



---

# **Handbuch der Informatik**

herausgegeben von

Prof. Dr. Albert Endres

Prof. Dr. Hermann Krallmann

Dr. Peter Schnupp

---

Band 1.5

---

# Mathematische Grundlagen der Informatik

---

von

Prof. Dr. Wolfram Pohlers  
Westfälische Wilhelms-Universität Münster

---

R. Oldenbourg Verlag München Wien 1993

**Prof. Dr. Wolfram Pohlers**

**Anschrift:**  
Institut für Mathematische Logik  
und Grundlagenforschung  
der Westfälischen Wilhelms-Universität  
Einsteinstr. 62  
48149 Münster

**Die Deutsche Bibliothek – CIP-Einheitsaufnahme**

**Handbuch der Informatik** : [die umfassende Darstellung der Informatik in Einzelbänden] / hrsg. von Albert Endres ... – München ; Wien : Oldenbourg.

NE: Endres, Albert [Hrsg.]

Bd. 1.5. Pohlers, Wolfram: Mathematische Grundlagen der Informatik. – 1993

**Pohlers, Wolfram:**

Mathematische Grundlagen der Informatik / von Wolfram Pohlers. – München ; Wien : Oldenbourg, 1993  
(Handbuch der Informatik ; Bd. 1.5)  
ISBN 3-486-22113-2

© 1993 R. Oldenbourg Verlag GmbH, München

Das Werk einschließlich aller Abbildungen ist urheberrechtlich geschützt. Jede Verwertung außerhalb der Grenzen des Urheberrechtsgesetzes ist ohne Zustimmung des Verlages unzulässig und strafbar. Das gilt insbesondere für Vervielfältigungen, Übersetzungen, Mikroverfilmungen und die Einspeicherung und Bearbeitung in elektronischen Systemen.

Gesamtherstellung: R. Oldenbourg Graphische Betriebe GmbH, München

ISBN 3-486-22113-2

---

# Inhaltsverzeichnis

Vorwort der Herausgeber . . . . .	7
Vorwort des Autors . . . . .	9
<b>1 Vorberachtungen . . . . .</b>	<b>11</b>
1.1 Historische Entwicklung . . . . .	11
1.2 Bedeutung der Berechenbarkeitstheorie . . . . .	15
1.3 Ziel der Darstellung . . . . .	16
<b>2 Maschinenberechenbare Funktionen . . . . .</b>	<b>19</b>
2.1 Abstrakte Maschinen . . . . .	19
2.2 Turingmaschinen . . . . .	25
2.3 Registermaschinen . . . . .	28
2.4 Simulation abstrakter Maschinen . . . . .	30
2.5 Simulation der Registermaschine auf der Turingmaschine . . . . .	33
<b>3 <math>\mu</math>-rekursive Funktionen . . . . .</b>	<b>37</b>
3.1 Primitiv rekursive Funktionen . . . . .	37
3.2 Primitiv rekursive Prädikate . . . . .	44
3.3 Primitiv rekursive Kodierung . . . . .	48
3.4 Simultane Rekursion und Wertverlaufsrekursion . . . . .	51
3.5 Kodes für primitiv rekursive Funktionen . . . . .	53
3.6 $\mu$ -partiellrekursive Funktionen . . . . .	55
3.7 Maschinenberechenbarkeit und $\mu$ -partiellrekursive Funktionen	57
3.8 Der Normalformensatz von KLEENE . . . . .	64
3.9 Rekursive, semirekursive und rekursiv aufzählbare Prädikate	70
3.10 Universelle Funktionen und der Rekursionssatz . . . . .	78
<b>4 <math>\lambda</math>-definierbare Funktionen . . . . .</b>	<b>89</b>
4.1 Der $\lambda$ -Kalkül . . . . .	90
4.2 $\beta$ -Gleichheit von $\lambda$ -Termen . . . . .	94
4.3 Rekursive Aufzählbarkeit der $\beta$ -Gleichheit . . . . .	98
4.4 $\lambda$ -definierbare Funktionen . . . . .	101
4.5 $\lambda$ -Kalküle über Typenstrukturen . . . . .	108

## INHALTSVERZEICHNIS

---

<b>5</b>	<b>Unentscheidbare Probleme . . . . .</b>	115
5.1	Unlösbarkeitsgrade . . . . .	115
5.2	CHOMSKY Grammatiken . . . . .	118
5.3	Prädikatenlogik . . . . .	126
<b>6</b>	<b>Grundzüge der Komplexitätslehre . . . . .</b>	137
6.1	TIME- und SPACE-beschränkte Funktionen . . . . .	137
6.2	Abstrakte Komplexitätsmaße . . . . .	152
6.3	<i>P</i> - und <i>NP</i> - entscheidbare Probleme . . . . .	155
<b>A</b>	<b>Grundlagen der Graphentheorie . . . . .</b>	165
A.1	Grundbegriffe . . . . .	165
A.2	Subgraphen und Homomorphismen . . . . .	170
A.3	Pfadprobleme . . . . .	172
A.4	Markierte und gefärbte Graphen . . . . .	175
<b>B</b>	<b>Literaturempfehlungen . . . . .</b>	179
	<b>Literaturverzeichnis . . . . .</b>	181
	<b>Register . . . . .</b>	185

---

## Vorwort der Herausgeber

Das *Handbuch der Informatik* versucht, das Gesamtgebiet der Informatik mit seinen Grundlagen, seinen Teilgebieten und seinen wichtigsten Anwendungen zusammenhängend darzustellen. Es informiert Lehrende und Lernende, DV-Praktiker und DV-Nutzer über Konzepte, Methoden und Techniken, deren grundlegende Bedeutung anerkannt und deren Nützlichkeit in der Praxis sich gezeigt hat. Grenzen und Entwicklungstendenzen eines Gebiets werden angesprochen.

Jeder Band des *Handbuchs der Informatik* behandelt ein in sich abgeschlossenes Thema. Der Leser kann sich – auch unabhängig von den anderen Bänden des Handbuchs – in das betreffende Gebiet neu einarbeiten, vorhandenes Wissen auffrischen oder im Sinne eines Nachschlagewerks einzelne Themen aufspüren und vertiefen. Hierbei helfen die Strukturierung und Typographie des Textes und die Hinweise auf weiterführende Literatur.

Weder im Mathematikunterricht der höheren Schulen noch in den einführenden Mathematikvorlesungen der Hochschule erhielt der Informatiker üblicherweise Informationen über die Zweige der Mathematik, die für seine spätere Praxis vermutlich die wichtigsten werden: die Theorie der Berechenbarkeit, den  $\lambda$ -Kalkül und die Grundlagen der Komplexitätstheorie. Dabei beruhen hierauf nicht nur das Verständnis und die Analyse von Algorithmen, sondern auch und vor allem die Abschätzung und vielleicht noch wichtiger – das „Gefühl“ für ihre Komplexität: die Abhängigkeit des Rechenaufwands von der Zahl der bearbeiteten Objekte. Die Folge erleben wir immer wieder: die „kombinatorische Explosion“ der Antwortzeiten eleganter Prototypen, sobald sie „evolutorsch“ zu Produktionssystemen über echten Datenbasen ausgebaut werden. Pohlers vermittelt in einer kompakten, aber trotzdem fundierten Darstellung genau dieses Wissen, dem Informatiker in der Ausbildung ebenso wie dem in der Praxis, der sich und seine Auftraggeber vor solchen Entwurfskatastrophen sicher schützen will.

A. Endres

H. Krallmann

P. Schnupp



---

## Vorwort des Autors

Der Bitte des Verlages, den vorliegenden Band des Handbuches der Informatik zu übernehmen, bin ich anfänglich nur sehr zögerlich gefolgt. Zunächst sah ich die Schwierigkeit, die mathematischen *Grundlagen* von den mathematischen *Werkzeugen* der Informatik zu trennen. Eine auch nur annähernd erschöpfende Behandlung der mathematischen Werkzeuge des Informatikers hätte den Rahmen diese Bandes gesprengt und hätte auch außerhalb meiner Kompetenz gelegen. Ich mußte mich daher auf die eigentliche mathematische Grundlage der Informatik beschränken, die Theorie der Berechenbarkeit. Diese ist aber hinreichend in einer Fülle von Lehrbüchern dargestellt. Wodurch sollte sich die Darstellung in diesem Band von ihnen unterscheiden?

Hier kam mir die Intention des Handbuches entgegen, seinen Ge genstand möglichst beweisfrei darstellen zu wollen. Natürlich ist es unmöglich, eine mathematische Theorie völlig beweisfrei darzustellen, ohne ins belanglose Plaudern zu geraten. Ein solches Buch wäre sinnlos.

Was mir vorschwebte, war ein Buch, das einen nur an einem Über blick interessierten Leser in die Lage versetzt, sich diesen zu verschaffen, indem er allzu detaillierte Ausführungen überliest, aber dennoch geeignet ist, als Leitfaden für einen Studierenden zu dienen, der sich genauer mit der Theorie der Berechenbarkeit auseinanderzusetzen hat.

Dies sind natürlich im Prinzip unvereinbare Ziele. Als Kompromiß habe ich versucht, die Theorie möglichst zwanglos zu entwickeln und dabei das übliche Schema ... – Definition – Lemma – Beweis – Satz – Beweis – ... zu vermeiden. Dennoch finden sich in dieser Darstellung Definitionen und Sätze. Dies erscheint mir zur Darstellung einer mathematischen Theorie unverzichtbar zu sein. Allerdings habe ich nach Möglichkeit versucht, Definitionen als Präzisierungen anschaulicher Sachverhalte zu erarbeiten und Sätze als Zusammenfassungen vorhergegangener Über legungen zu formulieren. Das kann natürlich nicht immer gelingen. So erfordern viele Partien dieser Darstellung doch eine intensivere Mitarbeit des Lesers. Auf der anderen Seite konnten allzu komplexe Zusam menhänge oft nur sehr grob skizziert werden oder mußten ganz weg fallen. Für ein gründlicheres Studium sind diese an Hand der zitierten Lehrbücher zu ergänzen.

Dennoch hoffe ich, daß dieses Buch dem einen oder anderen Leser von Nutzen sein wird.

All denen, die mich bei der Abfassung dieses Textes direkt und indirekt unterstützt haben, sage ich hier herzlichen Dank. Dies sind ins besondere die Mitarbeiter des Institutes für Mathematische Logik der Universität Münster, die mir oft mit Rat zur Seite standen. Großen Anteil

---

haben auch die Münsteraner Studierenden, die in den Einführungsvorlesungen zur theoretischen Informatik den hier dargestellten Stoff – in ausführlicherer Form – kritisch aufgenommen haben. Ihrer Kritik verdanke ich viele Verbesserungen.

Mein besonderer Dank gilt Frau Sabine Melles, die die erste (nicht-mathematische) Korrektur gelesen hat und Herrn stud. math. Martin Wabnik, der die Endfassung nochmals durchgesehen hat. Den Mitarbeitern des Oldenbourg Verlages danke ich für ihre Unterstützung.

Auch ich habe während der Abfassung des Textes etwas gelernt – einige der Geheimnisse der  $\text{\TeX}$ - und  $\text{\LaTeX}$ -Systeme. Das Buch wurde vollständig mit  $\text{\LaTeX}$  erzeugt. Allerdings mußten sämtliche .sty Dateien (und einige  $\text{\LaTeX}$ -Macros) verändert werden, um den Layout-Vorgaben zu genügen. Hier möchte ich mich bei der Systemgruppe unseres Fachbereiches bedanken, die mich durch den Dschungel des hiesigen “file system” geleitet hat. Auch meiner Familie, insbesondere meiner Frau, möchte ich für die Geduld danken, die sie immer dann zeigten, wenn ich durch die Tücken des Systems wieder einmal völlig entnervt war.

Münster, im April 1993  
Wolfram Pohlers

# 1. Vorbetrachtungen

Die mathematische Grundlage der Informatik ist die Theorie der Berechenbarkeit. In ihr versucht man, den intuitiven Begriff des Algorithmus mathematisch präzise zu fassen und zu untersuchen.

Etymologisch lässt sich das Wort Algorithmus auf den Namen des arabischen Rechenmeisters AL-KHARIZMI, ALCHWARAZM in anderen Transskriptionen, zurückführen, der ein Rechenbuch verfasste, das uns von den Indern überliefert wurde. Dieses Buch wurde im Lateinischen als *Liber algoritmiz* zitiert. Im Laufe der Zeit schliff sich dieser Name zu dem Wort *Algorithmus* ab, das gleichzeitig zu einem Synonym für den Inhalt des Buches wurde, das im wesentlichen Rechenvorschriften enthielt. So verstehen wir heute unter einem Algorithmus ganz generell eine Rechenvorschrift.

## 1.1 Historische Entwicklung

Die Suche nach Rechenvorschriften hat die Entwicklung der Mathematik wesentlich geprägt und kann wohl mit Recht als eine der Wurzeln der Informatik angesehen werden. Eine besondere Rolle kommt hier der Schrift *Ars Magna* des RAIMUNDUS LULLUS [\*1235, †1315] zu, in der er die Idee pflegt, daß alle wissenschaftliche Erkenntnis aus einigen Grundideen abgeleitet werden kann. Obwohl dieses Buch keine eigentlichen Vorschläge enthält, wie dies zu erreichen sei, beeinflußte es doch die weitere Entwicklung. Die Forschungen von GOTTFRIED W. LEIBNIZ sollen wesentlich von dieser Schrift beeinflußt gewesen sein. LEIBNIZ betrachtete die Mathematik als eine Schlüsselwissenschaft, mit deren Hilfe er alle wissenschaftlichen Fragen entscheiden wollte. Dazu regte er in seinem Werk *De arte combinatoria* eine Mathematik der Ideen an und wurde so zu einem der Begründer der heutigen mathematischen Logik. Sein Ziel war es noch immer, einen allumfassenden Algorithmus zu entwickeln, eben die *Ars Magna*, der alle hinreichend formalisierbaren Fragen entscheiden sollte. Sein letztes Ziel dabei war es wohl, mit Hilfe der *Ars Magna* auch die theologische Frage nach der Existenz Gottes entscheiden zu können.

Die Suche nach dem allumfassenden Algorithmus ist lange Zeit eine starke Triebfeder mathematischer Forschung geblieben. Bis heute lebt

## 1. Vorberachtungen

---

der Traum von der Mechanisierbarkeit der Denkarbeit fort und befügt die Forschung. Dank immer leistungsfähiger und schneller werdender Rechenanlagen scheint die endgültige Verwirklichung dieses Traumes in immer greifbarere Nähe zu rücken.

Daher erscheint es uns auch für den praktisch arbeitenden Informatiker wesentlich zu sein, über die prinzipiellen Grenzen der Mechanisierbarkeit mathematischen Arbeitens informiert zu sein.

Es ist sicherlich kein Zufall, daß eine mathematische Theorie der Algorithmen, eben die Berechenbarkeitstheorie, erst in unserem Jahrhundert entwickelt wurde, in dem der Glaube an die Existenz des allumfassenden Algorithmus ins Wanken geriet. Erst jetzt versuchte man nachzuweisen, daß gewisse Probleme aus prinzipiellen Gründen nicht algorithmisch lösbar sein können. Dies ist ein konzeptuell viel schwierigeres Unterfangen, als die Lösbarkeit eines Problems nachzuweisen. Im letzteren Fall hat man einfach einen Algorithmus anzugeben, was im Einzelfall schwierig sein mag, aber keine besondere Theorie der Algorithmen erfordert. Der Unlösbarkeitsbeweis läßt sich hingegen nicht durch Austesten aller Algorithmen führen. Davon gibt es unendlich viele. Hier wird wirklich eine Theorie der Algorithmen benötigt.

Die Entwicklung dieser Theorie ist ein Werk der letzten 70 Jahre. Die Geschichte dieser Entwicklung ist auf Grund des geringen zeitlichen Abstandes daher nur äußerst unzuverlässig darzustellen. Eine Beurteilung der Quellen nach wissenschaftshistorischen Gesichtspunkten steht unserer Kenntnis noch aus. Daher bitten wir, den folgenden historischen Abriß mit der nötigen Vorsicht zu betrachten. Insbesondere kann keine Garantie dafür übernommen werden, daß die Prioritäten für die frühen Ergebnisse hier richtig eingeordnet werden. Da dies nicht das eigentliche Anliegen dieses Buches ist, wäre die dazu erforderliche Literaturrecherche viel zu aufwendig gewesen. Wir folgen hier einfach der herrschenden Meinung. Eine wesentliche Quelle hierfür war die von MARTIN DAVIS herausgebene Anthologie [DAV64]. Eine exakte Aufhellung der historischen Entwicklung der Berechenbarkeitstheorie ist eine Aufgabe, die einem Wissenschaftshistoriker vorbehalten bleiben muß.

Als sicher anzusehen ist die enge Verzahnung der Berechenbarkeitstheorie mit der Entwicklung der mathematischen Logik. Im Vordergrund stand hier zunächst das *Entscheidungsproblem*, die Frage, ob sich die Herleitbarkeit einer logischen Formel in einem Logikkalkül entscheiden läßt. Also wieder das Motiv der *Ars Magna*. Formalisierte Logikkalküle standen zur Verfügung seitdem, beginnend mit dem Jahr 1910, BERTRAND RUSSELS [\*1872, †1970] und ALFRED N. WHITEHEADS [\*1861, †1947] Werk *Principia Mathematica* veröffentlicht wurde. Allerdings war man sich zu diesem Zeitpunkt der Vollständigkeit dieser Kalküle noch nicht sicher. Diese wurde erst im Jahre 1930 von KURT GÖDEL [\*1906, †1978] nachgewiesen. Doch bereits im Jahre 1915 erschien eine Schrift von LEOPOLD LÖWENHEIM [\*1878, †1957], in der er ein Ent-

scheidungsverfahren für ein Fragment der Prädikatenlogik angibt. Diese Arbeit konnte 1923 von THORALF SKOLEM [\*1878, †1963] zu einem Beweis der Entscheidbarkeit eines Fragments der Logik erweitert werden, das wir heute als monadische Prädikatenlogik der ersten Stufe bezeichnen. Weitere positive Lösungen des Entscheidungsproblems wurden von MOJEZESZ PRESBURGER [\*1904, †wahrscheinlich 1943 im Warschauer Ghetto] (Entscheidbarkeit der Arithmetik mit der Addition als einzigem Funktionssymbol), ALFRED TARSKI [\*1901, †1983] (Entscheidbarkeit von Formeln der Theorie reeller Zahlen, die nur die Symbole + und  $\leq$  enthalten) sowie nochmals T. SKOLEM (Entscheidbarkeit der Arithmetik mit der Multiplikation als einzigem Funktionssymbol) erzielt.

Der Begriff der Definition durch Rekursion scheint unter Mathematikern der Jahrhundertwende mehr oder minder explizit bekannt gewesen zu sein. DAVID HILBERT [\*1862, †1943] gibt in seiner Schrift „*Über das Unendliche*“ bereits einen allgemeinen Rekursionsbegriff an. Im Jahre 1931 veröffentlicht K. GÖDEL in seinem Artikel „*Über formal unentscheidbare Sätze der Principia Mathematica und verwandter Systeme*“ seine berühmten Unvollständigkeitssätze. In dieser Arbeit klassifiziert er diejenigen Funktionen als rekursiv, die wir heute als primitiv rekursiv bezeichnen (vgl. Abschnitt 3.1). RÓZSA PÉTER [\*1905, †1977] führt 1934 in [PET34] den Terminus der *primitiven Rekursion* ein und untersucht die Abschlußeigenschaften der Klasse der im Sinne GÖDELS rekursiven (also im heutigen Sinne primitiv rekursiven) Funktionen gegenüber verschiedenen Formen der Rekursion, wie etwa Wertverlaufsrekursion, mehrfacher Rekursion etc. In ihrer Arbeit [PET35], die im Jahre 1935 erscheint, zeigt sie, daß eine von WILHELM ACKERMANN [\*1896, †1962] bereits im Jahre 1928 angegebene und von ihr stark vereinfachte Funktion zwar berechenbar ist, jedoch nicht in der von GÖDEL beschriebenen Klasse liegen kann.

In der Zeit von 1933-1934 hält sich GÖDEL im *Institute for Advanced Studies* in Princeton, New Jersey auf. In einer Vorlesung im Jahre 1934, unter deren Hörern sich ALONZO CHURCH [\*1903], STEPHEN C. KLEENE [\*1909] und JOHN B. ROSSER [\*1907, †1989] befinden, – die von den beiden letztgenannten angefertigte Mitschrift der Vorlesung ist in [DAV64] abgedruckt – geht er auf die ACKERMANNsche Funktion ein und entwickelt den Begriff einer allgemeinrekursiven Funktion, wobei er sich auf Ideen stützt, die ihm, seinen eigenen Angaben nach, von JACQUES HERBRAND [\*1908, †1931] mündlich mitgeteilt wurden. A. CHURCH veröffentlicht im Jahre 1936 die Arbeit [CHUR36a], in der er seinen  $\lambda$ -Kalkül (vgl. Abschnitt 4.1) entwickelt, und zeigt, daß jede allgemeinrekursive Funktion  $\lambda$ -definierbar ist. Diese Arbeit enthält auch die These, daß jede berechenbare Funktion allgemeinrekursiv ist; sie

## 1. Vorbetrachtungen

---

ist heute als CHURCHsche These (vgl. Abschnitt 3.7) bekannt. Er zeigte darüber hinaus die Existenz eines algorithmisch unlösbaren Problems und erweiterte dies in der Note [CHUR36b], die ebenfalls 1936 erschien, zu einem Beweis der Unlösbarkeit des Entscheidungsproblems der Prädikatenlogik. Ebenfalls im Jahre 1936 erscheint KLEENEs Arbeit [KLEE36], in der er den GÖDEL-HERBRANDschen Ansatz analysiert und zu einer Charakterisierung rekursiver Funktionen gelangt, indem er die primitiv rekursiven Funktionen gegenüber dem Suchoperator abschließt. (Dieser Ansatz ähnelt daher dem, der hier im Abschnitt 3.6 präsentiert wird.) Diese Arbeit enthält bereits eine gut entwickelte Theorie. Man findet das Normalformentheorem, der Begriff einer rekursiv aufzählbaren Menge wird eingeführt und KLEENE zeigt, daß die Menge der Indizes rekursiver Funktionen nicht rekursiv aufzählbar sein kann. Eine Abrundung dieser Ergebnisse findet man dann in der späteren Arbeit [KLEE43], in der auch der Begriff der partiellen Funktion explizit eingeführt wird.

Unabhängig davon publiziert ALAN M. TURING [\*1912, †1954], ebenfalls im Jahre 1936, in [TUR36] seinen Ansatz zur präzisen Erfassung der Berechenbarkeit. Dort entwickelt er die Turingmaschine und kann nachweisen, daß turingberechenbare und  $\lambda$ -definierbare Funktionen übereinstimmen. Er zeigt die Unlösbarkeit des Halteproblems und überträgt dies auf die Unentscheidbarkeit der Prädikatenlogik. So erhält er unabhängig von CHURCH ebenfalls die Unlösbarkeit des Entscheidungsproblems.

EMIL POST [\*1897, †1954] entwickelt unabhängig von TURING einen praktisch identischen Ansatz. In Kenntnis der Arbeit CHURCHs veröffentlicht er im Jahre 1936 die Arbeit [POST36], in der exakt die abstrakte Maschine beschrieben wird, die wir heute als Turingmaschine bezeichnen.

Die Zeit bis zum Jahre 1936 kann demnach als die Periode angesehen werden, in der der Begriff der intuitiven Berechenbarkeit seine mathematische Form erhalten hat. Die folgenden Jahre bringen die Befestigung der Theorie. So enthält die von KLEENE im Jahre 1943 publizierte Arbeit „*Recursive Predicates and Quantifiers*“ eine bereits völlig modern anmutende Darstellung der Berechenbarkeitstheorie. Er führt dort partiellrekursive Funktionen ein und betrachtet bereits relativierte Rekursivität. Eine ähnliche Begriffsbildung findet sich auch in TURINGS Arbeit [TUR39], die 1939 erscheint und Turingmaschinen mit Orakeln betrachtet. Im Jahre 1947 zeigt POST, daß das Wortproblem für Halbgruppen unentscheidbar ist.

Alle diese Untersuchungen waren sehr mathematisch motiviert – obwohl Forscher wie TURING durchaus auch den Bau von Rechenmaschinen im Sinn hatten. Wesentlichen Niederschlag in der Konstruktion

von Rechenmaschinen fanden die Ideen der Berechenbarkeitstheorie in der Anregung JOHN V. NEUMANNS [\*1903, †1957], universelle Maschinen zu bauen. Dies kann als der Durchbruch in der Entwicklung zum modernen Computer angesehen werden. Neuere Entwicklungen der Theorie, wie sie sich beispielsweise in der Komplexitätslehre finden, sind bereits viel stärker von Motiven geprägt, die aus der rechnergerechten Aufbereitung praktischer Probleme stammen. Hier beobachtet man ein fruchtbare Wechselspiel zwischen Theorie und Praxis, vergleichbar dem, wie es sich zwischen Mathematik und Physik abspielt. Die Historie dieser Entwicklungen ist jedoch viel zu jung, als daß sie mit hinreichender Zuverlässigkeit hier dargestellt werden könnte.

### 1.2 Bedeutung der Berechenbarkeitstheorie

Die ersten wesentlichen Ergebnisse der Berechenbarkeitstheorie waren negativer Art, indem sie zeigten, daß gewisse Probleme, die man gerne mechanisch mit Hilfe von Algorithmen entschieden hätte, sich aus prinzipiellen Gründen so nicht entscheiden lassen. Insbesondere bedeutete das, daß der Traum von der *Ars Magna* sich zumindest nicht global verwirklichen läßt.

Dieses Ergebnis darf man aber auf keinen Fall dahingehend interpretieren, daß damit die Suche nach Algorithmen obsolet geworden wäre. Ganz im Gegenteil. Wir müssen davon ausgehen, daß die Kollektion der Erkenntnisse, die von denkenden Menschen gewonnen wird, eine im Sinne des Abschnitts 3.9 rekursiv aufzählbare Menge ist. Dies trifft zumindest für die in der Mathematik beweisbaren Sätze zu und läßt sich auf alle solche Erkenntnisse erweitern, die durch Folgern aus entscheidbaren Fakten gewonnen werden können. Das bedeutet aber, daß ein Algorithmus existiert, der alle diese Erkenntnisse erzeugt. Es gibt also keinen mathematischen Grund dafür, daß nicht alles das, was von Menschen geleistet wird, nicht auch von einer Maschine zu leisten wäre. Allerdings fehlt der Maschine bis heute noch immer die selektive Fähigkeit des Menschen, aus der Beurteilung von Ergebnissen und Zwischenergebnissen uninteressante Resultate zu unterdrücken und aussichtslose Wege gar nicht erst weiterzuverfolgen. Diese Situation erinnert etwas an die der frühen Schachprogramme, denen man auf Grund ihrer Unfähigkeit, Stellungen zu beurteilen und somit aussichtslose Varianten von vornehmlich auszuschließen, keine wesentliche Steigerung der Spielstärke zugetraut hatte. Heute liegen aber Programme vor, die durchaus chancenreich gegen Großmeister antreten können. Aus dieser Analogie läßt sich die Hoffnung schöpfen, daß eine ähnliche Entwicklung auch in anderen Bereichen erzielt werden kann. Die Entwicklung und Implementierung von Selektionsalgorithmen ist jedoch noch immer eine Herausforderung der Zukunft, die hohe Ansprüche an die Kreativität der beteiligten Forscher und Entwickler stellen wird.

Schon um zu verhindern, daß kostbare Zeit an die Lösung prinzipiell unlösbarer Probleme verschwendet wird, sollten die wesentlichsten

## 1. Vorberachtungen

---

Unlösbarkeitsergebnisse daher auch dem praktisch arbeitenden Informatiker bekannt sein.

An positiven Resultaten hat die Berechenbarkeitstheorie andererseits tiefe Einsichten in die Natur berechenbarer Funktionen geliefert. Hier ist unter anderem das Normalformentheorem des Abschnitts 3.8 zu nennen, mit dessen Hilfe sich universelle Funktionen konstruieren lassen. Dies eröffnet die Möglichkeit der Konstruktion abstrakter universeller Maschinen, deren physikalische Realisierung, wie von v. NEUMANN angeregt, unsere heutigen Rechner sind.

Die in der Berechenbarkeitstheorie entwickelten Methoden fanden in sehr weite Bereiche der Informatik Eingang. Dies wurde immer stärker der Fall, je mehr sich die Informatik von einer "ad hoc Beschäftigung mit Rechenmaschinen" zu einer von allgemeinen Prinzipien geleiteten Ingenieurwissenschaft entwickelte. Beispiele hierfür sind die Entwicklung effizienter, benutzernaher Sprachen und deren Implementierung über abstrakte Maschinen.

Die in der Berechenbarkeitstheorie entwickelten Methoden bilden natürlicherweise die Basis der *theoretischen Informatik*.

Die äußerst befruchtende Wirkung der Berechenbarkeitstheorie auf die mathematische Logik sei hier nur am Rande erwähnt.

## 1.3 Ziel der Darstellung

Der vorliegende Band soll einen Überblick über die Berechenbarkeitstheorie geben. Dabei erlaubt es der dieser Darstellung gesteckte Rahmen nicht, alle Ergebnisse in allen Details zu erarbeiten. Da die Darstellung einer mathematischen Theorie ohne alle Begründungen jedoch äußerst unbefriedigend bleiben muß, haben wir uns bemüht, die Theorie zumindest entlang ihrer Hauptlinien in groben Zügen zu entwickeln.

In der Darstellung folgen wir nicht der historischen Entwicklung, sondern beginnen mit dem Ansatz von POST und TURING. Dazu stellen wir in Kapitel 2 zunächst ein abstraktes Maschinenkonzept vor und präzisieren den Begriff des Algorithmus als Steuerprogramm für eine abstrakte Basismaschine. Dies führt uns zur Klasse der maschinenberechenbaren partiellen Funktionen. Turing- und registermaschinenberechenbare Funktionen sind Spezialfälle davon. Als wesentliches Konzept wird die Simulation abstrakter Maschinen entwickelt. Dieses verwenden wir dann, um einzusehen, daß sich alle auf Registermaschinen berechenbaren Funktionen auch auf Turingmaschinen berechnen lassen.

In Kapitel 3 verfolgen wir einen mehr mathematisch motivierten Ansatz. Ausgehend von einfachen und sicherlich berechenbaren Grund-

### 1.3. Ziel der Darstellung

---

funktionen erhalten wir mit Hilfe der Grundoperationen *Komposition* und *primitiver Rekursion* die Klasse der *primitiv rekursiven Funktionen*. Wir studieren deren wesentliche Eigenschaften und gelangen dann über eine Gödelisierung dieser Funktionen zur Einsicht, daß diese Klasse nicht alle intuitiv berechenbaren Funktionen erfassen kann. Daher erweitern wir die Grundoperatoren um den *unbeschränkten Suchoperator* und schließen dann die Grundfunktionen gegenüber diesem erweiterten Operatorenbegriff ab. Dies zwingt uns wieder, partielle Funktionen zu betrachten, und wir erhalten die Klasse der *partiellrekursiven Funktionen*. Wir überzeugen uns dann, daß diese mit der Klasse der maschinenberechenbaren partiellen Funktionen übereinstimmt. Dieser mathematisch stringente Zugang wird es uns erlauben, die wesentlichen Aussagen der Berechenbarkeitstheorie, wie das Normalformentheorem, das  $S_n^m$ -Theorem, den Rekursionssatz u.a., zu entwickeln.

In Kapitel 4 stellen wir dann den auf CHURCH zurückgehenden kombinatorischen Zugang vor. Wir entwickeln die Grundbegriffe des  $\lambda$ -Kalküls, diskutieren das CHURCH-ROSSER Theorem und führen schließlich den Begriff der  $\lambda$ -definierbaren Funktion ein. Auch hier können wir feststellen, daß dies zu der bereits bekannten Klasse von Funktionen führt. Das Kapitel schließt mit einer knappen Einführung in die Theorie der  $\lambda$ -Kalküle mit Typen ein.

In dem folgenden Kapitel 5 untersuchen wir als Anwendungsbeispiele zwei Entscheidungsprobleme. Wir fragen erst nach der Lösbarkeit des Wortproblems für CHOMSKY Sprachen und stellen dessen Unlösbarkeit fest. Wir gehen dann kurz auf Sprachen mit lösbarem Wortproblem ein. Als zweites Beispiel studieren wir das Entscheidungsproblem der Prädikatenlogik. Wir skizzieren, wie dies auf das Halteproblem für Registermaschinen reduziert werden kann und folgern daraus dessen Unlösbarkeit.

Die Darstellung endet in Kapitel 6 mit einem Abriß der Grundlagen der Komplexitätslehre, in der der Versuch gemacht wird, durch Komplexitätsbetrachtungen Funktionen nicht nur auf ihre theoretische, sondern auch auf ihre praktische Berechenbarkeit hin zu untersuchen.

In einem Anhang stellen wir dann einige Grundbegriffe der Graphentheorie zusammen.



## 2. Maschinenberechenbare Funktionen

Eines der Charakteristika eines Algorithmus besteht offensichtlich darin, daß die Rechenvorschrift, die durch ihn gegeben wird, nicht an die Intelligenz des Rechnenden appellieren darf. Sie muß so gegeben sein, daß sie völlig mechanisch ausführbar ist. Damit eröffnet sich natürlich die Möglichkeit, auch eine Maschine mit der Rechnung zu betrauen. Dieses wird der erste Weg sein, auf dem wir zu einer mathematisch präzisen Definition der algorithmisch berechenbaren Funktion gelangen werden.

### 2.1 Abstrakte Maschinen

Die Mindestanforderungen an eine Rechenmaschine sind offenbar die folgenden:

1. Eine Rechenmaschine benötigt einen Speicher, in dem sie Ergebnisse speichern kann.
2. Eine Rechenmaschine muß über einen bestimmten Satz  $\mathcal{S}$  von Modifikationsinstruktionen verfügen, die es erlauben, den Speicherinhalt zu modifizieren.
3. Die Rechenmaschine muß darüber hinaus über einen Satz  $\mathcal{T}$  von Testbefehlen verfügen, die es ihr erlauben, den Inhalt des Speichers zu überprüfen.
4. Die Maschine benötigt Eingabefunktionen, die Daten aus  $\mathcal{D}_{in}$ , der Menge der Eingabedaten, in den Speicher einlesen und Ausgabefunktionen, die Speicherinhalte in Elemente von  $\mathcal{D}_{out}$ , der Menge der Ausgabedaten, transferieren können.

Die bislang beschriebenen Dinge stellen die Basingredienzen für eine Rechenmaschine dar, durch die ein Maschinentypus festgelegt wird. Sie charakterisieren eine *Basismaschine*, die wir formal folgendermaßen einführen wollen.

**Definition 1** Eine abstrakte Basismaschine ist ein Quintupel

Abstrakte  
Basismaschinen

$$\mathcal{B} = (\mathcal{M}, \mathcal{S}, \mathcal{T}, \text{IN}, \text{OUT})$$

mit folgenden Eigenschaften:

1.  $\mathcal{M}$  ist eine Menge, sie stellt die Menge aller Speicherinhalte dar.
2.  $\mathcal{S}$  ist eine Menge von Abbildungen  $S: \mathcal{M} \rightarrow \mathcal{M}$ . Wir nennen  $\mathcal{S}$  die Menge der Modifikationsinstruktionen der Basismaschine.
3.  $\mathcal{T}$  ist eine Menge von Abbildungen  $T: \mathcal{M} \rightarrow \{0, 1\}$ .  $\mathcal{T}$  ist die Men-

## 2. Maschinenberechenbare Funktionen

---

ge der Testinstruktionen. Dabei wollen wir sagen, daß der Test eines Speicherinhaltes  $m \in \mathcal{M}$  positiv verlaufen ist, wenn  $T(m) = 1$  war, anderenfalls ist er negativ verlaufen.

4. IN ist eine Abbildung, die jedem Datum  $d \in \mathcal{D}_{in}$  einen Speicherzustand  $IN(d) \in \mathcal{M}$  zuordnet.
5. OUT ist eine Abbildung, die jedem Speicherzustand  $m \in \mathcal{M}$  ein Datum  $OUT(d) \in \mathcal{D}_{out}$  zuordnet.

Steuereinheit einer abstrakten Basismaschine

Steuerprogramme für abstrakte Basismaschinen

Flußdiagramme

Mit einer Basismaschine allein läßt sich allerdings noch nicht rechnen. Es fehlt noch eine *Steuereinheit*, die den Rechenvorgang in der Maschine steuert. Wir wollen hier von der Vorstellung ausgehen, daß die Steuereinheit die Maschine durch ein *Programm* steuert. Wir haben also den Begriff eines Programmes zu entwickeln. Ein Programm besteht aus *Anweisungen*. Wir unterscheiden dabei zwei Typen von Anweisungen. Einmal haben wir *Modifikationsanweisungen* von der Form  $(k, S, m)$ , wobei  $k$  und  $m$  natürliche Zahlen sind und  $S \in \mathcal{S}$  eine Modifikationsinstruktion ist. Wir nennen  $k$  und  $m$  die *Marken* der Anweisung, wobei  $k$  die *Kenn-* und  $m$  die *Sprungmarke* der Anweisung heißt. Zum anderen haben wir *Testanweisungen* der Form  $(k, T, v, m)$ , wobei  $T \in \mathcal{T}$  eine Testinstruktion von  $\mathcal{B}$  ist und die Marken  $k, m, v$  wieder natürliche Zahlen sind. Wir nennen  $k$  wieder die *Kennmarke* der Anweisung, während  $m$  die *Sprung-* und  $v$  die *Verzweigungsmarke* der Anweisung ist. Ein *Programm* ist nun eine *endliche* Menge von Anweisungen zusammen mit einer ausgezeichneten Marke  $k_P$ , die die *Startmarke* des Programmes  $P$  heißt. Wir werden daher Programme oft in der Form  $P = (k_P, A)$  notieren, wobei  $k_P$  für die Startmarke und  $A$  für die endliche Menge der Anweisungen steht. Allerdings wollen wir die Schreibweise sehr liberal handhaben und oft einfach  $a \in P$  schreiben, wenn wir ausdrücken wollen, daß  $a$  eine Anweisung des Programmes  $P$  ist.

Programme werden oft in Form von *Flußdiagrammen* angegeben. So wird beispielsweise das Programm

$$P = (k_P, \{(k_P, S_1, n_1), (n_1, T, v, m), (m, S_3, n_2), (v, S_2, n_3)\}) \quad (2.1)$$

durch das in Abbildung 2.1 - 1 gezeigte Flußdiagramm dargestellt. Formal sind Flußdiagramme markierte gerichtete Graphen (vgl. Anhang A). Mit Ausnahme der Start- und Endmarken, die als ausgezeichnete Knoten auftreten, werden die Programmarken durch unmarkierte Kanten, Befehle durch markierte Knoten repräsentiert. Die Abfolge von Kenn- und Sprung- bzw. Verzweigungsmarken wird durch die Richtung der Kanten angezeigt. Hier kommt die Eigenschaft von Programmen zum Ausdruck, daß ihre Wirkung bei (korrekter) Umbenennung von Marken nicht verändert wird. Man kann Programme als identisch betrachten, wenn sie

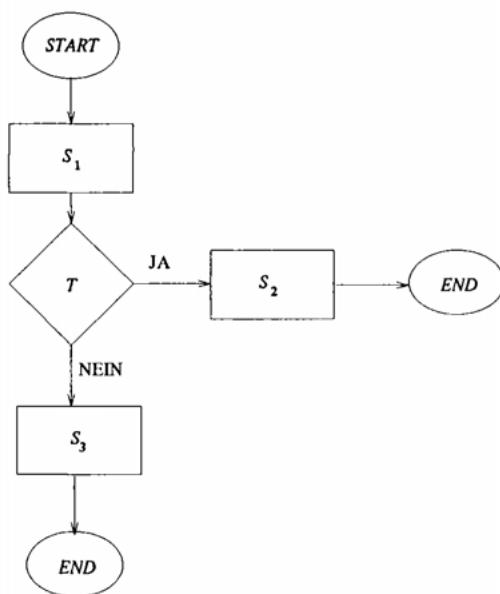


Abb. 2.1 - 1 : Beispiel eines Flußdiagrammes

durch die gleichen Flußdiagramme dargestellt werden, d.h. durch geeignete Umbenennung von Programmmarken auseinander hervorgehen. Dies wird sofort klar, wenn man sich von der anschaulichen Vorstellung leiten läßt, daß ein Programm die *Steuereinheit* (finite control) der Maschine beschreibt. Die Marken markieren dabei die endlich vielen *Zustände*, derer eine Maschine fähig ist. Die Aktion der Maschine erfolgt in Abhängigkeit ihres augenblicklichen Zustandes und des Speicherinhaltes. Die Startmarke markiert den Anfangszustand der Maschine, in dem sie sich nach dem Einschalten befindet.

Die formale Charakterisierung abstrakter Rechenmaschinen gibt die folgende Definition.

**Definition 2** Eine abstrakte Maschine  $\mathcal{C}$  ist eine abstrakte Basismaschine  $\mathcal{B}$  zusammen mit einem Programm  $P$  für  $\mathcal{B}$ . Wir notieren dies als  $\mathcal{C} = (\mathcal{B}, P)$ .

Bei festgehaltener Basismaschine ist die abstrakte Maschine also durch ihre Steuereinheit, d. h. durch ihr Programm bestimmt. Wir werden daher nicht immer zwischen der Maschine und dem Programm, das sie bestimmt, unterscheiden.

Haben wir eine abstrakte Maschine  $\mathcal{C}$  vorliegen, so wollen wir den Rechenverlauf in der Maschine beschreiben. Wesentlich dafür ist die

## 2. Maschinenberechenbare Funktionen

---

*Rechenschritt- oder Übergangsfunktion*  $RS_{\mathcal{C}}$ , die einen Schritt in einem Berechnungsvorgang in  $\mathcal{C}$  beschreibt. Die Rechenschrittfunktion arbeitet auf dem *Konfigurationsraum* der Maschine  $\mathcal{C}$ . Der Konfigurationsraum der Maschine  $\mathcal{C}$  mit Basismaschine  $\mathcal{B} = (\mathcal{M}, \mathcal{S}, \mathcal{T}, \text{IN}, \text{OUT})$  ist der Raum

$$K(\mathcal{C}) := M(\mathcal{C}) \times \mathcal{M},$$

wobei  $M(\mathcal{C})$  die Menge aller Marken des Programmes von  $\mathcal{C}$  bedeutet und damit – intuitiv betrachtet – die Menge aller möglichen Zustände der Maschine beschreibt. Formal lässt sich die Wirkungsweise von  $RS_{\mathcal{C}}$  so beschreiben, daß, ausgehend von der Konfiguration  $(k, z)$ , im Programm nachgesehen wird, ob eine Anweisung mit der Kennmarke  $k$  vorhanden ist. Ist dies der Fall und ist  $k$  Kennmarke einer Modifikationsanweisung  $(k, S, p)$ , so wird der Speicherinhalt zu  $S(z)$  modifiziert und die Maschine begibt sich in den Zustand  $p$ , d.h. es wird die Konfiguration  $(p, S(z))$  angenommen. Liegt eine Testanweisung  $(k, T, p, q)$  vor, so wird der Speicherinhalt nicht modifiziert, je nach Ergebnis der Testinstruktion wird jedoch der Zustand  $p$  oder  $q$  angenommen. Verläuft der Test positiv, so begibt sich die Maschine in den Zustand  $p$ , fällt er negativ aus, so wird der durch die Sprungmarke markierte Zustand  $q$  angenommen. Formal erhalten wir also:

$$RS_{\mathcal{C}}(k, z) := \begin{cases} (p, S(z)), & \text{falls } (k, S, p) \in \mathcal{C}, \\ (p, z), & \text{falls } (k, T, p, q) \in \mathcal{C} \text{ und } T(z) = 1, \\ (q, z), & \text{falls } (k, T, p, q) \in \mathcal{C} \text{ und } T(z) = 0, \\ (k, z), & \text{sonst.} \end{cases}$$

Die Rechenschrittfunktion ist natürlich nur dann eine Funktion im strengen Sinne, wenn die Maschine *deterministisch* ist, d.h. wenn jede Anweisung im Programm durch ihre Kennmarke eindeutig bestimmt ist. Wir sprechen dann von *deterministischen Programmen*. Es ist durchaus sinnvoll auch nichtdeterministische Maschinen zu betrachten. Wir werden darauf zurückkommen. Im Augenblick wollen wir uns jedoch auf die Untersuchung deterministischer Maschinen beschränken.

