = (\text{null } i \to \text{error}, (\text{hd } i, (m, \text{tl } i, o)))$

mit

$$\text{null } i = \begin{cases} \text{TRUE} & \text{falls } i = \Lambda \\ \text{FALSE} & \text{sonst} \end{cases}$$

und hd $i = i_1$ und tl $i = i_2 \ldots i_n$ für $i = i_1 i_2 \ldots i_n$ $(i \neq \Lambda)$. Die Erklärung für diese Definition ist wie folgt: wenn das Eingabeband leer ist, so bekommen wir bei der Auswertung von read einen Fehler. Andernfalls ist der Wert das erste Zeichen der noch vorhanden Eingabe und die Eingabe wird um dieses Zeichen verkürzt.

(E4) $\mathbb{E}[I](m,i,o) = (mI = \text{unbound} \to \text{error}, (mI, (m,i,o)))$.

$\mathbb{E}[I]$ liefert den an den Identifier I gebundenen Wert im Speicher (mI), dabei ändert sich der Zustand nicht. Ist an I kein Wert gebunden, d.h. gilt $mI = \text{unbound}$, so verursacht dies einen Fehler.

(E5) $\mathbb{E}[\text{not } E]s = ((\mathbb{E}[E]s = (v,s')) \to (\text{isBool } v \to (\neg v, s'), \text{error}), \text{error})$

mit

$$\text{isBool } x = \begin{cases} \text{TRUE} & \text{falls } x \in \text{BOOL} \\ \text{FALSE} & \text{sonst} \end{cases}$$

$\mathbb{E}(\text{not } E)$ gibt den negierten Wert von $\mathbb{E}[E]$, falls $\mathbb{E}[E]$ ein Boolescher Wert ist, ansonsten verursacht dies einen Fehler. Durch die Auswertung von E im Zustand s ändert sich der Zustand von s in s'.

(E6) $\mathbb{E}[E_1 = E_2]s = (\mathbb{E}[E_1]s = (v_1, s_1)) \to ((\mathbb{E}[E_2]s_1 = (v_2, s_2)) \to (v_1 = v_2, s_2), \text{error}), \text{error}.$

$\mathbb{E}[E_1 = E_2]s$ wertet zunächst den Ausdruck E_1 im Zustand s aus, wodurch der Zustand s_1 entsteht. Dann wird E_2 in s_1 ausgewertet, dies gibt den Zustand s_2. Falls eine der Auswertungen einen Fehler erzeugt hat, so erzeugt dies einen Fehler, sonst werden die beiden Werte v_1 und v_2 im Zustand s_2 verglichen und das Resultat zurückgegeben.

(E7) $\mathbb{E}[E_1 + E_2]s = (\mathbb{E}[E_1]s = (v_1, s_1)) \to ((\mathbb{E}[E_2]s_1 = (v_2, s_2)) \to$
$(\text{isNum } v_1 \wedge \text{isNum } v_2 \to (v_1 + v_2, s_2), \text{error}), \text{error}), \text{error}$

mit

$$\text{isNum } x = \begin{cases} \text{TRUE} & \text{falls } x \in \text{NUM} \\ \text{FALSE} & \text{sonst} \end{cases}$$

Die Semantik ist analog zu **(E6)**.

(C1) $\mathbb{C}[I := E]s = ((\mathbb{E}[E]s = (v, (m,i,o))) \to (m[I := v], i, o), \text{error})$

mit

$$(m[I := v])I' = \begin{cases} v & \text{falls } I' = I \\ mI' & \text{sonst} \end{cases}$$

Die Auswertung von $I := E$ bewirkt eine Zustandsänderung, indem der Identifier I an den Wert von E gebunden wird.

(C2) $\mathbb{C}[\text{output } E]s = ((\mathbb{E}[E]s = (v, (m,i,o))) \to (m, i, v.o), \text{error})$

Der Zustand der Ausgabefolge wird geändert, indem der Wert von E an den Anfang der Ausgabefolge gehängt wird $(o \to v.o)$.

(C3) $\mathbb{C}[\text{if } E \text{ then } C_1 \text{ else } C_2]s = ((\mathbb{E}[E]s = (v, s')) \to (\text{isBool } v \to (v \to \mathbb{C}[C_1]s', \mathbb{C}[C_2]s'), \text{error}), \text{error})$

Wenn die Auswertung von E in s den Wert (v, s') ergibt, so wird C_1 in s' ausgeführt, falls der Wert v von E wahr ist, ansonsten wird C_2 in s' ausgeführt. Falls v kein Boolescher Wert ist, oder ein Fehler bei der Auswertung von E auftrat, so ist das Ergebnis ebenfalls ein Fehler.

(C4) $\mathbb{C}[\text{while } E \text{ do } C]s = ((\mathbb{E}[E]s = (v, s')) \to (\text{isBool } v \to (v \to ((\mathbb{C}[C]s' = s'') \to \mathbb{C}[\text{while } E \text{ do } C]s'', \text{error}), s'), \text{error}), \text{error})$

Falls die Auswertung von E in s den Wert (v, s') ergibt und v wahr ist, so wird C in s' ausgeführt. Dies führt zum Zustand s'', in dem dann der Befehl while E do C erneut ausgeführt wird. Dies geschieht solange, bis entweder der Wert von E falsch wird oder ein Fehler auftritt. Man beachte hierbei die rekursive Definition!