Die Iteration der Rechenschrittfunktion definieren wir durch

$$RS_{\mathcal{C}}^n(k, z) := \begin{cases} (k, z), & \text{falls } n = 0, \\ RS_{\mathcal{C}}(RS_{\mathcal{C}}^{n-1}(k, z)), & \text{falls } n = m + 1. \end{cases}$$

Damit ist  $RS_{\mathcal{C}}^n(k, z)$  die  $n$ -fache Anwendung von  $RS_{\mathcal{C}}$  auf  $(k, z)$ .

Die Abarbeitung eines Programmes erfolgt nun in der Weise, daß die Eingabe zunächst vermöge der Eingabefunktion IN in den Speicher gelesen wird. Dies ergibt die *Startkonfiguration*  $(k_P, \text{IN}(d))$ , wenn  $k_P$  die Startmarke des Programmes  $P$  und  $d$  das Datum war, das eingegeben wurde. Dann wird die Rechenschrittfunktion so lange iteriert, bis eine

*Endkonfiguration* erreicht ist. Endkonfigurationen sind Konfigurationen der Form  $(e, z)$ , in denen  $e$  eine *Endmarke* des Programmes ist, d.h. eine Marke von  $P$ , die selbst nicht wieder Kennmarke einer Anweisung in  $P$  ist. Endmarken markieren demnach Zustände der Maschine, von denen aus sie nicht mehr vernünftig weiterarbeiten kann und daher tunlichst stoppen sollte. Mit  $E(\mathcal{C})$  notieren wir die Menge der Endkonfigurationen des Programmes der Maschine  $\mathcal{C}$ .

Wir wollen die Ein- und Ausgabefunktionen erst einmal außer acht lassen und uns auf die Vorgänge in der Maschine beschränken. Wir betrachten zunächst die *inneren* Funktionen der Maschine. Die *innere Rechenzeitfunktion*  $RZ_{\mathcal{C}}$  ordnet einem Speicherinhalt  $z \in \mathcal{M}$  die Anzahl der Schritte zu, um die die Rechenschrittfunktion iteriert werden muß, bis eine Endkonfiguration vorliegt. Das heißt:

$$RZ_{\mathcal{C}}(z) := \min\{t \mid RSc^t(k_p, z) \in E(\mathcal{C})\}.$$

Wir bemerken hier, daß  $RZ_{\mathcal{C}}$  keine Funktion im üblichen Sinne ist. Dazu betrachten wir das Programm

$$(k_p, T, k_p, k_p)$$

mit der Startmarke  $k_p$ . Offensichtlich ist dies ein wohldefiniertes deterministisches Programm. Allerdings besitzt es keine Endmarken. Damit kann dessen Rechenschrittfunktion niemals eine Endkonfiguration erreichen. Also ist die Rechenzeitfunktion nicht definiert. Es ist natürlich ebenso leicht Programme anzugeben, deren Rechenzeitfunktion für gewisse Speicherzustände definiert, für andere jedoch undefiniert ist. Wir stehen daher vor der Notwendigkeit, *partielle Funktionen* einzuführen, die nicht unbedingt auf ihrer gesamten Quelle definiert sind.

**Definition 3** Seien  $Q, Z$  Mengen und  $D \subseteq Q$ . Eine Abbildung

Partielle Funktionen

$$f: D \longrightarrow Z$$

heißt eine partielle Funktion mit Quelle  $Q$  und Ziel  $Z$ . Wir notieren dies als

$$f: Q \longrightarrow_p Z.$$

Die Menge  $D$  heißt der *Definitionsbereich* von  $f$  und wird üblicherweise mit  $\text{dom}(f)$  bezeichnet. D.h.  $\text{dom}(f) = \{x \in Q \mid (\exists y)[f(x) = y]\}$ . Eine partielle Funktion heißt *total*, wenn  $\text{dom}(f) = Q$  ist, d.h. wenn ihre Quelle und ihr Definitionsbereich übereinstimmen.

Oft ist es angenehm, sich partielle Funktionen  $f: Q \longrightarrow_p Z$  als Abbildungen

## 2. Maschinenberechenbare Funktionen

---

$$\tilde{f}: Q \longrightarrow Z \cup \{\infty\}$$

vorzustellen, die durch

$$\tilde{f}(z) := \begin{cases} f(z), & \text{falls } z \in \text{dom}(f), \\ \infty, & \text{sonst,} \end{cases}$$

gegeben sind. Zwei partielle Funktionen

$$f, g: Q \longrightarrow_p Z$$

sind als gleich anzusehen, wenn  $\tilde{f}$  und  $\tilde{g}$  übereinstimmen. Wir definieren daher

$$f(z) \simeq g(z) : \iff \tilde{f}(z) = \tilde{g}(z),$$

d.h.

$$f(z) \simeq g(z) \iff (z \notin \text{dom}(f) \wedge z \notin \text{dom}(g)) \vee (z \in \text{dom}(f) \cap \text{dom}(g) \wedge f(z) = g(z)).$$

Schreiben wir

$$f(x) \simeq u,$$

so bedeutet dies, da  $u$  immer definiert ist, daß  $x \in \text{dom}(f)$  ist und  $f(x)$  den Wert  $u$  ergibt.

Bei der oben definierten inneren Rechenzeitfunktion  $RZ_C$  handelt es sich offenbar um eine partielle Funktion  $RZ_C: M \longrightarrow_p N$ , wenn wir mit  $N$  die Menge der natürlichen Zahlen (einschließlich der 0) bezeichnen.

Um den Speicherinhalt  $z$  einer Konfiguration  $(m, z)$  zurückzuerhalten, bedienen wir uns der Projektionsfunktion  $\pi_2$ . Allgemein sei die  $i$ -te (mengentheoretische) *Projektion* definiert als

$$\begin{aligned} \pi_i: M^n &\longrightarrow M \\ \pi_i(m_1, \dots, m_n) &:= m_i, \end{aligned}$$

wobei  $M$  eine beliebige Menge bezeichne und  $1 \leq i \leq n$  ist.

Mit Hilfe der inneren Rechenzeitfunktion und der Projektion definieren wir die *innere Resultatsfunktion* als

$$RES_C(z) := \pi_2(RS_C^{RZ_C(z)}(k_P, z)),$$

was so zu lesen ist, daß der Funktionswert  $RES_C(z)$  nur dann definiert ist, wenn auch  $\pi_2(RS_C^{RZ_C(z)}(k_P, z))$  definiert ist, und dann dessen Wert annimmt.

Die innere Resultatsfunktion liefert uns den Speicherinhalt nach Beendigung der Rechnung. Wir erhalten schließlich die *äußere Rechenzeitfunktion*  $Rz_C$  und die *äußere Resultatsfunktion*  $Res_C$  einer Maschine  $C$  durch Komposition der inneren Funktionen mit den Ein- und Ausgabefunktionen, d.h.

Äußere Rechenzeit- und äußere Resultatsfunktion

$$Rz_C(d) : \simeq RZ_C(\text{IN}(d))$$

und

$$Res_C(d) : \simeq \text{OUT}(RES_C(\text{IN}(d))).$$

Die von einer Basismaschine  $B$  partiell berechenbaren Funktionen definieren wir als

$$\mathbf{P}_B := \{f \mid f = Res_C \text{ für eine abstrakte Maschine } C \text{ über } B\}.$$

Die von einer Basismaschine  $B$  berechenbaren Funktionen sind gegeben durch

$$\mathbf{F}_B := \{f \in \mathbf{P}_B \mid f \text{ ist total}\}.$$

Bevor wir die Theorie der abstrakten Maschinen weiterverfolgen, wollen wir die zwei wichtigsten Beispiele solcher Maschinen vorstellen.

## 2.2 Turingmaschinen

Das historisch älteste Beispiel einer abstrakten Maschine ist die von ALAN M. TURING eingeführte Turingmaschine. Die Turingmaschine soll das Rechnen mit Bleistift und Papier formalisieren. Anschaulich stellen wir uns die Turingbasismaschine als ein nach beiden Seiten hin unendliches Band vor, das in einzelne Felder unterteilt ist. Auf dem Band arbeitet ein Schreib-Lese-Kopf, der jeweils ein Feld, das *Arbeitsfeld*, bearbeiten kann. Auf diesem Feld kann er ein Zeichen  $z$  eines Alphabets drucken ( $PRT(z)$ ), das Feld löschen ( $CLR$ ) und testen, ob das Zeichen auf dem Feld mit einem vorgegebenen Zeichen  $z$  übereinstimmt ( $BEQ(z)$ ). Schließlich kann die Maschine noch das Arbeitsfeld um ein Feld nach links ( $LFT$ ) oder nach rechts verlegen ( $RHT$ ).

Informale Beschreibung einer Turingmaschine

Die Menge der Speicherzustände einer Turingmaschine ist durch die Menge ihrer Bandinschriften gegeben. Wir haben daher diese Menge genauer zu beschreiben. Zunächst wollen wir vereinbaren, daß immer nur endlich viele Felder des Bandes beschrieben sind. Dies ist eine vernünftige Forderung, wenn wir davon ausgehen, daß im Einlesevorgang nur endlich viele Daten auf das Band geschrieben werden können, und zu

## 2. Maschinenberechenbare Funktionen

---

jedem Zeitpunkt in der Rechnung nur endlich viele Rechenschritte absolviert sein können. Auf dem Band stehen also Zeichen. Um dies zu präzisieren, führen wir den Begriff eines *Alphabets* und der über diesem Alphabet gebildeten *Wörter* ein.

**Definition 4** Ein Alphabet ist eine endliche, nicht leere Menge  $A$ . Die Elemente von  $A$  heißen die Zeichen oder manchmal auch die Buchstaben des Alphabets  $A$ .

Ein Wort über  $A$  ist ein Tupel  $(z_1, \dots, z_n) \in A^n$ .

Die Menge aller Wörter über  $A$  bezeichnen wir üblicherweise mit

$$A^* := \bigcup_{n \in \mathbb{N}} A^n,$$

wobei wir  $n = 0$ , d.h. das leere Wort  $\Lambda$ , zulassen wollen. Die Menge der nicht leeren Wörter bezeichnen wir mit

$$A^+ := \bigcup_{n \in \mathbb{N}^+} A^n,$$

wobei  $\mathbb{N}^+$  die Menge der positiven natürlichen Zahlen bedeutet. Üblicherweise schreiben wir Wörter unter Weglassen der Klammerung als  $z_1 \dots z_n$ .

Die Länge eines Wortes  $w \in A^*$  ist gegeben durch

$$l(w) := \min\{n \mid w \in A^n\}.$$

Die Verkettung zweier Wörter  $u = z_1 \dots z_n$  und  $v = a_1 \dots a_m$  ist das Wort

$$u \cdot v := z_1 \dots z_n a_1 \dots a_m.$$

Das nach beiden Seiten unendliche Band der Turingmaschine lässt sich gut durch die Menge  $\mathbb{Z}$  der ganzen Zahlen darstellen. Wir stellen uns dabei die Felder des Bandes mit ganzen Zahlen nummeriert vor, wobei das Arbeitsfeld die Nummer 0 erhält. Eine Bandinschrift für eine Turingmaschine über dem Alphabet  $A$  ist dann eine Abbildung

$$f: \mathbb{Z} \longrightarrow A \cup \{B\}$$

derart, daß das Urbild  $f^{-1}[A] := \{x \in \mathbb{Z} \mid f(x) \in A\}$  der Menge  $A$  endlich ist. Hier haben wir das Zeichen  $B$  (Blank) gewählt, um mitzuteilen, daß ein Feld unbeschrieben (d.h. mit einem ‘‘Blank’’ beschrieben) ist. Damit können wir die Menge der Speicherzustände einer Turingmaschine durch

$$\mathcal{M} = \{f \mid f: \mathbb{Z} \longrightarrow A \wedge f^{-1}[A] \text{ ist endlich}\}$$

beschreiben. Informal wollen wir Bandinschriften immer in der Form

$$B^\infty a_1 \dots a_m \underline{b} c_1 \dots c_n B^\infty$$

beschreiben, wobei das Zeichen  $B$  wieder für ein unbeschriebenes Feld steht und  $a_i, b$  und  $c_j$  Zeichen des Alphabets oder Blanks sind. Das Zeichen auf dem Arbeitsfeld ist dabei durch die Unterstreichung hervorgehoben.

Bei der Definition von Eingabe- und Ausgabefunktionen haben wir natürlich sehr viele Möglichkeiten. Eine einfache Version besteht darin, daß die Datenmenge durch

$$\mathcal{D} := \bigcup_{n \in \mathbb{N}} (A^*)^n,$$

der Menge der  $n$ -Tupel von Wörtern über dem Alphabet  $A$ , gegeben ist. Dann haben wir eine einfache Eingabefunktion

$$\text{IN}(u_1, \dots, u_n) := B^\infty \underline{B} u_1^1 \dots u_1^{l(u_1)} B \dots B u_n^1 \dots u_n^{l(u_n)} B^\infty,$$

wenn jedes Wort die Form  $u_i = u_i^1 \dots u_i^{l(u_i)}$  hat und die Ausgabefunktion

$$\text{OUT}(B^\infty \dots \underline{a} u_1^1 \dots u_1^{l(u_1)} B \dots B u_n^1 \dots u_n^{l(u_n)} B^\infty) := (u_1, \dots, u_n).$$

Interessant sind vor allem die mit Turingmaschinen berechenbaren Funktionen  $f: \mathbb{N}^k \rightarrow \mathbb{N}^l$ . Zu ihrer Beschreibung benötigt man eine Kodierung natürlicher Zahlen. Hier kann man bereits mit Turingmaschinen über einem eлементigen Alphabet  $A = \{\star\}$  auskommen. Für deren Basismaschinen läßt sich der Instruktionssatz etwas einfacher schreiben. Statt  $PRT(\star)$  schreiben wir einfach  $PRT$ , und  $BEQ$  steht für  $BEQ(B)$ , d.h.  $BEQ$  nimmt genau dann den Wert 1 an, wenn das Arbeitsfeld leer ist. Natürliche Zahlen  $n$  kodieren wir durch das Wort  $\underbrace{\star \dots \star}_{n\text{-fach}}$ . Wir setzen

$$\star^0 = \Lambda$$

Turingbasismaschinen für die Berechnung zahlentheoretischer Funktionen

Kodierung natürlicher Zahlen über einem elementigen Alphabet

und

$$\star^{n+1} = \star \hat{\ } \star^n$$

und erhalten so  $\star^n = \underbrace{\star \dots \star}_{n\text{-fach}}$ . Dann können wir Eingabefunktionen

$$\text{IN}_k: \mathbb{N}^k \rightarrow \mathcal{M}$$

und Ausgabefunktionen

$$\text{OUT}_l: \mathcal{M} \rightarrow \mathbb{N}^l$$

definieren durch:

## 2. Maschinenberechenbare Funktionen

---

$$\text{IN}_k(n_1, \dots, n_k) := B^\infty \underline{B} \star^{n_1} B \dots B \star^{n_k} B^\infty \quad (2.2)$$

und

$$\text{OUT}_l(B^\infty v \underline{z} \star^{n_1} B \dots B \star^{n_k} B^\infty) := (n_1, \dots, n_l). \quad (2.3)$$

Dabei ist zu bemerken, daß wir ohne Einschränkung der Allgemeinheit  $l \leq k$  annehmen können, denn  $B^\infty$  kodiert ja beliebig viele Nullen.

Ganz offensichtlich hängt die Klasse der auf Turingmaschinen berechenbaren Funktionen wesentlich von der Definition der Ein- und Ausgabefunktionen ab. Da wir uns in der Entwicklung der Theorie der berechenbaren Funktionen ohne Einschränkung der Allgemeinheit auf Funktionen über den natürlichen Zahlen beschränken können, werden wir im folgenden die Funktionen von  $\mathbb{N}^k$  nach  $\mathbb{N}^l$  als turingberechenbar ansehen, die sich über der Turingbasismaschine  $Tur$  mit dem einelementigen Alphabet  $\{\star\}$  und den in (2.2) bzw. (2.3) definierten Ein- und Ausgabefunktionen berechnen lassen. Das ist zur Entwicklung der allgemeinen Theorie ausreichend. Für weitergehende Betrachtungen, insbesondere für Komplexitätsbetrachtungen, kann es notwendig werden, elaboriertere Kodierungen natürlicher Zahlen und, damit verbunden, auch veränderte Ein- und Ausgabefunktionen zu betrachten.

Turingberechenbare  
Funktionen

Wir definieren die Klasse der turingberechenbaren partiellen Funktionen zu

$$\mathbf{P}_{Tur}^{(k,l)} := \{f \mid f: \mathbb{N}^k \longrightarrow_p \mathbb{N}^l \text{ und } f = \text{Res}_{\mathcal{C}}\},$$

wobei  $\mathcal{C}$  alle abstrakten Maschinen über der Turingbasismaschine

$$Tur_{(k,l)} := (\mathcal{M}, \{LFT, RHT, PRT, CLR\}, \{BEQ\}, \text{IN}_k, \text{OUT}_l)$$

mit dem Alphabet  $\{\star\}$  durchläuft. Die Klasse der turingberechenbaren Funktionen notieren wir als

$$\mathbf{F}_{Tur}^{(k,l)} := \{f \in \mathbf{P}_{Tur}^{(k,l)} \mid f \text{ ist total}\}.$$

### 2.3 Registermaschinen

Eine weitere abstrakte Maschine erhalten wir aus der Abstrahierung des Abaccus. In vereinfachter Form haben viele von uns derartige Rechenbretter bereits in der Kinderzeit kennengelernt. Sie bestehen aus einem Holzrahmen, in dem parallele Drähte gespannt sind. Auf jedem dieser Drähte befinden sich eine Anzahl von Kugeln, die sich verschieben lassen. Jeder dieser Drähte stellt eine Speichermöglichkeit, ein *Register* dar. Durch Verschieben der Kugeln läßt sich der Inhalt eines Registers

herauf- oder herunterzählen. In Abstraktion dieser Maschine definieren Registermaschinen wir die  $r$ -Registerbasismaschine als

$$RM_r := (\mathbb{N}^r, \mathcal{S}, \mathcal{T}, \text{IN}, \text{OUT}) \quad (2.4)$$

mit  $\mathcal{S} := \{INC_j, DEC_j \mid 0 \leq j \leq r\}$  und  $\mathcal{T} := \{BEQ_j \mid 0 \leq j \leq r\}$ . Dabei zählen die Modifikationsinstruktionen  $INC_j$  das  $j$ -te Register um eines herauf, d.h.

$$INC_j(z_1, \dots, z_j, \dots, z_r) := (z_1, \dots, z_j + 1, \dots, z_r)$$

und  $DEC_j$  das  $j$ -te Register um eines herunter, d.h.

$$DEC_j(z_1, \dots, z_j, \dots, z_r) := (z_1, \dots, z_j - 1, \dots, z_r).$$

Hier haben wir die sogenannte *arithmetische Differenz*  $a \dot{-} b$  verwendet, die definiert ist durch

$$a \dot{-} b := \begin{cases} a - b, & \text{falls } b \leq a, \\ 0, & \text{sonst.} \end{cases}$$

Die Testinstruktionen  $BEQ_j$  testen, ob das  $j$ -te Register leer ist (d.h. die Zahl 0 enthält).

Mit der Registermaschine berechnen wir ebenfalls Funktionen von  $\mathbb{N}^k$  nach  $\mathbb{N}^l$ . Die natürlichen Ein- und Ausgabefunktionen sind daher gegeben durch die Funktionen

$$\text{IN}_k: \mathbb{N}^k \longrightarrow \mathbb{N}^r$$

$$\text{IN}_k(z_1, \dots, z_k) := (z_1, \dots, z_k, \underbrace{0, \dots, 0}_{(r-k)\text{-fach}})$$

und

$$\text{OUT}_l: \mathbb{N}^r \longrightarrow \mathbb{N}^l$$

$$\text{OUT}_l(z_1, \dots, z_r) := (z_1, \dots, z_l),$$

wobei wir, ohne damit die Allgemeinheit wesentlich einzuschränken, stillschweigend  $k, l \leq r$  vorausgesetzt haben. Wir definieren die Klasse der auf  $r$ -Registermaschinen berechenbaren partiellen Funktionen als

$$\mathbf{P}_{r\text{-RM}}^{(k,l)} := \{f \mid f: \mathbb{N}^k \longrightarrow_p \mathbb{N}^l \text{ und } f = \text{Res}_c\},$$

wobei  $\mathcal{C}$  alle abstrakten Maschinen über der Basisregistermaschine

$$(\mathbb{N}^r, \{INC_i, DEC_j \mid 1 \leq i, j \leq r\}, \{BEQ_i \mid 1 \leq i \leq r\}, \text{IN}_k, \text{OUT}_l)$$

durchläuft. Die Klasse der auf  $r$ -Registermaschinen berechenbaren Funktionen ist dann

$$\mathbf{F}_{r\text{-RM}}^{(k,l)} := \{f \in \mathbf{P}_{r\text{-RM}}^{(k,l)} \mid f \text{ ist total}\}.$$

## 2. Maschinenberechenbare Funktionen

---

Mit

$$P_{\text{RM}} = \bigcup \{ P_{r-\text{RM}}^{(k,l)} \mid r \in \mathbb{N}, 1 \leq k, l \leq r \}$$

bezeichnen wir die auf Registermaschinen (bliebiger Registerzahl) berechenbaren partiellen Funktionen und mit  $F_{\text{RM}}$  die in  $P_{\text{RM}}$  liegenden totalen Funktionen.

Die nun naheliegende Frage ist, in wie weit die bislang definierten Klassen maschinenberechenbarer Funktionen übereinstimmen. Um dies zu klären, benötigen wir einige Vorbereitungen.

### 2.4 Simulation abstrakter Maschinen

Die übliche Methode, nachzuweisen, daß für zwei Basismaschinen  $\mathcal{B}_1$  und  $\mathcal{B}_2$ , die Klasse  $P_{\mathcal{B}_2}$  die Klasse  $P_{\mathcal{B}_1}$  umfaßt, besteht darin, die Maschine  $\mathcal{B}_1$  auf der Maschine  $\mathcal{B}_2$  zu simulieren. Das Kernstück der Maschinensimulation besteht in der Definition von Maschinenhomomorphismen. Ein Homomorphismus der Basismaschine

$$\mathcal{B}_1 = (\mathcal{M}_1, \mathcal{S}_1, \mathcal{T}_1, \text{IN}_1, \text{OUT}_1)$$

in die Basismaschine

$$\mathcal{B}_2 = (\mathcal{M}_2, \mathcal{S}_2, \mathcal{T}_2, \text{IN}_2, \text{OUT}_2)$$

besteht aus einem Tripel  $\Phi = (\Phi_1, \Phi_2, \Phi_3)$  von Abbildungen

$$\begin{aligned} \Phi_1: \mathcal{M}_1 &\longrightarrow \mathcal{M}_2 \\ \Phi_2: \mathcal{S}_1 &\longrightarrow \mathcal{S}_2 \\ \Phi_3: \mathcal{T}_1 &\longrightarrow \mathcal{T}_2, \end{aligned}$$

die das in Abbildung 2.4 - 1 gezeigte Diagramm kommutativ machen. Das heißt, daß die folgenden Beziehungen gelten:

$$\begin{aligned} \Phi_1(\text{IN}_1(d)) &= \text{IN}_2(d), & \text{OUT}_2(\Phi_1(z)) &= \text{OUT}_1(z) \\ \Phi_1(S(z)) &= (\Phi_2(S))(\Phi_1(z)), & T(z) &= (\Phi_3(T))(\Phi_1(z)), \end{aligned} \quad (2.5)$$

für  $d \in \mathcal{D}_{in}$ ,  $z \in \mathcal{M}_1$ ,  $S \in \mathcal{S}_1$  und  $T \in \mathcal{T}_1$ .

In Zukunft werden wir die unteren Indizes weglassen und einfach  $\Phi(z)$ ,  $\Phi(S)$  und  $\Phi(T)$  schreiben. Welche Komponente anzuwenden ist, ergibt sich aus dem Zusammenhang.

Sei nun  $\mathcal{C}$  eine  $\mathcal{B}_1$ -Maschine und

$$\Phi: \mathcal{B}_1 \longrightarrow \mathcal{B}_2$$

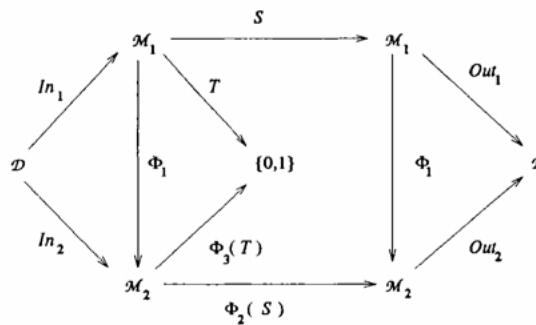


Abb. 2.4 - 1: Diagramm eines Maschinenhomomorphismus

ein Homomorphismus der Basismaschinen. Wir beschreiben, wie sich  $\Phi$  auf die Maschine  $\mathcal{C}$ , deren Konfigurationsraum  $K(\mathcal{C})$  und die darauf operierende Rechenschrittfunktion fortsetzen lässt. Für eine Modifikationsanweisung  $a = (n, S, m)$  sei

$$\Phi(a) := (n, \Phi(S), m)$$

und für eine Testanweisung  $a = (n, T, p, q)$

$$\Phi(a) := (n, \Phi(T), p, q).$$

Für ein Programm  $P = (k_P, A)$ , wobei  $k_P$  die Startmarke und  $A$  die Anweisungsmenge bedeuten, definieren wir

$$\Phi(P) := (k_P, \{\Phi(a) \mid a \in A\}).$$

War  $\mathcal{C} = (\mathcal{B}_1, P)$ , so setzen wir

$$\Phi(\mathcal{C}) := (\mathcal{B}_2, \Phi(P)).$$

Ist nun  $(n, z)$  eine Konfiguration der Maschine  $\mathcal{C}$ , so definieren wir

$$\Phi(n, z) := (n, \Phi(z)),$$

d.h.

$$\pi_2(\Phi(k)) = \Phi(\pi_2(k)) \quad \text{für } k \in K(\mathcal{C}).$$

Damit haben wir

$$RS_{\Phi(\mathcal{C})}(\Phi(n, z)) = \Phi(RS_{\mathcal{C}}(n, z)). \tag{2.6}$$

Da die Endmarken von  $\mathcal{C}$  und  $\Phi(\mathcal{C})$  übereinstimmen, ist  $(e, z)$  genau dann eine Endkonfiguration von  $\mathcal{C}$ , wenn  $\Phi(e, z)$  eine Endkonfiguration von

Fortsetzung von  
Homomorphismen  
von Basismaschinen

## 2. Maschinenberechenbare Funktionen

---

$\Phi(\mathcal{C})$  ist. Die Rechenschrittfunktion von  $\mathcal{C}$  wird aber schrittweise in die von  $\Phi(\mathcal{C})$  übertragen. Damit gilt

$$RZ_{\mathcal{C}}(z) \simeq RZ_{\Phi(\mathcal{C})}(\Phi(z)).$$

Somit ergibt sich für die innere Resultatsfunktion

$$\begin{aligned}\Phi(RES_{\mathcal{C}}(z)) &\simeq \Phi(\pi_2(RS_{\mathcal{C}}^{RZ_{\mathcal{C}}(z)}(k_p, z))) \simeq \pi_2(\Phi(RS_{\mathcal{C}}^{RZ_{\mathcal{C}}(z)}(k_p, z))) \\ &\simeq \pi_2(RS_{\Phi(\mathcal{C})}^{RZ_{\Phi(\mathcal{C})}(\Phi(z))}(k_p, \Phi(z))) \simeq RES_{\Phi(\mathcal{C})}(\Phi(z)),\end{aligned}$$

und damit für die äußere Resultatsfunktion

$$\begin{aligned}Res_{\mathcal{C}}(d) &\simeq OUT_1(RES_{\mathcal{C}}(IN_1(d))) \simeq OUT_2(\Phi(RES_{\mathcal{C}}(IN_1(d)))) \\ &\simeq OUT_2(RES_{\Phi(\mathcal{C})}(\Phi(IN_1(d)))) \simeq OUT_2(RES_{\Phi(\mathcal{C})}(IN_2(d))) \\ &\simeq Res_{\Phi(\mathcal{C})}(d).\end{aligned}$$

Die äußeren Resultatsfunktionen für  $\mathcal{C}$  und  $\Phi(\mathcal{C})$  stimmen demnach überein. Wir haben also den folgenden Satz:

**Satz 5** *Gibt es einen Homomorphismus der Basismaschine  $\mathcal{B}_1$  in die Basismaschine  $\mathcal{B}_2$ , so ist  $\mathbf{P}_{\mathcal{B}_1} \subseteq \mathbf{P}_{\mathcal{B}_2}$  und  $\mathbf{F}_{\mathcal{B}_1} \subseteq \mathbf{F}_{\mathcal{B}_2}$ .*

Satz 5 gibt uns bereits eine Möglichkeit, Klassen von auf unterschiedlichen Basismaschinen berechenbaren Funktionen zu vergleichen. Allerdings ist diese Möglichkeit noch sehr eingeschränkt, verlangt sie doch, daß jeder Instruktion  $i$  auf der einen Maschine genau eine Instruktion  $\Phi(i)$  auf der anderen entspricht. Anschaulich ist es jedoch völlig klar, daß man zur gleichen Berechenbarkeitsklasse gelangen sollte, wenn man jede Instruktion  $i$  auf der einen Maschine durch ein Programm auf der anderen simulieren kann. Um dies zu präzisieren, führen wir einige Begriffe ein.

Ist  $P$  ein Programm auf einer Basismaschine

$$\mathcal{B} = (\mathcal{M}, \mathcal{S}, \mathcal{T}, IN, OUT),$$

so nennen wir  $P$  eine *funktionale Prozedur*, wenn  $\text{dom}(RES_P) = \mathcal{M}$  ist. Wir nennen  $P$  eine *relationale Prozedur*, wenn  $RES_P$  die Identität auf  $\mathcal{M}$  ist und  $P$  genau zwei verschiedene Endmarken  $e_0$  und  $e_1$  besitzt.

Wir können uns nun eine Erweiterung der Basismaschine  $\mathcal{B}$  konstruieren, indem wir für jede funktionale Prozedur  $P$  die Resultatsfunktion  $RES_P$  als neue Modifikationsinstruktion hinzunehmen und zu jeder relationalen Prozedur  $P$  eine neue Testinstruktion  $T_p$  vermöge

$$T_p(z) := \begin{cases} 0, & \text{falls } RS_P^{RZ_{\mathcal{C}}(z)}(k_p, z) = (e_0, z), \\ 1, & \text{falls } RS_P^{RZ_{\mathcal{C}}(z)}(k_p, z) = (e_1, z) \end{cases}$$

## 2.5. Simulation der Registermaschine auf der Turingmaschine

definieren. Diese erweiterte Maschine nennen wir den *Abschluß* der Maschine  $\mathcal{B}$  und bezeichnen ihn mit  $Cl(\mathcal{B})$ . Nach ihrer Konstruktion berechnen  $\mathcal{B}$  und  $Cl(\mathcal{B})$  die gleichen Funktionen. Jede auf  $\mathcal{B}$  berechenbare totale Funktion wird auf  $Cl(\mathcal{B})$  sogar in einem Schritt berechnet. Also haben wir

$$P_{\mathcal{B}} = P_{Cl(\mathcal{B})} \quad (2.7)$$

und

$$F_{\mathcal{B}} = F_{Cl(\mathcal{B})}. \quad (2.8)$$

Nun sind wir in der Lage, die Maschinensimulation zu definieren.

**Definition 6** Eine Simulation einer Basismaschine  $\mathcal{B}_1$  auf einer Basismaschine  $\mathcal{B}_2$  ist ein Homomorphismus

$$\Phi: \mathcal{B}_1 \longrightarrow Cl(\mathcal{B}_2).$$

Als eine Folgerung von Satz 5, (2.7) und (2.8) erhalten wir dann

**Satz 7** Existiert eine Simulation der Basismaschine  $\mathcal{B}_1$  auf der Basismaschine  $\mathcal{B}_2$ , so gilt

$$P_{\mathcal{B}_1} \subseteq P_{\mathcal{B}_2},$$

und

$$F_{\mathcal{B}_1} \subseteq F_{\mathcal{B}_2}.$$

## 2.5 Simulation der Registermaschine auf der Turingmaschine

Um einzusehen, daß alle auf Registermaschinen berechenbaren Funktionen auch TURING-berechenbar sind, genügt es nach dem vorherigen Abschnitt, die Registermaschine auf einer Turingmaschine zu simulieren. Wir gehen also von einer  $r$ -Registermaschine  $\mathcal{R}$  aus. Die Menge der Speicherzustände von  $\mathcal{R}$  wird durch  $\mathbb{N}^r$  beschrieben. Die Bandinschriften der Turingmaschine wollen wir wieder mit  $\mathcal{M}$  notieren. Wir definieren eine Abbildung

$$\begin{aligned} \Phi: \mathbb{N}^r &\longrightarrow \mathcal{M} \\ (z_1, \dots, z_r) &\longmapsto B^\infty \underline{B} *^{z_1} B \dots B *^{z_r} B^\infty \end{aligned} \quad (2.9)$$

und wollen diese zu einem Homomorphismus in den Abschluß der Basis-turingmaschine fortsetzen. Dazu definieren wir

Der Abschluß  
abstrakter  
Basismaschinen

Simulation  
abstrakter  
Maschinen

## 2. Maschinenberechenbare Funktionen

---

$$\Phi(INC_i)(B^\infty u \underline{z} \star^{z_1} B \dots B \star^{z_i} B v B^\infty) := (B^\infty u \underline{z} \star^z B \dots B \star^{z_i+1} B v B^\infty),$$

$$\Phi(DEC_i)(B^\infty u \underline{z} \star^{z_1} B \dots B \star^{z_i} B v B^\infty) := (B^\infty u \underline{z} \star^{z_1} B \dots B \star^{z_i-1} B v B^\infty)$$

und

$$\Phi(BEQ_i)(B^\infty u \underline{z} \star^{z_1} B \dots B \star^{z_i} B v B^\infty) = \begin{cases} 0, & \text{falls } z_i = 0, \\ 1, & \text{falls } z_i = 1. \end{cases}$$

Die in (2.9) definierte Abbildung ist genau die in (2.2) definierte Inputfunktion für die Basisturingmaschine. Damit folgt

$$IN_{TUR} = \Phi \circ IN_{RM}$$

und

$$OUT_{RM} = OUT_{TUR} \circ \Phi,$$

wobei  $\circ$  die Hintereinanderausführung von Funktionen bedeutet, d.h. für zwei Funktionen  $f$  und  $g$  ist  $(f \circ g)(x) := f(g(x))$ .

Unmittelbar aus den Definitionen erhalten wir auch

$$\Phi(INC_i(z)) = (\Phi(INC_i))(\Phi(z)), \quad \Phi(DEC_i(z)) = (\Phi(DEC_i))(\Phi(z))$$

und

$$BEQ_i(z) = (\Phi(BEQ_i))(\Phi(z)).$$

Alles, was daher zu einer Simulation der Registermaschine auf der Turingmaschine noch fehlt, ist der Nachweis, daß  $\Phi(INC_j)$  und  $\Phi(DEC_j)$  funktionale Prozeduren und  $\Phi(BEQ_j)$  relationale Prozeduren auf der Turingmaschine sind.

Hilfsprozeduren für Registermaschinen

Um dies einzusehen, definieren wir uns zunächst einige Hilfsprozeduren auf der Turingmaschine. Zum einen benötigen wir eine *große Rechtsverschiebung RHS*, die durch das Flußdiagramm in Abbildung 2.5 - 1 gegeben ist. Man überzeugt sich leicht, daß RHS das Arbeitsfeld genau um ein Wort nach rechts verschiebt, d.h.

$$RHS(\dots \underline{b} \star^n B \dots) = \dots b \star^n \underline{B} \dots$$

für  $n \geq 0$ . Analog definiert man eine *große Linkverschiebung LFS* mit der Wirkung

$$LFS(\dots B \star^n \underline{b} \dots) = \dots \underline{B} \star^n b \dots$$

Programme für  $\Phi(INC_j)$ ,  $\Phi(DEC_j)$  und  $\Phi(BEQ_j)$  erhalten wir, wie in den Abbildungen 2.5 - 2, 2.5 - 3 und 2.5 - 4 angegeben. Also gilt:

## 2.5. Simulation der Registermaschine auf der Turingmaschine

---

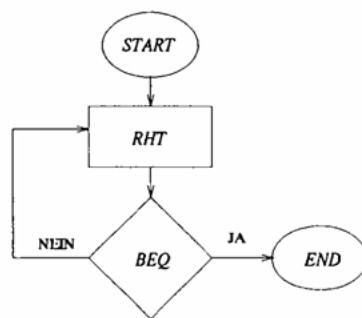


Abb. 2.5 - 1: Ein Programm für die Rechtsverschiebung RHS

---

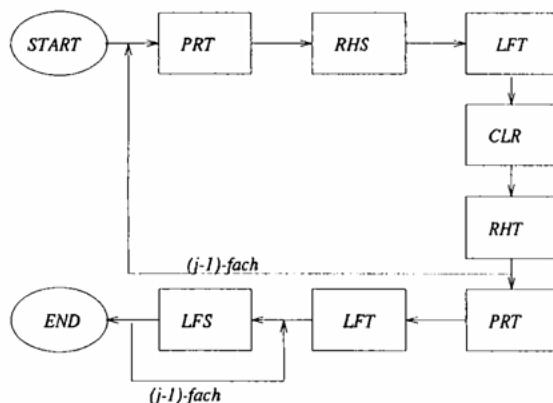


Abb. 2.5 - 2: Programm für  $\Phi(INC_j)$

---

**Satz 8** Jede auf einer  $r$ -Registermaschine berechenbare partielle oder totale Funktion ist auch auf der Turingmaschine berechenbar.

Registermaschinen-berechenbare  
Funktionen sind  
turingberechenbar

Man kann nun die Gegenrichtung des Satzes 8 in analoger Weise durch eine Simulation der Turingbasismaschine auf der Registermaschine beweisen. In dieser Übersicht wollen wir jedoch einen anderen Weg einschlagen, der uns gleichzeitig weitere Einsichten in die Natur berechenbarer Funktionen erlaubt. Wir werden versuchen, eine mehr mathematisch motivierte Theorie der Berechenbarkeit zu entwickeln, und so zu der Klasse  $P$  der  $\mu$ -partiellrekursiven Funktionen zu gelangen. Dann werden wir sehen, daß jede turingberechenbare partielle Funktion schon  $\mu$ -partiellrekursiv ist, und sich umgekehrt jede  $\mu$ -partiellrekursive Funktionen leicht auf einer Registermaschine berechnen läßt.

## 2. Maschinenberechenbare Funktionen

---

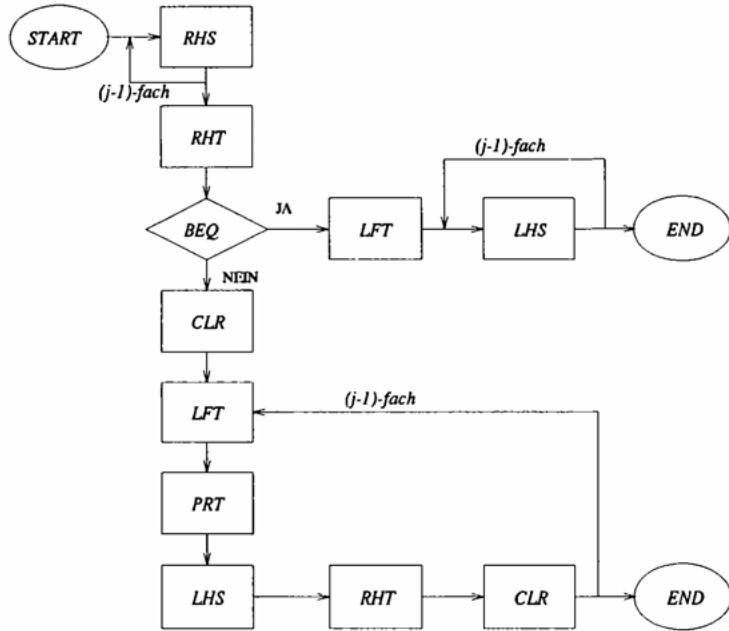


Abb. 2.5 - 3: Programm für  $\Phi(DEC_j)$

---

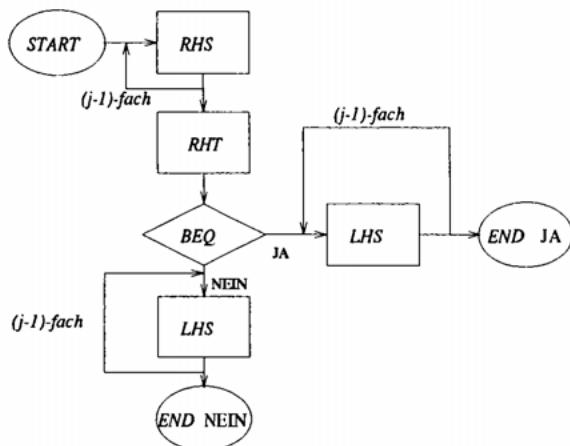


Abb. 2.5 - 4: Programm für  $\Phi(BEQ_j)$

---

---

## 3. $\mu$ -rekursive Funktionen

Der Weg zu einer exakten Definition des Begriffes einer berechenbaren Funktion, den wir in diesem Kapitel beschreiten wollen, besteht darin, daß wir von sehr einfachen Funktionen ausgehen, die ganz offensichtlich berechenbar sind. Wir überlegen uns dann Operationen, die berechenbare Funktionen in berechenbare Funktionen überführen und schließen die Grundfunktionen unter diesen Operationen ab. Es bleibt dann festzustellen, inwieweit wir damit die intuitiv berechenbaren Funktionen erfaßt haben. Insbesondere wird die so erhaltene Klasse von Funktionen mit den maschinenberechenbaren zu vergleichen sein.

### 3.1 Primitiv rekursive Funktionen

Bei den folgenden Untersuchungen wollen wir uns auf Funktionen beschränken, die Tupel natürlicher Zahlen auf natürliche Zahlen abbilden. Wir werden später einsehen, daß dies keine wesentliche Einschränkung der Allgemeinheit bedeutet.

Eine natürliche Zahl ist dadurch charakterisiert, daß sie entweder 0 oder der Nachfolger einer anderen natürlichen Zahl ist. Zur Darstellung natürlicher Zahlen genügt daher ein Symbol für die 0 zusammen mit einem Symbol S für die Nachfolgerfunktion. Wir können dann jede natürliche Zahl  $n$  durch eine Ziffer

$$\underline{n} = (\underbrace{S(\dots S(0)\dots)}_{n\text{-fach}})$$

repräsentieren. Um die Schreibweise nicht zu verwirrend zu gestalten, werden wir allerdings nicht immer peinlich zwischen natürlichen Zahlen und ihrer Zifferndarstellung unterscheiden.

Wenn wir offensichtlich berechenbare Funktionen suchen, denken wir sofort an Funktionen wie die *konstanten Funktionen*, die jedem Tupel von Argumenten immer eine konstante natürliche Zahl zuordnen oder die *Projektionsfunktionen*, die jedes Tupel  $(x_1, \dots, x_n)$  auf eine seiner Komponenten projizieren. Auch die Nachfolgerfunktion ist offensichtlich eine berechenbare Funktion. Natürlich fallen uns daneben noch eine ganze Reihe weiterer Funktionen ein, die uns als intuitiv berechenbar geläufig sind. Wir werden jedoch sehen, daß die eben genannten Funktionen bereits ausreichen. Daher treffen wir die folgende Festsetzung:

### 3. $\mu$ -rekursive Funktionen

---

Grundfunktionen

**Definition 9** (*Klasse der Grundfunktionen*)

Grundfunktionen sind die konstanten Funktionen  $C_k^n$  mit

$$C_k^n(x_1, \dots, x_n) := k,$$

die Projektionsfunktionen  $P_k^n$  mit

$$P_k^n(x_1, \dots, x_n) := x_k$$

für  $1 \leq k \leq n$  und die Nachfolgerfunktion  $S$ .

Substitution

Eine Operation, die berechenbare Funktionen wieder in berechenbare Funktionen überführt, ist durch die *Komposition* zweier Funktionen oder die *Substitution* von Funktionen gegeben, wie man diese Operation auch nennt, falls mehr als zwei Funktionen beteiligt sind. Sind

$$g: \mathbb{N}^m \longrightarrow \mathbb{N}$$

und

$$h_i: \mathbb{N}^n \longrightarrow \mathbb{N}$$

für  $i = 1, \dots, m$  gegeben, so ist deren Komposition (bzw. die Substitution von  $h_1, \dots, h_m$  in  $g$ ) definiert durch

$$\text{Sub}(g, h_1, \dots, h_m)(\vec{x}) := g(h_1(\vec{x}), \dots, h_m(\vec{x})), \quad (3.1)$$

wobei  $\vec{x}$  ein beliebiges  $n$ -Tupel natürlicher Zahlen bedeutet.