(C5) $\mathbb{C}[C_1; C_2]s = (\mathbb{C}[C_1]s = \text{error}) \to \text{error}, \mathbb{C}[C_2](\mathbb{C}[C_1]s)$

Die Befehle C_1 und C_2 werden nacheinander ausgeführt, wobei C_2 im Zustand $s' = (\mathbb{C}[C_1]s)$ nach der Ausführung von C_1 ausgeführt wird.

Wir werden nun TINY erweitern und auch parameterlose Prozeduren, die nur den Zustand ändern, als Ausdrücke zulassen. Wir erweiterten die BNF wie folgt:

$$\begin{aligned} < \underline{\text{E}} > &::= \text{proc} < \underline{\text{C}} > \\ < \underline{\text{C}} > &::= < \underline{\text{I}} > . \end{aligned}$$

Wir müssen auch unsere Bereichsgleichungen erweitern:

$$\begin{aligned} \text{VALUE} &= \text{NUM} + \text{BOOL} + \text{PROC} \\ \text{INPUT} &= \text{VALUE}^* \\ \text{OUTPUT} &= \text{VALUE}^* \\ \text{MEMORY} &= \underline{\text{I}} \to [\text{VALUE} + \{\text{unbound}\}] \\ \text{STATE} &= \text{MEMORY} \times \text{INPUT} \times \text{OUTPUT} \\ \text{PROC} &= \text{STATE} \to [\text{STATE} + \{\text{error}\}] \end{aligned}$$

Wir geben noch die Semantik der neuen Ausdrücke an.

(E8) $\mathbb{E}[\text{proc } C]s = (\mathbb{C}[C], s)$

(C6) $\mathbb{C}[I](m, i, o) = ((mI = \text{unbound}) \to \text{error},\ \text{isProc } (mI) \to mI(m, i, o), \text{error})$

mit

$$\text{isProc } x = \begin{cases} \text{TRUE} & \text{falls } x \in \text{PROC} \\ \text{FALSE} & \text{sonst} \end{cases}$$

Man beachte die rekursiv definierten Bereichsgleichungen: PROC ist mit Hilfe von STATE, STATE mit Hilfe von MEMORY, MEMORY mit Hilfe von VALUE und VALUE mit Hilfe von PROC definiert. Um Probleme, die hieraus entstehen können, aus dem Wege zu gehen, müssen wir ein noch stärkeres und präziseres Modell aufbauen. Die nötigen mathematischen Grundlagen werden wir im nächsten Abschnitt legen.

7.3 Das λ - Kalkül

In diesem Abschnitt beschäftigen wir uns mit dem *λ-Kalkül*, das 1941 von CHURCH eingeführt wurde. Mit Hilfe des λ-Kalküls ist es möglich, Funktionen einheitlich zu benennen. Die Benennung von Funktionen (Default Naming) geschieht nach einem ähnlichen Prinzip wie der Aufbau prädikatenlogischer Formeln und Terme und spiegelt den operationellen Aufbau wieder.

λ-Ausdrücke lassen sich sehr leicht bilden. Sei $E[x]$ ein Ausdruck, der für $x \in D_1$ Werte in D_2 annimmt. Dann bezeichnet $\lambda x.E[x] : D_1 \to D_2$ die eindeutige Funktion $f : D_1 \to D_2$, mit

$$\forall\, d \in D_1 :\ f(d) = E[d].$$

Die Variable x, die hinter dem λ steht, bezeichnet man als *gebundene Variable*, $E[x]$ nennt man den *Rumpf* (engl:body) des λ-Ausdrucks. Später werden wir die Menge der λ-Ausdrücke formal definieren.

Falls sich der Wertebereich D_2 aus dem Kontext ergibt, schreiben wir abkürzend auch $\lambda(x : D_1).E[x]$. Ist auch D_1 fest, so schreiben wir lediglich $\lambda x.E[x]$.

Beispiel 7.1

- $\lambda x.x + 1$ bezeichnet die Nachfolgefunktion vom Typ $\underline{\text{NUM}} \to \underline{\text{NUM}}$.
- $\lambda x.xx$ bezeichnet die Quadratfunktion vom gleichen Typ.
- $\lambda x.(x \geq 0) \to 1, 0$ bezeichnet die Signumfunktion. Hier bei haben wir die MCCARTHYschen bedingten Ausdrücke verwendet.

Neben Funktionen mit nur einem Argument, lasssen sich durch λ-Ausdrücke auch mehrstellige Funktionen darstellen. So beschreibt der λ-Ausdruck

$$\lambda(x_1, \ldots, x_n).E[x_1, \ldots, x_n]$$

die Funktion

$$\begin{array}{rcl} f : [D_1 \times \ldots \times D_n] & \to & D' \\ (d_1, \ldots, d_n) & \mapsto & E[d_1, \ldots, d_n] \end{array}$$

Beispiel 7.2

- Die Additionsfunktion vom Typ $[\underline{\text{NUM}} \times \underline{\text{NUM}}] \to \underline{\text{NUM}}$ wird durch

  $$\lambda(x, y).x + y$$

  ausgedrückt.
- Die Subtraktionsfunktion vom Typ $[\underline{\text{NUM}} \times \underline{\text{NUM}}] \to [\underline{\text{NUM}} + \{\text{error}\}]$ hat die Form

  $$\lambda(x, y).(x < y) \to \text{error}, x - y$$

Wir führen folgende Konventionen ein:

- $fx_1x_2\ldots x_n \equiv (\ldots((fx_1)x_2)\ldots)$, d.h. $fgx = (fg)x$ und <u>nicht</u> $fgx = f(gx)$.
- $\lambda x_1x_2\cdots x_n.E[x_1,\ldots,x_n] \equiv \lambda x_1.(\lambda x_2.\cdots(\lambda x_n.E[x_1,\ldots,x_n])\ldots)$. Achtung: dies ist nicht äquivalent zu $\lambda(x_1,\ldots,x_n).E[x_1,\ldots,x_n]$.
- $\lambda x_1\cdots x_n.E[x_1,\ldots,x_n](d_1,\ldots,d_n) = E[d_1,\ldots d_n]$. Die Auswertung eines λ-Ausdrucks heißt auch *Applikation*, dabei beginnt die Auswertung bei der äußersten Variable (oben x_1).
- $D_1 \to D_2 \to \ldots \to D_n \equiv [D_1 \to [D_2 \to [\ldots[D_{n-1} \to D_n]\ldots]]]$
- $\lambda x.fxyz \equiv \lambda x.(fxyz) \not\equiv (\lambda x.f)xyz$, d.h. der Körper eines λ-Ausdruckes dehnt sich so weit wie möglich nach rechts aus.

Wie wir bereits oben gesehen haben, ist die Funktion plus : $[\underline{\text{NUM}} \times \underline{\text{NUM}}] \to \underline{\text{NUM}}$ definiert durch

$$\text{plus} = \lambda(n,m).n + m$$

eine zweistellige Funktion. Andererseits ist die Funktion

$$\text{plusc} = \lambda n.\lambda m.n + m = \lambda nm.n + m$$

eine einstellige Funktion mit $\text{plus}(n,m) = \text{plusc}\, nm$. Diese Möglichkeit mehrstellige Funktionen als einstellige Funktionen zu schreiben, wurde 1968 von CURRY entdeckt. Sein Verfahren beruht darauf, das karthesische Produkt im Definitionsbereich durch geschachtelte Abbildungen aufzulösen. Nehmen wir zum Beispiel die zweistellige Funtion $f : (D_1 \times D_2) \to D_3$. Wir können f zu der einstelligen Funktion $f' : D_1 \to (D_2 \to D_3)$ machen. Dann ist $f'(x)$ eine Funktion $D_2 \to D_3$. Es gilt offensichtlich $f(x_1,x_2) = f'(x_1)(x_2)$. Allgemein können wir den Operator curry definieren durch:

$$\begin{array}{rcl} \text{curry} : [[D_1 \times \ldots \times D_n] & \to & D] \to [D_1 \to \ldots \to D_n \to D] \\ f & \mapsto & \lambda x_1\cdots x_n.f(x_1,\cdots,x_n) \end{array}$$

Wir können die curry-Funktion auch im λ-Kalkül schreiben als

$$\text{curry} = \lambda f x_1\ldots x_n.f(x_1,\ldots,x_n).$$

Falls f selbst ein λ-Ausdruck $f = \lambda(x_1,\ldots,x_n).E[x_1,\ldots,x_n]$ ist, so gilt

$$\text{curry}(\lambda(x_1,\ldots,x_n).E[x_1,\ldots,x_n]) = \lambda x_1\ldots x_n.E[x_1,\ldots,x_n].$$

Eine λ-*Applikation* ist die Anwendung eines λ-Ausdrucks auf einen anderen λ-Ausdruck.

Die Darstellung eines Ausdruckes M als eine Funktion durch $\lambda x.M$ nennt man λ-*Abstraktion.*

Die beiden folgenden Axiome garantieren die Existenz und Eindeutigkeit von λ-Abstraktionen:

Axiom of Comprehension: Sei M ein Ausdruck mit Werten aus D, dann existiert eine Funktion

$$f : D' \to D \text{ mit } \forall x \in D' :\ fx = M.$$

Insbesondere existiert eine λ-Abstraktion vermöge $f = \lambda x.M$.

Axiom of Extensionality: Seien $f, g : D' \to D$ zwei Funktionen. Dann gilt

$$(\forall x \in D' :\ fx = gx) \Rightarrow f = g.$$

Insbesondere ist also die λ - Abstraktion eindeutig.

Nun wollen wir λ-Ausdrücke genauer betrachten. Wir geben daher eine BNF für λ-Ausdrücke an:

$$< \underline{\text{E}} >::=< \underline{\text{VAR}} >|< \underline{\text{E}} >< \underline{\text{E}} >|\ \lambda < \underline{\text{VAR}} > . < \underline{\text{E}} >|\ (< \underline{\text{E}} >)$$

Ein λ-Ausdruck ist somit entweder eine Variable, eine λ-Applikation ($< \underline{\text{E}} >< \underline{\text{E}} >$) oder eine λ-Abstraktion ($\lambda < \underline{\text{VAR}} > . < \underline{\text{E}} >$). Zusätzlich dürfen λ-Ausdrücke noch geklammert werden.

Beispiel 7.3 Gültige λ-Ausdrücke sind z.B. xx, $\lambda x.xx$, $\lambda xy.yx \equiv \lambda x.\lambda y.yx$, $\lambda xy.yz(\lambda z.z)$.

Durch die BNF ist die Menge Λ aller λ - Ausdrücke induktiv wie folgt definiert:

1. Jede Variable x ist ein λ - Ausdruck, d.h. $x \in \Lambda$.
2. Wenn M ein λ-Ausdruck ist, dann ist auch die λ-Abstraktion $\lambda x.M$ ein λ - Ausdruck.
3. Wenn M, N λ-Ausdrücke sind, dann ist auch die λ-Applikation MN ein λ-Ausdruck.

Analog zu prädikatenlogischen Termen und Formeln können wir freie und gebundene Variablen von λ- Ausdrücken definieren. Wir bezeichnen die freien Variablen eines λ-Ausdrucke M mit $\text{FV}(M)$.

1. Die Variable x ist frei in sich selbst, d.h. $\{x\} = \text{FV}(x)$.
2. x kommt frei in einer λ-Applikation MN vor, wenn x in M oder in N frei ist, d.h.
$$\text{FV}(MN) = \text{FV}(M) \cup \text{FV}(N).$$
3. x kommt frei in einer λ-Abstraktion $\lambda y.M$ vor, wenn $x \neq y$ und $x \in \text{FV}(M)$ in M frei ist, d.h.
$$\text{FV}(\lambda y.M) = \text{FV}(M) \setminus \{y\}.$$
4. x kommt frei in einem geklammerten λ-Ausdruck vor, wenn x in dem ungeklammerten λ-Ausdruck frei vorkommt, d.h.
$$\text{FV}((M)) = \text{FV}(M).$$

Wir definieren die Menge BV der gebundenen Variablen analog.

1. Keine Variable x kommt in sich selbst gebunden vor, d.h. $\text{BV}(x) = \emptyset$.
2. x kommt gebunden in einer λ-Applikation MN vor, wenn x in M oder in N gebunden ist, d.h.
$$\text{BV}(MN) = \text{BV}(M) \cup \text{BV}(N).$$
3. x kommt gebunden in einer λ-Abstraktion $\lambda y.M$ vor, wenn $x = y$ oder $x \in \text{BV}(M)$ in M gebunden ist, d.h.
$$\text{BV}(\lambda y.M) = \text{BV}(M) \cup \{y\}.$$
4. x kommt gebunden in einem geklammerten λ-Ausdruck vor, wenn x in dem ungeklammerten λ-Ausdruck gebunden vorkommt, d.h.
$$\text{BV}((M)) = \text{BV}(M).$$

In der Prädikatenlogik hatten wir den Begriff des Teilterms eingeführt. Dies werden wir auch bei λ-Ausdrücken machen. Wir bezeichen die Menge der Subterme eines λ-Ausdrucks N mit $\mathrm{SUB}(N)$ und schreiben $M \sqsubset N$, wenn M ein Subterm von N ist. Die Menge SUB kann offensichtlich wie folgt definiert werden.

1. $\mathrm{SUB}(x) = \{x\}$ für alle Variablen x.
2. $\mathrm{SUB}(\lambda x.N) = \mathrm{SUB}(N) \cup \{\lambda x.N\}$.
3. $\mathrm{SUB}(MN) = \mathrm{SUB}(N) \cup \mathrm{SUB}(M) \cup \{MN\}$.
4. $\mathrm{SUB}((M)) = \mathrm{SUB}(M)$

Für λ-Ausdrücke gelten die folgenden Axiome bzgl. der Gleichheit von λ-Ausdrücken. Dabei seien $M, N, Z \in \Lambda$ beliebige λ-Ausdrücke und x eine beliebige Variable:

1. $M = M$
2. $M = N \;\Rightarrow\; N = M$
3. $M = N, N = Z \;\Rightarrow\; M = Z$
4. $M = N \;\Rightarrow\; MZ = NZ$
5. $M = N \;\Rightarrow\; ZM = ZN$
6. $M = N \;\Rightarrow\; \lambda x.M = \lambda x.N$

Axiome 1).– 3). besagen, daß $=$ eine Äquivalenzrelation ist, 4. – 5. beziehen sich auf die λ-Applikation. Regel 6. bezieht sich auf die λ-Abstraktion und heißt auch ξ-Regel .

Eine λ-Applikation kann auch als Substitution verstanden werden. So substituieren wir in der λ-Applikation $(\lambda x.E[x])a$ die Variable x durch a. Ähnlich zur Prädikatenlogik muß man Vorsicht walten lassen, damit keine innen gebundenen Variablen "überschrieben" werden: $(\lambda x.(\lambda x.x))a = \lambda x.x \neq \lambda a.a$.

Die Operation der Substitution wird den folgenden Regeln genügen.

Definition 7.4 Seien x, y Variablen und M, A, B λ-Ausdrücke. Das Ergebnis einer Substitution $[M|x]A \equiv A[x := M]$ wird wie folgt spezifiziert:

1. $x[x := M] \equiv M$.
2. $y[x := M] \equiv y$, falls $x \neq y$.
3. $(\lambda y.A)[x := M] = \lambda y.(A[x := M])$, falls $x \neq y$ und $y \notin FV(M)$.
4. $(AB)[x := M] \equiv (A[x := M])(B[x := M])$.

Folgende Rechenregeln (Konversionen) betreffen die Substitution *gebundener Variablen.*

α - Konversion: ist die Umbenennung der gebundenen Variablen eines λ-Ausdrucks:

$$\lambda x.M = \lambda y.M[x := y], \text{ falls } y \notin \mathrm{BV}(M) \wedge y \notin \mathrm{FV}(M)$$

β - **Konversion:** ist die Substitution im Rumpf eines λ-Ausdrucks, d.h. die (partielle) Auswertung des λ-Ausdrucks:

$$(\lambda x.M)N = M[x := N]$$

η - **Konversion:** ist die "Verhinderung" der Durchführung einer Substitution, wenn die zu substituierende Variable im Rumpf des λ-Ausdrucks nicht frei vorkommt:

$$\lambda x.Ma = M, \text{ falls } x \notin FV(M)$$

Damit gilt der folgende Satz (Extensionalitätsaxiom).