Im Falle von  $m = 1$  werden wir aber anstelle von  $\text{Sub}(g, h)$  weiterhin  $g \circ h$  schreiben.

Primitive Rekursion

Eine etwas komplexere Operation ist durch die *primitive Rekursion* gegeben. Hier gehen wir von zwei Funktionen

$$g: \mathbb{N}^n \longrightarrow \mathbb{N}$$

und

$$h: \mathbb{N}^{n+2} \longrightarrow \mathbb{N}$$

aus und definieren die durch primitive Rekursion aus ihnen hervorgegangene Funktion

$$\text{Rec}(g, h): \mathbb{N}^{n+1} \longrightarrow \mathbb{N}$$

durch

$$\text{Rec}(g, h)(z, \vec{x}) := \begin{cases} g(\vec{x}), & \text{falls } z = 0, \\ h(z_0, \text{Rec}(g, h)(z_0, \vec{x}), \vec{x}), & \text{falls } z = S(z_0), \end{cases} \quad (3.2)$$

wobei  $\vec{x}$  wieder ein beliebiges  $n$ -Tupel natürlicher Zahlen bedeutet.

Hier bedarf es eines Augenblickes des Nachdenkens, um einzusehen, daß mit berechenbaren  $g$  und  $h$  auch  $\text{Rec}(g, h)$  berechenbar ist. Der Algorithmus zur Berechnung von  $\text{Rec}(g, h)(z, x_1, \dots, x_n)$  läßt sich folgendermaßen beschreiben:

Definieren wir

$$e_0 = g(\vec{x})$$

und

$$e_{z+1} := h(z, e_z, \vec{x}),$$

so lassen sich die Werte  $e_z$  für  $z = 0, 1, \dots$  sukzessive berechnen, da wir die Funktionen  $g$  und  $h$  als berechenbar vorausgesetzt haben. Durch Induktion nach  $z$  läßt sich nun aber leicht einsehen, daß

$$\text{Rec}(g, h)(z, \vec{x}) = e_z$$

ist. Damit erhalten wir den Wert von  $\text{Rec}(g, h)(z, \vec{x})$  durch Berechnung von  $e_0, e_1, \dots, e_z$ .

Wir nennen die Operationen **Sub** und **Rec** die *Grundoperationen*. Grundoperationen

**Definition 10** Die Klasse **PR** der primitiv rekursiven Funktionen ist die kleinste Klasse von Funktionen, die die Grundfunktionen enthält und die gegenüber den Grundoperationen abgeschlossen ist. Primitiv rekursive Funktionen

Hier erscheint es uns nötig zu sein, etwas zu dem Unterschied zwischen *extensionaler* Auffassung und *intensionaler* Auffassung von Funktionen zu bemerken. In der Mathematik werden Funktionen

$$f: A \longrightarrow B$$

üblicherweise als durch ihre *Graphen*

$$G_f := \{(a, f(a)) \mid a \in A\}$$

Extensionale Auffassung von Funktionen

beschriebene *Mengen* aufgefaßt. Da Mengen allein durch ihre Extension, d.h. durch ihre Elemente bestimmt sind, nennen wir zwei Funktionen extensional gleich, wenn ihre Graphen als Mengen übereinstimmen. Wir notieren diese Gleichheit durch

$$f = g$$

und haben

$$f = g \iff \text{dom}(f) = \text{dom}(g) \wedge (\forall x \in \text{dom}(f)) [f(x) = g(x)].$$

Bislang haben wir die Gleichheit von Funktionen auch immer so aufgefaßt. Für die primitiv rekursive Funktion

### 3. $\mu$ -rekursive Funktionen

---

$$F \equiv \text{Sub}(\mathbf{P}_1^2, \mathbf{P}_1^1, \mathbf{P}_1^1) \quad (*)$$

gilt dann offenbar

$$F = \mathbf{P}_1^1,$$

denn es ist  $\text{dom}(F) = \text{dom}(\mathbf{P}_1^1) = \mathbb{N}$  und nach (3.1) gilt

$$F(x) = \mathbf{P}_1^2(\mathbf{P}_1^1(x), \mathbf{P}_1^1(x)) = \mathbf{P}_1^2(x, x) = x = \mathbf{P}_1^1(x),$$

obwohl  $F$  und  $\mathbf{P}_1^1$ , als *Terme* betrachtet, verschieden sind. Wir haben schon durch das Zeichen  $\equiv$  in  $(*)$  zum Ausdruck bringen wollen, daß  $F$  nicht bloß extensional, sondern tatsächlich als Term gleich  $\text{Sub}(\mathbf{P}_1^2, \mathbf{P}_1^1, \mathbf{P}_1^1)$  sein soll. Dieser *intensionalen Gleichheit* liegt eine andere Auffassung über das Wesen einer Funktion zugrunde. Hier geht man davon aus, daß eine Funktion nicht im mengentheoretischen Sinn als ihr Graph verstanden werden darf, sondern nur durch ihren Algorithmus gegeben ist. Da der Aufbau des Termes uns den Algorithmus zur Berechnung der dazugehörigen Funktion liefert, läßt sich die intensionale Gleichheit von Funktionen durch die Gleichheit der repräsentierenden Terme in befriedigender Weise beschreiben. Wir werden in Zukunft die intensionale Gleichheit durch  $\equiv$  notieren, obwohl diese hier nur selten eine Rolle spielen wird.

Intensionale  
Auffassung von  
Funktionen

Um völlige mathematische Strenge zu wahren, hätten wir eigentlich von Termen für primitiv rekursive Funktionen ausgehen müssen, die sich induktiv durch die folgenden Klauseln definieren lassen:

*Für alle natürlichen Zahlen  $n$  und  $k$  ist  $\mathbf{C}_k^n$  ein  $n$ -stelliger primitiv rekursiver Funktionsterm.*

*Für alle natürlichen Zahlen  $n$  und alle  $k \in \{1, \dots, n\}$  ist  $\mathbf{P}_k^n$  ein  $n$ -stelliger primitiv rekursiver Funktionsterm.*

*Ist  $g$  ein  $m$ -stelliger primitiv rekursiver Funktionsterm und sind  $h_1, \dots, h_m$   $n$ -stellige primitiv rekursive Funktionsterme, so ist auch  $\text{Sub}(g, h_1, \dots, h_m)$  ein  $n$ -stelliger primitiv rekursiver Funktionsterm.*

*Ist  $g$  ein  $n$ -stelliger und  $h$  ein  $n+2$ -stelliger primitiv rekursiver Funktionsterm, so ist  $\text{Rec}(g, h)$  ein  $n+1$ -stelliger primitiv rekursiver Funktionsterm.*

Ist nun  $T$  ein  $n$ -stelliger primitiv rekursiver Funktionsterm und sind  $z_1, \dots, z_n$  Ziffern, so läßt sich durch die folgenden Klauseln die Auswertung  $\text{Val}(T, z_1, \dots, z_n)$  definieren:

$$\text{Val}(\mathbf{C}_k^n, z_1, \dots, z_n) = k$$

$$\text{Val}(\mathbf{P}_k^n, z_1, \dots, z_n) = z_k$$

$$\text{Val}(\text{Sub}(g, h_1, \dots, h_m), z_1, \dots, z_n) = \text{Val}(g, \text{Val}(h_1, z_1, \dots, z_n), \dots, \text{Val}(h_m, z_1, \dots, z_n))$$

$$\text{Val}(\text{Rec}(g, h), 0, z_1, \dots, z_n) = \text{Val}(g, z_1, \dots, z_n)$$

$$\text{Val}(\text{Rec}(g, h), \mathbf{S}(n), z_1, \dots, z_n) = \text{Val}(h, \text{Val}(\text{Rec}(g, h), n, z_1, \dots, z_n), z_1, \dots, z_n)$$

Über seine Auswertung können wir somit jeden  $n$ -stelligen primitiv rekursiven Funk-

tionsterm als eine  $n$ -stellige Funktion auffassen (was wir auch immer tun werden) und erhalten, daß eine Funktion genau dann primitiv rekursiv ist, wenn es einen primitiv rekursiven Funktionsterm gibt, dem sie extensional gleich ist. Wie wir bereits am obigen einfachen Beispiel gesehen haben, können verschiedene Funktionsterme durchaus die gleiche Funktion liefern. Später werden wir einsehen, daß es sogar unendlich viele verschiedene solche Funktionsterme geben muß.

Die explizite Angabe des Funktionstermes, der eine primitiv rekursive Funktion definiert, stellt sich im allgemeinen jedoch als recht umständlich heraus. Wir können uns dies am Beispiel der *Addition* natürlicher Zahlen klarmachen. Die Addition gehorcht bekanntlich den Rekursionsgleichungen

$$x + 0 = x \quad (= P_1^1(x)) \quad (3.3)$$

und

$$x + S(z) = S(x + z). \quad (3.4)$$

Ein Funktionsterm, der die Addition repräsentiert, ist beispielsweise

$$\oplus := \text{Rec}(P_1^1, \text{Sub}(S, P_2^3)).$$

Die Tatsache, daß

$$\oplus(z, x) = x + z$$

gilt, erhalten wir leicht durch Induktion nach  $z$ . Für  $z = 0$  ist

$$\oplus(0, x) = P_1^1(x) = x = x + 0$$

und für  $z = S(z_0)$  ist

$$\oplus(z, x) = \text{Sub}(S, P_2^3)(z_0, (x + z_0), x) = S(x + z_0) = x + z.$$

Damit ist die Addition natürlicher Zahlen als primitiv rekursiv nachgewiesen. Man kann sich jedoch überlegen, daß bereits die Kenntnis der Rekursionsgleichungen (3.3) und (3.4) zum Nachweis der primitiven Rekursivität der Addition ausgereicht hätte. Dazu zeigt man durch Induktion nach  $z$  etwas allgemeiner, daß jede Funktion  $f$ , die den Rekursionsgleichungen

$$f(0, \vec{x}) = g(\vec{x}) \quad (3.5)$$

und

$$f(S(z), \vec{x}) = h(z, f(z, \vec{x}), \vec{x}) \quad (3.6)$$

gehorcht, der durch den Term  $\text{Rec}(g, h)$  repräsentierten Funktion (ex-

Die Addition  
natürlicher Zahlen  
ist primitiv rekursiv

### 3. $\mu$ -rekursive Funktionen

---

tensional) gleich ist. Damit ist  $f$  insbesondere (extensional) eindeutig bestimmt. Unter der Voraussetzung, daß die Funktionen  $g$  und  $h$  in (3.5) und (3.6) primitiv rekursiv sind – und somit durch primitiv rekursive Funktionsterme repräsentiert werden können –, folgt damit also, daß auch  $f$  eine primitiv rekursive Funktion ist, – die dann durch den Term  $\text{Rec}(g, h)$  repräsentiert wird.

Da in den Rekursionsgleichungen (3.3) und (3.4) nur die primitiv rekursiven Funktionen  $P_1^1$  und  $S$  auftreten, folgt aus ihnen bereits die primitive Rekursivität der Addition.

Wir wollen die gleiche Argumentation verwenden, um nachzuweisen, daß auch die Multiplikation eine primitiv rekursive Funktion ist. Die Multiplikation gehorcht offenbar den Rekursionsgleichungen

$$a \cdot 0 = 0$$

und

$$a \cdot (S(z)) = (a \cdot z) + a.$$

Die Multiplikation  
natürlicher Zahlen  
ist primitiv rekursiv

Exponentialfunktion

Nachdem wir die Addition bereits als primitiv rekursiv nachgewiesen haben, folgt daraus, daß es einen primitiv rekursiven Funktionsterm geben muß, der die Multiplikation repräsentiert. In ähnlicher Weise erhalten wir die primitive Rekursivität der Exponentialfunktion, die den Rekursionsgleichungen

$$\exp(a, 0) = 1$$

und

$$\exp(a, S(z)) = \exp(a, z) \cdot a$$

Super-  
Exponentiation

gehört. Man kann das Verfahren iterieren, indem man eine *Super-Exponentialfunktion* durch

$$\text{sexp}(a, 0) = 1$$

und

$$\text{sexp}(a, S(z)) = \exp(a, \text{sexp}(a, z))$$

definiert. Wir erhalten dann

$$\text{sexp}(a, n) = \exp(a, \underbrace{(\exp(a, (\dots \exp(a, 0) \dots)))}_{n\text{-fach}}).$$

Dies Verfahren läßt sich noch weiter fortsetzen, indem man eine *Super-Super-Exponentialfunktion*, *Super-Super-Super-Exponentialfunktion*, u.s.f. definiert. Es ist bereits daraus zu sehen, daß die primitiv rekursiven Funktionen Funktionen mit ungeheuer

starkem Wachstum umfassen. Eine Berechnung aller primitiv rekursiven Funktionen auf physikalisch realisierbaren Maschinen ist somit prinzipiell unmöglich, da sie alle physikalisch gegebenen Grenzen sprengen würde. Eine physikalische Grenze wäre bei Zugrundelegung einer Speicherung mittels elektrischer Ladungen durch die im Universum vorhandenen Ladungsträger gesetzt. Das Wachstum primitiv rekursiver Funktionen würde deren Zahl schnell überschreiten, selbst wenn wir davon ausgehen, daß bis heute nur Bruchteile der im Universum existierenden Elektronenmassen tatsächlich entdeckt sind. Da es uns im Augenblick allerdings nur darauf ankommt, den Begriff der berechenbaren Funktion *im allgemeinen* zu klären, wollen wir hier auf die Möglichkeiten einer physikalischen Realisation der prinzipiellen Berechenbarkeit noch nicht eingehen.

Beispiele weiterer primitiv rekursiver Funktionen, die in unseren Untersuchungen eine Rolle spielen werden, sind:

die *Fakultätsfunktion*  $a!$  mit den Rekursionsgleichungen

$$0! = 1$$

Weitere Beispiele  
primitiv rekursiver  
Funktionen

und

$$(\mathbf{S}(z))! = z! \cdot \mathbf{S}(z),$$

die *Fallunterscheidungsfunktionen*  $\text{sg}$  und  $\overline{\text{sg}}$  mit den Rekursionsgleichungen

$$\text{sg}(0) = 0 \quad \text{sg}(\mathbf{S}(z)) = 1$$

und

$$\overline{\text{sg}}(0) = 1 \quad \overline{\text{sg}}(\mathbf{S}(z)) = 0,$$

die *Vorgängerfunktion*  $\text{Pd}$  mit den Rekursionsgleichungen

$$\text{Pd}(0) = 0 \quad \text{und} \quad \text{Pd}(\mathbf{S}(z)) = z,$$

die *arithmetische Differenz* mit den Rekursionsgleichungen

$$a \div 0 = a \quad a \div \mathbf{S}(z) = \text{Pd}(a \div z).$$

Wir wollen noch bemerken, daß *endliche Summen* und *endliche Produkte* als Funktionen ihrer oberen Grenzen primitiv rekursiv sind. Auch dies folgt sofort durch Hinschreiben der Rekursionsgleichungen

$$\sum_{i=0}^0 f(i, \vec{x}) = f(0, \vec{x}) \quad \sum_{i=0}^{\mathbf{S}(z)} f(i, \vec{x}) = \left( \sum_{i=0}^z f(i, \vec{x}) \right) + f(\mathbf{S}(z), \vec{x})$$

und

$$\prod_{i=0}^0 f(i, \vec{x}) = f(0, \vec{x}) \quad \prod_{i=0}^{\mathbf{S}(z)} f(i, \vec{x}) = \left( \prod_{i=0}^z f(i, \vec{x}) \right) \cdot f(\mathbf{S}(z), \vec{x}).$$

### 3. $\mu$ -rekursive Funktionen

---

Weitere primitiv rekursive Funktionen werden wir bei Bedarf zur Verfügung stellen.

## 3.2 Primitiv rekursive Prädikate

Neben der Berechenbarkeit von Funktionen spielt die *Entscheidbarkeit* von *Prädikaten* eine bedeutende Rolle. Dabei werden Prädikate in der Regel extensional, d.h. als Mengen aufgefaßt. Dies ist der Hintergrund der folgenden Definition:

**Definition 11** Unter einem  $n$ -stelligen zahlentheoretischen Prädikat  $P$  verstehen wir eine Menge  $P \subseteq \mathbb{N}^n$ .

Wir werden Prädikate oft auch als *Relationen* bezeichnen und anstatt  $(x_1, \dots, x_n) \in P$  oft  $P(x_1, \dots, x_n)$  schreiben. Es hat sich eingebürgert, diese Ausdrücke völlig synonym zu verwenden.

Das *Entscheidungsproblem* für ein Prädikat  $P$  ist dann die Frage, ob es einen Algorithmus gibt, der

$$\vec{x} \in P?$$

entscheidet. Die *charakteristische Funktion* eines Prädikats  $P$  ist gegeben durch

$$\chi_P(\vec{x}) := \begin{cases} 1, & \text{falls } \vec{x} \in P, \\ 0, & \text{falls } \vec{x} \notin P. \end{cases}$$

Offenbar läßt sich das Entscheidungsproblem für ein Prädikat  $P$  sofort auf das Berechnungsproblem für dessen charakteristische Funktion  $\chi_P$  zurückführen. Daher nennen wir ein Prädikat *primitiv rekursiv*, wenn dessen charakteristische Funktion  $\chi_P$  primitiv rekursiv ist.

Für primitiv rekursive Prädikate können wir also das Entscheidungsproblem in primitiv rekursiver Weise lösen.

Ist nun  $P$  ein Prädikat, so ist dessen *Komplement* die Menge

$$\mathcal{C}(P) := \{x \mid x \notin P\}.$$

Die charakteristische Funktion von  $\mathcal{C}(P)$  ist ganz offensichtlich durch

$$\chi_{\mathcal{C}(P)} = \overline{\text{sg}} \circ \chi_P$$

gegeben. Für primitiv rekursives  $P$  ist somit auch  $\mathcal{C}(P)$  primitiv rekursiv. Ein ähnliches Ergebnis erhalten wir auch für Vereinigungen und Durchschnitte primitiv rekursiver Prädikate. Hier beobachten wir, daß

$$\chi_{P \cap Q}(\vec{x}) = \chi_P(\vec{x}) \cdot \chi_Q(\vec{x})$$

und

$$\chi_{P \cup Q}(\vec{x}) = \text{sg}(\chi_P(\vec{x}) + \chi_Q(\vec{x}))$$

ist. Damit gilt:

*Die primitiv rekursiven Prädikate sind gegenüber Komplementen, endlichen Vereinigungen und endlichen Durchschnitten abgeschlossen.*

Abschlußeigenschaften primitiv rekursiver Prädikate

Wählen wir die Schreibweise  $P(\vec{x})$  anstatt  $\vec{x} \in P$ , so haben wir

$$\vec{x} \in \mathcal{C}(P) \Leftrightarrow \neg P(\vec{x}),$$

$$\vec{x} \in P \cup Q \Leftrightarrow P(\vec{x}) \vee Q(\vec{x})$$

und

$$\vec{x} \in P \cap Q \Leftrightarrow P(\vec{x}) \wedge Q(\vec{x}).$$

Daher sprechen wir auch davon, daß die primitiv rekursiven Prädikate gegenüber den logischen Operationen  $\neg$ ,  $\vee$  und  $\wedge$ , die oft auch als die *boolesche Operationen* (vgl. Abschnitt 6.3) bezeichnet werden, abgeschlossen sind.

Boolesche Operationen

Ein  $m$ -stelliges Prädikat  $Q$  entsteht aus einem  $n$ -stelligen Prädikat  $P$  durch Substitution mit den Funktionen  $f_1, \dots, f_n$ , wenn die  $f_i$  alle  $m$ -stellig sind und

$$Q = \{\vec{x} \in \mathbb{N}^m \mid (f_1(\vec{x}), \dots, f_n(\vec{x})) \in P\} \quad (3.7)$$

ist. Dann gilt

$$\chi_Q = \text{Sub}(\chi_P, f_1, \dots, f_n)$$

und wir erhalten somit:

*Die primitiv rekursiven Prädikate sind abgeschlossen gegenüber Substitutionen mit primitiv rekursiven Funktionen.*

Das Prädikat  $Q$  in (3.7) bezeichnen wir mit  $\text{Sub}(P, f_1, \dots, f_n)$ .

Neben den bisher behandelten Operationen auf Prädikaten spielt auch der *beschränkte Suchoperator* eine wichtige Rolle. Wir wollen diesen mit  $\mu_b$  bezeichnen und wie folgt definieren:

Beschränkter Suchoperator

$$(\mu_b P)(\vec{x}, y) := \begin{cases} \min\{z \leq y \mid (\vec{x}, z) \in P\}, & \text{falls dies existiert,} \\ 0, & \text{sonst.} \end{cases}$$

Der *beschränkte Suchoperator*  $(\mu_b P)(\vec{x}, y)$  sucht demnach nach dem kleinsten  $z \leq y$  mit  $P(\vec{x}, z)$  und bricht die Suche unter Ausgabe des Wertes 0 ab, wenn er kein solches finden kann.

### 3. $\mu$ -rekursive Funktionen

---

Eine deutlich anschaulichere, wenn auch weniger exakte Schreibweise für den beschränkten Suchoperator ist

$$\mu z \leq y. (\vec{x}, z) \in P,$$

der wir im folgenden oft den Vorzug geben werden.

Für ein  $n + 1$ -stelliges Prädikat  $P$  ist  $(\mu_b P)$  eine  $n + 1$ -stellige Funktion. Die Funktion  $(\mu_b P)$  gehorcht den Rekursionsgleichungen

$$(\mu_b P)(\vec{x}, 0) = 0$$

und

$$(\mu_b P)(\vec{x}, S(n)) = (\mu_b P)(\vec{x}, n) \\ + S(n) \cdot \chi_P(\vec{x}, S(n)) \cdot \overline{sg}((\mu_b P)(\vec{x}, n)) \cdot \overline{sg}(\chi_P(\vec{x}, 0)),$$

wie man leicht nachrechnet. Damit folgt:

*Für ein  $n$ -stelliges primitiv rekursives Prädikat ist  $(\mu_b P)$  eine  $n$ -stellige primitiv rekursive Funktion.*

Wir wollen diese Tatsache benutzen, um nachzuweisen, daß die primitiv rekursiven Prädikate auch gegenüber beschränkter Quantifikation abgeschlossen sind. Ein Prädikat  $Q$  entsteht aus  $P$  durch beschränkte  $\forall$ -Quantifikation, falls

$$Q = \{(\vec{x}, n) \mid (\forall y \leq n)[(\vec{x}, y) \in P]\}$$

und durch beschränkte  $\exists$ -Quantifikation, falls

$$Q = \{(\vec{x}, n) \mid (\exists y \leq n)[(\vec{x}, y) \in P]\}$$

gilt. Die charakteristische Funktion für das durch beschränkte  $\exists$ -Quantifikation erhaltene Prädikat, das wir mit  $\exists^{\leq}(P)$  bezeichnen wollen, ist gegeben durch

$$\chi_{\exists^{\leq}(P)}(\vec{x}, n) = sg(\mu y \leq n. (\vec{x}, y) \in P) + \chi_P(\vec{x}, 0).$$

Wegen

$$(\forall y \leq n)[(\vec{x}, y) \in P] \iff \neg(\exists y \leq n)[(\vec{x}, y) \in C(P)]$$

erhalten wir

$$\forall^{\leq}(P) = C(\exists^{\leq}(C(P))).$$

Da die primitiv rekursiven Prädikate gegenüber Komplementen abgeschlossen sind, haben wir zusammenfassend:

*Die primitiv rekursiven Prädikate sind abgeschlossen gegenüber beschränkter Quantifikation.*

Prädikat	primitiv rekursive Definition
$x = y$	$\chi_=(x, y) =  x - y  := (x \dot{-} y) + (y \dot{-} x)$
$x < y$	$\chi_<(x, y) = \text{sg}(y \dot{-} x)$
$x \leq y$	$x < y \vee x = y$
$x/y$ “ $x$ teilt $y$ ”	$(\exists z \leq y)(y = x \cdot z)$
$\text{Prim}(x)$ “ $x$ ist Primzahl”	$x \neq 0 \wedge (\forall y \leq x)(\neg(y/x) \vee y = 1 \vee y = x)$
$x \in \{n_1, \dots, n_k\}$ “ $x$ ist Element einer endlichen Menge”	$x = n_1 \vee \dots \vee x = n_k$

Abb. 3.2 - 1: Einige primitiv rekursive Prädikate

Die bisher erarbeiteten Abschlußeigenschaften primitiv rekursiver Prädikate erlauben es uns, bereits eine ganze Reihe von Prädikaten als primitiv rekursiv zu erkennen. Eine Auswahl findet sich in der Tabelle in Abbildung 3.2 - 1.

Wir können beschränkte Quantifikationen als Verallgemeinerungen boolescher Operationen betrachten. Schließlich sind die Bedeutungen von

Erweiterte boolesche Operationen

$$(\forall y \leq n)R(\vec{x}, y) \text{ und } (\exists y \leq n)R(\vec{x}, y)$$

die von

$$R(\vec{x}, 0) \wedge R(\vec{x}, 1) \wedge \dots \wedge R(\vec{x}, n)$$

beziehungsweise

$$R(\vec{x}, 0) \vee R(\vec{x}, 1) \vee \dots \vee R(\vec{x}, n).$$

Erhalten wir ein Prädikat  $Q$  aus Prädikaten  $P_1, \dots, P_n$  durch sukzessive Anwendung von booleschen Operationen und beschränkten Quantifikationen, so sagen wir, daß  $Q$  durch *erweiterte boolesche Operationen* aus  $P_1, \dots, P_n$  hervorgegangen ist. Wir werden dies manchmal als

$$Q = \mathcal{O}(P_1, \dots, P_n)$$

notieren. Aus unseren bisherigen Überlegungen folgt, daß es dann eine primitiv rekursive Funktion  $g$  so gibt, daß

### 3. $\mu$ -rekursive Funktionen

$$\chi_Q(\vec{x}) = g(\chi_{P_1}(\vec{x}), \dots, \chi_{P_n}(\vec{x})) \quad (3.8)$$

ist.

Wir wollen diesen Abschnitt schließen, indem wir auf die Möglichkeit hinweisen, primitiv rekursive Funktionen durch Fallunterscheidung nach primitiv rekursiven Prädikaten zu definieren. Gehen wir also davon aus, daß  $R_1, \dots, R_n$  paarweise disjunkte primitiv rekursive Prädikate sind, d.h. für  $i, j \in \{1, \dots, n\}$  mit  $i \neq j$  gilt immer  $R_i \cap R_j = \emptyset$ . Sind nun  $g_1, \dots, g_{n+1}$  primitiv rekursive Funktionen, so ist die Funktion  $f$ , die definiert ist durch die Fallunterscheidung

$$f(\vec{x}) := \begin{cases} g_1(\vec{x}), & \text{falls } R_1(\vec{x}), \\ \vdots & \\ g_n(\vec{x}), & \text{falls } R_n(\vec{x}), \\ g_{n+1}(\vec{x}), & \text{sonst,} \end{cases}$$

primitiv rekursiv. Dies folgt einfach daraus, daß

$$f(\vec{x}) = g_1(\vec{x}) \cdot \chi_{R_1}(\vec{x}) + \dots + g_n(\vec{x}) \cdot \chi_{R_n}(\vec{x}) + g_{n+1}(\vec{x}) \cdot \overline{\text{sg}}\left(\sum_{i=0}^n \chi_{R_i}(\vec{x})\right)$$

ist.

## 3.3 Primitiv rekursive Kodierung

*Kodierung* ist das Herzstück der Informatik. Unter einer Kodierung verstehen wir im weitesten Sinne die Verschlüsselung von Information zusammen mit der Möglichkeit, diese verschlüsselte Information wieder zu entschlüsseln. Wenn wir davon ausgehen, daß uns die Information in Form endlicher Folgen natürlicher Zahlen vorliegt, dann bedeutet Kodierung die Fähigkeit, endlichen Folgen natürlicher Zahlen eine natürliche Zahl so zuzuordnen, daß die ursprüngliche Folge aus dieser Zahl wieder rekonstruiert werden kann. Das heißt, daß wir eine *Kodierungsfunktion*

$$Kod: \bigcup_{n \in \mathbb{N}} \mathbb{N}^n \longrightarrow \mathbb{N}$$

zusammen mit einer *Dekodierungsfunktion*

$$Dkod: \text{rng}(Kod) \longrightarrow \bigcup_{n \in \mathbb{N}} \mathbb{N}^n$$

derart definieren können, daß

$$Dkod(Kod(x_1, \dots, x_n)) = (x_1, \dots, x_n)$$

gilt, wobei wir mit  $\text{rng}(f) := \{f(x) \mid x \in \text{dom}(f)\}$  das *Bild (range)* einer Funktion  $f$  bezeichnen. Wir erhalten eine *Längenfunktion*

$$\begin{aligned} lh : \text{rng}(Kod) &\longrightarrow \mathbb{N} \\ lh(x) &:= \min\{n \mid Dkod(x) \in \mathbb{N}^n\}, \end{aligned}$$

die aus dem Kode die Länge der kodierten Folge berechnet.

In diesem Abschnitt wollen wir nachweisen, daß es gelingt, Kodierungsfunktionen in primitiv rekursiver Weise zu definieren. Dabei betonen wir allerdings *nachdrücklich*, daß es hier *nicht* auf die Effizienz der Kodierung ankommt – obwohl dies im praktischen Bereich von erheblicher Wichtigkeit sein kann –, sondern es nur um den Nachweis der *prinzipiellen* Möglichkeit geht, eine Kodierung primitiv rekursiv zu definieren. Dabei wollen wir eine Kodierung primitiv rekursiv nennen, wenn die Funktionen  $Kod$ ,  $lh$ , das Prädikat  $\text{rng}(Kod)$  und die Komponentenfunktionen der Dekodierungsfunktion  $Dkod$  alle primitiv rekursiv sind.

Eine primitiv  
rekursive  
Kodierungsfunktion

Die Kodierung, die wir hier angeben wollen, beruht auf der eindeutigen Primzahlzerlegung für natürliche Zahlen. Dazu definieren wir uns zunächst eine Funktion  $Pz$ , die die Primzahlen aufzählt. Die Rekursionsgleichungen für diese Funktion sind

$$Pz(0) = 2 \tag{3.9}$$

und

$$Pz(S(z)) = \min\{x \mid \text{Prim}(x) \wedge Pz(z) < x\}. \tag{3.10}$$

An den Gleichungen (3.9) und (3.10) ist allerdings noch nicht zu sehen, daß die Funktion  $Pz$  primitiv rekursiv ist. Dies liegt an der unbeschränkten Suche, die mit der Minimumsbildung in (3.10) verbunden ist. Um  $Pz$  primitiv rekursiv definieren zu können, müssen wir diese Suche beschränken. Das gelingt, wenn man bedenkt, daß im Intervall

$$(Pz(n), Pz(n)! + 1) := \{x \mid Pz(n) < x \leq Pz(n)! + 1\}$$

mindestens eine Primzahl liegen muß. Also können wir (3.10) ersetzen durch

$$Pz(S(z)) = \mu x \leq (Pz(z)! + 1). (\text{Prim}(x) \wedge Pz(z) < x). \tag{3.11}$$

Die zunächst naheliegende Idee, eine Kodierung vermöge der Abbildung

$$(x_1, \dots, x_n) \longmapsto Pz(0)^{x_1} \cdots Pz(n-1)^{x_n}$$

zu definieren, stößt auf die Schwierigkeit, daß dann die Dekodierungen nicht eindeutig bestimmt sind. So wäre beispielweise der Kode eines Tu-

### 3. $\mu$ -rekursive Funktionen

---

pels, das nur aus Nullen besteht, immer 1. Die Länge des ursprünglichen Tupels kann aus dem Kode somit nicht berechnet werden. Eine Lösung dieses Problems bestünde darin, die Länge mitzukodieren. Wir wollen hier einen anderen Weg gehen, indem wir dafür sorgen, daß der Exponent 0 bei der Kodierung nicht auftreten kann. Wir definieren daher

$$\langle x_1, \dots, x_n \rangle := \begin{cases} 0, & \text{falls } n = 0, \\ Pz(0)^{x_1+1} \dots Pz(n-1)^{x_n+1}, & \text{sonst.} \end{cases}$$

Hier folgen wir der Schreibweise der Standardliteratur der Rekursionstheorie, in der die primitiv rekursive Kodierungsfunktion mit  $\langle \dots \rangle$  bezeichnet wird. Zur Definition der Dekodierung haben wir den Exponenten der Primzahl  $Pz(i)$  in der Primzahlzerlegung einer natürlichen Zahl  $x$  zu bestimmen. Dieser ist gegeben durch

$$\nu(i, x) := \min\{z \mid \neg(Pz(i)^{z+1}/x)\}.$$

Da der Exponent kleiner oder gleich  $x$  zu sein hat, können wir  $\nu$  vermöge

$$\nu(i, x) = \mu z \leq x . \neg(Pz(i)^{z+1}/x)$$

als primitiv rekursive Funktion erhalten. Wir definieren dann

$$(x)_i := \nu(i, x) \dot{-} 1$$

und

$$lh(x) := \mu z \leq x . \neg(Pz(z)/x)$$

und erhalten

$$\langle (x)_0, \dots, (x)_{lh(x) \dot{-} 1} \rangle = x. \quad (3.12)$$

Das Bild der Kodierungsfunktion ist dann die Menge aller “Folgennummern” (englisch “sequence numbers”) und wird daher üblicherweise mit  $Seq$  bezeichnet. Dann gilt

$$Seq := \text{rng}(\langle \dots \rangle) = \{x \mid (\forall z \leq x)(\neg(Pz(z+1)/x) \vee Pz(z)/x)\}, \quad (3.13)$$

womit  $Seq$  als primitiv rekursive Menge zu erkennen ist. Die Dekodierung erhalten wir durch

$$Dkod(x) := ((x)_0, \dots, (x)_{lh(x) \dot{-} 1}).$$

Damit haben wir eine primitiv rekursive Kodierung angegeben.

Eine wichtige Funktion auf den Kodes ist die *Konkatenation von Kodes*. Diese ist gegeben durch

$$x \widehat{\cdot} y := \langle (x)_0, \dots, (x)_{lh(x) \dot{-} 1}, (y)_0, \dots, (y)_{lh(y) \dot{-} 1} \rangle. \quad (3.14)$$

### 3.4. Simultane Rekursion und Wertverlaufsrekursion

Aus der in (3.14) gegebenen Definition erhalten wir sofort, daß

$$\langle x_1, \dots, x_n, y_1, \dots, y_m \rangle = \langle x_1, \dots, x_n \rangle \cdot \langle y_1, \dots, y_m \rangle$$

gilt und die Konkatenation eine primitiv rekursive Funktion ist.

## 3.4 Simultane Rekursion und Wertverlaufsrekursion

Als erste Anwendung der primitiv rekursiven Kodierung werden wir zeigen, daß wir mit ihrer Hilfe auch mehrere Funktionen simultan durch primitive Rekursion definieren können. Wir sagen, daß die Funktionen  $f_1, \dots, f_n$  durch *simultane Rekursion* definiert sind, wenn sie den Rekursionsgleichungen

Simultane primitive Rekursion

$$f_i(0, \vec{x}) = g_i(\vec{x}) \quad (3.15)$$

und

$$f_i(S(z), \vec{x}) = h_i(z, \vec{x}, f_1(z, \vec{x}), \dots, f_n(z, \vec{x})) \quad (3.16)$$

für  $i = 1, \dots, n$  gehorchen, wobei die Funktionen  $g_1, \dots, g_n$  und  $h_1, \dots, h_n$  gegebene Funktionen korrekter Stellenzahlen sind.

Es ist durch Induktion leicht nachzuweisen, daß die Rekursionsgleichungen die Funktionen  $f_1, \dots, f_n$  eindeutig bestimmen, und wir möchten gerne einsehen, daß bei primitiv rekursiven Funktionen  $g_1, \dots, g_n$  und  $h_1, \dots, h_n$  auch die Funktionen  $f_1, \dots, f_n$  primitiv rekursiv sind.

Dazu definieren wir eine Funktion  $\tilde{f}$  vermöge der Rekursionsgleichungen

$$\tilde{f}(0, \vec{x}) = \langle g_1(\vec{x}), \dots, g_n(\vec{x}) \rangle$$

und

$$\begin{aligned} \tilde{f}(S(z), \vec{x}) = \\ \langle h_1(z, \vec{x}, (\tilde{f}(z, \vec{x}))_0, \dots, (\tilde{f}(z, \vec{x}))_{n-1}), \dots, h_n(z, \vec{x}, (\tilde{f}(z, \vec{x}))_0, \dots, (\tilde{f}(z, \vec{x}))_{n-1}) \rangle. \end{aligned}$$

Damit haben wir  $\tilde{f}$  als primitiv rekursive Funktion und erhalten

$$f_i(z, \vec{x}) = (\tilde{f}(z, \vec{x}))_{i-1}$$

für  $i = 1, \dots, n$ .

Eine weitere Anwendung ist die sogenannte *Wertverlaufsrekursion*. Um diese zu beschreiben, definieren wir uns den *Wertverlauf*  $\bar{f}$  einer Funktion  $f$  durch die Rekursionsgleichungen

Wertverlauf einer Funktion

$$\bar{f}(0, \vec{x}) = \langle \rangle \quad (3.17)$$

### 3. $\mu$ -rekursive Funktionen

---

und

$$\bar{f}(\mathbf{S}(z), \vec{x}) = \bar{f}(z, \vec{x}) \cap \langle f(z, \vec{x}) \rangle. \quad (3.18)$$

Offenbar ist für  $z > 0$

$$\bar{f}(z, \vec{x}) = \langle f(0, \vec{x}), \dots, f(z-1, \vec{x}) \rangle,$$

woran zu sehen ist, daß  $\bar{f}(z, \vec{x})$  tatsächlich den Wertverlauf der Funktion  $f$  unterhalb des Arguments  $z$  beschreibt. Also haben wir stets

$$\bar{f}(z, \vec{x}) \in Seq$$

und

$$lh(\bar{f}(z, \vec{x})) = z.$$

Aus den Rekursionsgleichungen (3.17) und (3.18) folgt sofort, daß für eine primitiv rekursive Funktion  $f$  auch deren Wertverlauf  $\bar{f}$  primitiv rekursiv ist. Umgekehrt erhalten wir

$$f(z) = (\bar{f}(z+1))_z,$$

was zeigt, daß sich die ursprüngliche Funktion primitiv rekursiv aus ihrem Wertverlauf berechnen läßt.

Wir sagen nun, daß eine Funktion  $f$  durch *Wertverlaufsrekursion* definiert ist, wenn sie der Rekursionsgleichung

$$f(z, \vec{x}) = g(z, \bar{f}(z, \vec{x}), \vec{x}) \quad (3.19)$$

für eine gegebene Funktion  $g$  gehorcht.

Auch hier läßt sich durch eine einfache Induktion nach  $z$  nachweisen, daß  $f(z, \vec{x})$  eindeutig bestimmt ist. Aus den Gleichungen (3.17), (3.18) und (3.19) folgt nun, daß bei primitiv rekursivem  $g$  die durch Wertverlaufsrekursion definierte Funktion  $f$  ebenfalls primitiv rekursiv ist.

Eine wichtige und häufige Anwendung der Wertverlaufsrekursion ist der folgende Spezialfall. Wir gehen davon aus, daß wir eine primitiv rekursive Funktion  $g$  zusammen mit primitiv rekursiven Funktionen  $h_1, \dots, h_n$  haben, für die

$$h_i(z, \vec{x}) < z$$

für alle natürlichen Zahlen  $z$  gilt. Dann ist eine Funktion  $f$  durch die Gleichung

$$f(z, \vec{x}) = g(f(h_1(z, \vec{x}), \vec{x}), \dots, f(h_n(z, \vec{x}), \vec{x})) \quad (3.20)$$

Wertverlaufs-  
rekursion

eindeutig bestimmt und primitiv rekursiv. Dies ist mit Hilfe der Wertverlaufsrekursion unmittelbar einzusehen, da wir wegen  $h_i(z, \vec{x}) < z$  (3.20) schreiben können als

$$f(z, \vec{x}) = g((\bar{f}(z, \vec{x}))_{h_1(z, \vec{x})}, \dots, (\bar{f}(z, \vec{x}))_{h_n(z, \vec{x})}).$$

Dies wird oft Anwendung bei der Definition primitiv rekursiver Prädikate finden. Dort werden wir öfter Prädikate  $P$  vermöge der “Rekursionsgleichung”

$$P = \mathcal{O}(\dots, \text{Sub}(P, h_1, \dots, h_n), \dots)$$

definieren, wobei  $\mathcal{O}$  eine erweiterte boolesche Operation ist und die Funktionen  $h_i$  den obigen Einschränkungen unterliegen. Nach (3.8) gehorcht die charakteristische Funktion des so definierten Prädikats dann der Gleichung

$$\chi_P(\vec{x}, z) = g(\dots, \chi_P(h_1(\vec{x}, z), \dots, h_n(\vec{x}, z)), \dots) \quad (3.21)$$

für eine primitiv rekursive Funktion  $g$  und ist somit primitiv rekursiv.

Da  $(z)_x < z$  für alle natürlichen Zahlen  $x$  und  $z > 0$  gilt, können wir in (3.21) als Spezialfall  $h_i(x, z) := (z)_x$  wählen. Dies wird die häufigste Anwendung sein.

## 3.5 Kodes für primitiv rekursive Funktionen

Wir haben bereits in Abschnitt 3.1 bemerkt, daß der Algorithmus für eine primitiv rekursive Funktion  $f$  durch einen Funktionsterm  $F$  gegeben ist, dem  $f$  extensional gleich ist. In  $F$  ist daher die Information gespeichert, die es uns ermöglicht, den Wert  $f(\vec{x})$  zu dem Argument  $\vec{x}$  zu berechnen. Wenn unsere in 3.3 eingeführten Kodierungsfunktionen wirklich leistungsfähig sind, so müßte es gelingen, mit ihnen die in einem Funktionsterm enthaltene Information zu verschlüsseln.

Um zu zeigen, daß diese das tatsächlich leisten, wollen wir für jeden Funktionsterm  $F$  einen Kode  $\overline{F}$  definieren, der die in  $F$  vorhandene Information verschlüsselt.

**Definition 12** (*Kodes für primitiv rekursive Funktionsterme*)

1.  $\overline{C_k^n} = \langle 0, n, k \rangle$
2.  $\overline{P_k^n} = \langle 1, n, k \rangle$
3.  $\overline{S} = \langle 2, 1 \rangle$
4.  $\overline{\text{Sub}}(g, h_1, \dots, h_m) = \langle 3, (\overline{h_1}), g, (\overline{h_1}), \dots, (\overline{h_n}) \rangle$
5.  $\overline{\text{Rec}}(g, h) = \langle 4, (g)_1 + 1, g, \overline{h} \rangle$

Kodierung primitiv  
rekursiver  
Funktionsterme

Wir nennen  $\overline{F}$  die Gödelnummer des Termes  $F$ . Offenbar läßt sich

### 3. $\mu$ -rekursive Funktionen

---

der Funktionsterm  $F$  aus  $\lceil F \rceil$  völlig rekonstruieren. Das Verständnis der Kodierung in Definition 12 wird vielleicht erleichtert, wenn man sich vor Augen hält, daß ein Kode immer die Gestalt  $\langle nr, n, \dots \rangle$  hat, wobei  $nr$  für die laufende Nummer (beginnend mit 0) in der Reihenfolge der Definitionsklauseln steht,  $n$  die Stellenzahl des kodierten Termes ist und in  $\dots$  die weitere Information steht, wie sie beispielsweise durch die Kodes der Subterme gegeben wird.

Indizes für primitiv rekursive Funktionen

Ist  $f$  eine primitiv rekursive Funktion und  $F$  ein primitiv rekursiver Funktionsterm, dem  $f$  extensional gleich ist, so heißt  $\lceil F \rceil$  ein *Index* von  $f$ . Wir wollen Indizes von  $f$  auch durch  $\lceil f \rceil$  notieren, obwohl dies nicht ganz korrekt ist, da eine Funktion  $f$  verschiedene – sogar unendlich viele verschiedene – Indizes hat. Gerechtfertigt wird diese nachlässige Schreibweise dadurch, daß im extensionalen Sinne eine Funktion eindeutig durch einen ihrer Indizes bestimmt ist. Wir notieren diese eindeutig bestimmte Funktion durch  $[e]$ .