Satz 7.5 Seien M, N λ-Ausdrücke mit $Mx = Nx$. Wenn $x \notin \mathrm{FV}(MN)$, dann ist auch $M = N$.

BEWEIS: mittels ξ-Regel und η-Konversion. □

Die β-Normalform eines λ-Ausdrucks $M \in \Lambda$ ergibt sich durch wiederholte Anwendung der β-Konversion, solange diese anwendbar ist. Nicht jeder λ-Ausdruck läßt sich in β-Normalform bringen (z.B. wird $(\lambda x.xx)(\lambda x.xxx)$ durch β-Konversion immer länger). CHURCH und ROSSER haben jedoch bewiesen:

Satz 7.6 Die β-Normalform eines λ-Ausdrucks ist eindeutig (falls sie existiert).

REYNOLDS hat 1970 die Programmiersprache *Gedanken* entwickelt, die auf dem λ-Kalkül basiert. 1961 hat MCCARTHY versucht, Elemente der λ-Notation (=λ-Kalkül + MCCARTHYsche bedingte Ausdrücke) in die Programmiersprache LISP einzubringen.

7.4 Denotationelle Semantik von TINY

Mit Hilfe der im vorigen Kapitel eingeführten λ-Notation, werden wir nun eine knappe und präzise Beschreibung der Semantik von TINY geben. Diese Beschreibung ist jedoch äquivalent zu der in Abschnitt 7.2 gegebenen Semantik von TINY.

Um eine knappe Beschreibung zu ermöglichen, werden wir eine geeignete Komposition von Funktionen definieren, das sogenannte *Sequencing von Funktionen.*

Die Komposition $g \circ f$ zweier Funktionen $f : D_1 \to D_2$ und $g : D_2 \to D_3$ ist eine Funktion $g \circ f : D_1 \to D_3$ definiert durch $g \circ f = \lambda x.g(fx)$. Das Sequencing von Funktionen ist ähnlich der Komposition von Funktionen, ermöglicht jedoch das Weiterleiten von Fehlern, die während der Anwendung der einzelnen Funktionen der Verkettung auftreten können.

Wir bezeichnen das Sequencing von Funktionen mit dem Symbol $\star$. Die Funktion $f \star g$ kann operationell wie folgt beschrieben werden: wenn bei der Auswertung von f ein Fehler auftaucht, so ist das Ergebnis von $f \star g$ ebenfalls ein Fehler. Andernfalls wird g auf das Ergebnis von f angewendet. Im Gegensatz zur Komposition $g \circ f$ ist es bei dem Sequencing $f \star g$ möglich, Fehler zu erkennen. Außerdem werden wir auch zulassen, daß g eine "curried"-Funktion sein kann. Wir definieren $f \star g$ exemplarisch an Hand der folgenden Fälle:

1. Sei $f : D_1 \to [D_2 + \{\text{error}\}]$ und $g : D_2 \to [D_3 + \{\text{error}\}]$. Dann ist
$$f \star g : D_1 \to [D_3 + \{\text{error}\}]$$
definiert durch
$$f \star g = \lambda x.(fx = \text{error}) \to \text{error}, g(fx).$$
2. Sei $f : D_1 \to [[D_2 \times D_3] + \{\text{error}\}]$ und $g : D_2 \to D_3 \to [D_4 + \{\text{error}\}]$. Dann ist
$$f \star g : D_1 \to [D_4 + \{\text{error}\}]$$
definiert durch
$$f \star g = \lambda x.(fx = \text{error}) \to \text{error}, (fx = (d_2, d_3)) \to g d_2 d_3.$$

Mit Hilfe des Sequencing können wir z.B. die semantische Klause
$$\mathbb{C}[C_1; C_2]s = (\mathbb{C}[C_1]s = \text{error}) \to \text{error}, \mathbb{C}[C_2](\mathbb{C}[C_1]s)$$
vereinfachen zu
$$\mathbb{C}[C_1; C_2] = \mathbb{C}[C_1] \star \mathbb{C}[C_2]$$

Um eine kompakte Schreibweise für die semantischen Klausen der denotationellen Semantik von TINY angeben zu können, werden wir jetzt einige Hilfsfunktionen angeben:

- result v gibt die Bedeutung eines Ausdrucks, der v als Wert zurückgibt, den Zustand jedoch nicht ändert.
$$\begin{aligned} \text{result} &: \text{VALUE} \to \text{STATE} \to [[\text{VALUE} \times \text{STATE}] + \{\text{error}\}] \\ \text{result} &= \lambda vs.(v, s). \end{aligned}$$
- donothing ist die Bedeutung eines Befehls, der nichts bewirkt.
$$\begin{aligned} \text{donothing} &: \text{STATE} \to [\text{STATE} + \{\text{error}\}] \\ \text{donothing} &= \lambda s.s. \end{aligned}$$
- checkNum v ist die Bedeutung eines Ausdrucks, der v zurückgibt und den Zustand nicht ändert, falls $v \in$ NUM ist, und sonst error zurückgibt.
$$\begin{aligned} \text{checkNum} &: \text{VALUE} \to \text{STATE} \to [[\text{VALUE} \times \text{STATE}] + \{\text{error}\}] \\ \text{checkNum} &= \lambda vs.\text{isNum } v \to (v, s), \text{error}. \end{aligned}$$
- checkBool v ist die Bedeutung eines Ausdrucks, der v zurückgibt und den Zustand nicht ändert, falls $v \in$ BOOL ist, und sonst error zurückgibt.
$$\begin{aligned} \text{checkBool} &: \text{VALUE} \to \text{STATE} \to [[\text{VALUE} \times \text{STATE}] + \{\text{error}\}] \\ \text{checkBool} &= \lambda vs.\text{isBool } v \to (v, s), \text{error}. \end{aligned}$$

Zusätzlich benötigen wir noch ein Konstrukt um Fallunterscheidungen behandeln zu können. Dazu definieren wir für beliebige D die Funktion
$$\text{cond} : [D \times D] \to \text{BOOL} \to D$$
mit
$$\text{cond}(d_1, d_2)b = \begin{cases} d_1 & \text{falls } b = \text{TRUE} \\ d_2 & \text{falls } b = \text{FALSE} \end{cases}.$$

Nun können wir die semantischen Klausen folgendermaßen angeben:

(E1) $\mathbb{E}[\underline{0}] = \text{result } 0, \quad \mathbb{E}[\underline{1}] = \text{result } 1.$

(E2) $\mathbb{E}[\text{true}] = \text{result TRUE}, \quad \mathbb{E}[\text{false}] = \text{result FALSE}.$

(E3) $\mathbb{E}[\text{read}] = \lambda(m,i,o).\text{null } i \to \text{error}, (\text{hd } i, (m, \text{tl } i, o)).$

(E4) $\mathbb{E}[I] = \lambda(m,i,o).(mI = \text{unbound}) \to \text{error}, (mI, (m,i,o)).$

(E5) $\mathbb{E}[\text{not } E] = \mathbb{E}[E] \star \text{checkBool} \star (\lambda v.\text{result}(\text{not } v)).$

(E6) $\mathbb{E}[E_1 = E_2] = \mathbb{E}[E_1] \star (\lambda v_1.\mathbb{E}[E_2] \star (\lambda v_2.\text{result}(v_1 = v_2)))$

(E7) $\mathbb{E}[E_1 + E_2] = \mathbb{E}[E_1] \star \text{checkNum} \star (\lambda v_1.\mathbb{E}[E_2] \star \text{checkNum} \star \lambda v_2.\text{result}(v_1 + v_2)).$

(C1) $\mathbb{C}[I := E] = \mathbb{E}[E] \star (\lambda v(m,i,o).(m[I := v], i, o)).$

(C2) $\mathbb{C}[\text{output } E] = \mathbb{E}[E] \star (\lambda v(m,i,o).(m,i,v.o)).$

(C3) $\mathbb{C}[\text{if } E \text{ then } C_1 \text{ else } C_2] = \mathbb{E}[E] \star \text{checkBool} \star \text{cond}(\mathbb{C}[C_1], \mathbb{C}[C_2]).$

(C4) $\mathbb{C}[\text{while } E \text{ do } C] = \mathbb{E}[E] \star \text{checkBool} \star \text{cond}(\mathbb{C}[C] \star \mathbb{C}[\text{while } E \text{ do } C], \text{donothing}).$

(C5) $\mathbb{C}[C_1; C_2] = \mathbb{C}[C_1] \star \mathbb{C}[C_2].$

7.5 Standard Semantik

Bei der Untersuchung der Semantik von realen Programmiersprachen, wie z.B. Pascal, treten noch komplexere Probleme auf. So führt z.B. die direkte Bindung von Bezeichnern (Identifiern) an Werte $\underline{I} \to$ VALUE zu Problemen. Außerdem muß geklärt werden, wie die Semantik von Sprungbefehlen definiert und generell eine effektive Fehlerbehandlung modelliert werden kann.

Das folgende Beispiel verdeutlicht die Probleme bei der sogenannten "einstufigen" Bindung von Identifiern.

```
PROCEDURE P(VAR x:Real;VAR y:Real);
          ⋮
P(z,z)
```

Die "doppelte Rückgabe" der Variablen z erfordert einen doppelten Zugriff auf diese Variable, d.h. in der Prozedur `P` sind bei Aufruf `P(z,z)` die Variablen `x` und `y` identisch. Dieser Konflikt kann durch eine sogenannte *zweistufige Identifierbindung* gelöst werden. Anstatt einen Identifier direkt an Werte zu binden, benutzen wir eine Zwischenstufe und erhalten zwei Abbildungen :

$$\underline{\text{I}} \to \text{LOC}, \quad \text{LOC} \to \text{VALUE}.$$

Damit wird jedem Identifier eine (Speicher-)Stelle aus LOC (Location) zugeordnet, und jeder Speicherstelle ein Wert aus der Wertemenge VALUE.

Die erste Stufe dieser Bindung heißt Umgebung (Environment):

$$\text{ENV} = \underline{\text{I}} \to [\text{DV} + \{\text{unbound}\}],$$

wobei DV = LOC die Menge der "denotable values" bezeichnet. Die zweite Stufe der Bindung modelliert den Speicher (Store):

$$\text{STORE} = \text{LOC} \to [\text{SV} + \{\text{unused}\}],$$

wobei SV die Menge der "storable values" bezeichnet. Damit ist das Problem der direkten Bindung von Identifiern an Werte gelöst.