Definieren wir die Menge

$$Ind(Pri) := \{e \mid e \text{ ist Index einer primitiv rekursiven Funktion}\}, \quad (3.22)$$

so stellt sich sofort die Frage, ob

$$e \in Ind(Pri)$$

entscheidbar ist. Dazu haben wir zu untersuchen, wie  $e \in Ind(Pri)$  durch Definition 12 beschrieben ist. Es gilt

$$\begin{aligned} e \in Ind(Pri) \iff & Seq(e) \\ & \wedge \{[(e)_0 = 0 \wedge lh(e) = 3] \\ & \quad \vee [(e)_0 = 1 \wedge lh(e) = 3 \wedge 1 \leq (e)_2 \leq (e)_1] \\ & \quad \vee [(e)_0 = 2 \wedge lh(e) = 2 \wedge (e)_1 = 1] \\ & \quad \vee [(e)_0 = 3 \wedge lh(e) = (e)_{21} + 3 \\ & \quad \quad \wedge (\forall x < lh(e))(x < 2 \vee (e)_x \in Ind(Pri)) \\ & \quad \quad \wedge (\forall x < lh(e))(x < 3 \vee (e)_1 = (e)_{x1})] \\ & \quad \vee [(e)_0 = 4 \wedge lh(e) = 4 \wedge (e)_3 \in Ind(Pri) \\ & \quad \quad \wedge (e)_4 \in Ind(Pri) \wedge (e)_1 = (e)_{21} + 1 \\ & \quad \quad \wedge (e)_{31} = (e)_1 + 1]\}, \end{aligned}$$

wobei wir die Abkürzung

$$(e)_{ij} := ((e)_i)_j$$

benutzt haben. Das Prädikat  $Ind(Pri)$  ist durch Wertverlaufsrekursion definiert – dies ist eine der in Abschnitt 3.4 erwähnten Anwendungen – und damit primitiv rekursiv.

Wir können nun eine  $n + 1$ -stellige Funktion  $\phi_n$  definieren durch

$$\phi_n(e, \vec{x}) := \begin{cases} [e](\vec{x}), & \text{falls } e \in \text{Ind}(Pri) \text{ und } (e)_1 = n, \\ 0, & \text{sonst.} \end{cases}$$

Die Funktion  $\phi_n$  ist offensichtlich berechenbar. Ein Algorithmus ergibt sich wie folgt:

*Wir entscheiden zunächst, ob  $e \in \text{Ind}(Pri)$  gilt und berechnen, falls dies zutrifft,  $(e)_1$ . Das geschieht in primitiv rekursiver Weise. Ist  $e \notin \text{Ind}(Pri)$  oder  $(e)_1 \neq n$ , so ist  $\phi_n(e, \vec{x}) = 0$  und wir sind fertig. Andernfalls dekodieren wir  $e$  und erhalten so den Funktionsterm  $[e]$ . Dann können wir  $[e](\vec{x})$  primitiv rekursiv berechnen.*

Die Funktion  $\phi_n$  ist universell für die  $n$ -stelligen primitiv rekursiven Funktionen. Das bedeutet, daß wir mit  $\phi_n$  jede  $n$ -stellige primitiv rekursive Funktion berechnen können. Zu gegebenem  $f$  und Argumenten  $\vec{x}$  benötigen wir nur dessen Index  $e$  um  $f(\vec{x}) = \phi_n(e, \vec{x})$  zu erhalten. Wir werden auf die Definition und Bedeutung universeller Funktionen in den Abschnitten 3.8 und 3.10 noch genauer eingehen.

Die Familie  $\{\phi_n(e, \cdot) \mid e \in \text{Ind}(Pri)\}$  ist demnach eine Familie primitiv rekursiver Funktionen. Andererseits kann  $\phi_n(e, \vec{x})$ , als Funktion der Argumente  $e$  und  $\vec{x}$  aufgefaßt, nicht primitiv rekursiv sein. Nehmen wir nämlich an, sie wäre es, so wäre auch die  $n$ -stellige Funktion  $f$ , die durch

$$f(x_1, x_2, \dots, x_n) := \phi_n(x_1, x_1, x_2, \dots, x_n) + 1$$

definiert ist, primitiv rekursiv. Also gäbe es einen Index  $e_0$  für  $f$  und wir erhielten

$$\begin{aligned} \phi_n(e_0, e_0, x_2, \dots, x_n) + 1 &= f(e_0, x_2, \dots, x_n) = [e_0](e_0, x_2, \dots, x_n) = \\ &= \phi_n(e_0, e_0, x_2, \dots, x_n), \end{aligned}$$

d.h.  $1 = 0$ , was natürlich absurd ist.

An diesem Beispiel sehen wir, daß wir mit den primitiv rekursiven Funktionen den Bereich der intuitiv berechenbaren Funktionen offensichtlich noch nicht ausgeschöpft haben. Im nächsten Abschnitt werden wir daher eine Erweiterung der Klasse der primitiv rekursiven Funktionen einführen.

## 3.6 $\mu$ -partiellrekursive Funktionen

Um zu einer Erweiterung der Klasse der primitiv rekursiven Funktionen zu gelangen, erinnern wir uns des Suchoperators, den wir in seiner beschränkten Form bereits in 3.2 eingeführt hatten. Dort hatten wir den Suchoperator auf Prädikate angewandt und gelangten so zu Funktionen.

Eine nicht primitiv  
rekursive,  
berechenbare  
Funktion

### 3. $\mu$ -rekursive Funktionen

---

Natürlich kann man einen beschränkten Suchoperator zur Definition von Funktionen aus bereits gegebenen Funktionen heranziehen. Dies erreicht man, indem man

$$\mu_b f := \mu_b Ze(f)$$

setzt, wobei  $Ze(f) := \{(\vec{x}, z) \mid f(\vec{x}, z) = 0\}$  das Nullstellengebilde der Funktion  $f$  bedeutet. Ließe man nun eine unbeschränkte Suche zu, so erhielte man

$$(\mu f)(\vec{x}) = \min\{z \mid f(\vec{x}, z) = 0\}.$$

Der unbeschränkte Suchoperator

Da  $f$  nicht notwendigerweise Nullstellen besitzen muß, sehen wir, daß  $\mu f$  unter Umständen undefiniert sein kann. Wir müssen daher, wie schon früher bei den maschinenberechenbaren Funktionen, die Klasse der totalen Funktionen verlassen und zu partiellen Funktionen übergehen. Das bedeutet jedoch, daß wir den unbeschränkten Suchoperator bereits für partielle Funktionen zu definieren haben. Hierin liegt tatsächlich ein kleines Problem. Wir wollen wir verfahren, wenn  $f$  eine partielle Funktion ist, derart, daß  $f(\vec{x}, 0)$  undefiniert ist, aber bei  $(\vec{x}, 1)$  eine Nullstelle hat? Vom algorithmischen Standpunkt aus gesehen, können wir diese Nullstelle nicht mit Sicherheit finden. Der natürliche Algorithmus für  $\mu f$ , der darin besteht, der Reihe nach  $f(\vec{x}, 0), f(\vec{x}, 1), \dots$  zu berechnen und zu testen, ob das erhaltene Ergebnis eine Nullstelle ist, würde sich bereits bei der Berechnung von  $f(\vec{x}, 0)$  „aufhängen“. Für jeden vorstellbaren weiteren Algorithmus ergeben sich ähnliche Probleme. Daher definiert man den unbeschränkten Suchoperator in der folgenden Weise:

**Definition 13** Sei  $f: \mathbb{N}^{n+1} \xrightarrow{p} \mathbb{N}$  eine partielle Funktion. Dann ist die durch den Suchoperator gewonnene Funktion  $(\mu f)$  gegeben durch

$$(\mu f)(\vec{x}) \simeq \min\{z \mid f(\vec{x}, z) \simeq 0 \wedge (\forall u < z)(\exists y)[f(\vec{x}, u) \simeq y \wedge y \neq 0]\}.$$

Das bedeutet, daß  $(\mu f)(\vec{x})$  nur dann definiert und gleich  $z$  ist, wenn  $f(\vec{x}, z) \simeq 0$  ist und  $f(\vec{x}, u)$  für alle  $u < z$  definiert ist und dort einen von 0 verschiedenen Wert besitzt.

Natürlich läßt sich der unbeschränkte Suchoperator auch für Prädikate definieren.

Sei  $P$  ein  $n + 1$ -stelliges Prädikat. Die durch den Suchoperator gebildete Funktion  $\mu P$  ist definiert durch

$$(\mu P)(\vec{x}) \simeq \min\{z \mid (\vec{x}, z) \in P\}. \quad (3.23)$$

Der Suchoperator  $(\mu P)(\vec{x})$  sucht also nach dem kleinsten  $z$  mit  $P(\vec{x}, z)$ . Damit ordnet er einem  $n + 1$ -stelligen Prädikat eine  $n$ -stellige Funktion

### 3.7. Maschinenberechenbarkeit und $\mu$ -partiellrekursive Funktionen

zu.

Auch hier wollen wir wieder die einprägsameren Schreibweisen  $\mu z. [f(\vec{x}, z) \simeq 0]$  beziehungsweise  $\mu z. P(\vec{x}, z)$  für den Suchoperator vorziehen.

Unter Zuhilfenahme des Suchoperators können wir nun die  $\mu$ -partiellrekursiven Funktionen definieren.

**Definition 14** Die  $\mu$ -partiellrekursiven Funktionen sind die kleinste Klasse partieller Funktionen, die die Grundfunktionen umfaßt und gegenüber Substitutionen, primitiver Rekursion und dem Suchoperator abgeschlossen ist. Wir bezeichnen die Klasse der  $\mu$ -partiellrekursiven Funktionen mit  $P$ .

Eine Funktion heißt rekursiv, wenn sie  $\mu$ -partiellrekursiv und total ist.  $F$  bezeichne die Klasse der rekursiven Funktionen

Auch hier gilt die Bemerkung, wie wir sie im Anschluß an die Definition der primitiv rekursiven Funktionen (Definition 10) gemacht haben. Jede partiell rekursive Funktion stimmt extensional mit einer Funktion überein, die durch einen Funktionsterm gegeben ist, der aus den Grundfunktionen mittels der Operationen Sub, Rec und  $\mu$  aufgebaut ist.

Definition 14 erweitert die Definition der primitiv rekursiven Funktionen. Jeder primitiv rekursive Funktionsterm ist offenbar auch ein gemäß Definition 14 korrekt gebildeter Funktionsterm. Damit ist jede primitiv rekursive Funktion  $\mu$ -partiellrekursiv und, da alle primitiv rekursiven Funktionen total sind, somit auch rekursiv. Also gilt

$$PR \subseteq F \subseteq P.$$

Als nächstes gilt es nun zu klären, an welcher Stelle die maschinenberechenbaren Funktionen hier ins Bild kommen. Dies wollen wir im folgenden Abschnitt tun.

## 3.7 Maschinenberechenbarkeit und $\mu$ -partiellrekursive Funktionen

Es ist relativ einfach einzusehen, daß alle  $\mu$ -partiellrekursiven Funktionen maschinenberechenbar sind. Unsere Strategie zum Nachweis wird einfach darin bestehen, Registermaschinenprogramme für die Grundfunktionen zu schreiben und uns anschließend zu überlegen, daß die auf Registermaschinen berechenbaren Funktionen gegenüber den Grundoperatoren und dem unbeschränkten Suchoperator abgeschlossen sind.

Als Vorbereitung überlegt man sich zwei einfache Hilfsprogramme wie

$\mu$ -partiellrekursive Funktionen

Rekursive Funktionen

Maschinenberechenbarkeit  
 $\mu$ -partiellrekursiver Funktionen

### 3. $\mu$ -rekursive Funktionen

---

$MOVE(k, l)$ ,

das den Inhalt des  $k$ -ten Registers in das  $l$ -te Register verschiebt und dabei das  $k$ -te Register leert, und

$COPY(k, l, m)$ ,

das den Inhalt des  $k$ -ten Registers in das  $l$ -te Register kopiert und dabei das  $m$ -te Register als Zwischenspeicher benützt, das nach Beendigung des Programmes dann leer ist. Die Flußdiagramme dieser Programme finden sich in den Abbildungen 3.7 - 1 und 3.7 - 2.

---

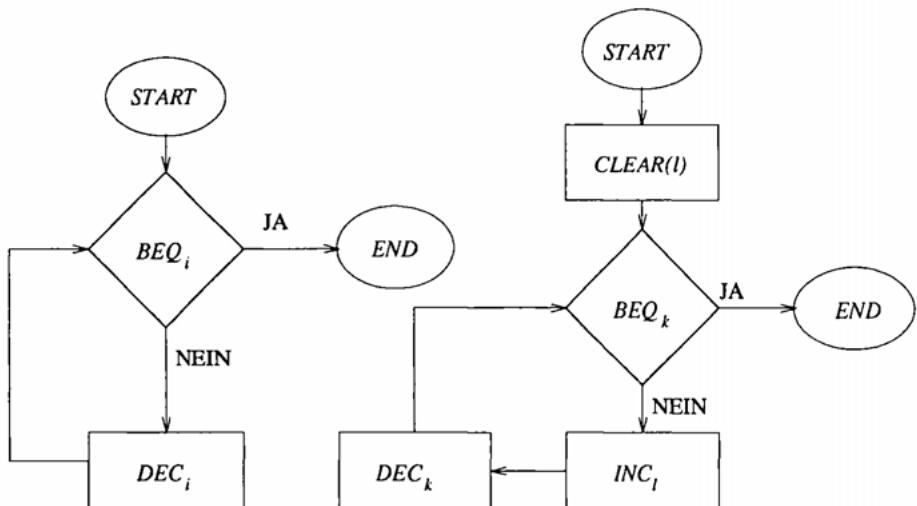


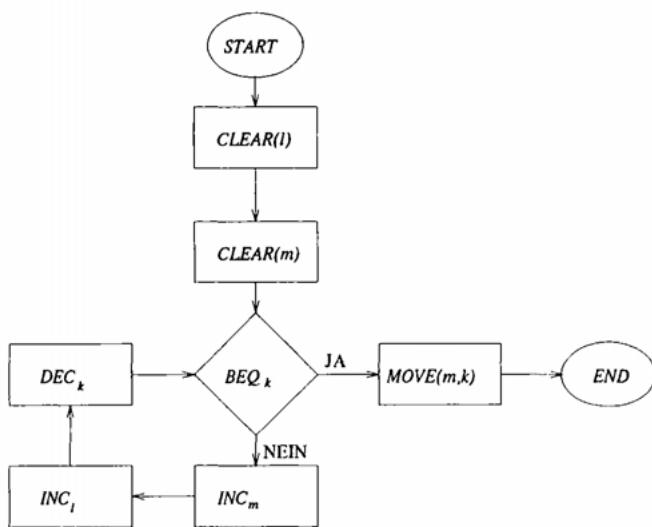
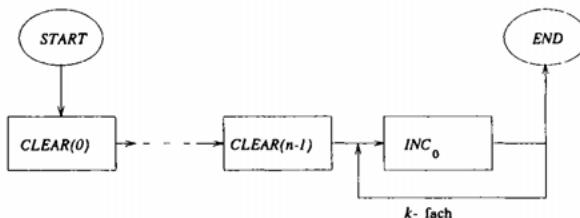
Abb. 3.7 - 1: Die Prozeduren  $CLEAR(i)$  und  $MOVE(k, l)$  zum Löschen bzw. Verschieben von Registern

---

Für die Grundfunktionen erhält man einfache Programme. Das für  $C'_k^n$  ist im Flußdiagramm 3.7 - 3 angegeben, das Programm für die Nachfolgerfunktion ist trivial und ein Programm für  $P'_k^n$  ist einfach durch  $COPY(k, 1, n + 1)$  gegeben.

Wir haben uns also nur noch den Abschluß unter den Grundoperationen und dem  $\mu$ -Operator klar zu machen. Beginnen wir mit der Substitution. Wir gehen davon aus, daß wir bereits Programme  $P_{h_j}$  für die Funktionen  $h_1, \dots, h_m$  und  $P_g$  für  $g$  haben und suchen ein Programm für  $\text{Sub}(g, h_1, \dots, h_m)$ . Wenn wir ein Registermaschinenprogramm  $P$  haben mit

$$\text{Res}_P(z_1, \dots, z_n) \simeq m,$$


 Abb. 3.7 - 2: Die Prozedur  $COPY(k, l, m)$  zum Kopieren von Registern

 Abb. 3.7 - 3: Flußdiagramm zur Berechnung der Funktion  $C_k^n$ 

so enthält dies nur endlich viele Anweisungen. Daher gibt es eine Zahl  $r$ , so daß alle von  $P$  angesprochenen Register einen Index kleiner als  $r$  haben. Es liegt somit ein  $r$ -Registermaschinenprogramm vor. Wir dürfen  $r$  so groß annehmen, daß keines der Programme  $P_{h_j}$  für  $j = 1, \dots, m$  und  $P_g$  ein Register mit einem größeren Index als  $r$  anspricht. Das in Abbildung 3.7 - 4 gezeigte Flußdiagramm liefert dann ein Programm für  $\text{Sub}(h, g_1, \dots, g_m)$ . In ähnlicher Weise erhalten wir auch Programme für  $\text{Rec}(g, h)$  und  $\mu f$ . Die Flußdiagramme hierfür finden sich in den Abbildungen 3.7 - 5 und 3.7 - 6.

Um umgekehrt einzusehen, daß die maschinenberechenbaren Funktionen  $\mu$ -partiellrekursiv sind, führen wir den Begriff einer primitiv rekursiven Basismaschine ein. Bislang haben wir nur die primitive Rekur-

### 3. $\mu$ -rekursive Funktionen

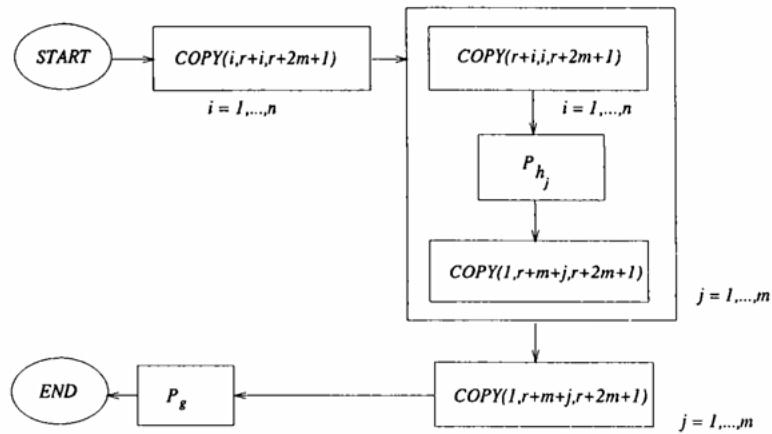


Abb. 3.7 - 4: Flußdiagramm zur Berechnung von  $\text{Sub}(g, h_1, \dots, h_m)$

Vektorwertige  
primitiv rekursive  
Funktionen

sivität von Funktionen  $f: \mathbb{N}^k \rightarrow \mathbb{N}$  definiert. Dies konnten wir ohne wesentliche Einschränkung der Allgemeinheit tun, da wir vermöge der Kodierungsfunktionen diese Definition leicht auf vektorwertige Funktionen ausdehnen können. Zu einer vektorwertigen Funktion

$$f: \mathbb{N}^k \rightarrow \mathbb{N}^l$$

mit

$$f(\vec{x}) = (f_1(\vec{x}), \dots, f_l(\vec{x}))$$

definieren wir deren *Kontraktion*

$$\langle f \rangle: \mathbb{N}^k \rightarrow \mathbb{N}$$

als

$$\langle f \rangle := \langle f_1(\vec{x}), \dots, f_l(\vec{x}) \rangle.$$

Wir nennen  $f$  (*primitiv*) *rekursiv*, wenn ihre Kontraktion  $\langle f \rangle$  (*primitiv*) *rekursiv* ist.

Wir wollen eine Basismaschine  $\mathcal{B} = (\mathcal{M}, \mathcal{S}, \mathcal{T}, \text{IN}, \text{OUT})$  *primitiv rekursiv* nennen, wenn es

1. ein  $k \in \mathbb{N}$  gibt, so daß  $\mathcal{M} = \mathbb{N}^k$  ist,
2. die Funktionen **IN**, **OUT** und alle Funktionen in  $\mathcal{S}$  primitiv rekursiv sind,
3. alle Relationen in  $\mathcal{T}$  primitiv rekursiv sind.

Fassen wir für eine primitiv rekursive Maschine  $\mathcal{C}$  den Konfigura-

Primitiv rekursive  
Basismaschinen

### 3.7. Maschinenberechenbarkeit und $\mu$ -partiellrekursive Funktionen

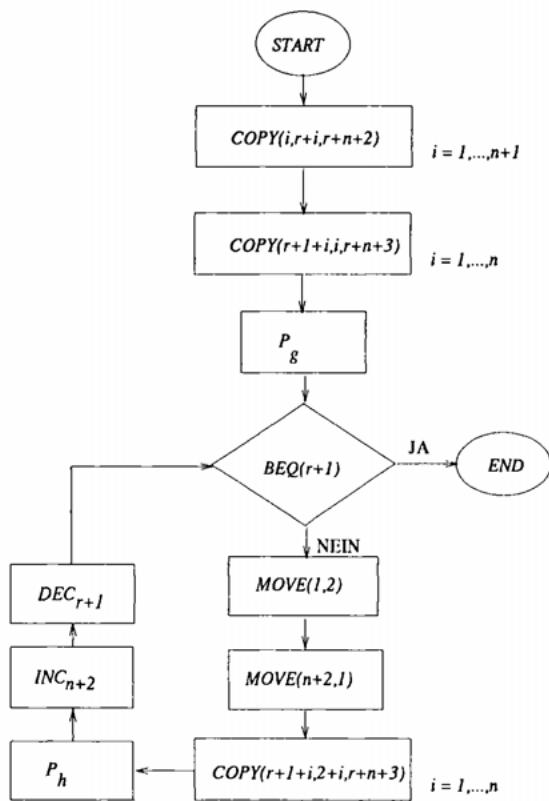


Abb. 3.7 - 5: Flußdiagramm zur Berechnung von  $\text{Rec}(g, h)$

tionsraum  $K(\mathcal{C}) = M(\mathcal{C}) \times \mathbb{N}^k$  als  $\mathbb{N}^{k+1}$  und damit die Rechenschritt-funktion als Funktion von  $\mathbb{N}^{k+1}$  nach  $\mathbb{N}^{k+1}$  auf, so ist sofort zu sehen, daß diese eine primitiv rekursive Funktion ist. Damit ergibt sich für die Rechenzeitfunktion

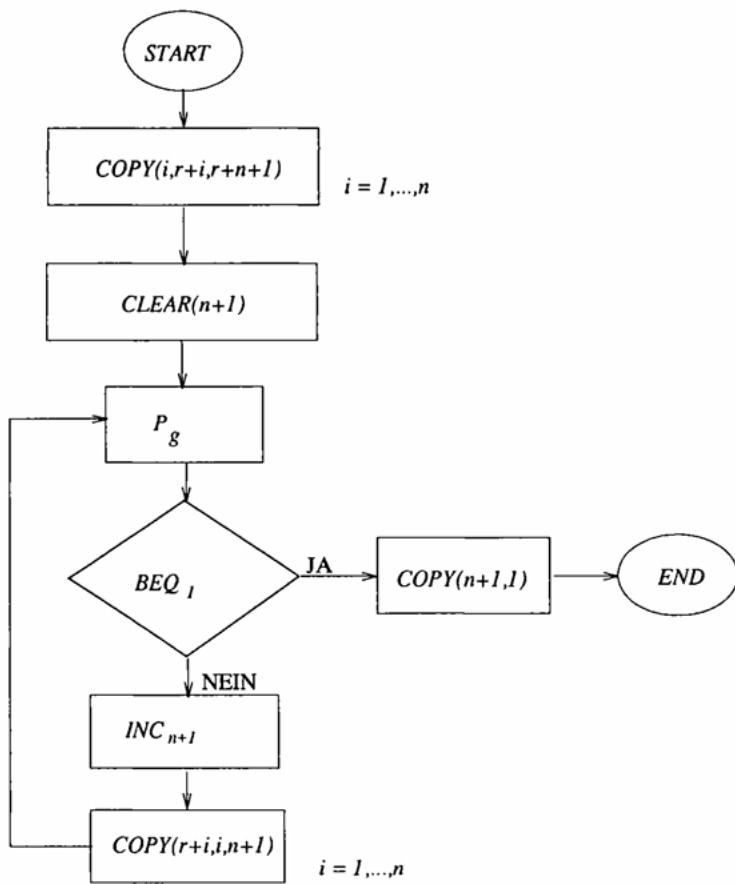
$$RZ_{\mathcal{C}}(z) \simeq \mu x. RS_{\mathcal{C}}^x \in E(\mathcal{C}),$$

womit  $RZ_{\mathcal{C}}$  als  $\mu$ -partiellrekursiv erkannt ist. Dann sind aber die innere Resultatsfunktion

$$RES_{\mathcal{C}}(z) \simeq (\mathsf{P}_2^{k+1}(RS_{\mathcal{C}} RZ_{\mathcal{C}^{(z)}}(k_P, z)), \dots, \mathsf{P}_{k+1}^{k+1}(RS_{\mathcal{C}} RZ_{\mathcal{C}^{(z)}}(k_P, z)))$$

und die äußere Resultatsfunktion

$$Res_{\mathcal{C}}(d) \simeq \text{OUT}(RES_{\mathcal{C}}(\text{IN}(d)))$$


 Abb. 3.7 - 6: Flußdiagramm zur Berechnung von  $\mu g$ 

ebenfalls  $\mu$ -partiellrekursiv.

Um einzusehen, daß die turingberechenbaren Funktionen  $\mu$ -partiellrekursiv sind, genügt es demnach, eine primitiv rekursive Basismaschine  $\mathcal{B}$  und einen Homomorphismus  $\Phi: Tur \longrightarrow \mathcal{B}$  anzugeben. Die allgemeine Bandinschrift einer Turingmaschine über  $Tur$  hat die Gestalt

$$i = B^\infty \star^{l_{m-1}} B \dots B \star^{l_0} z \star^{r_0} B \dots B \star^{r_{n-1}} B^\infty$$

mit  $z \in \{\star, B\}$ . Die Idee ist nun, diese Inschrift als das Tripel

$$\Phi(i) := (\langle l_0, \dots, l_{m-1} \rangle, \bar{z}, \langle r_0, \dots, r_{n-1} \rangle)$$

zu kodieren, wobei

### 3.7. Maschinenberechenbarkeit und $\mu$ -partiellrekursive Funktionen

---

$$\bar{z} := \begin{cases} 1, & \text{falls } z = \star, \\ 0, & \text{falls } z = B \end{cases}$$

sei. Es ist nun einfach, die Befehle der Turingbasismaschine auf diesen Tripeln zu simulieren. So ist, um mit einfachen Beispielen zu beginnen,

$$\Phi(PRT)(l, z, r) := (l, 1, r),$$

$$\Phi(CLR)(l, z, r) := (l, 0, r),$$

$$\Phi(BEQ)(l, z, r) := \begin{cases} 0, & \text{falls } z = 1 \\ 1, & \text{falls } z = 0 \end{cases}.$$

Etwas aufwendiger gestalten sich die Übersetzungen der Arbeitsfeldverschiebungen. Hier definieren wir

$$\Phi(LFT)(l, z, r) := (l', z', r')$$

mit

$$l' := \begin{cases} \langle 0 \rangle, & \text{falls } l = \langle 0 \rangle, \\ \langle (l)_1, \dots, (l)_{lh(l)-1} \rangle, & \text{falls } lh(l) > 1 \text{ und } (l)_0 = 0, \\ \langle (l)_0 + 1, \dots, (l)_{lh(l)-1} \rangle, & \text{falls } (l)_0 > 0, \\ 0, & \text{sonst} \end{cases}$$

und

$$s' := \begin{cases} 0, & \text{falls } (l)_0 = 0, \\ 1, & \text{falls } (l)_0 > 0, \end{cases}$$

und

$$r' := \begin{cases} \langle z \rangle, & \text{falls } r = \langle 0 \rangle, \\ \langle z \rangle^{\sim}, & \text{falls } lh(r) > 0 \\ \langle (r)_0 + 1, \dots, (r)_{lh(r)-1} \rangle, & \text{und } ((r)_0 = 0 \text{ oder } z = 0), \\ 0, & \text{falls } z = 1 \text{ und } (r)_0 \neq 0, \\ & \text{sonst.} \end{cases}$$

Analog erhält man  $\Phi(RHT)$ .

Definieren wir nun

$$\mathcal{B} := (\mathbb{N}^3, \{\Phi(PRT), \Phi(CLR), \Phi(LFT), \Phi(RHT)\}, \{\Phi(BEQ)\}, \text{IN}_k^{\mathcal{B}}, \text{OUT}_j^{\mathcal{B}}),$$

wobei wir

$$\text{IN}_k^{\mathcal{B}} := \Phi \circ \text{IN}_k^{Tur}$$

und

### 3. $\mu$ -rekursive Funktionen

---

$$\text{OUT}_j^B(l, z, r) := \begin{cases} ((l)_0, \dots, (l)_{j-1}), & \text{falls } j < lh(l), \\ ((l)_0, \dots, (l)_{lh(l)-1}, 0 \dots, 0), & \text{falls } lh(l) \leq j, \end{cases}$$

setzen, so ist  $B$  eine primitiv rekursive Basismaschine und

$$\Phi: \text{Tur} \longrightarrow \mathcal{B}$$

Turingberechenbare  
Funktionen sind  
 $\mu$ -partiellrekursiv

ein Maschinenhomomorphismus. Also gilt

$$\mathbf{P}_{\text{Tur}} \subseteq \mathbf{P}_B \subseteq \mathbf{P}$$

und

$$\mathbf{F}_{\text{Tur}} \subseteq \mathbf{F}_B \subseteq \mathbf{F}$$

und wir haben den Kreis geschlossen. Zusammenfassend haben wir damit den folgenden Satz bewiesen:

**Satz 15** *Die auf Register- oder Turingmaschinen berechenbaren Funktionen sind genau die  $\mu$ -rekursiven Funktionen. Die dort berechenbaren partiellen Funktionen sind genau die  $\mu$ -partiellrekursiven Funktionen.*

Wir haben mit Satz 15 eingesehen, daß alle bisherigen Versuche zur mathematischen Präzisierung des Begriffes einer berechenbaren Funktion zur gleichen Klasse von Funktionen geführt haben. Später werden wir noch einen weiteren Ansatz kennenlernen, der ebenfalls wieder zur Klasse der rekursiven Funktionen führen wird. Offensichtlich liegt hier einer der Fälle vor, in denen man einen intuitiven Begriff mathematisch gut fassen konnte. Dies wird in der CHURCHschen These ausformuliert.

Die CHURCHsche  
These

**These (ALONZO CHURCH)** *Jede berechenbare Funktion ist rekursiv.*

Wie ihr Name sagt, ist die CHURCHsche These eine *These* und *kein* mathematisch beweisbarer *Satz*. Ein Beweis würde ja eine mathematisch exakte Fassung des Begriffes ‘berechenbar’ voraussetzen. Man ist sich dieser Tatsache heute aber bereits so sicher, daß man sogar in mathematischen Beweisen mit der CHURCHschen These argumentiert. Dies geschieht in der Art, daß man, um die Rekursivität einer Funktion nachzuweisen, oft nur noch einen Algorithmus für die Funktion angibt. Allerdings kann man überall, wo man die CHURCHsche These bislang verwendet hat, diese durch einen exakten, wenn auch aufwendigeren Beweis ersetzen.

## 3.8 Der Normalformensatz von KLEENE

Nachdem die Klassen der maschinenberechenbaren und der  $\mu$ -partiellrekursiven Funktionen übereinstimmen, wollen wir diese Klassen einheitlich mit  $\mathbf{P}$  bezeichnen und nur noch von *partiellrekursiven Funktionen* sprechen.

Im Abschnitt 3.3 hatten wir gezeigt, daß sich die primitiv rekursiven Funktionen in primitiv rekursiver Weise kodieren lassen. Diese Kodierung läßt sich in natürlicher Weise zu einer Kodierung für die partiellrekursiven fortsetzen, indem wir die Definition 12 um eine Klausel für den unbeschränkten Suchoperator erweitern. Damit erhalten wir:

**Definition 16** (*Kodes für partiellrekursive Funktionsterme*)

1.  $\lceil C_k^n \rceil = \langle 0, n, k \rangle$
2.  $\lceil P_k^n \rceil = \langle 1, n, k \rangle$
3.  $\lceil S \rceil = \langle 2, 1 \rangle$
4.  $\lceil \text{Sub}(g, h_1, \dots, h_m) \rceil = \langle 3, (\lceil h_1 \rceil)_1, g, \lceil h_1 \rceil, \dots, \lceil h_n \rceil \rangle$
5.  $\lceil \text{Rec}(g, h) \rceil = \langle 4, (\lceil g \rceil)_1 + 1, g, \lceil h \rceil \rangle$
6.  $\lceil (\mu f) \rceil = \langle 5, (\lceil f \rceil)_1 \dot{-} 1, f \rangle$

Kodes für  
partiellrekursive  
Funktionsterme

Wir definieren dann die Menge

$$\text{Ind}(\mathbf{P}) := \{e \mid e \text{ ist Index einer partiellrekursiven Funktion}\}$$

und weisen wie in Abschnitt 3.5 nach, daß  $\text{Ind}(\mathbf{P})$  eine primitiv rekursive Menge ist.

Unser nächstes Ziel ist es, den ganzen Verlauf einer Rechnung zu kodieren. Denken wir an die maschinenberechenbaren Funktionen, so ist eine Kodierung des Rechenverlaufes nicht schwierig. Gehen wir wie im vorhergehenden Abschnitt vom Konfigurationsraum  $\mathbb{N}^{k+1}$  aus, so läßt sich eine Konfiguration

$$(k, z_1, \dots, z_k)$$

Entwicklung des Be-  
rechnungsprädikates

durch

$$\lceil (k, z_1, \dots, z_k) \rceil := \langle k, z_1, \dots, z_k \rangle$$

kodieren. Umgekehrt dekodieren wir eine Konfiguration durch

$$Dkod(b) := ((b)_0, \dots, (b)_{lh(b)-1}).$$

Eine erfolgreiche Berechnung einer Maschine  $\mathcal{C}$  beim Input  $d \in \mathcal{D}$  wird dann durch eine endliche Folge

$$\lceil (k_{\mathcal{C}}, \text{IN}(d)) \rceil, \lceil RSc((k_{\mathcal{C}}, \text{IN}(d)) \rceil, \lceil RSc^2(k_{\mathcal{C}}, \text{IN}(d)) \rceil, \dots, \lceil RSc^t(k_{\mathcal{C}}, \text{IN}(d)) \rceil$$

natürlicher Zahlen beschrieben, wobei  $t \simeq Rz_{\mathcal{C}}(d)$  die Rechenzeit der Maschine  $\mathcal{C}$  beim Input  $d$  bedeutet. Diese Folge läßt sich dann in eine einzige Zahl

$$b = \langle \lceil (k_{\mathcal{C}}, \text{IN}(d)) \rceil, \dots, \lceil RSc^t(k_{\mathcal{C}}, \text{IN}(d)) \rceil \rangle$$

### 3. $\mu$ -rekursive Funktionen

---

kodieren. Ziehen wir in Betracht, daß eine Maschine durch ihr Programm  $P = (k_P, A)$  mit einer *endlichen* Anweisungs menge  $A$  gegeben ist, so sehen wir wieder sofort ein, daß sich dieses Programm – und damit die Maschine – durch einen Kode  $e$  beschreiben läßt. Dieser Kode kann beispielsweise dadurch erhalten werden, daß wir jeder der Instruktionen  $i$  der Basismaschine eine Nummer  $\overline{i}$  zuordnen und so für jede Anweisung  $a = (n, i, m)$  oder  $a = (n, i, p, q)$  eine Gödelnummer  $\overline{a} := \langle n, \overline{i}, m \rangle$  bzw.  $\overline{a} := \langle n, \overline{i}, p, q \rangle$  erhalten. Ist  $A$  die endliche Menge  $\{a_1, \dots, a_k\}$ , so kodieren wir  $A$  als  $\overline{A} := \langle a_1, \dots, a_k \rangle$  und endlich das gesamte Programm durch  $e := \overline{P} := \langle k_P, \overline{A} \rangle$  für  $P$ . Wir haben nun ein Prädikat

$$T(e, d, b),$$

welches aussagt, daß die Maschine mit Kode  $e$  beim Input  $d$  die Berechnung mit Kode  $b$  liefert. Sehen wir uns dieses Prädikat etwas genauer an, so stellen wir fest, daß wir es in primitiv rekursiver Weise entscheiden können. Informal läßt sich dies folgendermaßen einsehen.

Zunächst beachten wir, daß wir die Funktion  $RS_C$ , die Startmarke  $k_C$  und die Endmarken von  $C$  aus  $e$  in primitiv rekursiver Weise erhalten. Wir dürfen auf Grund der Ergebnisse des Abschnitts 3.7 davon ausgehen, daß  $RS_C$ , IN und OUT primitiv rekursive Funktionen sind. Dann können wir die Tatsache, daß  $b$  eine Berechnung des Programms mit Kode  $e$  beim Input  $d$  darstellt, etwa durch die folgende Formel ausdrücken:

$$\begin{aligned} Seq(b) \wedge (b)_0 &= \overline{(k_P, \text{IN}(d))} \\ \wedge (\forall i < lh(b) - 2)[(b)_{i+1} &= \overline{RS_C(Dkod((b)_i))} \wedge Dkod((b)_{lh(b)-1}) \in E(C)] \end{aligned}$$

Das Berechnungsprädikat ist primitiv rekursiv

Diese Darstellung sollte genügen, um *prinzipiell* klar zu machen, daß sich das Berechnungsprädikat  $T(e, d, b)$  primitiv rekursiv definieren läßt.

Das Ergebnis der Rechnung erhalten wir nun als

$$\text{OUT}((b)_1, \dots, (b)_{lh(b)-1})$$

und somit als primitiv rekursive Funktion im Argument  $b$ . Betrachten wir hier wieder nur Funktionen, die  $n$ -Tupel natürlicher Zahlen auf natürliche Zahlen abbilden – ist also  $d = (x_1, \dots, x_n) \in \mathbb{N}^n$ , – so schreiben wir das Berechnungsprädikat als

$$T^n(e, x_1, \dots, x_n, b).$$

Allerdings hatten wir den Kode einer partiellrekursiven Funktion  $f$  hier nicht als Kodierung einer Maschine  $C$  für  $f$ , sondern als die Gödelnummer  $\overline{F}$  eines Funktionstermes  $F$  für  $f$  definiert. Wegen der Bedeutung des Berechnungsprädikats wollen wir skizzieren, daß wir es auch bei dieser Kodierung als primitiv rekursives Prädikat erhalten. Wie bereits mehrfach betont, läßt sich aus dem Funktionsterm  $F$ , und damit auch aus seiner Gödelnummer  $\overline{F}$ , ein Algorithmus für  $f$  entnehmen. Wir können uns

den Berechnungskode von  $F(\vec{x})$  als einen Kode

$$\langle e, x, z, b \rangle$$

vorstellen, in dem  $e$  ein Index der Funktion  $F$  ist,  $x$  den zu einer Zahl zusammenkodierten Input bedeutet,  $z$  für das Endergebnis der Rechnung steht und in  $b$  alle erforderlichen Nebenrechnungen kodiert sind. Für die Grundfunktionen erhalten wir dann die Berechnungskodes wie folgt:

1. Der Berechnungskode für  $C_j^n(x_1, \dots, x_n)$  ist

$$\langle [C_j^n], \langle x_1, \dots, x_n \rangle, j, \langle \rangle \rangle.$$

2. Der Berechnungskode für  $P_j^n(x_1, \dots, x_n)$  ist gegeben durch

$$\langle [P_j^n], \langle x_1, \dots, x_n \rangle, x_i, \langle \rangle \rangle.$$

3. Der Berechnungskode für  $S(x)$  ist gegeben durch

$$\langle [S], \langle x \rangle, S(x), \langle \rangle \rangle.$$

Im Falle der Operationen erhalten wir die Berechnungskodes durch Zusammensetzen der Kodes der Teilberechnungen. Auch dies wollen wir im einzelnen erhellen.

4. So ist der Berechnungskode für  $\text{Sub}(h, g_1, \dots, g_m)(x_1, \dots, x_n)$  gegeben durch

$$\langle [\text{Sub}(h, g_1, \dots, g_m)], \langle x_1, \dots, x_n \rangle, y, \langle b_h, b_{g_1}, \dots, b_{g_m} \rangle \rangle,$$

wenn wir davon ausgehen, daß  $b_{g_i}$  für  $i = 1, \dots, m$  einen Berechnungskode für  $g_i(x_1, \dots, x_n)$  bezeichnet, sowie  $b_h$  ein Berechnungskode für  $h(u_1, \dots, u_m)$  mit  $u_i := g_i(x_1, \dots, x_n)$ , und  $h(u_1, \dots, u_m) = y$  ist.

5. Ähnlich erhalten wir einen Berechnungskode für  $\text{Rec}(g, h)(z, x_1, \dots, x_n)$  zu

$$\langle [\text{Rec}(g, h)], \langle z, x_1, \dots, x_n \rangle, y, \langle b_0, \dots, b_z \rangle \rangle,$$

wobei  $b_0$  ein Berechnungskode für  $g(x_1, \dots, x_n)$  und  $b_{i+1}$  für  $i < z$  ein Berechnungskode für  $h(i, x_1, \dots, x_n, (b_i)_2)$  mit  $y = (b_z)_2$  ist.

6. Für  $(\mu f)(x_1, \dots, x_n)$  hat der Berechnungskode die Form

$$\langle [\mu f], \langle x_1, \dots, x_n \rangle, z, \langle b_0, \dots, b_z \rangle \rangle,$$

wobei  $b_i$  für  $i = 0, \dots, z$  ein Berechnungskode für  $f(x_1, \dots, x_n, i)$  ist,  $(b_i)_2 > 0$  für  $i = 0, \dots, z - 1$  gilt und  $(b_z)_2 = 0$  ist.

Wir erhalten daher durch Wertverlaufsrekursion ein primitiv rekursives Prädikat

$$B(\langle e, x, y, b \rangle),$$

welches zum Ausdruck bringt, daß  $e$  die Gödelnummer eines Funktionsterms  $F$  ist und  $b$  ein Berechnungskode für  $F((x)_0, \dots, (x)_{lh(x)-1})$  mit dem Ergebnis  $y$  ist. Wir wollen dies hier nicht in allen Details ausführen. Zur Illustration betrachten wir lediglich den oben unter 6. behandelten Fall des Suchoperators. Hier haben wir

### 3. $\mu$ -rekursive Funktionen

---

$$B_5(c) \iff$$

$$\begin{aligned} & Seq(c) \wedge lh(c) = 4 \wedge (c)_0 \in Ind(\mathbf{P}) \wedge (c)_{00} = 5 \wedge Seq((c)_1) \wedge lh((c)_1) = (c)_{01} \\ & \wedge Seq((c)_3) \wedge lh((c)_3) = (c)_2 + 1 \\ & \wedge (\forall i \leq (c)_2)[B((c)_{3i}) \wedge (c)_{3i0} = (c)_{02} \wedge (c)_{3i1} = (c)_1 \wedge (i = (c)_2 \vee (c)_{3i2} > 0)] \\ & \wedge (c)_{3((c)_2)2} = 0. \end{aligned}$$

Dies ist genau die Anwendung der Definition primitiv rekursiver Prädikate durch Wertverlaufsrekursion, wie wir sie im Abschnitt 3.4 erwähnt haben.

Um den Zusammenhang mit der oben angeführten Klausel 6. besser zu durchschauen, sollte man sich vor Augen halten, daß hier  $(c)_0$  der Index ' $\mu f$ ' ist,  $(c)_1$  den Input  $\langle x_1, \dots, x_n \rangle$  kodiert,  $(c)_2$  das Ergebnis  $z$  der Rechnung ist und schließlich  $(c)_3$  die oben mit  $\langle b_0, \dots, b_z \rangle$  bezeichnete Folge der Teilberechnungen repräsentiert. Die Formel gibt dann genau den in 6. beschriebenen Sachverhalt formal wieder.

Analog erhalten wir die Prädikate  $B_i(\langle e, x, y, b \rangle)$  für  $i = 0, \dots, 4$  und können dann

$$B(\langle e, x, y, b \rangle) : \iff B_0(\langle e, x, y, b \rangle) \vee B_1(\langle e, x, y, b \rangle) \vee B_2(\langle e, x, y, b \rangle) \vee B_3(\langle e, x, y, b \rangle) \vee B_4(\langle e, x, y, b \rangle) \vee B_5(\langle e, x, y, b \rangle)$$

definieren.

Gilt  $B(b)$ , so erhalten wir auch hier das Ergebnis der Rechnung in primitiv rekursiver Weise als

$$(b)_2,$$

und wir können

$$T^n(e, x_1, \dots, x_n, b) : \iff B(b) \wedge (b)_0 = e \wedge (b)_1 = \langle x_1, \dots, x_n \rangle$$

definieren.