Das zweite oben erwähnte Problem kann durch eine "Weiterführungssemantik" (continuation semantics) gelöst werden. Dabei machen wir die Bedeutung von Befehlen und Ausdrücken abhängig vom "Rest des Programms". Dies erlaubt z.B., daß ein Fehler bei der Auswertung eines Ausdruckes oder der Ausführung eines Befehles direkt zu einem Programmabbruch führt. So kann direkt auf eine mögliche Ausgabe des Programms Bezug genommen werden. Außerdem wird es möglich, Sprungbefehle semantisch zu verarbeiten, da der veränderte "Rest des Programms" bei Sprungbefehlen durch die Verwendung von Weiterführungssemantik darstellbar wird.

Für die "Continuation" von Befehlen geben wir den folgenden Bereich an:

$$\text{CONT} = \underbrace{\underline{\text{STATE}}}_{\substack{\text{Ergebnis des}\\ \text{Befehls}}} \to \underbrace{[\underline{\text{STATE}} + \{\text{error}\}]}_{\text{Ausgabe}}.$$

Für Ausdrücke spezifizieren wir:

$$\text{ECONT} = \underbrace{\underline{\text{VALUE}} \to \underline{\text{STATE}}}_{\substack{\text{Ergebnis von}\\ \text{Ausdrücken (curried)}}} \to \underbrace{[\underline{\text{STATE}} + \{\text{error}\}]}_{\text{Ausgabe}}.$$

Abkürzend werden wir die Schreibweise

$$\text{ECONT} = \text{VALUE} \to \text{CONT}$$

benutzen.

Die semantischen Funktionen $\mathbb{E}$ und $\mathbb{C}$ sind sowohl von Continuations als auch von den Zuständen abhängig. Damit gilt

$$\mathbb{E} : \underline{E} \to \text{ECONT} \to \text{STATE} \to [\text{STATE} + \{\text{error}\}]$$

und

$$\mathbb{C} : \underline{C} \to \text{CONT} \to \text{STATE} \to [\text{STATE} + \{\text{error}\}].$$

Abkürzend können wir auch schreiben:

$$\mathbb{E} : \underline{E} \to \text{ECONT} \to \text{CONT}$$
$$\mathbb{C} : \underline{C} \to \text{CONT} \to \text{CONT}.$$

Mit den Bezeichnungen $c, c_1, \ldots$ für Elemente aus CONT und $k, k_1, \ldots$ für Elemente aus ECONT können wir $\mathbb{E}$ und C wie folgt definieren:

$$\mathbb{E}[E]ks = \begin{cases} kvs' = k(v, s') & \text{falls } E \text{ den Wert } v \text{ hat und } s \text{ nach } s' \text{ transformiert wird} \\ \text{error} & \text{sonst} \end{cases}$$

und

$$\mathbb{C}[C]cs = \begin{cases} cs' & \text{falls der Befehl } C \text{ } s \text{ zu } s' \text{ transformiert} \\ \text{error} & \text{sonst} \end{cases}.$$

Damit können wir die folgenden Bereichsgleichungen aufstellen, dabei werden wir jedoch das alte Modell der Identifierbindung beibehalten:

$$\begin{aligned}
\text{VALUE} &= \text{NUM} \cup \text{BOOL} \\
\text{INPUT} &= \text{VALUE}^* \\
\text{OUTPUT} &= \text{VALUE}^* \\
\text{MEMORY} &= \underline{I} \to [\text{VALUE} \cup \{\text{unbound}\}] \\
\text{STATE} &= \text{MEMORY} \times \text{INPUT} \times \text{OUTPUT} \\
\text{CONT} &= \text{STATE} \to [\text{STATE} + \{\text{error}\}] \\
\text{ECONT} &= \text{VALUE} \to \text{CONT}
\end{aligned}$$

Damit ergeben sich die folgenden Gleichungen für die semantischen Klausen.

(E1) $\mathbb{E}[\underline{0}]ks = k0s$, $\mathbb{E}[\underline{1}]ks = k1s$
oder unter Anwendung des Extensionalitätsaxioms gleichwertig:

(E1)' $\mathbb{E}[\underline{0}]k = k0, \quad \mathbb{E}[\underline{1}]k = k1.$

(E2) $\mathbb{E}[\text{true}]k = k\ \text{TRUE}, \quad \mathbb{E}[\text{false}]k = k\ \text{FALSE}.$

(E3) $\mathbb{E}[\text{read}]k(m,i,o) = \text{null}\ i \to \text{error}, k(\text{hd}\ i)(m, \text{tl}\ i, o).$

(E4) $\mathbb{E}[I]k(m,i,o) = (mI = \text{unbound}) \to \text{error}, k(mI)(m,i,o).$

(E5) $\mathbb{E}[\text{not}\ E]ks = \mathbb{E}[E](\lambda vs'.\text{isBool}\ v \to k(\text{not}\ v)s', \text{error})s.$
Wenn wir err $:= \lambda s.\text{error}$ setzen, so können wir verkürzt schreiben:

(E5)' $\mathbb{E}[\text{not}\ E]k = \mathbb{E}[E](\lambda v.\text{isBool}\ v \to k(\text{not}\ v), \text{err}).$