Eigenschaften des Berechnungsprädikates

Was wir von diesen Überlegungen festzuhalten haben, sind die folgenden Tatsachen:

- Zu jeder partiellrekursiven Funktion  $f$  läßt sich ein Kode ' $f$ ' definieren, der den Algorithmus (also entweder das Programm oder den Funktionsterm für  $f$ ) kodiert. Wir nennen ' $f$ ' einen Index für  $f$ .
- Es gibt ein primitiv rekursives Prädikat  $T^n(e, x_1, \dots, x_n, b)$ , das ausdrückt, daß  $b$  eine Berechnung einer partiellrekursiven Funktion mit Index  $e$  beim Input  $x_1, \dots, x_n$  kodiert.
- Das Endergebnis der Berechnung läßt sich in primitiv rekursiver Weise aus dem Berechnungskode  $b$  errechnen.

Zusammenfassend erhalten wir damit den folgenden Satz:

Indizes partiellrekursiver Funktionen

Normalformensatz von KLEENE

**Satz 17 (Normalformensatz von KLEENE)** Es gibt ein  $n + 2$ -stelliges primitiv rekursives Prädikat  $T^n$  und eine primitiv rekursive Funktion  $U$  derart, daß zu jeder  $n$ -stelligen partiellrekursiven Funktion  $f$  ein Index  $e \in \mathbb{N}$  existiert mit

$$f(x_1, \dots, x_n) \simeq U(\mu b. T^n(e, x_1, \dots, x_n, b)).$$

Es gibt viele Möglichkeiten, den Normalformensatz zu beweisen. Wir haben zwei davon angedeutet. Obwohl es völlig unwesentlich ist, wie das KLEENEsche  $T$ -Prädikat gewonnen wurde, ist es doch oft nützlich, sich  $T^n(e, x_1, \dots, x_n, b)$  als die Aussage

*" $b$  ist eine Berechnung für die Funktion mit Index  $e$  beim Input  $x_1, \dots, x_n$ "*

vorzustellen.

Wir führen die Notation

$$\{e\}^n(x_1, \dots, x_n) : \simeq U(\mu b. T^n(e, x_1, \dots, x_n, b))$$

ein. Definieren wir dann

$$\Phi^n(e, x_1, \dots, x_n) : \simeq \{e\}^n(x_1, \dots, x_n),$$

so ist  $\Phi^n$  eine *universelle* Funktion für die  $n$ -stelligen partiellrekursiven Funktionen. Dabei nennen wir eine Funktion  $UN$  universell für eine Klasse  $\mathcal{F}$  partieller Funktionen, wenn zu jedem  $f \in \mathcal{F}$  eine natürliche Zahl  $e$  existiert mit

$$f(\vec{x}) \simeq UN(e, \vec{x}).$$

Am Beispiel der maschinenberechenbaren Funktionen kann man sich klarmachen, daß die Bedeutung universeller Funktionen nicht unterschätzt werden darf. Üblicherweise benötigen wir zur Berechnung einer maschinenberechenbaren Funktion  $f$  eine Maschine  $C_f$ . Wir haben uns die Maschine  $C_f$  so vorzustellen, daß das Steuerprogramm in eine Steuereinheit eingegeben werden muß. Zur Berechnung unterschiedlicher Funktionen müssen wir daher, selbst bei unveränderter Basismaschine, die Steuereinheit umprogrammieren (wie dies bei den ersten programmgesteuerten Maschinen auch tatsächlich der Fall war). Da die maschinenberechenbaren Funktionen aber genau die partiellrekursiven Funktionen sind, wissen wir nun, daß  $\Phi$  eine universelle Funktion für die maschinenberechenbaren Funktionen ist. Da die Funktion  $\Phi$  offenbar partiellrekursiv und damit maschinenberechenbar ist, so können wir eine *universelle Maschine*  $C_\Phi$  bauen, die  $\Phi$  berechnet. Um nun  $f(x_1, \dots, x_n)$  für eine beliebige maschinenberechenbare Funktion  $f$  zu berechnen, müssen wir nicht mehr die Steuereinheit der Basismaschine umbauen, sondern es genügt, den Index, d.h. einen Algorithmus für  $f$  zu kennen. Ist dieser Index  $e$ , so setzen wir  $C_\Phi$  einfach auf den Input  $(e, x_1, \dots, x_n)$  an. Die universelle Maschine berechnet nun  $\Phi(e, x_1, \dots, x_n)$ , d.h.  $\{e\}^n(x_1, \dots, x_n)$ , was, falls die Berechnung erfolgreich war, der Wert  $f(x_1, \dots, x_n)$  ist. Die Maschinenberechenbarkeit der universellen Funktion versetzt uns also in die Lage, das Programm für die Funktion  $f$  (d.h. ihren Index  $e$ ) in den Datenspeicher zu schreiben. Sie ermöglicht uns damit den Schritt von

Universelle  
Funktionen

Bedeutung  
universeller  
Funktionen

der Hardware, die durch die Basismaschine und deren Steuerprogramm gegeben ist, zur Software, die durch den Index der Funktion gegeben ist. Offensichtlich ist dies genau die Art, in der wir heute unsere Rechner programmieren. Der Algorithmus für die universelle Funktion gehört dabei zu dem maschinennahen Teil des Betriebssystems, und liegt in der Regel in Form von Hard- oder Firmware (EPROMs) vor. Das Programm schreiben wir in den Datenspeicher. Die heutigen Rechner sind also physikalische Realisationen universeller Maschinen, die natürlich mit all den Unzulänglichkeiten physikalischer Realisationen gegenüber der Idealisierung behaftet sind, wie beispielsweise endlichem Speicher, und daher im strengen Sinne keine universellen Maschinen sein können.

### 3.9 Rekursive, semirekursive und rekursiv aufzählbare Prädikate

In den in 3.2 eingeführten primitiv rekursiven Prädikaten haben wir bereits entscheidbare Relationen kennengelernt. Um  $R(x_1, \dots, x_n)$  zu entscheiden, hat man die charakteristische Funktion  $\chi_P(x_1, \dots, x_n)$  zu berechnen und erhält eine positive Antwort auf das Entscheidungsproblem, falls das Ergebnis 1, eine negative, falls das Ergebnis 0 ist. Das lässt sich natürlich sofort auf Relationen mit rekursiven charakteristischen Funktionen übertragen. Also definieren wir:

Rekursive Prädikate

**Definition 18** Eine Relation (Prädikat)  $P$  heißt rekursiv, wenn ihre charakteristische Funktion  $\chi_P$  eine rekursive Funktion ist.

Dieser Definition schließt sich dann unmittelbar die folgende Beobachtung an.

**Satz 19** Ist  $P$  ein  $n + 1$ -stelliges rekursives Prädikat, so ist die durch den unbeschränkten Suchoperator definierte  $n$ -stellige Funktion

$$\mu x . P(x, y_1, \dots, y_n)$$

partiell rekursiv.

Abschlußeigenschaften rekursiver Prädikate

Der erste Schritt in der Untersuchung rekursiver Prädikate wird natürlich wieder im Studium ihrer Abschlußeigenschaften bestehen. Die gleichen Überlegungen, wie wir sie in Abschnitt 3.2 geführt haben, zeigen uns, daß alle die Abschlußeigenschaften, die den primitiv rekursiven Prädikaten zukommen, auch für die rekursiven gelten müssen. Dabei können wir die Substitution rekursiver (also totaler) Funktionen zulassen. Wir haben daher den folgenden Satz:

### 3.9. Rekursive, semirekursive und rekursiv aufzählbare Prädikate

**Satz 20** Die rekursiven Relationen sind abgeschlossen gegenüber allen booleschen Operationen, beschränkter Quantifikation und der Substitution mit rekursiven Funktionen.

Neben dem Problem der Entscheidbarkeit hat man oft auch das Problem der *positiven Entscheidbarkeit* zu betrachten. Hier verlangt man nicht nach einem Algorithmus, der die Frage

$$\vec{x} \in P?$$

entscheidet, sondern bescheidet sich mit einem Algorithmus, der  $\vec{x} \in P$  bestätigt, d.h. der nur dann eine positive Antwort liefert, wenn  $\vec{x}$  tatsächlich zu  $P$  gehört. Im negativen Falle darf sein Verhalten völlig unbestimmt bleiben. Dies erinnert natürlich an die Situation, wie wir sie bei partiellrekursiven Funktionen vorfinden. Auch dort führt der Algorithmus zur Berechnung von  $f(\vec{x})$  nur dann zum Ziel, wenn  $\vec{x} \in \text{dom}(f)$  gilt, während andererfalls der Algorithmus nicht terminieren muß. Wir können daher die *semirekursiven Relationen* in der folgenden Weise als Präzisierung der positiv entscheidbaren Relationen einführen:

**Definition 21** Eine Relation  $P \subseteq \mathbb{N}^n$  heißt *semirekursiv*, wenn  $P$  der Definitionsbereich einer partiellrekursiven Funktion  $f$  ist, d.h. wenn

positive Entscheidbarkeit  
Semirekursive Relationen

$$P = \text{dom}(f)$$

gilt.

Ein weiterer wichtiger Begriff ist der der *rekursiv aufzählbaren Menge*, der dadurch gegeben ist, daß sich die Elemente der Menge durch eine rekursive Funktion aufzählen lassen. Exakt läßt sich dies folgendermaßen definieren:

**Definition 22**

1. Eine Menge  $M \subseteq \mathbb{N}$  heißt *rekursiv aufzählbar*, wenn  $M$  leer ist, oder es eine rekursive (und damit totale) Funktion  $f$  gibt mit

rekursiv aufzählbare Mengen und Relationen

$$M = \text{rng}(f) := \{f(x) \mid x \in \mathbb{N}\}.$$

2. Eine Relation  $R \subseteq \mathbb{N}^n$  heißt *rekursiv aufzählbar*, wenn ihre Kontraktion

$$\langle R \rangle := \{\langle x_1, \dots, x_n \rangle \mid R(x_1, \dots, x_n)\}$$

rekursiv aufzählbar ist.

Es sollte auffallen, daß rekursiv aufzählbare Mengen sicherlich auch positiv entscheidbar sind. Einen Bestätigungsalgorithmus für eine nicht leere rekursiv aufzählbare Menge  $M$  mit der Aufzählungsfunktion  $f$

### 3. $\mu$ -rekursive Funktionen

---

erhält man, indem man sich die Liste

$$f(0), f(1), f(2), \dots$$

erzeugt. Ist  $x \in M$ , so gibt es ein  $n \in \mathbb{N}$  mit  $x = f(n)$ , was bedeutet, daß  $x \in M$  nach endlich vielen Schritten bestätigt wird. Für den Fall, daß  $x$  nicht zu  $M$  gehört, liefert dieser Algorithmus jedoch keine Information, da wir zu keiner Zeit  $n$  wissen, ob nicht ein  $m > n$  mit  $x = f(m)$  existiert. Dies legt die Vermutung nahe, daß die semirekursiven und rekursiv aufzählbaren Relationen übereinstimmen könnten. Um dies weiter zu untersuchen, stellen wir zunächst fest, daß wir eine nicht leere rekursiv aufzählbare Menge  $M$  mit Aufzählungsfunktion  $f$  immer durch

$$x \in M \iff (\exists y)[f(y) = x]$$

darstellen können. Da  $f$  eine rekursive Funktion ist, handelt es sich bei dem Prädikat

$$R(x, y) : \iff f(y) = x$$

nach Satz 20 um ein rekursives Prädikat. Ist umgekehrt

$$M = \{x \mid (\exists y)R(x, y)\}$$

nicht leer und  $R$  dabei ein rekursives Prädikat, so wählen wir ein  $u \in M$  und definieren eine rekursive Funktion  $f$  vermöge

$$f(x) := \begin{cases} (x)_0, & \text{falls } R((x)_0, (x)_1), \\ u, & \text{sonst.} \end{cases}$$

Dann gilt offenbar

$$M = \text{rng}(f)$$

und  $M$  ist damit rekursiv aufzählbar. Da wir die leere Menge leicht durch

$$x \in \emptyset \iff (\exists y)[x \neq x]$$

beschreiben können, erhalten wir, wenn wir die Kontraktionen rekursiv aufzählbarer Relationen betrachten, den folgenden Satz:

Charakterisierung  
rekursiv  
aufzählbarer  
Relationen

**Satz 23** Eine  $n$ -stellige Relation  $P \subseteq \mathbb{N}^n$  ist genau dann rekursiv aufzählbar, wenn es eine  $n + 1$ -stellige rekursive Relation  $R$  gibt mit

$$(x_1, \dots, x_n) \in P \iff (\exists y)R(x_1, \dots, x_n, y).$$

Da jede rekursive Relation  $R$  sich durch Hinzufügen leerer Existenzquantoren auf die Gestalt  $(\exists y)R(y, \dots)$  bringen läßt, erhalten wir als Korollar

### 3.9. Rekursive, semirekursive und rekursiv aufzählbare Prädikate

zu Satz 23, daß die Klasse  $Rel_r$ , der rekursiven Relationen eine Teilklasse der Klasse  $Rel_{r.a.}$ , der rekursiv aufzählbaren Relationen ist.

Wir erhalten aber aus Satz 23 auch, daß jede rekursiv aufzählbare Relation bereits semirekursiv ist, denn gilt

$$P(\vec{x}) \iff (\exists y)R(\vec{x}, y)$$

für eine rekursive Relation  $R$ , so ist offenbar

$$P = \text{dom}(\mu y. R(\vec{x}, y))$$

und  $P$  ist damit Definitionsbereich einer partiellrekursiven Funktion.

Nun gilt aber für eine partiellrekursive Funktion nach dem KLEENEschen Normalformentheorem

$$f(\vec{x}) \simeq U(\mu b. T^n(e, \vec{x}, b)),$$

woraus sich

$$\vec{x} \in \text{dom}(f) \iff (\exists y)T^n(e, \vec{x}, y) \quad (3.24)$$

ergibt. Setzen wir

$$W_e^n := \{(x_1, \dots, x_n) \mid (\exists y)T^n(e, x_1, \dots, x_n, y)\},$$

so ergibt sich die Klasse der  $n$ -stelligen semirekursiven Prädikate zu

$$Rel_{s.r.}^n = \{W_e^n \mid e \in \text{Ind}(\mathbf{P})\}. \quad (3.25)$$

Nach Satz 23 ist aber jede der Relationen  $W_e^n$  rekursiv aufzählbar und wir erhalten so das folgende Theorem.

**Satz 24** Die Klasse  $Rel_{r.a.}$  der rekursiv aufzählbaren Relationen stimmt mit der Klasse  $Rel_{s.r.}$  der semirekursiven Relationen überein.

Semirekursive vs.  
rekursiv aufzählbare  
Relationen

Wegen Satz 24 werden wir im folgenden nur noch von rekursiv aufzählbaren Relationen sprechen. Damit folgen wir der Konvention, wie sie insbesondere in der älteren Literatur üblich ist. Der Begriff der semirekursiven Relation – der eigentlich der entscheidendere ist – begann erst eine Rolle zu spielen, als man entdeckte, daß in gewissen Verallgemeinerungen der Rekursionstheorie die Begriffe von “semirekursiv” und “rekursiv aufzählbar” auseinanderklaffen können.

Die Charakterisierung der rekursiv aufzählbaren Relationen, wie sie durch Satz 23 gegeben wird (und wie sie sich für semirekursive Relation aus dem KLEENEschen Normalformensatz ergibt), wird uns auch erlauben, deren Abschlußeigenschaften zu studieren. So erkennen wir beispielsweise sofort, daß die rekursiv aufzählbaren Relationen unter den *positiven booleschen* Operationen  $\vee$  und  $\wedge$  sowie gegenüber der

Positive boolesche  
Operationen

### 3. $\mu$ -rekursive Funktionen

---

Substitution mit rekursiven Funktionen abgeschlossen sind. Dazu stellen wir die rekursiv aufzählbaren Relationen  $P_1$  und  $P_2$  als

$$P_i(\vec{x}) \iff (\exists y) R_i(\vec{x}, y)$$

mit rekursiven Relationen  $R_i$  für  $i = 1, 2$  dar und erhalten

$$P_1(\vec{x}) \wedge P_2(\vec{x}) \iff (\exists z)[R_1(\vec{x}, (z)_0) \wedge R_2(\vec{x}, (z)_1)].$$

Die Relation in eckigen Klammern auf der rechten Seite der Äquivalenz ist nach Satz 20 rekursiv. Somit ist nach Satz 23 die links stehende Relation rekursiv aufzählbar. Analog verfahren wir für  $\vee$ , und in ähnlicher Weise erhalten wir für die Substitution mit rekursiven Funktionen  $f_1, \dots, f_n$

$$P_i(f_1(\vec{x}), \dots, f_n(\vec{x})) \iff (\exists y) R_i(f_1(\vec{x}), \dots, f_n(\vec{x}), y)$$

und schließen wie eben auf die rekursive Aufzählbarkeit des links stehenden Prädikates. Etwas aufwendiger ist der Nachweis des Abschlusses gegenüber beschränkten Quantoren. Beginnen wir mit dem  $\forall$ -Quantor. Mit Satz 23 ergibt sich für ein rekursiv aufzählbares Prädikat  $P$

$$(\forall z < x) P(\vec{x}, z) \iff (\forall z < x) (\exists y) R(\vec{x}, z, y) \quad (3.26)$$

mit einem rekursiven Prädikat  $R$ . Gilt die linke Seite von (3.26), so haben wir zu jedem  $z < x$  ein  $y_z$  mit  $R(\vec{x}, z, y_z)$ . Alle diese  $y_z$  kodieren wir zu einer Folge  $s := \langle y_0, \dots, y_{x-1} \rangle$  der Länge  $x$  zusammen und erhalten somit die Richtung von links nach rechts in der folgenden Äquivalenz.

$$(\forall z < x) P(\vec{x}, z) \iff (\exists s)[\text{Seq}(s) \wedge (\forall z < x) R(\vec{x}, z, (s)_z)].$$

Die Richtung von rechts nach links ergibt sich trivialerweise aus (3.26). Mit Hilfe von Satz 23 erkennen wir den Ausdruck in eckigen Klammern wieder als rekursiv, woraus sich wieder die rekursive Aufzählbarkeit des mit Hilfe des beschränkten Allquantors aus  $P$  definierten Prädikates ergibt.

Die Abgeschlossenheit gegenüber dem beschränkten Existenzquantor ergibt sich aus einer noch allgemeineren Abschlußeigenschaft. Wenn wir wieder von einem rekursiv aufzählbaren  $P$  ausgehen, so erhalten wir nämlich

$$\begin{aligned} (\exists z) P(\vec{x}, z) &\iff (\exists z) (\exists y) R(\vec{x}, z, y) \\ &\iff (\exists u) R(\vec{x}, (u)_0, (u)_1). \end{aligned}$$

Damit folgt, daß die rekursiv aufzählbaren Relationen auch gegenüber globaler Existenzquantifikation abgeschlossen sind. Wegen

### 3.9. Rekursive, semirekursive und rekursiv aufzählbare Prädikate

$$(\exists z < x) P(\vec{x}, z) \iff (\exists z)[z < x \wedge P(\vec{x}, z)]$$

folgt damit auch die Abgeschlossenheit gegenüber beschränkter Existenzquantifikation.

Zusammenfassend erhalten wir:

**Satz 25** Die rekursiv aufzählbaren Relationen sind abgeschlossen gegenüber Durchschnitten und Vereinigungen, der Substitution mit rekursiven Funktionen, der Quantifikation mit beschränkten Quantoren und dem unbeschränkten Existenzquantor.

Abschlußeigenschaften rekursiv aufzählbarer Relationen

Zu klären bleibt noch die Frage, ob die rekursiv aufzählbaren Relationen auch gegenüber der Negation abgeschlossen sind. Da wir bereits den Abschluß gegenüber der Existenzquantifikation haben und

$$\neg(\exists y) R(y, \dots) \iff (\forall y)[\neg R(y, \dots)]$$

gilt, würde Abschluß gegenüber Negation auch den Abschluß gegenüber unbeschränkter Allquantifikation nach sich ziehen. Damit wären aber alle Relationen rekursiv aufzählbar, die sich ausgehend von primitiv rekursiven Relationen durch Quantifikationen bilden lassen. Da dies die in der Sprache der Arithmetik definierbaren Relationen sind, nennt man diese die *arithmetischen Relationen*. Die arithmetischen Relationen lassen sich nach ihrer Quantorenkomplexität in eine Hierarchie, die *arithmetische Hierarchie* einordnen. Es hat sich eingebürgert, für den  $\forall$ -Quantor das Symbol  $\Pi$  und für den  $\exists$ -Quantor das Symbol  $\Sigma$  zu verwenden. Da sich zwei Quantoren gleicher Art wegen der Äquivalenz

Die arithmetische Hierarchie

$$(Qx)(Qy)[\dots x \dots y \dots] \iff (Qu)[\dots (u)_0 \dots (u)_1 \dots]$$

kontrahieren lassen – dies haben wir benutzt, als wir die Abgeschlossenheit der rekursiv aufzählbaren Relationen gegenüber dem  $\exists$ -Quantor gezeigt haben –, lässt sich höchstens von einem Quantorenwechsel ein Komplexitätszuwachs erwarten. Man spricht nun von einer  $\Pi_n$ -Relation  $P$ , wenn

$\Pi_n$ - und  $\Sigma_n$ -Relationen

$$P(x_1, \dots, x_m) \iff (\forall y_1)(\exists y_2)(\forall y_3) \dots (Qy_n)R(x_1, \dots, x_m, y_1, \dots, y_n)$$

ist, wobei  $R$  eine primitiv rekursive Relation ist und die Quantoren alternieren. Für gerades  $n$  ist daher  $Q$  ein  $\exists$ - und für ungerades  $n$  ein  $\forall$ -Quantor. Dual spricht man von  $\Sigma_n$ -Relationen  $P$ , wenn

$$P(x_1, \dots, x_m) \iff (\exists y_1)(\forall y_2)(\exists y_3) \dots (\check{Q}y_n)R(x_1, \dots, x_m, y_1, \dots, y_n)$$

gilt, wobei  $R$  wieder rekursiv und  $\check{Q}$  der zu  $Q$  duale Quantor ist. Die Klassen  $\Pi_n$  und  $\Sigma_n$  sind also in dem Sinne dual definiert, als daß

### 3. $\mu$ -rekursive Funktionen

---

$$P \in \Pi_n \iff \mathcal{C}(P) \in \Sigma_n \quad \text{und} \quad P \in \Sigma_n \iff \mathcal{C}(P) \in \Pi_n$$

geln. Oft spricht man auch von  $\Pi_n^0$ - bzw.  $\Sigma_n^0$ -Relationen, wenn man deutlich machen will, daß die zur Definition benutzten Quantoren nur über den Typ 0, den Grundbereich, quantifizieren, der in unserem Falle der Bereich der natürlichen Zahlen ist.

In der Rekursionstheorie wird nun gezeigt, daß diese Hierarchie in der Tat eine echte Hierarchie ist, was heißen soll, daß die Klasse der  $\Pi_{n+1}$ -Relationen eine *echte* Oberklasse der  $\Sigma_n$ -Relationen und analog die Klasse der  $\Sigma_{n+1}$ -Relationen eine echte Oberklasse der  $\Pi_n$ -Relationen ist. Man kann dies noch dazu verstärken, daß der Durchschnitt  $\Pi_{n+1} \cap \Sigma_{n+1}$ , der üblicherweise als die Klasse der  $\Delta_{n+1}$ -Relationen bezeichnet wird, eine echte Oberklasse sowohl der  $\Pi_n$ - als auch der  $\Sigma_n$ -Relationen ist.

$\Delta_n$ -Relationen

Relevant für die Informatik sind allerdings nur die Grundlevel dieser Hierarchie.

Da wir jede  $n$ -stellige rekursiv aufzählbare Relation  $P$  als eine Relation  $W_e^n$  erhalten, wobei

$$W_e^n(\vec{x}) \iff (\exists y) T^n(e, \vec{x}, y)$$

für das primitiv rekursive KLEENE-Prädikat  $T^n$  ist, sehen wir, daß die rekursiv aufzählbaren Relationen offenbar genau die  $\Sigma_1$ -Relationen sind.

Wir haben uns bereits klargemacht, daß die rekursiv aufzählbaren Relationen eine Präzisierung der intuitiv positiv entscheidbaren Relationen sind. Haben wir nun eine rekursiv aufzählbare Relation  $P$  so vorliegen, daß auch deren Komplement  $\mathcal{C}(P)$  rekursiv aufzählbar ist, so erhalten wir offenbar ein Entscheidungsverfahren für  $P(\vec{x})$ , indem wir die Bestätigungsalgorithmen für  $P(\vec{x})$  und  $\neg P(\vec{x})$  abwechselnd jeweils einen Schritt laufen lassen. Da entweder  $P(\vec{x})$  oder  $\neg P(\vec{x})$  gelten muß, wird einer der beiden Algorithmen schließlich eine Antwort liefern. Damit folgt, daß eine Relation genau dann entscheidbar ist, wenn sie und ihr Komplement positiv entscheidbar sind. Benutzen wir hier der Bequemlichkeit halber die CHURCHsche These, so erhalten wir den folgenden Satz:

Rekursive vs.  
 $\Delta_1$ -Relationen

**Satz 26** (Satz von POST) Eine Relation  $P$  ist genau dann rekursiv, wenn  $P$  und dessen Komplement  $\mathcal{C}(P)$  rekursiv aufzählbar sind. Das bedeutet, daß die Klasse der rekursiven Relationen genau die Klasse der  $\Delta_1$ -Relationen in der arithmetischen Hierarchie ist.

Wir wollen uns nun klarmachen, daß die rekursiv aufzählbaren Relationen nicht unter Komplementen abgeschlossen sind. Dazu gehen wir von der Menge

$$K := \{x \mid (\exists y) T^1(x, x, y)\}$$

### 3.9. Rekursive, semirekursive und rekursiv aufzählbare Prädikate

Relationenklasse	$\neg$	$\vee$	$\wedge$	$\exists \leq$	$\forall \leq$	$\exists$	$\forall$	Substitution mit
Primitiv rekursiv	ja	ja	ja	ja	ja	nein	nein	primitiv rekursiven Funktionen
Rekursiv	ja	ja	ja	ja	ja	nein	nein	rekursiven Funktionen
Rek. aufzählbar	nein	ja	ja	ja	ja	ja	nein	rekursiven Funktionen
$\Pi_1$ -Relationen	nein	ja	ja	ja	ja	nein	ja	rekursiven Funktionen

Abb. 3.9 - 1: Abschlußeigenschaften von Relationenklassen

aus. Die Menge  $K$  ist  $\Sigma_1$  und somit rekursiv aufzählbar. Wären alle rekursiv aufzählbaren Mengen gegenüber Komplementen abgeschlossen, so wäre auch das Komplement  $\mathcal{C}(K)$  von  $K$  rekursiv aufzählbar. Dann hätten wir aber

$$\mathcal{C}(K) = W_e^1$$

für ein  $e \in \text{Ind}(\mathbf{P})$ . Dann folgte aber

$$\begin{aligned} e \in K &\iff (\exists y)T^1(e, e, y) \\ &\iff e \in W_e^1 \\ &\iff e \in \mathcal{C}(K), \end{aligned}$$

was ein offensichtlicher Widerspruch ist. Damit ist das Komplement von  $K$  nicht rekursiv aufzählbar. Natürlich lässt sich der gleiche Widerspruch nicht nur für rekursiv aufzählbare Mengen, sondern auch für rekursiv aufzählbare Relationen beliebiger Stellenzahl erreichen.

Eine Zusammenfassung der Abschlußeigenschaften der bisher untersuchten Prädikate finden sich in der Tabelle in Abbildung 3.9 - 1. Wir haben aber mit dem gerade geführten Widerspruchsbeweis noch mehr eingesehen als nur die Tatsache, daß die rekursiv aufzählbaren Relationen nicht unter Komplementen abgeschlossen sind. Es folgt nämlich auch, daß die oben definierte Menge  $K$  nicht rekursiv sein kann. Denn dann wären nach dem Satz von POST (Satz 26) sowohl  $K$  als auch dessen Komplement  $\mathcal{C}(K)$  rekursiv aufzählbar. Wir haben uns aber gerade überlegt, daß dies unmöglich ist. Auch dieses Ergebnis wollen wir in einen Satz fassen.

Eine rekursiv aufzählbare Menge, die nicht rekursiv ist

**Satz 27** *Es gibt eine rekursiv aufzählbare Relation, die nicht rekursiv ist. Damit sind die rekursiv aufzählbaren Relationen gegenüber Komplementbildung nicht abgeschlossen.*

Mit Satz 27 haben wir die Echtheit der Anfangslevel der arithmetischen Hierarchie nachgewiesen. Da die rekursiven Relationen gerade die  $\Delta_1$ -

### 3. $\mu$ -rekursive Funktionen

---

Relationen sind, besagt Satz 27 ja gerade

$$\Delta_1 \neq \Sigma_1.$$

Daraus folgt aber auch

$$\Delta_1 \neq \Pi_1.$$

Nehmen wir nämlich  $\Delta_1 = \Pi_1$  an, so erhalten wir für  $P \in \Sigma_1$  dann  $C(P) \in \Pi_1 = \Delta_1$  und daraus wiederum  $P \in \Delta_1$ , da  $\Delta_1$  gegenüber Komplementbildung abgeschlossen ist. Also folgte  $\Sigma_1 = \Delta_1$ , was wir aber gerade als falsch erkannt haben.

Das Halteproblem

Die Menge  $K$  gilt als Paradigma für unentscheidbare Mengen. Sie ist auch als *Halteproblem* bekannt. Diesen Zusammenhang wollen wir näher erläutern.

Wie wir in Abschnitt 3.8 klargelegt haben, lässt sich das KLEENEsche Prädikat  $T(e, \vec{x}, y)$  als

' $y$  kodiert eine Berechnung der Funktion mit Index  $e$  beim Input  $\vec{x}$ ' veranschaulichen. Wie wir weiter skizziert haben, lässt sich der Index einer Funktion  $f$  als der Kode eines Turingprogrammes (oder anderen Maschinenprogrammes) für die Funktion  $f$  auffassen. Das Halteproblem für Turingmaschinen kann man nun wie folgt formulieren:

*Gibt es einen Algorithmus, der entscheidet, ob ein Turingprogramm mit Kode  $e$  bei dem Input  $\vec{x}$  terminiert?*

D.h. unter Benutzung des KLEENEschen  $T$ -Prädikats:

'Ist die Relation

$$\{(e, x_1, \dots, x_n) \mid (\exists y) T^n(e, x_1, \dots, x_n, y)\}$$

eine rekursive Relation?'

Das Halteproblem  
ist unentscheidbar

Wäre die Antwort darauf positiv, so müßte als Spezialfall davon auch die Menge  $K$  rekursiv sein, was gemäß unseren vorherigen Überlegungen nicht richtig sein kann. Also ist das Halteproblem für Turingmaschinen nicht entscheidbar.

## 3.10 Universelle Funktionen und der Rekursionssatz

Den Begriff einer universellen Funktion hatten wir bereits in Abschnitt 3.8 eingeführt und kurz diskutiert. Dort hatten wir über den KLEENEschen Normalformensatz feststellen können, daß die Klasse der partiellrekursiven Funktionen, die ja mit der Klasse der partiellen maschinenberechenbaren Funktionen übereinstimmt, eine universelle Funktion besitzt. Im vorliegenden Abschnitt wollen wir die Konsequenzen der

Existenz universeller Funktionen genauer studieren.

Betrachten wir zunächst den *Graphen*

$$G_f := \{(\vec{x}, y) \mid f(\vec{x}) \simeq y\}$$

einer partiellrekursiven Funktion, so erhalten wir mit dem Normalformensatz

$$(\vec{x}, y) \in G_f \iff (\exists z)[T^n(e, \vec{x}, z) \wedge (\forall u < z) \neg T^n(e, \vec{x}, u) \wedge U(z) = y].$$

Der Ausdruck innerhalb der eckigen Klammern ist primitiv rekursiv, was zeigt, daß  $G_f$  für jede partiellrekursive Funktion rekursiv aufzählbar ist. Liegt uns umgekehrt eine partielle Funktion  $f$  mit rekursiv aufzählbarem  $G_f$  vor, so gibt es eine rekursive Relation  $R$  mit

$$f(\vec{x}) \simeq y \iff (\exists z)R(\vec{x}, y, z).$$

Offenbar gilt dann aber

$$f(\vec{x}) \simeq (\mu z . R(\vec{x}, (z)_0, (z)_1))_0$$

was zeigt, daß  $f$  partiellrekursiv ist. Damit haben wir folgenden Satz:

**Satz 28** Eine Funktion ist genau dann partiellrekursiv, wenn ihr Graph rekursiv aufzählbar ist.

Funktionen mit  
rekursiv  
aufzählbarem  
Graphen

Dieser Satz ist ein bequemes Instrument beim Nachweis der partiellen Rekursivität von Funktionen. Wir wollen ihn hier verwenden, um die partielle Rekursivität der in Abschnitt 3.8 eingeführten universellen Funktion, die wir damals nicht näher begründet hatten, zu bestätigen. Dort hatten wir

$$\Phi^n(e, \vec{x}) : \simeq \{e\}^n(\vec{x})$$

definiert. Also gilt

$$\Phi^n(e, \vec{x}) \simeq y \iff (\exists z)[T^n(e, \vec{x}, z) \wedge (\forall u < z) \neg T^n(e, \vec{x}, u) \wedge U(z) = y],$$

womit gezeigt ist, daß  $G_{\Phi^n}$  einen rekursiv aufzählbaren Graphen hat. Damit ist  $\Phi^n$  als partiellrekursiv erkannt. Also gilt:

**Satz 29** Die Funktionen  $\Phi^n(e, \vec{x}) : \simeq \{e\}^n(\vec{x})$  sind partiellrekursiv und universell für die Klasse der  $n$ -steligen partiellrekursiven Funktionen.

Dieser Satz liefert uns den theoretischen Hintergrund für die Möglichkeit des Baues universeller Maschinen, die sich über Software programmieren lassen, wie wir dies am Ende des Abschnittes 3.8 diskutiert haben.

Theoretischer  
Hintergrund  
universeller  
Maschinen

Wir haben bei unserem Vorgehen die universellen Funktionen über eine Indizierung der partiellrekursiven Funktionen gefunden. Umgekehrt

### 3. $\mu$ -rekursive Funktionen

---

induziert natürlich jede universelle Funktion  $UN^n$  für die  $n$ -stelligen partiellrekursiven Funktionen auch eine Indizierung, denn zu jeder  $n$ -stelligen partiellrekursiven Funktion existiert dann eine natürliche Zahl  $e$  mit

$$f(\vec{x}) \simeq UN^n(e, \vec{x}),$$

und wir definieren  $e$  als einen Index von  $f$  bezüglich der universellen Funktion  $UN^n$ . Natürlich muß nicht jede sich so ergebende Indizierung geeignet sein, wobei wir eine Indizierung als geeignet bezeichnen wollen, wenn sich die Indizes bezüglich  $UN^n$  effektiv in solche bezüglich  $UN^m$  umrechnen lassen. Präziser formuliert soll das heißen:

Geeignete  
universelle  
Funktionen

**Definition 30** Eine Familie universeller Funktionen  $\{UN^n \mid n \in \mathbb{N}\}$  heißt geeignet, wenn alle Funktionen  $UN^n$  partiellrekursiv sind und es zu jedem  $m$  und  $n$  eine  $m+1$ -stellige primitiv rekursive Funktion  $S_m^n$  gibt, derart daß

$$UN^{m+n}(e, x_1, \dots, x_m, y_1, \dots, y_n) \simeq UN^n(S_m^m(e, x_1, \dots, x_m), y_1, \dots, y_n)$$

gilt.

Wir wollen uns davon überzeugen, daß die von uns gewählte Indizierung tatsächlich geeignet ist. Wir wissen bereits, daß die Funktionen  $\Phi^n$  alle partiellrekursiv sind. Nachzuprüfen ist also nur noch deren primitiv rekursive Umrechenbarkeit. Setzen wir

$$F(y_1, \dots, y_n) \simeq \Phi^{m+n}(e, x_1, \dots, x_m, y_1, \dots, y_n),$$

so ist  $F$  durch den Funktionsterm

$$\text{Sub}(\{e\}^{m+n}, C_{x_1}^n, \dots, C_{x_m}^n, P_1^n, \dots, P_n^n)$$

gegeben. Dessen Index berechnet sich aber zu

$$\langle 3, n, e, \langle 0, n, x_1 \rangle, \dots, \langle 0, n, x_m \rangle, \langle 1, n, 1 \rangle, \dots, \langle 1, n, n \rangle \rangle. \quad (3.27)$$

Dies hängt bei festem  $m$  und  $n$  aber offenbar nur in primitiv rekursiver Weise von den Argumenten  $e$  und  $x_1, \dots, x_m$  ab. Definieren wir  $S_n^m(e, x_1, \dots, x_n)$  als den Wert in (3.27), so gilt

$$\Phi^{m+n}(e, \vec{x}, \vec{y}) \simeq F(\vec{y}) \simeq \{S_n^m(e, \vec{x})\}^n(\vec{y}) \simeq \Phi^n(S_n^m(e, \vec{x}), \vec{y}).$$

Also haben wir den folgenden Satz gezeigt:

Das  $S_n^m$ -Theorem

**Satz 31** Die durch  $\Phi^n(e, \vec{x}) \simeq \{e\}^n(\vec{x})$  gegebene Familie universeller Funktionen ist geeignet.

In der Literatur ist Satz 31 auch als  $S_n^m$ -Theorem bekannt, da durch ihn

die Existenz einer primitiv rekursiven Umrechnungsfunktion  $S_n^m$  garantiert wird. Das  $S_n^m$ -Theorem wird auch die einzige Eigenschaft sein, die wir von unserer Wahl der Indizierung der partiellrekursiven Funktionen ausnutzen werden. Jede andere geeignete Indizierung – so z.B. die über die Kodierung von Maschinenprogrammen – leistet das Gleiche.

Eine der verblüffenden Folgerungen des  $S_n^m$ -Theorems ist der Rekursionssatz. Im Rekursionssatz versucht man die folgende Funktionalgleichung

$$f(\vec{x}) \simeq g(f, \vec{x}) \quad (3.28)$$

für eine partiellrekursive Funktion  $g$  im Bereich der partiellrekursiven Funktionen zu lösen. Das kann natürlich in dieser Form nicht gelingen, da wir den Begriff der partiellen Rekursivität für Funktionen, deren Argumente selbst Funktionen sind, nicht eingeführt haben. Die Theorie der sogenannten *partiellrekursiven Funktionale*, unter deren Argumenten auch totale Funktionen vorkommen dürfen, lässt sich tatsächlich in befriedigender Weise entwickeln. Allerdings müssten wir zur Formulierung der Aufgabe in (3.28) *partielle* Funktionen als Argumente zulassen. Eine derartige Theorie ist schwieriger zu entwickeln. Da wir jedoch Indizes für partiellrekursive Funktionen zur Verfügung haben, können wir uns hier behelfen, indem wir einen Index  $[f]$  für  $f$  wählen und (3.28) abändern zu

$$f(\vec{x}) \simeq g([f], \vec{x}). \quad (3.29)$$

Jetzt ist der Ausdruck auf der rechten Seite wohldefiniert. Um einzusehen, daß partiellrekursive Funktionen  $f$  existieren, die (3.29) lösen, betrachten wir die Funktion

$$g(S_1^1(u, u), \vec{x}).$$

Diese ist wieder partiellrekursiv und besitzt daher einen Index  $e_0$ . Setzen wir nun

$$e := S_1^1(e_0, e_0),$$

so erhalten wir mit Hilfe des  $S_n^m$ -Theorems

$$\{e\}^n(\vec{x}) \simeq \{S_1^1(e_0, e_0)\}^n(\vec{x}) \simeq \{e_0\}^{n+1}(e_0, \vec{x}) \simeq g(S_1^1(e_0, e_0), \vec{x}) \simeq g(e, \vec{x}).$$

Also löst  $\{e\}^n$  die Gleichung (3.29) und wir haben den folgenden Satz:

**Satz 32 (Rekursionssatz)** Zu jeder  $n+1$ -stelligen partiellrekursiven Funktion  $g$  gibt es einen Index  $e \in \text{Ind}(\mathbf{P})$  derart, daß

$$\{e\}^n(\vec{x}) \simeq g(e, \vec{x})$$

### 3. $\mu$ -rekursive Funktionen

---

ist.

Der Rekursionssatz hat vielfältige Anwendungen und stellt ein wesentliches Mittel dar, die partielle Rekursivität von Funktionen nachzuweisen, die nicht mehr primitiv rekursiv sind. Dennoch soll man seine Aussage nicht überschätzen. Seiner Natur nach ist er ein kombinatorischer Satz. Er liefert lediglich partiellrekursive Funktionen und sagt nichts über deren Definitionsbereich aus. So existiert nach dem Rekursionssatz beispielsweise ein Index  $e$ , der die Gleichung

$$\{e\}^1(x) \simeq \Phi^1(e, x) + 1$$

löst. Für diesen Index gilt

$$\{e\}^1(x) \simeq \{e\}^1(x) + 1,$$

was bedeutet, daß die Funktion  $\{e\}^1$  einen leeren Definitionsbereich besitzt.

Die ACKERMANN  
Funktion

Um zu veranschaulichen, wie der Rekursionssatz eingesetzt werden kann, wollen wir die Rekursivität der ACKERMANNschen Funktion nachweisen. Diese Funktion ist definiert durch die Rekursionsgleichungen

$$f(0, y) = S(y),$$

$$f(S(x), 0) = f(x, 1)$$

und

$$f(S(x), S(y)) = f(x, f(S(x), y)).$$

Dies ist im wesentlichen die Funktion, die 1928 von W. ACKERMANN benutzt wurde, um nachzuweisen, daß es berechenbare, aber nicht primitiv rekursive Funktionen gibt. Man kann nämlich zeigen, daß sich für jede primitiv rekursive Funktion  $g$  eine natürliche Zahl  $c$  so finden läßt, daß

$$g(x_1, \dots, x_n) < f(c, \max\{x_1, \dots, x_n\})$$

gilt. Das zieht nach sich, daß die ACKERMANN Funktion  $f$  selbst nicht primitiv rekursiv sein kann. Wäre sie es, so gälte dies auch für ihre Diagonalisierung

$$h(x) := f(x, x),$$

und wir hätten ein  $c$  mit  $h(x) < f(c, x)$  für alle natürlichen Zahlen  $x$ . Wählen wir  $x$  als  $c$ , so ergibt dies aber den Widerspruch  $f(c, c) = h(c) < f(c, c)$ .

Andererseits ist die ACKERMANN Funktion offensichtlich intuitiv berechenbar. Wir wollen uns überzeugen, daß sie in der Tat auch rekursiv

### 3.10. Universelle Funktionen und der Rekursionssatz

ist. Dazu verschaffen wir uns vermöge des Rekursionssatzes einen Index  $e$ , der die folgende Gleichung löst:

Rekursivität der  
ACKERMANN  
Funktion

$$\begin{aligned} \{e\}^2(x, y) &\simeq S(y) \cdot \bar{sg}(x) + \Phi^2(e, Pd(x), 1) \cdot sg(x) \cdot \bar{sg}(y) + \quad (3.30) \\ &\quad \phi^2(e, Pd(x), \Phi^2(e, x, Pd(y))) \cdot sg(x) \cdot sg(y). \end{aligned}$$

Definieren wir  $f := \{e\}^2$ , so haben wir eine partiellrekursive Funktion mit:

$$f(x, y) \simeq \begin{cases} S(y), & \text{falls } x = 0, \\ f(Pd(x), 1), & \text{falls } x \neq 0 \text{ und } y = 0, \\ f(Pd(x), f(x, Pd(y))), & \text{falls } x \neq 0 \text{ und } y \neq 0. \end{cases}$$

Damit erfüllt  $f$  die Rekursionsgleichungen der ACKERMANN Funktion. Zu zeigen bleibt, daß  $f$  auch total ist. Dazu ist eine geschachtelte Induktion nötig. Man macht Hauptinduktion nach  $x$  und Nebeninduktion nach  $y$ . Für  $x = 0$  ist  $f(x, y) \simeq S(y)$ , was für alle  $y$  definiert ist. Für  $x \neq 0$  ist  $f(x, 0) \simeq f(Pd(x), 1)$ , wobei die rechte Seite nach Hauptinduktionsvoraussetzung definiert ist, und  $f(x, S(y)) \simeq f(Pd(x), f(x, y))$ . Dann ist  $f(x, y)$  aber nach Nebeninduktionsvoraussetzung definiert und hat einen Wert  $u$ . Nach Hauptinduktionsvoraussetzung folgt dann auch die Definiertheit von  $f(Pd(x), u)$ .

Dieses Beispiel mag als Paradigma dafür dienen, wie die Rekursivität von Funktionen nachgewiesen werden kann. Der eine, kombinatorische Teil umfaßt den Nachweis der partiellen Rekursivität, im anderen, nichtelementaren Teil hat man dann die Totalität der Funktion nachzuweisen, was im allgemeinen starke Induktionen erfordert. Dies spiegelt in gewisser Weise die Tatsache wider, daß  $e \in Ind(\mathbf{P})$ , also die partielle Rekursivität, primitiv rekursiv zu entscheiden ist, während  $\vec{x} \in \text{dom}(f)$ , d.h. die Definiertheit von  $f(x)$ , im allgemeinen nicht mehr rekursiv ist.

Die Länge der Induktion, die erforderlich ist, um die Totalität zu zeigen, liefert sogar ein gewisses Maß für die Komplexität der betrachteten rekursiven Funktion. Betrachtungen dieser Art führen in das Gebiet der *Beweistheorie*.

Beweistheorie

Der Vollständigkeit halber sei noch angemerkt, daß die von uns in Abschnitt 3.5 angegebene berechenbare, aber nicht primitiv rekursive Funktion  $\phi_n$ , natürlich auch rekursiv ist. Dies ergibt sich einfach aus der Überlegung, daß diese Funktion für Indizes primitiv rekursiver Funktionen mit der universellen Funktion  $\Phi^n$  übereinstimmt. Damit erhalten wir

$$\phi_n(e, \vec{x}) := \begin{cases} \Phi^n(e, \vec{x}), & \text{falls } e \in Ind(Pri), \\ 0, & \text{sonst.} \end{cases}$$

Diese Fallunterscheidung, die wir ja als

$$\phi_n(e, \vec{x}) \simeq \Phi^n(e, \vec{x}) \cdot sg(\chi_{Ind(Pri)}(e))$$

### 3. $\mu$ -rekursive Funktionen

---

schreiben können, liefert also eine partiellrekursive Funktion, deren Totalität sich dann durch Induktion nach der Definition von  $e \in \text{Ind}(\text{Pri})$  relativ leicht zeigen lässt.

Definition partieller  
Funktionen durch  
Fallunterscheidung

Rekursive – also totale – Funktionen lassen sich ganz analog zu den primitiv rekursiven Funktionen durch Fallunterscheidung nach rekursiven Prädikaten mit Hilfe der Fallunterscheidungsfunktionen  $\text{sg}$  und  $\overline{\text{sg}}$  definieren. In den obigen Beispielen haben wir dies bereits getan. Bei partiellen Funktionen führt dies jedoch unter Umständen zu Problemen mit dem Definitionsbereich der durch Fallunterscheidung definierten Funktion. Sind  $R_1, \dots, R_n$  rekursive Prädikate, die paarweise disjunkt sind und  $g_1, \dots, g_n$  partiellrekursive Funktionen, so möchten wir gerne eine Funktion  $f$  durch die Fallunterscheidung

$$f(\vec{x}) : \simeq \begin{cases} g_1(\vec{x}), & \text{falls } R_1(\vec{x}) \\ \vdots \\ g_n(\vec{x}), & \text{falls } R_n(\vec{x}) \end{cases} \quad (3.31)$$

definieren. Tun wir dies vermöge

$$f(\vec{x}) : \simeq g_1(\vec{x}) \cdot \chi_{R_1}(\vec{x}) + \dots + g_n(\vec{x}) \cdot \chi_{R_n}(\vec{x}), \quad (3.32)$$

so mag es geschehen, daß  $R_k(\vec{x})$  zutrifft und  $g_k(\vec{x})$  definiert ist, wir somit also  $f(\vec{x}) \simeq g_k(\vec{x})$  erwarten, aber für ein von  $k$  verschiedenes  $j$  der Funktionswert  $g_j(\vec{x})$  nicht definiert ist. Dann ist aber die gemäß (3.32) definierte Funktion an der Stelle  $\vec{x}$  undefiniert, was nicht unsere Intention war. Man kann sich nun so behelfen, daß man Indizes  $e_1, \dots, e_n$  für die Funktionen  $g_1, \dots, g_n$  und einen Index  $e_0$  für eine partiellrekursive Funktion mit leerem Definitionsbereich wählt und die Hilfsfunktion

$$h(\vec{x}) := \sum_{i=0}^n (e_i \cdot \chi_{R_i}(\vec{x})) + e_0 \cdot \overline{\text{sg}} \sum_{i=0}^n \chi_{R_i}(\vec{x})$$

definiert. Damit gilt

$$h(\vec{x}) = \begin{cases} e_1, & \text{falls } R_1(\vec{x}), \\ \vdots \\ e_n, & \text{falls } R_n(\vec{x}), \\ e_0, & \text{sonst,} \end{cases}$$

und man benutzt nun die universelle Funktion um

$$f(\vec{x}) : \simeq \Phi^n(h(\vec{x}), \vec{x})$$

zu definieren. Offenbar genügt  $f$  den Bedingungen in (3.31).

Der Rekursionssatz in Verbindung mit der eben gesicherten Definition durch Fallunterscheidung erlaubt es uns, weitere Eigenschaften geeigneter Indizierungen der partiellrekursiven Funktionen zu studieren.

### 3.10. Universelle Funktionen und der Rekursionssatz

Insbesondere interessiert uns hier die Frage, ob die Indizierung eindeutig gewählt werden kann, d.h. so gewählt werden kann, daß jede Funktion einen eindeutig bestimmten Index erhält.

Wir hatten die Indizes partiellrekursiver Funktionen als Kodes von Funktionstermen eingeführt und bereits damals betont, daß jeder Funktionsterm einen Algorithmus darstellt. Gewinnt man Indizierungen über Kodierungen von Maschinenprogrammen, so ist man wieder in der Situation, daß der Index einer Funktion  $f$  einen Algorithmus für  $f$  kodiert. Allgemein empfiehlt es sich, bei Indizes, die über eine geeignete Indizierung gewonnen wurden, immer an Kodes für Algorithmen zu denken.

Gehen wir von dieser Anschauung aus, so erscheint es natürlich völlig unplaublich, eine eineindeutige Korrespondenz zwischen Funktionen und deren Indizes annehmen zu wollen. Wir hatten uns ja bereits anlässlich der Definition der primitiv rekursiven und der partiellrekursiven Funktionen klargemacht, daß völlig unterschiedliche Funktionsterme die (extensional betrachtet) gleiche Funktion berechnen.

Wenn sich also eine eineindeutige Korrespondenz zwischen Funktionen und deren Indizes nicht herstellen läßt, so ist die dann nächstliegende weitere Frage, ob es sich denn wenigstens entscheiden ließe, ob  $e$  Index einer Funktion  $f$  ist.

Ist die Zuordnung eines Index zu einer Funktion entscheidbar?

Wir wollen diese Fragestellung noch etwas allgemeiner anpacken und gehen dazu von einer Menge  $\mathcal{F}$   $n$ -stelliger partiellrekursiver Funktionen aus, von der wir annehmen wollen, daß sie weder leer ist, noch alle  $n$ -stelligen partiellrekursiven Funktionen umfaßt. Zur Abkürzung setzen wir

$$M := \{e \mid \{e\}^n \in \mathcal{F}\}.$$

Wir wollen nun untersuchen, ob  $M$  rekursiv ist. Dazu wählen wir partiellrekursive Funktionen  $f_J \in \mathcal{F}$  und  $f_N \notin \mathcal{F}$ . Nach den Voraussetzungen, die wir über  $\mathcal{F}$  gemacht haben, ist diese Wahl möglich. Wenn wir annehmen, daß  $M$  rekursiv ist, so erhalten wir eine partiellrekursive Funktion  $f$  durch die Fallunterscheidung

$$f(e, \vec{x}) : \simeq \begin{cases} f_J(\vec{x}), & \text{falls } e \notin M, \\ f_N(\vec{x}), & \text{falls } e \in M. \end{cases}$$

Nach dem Rekursionssatz existiert nun ein Index  $e$  einer partiellrekursiven Funktion mit

$$\{e\}^n(\vec{x}) \simeq f(e, \vec{x}).$$

Nehmen wir nun  $e \in M$  an, so folgt

$$\{e\}^n(\vec{x}) \simeq f(e, \vec{x}) \simeq f_N(\vec{x})$$

### 3. $\mu$ -rekursive Funktionen

---

für alle  $\vec{x}$ . D.h.  $\{e\}^n = f_N$ , was unmöglich ist, da wir ja  $f_N \notin \mathcal{F}$  und  $e \in M$  haben. Die Annahme  $e \notin M$  führt jedoch in analoger Weise zu dem Widerspruch  $\{e\}^n = f_J$ . Demnach kann  $M$  nicht rekursiv gewesen sein.

Die eben nachgewiesene Eigenschaft geeigneter Indizierungen ist als der Satz von RICE bekannt, den wir nochmals hervorheben möchten.

Der Satz von RICE

**Satz 33** (*Satz von RICE*) Ist  $\mathcal{F}$  eine nicht leere Menge  $n$ -stelliger partiellrekursiver Funktionen, die aber auch nicht alle  $n$ -stelligen Funktionen enthält, so ist die Menge aller Indizes von Funktionen in  $\mathcal{F}$  nicht rekursiv.

Man soll übrigens die Forderung der  $n$ -Stelligkeit im Satz von RICE nicht überbewerten. Durch Hinzufügen leerer Argumentstellen oder Diagonalisieren lässt sich die Stelligkeit partiellrekursiver Funktionen leicht manipulieren.

Eine Konsequenz des Satzes von RICE ist die Antwort auf unsere vorherigen Fragen. Haben wir nämlich eine partiellrekursive Funktion  $f$ , so können wir die Menge  $\mathcal{F}$  im Satz von RICE als das Singleton  $\{f\}$  wählen. Dies erfüllt alle Voraussetzungen, und wir erhalten daher als Folgerung den Satz:

**Satz 34** Bei jeder geeigneten Indizierung ist die Menge der Indizes einer partiellrekursiven Funktion nicht rekursiv. Damit muß in einer geeigneten Indizierung jede partiellrekursive Funktion unendlich viele verschiedene Indizes erhalten.

Konsequenzen des Satzes von RICE für die generelle Programmverifikation

Dieser Satz hat Konsequenzen für das Problem der automatischen Verifikation von Programmen. Wenn wir davon ausgehen, daß eine Funktion  $f$  durch eine Spezifikation gegeben ist, so beschreibt diese (zumindest im Idealfall) den Graphen von  $f$ . Jedes Programm zur Berechnung von  $f$  läßt sich natürlich kodieren (so ist beispielsweise sein Maschinenkode eine derartige Kodierung), und es ist sehr leicht einzusehen, daß diese Art von Kodierung immer geeignet sein wird. Ist nun  $e$  der Kode eines Programmes, so besteht die Aufgabe der Verifikation darin, festzustellen, ob  $e$  ein Index für  $f$  ist. Wie wir gesehen haben, ist dies ein nicht rekursives und damit nach der CHURCHSchen These algorithmisch unentscheidbares Problem. Also kann es kein Programm geben, daß eine derartige Frage global und generell entscheidet.

Damit ist natürlich nicht gesagt, daß Programmverifikation in jedem Fall unmöglich ist. Nur die globale Fragestellung kann algorithmisch nicht entschieden werden. Allerdings muß man sich bei allen Verifikationsproblemen diese prinzipielle Schwierigkeit vor Augen halten und darf nicht zu generelle Lösungen anstreben.

Eine weitere unmittelbare Folgerung aus dem Satz von RICE ist die

### 3.10. Universelle Funktionen und der Rekursionssatz

Tatsache, daß es sich nicht entscheiden läßt, ob ein Index  $e$  der Index einer totalen Funktion – und somit rekursiven Funktion – ist. Dies liegt einfach daran, daß es rekursive Funktionen gibt (beispielsweise die primiv rekursiven), wir anderseits aber auch eingesehen haben, daß nicht jede partiellrekursive Funktion auch rekursiv ist (man erinnere sich an das Beispiel der Funktion mit leerem Definitionsbereich im Anschluß an den Rekursionssatz). Damit sind die rekursiven Funktionen eine nicht leere Menge partiellrekursiver Funktionen, die aber auch nicht alle partiellrekursiven Funktionen enthält. Dies bedeutet aber, daß die Menge der Indizes rekursiver Funktionen nicht rekursiv sein kann.

Wir wollen hier aber eine noch weitergehende Frage untersuchen, nämlich ob wir die Menge der Indizes rekursiver Funktionen wenigstens rekursiv aufzählen können. Eine positive Antwort auf diese Frage hätte tatsächlich Konsequenzen für die Entwicklung der Theorie. Die Existenz einer Aufzählungsfunktion für die Indizes würde ja bedeuten, daß ein Kalkül existiert, der die Funktionsterme für rekursive Funktionen erzeugt, ähnlich wie dies für die partiellrekursiven der Fall ist. Das bedeutete aber, daß zur Entwicklung der Theorie der doch unbequeme Weg der Betrachtung partieller Funktionen gar nicht nötig gewesen wäre.

Um die Frage nach der rekursiven Aufzählbarkeit der Indizes rekursiver Funktionen zu klären, nehmen wir an, wir hätten eine rekursive Funktion  $h$ , die die Indizes aller 1-stelligen rekursiven Funktionen aufzählt und definieren eine partiellrekursive Funktion  $g$  vermöge

$$g(e, x) := \{h(e)\}^1(x).$$

Die Tatsache, daß  $\{h(e)\}^1$  für alle natürlichen Zahlen  $e$  eine totale Funktion ist, impliziert nun, daß auch die Funktion  $g$  für alle  $e$  und  $x$  definiert ist. Damit ist aber

$$f(x) := g(x, x) + 1$$

eine 1-stellige rekursive Funktion. Also liegt jeder Index  $e_0$  von  $f$  im Bild von  $h$ , d.h. es gibt eine natürliche Zahl  $y$  mit

$$e_0 = h(y).$$

Dies führt jedoch zu dem Widerspruch

$$f(y) = g(y, y) + 1 = \{h(y)\}^1(y) + 1 = \{e_0\}^1(y) + 1 = f(y) + 1.$$

Damit haben wir den folgenden Satz:

**Satz 35** *Die Menge der Indizes rekursiver Funktionen ist nicht rekursiv aufzählbar.*

Wir hatten bereits eine Menge kennengelernt, die rekursiv aufzählbar, aber nicht

Unentscheidbarkeit  
der Totalität  
partiellrekursiver  
Funktionen

### 3. $\mu$ -rekursive Funktionen

---

rekursiv ist und daraus geschlossen, daß die  $\Delta_1$ -Mengen, die ja genau die rekursiven sind, eine echte Teilmenge der  $\Sigma_1$ -Mengen bilden, die genau die rekursiv aufzählbaren sind. Wir haben nun ein Beispiel einer Menge, nämlich die der Indizes rekursiver Funktionen, von der wir wissen, daß sie nicht rekursiv aufzählbar ist und damit keine  $\Sigma_1$ -Menge ist. Ihre Komplexität wird sichtbar, wenn wir die Formel aufschreiben, die aussagt, daß “ $e$  Index einer 1-stelligen rekursiven Funktion ist”. Es gilt:

$$\text{Tot}(e) \iff (\forall \vec{x})(\exists y)T^1(e, \vec{x}, y).$$

Damit stellt sich  $\text{Tot}$  als eine  $\Pi_2$ -Menge heraus. Es läßt sich darüber hinaus noch zeigen, daß  $\text{Tot}$  keine  $\Pi_1$ -Menge sein kann, so daß wir eine echte  $\Pi_2$ -Menge vorliegen haben.

---

## 4. $\lambda$ -definierbare Funktionen

Im letzten Kapitel hatten wir die Theorie der Berechenbarkeit auf der Basis der klassischen, extensionalen Auffassung des Funktionsbegriffes entwickelt. Wie uns der Satz von RICE gelehrt hat, hat dies zur Folge, daß dann keine entscheidbare Korrespondenz zwischen Funktionen und deren Indizes existieren kann. Bei Zugrundelegung dieser Auffassung erscheinen Funktionen und deren Argumente – also natürliche Zahlen – als Objekte unterschiedlichen Typs, was in diesem Zusammenhang nur soviel heißen soll, daß Funktionen nicht auf Funktionen und natürliche Zahlen weder auf Funktionen noch auf natürliche Zahlen anwendbar sind.

Fassen wir den Begriff der partiellrekursiven Funktion jedoch intentional auf, was hier bedeuten soll, daß wir die Funktionsterme (oder äquivalent dazu die Algorithmen) als die eigentlichen Objekte betrachten, so haben wir eine eindeutige Korrespondenz zwischen unseren Objekten und ihren Indizes. Jedem Funktionsterm  $F$  entspricht eindeutig seine Gödelnummer  $[F]$  und dies versetzt uns in die Lage, einen Funktionsterm  $F$  auf einen Funktionsterm  $G$  vermöge der Vereinbarung

$$F(G) : \simeq F([G])$$

anzuwenden. Dies war genau der Trick, mit dem wir uns im Rekursionssatz beholfen haben. Darüber hinaus können wir eine Zahl  $e$  sowohl auf einen Funktionsterm  $G$  als auch auf eine Zahl  $x$  vermöge der Vereinbarungen

$$e(G) : \simeq \Phi^1(e, [G]) \quad \text{bzw.} \quad e(x) : \simeq \Phi^1(e, x)$$

anwenden. Dann können wir aber auf eine Typenunterscheidung zwischen Funktionstermen und ihren Argumenten verzichten.

Diese ‘Selbstanwendung’ hat bei dem Nachweis der Existenz nicht rekursiver, aber rekursiv aufzählbarer Mengen eine wesentliche Rolle gespielt. Man beachte, daß das Halteproblem für Turingmaschinen ein typisches Selbstanwendungsproblem ist. Dort haben wir nach der Definiertheit von  $\Phi^1(e, x)$  gefragt, also nach der Definiertheit von  $e(x)$  in der eben eingeführten Terminologie.

Die Ununterscheidbarkeit der Typen von Funktionen und Argumenten spiegelt auch die Situation in einem realen Rechner wider. Ein solcher ist als eine (physikalische Approximation einer) universelle(n) Maschine anzusehen. In dieser wird aber nicht zwischen Programmen und Daten unterschieden. Beide erscheinen völlig gleichwertig als Inhalte des Datenspeichers.

## 4. $\lambda$ -definierbare Funktionen

---

In diesem Kapitel wollen wir einen Zugang zur Theorie der Berechenbarkeit studieren, in dem nicht zwischen Argumenten und Funktionen unterschieden wird, sondern in dem beide Objekte einfach als Daten behandelt werden, die in elementarer Weise manipulierbar sind.

Dieser Zugang ist als *kombinatorische Logik* bekannt und wurde zuerst von ALONZO CHURCH betrachtet. Neben ihm war HASKEL CURRY einer der Pioniere der kombinatorischen Logik. Die kombinatorische Logik – insbesondere in ihrer Formalisierung durch den  $\lambda$ -Kalkül – spielt eine wesentliche Rolle bei der Entwicklung der funktionalen Programmiersprachen, insbesondere der Sprachenfamilie LISP.

### 4.1 Der $\lambda$ -Kalkül

Der  $\lambda$ -Kalkül beabsichtigt eine Formalisierung der endlichen Kombinatorik. Endliche Kombinatorik kann als Manipulation endlicher Zeichenreihen aufgefaßt werden. Dabei wird im  $\lambda$ -Kalkül das Einsetzen einer Zeichenreihe in eine andere als die Grundmanipulation angesehen. Versucht man dies zu formalisieren, so hat man in einer Zeichenreihe

$$z_1 \dots z_n$$

eine Stelle zu markieren. Dies wollen wir dadurch symbolisieren, daß wir die Adressmarke  $x$  an diese Stelle schreiben. Dies könnte in der Form

$$z_1 \dots x \dots z_n$$

geschehen. Wenden wir nun die so markierte Zeichenreihe auf eine schon gebildete Zeichenreihe  $u$  an, so erhalten wir die Konversion

$$(z_1 \dots x \dots z_n)u \rightarrow (z_1 \dots u \dots z_n).$$

Schwieriger wird dies jedoch, wenn wir mehrere Stellen in einer Zeichenreihe markieren wollen. Bilden wir

$$z_1 \dots x \dots y \dots z_n,$$

so ist bei späteren Konversionen nicht mehr ohne weiteres zu erkennen, in welcher Reihenfolge die Adressen zu ersetzen sind. Dies ließe sich natürlich über die generelle Vereinbarung regeln, daß zuerst  $x$ , dann  $y$  u.s.f. zu ersetzen ist, doch das würde unseren Formalismus so einengen, daß er nicht mehr handhabbar sein würde. Den Ausweg bietet die CHURCHsche  $\lambda$ -Notation. Anstatt die Adresse einfach durch ein  $x$  zu markieren, schreiben wir den markierten Term als

$$\lambda x. z_1 \dots x \dots z_n,$$

wobei das Prefix  $\lambda x$  nichts weiter bedeutet, als daß  $x$  eine Adresse in der Zeichenreihe  $z_1 \dots x \dots z_n$  ist, die dem Prefix nachfolgt. Wollen wir nun mehrere Adressen in einer Zeichenreihe markieren, so gelingt dies, indem wir

$$\lambda x_1 \dots \lambda x_m . Z \quad \text{oder kurz} \quad \lambda x_1 \dots x_m . Z$$

schreiben. Sind nun  $u_1, \dots, u_m$  weitere Zeichenreihen, so erhalten wir die Konversion einfach zu

$$(\lambda x_1 \dots x_m . Z) u_1 \dots u_m \rightarrow Z_{x_1, \dots, x_m}(u_1, \dots, u_m),$$

wobei der Ausdruck auf der rechten Seite bedeutet, daß in  $Z$  zuerst  $x_1$  durch  $u_1$ , dann  $x_2$  durch  $u_2$  usf. ersetzt wird, bis wir schließlich  $x_m$  durch  $u_m$  ersetzt haben. Die Adressen in den Zeichenreihen haben hier die Funktion von Platzhaltern für einzusetzende Zeichenreihen, weshalb wir sie auch als Variablen bezeichnen werden.

Entsprechend unserer heuristischen Überlegung sollten wir in der Lage sein, eine Theorie der Kombinationen unter alleiniger Ausnutzung des  $\lambda$ -Abstraktors und der Konversionsrelation zu entwickeln. Die formalen Objekte dieser Theorie werden allgemein als  $\lambda$ -Terme bezeichnet. Zur Bildung der  $\lambda$ -Terme benötigen wir nach unseren Vorüberlegungen offensichtlich nur Variablen und den  $\lambda$ -Abstraktor.

### **Definition 36 (Induktive Definition der $\lambda$ -Terme)**

1. Jede Variable  $x, y, z, \dots$  ist ein  $\lambda$ -Term.  $\lambda$ -Terme
2. Ist  $Z$  ein  $\lambda$ -Term und ist  $x$  eine Variable, so ist  $(\lambda x . Z)$  ein  $\lambda$ -Term.
3. Sind  $U$  und  $V$   $\lambda$ -Terme, so ist auch  $(UV)$  ein  $\lambda$ -Term.

Wir vereinbaren eine Klammerung von  $\lambda$ -Termen nach links. Um Klammern zu sparen, lassen wir überflüssige Klammern weg und schreiben generell

$$\lambda x_1 \dots x_n . Z \quad \text{anstatt} \quad \lambda x_1 . (\lambda x_2 . (\dots (\lambda x_n . Z) \dots)).$$

Betrachten wir einige Beispiele. Ein einfacher  $\lambda$ -Term ist der Term

$$((xz)(yz)), \quad d.h. \quad xz(yz)$$

unter Ausnutzung der Klammerersparnisregeln. Hier treten die Variablen  $x, y$  und  $z$  frei auf. Bilden wir nun den Term

$$(\lambda x . (\lambda y . (\lambda z . ((xz)(yz))))), \quad \text{also} \quad \lambda xyz . xz(yz)$$

nach Anwendung der Klammerersparnisregeln, so sind aus den Variablen Adressen – oder wie wir sie in Zukunft nennen wollen – gebundene Variablen geworden. Offenbar spielt es dabei keine Rolle, welchen Namen

#### 4. $\lambda$ -definierbare Funktionen

---

wir der gebundenen Variablen geben. Die Terme

$$\lambda xyz . \, xz(yz)$$

und

$$\lambda uyz . \, uz(yz)$$

sind absolut gleichwertig, da in beiden die gleichen Adressen markiert sind. Wir wollen daher  $\lambda$ -Terme, die sich nur durch die Namen der in ihnen gebundenen Variablen unterscheiden, identifizieren. Dabei wollen wir so großzügig sein, daß es uns nur auf die Verständlichkeit eines Termes ankommt. So wollen wir beispielsweise den Term

$$(\lambda x . \, xx)(\lambda x . \, xx)$$

als wohlgeformten  $\lambda$ -Term betrachten. Ebenso ist es tolerabel, Terme der Form

$$\lambda xy . \, x(\lambda x . \, xy)$$

zu betrachten, da klar ist, welche Adressen durch das erste und welche durch das zweite  $\lambda x$  markiert werden.

Wir haben noch die formale Definition der Konversion nachzutragen:

**Definition 37** ( $\beta$ -Konversion)

$\beta$ -Konversion

1.  $(\lambda x . \, Z)U \rightarrow_1 Z_x(U)$ .
2. Gilt  $U \rightarrow_1 X$ , so auch  $\lambda x . \, U \rightarrow_1 \lambda x . \, X$ .
3. Gilt  $U \rightarrow_1 X$ , so gelten sowohl  $UV \rightarrow_1 XY$  als auch  $VU \rightarrow_1 YX$ .

Klauseln 2 und 3 in der obigen Definition sagen dabei nur aus, daß eine Konversion an einer beliebigen Stelle innerhalb eines  $\lambda$ -Terms ausgeführt werden darf.

Reduktionen

Unter einer *Reduktion* verstehen wir eine Kette aufeinanderfolgender Konversionen. Formal definieren wir zunächst

1.  $U \rightarrow_1^0 V$ , falls  $U = V$
2.  $U \rightarrow_{n+1} V$ , falls es ein  $W$  gibt mit  $U \rightarrow_n W$  und  $W \rightarrow_1 V$ .

Der Ausdruck  $U \rightarrow_n V$  besagt also, daß  $V$  durch eine Kette von  $n$  aufeinanderfolgenden Konversionen aus  $U$  hervorgeht.

Die *Reduktionsrelation*  $U \rightarrow V$  definieren wir dann durch

$$U \rightarrow V : \iff (\exists n)[U \rightarrow_n V].$$

Die Reduktionsrelation  $\rightarrow$  ist somit die *reflexive* und *transitive Hülle* der Konversionsrelation  $\rightarrow_1$ , was insbesondere bedeutet, daß

$$U \rightarrow U$$

und

$$U \rightarrow V, V \rightarrow W \implies U \rightarrow W$$

gelten.

Man überzeugt sich sofort, daß wir die Klammerersparnisregeln so eingerichtet haben, daß

$$(\lambda x_1 \dots x_n . Z) u_1 \dots u_n \rightarrow Z_{x_1, \dots, x_n}(u_1, \dots, u_n)$$

gilt.

Es erscheint vielleicht etwas verwunderlich, daß wir keine Atome eingeführt haben, aus denen die Zeichenreihen des  $\lambda$ -Kalküls zusammengesetzt werden sollen. Variablen sind keine Atome in diesem Sinne, da sie lediglich als Platzhalter für weitere Zeichenreihen anzusehen und somit interpretationsbedürftig sind. Es hat sich einfach herausgestellt, daß sich die allgemeine Theorie auch ohne Zuhilfenahme von Atomen entwickeln läßt. Die Grundidee dabei ist, daß ein Term ohne freie Variablen – z.B.  $\lambda x . x$  – eine wohldefinierte kombinatorische Bedeutung hat. So repräsentiert das Beispiel  $\lambda x . x$  die Identität, d.h. es gilt

$$(\lambda x . x) U \rightarrow U$$

für jede Zeichenreihe  $U$ , und für das schon früher erwähnte Beispiel

$$\lambda xyz . xz(yz)$$

haben wir

$$(\lambda xyz . xz(yz))UVW \rightarrow UW(VW)$$

für alle Zeichenreihen  $U, V, W$ . Wir nennen  $\lambda$ -Terme ohne freie Variablen *geschlossene  $\lambda$ -Terme* oder *Kombinationen*.

Kombinationen

Eine alternative Entwicklung der kombinatorischen Logik beruht auf Grundkombinatoren anstelle von  $\lambda$ -Abstraktionen. Hier geht man von Variablen

Eine alternative Entwicklung der kombinatorischen Logik

$$x, y, \dots$$

und den *Grundkombinatoren*

$$K \text{ und } S$$

aus, die durch die Konversionen

$$Kxy \rightarrow_1 x$$

und

$$Sxyz \rightarrow_1 xz(yz)$$

## 4. $\lambda$ -definierbare Funktionen

---

definiert sind und bildet Terme nur durch Applikation. Der Kombinatorenkalkül ist dann im  $\lambda$ -Kalkül darstellbar, da die Grundkombinatoren durch

$$K := \lambda xy. x \quad \text{und} \quad S := \lambda xyz. xz(yz)$$

definiert werden können. Umgekehrt lässt sich die  $\lambda$ -Abstraktion im Kombinatorenkalkül vermöge

$$\lambda x. M := \begin{cases} KM, & \text{falls } x \text{ nicht in } M \text{ auftritt,} \\ SKK, & \text{falls } M = x, \\ S(\lambda x. U)(\lambda x. V), & \text{falls } x \text{ in } M \text{ auftritt und } M = UV \text{ ist,} \end{cases}$$

induktiv definieren.

Tritt  $x$  nicht in  $M$  auf, so ist

$$(\lambda x. M)N = KMN \rightarrow_1 M = M_x(N).$$

Andernfalls zeigt man

$$(\lambda x. M)N \rightarrow M_x(N),$$

durch Induktion nach dem Aufbau von  $M$ , denn für  $M = x$  gilt

$$(\lambda x. x)N = SKKN \rightarrow_1 KN(KN) \rightarrow_1 N = x_x(N)$$

und für  $M = UV$

$$\begin{aligned} (\lambda x. UV)N &= S(\lambda x. U)(\lambda x. V)N \rightarrow_1 (\lambda x. U)N((\lambda x. V)N) \rightarrow (U_x(N))(V_x(N)) \\ &= M_x(N), \end{aligned}$$

wobei wir im letzten Fall die Induktionsvoraussetzung benutzt haben.

## 4.2 $\beta$ -Gleichheit von $\lambda$ -Termen

Normalformen für  
 $\lambda$ -Terme

In den obigen Beispielen haben wir bereits anklingen lassen, daß wir Reduktionen irgendwie als eine Gleichheitsrelation betrachten wollen. Die intuitive Idee dahinter ist, daß eine Konversion einen Rechenschritt einer kombinatorischen Berechnung repräsentiert. Wir sagen, daß ein  $\lambda$ -Term in *Normalform* ist, wenn er sich nicht weiter reduzieren läßt. Offenbar ist ein  $\lambda$ -Term genau dann in Normalform, wenn er kein *Redex* enthält, d.h. keinen Teilterm der Gestalt  $\dots (\lambda x. Z)U \dots$

Eine Reduktionskette

$$U \rightarrow_1 U_1 \rightarrow_1 \dots \rightarrow_1 U_n \rightarrow_1 \dots$$

repräsentiert also eine Berechnung des Termes  $U$ . Wir sagen, daß diese Rechnung terminiert, wenn die Reduktionskette abbricht, d.h. wenn es eine Normalform  $V$  gibt, mit

$$U \rightarrow_n V.$$

In diesem Fall nennen wir  $V$  eine Normalform von  $U$ .

Wir möchten nun gerne zwei Terme als gleich betrachten, wenn ihre Berechnungen den gleichen Wert liefern, d.h. wenn sie eine gemeinsame Normalform besitzen. Betrachten wir jedoch den Term

$$\Omega := (\lambda x. xx)(\lambda x. xx),$$

so sehen wir, daß wir eine unendliche Reduktionskette

$$(\lambda x. xx)(\lambda x. xx) \rightarrow_1 (\lambda x. xx)(\lambda x. xx) \rightarrow_1 \Omega \rightarrow_1 \dots \rightarrow_1 \Omega \rightarrow_1 \dots$$

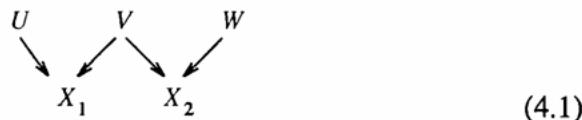
Ein Term mit  
unendlicher  
Reduktionskette

erhalten. Also besitzt nicht jeder Term eine Normalform und die Gleichheit wäre nicht für alle Terme definiert. Man behilft sich, indem man

$$U =_{\beta} V: \iff \text{es gibt einen } \lambda\text{-Term } W \text{ mit } U \rightarrow W \text{ und } V \rightarrow W$$

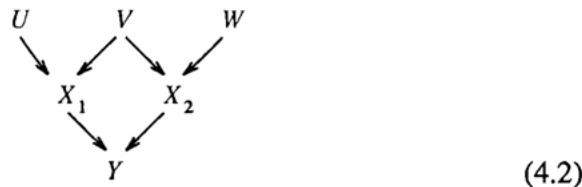
definiert. Wir haben diese Gleichheit zur Unterscheidung von der Gestaltsgleichheit von Termen, die wir bisher durch  $=$  ausgedrückt haben, mit  $=_{\beta}$  bezeichnet. Das  $\beta$  soll andeuten, daß sie von der  $\beta$ -Konversion erzeugt wird.

Die so definierte  $\beta$ -Gleichheit ist ihrer Definition nach reflexiv und symmetrisch. Sehr viel schwieriger ist die Frage nach ihrer Transitivität zu beantworten. Gilt  $U =_{\beta} V$  und  $V =_{\beta} W$ , so haben wir Terme  $X_1$  und  $X_2$  mit



(4.1)

Die Frage, die sich nun stellt, ist, ob wir einen Term  $Y$  so finden können, daß wir



(4.2)

erhalten. Man hat also zu zeigen, daß zu Termen  $V$ ,  $X_1$  und  $X_2$  mit  $V \rightarrow X_1$  und  $V \rightarrow X_2$  ein Term  $Y$  existiert mit



(4.3)

#### 4. $\lambda$ -definierbare Funktionen

---

Obwohl dies richtig ist, ist diese Tatsache keinesfalls einfach einzusehen. Der Satz, der die Existenz von  $Y$  in (4.3) garantiert, ist als CHURCH-ROSSER Theorem bekannt und galt lange Zeit als einer der am aufwendigsten zu beweisenden Sätze der Theorie des  $\lambda$ -Kalküls. Wir wären fertig, wenn wir (4.3) für Konversionen hätten, das hieße



Denn dann dürften wir  $V \rightarrow_n X_1$  und  $V \rightarrow_m X_2$  annehmen und könnten uns, wie in Figur 4.2 - 1 dargestellt, vermöge des Maschenlemma ein  $Y$  erhäkeln. Dummerweise gilt das Diamantenlemma, wie es in (4.4)

---

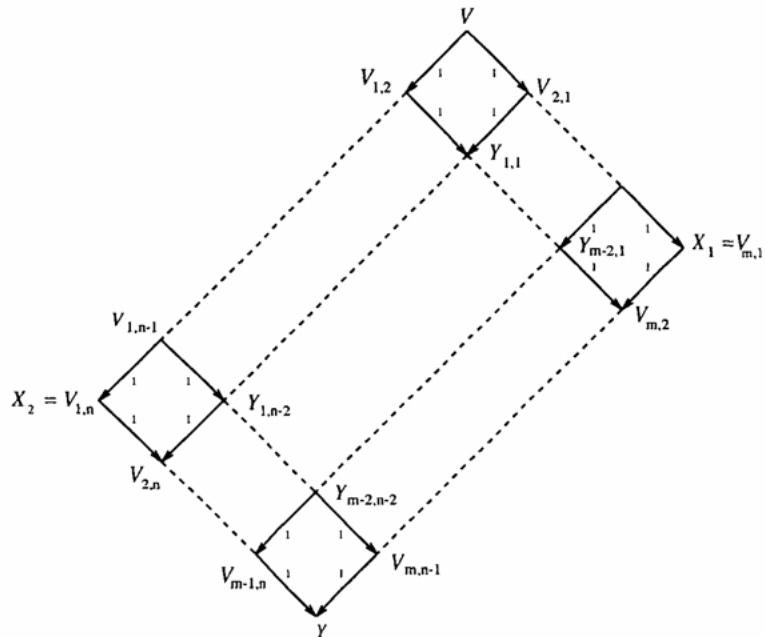


Abb. 4.2 - 1 : Das *Maschenlemma* in graphischer Darstellung. Es zeigt, wie man, die Gültigkeit des Diamantenlemmas vorausgesetzt, sich aus  $V \rightarrow_m X_1$  und  $V \rightarrow_n X_2$  ein  $Y$  mit  $X_1 \rightarrow_n Y$  und  $X_2 \rightarrow_m Y$  erhäkeln kann.

---

dargestellt ist, für die  $\beta$ -Konversion nicht. Das sieht man an den folgenden Beispielen: Betrachten wir den Term

$$M := (\lambda x. xy)(\lambda x. x)\Omega,$$

wobei  $\Omega$  wie eben definiert wurde, so haben wir einmal  $\Omega$  als Redex und somit

$$M \rightarrow_1 M,$$

aber auch  $(\lambda x. xy)(\lambda x. x)$  als Redex, was zu

$$M \rightarrow_1 \lambda x. x$$

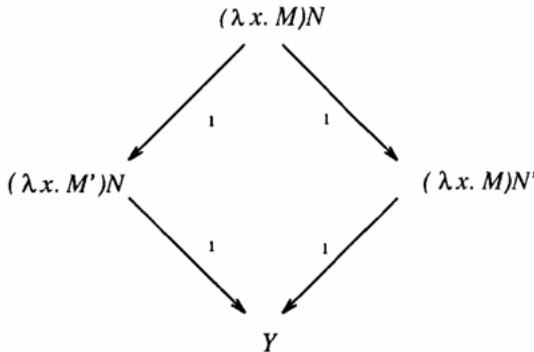
führt. Also gilt



und der Diamant kann nicht geschlossen werden, da offenbar

$$\lambda x. x \rightarrow_1 \lambda x. x$$

nicht gilt. Allerdings sieht man an diesem Beispiel bereits den ersten Schritt, den man zur Rettung der Situation leisten kann. Hätte man die triviale Konversion  $\lambda x. x \rightarrow_1 \lambda x. x$ , so könnte man den Diamanten in (4.5) schließen. Allerdings wäre damit noch nicht alles geheilt, denn nehmen wir  $M \rightarrow_1 M'$  und  $N \rightarrow_1 N'$  an, so finden wir offensichtlich kein  $Y$  mit



Zwar wäre der Term  $Y = M'_x(N')$  ein geeigneter Kandidat, doch wir haben weder  $(\lambda x. M')N \rightarrow_1 M'_x(N')$  noch  $(\lambda x. M)N' \rightarrow_1 M'_x(N')$ . Was wir hier benötigten, wäre ein *parallel* ablaufendes Berechnungsverfahren, das es uns erlaubt  $M'_x(N')$  sowohl aus  $(\lambda x. M')N$ , als auch aus  $(\lambda x. M)N'$  in einem Schritt zu berechnen. Diese Überlegungen führten

## 4. $\lambda$ -definierbare Funktionen

---

WILLIAM TAIT zu der Idee, eine neue Konversionsregel  $\rightarrow_1^*$  einzuführen, die sowohl triviale als auch parallele Rechnungen in einem Schritt erlaubt. Die neue Konversionsregel  $\rightarrow_1^*$  wird durch die folgenden Klauseln definiert:

1.  $M \rightarrow_1^* M$
2. Gelten  $Z \rightarrow_1^* Z'$  und  $U \rightarrow_1^* U'$ , so folgt  $(\lambda x . Z)U \rightarrow_1^* Z'_x(U')$ .
3. Gilt  $U \rightarrow_1^* X$ , so auch  $\lambda x . U \rightarrow_1^* \lambda x . X$ .
4. Gilt  $U \rightarrow_1^* X$  und  $V \rightarrow_1^* Y$ , so gilt  $UV \rightarrow_1^* XY$ .

Bezeichnen wir mit  $\rightarrow^*$  wieder die reflexive und transitive Hülle von  $\rightarrow_1^*$ , so beobachtet man sehr einfach, daß

$$U \rightarrow V \iff U \rightarrow^* V \quad (4.6)$$

gilt. Darüber hinaus gilt das Diamantenlemma für  $\rightarrow_1^*$ . D.h., wir haben

**Satz 38 (Diamantenlemma)** Gilt  $U \rightarrow_1^* V$  und  $U \rightarrow_1^* W$ , so gibt es einen Term  $Y$  mit  $V \rightarrow_1^* Y$  und  $W \rightarrow_1^* Y$ .

Das Diamantenlemma zieht aber, wie in Abbildung 4.2 - 1 skizziert, das CHURCH-ROSSER-Theorem für die  $\rightarrow^*$ -Relation nach sich. Wegen (4.6) erhalten wir damit

**Satz 39 (CHURCH-ROSSER Theorem)** Sind  $M$ ,  $U$  und  $V$   $\lambda$ -Terme, mit  $M \rightarrow U$  und  $M \rightarrow V$ , so gibt es einen  $\lambda$ -Term  $Y$  mit  $U \rightarrow Y$  und  $V \rightarrow Y$ .

Als unmittelbares Korollar zum CHURCH-ROSSER Theorem folgt die Eindeutigkeit der Normalform eines  $\lambda$ -Terms. Damit haben wir

**Satz 40** Jeder  $\lambda$ -Term besitzt höchstens eine Normalform.

Wie wir bereits bemerkt hatten, ist die Transitivität von  $=_\beta$  eine unmittelbare Folgerung des CHURCH-ROSSER Theorems. Damit haben wir den Satz, der das Ziel unserer Überlegungen war.

**Satz 41** Die  $\beta$ -Gleichheit  $=_\beta$  ist eine Äquivalenzrelation auf den  $\lambda$ -Termen.

### 4.3 Rekursive Aufzählbarkeit der $\beta$ -Gleichheit

Die Reduktion von  $\lambda$ -Termen stellt offenbar ein effektives Verfahren dar. Es sollte daher ein weiterer Test für die CHURCHsche These sein, nachzuweisen, daß es sich hierbei um ein rekursives Verfahren handelt. Da wir die Theorie der rekursiven Mengen und Funktionen jedoch nur auf den natürlichen Zahlen entwickelt haben, müssen wir dazu die Sprache des  $\lambda$ -Kalküls in die der natürlichen Zahlen übertragen. Dies gelingt mit Hilfe einer Arithmetisierung. Dazu gehen wir von einer Aufzählung

Das  
Diamantenlemma

Das  
CHURCH-ROSSER  
Theorem

Eindeutigkeit der  
Normalform

$x_0, x_1 \dots$

der Variablen des  $\lambda$ -Kalküls aus, die wir im folgenden als fest gewählt betrachten wollen. Wir definieren dann den Kode einer Variablen zu

$$x_i := \langle 0, i \rangle$$

und setzen dies vermöge der Klauseln

Arithmetisierung des  
 $\lambda$ -Kalküls

$$\begin{aligned} [M] &:= x, \text{ wenn } M \text{ die Variable } x \text{ ist,} \\ [\lambda x . M] &:= \langle 1, x, [M] \rangle, \\ [MN] &:= \langle 2, [M], [N] \rangle \end{aligned}$$

auf alle  $\lambda$ -Terme fort.

Es ist dann sehr leicht einzusehen, daß die Menge  $\Lambda$  der Kodes von  $\lambda$ -Termen wieder eine primitiv rekursive Menge ist. Durch Wertverlaufsrekursion läßt sich eine primitiv rekursive Funktion  $Sb$  angeben, für die

$$Sb(x, [M], [N]) = [M_x(N)]$$

gilt.

Formal definiert man  $Sb$  durch

$$Sb(x, y, z) := \begin{cases} y, & \text{falls } (y)_0 = 0 \text{ und } x \neq y, \\ z, & \text{falls } (y)_0 = 0 \text{ und } x = y, \\ \langle 1, (y)_1, Sb(x, (y)_2, z) \rangle, & \text{falls } y \in \Lambda, (x)_0 = 0 \text{ und } (y)_0 = 1, \\ \langle 2, Sb(x, (y)_1, z), Sb(x, (y)_2, z) \rangle, & \text{falls } y \in \Lambda, (x)_0 = 0 \text{ und } (y)_0 = 2, \\ 0, & \text{sonst.} \end{cases}$$

Es ist nicht so wesentlich, die genaue Definition von  $Sb$  zu verstehen. Wichtig ist nur, daß es *prinzipiell* möglich ist, die Substitution eines Termes an einer Adresse primitiv rekursiv zu beschreiben.

Wir wollen an dieser Stelle auch nicht verhehlen, daß wir uns um eine noch viel schwieriger zu lösende Problematik herumgedrückt haben, nämlich die Identifikation von Termen, die sich nur durch die Namen gebundener Variablen voneinander unterscheiden. Auch diese Identifikation kann in primitiv rekursiver Weise beschrieben werden. Eine Skizze des Vorgehens würde uns allerdings wenig Einsicht vermitteln. Deshalb haben wir hier darauf verzichtet.