(E6) $\mathbb{E}[E_1 = E_2]k = \mathbb{E}[E_1]\lambda v_1.\mathbb{E}[E_2]\lambda v_2.k(v_1 = v_2).$

(E7) $\mathbb{E}[E_1 + E_2]k = \mathbb{E}[E_1]\lambda v_1.\mathbb{E}[E_2]\lambda v_2.(\text{isNum}\ v_1 \wedge \text{isNum}\ v_2) \to k(v_1 + v_2), \text{err}.$

(C1) $\mathbb{C}[I := E]c = \mathbb{E}[E]\lambda v(m,i,o).c(m[I := v], i, o)$

(C2) $\mathbb{C}[\text{output}\ E]c = \mathbb{E}[E]\lambda v(m,i,o).c(m,i,v.o)$

(C3) $\mathbb{C}[\text{if}\ E\ \text{then}\ C_1\ \text{else}\ C_2]c = \mathbb{E}[E]\lambda v.\text{isBool}\ v \to (v \to \mathbb{C}[C_1]c, \mathbb{C}[C_2]c), \text{err}$

(C4) $\mathbb{C}[\text{while}\ E\ \text{do}\ C]c = \mathbb{E}[E]\lambda v.\text{isBool}\ v \to (v \to \mathbb{C}[C](\mathbb{C}[\text{while}\ E\ \text{do}\ C]c), c), \text{err}$

(C5) $\mathbb{C}[C_1; C_2]c = (\mathbb{C}[C_1] \circ \mathbb{C}[C_2])c = \mathbb{C}[C_1](\mathbb{C}[C_2]c)$

Literaturverzeichnis

[AhHoUl83] A. V. Aho, J. E. Hopcroft, J. D. Ullman, **Data Structures and Algorithms**, Addison-Wesley, Reading, Massachusetts, 1983.

[BaNa62] J. W. Backus, P. Naur et al., *Revised report on the algorithmic language ALGOL 60*, ed. by Peter Naur, The Computer Journal, **5**, 349–367, 1962.

[BeNo77] E. Bergmann, H. Noll, **Mathematische Logik mit Informatik-Anwendungen**, Springer-Verlag, Berlin/Heidelberg/New York, 1977.

[Bor77] A. Borodin, *On Relating Time and Space to Size and Depth*, SIAM Journal on Comput. **6**, 733–744,1979.

[Bra79] J. M. Brady, **The Theory of Computer Science. A Programming Approach**, Chapman and Hall, 1979.

[Coo71] S. A. Cook, *The complexity of theorem proving procedures*, Proc. 1^{st} Annual ACM STOC, 233–246, 1971.

[Col86] R. Cole, *Parallel Merge Sort*, 26^{th} Annual IEEE FOCS, 511–516, 1986.

[CoLeRi90] T. H. Cormen, C. E. Leiserson, R. L. Rivest, **Introduction to Algorithms**, MIT-Press, 1990.

[Fel78] W. Felscher, **Naive Mengen und abstrakte Zahlen I**, B.I.-Wissenschaftsverlag, Mannheim/Wien/Zürich, 1978.

[Gib85] A. Gibbons, **Algorithmic graph theory**, Cambridge University Press, Cambridge/London/New York, 1985.

[GiRy88] A. Gibbons, W. Rytter, **Efficient Parallel Algorithms**, Cambridge University Press, Cambridge/London/New York, 1988.

[Gor79] M. J. C. Gordon, **The Denotational Description of Programming Languages, An Introduction**, Springer-Verlag, New York/Heidelberg/Berlin, 1979.

[HaRu90] T. Hagerup, C. Rüb, *A Guided Tour of Chernoff Bounds*, Inform. Proc. Letters **33**, 305–308, 1990.

[HoUl79] J. E. Hopcroft, J. D. Ullman, **Introduction to Automata Theory, Languages and Computation**, Addison-Wesley, Reading, Massachusetts, 1979.

[Kla83] R. Klar, **Digitale Rechenautomaten**, Walter de Gruyter, Berlin/New York, 1983.

[LoMeWi86] J. Loeckx, K. Mehlhorn, R. Wilhelm, **Grundlagen der Programmiersprachen**, Teubner, Stuttgart, 1986.

[Man89] U. Manber, **Introduction to Algorithms, a Creative Approach**, Addison-Wesley, Reading, Massachusetts, 1989.

[Meh88] K. Mehlhorn, **Datenstrukturen und effiziente Algorithmen**, Teubner, Stuttgart, 1988.

[Mor64] D. Morgenstern, **Einführung in die Wahrscheinlichkeitsrechnung und mathematische Statistik**, Springer-Verlag, Berlin/Heidelberg/New York, 1968.

[Knu81] D. E. Knuth, **The Art of Computer Programming, vol. 1** , Addison-Wesley, Reading, Massachusetts, 1981.

[Sch80] J. T. Schwartz, *Fast Probabilistic Algorithms for Verification of Polynomial Identities*, Journal of the ACM **27**, 701–717, 1980.

[ShSt63] J. C. Shepherdson, H. E. Sturgis, *Computability of Recursive Functions*, Journal of the ACM **10**, 217–255, 1963.

[WeEd79] J. Welsh, J. Elder, **Introduction to Pascal**, Prentice-Hall, 1979.

[WuShHiFl80] W. A. Wulf, M. Shaw, P. N. Hilfinger, L. Flon, **Fundamental Structures of Computer Science**, Addison-Wesley, Reading, Massachusetts, 1980.