Nun sind wir in der Lage, wieder durch Wertverlaufsrekursion ein primitiv rekursives Prädikat  $Kon$  zu definieren, für das

$$U \rightarrow_1 V \iff Kon([U], [V])$$

gilt. Formal definieren wir

$$\begin{aligned} Kon(x, y) :&\iff x \in \Lambda \wedge y \in \Lambda \\ &\wedge [((x)_0 = 2 \wedge (x)_{10} = 1 \wedge y = Sb((x)_{11}, (x)_{12}, (x)_2)) \\ &\vee ((x)_0 = (y)_0 = 1 \wedge (x)_1 = (y)_1 \wedge Kon((x)_2, (y)_2)) \\ &\vee ((x)_0 = (y)_0 = 2 \wedge Kon((x)_1, (y)_1) \wedge Kon((x)_2, (y)_2))]. \end{aligned}$$

## 4. $\lambda$ -definierbare Funktionen

---

Nach der Definition der Reduktionsrelation haben wir

$$U \rightarrow V \iff U = S_0 \rightarrow_1 S_1 \rightarrow_1 \dots \rightarrow_1 S_{n-1} = V,$$

wobei alle  $S_i$  für  $i = 0, \dots, n - 1$   $\lambda$ -Terme sind. In der Arithmetisierung ergibt dies ein Prädikat

$$\begin{aligned} \text{Red}(x, y) : \iff & (\exists s) \{ \text{Seq}(s) \wedge (s)_0 = x \wedge (s)_{lh(s-1)} = y \\ & \wedge (\forall i < lh(s-1)) [\text{Kon}((s)_i, (s)_{i+1})] \}. \end{aligned}$$

Der Ausdruck innerhalb der geschweiften Klammern ist dann unschwer als primitiv rekursiv zu erkennen. Der Existenzquantor zu Beginn macht dies somit zu einer rekursiv aufzählbaren Relation.

Zu guter Letzt können wir die  $\beta$ -Gleichheit durch das Prädikat

$$EQ_\beta(x, y) : \iff (\exists z) [\text{Red}(x, z) \wedge \text{Red}(y, z)]$$

ausdrücken. Da die rekursiv aufzählbaren Prädikate gegenüber Konjunktionen und Existenzquantifikation abgeschlossen sind, erhalten wir  $EQ_\beta$  als rekursiv aufzählbares Prädikat.

Rekursive  
Aufzählbarkeit der  
 $\beta$ -Gleichheit

Ein  $\lambda$ -Term der Gestalt  $\lambda x_1 \dots x_n . M$  kann als  $n$ -stellige Funktion aufgefaßt werden. Die Argumente dieser Funktion sind wieder  $\lambda$ -Terme. Dies ist die eingangs bereits betonte *Typenfreiheit* des  $\lambda$ -Kalküls. Wir können daher einen  $\lambda$ -Term als eine Rechenvorschrift – d.h. als ein Programm in einem sehr abstrakten Sinne – betrachten, die bereits alle von ihr benötigten Daten mit sich trägt. Die Berechnung ist erfolgreich, sobald eine Normalform erreicht wird. Als den ‘Graphen’ eines  $\lambda$ -Terms  $M$  läßt sich dann die Menge

$$\{(M, N) \mid M \rightarrow N \wedge N \text{ ist in Normalform}\} \tag{4.7}$$

betrachten. Da es sich rein syntaktisch durch Betrachten der Gestalt eines Termes feststellen läßt, ob er sich in Normalform befindet, können wir wieder leicht eine Arithmetisierung des Prädikates ‘ $N$  ist in Normalform’ definieren, und dies stellt sich völlig zwanglos als primitiv rekursiv heraus. Damit ist die Arithmetisierung der Menge in (4.7) durch

$$\{([M], [N]) \mid \text{Red}([M], [N]) \wedge N \text{ ist in Normalform}\}$$

gegeben. Dies ist eine rekursiv aufzählbare Menge. Wie wir in Abschnitt 3.9 gesehen haben, sind die Funktionen mit rekursiv aufzählbarem Graphen aber genau die partiellrekursiven Funktionen. Die Berechnung der Normalform eines  $\lambda$ -Terms hat damit also die Komplexität der Berechnung einer partiellrekursiven Funktion. Um diesen Zusammenhang genauer zu untersuchen, wollen wir im nächsten Abschnitt  $\lambda$ -definierbare Funktionen einführen.

## 4.4 $\lambda$ -definierbare Funktionen

Bei der Definition  $\lambda$ -definierbarer Funktionen, wobei wir uns wieder auf Funktionen von  $\mathbb{N}^n \rightarrow \mathbb{N}$  beschränken wollen, haben wir den umgekehrten Weg zu gehen. Im Gegensatz zum vorherigen Abschnitt geht es nun darum, natürliche Zahlen und auf ihnen operierende Funktionen als  $\lambda$ -Terme darzustellen.

Damit dies überhaupt gelingen kann, müssen wir uns zunächst davon überzeugen, daß der  $\lambda$ -Kalkül überhaupt Kodierungen erlaubt. Dies ist Kodierungsfunktionen im  $\lambda$ -Kalkül

### Definition 42

$$\Pi^n(N_1, \dots, N_n) := \lambda x. x N_1 \dots N_n$$

und

$$\Pi_j^n := \lambda x. x(\lambda x_1 \dots x_n. x_j),$$

und rechnen sofort nach, daß für  $1 \leq j \leq n$  dann

$$\Pi_j^n(\Pi^n(N_1, \dots, N_n)) =_\beta N_j$$

gilt. Damit liegen Kodierungsfunktionen vor.

Wir definieren nun die Fallunterscheidungsterme

Fallunterscheidungsterme

$$T := \lambda xy. x \quad \text{und} \quad F := \lambda xy. y,$$

die man oft auch als Wahrheitswerte ‘wahr’ und ‘falsch’ bezeichnet. Der Grund dafür liegt in der folgenden Beobachtung, die man einfach nachrechnet:

**Lemma 43** Ist  $B$  ein  $\lambda$ -Term, der einen der Werte  $T$  oder  $F$  annehmen kann, so gilt

$$BMN = \begin{cases} M, & \text{falls } B =_\beta T, \\ N, & \text{falls } B =_\beta F. \end{cases}$$

Somit steht uns die Möglichkeit der Definition durch Fallunterscheidung im  $\lambda$ -Kalkül zur Verfügung.

Man notiert dies oft in der Form

**IF  $B$  THEN  $M$  ELSE  $N$ .**

Nun können wir daran gehen, Kodes für natürliche Zahlen einzuführen. Wir definieren

Darstellung natürlicher Zahlen

#### 4. $\lambda$ -definierbare Funktionen

---

$$\underline{0} := \lambda x . x$$

und

$$\underline{S}(n) := \Pi^2(F, \underline{n}).$$

Diese im ersten Augenblick etwas willkürlich erscheinende Festsetzung wird durch die folgende Beobachtung motiviert.

**Lemma 44** *Definieren wir den  $\lambda$ -Term  $\text{NULL} := \lambda x . xT$ , so gilt*

$$\text{NULL } \underline{n} = \begin{cases} T, & \text{falls } n = 0 \\ F, & \text{falls } n \neq 0. \end{cases}$$

Dies ist ganz offenbar wegen

$$(\lambda x . xT)(\lambda x . x) =_{\beta} (\lambda x . x)T =_{\beta} T$$

und

$$(\lambda x . xT)\Pi^2(F, \underline{n}) =_{\beta} (\lambda x . xT)(\lambda y . yF \underline{n}) =_{\beta} (\lambda y . yF \underline{n})T =_{\beta} TF \underline{n} =_{\beta} F.$$

Nun können wir ausdrücken, was wir unter einer  $\lambda$ -definierbaren Funktion verstehen.

$\lambda$ -definierbare  
Funktionen

**Definition 45** Eine Funktion  $f: \mathbb{N}^k \rightarrow \mathbb{N}$  heißt  $\lambda$ -definierbar, wenn es einen  $\lambda$ -Term  $\underline{f}$  derart gibt, daß

$$\underline{f}(n_1, \dots, n_k) =_{\beta} \underline{f} \underline{n}_1 \dots \underline{n}_k$$

für alle  $k$ -Tupel  $n_1, \dots, n_k$  gilt.

$\lambda$ -Definierbarkeit  
der Grundfunktionen

Es ist dann sofort zu sehen, daß alle Grundfunktionen  $\lambda$ -definierbar sind.  
Wir erhalten nämlich

$$\underline{C}_j^n = \lambda x_1 \dots x_n . \underline{j},$$

$$\underline{P}_j^n = \lambda x_1 \dots x_n . x_j$$

und

$$\underline{S} = \lambda x . \Pi^2(F, x).$$

Abschluß gegenüber  
Substitutionen

Sind  $g$  und  $h_1, \dots, h_m$  bereits als  $\lambda$ -definierbare Funktionen erkannt, so erhalten wir

$$\underline{\text{Sub}}(g, h_1, \dots, h_m) = \lambda x_1 \dots x_k . \underline{g}(\underline{h}_1 x_1 \dots x_k) \dots (\underline{h}_m x_1 \dots x_k).$$

Sehr viel schwieriger ist der Abschluß der  $\lambda$ -definierbaren Funktionen

gegenüber der primitiven Rekursion festzustellen. In der primitiven Rekursion wird eine Funktion implizit definiert, auch wenn nur auf Funktionswerte an vorhergehenden Argumenten rekuriert wird. Bei der impliziten Definition von Funktionen hilft aber der Rekursionssatz. Wenn wir also davon ausgehen, daß wir den Rekursionssatz schon haben, so erhalten wir die primitive Rekursion  $\text{Rec}(g, h)$  als eine Funktion mit dem Index  $e$ , der

$$\{e\}^k(n, \vec{x}) \simeq \begin{cases} g(\vec{x}), & \text{falls } n = 0, \\ h(\text{Pd}(n), \Phi^k(e, \text{Pd}(n), \vec{x}), \vec{x}), & \text{sonst} \end{cases} \quad (4.8)$$

erfüllt. Wir haben bei der Entwicklung des Rekursionssatzes ja bereits betont, daß es sich um einen rein kombinatorischen Satz handelt. Daher wird man im  $\lambda$ -Kalkül versuchen, zuerst ein Äquivalent für den Rekursionssatz zu erhalten und dieses dann zum Nachweis der Abgeschlossenheit gegenüber primitiver Rekursion zu verwenden.

Die Funktionalgleichung  $f(\vec{x}) \simeq g(f, \vec{x})$  läßt sich im  $\lambda$ -Kalkül, in dem wir Funktionen und deren Argumente als Einheit betrachten dürfen, einfach als

$$F =_{\beta} GF \quad (4.9)$$

schreiben. Nehmen wir  $G$  als gegeben an, und definieren wir

Der Fixpunktsatz  
des  $\lambda$ -Kalküls

$$F := (\lambda x. G(xx))(\lambda x. G(xx)),$$

so erhalten wir

$$F \rightarrow_1 G((\lambda x. G(xx))(\lambda x. G(xx))) = GF,$$

was einen Fixpunkt liefert. Man beachte hier die Ähnlichkeit zum Beweis des Rekursionssatzes. Der Zusammenhang zwischen dem Term  $G$  und seinem Fixpunkt  $(\lambda x. G(xx))(\lambda x. G(xx))$  ist völlig uniform und läßt sich daher durch eine Kombination beschreiben. Hier wäre es

$$\mathbf{FO} := \lambda y. (\lambda x. y(xx))(\lambda x. y(xx)),$$

die

$$G(\mathbf{FO}G) =_{\beta} \mathbf{FO}G$$

liefert. Also können wir den Fixpunktsatz (4.9) wie folgt verschärfen:

**Satz 46** Es gibt einen Fixpunktoperator  $\mathbf{FO}$ , der jedem  $\lambda$ -Term  $G$  einen Fixpunkt  $\mathbf{FO}G$  zuordnet. D.h., wir haben

$$G(\mathbf{FO}G) =_{\beta} \mathbf{FO}G.$$

## 4. $\lambda$ -definierbare Funktionen

---

Es gibt jedoch noch viele weitere Möglichkeiten, Fixpunktoperatoren zu definieren.

Mit Hilfe des Fixpunktsatzes lassen sich nun die Abschlußeigenschaften der  $\lambda$ -definierbaren Funktionen weiter studieren.

Um die Gleichung (4.8) behandeln zu können, benötigen wir die Vorgängerfunktion  $\underline{Pd}$ , für die ein  $\lambda$ -Repräsentant sich wie folgt darstellen läßt:

$$\underline{Pd} := \lambda x. \text{NULL}_x(\lambda x. x)(\Pi_2^2 x).$$

Um den Umgang mit dem  $\lambda$ -Kalkül etwas vertrauter zu machen, wollen wir dies kurz nachrechnen. Wegen  $\text{NULL}_0 =_\beta T$ , folgt

$$\underline{Pd} 0 =_\beta \lambda x. x = 0,$$

und wegen  $\text{NULL}_n =_\beta F$  für  $n \neq 0$ , folgt

$$\underline{Pd}(n+1) =_\beta \Pi_2^2 \Pi^2(F, n) =_\beta n.$$

Übersetzen wir (4.8) in die Sprache des  $\lambda$ -Kalküls, so sehen wir, daß wir einen  $\lambda$ -Term  $F$  benötigen, für den gilt

$$\begin{aligned} FNX_1 \dots X_n &=_\beta \text{IF } \text{NULL}_N \text{ THEN } (GX_1 \dots X) \\ &\quad \text{ELSE} \\ &\quad (H(\underline{Pd} N)(F(\underline{Pd} N)X_1 \dots X_n)X_1 \dots X_n). \end{aligned} \tag{4.10}$$

Ohne die Verwendung von `IF THEN ELSE` heißt das aber nur

$$FNX_1 \dots X_n =_\beta \text{NULL}_N(GX_1 \dots X_n)(H(\underline{Pd} N)(F(\underline{Pd} N)X_1 \dots X_n)X_1 \dots X_n).$$

Definieren wir

$$U := \lambda u v x_1 \dots x_n. \text{NULL}_v(Gx_1 \dots x_n)(H(\underline{Pd} v)(u(\underline{Pd} v)x_1 \dots x_n)x_1 \dots x_n),$$

so gibt es nach dem Fixpunktsatz ein  $F$  mit  $F =_\beta UF$ , und dieses  $F$  erfüllt offenbar (4.10). Damit haben wir eingesehen, daß die  $\lambda$ -definierbaren Funktionen gegenüber der primitiven Rekursion abgeschlossen sind.

Etwas problematischer ist der Abschluß gegenüber dem  $\mu$ -Operator. Wir haben nämlich mit Vorbedacht die  $\lambda$ -Definierbarkeit nur für totale Funktionen eingeführt. Damit haben wir vermieden, erklären zu müssen, wann wir eine Funktion als undefiniert betrachten. Wir wollen anschließend auf diese Problematik noch kurz eingehen. Wenn wir den Abschluß der  $\lambda$ -definierbaren Funktionen gegenüber dem  $\mu$ -Operator untersuchen wollen, können wir daher davon ausgehen, daß  $\mu x. f(x, \vec{y})$  eine totale Funktion ist. Dann muß aber sowohl  $f$  total sein als auch zu jedem Tupel  $\vec{y}$  ein  $x$  existieren mit  $f(x, \vec{y}) = 0$ . Wir dürfen nun annehmen, daß

Abschluß gegenüber  
primitiver Rekursion

Abschluß gegenüber  
dem  $\mu$ -Operator

wir die Funktion  $f$  bereits durch den  $\lambda$ -Term  $\underline{f}$  repräsentieren können. Was wir nun suchen, ist ein Repräsentant für die Funktion

$$g(x, \vec{y}) = \begin{cases} x, & \text{falls } f(x, \vec{y}) = 0, \\ g(x + 1, \vec{y}), & \text{sonst,} \end{cases}$$

denn  $g(0, \vec{y})$  liefert offenbar gerade den Wert  $\mu x. f(x, \vec{y})$ . Definieren wir

$$H := \lambda uxy_1 \dots y_n. \text{NULL}(\underline{f} xy_1 \dots y_n)x(u(\underline{\text{S}}x)y_1 \dots y_n),$$

so liefert der Fixpunkt  $G$  von  $H$  offenbar genau einen Repräsentanten für die oben angegebene Funktion  $g$ , und wir können

$$\underline{\mu f} := H 0$$

definieren.

Damit haben wir nachgewiesen, daß jede rekursive Funktion auch  $\lambda$ -definierbar ist.

Umgekehrt ist der Graph einer  $\lambda$ -definierbaren Funktion  $f$  durch

$$\{(\vec{n}, m) \mid \underline{f} \underline{n}_1 \dots \underline{n}_k =_\beta \underline{m}\}$$

gegeben, woraus nach dem vorherigen Abschnitt folgt, daß jede  $\lambda$ -definierbare Funktion einen rekursiv aufzählbaren Graphen hat. Beachten wir weiter, daß jede  $\lambda$ -definierbare Funktion per definitionem total ist, so erhalten wir den folgenden Satz:

**Satz 47** Die  $\lambda$ -definierbaren Funktionen sind genau die rekursiven Funktionen.

Rekursive vs.  
 $\lambda$ -definierbare  
Funktionen

Dieses Ergebnis hat selbst wieder Rückwirkungen auf die Theorie der  $\lambda$ -Terme. Wir wollen als Beispiel dafür folgern, daß sich rekursiv aufzählbare Mengen von  $\lambda$ -Termen angeben lassen, die nicht rekursiv sind.

Wir benützen die CHURCHsche These um einzusehen, daß sich rekursive Funktionen  $Ap$  und  $Num$  definieren lassen, für die gilt:

$$Ap(\lceil M \rceil, \lceil N \rceil) = \lceil MN \rceil$$

und

$$Num(n) = \lceil n \rceil.$$

Gemäß dem Repräsentationssatz (Satz 47) existieren daher repräsentierende  $\lambda$ -Terme **Ap** und **Num** für  $Ap$  und  $Num$ . Definieren wir für einen  $\lambda$ -Term  $M$

$$\underline{M} := \lceil M \rceil,$$

#### 4. $\lambda$ -definierbare Funktionen

---

so wird dies kaum zu Verwechslungen mit  $n$  führen, wie es für natürliche Zahlen  $n$  definiert ist. Für die repräsentierenden  $\lambda$ -Terme **Ap** und **Num** gilt nun

$$\mathbf{Ap} \underline{M} \underline{N} =_{\beta} \underline{M} \underline{N}$$

und

$$\mathbf{Num} \underline{M} =_{\beta} \underline{M}.$$

Ist nun  $F$  ein beliebiger  $\lambda$ -Term, so definieren wir

$$U := \lambda x. F(\mathbf{Ap} x \mathbf{Num} x) \quad \text{und} \quad X := U \underline{U}.$$

Dann folgt

$$X =_{\beta} U \underline{U} =_{\beta} F(\mathbf{Ap} \underline{U} \mathbf{Num} \underline{U}) =_{\beta} F(\mathbf{Ap} \underline{U} \underline{U}) =_{\beta} F \underline{U} \underline{U} \equiv F \underline{X}.$$

Wir haben daher eine Variante des Fixpunktsatzes erhalten, die besagt:

**Satz 48** Zu jedem  $\lambda$ -Term  $F$  existiert ein  $\lambda$ -Term  $X$  mit

$$F \underline{X} =_{\beta} X.$$

Wir nennen eine Menge  $\mathcal{A}$  von  $\lambda$ -Termen  $\beta$ -abgeschlossen, wenn

$$M \in \mathcal{A} \wedge M =_{\beta} N \implies N \in \mathcal{A}$$

gilt. Eine der klassischen Fragen der deskriptiven Mengenlehre ist die nach der *rekursiven Trennbarkeit* von Mengen. Dies bedeutet hier, daß wir von zwei nicht leeren Mengen  $\mathcal{A}$  und  $\mathcal{B}$  von (Kodes von)  $\lambda$ -Termen ausgehen und die Frage stellen, ob wir wohl eine rekursive Menge  $\mathcal{C}$  von  $\lambda$ -Termen finden können, derart daß  $\mathcal{A} \subseteq \mathcal{C}$  und  $\mathcal{C} \cap \mathcal{B} = \emptyset$  ist. Wenn wir annehmen, daß ein solches rekursives  $\mathcal{C}$  existiert, so haben wir einen  $\lambda$ -Term  $F$ , der dessen charakteristische Funktion repräsentiert. Für  $N \in \mathcal{C}$  gilt daher

$$F \underline{N} =_{\beta} \begin{cases} \underline{1}, & \text{falls } N \in \mathcal{C} \\ \underline{0}, & \text{falls } N \notin \mathcal{C}. \end{cases}$$

Wir nehmen  $\mathcal{A} \cap \mathcal{B} = \emptyset$  an, wählen  $M_0 \in \mathcal{A}$  und  $M_1 \in \mathcal{B}$  und definieren

$$H := \lambda x. \mathbf{NULL}(Fx) M_0 M_1. \tag{4.11}$$

Dann gilt

$$H \underline{N} =_{\beta} \begin{cases} M_0, & \text{falls } N \notin \mathcal{C} \\ M_1, & \text{falls } N \in \mathcal{C}. \end{cases}$$

Nach der Variante des Fixpunktsatzes, wie sie durch Satz 48 gegeben ist, haben wir nun einen  $\lambda$ -Term  $G$  mit der Eigenschaft, daß

$$H\underline{G} =_{\beta} G$$

gilt. Ist nun  $G =_{\beta} M_0$ , so folgt wegen  $M_0 \in \mathcal{A} \subseteq \mathcal{C}$  aber nach (4.11)  $G =_{\beta} M_1$  und für  $G \equiv M_1 \notin \mathcal{C}$  in analoger Weise  $G =_{\beta} M_0$ . Da  $\mathcal{A}$  und  $\mathcal{B}$  als  $\beta$ -abgeschlossen vorausgesetzt waren, steht dies aber im Konflikt zur Voraussetzung  $\mathcal{A} \cap \mathcal{B} = \emptyset$ . Es ist daher generell unmöglich, nicht leere  $\beta$ -abgeschlossene Mengen von  $\lambda$ -Terminen rekursiv zu trennen.

Als Korollar der rekursiven Untrennbarkeit nicht leerer  $\beta$ -abgeschlossener Mengen von  $\lambda$ -Terminen erhalten wir eine Variante des Satzes von RICE für den  $\lambda$ -Kalkül.

**Satz 49** *Ist  $\mathcal{A}$  eine nicht leere  $\beta$ -abgeschlossene echte Teilmenge aller  $\lambda$ -Terme, so ist  $\mathcal{A}$  nicht rekursiv.*

Eine Variante des Satzes von RICE für den  $\lambda$ -Kalkül

Als Folgerung erhalten wir dann, daß für einen  $\lambda$ -Term  $N$  die Menge

$$[N]_{\beta} := \{M \mid M =_{\beta} N\}$$

eine zwar rekursiv aufzählbare, jedoch nicht rekursive Menge ist. Damit folgt, daß die Relation

$$M =_{\beta} N,$$

die wir bereits als rekursiv aufzählbar erkannt haben, nicht rekursiv und nach der CHURCHSchen These daher algorithmisch nicht entscheidbar ist.

Wir wollen noch kurz auf die Problematik partieller  $\lambda$ -definierbarer Funktionen eingehen.

Die naheliegende Definition für die  $\lambda$ -Definierbarkeit partieller Funktionen ist natürlich die Folgende:

$\lambda$ -Definierbarkeit partieller Funktionen

$f: \mathbb{N}^k \longrightarrow_p \mathbb{N}$  heißt  $\lambda$ -definierbar, wenn es einen  $\lambda$ -Term  $F$  gibt mit

$$f(n_1, \dots, n_k) \simeq m \iff F \underline{n}_1 \dots \underline{n}_k =_{\beta} m.$$

Bei dieser Definition ist allerdings die Kohärenz von der Undefiniertheit partiellrekursiver Funktionen und deren repräsentierenden  $\lambda$ -Terminen nicht so trivial zu zeigen. Wollen wir den für totale Funktionen geführten Beweis einfach kopieren, so geraten wir bei der Substitution in Schwierigkeiten, wo die Undefiniertheit von  $h_i(\vec{x})$  für ein  $i \in \{1, \dots, n\}$  schon die Undefiniertheit von  $\text{Sub}(g, h_1, \dots, h_m)(\vec{x})$  nach sich zieht. Der repräsentierende  $\lambda$ -Term  $\underline{g}(\underline{h}_1 \vec{X}) \dots (\underline{h}_m \vec{X})$  kann aber selbst dann zu einer Normalform reduzieren, wenn eines der  $(\underline{h}_i \vec{X})$  undefiniert ist. Ein Beispiel für diese Situation bietet der Term

$$\lambda xy . x(\lambda x . x)\Omega,$$

der die Normalform  $\lambda x . x$  besitzt, obwohl  $\Omega$  selbst keine Normalform hat. Diese Schwierigkeit läßt sich allerdings umgehen. Dazu geht man von dem Normalformen-

#### 4. $\lambda$ -definierbare Funktionen

---

theorem für partiellrekursive Funktionen aus und stellt eine partiellrekursive Funktion  $f$  durch

$$f(\vec{x}) \simeq U(\mu y \cdot \overline{\text{sg}}(\chi_{T^n}(e, \vec{x}, y)) = 0)$$

dar. Die Funktion  $\overline{\text{sg}} \circ \chi_{T^n}$  ist primitiv rekursiv – und somit rekursiv – und besitzt damit nach dem oben gezeigten einen repräsentierenden  $\lambda$ -Term  $G$ . Ferner definieren wir  $F$  als repräsentierenden Term für  $f$  so, wie wir es gemäß dem obigen Beweis getan hätten. Ist dann  $f(\vec{x})$  definiert, so gilt nach dem oben gezeigten, daß

$$\underline{f(x_1, \dots, x_n)} = F \underline{x_1} \dots \underline{x_n}$$

ist. Ist  $f(\vec{x})$  aber undefiniert, so ist nicht garantiert, daß  $F \underline{x_1} \dots \underline{x_n}$  ebenfalls nicht definiert ist. Um dies zu erreichen, definiert man einen Repräsentanten

$$H := \lambda x_1 \dots x_n. P(x_1, \dots, x_n)(F x_1 \dots x_n)$$

mit einem ‘korrigierenden Term’  $P(x_1, \dots, x_n)$  derart, daß

$$P(\underline{x_1}, \dots, \underline{x_n}) =_\beta I(:= \lambda x. x)$$

ist, falls es ein  $k$  gibt mit  $\overline{\text{sg}}(\chi_{T^n}(x_1, \dots, x_n, k)) = 0$  und  $P(\underline{x_1}, \dots, \underline{x_n})$  eine unendliche Reduktionskette erzeugt, falls  $\overline{\text{sg}}(\chi_{T^n}(x_1, \dots, x_n, k)) = 1$  für alle natürlichen Zahlen  $k$  ist. Die Definition von  $P(x_1, \dots, x_n)$  ist nicht ganz einfach, weswegen wir hier darauf verzichten wollen.

### 4.5 $\lambda$ -Kalküle über Typenstrukturen

Neben dem in den vorherigen Abschnitten behandelten typenfreien  $\lambda$ -Kalkül spielen – gerade in jüngerer Zeit – auch  $\lambda$ -Kalküle über Typenstrukturen eine gewisse Rolle. In ihnen versucht man die Vorteile der einfachen Kombinatorik des  $\lambda$ -Kalküls mit der größeren Übersichtlichkeit einer Sprache mit Typen zu verbinden. Ihren Ursprung haben Kalküle über Typenstrukturen in beweistheoretischen Untersuchungen. KURT GÖDEL war einer der ersten, der auf die Verwendbarkeit von Funktionalen mit Typen in grundlagentheoretischen Untersuchungen hinwies. Wir wollen – um den Rahmen des Buches nicht zu sprengen – allerdings nur grob auf die Grundbegriffe getypter Kalküle eingehen.

Da wir hier nur auf einige prinzipielle Eigenschaften von Kalkülen mit Typen eingehen wollen, werden wir uns zuerst auf eine sehr einfache Typenstruktur beschränken, die als *endliche Typen* bekannt sind.

Endliche Typen sind zum einen der Grundtyp  $o$  und, falls  $\sigma$  und  $\tau$  bereits endliche Typen sind, auch der Typ  $\sigma \rightarrow \tau$ .

Wir werden Typen im folgenden immer mit kleinen griechischen Buchstaben bezeichnen. Die Intuition, die hinter der Definition der Typen steckt, ist, daß es sich bei Objekten des Grundtyps  $o$  um bestimmte vorgegebene Objekte handelt, wie z. B. natürliche Zahlen oder Zeichenreihen,

während Objekte eines Typs  $\sigma \rightarrow \tau$  Abbildungen sind, die Objekte vom Typ  $\sigma$  auf Objekte des Typs  $\tau$  abbilden. Bezeichnen wir mit  $\text{Typ}(\tau)$  die Menge aller Objekte des Typs  $\tau$ , so wäre eine mögliche Interpretation für die Struktur der endlichen Typen geben durch

$$\text{Typ}(o) := \mathbb{N}$$

und

$$\text{Typ}(\sigma \rightarrow \tau) := \{f \mid f: \text{Typ}(\sigma) \longrightarrow \text{Typ}(\tau)\}.$$

Um einen getypten  $\lambda$ -Kalkül zu erhalten, geht man nun davon aus, daß für jeden Typ  $\sigma$  beliebig viele Variablen zur Verfügung stehen. Wir notieren mit  $x^\sigma$ , daß  $x$  eine Variable des Typs  $\sigma$  ist. In Analogie zum typenfreien  $\lambda$ -Kalkül definiert man die Terme des getypten  $\lambda$ -Kalküls durch die folgenden Klauseln:

### Definition 50

Getypte  $\lambda$ -Terme

1. Jede Variable  $x^\sigma$  ist ein  $\lambda$ -Term des Typs  $\sigma$ .
2. Ist  $T$  ein  $\lambda$ -Term des Typs  $\tau$ , so ist  $\lambda x^\sigma . T$  ein  $\lambda$ -Term des Typs  $\sigma \rightarrow \tau$ .
3. Ist  $T$  ein  $\lambda$ -Term des Typs  $\sigma \rightarrow \tau$  und  $S$  ein  $\lambda$ -Term des Typs  $\sigma$ , so ist  $(TS)$  ein  $\lambda$ -Term des Typs  $\tau$ .

Wir sehen allerdings sofort, daß wir mit den so eingeführten  $\lambda$ -Termen nicht einmal in der Lage sind, natürliche Zahlen zu definieren. Natürliche Zahlen sollen Objekte vom Typ  $o$  werden. Die einzigen Objekte dieses Typs, die uns im bisher definierten Kalkül zur Verfügung stehen, sind Variablen. Konstante Größen, wie beispielsweise natürliche Zahlen, lassen sich jedoch nicht durch Variablen repräsentieren, die ja noch interpretationsbedürftig sind, sondern höchsten durch geschlossene Terme, d.h. Terme, die keine freien Variablen mehr enthalten. Wann immer wir jedoch eine Variable durch  $\lambda$ -Abstraktion (d.h. durch die Klausel 2 in Definition 50 ) binden, verlassen wir den Bereich der Grundtypen. Dies bedeutet, daß wir im Gegensatz zum typenfreien  $\lambda$ -Kalkül, in dem wir natürliche Zahlen durch geschlossene  $\lambda$ -Terme repräsentieren, im  $\lambda$ -Kalkül mit Typen schon zur Darstellung natürlicher Zahlen nicht ohne Atome auskommen. Wir werden also gewisse Atome (oder Konstanten, wie wir sie im folgenden nennen wollen) zur Sprache des getypten  $\lambda$ -Kalküls hinzunehmen und Definition 50 um die Klausel

0. Jede Konstante eines Typs  $\sigma$  ist eine  $\lambda$ -Term vom Typ  $\sigma$

erweitern. Um natürliche Zahlen definieren zu können, reicht es aus, eine Konstante  $0$  vom Typ  $o$  und eine Konstante  $S$  vom Typ  $o \rightarrow o$  für die Nachfolgerfunktion einzuführen. Wir definieren die  $\lambda$ -Terme

$$\underline{0} := 0$$

#### 4. $\lambda$ -definierbare Funktionen

---

und

$$\underline{n+1} := \mathbf{S}\underline{n}.$$

Dann ist  $\underline{n}$  für jede natürliche Zahl  $n$  ein  $\lambda$ -Term vom Typ  $o$ , der  $n$  repräsentiert.

Konversionen und Reduktionen definieren wir wie für den typenfreien  $\lambda$ -Kalkül. Wir haben also

$$(\lambda x^\sigma . T)S \rightarrow_1 T_{x^\sigma}(S)$$

und

$$U \rightarrow_1 V \implies \lambda x . U \rightarrow_1 \lambda x . V, US \rightarrow_1 VS \text{ und } SU \rightarrow_1 SV,$$

wobei wir immer stillschweigend voraussetzen, daß diese Terme alle typengerecht gebildet sind. Insbesondere ist im wesentlichen Konversionschritt

$$(\lambda x^\sigma . T)S \rightarrow_1 T_{x^\sigma}(S)$$

$S$  ein  $\lambda$ -Term des Typs  $\sigma$ , woraus sich ergibt, daß  $T_{x^\sigma}(S)$  wieder ein korrekt gebildeter  $\lambda$ -Term ist.

Mit Hilfe der neuen Konstanten werden die Grundfunktionen auch im getypten  $\lambda$ -Kalkül definierbar. So ist  $\mathbf{S}$  als Konstante vorhanden und wir definieren

$$\mathbf{C}_k^n = \lambda x_1^o \dots x_n^o . k$$

sowie

$$\mathbf{P}_k^n = \lambda x_1^o \dots x_n^o . x_k^o.$$

Des Weiteren zeigt die gleiche Konstruktion wie im typenfreien  $\lambda$ -Kalkül, daß die bislang definierbaren Funktionen auch gegenüber dem Substitutionsoperator abgeschlossen sind. Aber bereits der Abschluß gegenüber der primitiven Rekursion ist nicht gegeben. Beim Nachweis des Abschlusses der im typenfreien  $\lambda$ -Kalkül definierbaren Funktionen gegenüber der primitiven Rekursion, hat der Fixpunktsatz eine entscheidende Rolle gespielt. Dieser Satz hing seinerseits **wesentlich** von der Typenfreiheit des  $\lambda$ -Kalküls ab, die es erlaubt,  $\lambda$ -Terme auf sich selbst anzuwenden. Diese Selbstanwendung steht uns aber im getypten  $\lambda$ -Kalkül nicht mehr zur Verfügung. Damit erhalten wir mit der bisherigen Definition nicht einmal alle primitiv rekursiven Funktionen. Wollen wir eine größere Klasse von Funktionen erfassen, so müssen wir weitere Konstanten zum Kalkül hinzunehmen. Wir führen daher für jeden Typ  $\tau$  eine Konstante  $R_\tau$  ein, die es erlauben wird, Funktionen durch Rekursion zu definieren. Um

die Typenbezeichnung übersichtlicher zu gestalten, wollen wir Typen als nach rechts geklammert schreiben und die Abkürzung

$$\sigma_1, \dots, \sigma_n \rightarrow \tau := \sigma_1 \rightarrow \dots \rightarrow \sigma_n \rightarrow \tau := (\sigma_1 \rightarrow (\dots \rightarrow (\sigma_n \rightarrow \tau) \dots))$$

einführen. Diese Schreibweise ist einprägsam, da für Terme  $T$  vom Typ  $\sigma_1, \dots, \sigma_n \rightarrow \tau$  und  $S_1, \dots, S_n$  der entsprechenden Typen  $\sigma_1, \dots, \sigma_n$  der Term  $TS_1 \dots S_n$  vom Typ  $\tau$  ist. Der Rekursor  $R_\tau$  ist dann vom Typ

$$\tau, (o, \tau \rightarrow \tau), o \rightarrow \tau$$

und wird durch die Konversionsregeln

$$R_\tau GH \rightarrow_1 G$$

und

$$R_\tau GH (ST) \rightarrow_1 HT(R_\tau GHT)$$

definiert. Alle übrigen Definitionen, inklusive der Relation  $=_\beta$ , übernehmen wir vom typenfreien  $\lambda$ -Kalkül. Wir erhalten so die ursprünglich von K. GöDEL eingeführte Theorie **T** der *Funktionale endlicher Typen*. Dabei wollen wir einen getypten  $\lambda$ -Term  $F$  genau dann ein Funktional nennen, wenn  $F$  keine freien Variablen enthält.

Die Theorie **T** der  
Funktionale  
endlicher Typen

Es ist nicht sonderlich schwer einzusehen, daß auch der getypte  $\lambda$ -Kalkül die CHURCH-ROSSER Eigenschaft besitzt, womit die  $=_\beta$ -Relation auch hier eine Äquivalenzrelation wird. Im Unterschied zum typenfreien  $\lambda$ -Kalkül besitzt aber jeder Term des getypten Kalküls eine Normalform. Insbesondere gilt der folgende Satz.

**Satz 51** *Jedes Funktional der Theorie **T** besitzt eine eindeutig bestimmte Normalform.*

Ein einfacher Beweis dafür findet sich in [SHOE67]. Ein tiefer gehender Beweis, der mehr Information über die Komplexität der durch die Theorie **T** definierbaren Funktionen liefert, geht auf W. HOWARD zurück und ist beispielsweise in [SCHÜ77] dargestellt. Wir nennen dabei eine Funktion  $f: \mathbb{N}^n \rightarrow \mathbb{N}$  in der Theorie **T** definierbar, wenn es ein Funktional  $F$  vom Typ  $\underbrace{o, \dots, o}_{n-fach} \rightarrow o$  so gibt, daß

$$F \underline{z}_1 \dots \underline{z}_n =_\beta f(z_1, \dots, z_n)$$

für alle  $n$ -Tupel  $z_1, \dots, z_n$  natürlicher Zahlen gilt. Aus Satz 51 folgt nun, daß alle in **T** definierbaren Funktionen total sind. Es ist natürlich wieder leicht einzusehen, daß die Relation  $=_\beta$  rekursiv aufzählbar ist, woraus sich ergibt, daß alle in **T** definierbaren Funktionen rekursiv sind. Es ist

## 4. $\lambda$ -definierbare Funktionen

---

Beweisbar rekursive  
Funktionen der  
PEANO Arithmetik

auch sehr leicht einzusehen, daß jede primitiv rekursive Funktion in  $T$  definierbar ist. Die Klasse der in  $T$  definierbaren Funktionen ist jedoch eine *echte* Oberklasse der primitiv rekursiven Funktionen. Sie umfaßt genau die Klasse der rekursiven Funktionen, deren Totalität sich aus den PEANO-Axiomen ableiten läßt. Man charakterisiert sie daher oft als die *beweisbar rekursiven Funktionen der PEANO Arithmetik*.

Die Typenstruktur läßt sich natürlich leicht erweitern, ohne daß dies zu einer tatsächlichen Vergrößerung der Klasse der definierbaren Funktionen führt. Man kann weitere Grundtypen einführen (wie beispielsweise den Typ BOOLEAN für Wahrheitswerte oder REAL für Gleitkommazahlen). Man kann Typen  $\sigma_1 \times \dots \times \sigma_n$  für kartesische Produkte hinzufügen und so fort. So lassen sich die in Programmiersprachen üblichen komplexeren Datentypen ohne weiteres als endliche Typen auffassen. Eine Programmiersprache, die sich exakt am getypten  $\lambda$ -Kalkül orientierte, hätte den Vorteil, daß jedes in ihr geschriebene, syntaktisch korrekte Programm auf Grund von Satz 51 auch terminierte. Der Nachteil, daß die durch so eine Sprache erfaßte Klasse berechenbarer Funktionen nur eine echte Teilklasse der theoretisch berechenbaren Funktionen ist, dürfte angesichts der Tatsache, daß bereits die Berechnung primitiv rekursiver Funktionen jenseits aller physikalisch gegebenen Grenzen liegt, kaum ins Gewicht fallen. Allerdings wäre eine solche Sprache mit hoher Wahrscheinlichkeit viel zu schwerfällig, als daß sie allgemein angenommen würde.

Polymorphe Typen

Eine deutliche Verstärkung erfährt der getypte  $\lambda$ -Kalkül durch das Zulassen von *Typenvariablen*. Man spricht dann von *polymorphen Typen*. Die polymorphen Typen werden durch die folgenden Regeln erzeugt:

- Jede Typenvariable – mitgeteilt durch  $\alpha, \beta, \dots$  – ist ein Typ.
- Mit  $\sigma$  und  $\tau$  ist auch  $\sigma \rightarrow \tau$  ein Typ.
- Mit  $\sigma$  ist auch  $\Pi\alpha. \sigma$  ein Typ.

Polymorph getypte  
 $\lambda$ -Terme

Die Terme des zur polymorphen Typenstruktur gehörenden  $\lambda$ -Kalküls erhält man durch die folgenden Klauseln:

- Ist  $x$  eine Objektvariable und  $\sigma$  ein Typ, so ist  $x^\sigma$  ein Term des Typs  $\sigma$ .
- Ist  $U$  ein Term des Typs  $\tau$ ,  $x$  eine Objektvariable und  $\sigma$  ein Typ, so ist  $\lambda x^\sigma. U$  ein Term des Typs  $\sigma \rightarrow \tau$ .
- Ist  $U$  ein Term des Typs  $\sigma$  und ist  $\alpha$  eine Typenvariable, die nicht frei im Typ einer frei in  $U$  auftretenden Objektvariablen auftritt, so ist  $\Lambda\alpha. U$  ein Term vom Typ  $\Pi\alpha. \sigma$ .
- Sind  $U$  und  $V$  Terme der Typen  $\sigma \rightarrow \tau$  und  $\sigma$ , so ist  $(UV)$  ein Term des Typs  $\tau$ .

- Ist  $U$  ein Term des Typs  $\Pi\alpha.\tau$  und  $\sigma$  ein Typ, so ist  $(U\sigma)$  ein Term des Typs  $\tau_\alpha(\sigma)$ .

Die Konversionsregeln des getypten  $\lambda$ -Kalküls werden um die Regel

$$(\Lambda\alpha.U)\tau \rightarrow_1 U_\alpha(\tau)$$

erweitert, wobei wir nochmals in Erinnerung rufen wollen, daß  $U_\alpha(\tau)$  die Zeichenreihe bedeutet, die aus  $U$  entsteht, wenn die Typenvariable  $\alpha$  überall, wo sie frei auftritt, durch  $\tau$  ersetzt wird.

Das hier angegebene Termsystem wurde zuerst von J. Y. GIRARD entwickelt und von ihm *System F* genannt. Die ursprüngliche Motivation zur Entwicklung des Systems **F** waren beweistheoretische Untersuchungen der Arithmetik der zweiten Stufe. Tatsächlich lassen sich vermöge des sogenannten HOWARD-CURRY Isomorphismus die Typen des Systems **F** als Formeln der Prädikatenlogik zweiter Stufe auffassen und das System **F** entspricht dann ziemlich exakt einem Kalkül des natürlichen Schließens für die intuitionistische Prädikatenlogik der zweiten Stufe. In **F** lassen sich die gängigen Typen relativ leicht definieren. Auch der Typ der natürlichen Zahlen ist definierbar, ähnlich, wie sich auch die natürlichen Zahlen in der Prädikatenlogik der zweiten Stufe definieren lassen. Daher kann man auch auf Konstanten für 0, die Nachfolgerfunktion und den Rekursor verzichten.

Die Reduktionen im System **F** werden natürlich wieder durch rekursiv aufzählbare Relationen beschrieben. Ähnlich wie für das System **T** lässt sich auch für **F** zeigen, daß jeder Term zu einer Normalform reduziert. Allerdings ist der Nachweis deutlich schwieriger als der für **T**. Er wurde erstmals von GIRARD geführt. Damit sind alle in **F** repräsentierbaren Funktionen total und somit rekursiv. Es läßt sich zeigen, daß die in **F** definierbaren Funktionen genau die rekursiven Funktionen sind, deren Totalität in der Arithmetik der zweiten Stufe beweisbar ist.

Was das System **F** so interessant macht, ist die Tatsache, daß sich durch den Typ  $\Pi\alpha.\sigma$  Funktionen beschreiben lassen, die uniform auf allen Typen  $\sigma_\alpha(\tau)$  arbeiten. Da Interpretationen polymorpher Typen nicht mehr so leicht anzugeben sind wie für einfachen Typen, wollen wir es bei diesen etwas vagen Andeutungen belassen. Wer sich detaillierter damit befassen will, sei auf [GIR89] verwiesen.

Das System **F**  
polymorphe  
Funktionale



---

## 5. Unentscheidbare Probleme

Als eine Anwendung der Theorie rekursiver und rekursiv aufzählbarer Mengen wollen wir uns nun mit unlösbaren Problemen beschäftigen. Bezeichnet man ein Problem als *unlösbar* oder *unentscheidbar*, so meint man im allgemeinen damit, daß es algorithmisch unlösbar ist, d. h., daß es keine Maschine geben kann, die das vorgelegte Problem entscheidet. Man kennt sehr viele Probleme, die in diesem Sinne unlösbar sind. Wir wollen uns hier auf zwei Beispiele beschränken, die von besonderem Interesse für die Informatik sind.

Das Urbeispiel eines unlösbaren Problems haben wir ja bereits kennengelernt. Es ist das Halteproblem für Turingmaschinen. Der Schlüssel dort war die Menge

$$K = \{x \mid (\exists y)T^1(x, x, y)\} = \{x \mid x \in \text{dom}(\{x\}^1)\} = \{x \mid x \in W_x\},$$

die wir als nicht rekursiv und somit nicht entscheidbar erkannt haben. Es ist eine bewährte Technik, die Unlösbarkeit eines Problems dadurch nachzuweisen, daß man es auf das Halteproblem reduziert. Wir wollen uns daher zunächst kurz mit Reduktionen von Entscheidungsproblemen beschäftigen.

### 5.1 Unlösbarkeitsgrade

Die Reduktion eines Entscheidungsproblems für eine Menge  $M$  auf eine Menge  $M'$ , dessen Entscheidungsproblem bereits gelöst ist, führt auf den wichtigen Begriff der *Umlösbarkeitsgrade*. Unter einem Umlösbarkeitsgrad versteht man eine Klasse von Relationen, deren Entscheidungsproblem "gleich schwer" ist. Dabei ist allerdings noch zu klären, was es heißen soll, daß zwei Relationen ein "gleich schweres" Entscheidungsproblem haben. Haben wir zwei Relationen  $R_1$  und  $R_2$  gegeben, so können wir sagen, daß das Entscheidungsproblem von  $R_1$  auf das von  $R_2$  reduziert ist, wenn die Annahme der Lösung des Entscheidungsproblems für  $R_2$  immer auch eine Lösung des Entscheidungsproblems für  $R_1$  beinhaltet. Es sind eine Reihe solcher Reduktionsmöglichkeiten bekannt. Die Allgemeinste wird durch die sogenannte *relative Rekursivität* erzeugt. Um diesen Begriff zu präzisieren, nennen wir eine Funktion  $F$  partiellrekursiv in einer Funktion  $G$ , wenn  $F$  durch einen Funktionsterm dargestellt werden kann, der sich aus der Funktion  $G$  und den Grundfunktionen  $C_k^n$ ,  $P_k^n$  und  $S$  durch die Grundoperationen Sub, Rec und

Umlösbarkeitsgrade

Relative Rekursivität

## 5. Unentscheidbare Probleme

---

dem unbeschränkten Suchoperator zusammensetzen läßt. In der Sprache der maschinenberechenbaren Funktionen können wir uns die Tatsache, daß eine Funktion  $F$  relativ zu einer Funktion  $G$  maschinenberechenbar ist, so veranschaulichen, daß ein Orakel vorliegt, das uns im Laufe der Berechnung von  $F(\vec{m})$  immer dann die Werte  $G(\vec{n}_i)$  offenbart, wenn dies für die Berechnung von  $F(\vec{m})$  erforderlich ist. Da jede Berechnung endlich zu sein hat, werden im Verlaufe der Rechnung an das Orakel für  $G$  nur endlich viele Fragen gestellt. Das bedeutet, daß für eine in  $G$  rekursive Funktion  $F$  während der Berechnung von  $F(\vec{m})$  das Orakel für  $G$  nur an endlich vielen Argumenten  $\vec{n}_1, \dots, \vec{n}_k$  nach den Werten von  $G(\vec{n}_i)$  befragt wurde. Im Regelfall betrachtet man Funktionen, die rekursiv in einer Menge  $A$  sind. In dieser Situation geht man dann davon aus, daß ein Orakel existiert, das uns endlich viele Anfragen  $\vec{n}_i \in A?$  zu beantworten hat. Ist eine Relation  $B$  rekursiv in einer Menge  $A$ , so läßt sich das Entscheidungsproblem für  $B$  auf das der Menge  $A$  zurückführen. Wollen wir  $x \in B?$  entscheiden und ist  $B$  rekursiv in  $A$ , so lassen wir den "relativen Algorithmus" für  $x \in B$  ablaufen und setzen jedesmal, wenn im Verlaufe des Entscheidungsverfahrens nach  $n \in A?$  gefragt wird, den vorausgesetzten Entscheidungsalgorithmus für  $A$  in Gang. Diese Art von Reduzibilität ist als *Turingreduzibilität* bekannt und man notiert durch

$$B \leq_T A,$$

daß  $B$  turingreduzibel auf  $A$  ist. Gilt  $A \leq_T B$  und  $B \leq_T A$ , so haben  $A$  und  $B$  offenbar ein gleich schweres Entscheidungsproblem, was nur soviel heißen mag, daß jede Lösung des Entscheidungsproblems für  $A$  auch eine für  $B$  erzeugt und umgekehrt. Man sagt dann, daß  $A$  und  $B$  *turingäquivalent* sind und notiert dies durch

$$A \equiv_T B.$$

Die Turingäquivalenz ist offensichtlich eine Äquivalenzrelation auf den Teilmengen der natürlichen Zahlen. Die Klasse aller turingäquivalenten Relationen bilden einen *Unlösbarkeitsgrad*.

Das Studium der Struktur der Unlösbarkeitsgrade ist ein wesentliches Anliegen der klassischen Rekursionstheorie. Zwei Grade sind uns bereits bekannt. Zum einen ist dies der Grad der entscheidbaren Relationen, der oft mit 0 bezeichnet wird und alle Relationen enthält, die turingäquivalent zur leeren Menge (und damit rekursiv) sind, und der Grad  $0'$ , in dem die eingangs erwähnte Menge  $K$  liegt. Beide Grade sind offensichtlich verschieden, da  $K$  nicht im Grad 0 liegen kann. Lange Zeit offen war die von E. POST 1944 gestellte Frage, ob es zwischen 0 und  $0'$  noch weitere rekursiv aufzählbare Turinggrade gibt. Erst 1956 konnten unabhängig voneinander R. FRIEDBERG und A. A. MUCHNIK diese Frage

Turingreduzibilität

Turingäquivalenz

POSTs Problem

positiv beantworten.

Neben den Turinggraden betrachtet man jedoch auch noch feinere Unlösbarkeitsgrade. Unter diesen sind die sogenannten m-Grade besonders wichtig, die wir kurz einführen wollen.

**Definition 52** Eine Relation  $R_1$  heißt "many-one reduzibel" (kurz  $m$ -Reduzibilität reduzibel) auf eine Relation  $R_2$ , notiert als

$$R_1 \leq_m R_2,$$

wenn es eine rekursive (und damit totale) Funktion  $f$  derart gibt, daß

$$R_1(\vec{x}) \iff R_2(f(\vec{x}))$$

für alle Tupel  $\vec{x}$  gilt.

Ist die Funktion  $f$  injektiv, so spricht man von "one-one-Reduzibilität" oder kurz 1-Reduzibilität und notiert dies durch

$$R_1 \leq_1 R_2.$$

Es ist ganz offensichtlich, daß

$$A \leq_1 B \implies A \leq_m B \implies A \leq_T B$$

gilt. Aus der  $m$ -Reduzibilität erhalten wir wieder in kanonischer Weise die m-Äquivalenz durch

$$R_1 \equiv_m R_2 \text{ falls } R_1 \leq_m R_2 \text{ und } R_2 \leq_m R_1.$$

Die m-äquivalenten Mengen bilden einen *m-Grad*. Völlig analog erhält m-Grade man 1-Grade.

Von besonderer Bedeutung sind sogenannte *m-vollständige Relationen*. Eine Relation  $R$  heißt m-vollständig für eine Klasse  $\mathcal{K}$  von Relationen, wenn  $R \in \mathcal{K}$  ist und  $P \leq_m R$  für jedes  $P \in \mathcal{K}$  gilt. Intuitiv bedeutet dies, daß die Lösung des Entscheidungsproblems für eine vollständige Relation  $R$  schon das Entscheidungsproblem für alle Relationen in  $\mathcal{K}$  löst.

Eine wesentliche Eigenschaft des Halteproblems für Turingmaschinen liegt darin, m-vollständig für die rekursiv aufzählbaren Mengen zu sein. Dies ist ganz einfach einzusehen. Wir wissen bereits, daß  $K$  rekursiv aufzählbar ist. Ist nun  $R$  eine beliebige rekursiv aufzählbare Menge, so gibt es einen Index  $e$  mit  $R = W_e$ , d.h., es gilt

$$x \in R \iff \exists y T(e, x, y).$$

Dies können wir als

## 5. Unentscheidbare Probleme

---

$$(\langle e, x \rangle)_1 \in W_{(\langle e, x \rangle)_0}$$

schreiben. Definieren wir

$$h(u, v) \simeq 1 \text{ falls } (u)_1 \in W_{(u)_0},$$

so hat  $h$  einen rekursiv aufzählbaren Graphen und ist daher partiellrekursiv. Für einen Index  $e_0$  von  $h$  erhalten wir mit dem  $S_n^m$ -Theorem

$$\{S(e_0, u)\}(v) \simeq 1 \iff (u)_1 \in W_{(u)_0}.$$

Da der Wert von  $\{S(e_0, u)\}(v)$  offenbar nicht von  $v$  abhängt, gilt schließlich

$$\begin{aligned} (\langle e, x \rangle)_1 \in W_{(\langle e, x \rangle)_0} &\iff \{S(e_0, \langle e, x \rangle)\}(S(e_0, \langle e, x \rangle)) \simeq 1 \\ &\iff S(e_0, \langle e, x \rangle) \in W_{S(e_0, \langle e, x \rangle)} \\ &\iff S(e_0, \langle e, x \rangle) \in K. \end{aligned}$$

Die Funktion  $g(x) := S(e_0, \langle e, x \rangle)$  ist offenbar rekursiv und reduziert  $R$  auf  $K$ , d.h. wir haben

$$x \in R \iff \exists y T(g(x), g(x), y) \iff g(x) \in K.$$

Gelingt es uns zu zeigen, daß sich das Halteproblem – und damit die Menge  $K$  – auf ein vorgegebenes Problem reduzieren läßt, so kann dieses nicht algorithmisch lösbar sein, da jede Lösung auch eine Lösung des Halteproblems nach sich ziehen würde. Wir wollen in den beiden folgenden Abschnitten diese Technik an zwei Beispielen erläutern.

## 5.2 CHOMSKY Grammatiken

Das erste dieser Beispiele soll ein Wortproblem behandeln. Dazu gehen wir von einem Alphabet  $\Sigma$  aus, worunter wir lediglich eine nicht leere endliche Menge zu verstehen haben. Die Elemente von  $\Sigma$  nennen wir die *Zeichen* oder manchmal auch die *Buchstaben* des Alphabets. Die Menge der *Wörter* über  $\Sigma$  bezeichnen wir wieder mit  $\Sigma^*$ , die der nichtleeren Wörter mit  $\Sigma^+$ . Eine *Sprache* über dem Alphabet  $\Sigma$  ist eine Menge

$$\mathcal{L} \subseteq \Sigma^*.$$

Das Wortproblem

Das *Wortproblem* für eine Sprache  $\mathcal{L}$  besteht nun darin, einen Algorithmus zu finden, der für ein Wort  $w \in \Sigma^*$  die Frage

$$w \in \mathcal{L}?$$

entscheidet. Endliche Alphabete lassen sich natürlich wieder problemlos durch natürliche Zahlen kodieren. Wörter über einem Alphabet treten dann als Kodes endlicher Folgen natürlicher Zahlen auf. Damit läßt sich jede Sprache als eine Menge natürlicher Zahlen kodieren und das Wortproblem für eine Sprache erhält dann die Form:

*Ist die Menge  $\tilde{\mathcal{L}} := \{w^n \mid w \in \mathcal{L}\}$  rekursiv?,*

wobei  $w^n$ , wie schon früher, den Kode eines Wortes  $w$  bezeichnet.

Wenn keine näheren Informationen über  $\mathcal{L}$  vorliegen, können wir natürlich auch keine Hoffnung auf eine positive Beantwortung dieser Frage hegen. Wir wollen daher Sprachen untersuchen, die in gewisser Weise selbst algorithmisch erzeugt wurden. Dazu betrachten wir zunächst Regelsysteme.

Ein *Regelsystem* (oder *Semi-THUE-System*) für das Alphabet  $\Sigma$  ist *Semi-THUE-Systeme* eine endliche Menge

$$R \subset \Sigma^* \times \Sigma^*.$$

Wir wollen die Wörter aus  $\Sigma^*$  im folgenden mit

$$u, v, w, \dots,$$

die Buchstaben mit

$$z, z_1, \dots$$

bezeichnen.  $uv$  bezeichnet die Aneinanderreihung der Wörter  $u$  und  $v$ . Oft schreiben wir auch  $uzv$  um auszudrücken, daß der Buchstabe  $z$  im Wort  $uzv$  zwischen den Teilwörtern  $u$  und  $v$  steht.

Ist nun ein Regelsystem  $R$  für das Alphabet  $\Sigma$  gegeben, so heißt

$$w_1 uw_2 \xrightarrow{R} w_1 vw_2$$

Direkte  
Ableitungsschritte

ein *direkter Ableitungsschritt* bezüglich  $R$ , wenn

$$(u, v) \in R$$

ist und  $w_1, w_2$  beliebige Wörter über  $\Sigma$  sind.

Wie im  $\lambda$ -Kalkül definieren wir die  $n$ -fache Iteration direkter Ableitungsschritte durch

$$u \xrightarrow{R}_0 v, \quad \text{falls } u = v,$$

und

$$u \xrightarrow{R}_{n+1} v, \quad \text{falls es ein } w \text{ gibt mit } u \xrightarrow{R}_n w \text{ und } w \xrightarrow{R}_1 v.$$

Die reflexive und transitive Hülle der Relation  $\xrightarrow{R}$  ist dann wieder gegeben durch

## 5. Unentscheidbare Probleme

---

$$u \xrightarrow{R} v \iff (\exists n)[u \xrightarrow{n} v].$$

Ableitung  
CHOMSKY  
Grammatiken

Wir sagen dann, daß das Wort  $v$  aus dem Wort  $u$  *abgeleitet* werden kann.

Eine CHOMSKY Grammatik ist durch ein Quadrupel

$$\mathcal{G} = (\Sigma, S, \Omega, R)$$

definiert, wobei

- $\Sigma$  ein Alphabet ist,
- $S \in \Sigma$  das *Startsymbol* und
- $\Omega \subseteq \Sigma$  das *terminale Alphabet* heißt, und schließlich
- $R$  ein Regelsystem für  $\Sigma$  ist.

Die von einer CHOMSKY Grammatik  $\mathcal{G} = (\Sigma, S, \Omega, R)$  erzeugte Sprache ist die Menge

$$\mathcal{L}_G := \{w \in \Omega^* \mid S \xrightarrow{R} w\}.$$

CHOMSKY Sprachen

Eine Sprache  $\mathcal{L}$ , d.h. eine Menge  $\mathcal{L} \subseteq \Sigma^*$  von Wörtern über einem Alphabet  $\Sigma$ , heißt eine CHOMSKY Sprache, wenn es eine CHOMSKY Grammatik gibt, die  $\mathcal{L}$  erzeugt.

Ähnlich, wie wir dies bei der Untersuchung von Reduktionsketten im  $\lambda$ -Kalkül getan haben, können wir nun nachweisen, daß die Relation

$$\{(u, v) \mid u \xrightarrow{R} v\}$$

primitiv rekursiv ist. Daraus folgt aber ebenso unmittelbar, daß die Relation

$$\{(u, v) \mid u \xrightarrow{R} v\}$$

rekursiv aufzählbar ist.

Also gilt:

**Satz 53** Für eine CHOMSKY Sprache mit zugrunde liegender Grammatik  $\mathcal{G} = (\Sigma, S, \Omega, R)$  ist die Menge

$$[\mathcal{L}_G] := \{w \mid w \in \Omega^* \wedge S \xrightarrow{R} w\}$$

eine rekursiv aufzählbare Menge.

Damit haben wir die Komplexität von CHOMSKY Sprachen bereits eingeschränkt. Die weitergehende Frage ist nun, ob die CHOMSKY Sprachen nicht vielleicht sogar ein lösbares Wortproblem haben.

Wie es die Allgemeinheit der in CHOMSKY Sprachen erlaubten Termersetzungen bereits erwarten läßt, wird dies nicht der Fall sein. Um dies einzusehen, beschreibt man eine Turingmaschine mit Hilfe einer CHOMSKY Grammatik. Technisch nützlich ist hierbei die folgende Eigenschaft

von CHOMSKY Sprachen.

**Satz 54** Ist  $\mathcal{G} = (\Sigma, S, \Omega, R)$  eine CHOMSKY Grammatik,  $\Theta \subseteq \Sigma$  ein Alphabet und  $Q$  ein Semi-THUE-System, so ist auch die Sprache

$$\mathcal{L}_{\text{comp}} := \{w \in \Theta^* \mid (\exists v \in \mathcal{L}_{\mathcal{G}})[v \xrightarrow{Q} w]\}$$

eine CHOMSKY Sprache.

Eine Konfiguration einer Turingmaschine  $T$  über dem einelementigen Alphabet  $\{\star\}$  lässt sich in der Form

$$u \underset{n}{\overset{z}{\pi}} v$$

Beschreibung einer Turingmaschine durch eine CHOMSKY Grammatik

notieren, wobei das Wort

$$uzv$$

über dem Alphabet  $\{\star, B\}$  den endlichen Teil der Bandinschrift (mit dem Zeichen  $z$  auf dem Arbeitsfeld) darstellt und  $n$  der Zustand der Maschine ist. Eine derartige Konfiguration können wir als Wort über dem Alphabet

$$\Sigma = \{\star, B, \alpha, \beta\} \cup M(T)$$

darstellen, wobei

$$\alpha u \beta n z v \alpha$$

mit  $u, v \in \{\star, B\}^*$ ,  $n \in M(T)$  und  $z \in \{\star, B\}$  die Konfiguration

$$u \underset{n}{\overset{z}{\pi}} v$$

repräsentiert. Nun ist es ein Leichtes, eine CHOMSKY Grammatik anzugeben, mit der sich alle Startkonfigurationen erzeugen lassen. Man definiert eine Grammatik

$$\mathcal{G} = (\{S, \alpha, \beta, B, \star, k_T\}, S, \{\alpha, \beta, B, \star, k_T\}, R_1),$$

mit dem Semi-THUE System

$$R_1 := \left\{ \begin{array}{l} S \xrightarrow{R_1} \alpha \beta k_T \alpha \\ \beta k_T \xrightarrow{R_1} \beta k_T z \quad \text{für } z \in \{\star, B\} \end{array} \right\}.$$

Dann gibt man ein Semi-THUE System  $R_2$  an, das die Modifikations- und Testinstruktionen der Turingbasismaschine nachmacht. Das System  $R_2$  ist in Abbildung 5.2 - 1 dargestellt. Damit lässt sich nach Satz 54 zu jeder Turingmaschine  $T$  mit Endzustand  $e_T$  eine CHOMSKY Grammatik  $\mathcal{G}$  so angeben, daß für

$$\text{Res}_T(n_1, \dots, n_k) \simeq m$$

## 5. Unentscheidbare Probleme

---

Instruktion	Wirkung	Regel
$(n, PRT, p)$	$\begin{array}{l} u \frac{a}{\pi} v \rightarrow u \frac{\star}{p} v \\ \alpha u \beta n a v \alpha \rightarrow \alpha u \beta p \star v \alpha \end{array}$	$\beta n a \rightarrow_1^{R_2} \beta p \star$ für $a \in \{\star, B\}$
$(n, CLR, p)$	$\begin{array}{l} u \frac{a}{\pi} v \rightarrow u \frac{B}{\pi} v \\ \alpha u \beta n a v \alpha \rightarrow \alpha u \beta p B v \alpha \end{array}$	$\beta n a \rightarrow_1^{R_2} \beta p B$ für $a \in \{\star, B\}$
$(n, LFT, p)$	$\begin{array}{l} u z \frac{a}{\pi} v \rightarrow u z \frac{a}{p} v \\ \alpha u \beta n a v \alpha \rightarrow \alpha u \beta p z v \alpha \end{array}$	$z \beta n \rightarrow_1^{R_2} \beta p z$ für $z \in \{\star, B\}$
$(n, RHT, p)$	$\begin{array}{l} u \frac{a}{\pi} z v \rightarrow u a \frac{z}{p} v \\ \alpha u \beta n a v \alpha \rightarrow \alpha u \beta p z v \alpha \end{array}$	$z \beta n \rightarrow_1^{R_2} \beta p z$ für $z \in \{\star, B\}$
$(n, BEQ, p, q)$	$\begin{array}{l} u \frac{B}{\pi} v \rightarrow u \frac{B}{q} v \\ \alpha u \beta n B v \alpha \rightarrow \alpha u \beta q B v \alpha \\ u \frac{\star}{\pi} v \rightarrow u \frac{\star}{p} v \\ \alpha u \beta n \star v \alpha \rightarrow \alpha u \beta p \star v \alpha \end{array}$	$\begin{array}{l} \beta n B \rightarrow_1^{R_2} \beta q B \\ \beta n \star \rightarrow_1^{R_2} \beta p \star \end{array}$

Abb. 5.2 - 1: Das Regelsystem  $R_2$  für die Basisturingmaschine

---

das Wort

$$\alpha u \beta e_T \star^m B v \alpha$$

ein Element der von  $\mathcal{G}$  erzeugten CHOMSKY Sprache ist. Wir benötigen daher nur noch eine Ausgabe, die sich aber ebenso einfach durch ein Semi-THUE System verwirklichen lässt. Wir erweitern zunächst das Alphabet um eine Symbol  $\gamma$  und fügen das Regelsystem

$$R_3 := \{\beta e_T \rightarrow_1 \gamma\}$$

hinzzu. Dann ist die Sprache

$$\{w \in \{\alpha, \gamma, \star, B\}^* \mid (\exists v \in \mathcal{L}_G)[v \rightarrow^G w]\}$$

eine CHOMSKY Sprache, die genau die Wörter der Form

$$\alpha u \gamma \star^m B v \alpha w \tag{5.1}$$

erzeugt, für die

$$u \underline{z} \star^m B v$$

eine Endkonfiguration ist. Definieren wir nun ein Regelsystem vermöge

$$R_4 = \left\{ \begin{array}{l} Bz \rightarrow_1 B \text{ für } z \in \{\star, B\} \\ B\alpha \rightarrow_1 \Lambda \\ \alpha z \rightarrow_1 \alpha \text{ für } z \in \{\star, B\} \\ \alpha\gamma \rightarrow_1 \Lambda \end{array} \right\},$$

so sehen wir sofort, daß dieses Regelsystem die Wörter in (5.1) in

$\star^m$

überführt. Damit haben wir den folgenden Satz erhalten:

**Satz 55** Zu jeder Turingmaschine  $T$  gibt es eine CHOMSKY Sprache  $\mathcal{L}_T$  derart, daß

$$\mathcal{L}_T = \{\star^m \mid (\exists n_1) \dots (\exists n_k) [Res_T(n_1, \dots, n_k) \simeq m]\}$$

gilt.

Ist nun  $M$  eine nicht leere rekursiv aufzählbare Menge, so gibt es eine Turingmaschine  $T$  mit

$$M = \{m \mid (\exists n) [Res_T(n) \simeq m]\}.$$

Mit Satz 54 gibt es daher eine CHOMSKY Sprache  $\mathcal{L}_T$  mit

$$\mathcal{M}_T = \{\star^m \mid m \in M\}.$$

Der Kode  $\lceil \star^m \rceil$  berechnet sich natürlich primitiv rekursiv aus  $m$ . Wir haben daher gezeigt:

**Satz 56** Es gibt eine primitiv rekursive Funktion  $f$  und zu jeder rekursiv aufzählbaren Menge  $M$  eine CHOMSKY Sprache  $\mathcal{L}$  derart, daß

$$m \in M \iff f(m) \in \lceil \mathcal{L} \rceil := \{w^r \mid w \in \mathcal{L}\}$$

und somit

$$M \leq_m \lceil \mathcal{L} \rceil$$

gilt.

Wählen wir für  $M$  in Satz 56 die Menge  $K$ , so erhalten wir eine CHOMSKY Sprache  $\mathcal{L}_K$  mit  $K \leq_m \mathcal{L}_K$ . Wegen der m-Vollständigkeit von  $K$  gilt mit Satz 53 aber auch  $\mathcal{L}_K \leq_m K$ . Also haben wir

$$K \equiv_m \mathcal{L}_K,$$

was zeigt, daß die CHOMSKY Sprachen ebenso kompliziert wie die rekursiv aufzählbaren Mengen sind. Als unmittelbare Folgerung daraus wollen wir festhalten:

## 5. Unentscheidbare Probleme

---

**Satz 57** Es gibt CHOMSKY Sprachen mit unlösbarem Wortproblem.

Die eben erwähnte Sprache  $\mathcal{L}_K$  ist ein Beispiel für eine solche.

Wir können sogar noch weiter gehen und folgern, daß die Frage

$$u \rightarrow^Q v?$$

für beliebige Semi-THUE Systeme  $Q$  unentscheidbar ist, falls das zugrunde gelegte Alphabet wenigstens zwei Zeichen hat.

Um dies einzusehen, starten wir mit einer CHOMSKY Grammatik  $\mathcal{G} = (\Gamma, S, \Omega, R)$  mit unlösbarem Wortproblem. Wir nehmen an, daß das zugrunde liegende Alphabet  $\Sigma$  mindestens zwei verschiedene Zeichen  $*$ ,  $B$  enthält. Wir wollen weiter annehmen, daß  $\Gamma = \{0, \dots, r\}$  ist. Durch

$$\begin{aligned} k: \Gamma &\longrightarrow \Sigma^* \\ j &\longmapsto B *^j B \end{aligned}$$

definieren wir eine Kodierung, die sich kanonisch zu einer Kodierung

$$k: \Gamma^* \longrightarrow \Sigma^*$$

fortsetzen läßt. Das Semi-THUE System  $R$  der Grammatik  $\mathcal{G}$  übeträgt sich nun zu einem Semi-THUE System

$$k(R) := \{k(u) \rightarrow_1 k(v) \mid u \rightarrow_1^R v\}.$$

Dann gilt

$$k(u) \rightarrow^{k(R)} k(v) \iff u \rightarrow^R v,$$

und, da  $\mathcal{G}$  ein unlösbares Wortproblem hatte, kann man  $k(u) \rightarrow^{k(R)} k(v)$  nicht entscheiden. Daraus folgt aber, daß

$$u \rightarrow^Q v$$

für beliebige Semi-THUE Systeme  $Q$  über  $\Sigma$  nicht global entscheidbar sein kann.

Programmiersprachen (mit wohldefinierter Syntax) sind natürlich CHOMSKY Sprachen. Es liegt allerdings auf der Hand, daß Programmiersprachen mit unentscheidbarem Wortproblem unbrauchbar sind. Um zu einer Theorie von Programmiersprachen zu gelangen, hat man daher die Regeln für Sprachgrammatiken enger zu fassen, als man dies im allgemeinen Fall der CHOMSKY Sprachen getan hat. Dies führt zur sogenannten CHOMSKY Hierarchie von Sprachen. Grammatiken der Art, wie wir sie bisher betrachtet haben, heißen dann Typ 0 Grammatiken. Eine Grammatik  $\mathcal{G} = (\Sigma, S, \Omega, R)$  heißt eine *Typ 1* oder *kontextsensitive* Grammatik, wenn jede Regel des Semi-THUE Systems die Form

$uav \rightarrow_1^R uwv$  für  $a \in \Sigma \setminus \Omega$ ,  $u, v \in \Sigma^*$  und  $w \in \Sigma^+$

hat. Hat jede Regel die Gestalt

$a \rightarrow_1^R w$  für  $a \in \Sigma \setminus \Omega$  und  $w \in \Sigma^+$ ,

so spricht man von einer *Typ 2* oder *kontextfreien* Grammatik. *Typ 3* oder *reguläre* Grammatiken erhält man, wenn man nur Regeln der Gestalt

Kontextfreie und  
reguläre Sprachen

$a \rightarrow_1^R bt$  für  $a, b \in \Sigma \setminus \Omega$  und  $t \in \Omega$

zuläßt.

Es ist dann aus der Definition sofort klar, daß

$\text{Typ}i \subseteq \text{Typ}j$  für  $j \leq i$

gilt.

Typ 0 Sprachen haben, wie wir bereits gesehen haben, ein unlösbares Wortproblem. Wie steht es damit für Typ 1 Sprachen? Hier beobachtet man zunächst, daß aus  $u \rightarrow_1^R v$  immer  $l(u) \leq l(v)$  folgt, wobei  $l(u)$  wieder die Länge des Wortes  $u$  bedeutet. Daraus erhalten wir sofort

$$u \rightarrow^R v \implies l(u) \leq l(v).$$

Ist das Wortproblem  
für kontextsensitive  
Sprachen lösbar?

Wir wollen uns die Komplexität von

$$u \rightarrow_n^R v$$

als Relation von  $n$  und (den Gödelnummern von)  $u, v$  ansehen. Es gilt

$$u \rightarrow_0^R v \iff u = v$$

und

$$u \rightarrow_{n+1}^R v \iff (\exists w)[u \rightarrow_n^R w \wedge w \rightarrow_1^R v]. \quad (5.2)$$

Gelingt es, den Existenzquantor in (5.2) zu beschränken, so haben wir  $u \rightarrow_n^R v$  durch primitive Rekursion nach  $n$  definiert. Für ein Wort  $w$  mit  $w \rightarrow_1^R v$  gilt aber  $l(w) \leq l(v)$  und  $l(v)$ , die Länge von  $v$ , läßt sich natürlich in primitiv rekursiver Weise aus dem Kode ' $v$ ' von  $v$  berechnen. Über einem endlichen Alphabet gibt es aber nur endlich viele Wörter beschränkter Länge, und eine Oberschranke für die Kodeteile von Wörtern einer Länge  $\leq l$  läßt sich offensichtlich primitiv rekursiv aus  $l$  berechnen. Der Existenzquantor in (5.2) ist somit beschränkbar und die Relation  $u \rightarrow_n^R v$  damit primitiv rekursiv entscheidbar.

Wir wollen daraus folgern, daß kontextsensitive (und damit erst recht kontextfreie und reguläre) Sprachen ein entscheidbares Wortproblem haben. Ist  $\mathcal{G} = (\Sigma, S, \Omega, R)$  eine kontextsensitive Grammatik, so

gilt

$$w \in \mathcal{L}_G \iff (\exists n)[S \xrightarrow{n}^R w \wedge w \in \Omega^*]. \quad (5.3)$$

Die Relation  $S \xrightarrow{n}^R w$  haben wir schon als primitiv rekursiv erkannt. Wegen

$$w^n \in [\Omega^*] \iff Seq(w^n) \wedge (\forall i < lh(w^n))[(w^n)_i \in [\Omega]],$$

ist auch  $w \in \Omega^*$  primitiv rekursiv entscheidbar. Es genügt daher zu zeigen, daß sich der Existenzquantor in (5.3) beschränken läßt. Gilt

$$S = w_1 \xrightarrow{1}^R \dots \xrightarrow{1}^R w_i \xrightarrow{1}^R \dots \xrightarrow{1}^R w_n = w,$$

so dürfen wir davon ausgehen, daß alle  $w_i$  für  $i = 1, \dots, n$  verschieden sind. Nehmen wir an, daß  $\Sigma$   $k$ -viele Zeichen besitzt, so gibt es nur  $k^m$ -viele Wörter der Länge  $m$ . Nun haben wir aber für  $i = 1, \dots, n$  alle Wörter  $w_i$  Längen, die kleiner oder höchstens gleich der Länge von  $w$  sind. Da es jedoch nur  $\sum_{j \leq l(w)} k^j$ -viele verschiedene Wörter mit Längen  $\leq l(w)$  gibt, liefert  $\sum_{j \leq l(w)} k^j$ , was sich primitiv rekursiv aus dem Kode von  $w$  berechnen läßt, eine Schranke für den Existenzquantor in (5.3).

Zusammenfassend haben wir also den folgenden Satz:

Das Wortproblem  
für kontextsensitive  
Sprachen ist primitiv  
rekursiv

Formale Sprachen

**Satz 58** Alle CHOMSKY Sprachen mit Typen größer 0 haben ein primitiv rekursiv entscheidbares Wortproblem.

Satz 58 zeigt die *prinzipielle* Möglichkeit, Parser für CHOMSKY Sprachen zu bauen, wobei wir unter einem Parser eine Maschine verstehen, die das Wortproblem der Sprache entscheidet. Natürlich ist Satz 58 noch keine wirkliche Hilfe für den praktischen Bau von Parsern. Die Entscheidung von primitiv rekursiven Relationen kann immer noch soviel Rechenaufwand erfordern, daß sie von keinem Rechner praktisch zu leisten ist. Deshalb sind hier weitergehende Untersuchungen anzustellen, für welche Sprachtypen effiziente Algorithmen zur Worterkennung vorliegen. Intuitiv ist klar, daß das Wortproblem für eine kontextsensitive Sprache schwerer zu entscheiden sein sollte, als für eine kontextfreie oder gar reguläre Sprache. Diese Untersuchungen führen zur *Theorie der formalen Sprachen*, die wir hier allerdings nicht weiter vertiefen können.

### 5.3 Prädikatenlogik

Als nächstes Beispiel wollen wir das Entscheidungsproblem für die Prädikatenlogik behandeln. Wir können hier die Prädikatenlogik allerdings nur sehr grob skizzieren. Eingehendere Details darüber finden sich beispielsweise in Band 1.1 dieser Serie.

Das Alphabet der Sprache der Prädikatenlogik besteht aus zwei Komponenten, den *logischen* und den *nichtlogischen Zeichen*. Die logischen Zeichen ihrerseits untergliedern sich in Zeichen für

- *Variablen* –  $x, y, z, x_1, \dots$ ,
- *Junktoren* –  $\neg, \wedge, \vee, \rightarrow$

und

- *Quantoren* –  $\forall, \exists$ .

Die nichtlogischen Zeichen untergliedern sich in

- *Individuenkonstanten* –  $c, c_1, \dots$ ,
- *Funktionszeichen* –  $f, g, f_1, \dots$

und

- *Prädikatszeichen* –  $P, Q, P_1, \dots$ .

Jedem Funktionszeichen  $f$  und jedem Prädikatszeichen  $P$  ist eine Stellenzahl  $\#f$  beziehungsweise  $\#P$  zugeordnet.

Daneben benutzen wir noch runde Klammern als Hilfszeichen.

Es gibt zwei Arten sinnvoll gebildeter Wörter in der Sprache der Prädikatenlogik, nämlich *Terme* und *Formeln*. Diese wollen wir hier allerdings nicht über eine formale Grammatik definieren, sondern durch intuitiv besser verständliche Bildungsregeln beschreiben.

Die Bildungsregeln für Terme werden durch die folgenden Klauseln gegeben:

Terme

1. Jede Variable und jede Individuenkonstante ist ein Term.
2. Sind  $t_1, \dots, t_n$  Terme, und ist  $f$  ein Funktionszeichen der Stellenzahl  $n$ , so ist  $(ft_1 \dots t_n)$  ein Term.

Die Bildungsregeln für Formeln umfassen die folgenden Klauseln:

Formeln

1. Sind  $t_1, \dots, t_n$  Terme, und ist  $P$  ein Prädikatszeichen der Stellenzahl  $n$ , so ist  $(Pt_1 \dots t_n)$  eine (Atom-)Formel.
2. Sind  $A$  und  $B$  Formeln, so sind  $\neg A$ ,  $A \wedge B$ ,  $A \vee B$  und  $A \rightarrow B$  ebenfalls Formeln.
3. Ist  $A$  eine Formel, so sind auch  $\forall x A$  und  $\exists x A$  Formeln. Dabei wird  $x$  in  $\forall x A$  bzw.  $\exists x A$  durch den entsprechenden Quantor gebunden und wir wollen annehmen, daß  $x$  in  $A$  nicht bereits durch einen anderen Quantor gebunden war.

Durch diese Regeln ist die *Syntax* der Sprache der Prädikatenlogik festgelegt. Wir wollen noch vereinbaren, daß eine Variable  $x$  frei in einer Formel  $F$  vorkommt, wenn  $x$  in  $F$  nicht gebunden ist. Terme und Formeln ohne freie Variablen heißen *geschlossen*. Geschlossene Formeln bezeichnet man oft auch als *Sätze*. Die logischen Symbole sind für alle Sprachen

Sätze

der ersten Stufe die gleichen. Sprachen der ersten Stufe unterscheiden sich daher nur durch ihre nichtlogischen Symbole. Die nichtlogischen Zeichen legen die *Signatur* der Sprache fest. Wir notieren dies oft durch

Signatur einer Sprache

## 5. Unentscheidbare Probleme

---

$$\mathcal{L} = \mathcal{L}(\mathcal{C}, \mathcal{F}, \mathcal{P}),$$

wobei  $\mathcal{C}$  die Menge der Konstantensymbole,  $\mathcal{F}$  die Menge der Funktionssymbole und  $\mathcal{P}$  die Menge der Prädikatszeichen bedeutet.

Bislang haben wir nur die *Syntax* von Sprachen der ersten Stufe festgelegt. Die Bildungsregeln für Terme und Formeln sind ganz offensichtlich so angelegt, daß wir entscheiden können, ob ein wohlgeformtes Wort vorliegt. Ähnliche Überlegungen, wie wir sie für kontextsensitive Sprachen durchgeführt haben (die Sprache der Prädikatenlogik läßt sich sogar durch eine kontextfreie Grammatik definieren) zeigen, daß diese Entscheidung in primitiv rekursiver Weise gefällt werden kann.

Semantik

Die Bedeutung von Termen und Formeln wird erst durch eine *Semantik* gegeben. Die Semantik soll dabei so festgelegt werden, daß Terme als *Individuen* und Formeln als *Aussagen* interpretiert werden. Um dies tun zu können, benötigen wir eine *Struktur* mit einem *Träger*, dessen Elemente die Rolle der Individuen übernehmen sollen. Um auch die nichtlogischen Zeichen interpretieren zu können, muß die Struktur der Signatur der Sprache angepaßt sein, was hier nur soviel heißen soll, daß eine Struktur für eine Sprache  $\mathcal{L}(\mathcal{C}, \mathcal{F}, \mathcal{P})$  die Form

$$S := (S, C, F, P)$$

hat, wobei

- $C \subseteq S$  für jedes  $c \in \mathcal{C}$  ein  $c^S \in C$ ,
- $F$  für jedes  $f \in \mathcal{F}$  mit  $\#f = n$  eine Funktion  $f^S: S^n \rightarrow S$  und
- $P$  für jedes  $P \in \mathcal{P}$  der Stellenzahl  $\#P = n$  ein  $P^S \subset S^n$

enthält.

Interpretationen von  
Termen und Formeln

Wir wollen nun eine *Interpretation*  $S$  so definieren, daß  $t^S$  für jeden Term  $t$  ein Individuum, d.h. ein Element von  $S$ , und  $F^S$  für jede Formel  $F$  ein Wahrheitswert wird. Für Terme, die keine Variablen enthalten, ist ein solcher Wert offensichtlich bereits durch die Struktur gegeben. (Jede Konstante  $c$  wird durch  $c^S \in S$  interpretiert und für  $(t_1^S, \dots, t_n^S) \in S^n$  und  $f \in \mathcal{F}$  mit  $\#f = n$  ist  $f^S(t_1^S, \dots, t_n^S) \in S$ .) Um auch Terme zu erfassen, die Variablen enthalten, haben wir die Variablen zu *belegen*. Dabei wollen wir unter einer *Belegung*  $\Phi$  über einer Struktur  $S$  einfach eine Abbildung von den Variablen nach  $S$  verstehen. Bezüglich einer festen Belegung  $\Phi$  definieren wir dann den Wert  $t^S[\Phi]$  eines Termes  $t$  durch die Klauseln:

1.  $c^S[\Phi] =: c^S$ ,
2.  $x^S[\Phi] := \Phi(x)$ ,
3.  $(ft_1 \dots t_n)^S[\Phi] := f^S(t_1^S[\Phi] \dots t_n^S[\Phi])$ .

Unmittelbar aus der Definition erhalten wir, daß

Belegungen freier  
Variablen

$$t^S[\Phi] \in S$$

für jeden Term  $t$  und jede Belegung  $\Phi$  über der Struktur  $S$  gilt.

Nachdem wir die Semantik für die Terme fixiert haben, können wir daran gehen, auch Formeln zu interpretieren. Formeln sollen Aussagen werden. Dabei verstehen wir unter Aussagen solche Ausdrücke, die die Wahrheitswerte wahr ( $W$ ) und falsch ( $F$ ) annehmen können. Wir definieren:

1.  $(Pt_1 \dots t_n)^S[\Phi] = W : \iff (t_1^S[\Phi] \dots t_n^S[\Phi]) \in P^S$
2.  $(\neg A)^S[\Phi] = W : \iff A^S[\Phi] = F$
3.  $(A \wedge B)^S[\Phi] = W : \iff A^S[\Phi] = W \text{ und } B^S[\Phi] = W$
4.  $(A \vee B)^S[\Phi] = W : \iff A^S[\Phi] = W \text{ oder } B^S[\Phi] = W$
5.  $(\forall x A)^S[\Phi] = W : \iff A^S[\Psi] = W \text{ für jede Belegung } \Psi, \text{ die auf den von } x \text{ verschiedenen Variablen mit } \Phi \text{ übereinstimmt.}$
6.  $(\exists x A)^S[\Phi] = W : \iff A^S[\Psi] = W \text{ für eine Belegung } \Psi, \text{ die auf den von } x \text{ verschiedenen Variablen mit } \Phi \text{ übereinstimmt.}$

Anstelle von  $F^S[\Phi] = W$  schreibt man gewöhnlich

$$S \models F[\Phi]$$

und sagt, daß die Struktur  $S$  die Formel  $F$  bei der Belegung  $\Phi$  erfüllt. Offensichtlich hängt die Interpretation eines Termes  $t$  oder einer Formel  $F$  nur von den endlich vielen Werten  $\Phi(x)$  ab, für die  $x$  in  $t$  beziehungsweise  $F$  auch tatsächlich frei auftritt. Man schreibt daher auch manchmal

Die Erfüllungsrelation

$$S \models F[s_1, \dots, s_n],$$

wenn  $s_1, \dots, s_n$  Elemente des Trägers von  $S$  sind und aus dem Zusammenhang klar wird, wie die endlich vielen in  $F$  frei auftretenden Variablen durch  $s_1, \dots, s_n$  zu belegen sind.

Insbesondere haben geschlossene Formeln (das sind Formeln ohne freie Variablen) in jeder Struktur eine von Belegungen unabhängigen Wahrheitwert. Aus diesem Grund bezeichnet man geschlossenen Formeln auch als *Sätze*.

Die Klauseln 5. und 6. in der obigen Definition muten auf den ersten Blick etwas merkwürdig an. Dennoch treffen sie die intendierte Bedeutung. Die Formel  $A$  hat nämlich gegenüber der Formel  $\forall x A$  bzw.  $\exists x A$  höchstens die zusätzliche freie Variable  $x$ . Sind  $x_1, \dots, x_n$  alle in  $A$  frei auftretenden Variablen und gilt  $\Phi(x_i) = s_i$  für  $i = 1, \dots, n$ , so lesen sich die obigen Klauseln wie:

$S \models (\forall x A)[\Phi]$  genau dann, wenn  $S \models A[s_1, \dots, s_n, s]$  für alle  $s \in S$

und

$S[\Phi] = (\exists x A)[\Phi]$  genau dann, wenn  $S \models A[s_1, \dots, s_n, s]$  für ein  $s \in S$ .

## 5. Unentscheidbare Probleme

---

Die Quantoren  $\forall$  (für alle) und  $\exists$  (es gibt ein) werden daher tatsächlich in der intendierten Art interpretiert.

Alternative  
Definition der Erfüllbarkeitsrelation

In der Literatur findet man oft auch einen anderen Zugang, bei dem man zu einer vorgegebenen  $\mathcal{L}$ -Struktur  $S$  mit Träger  $S$  die Sprache  $\mathcal{L}$  zur Sprache  $\mathcal{L}_S$  und die Struktur  $S$  zur Struktur  $S_S$  erweitert, indem man für jedes Element  $s \in S$  eine Konstante  $\underline{s}$  hinzunimmt, deren Interpretation durch

$$\underline{s}^{S_S} := s$$

gegeben ist. Für  $\mathcal{L}_S$ -Formeln definiert man dann  $A^{S_S}[\Phi]$  wie oben, aber unter Abänderung der Klauseln 5. und 6. zu

$$5. (\forall x A)^{S_S}[\Phi] = W : \iff A_x(\underline{s})^{S_S}[\Phi] = W \text{ für alle } s \in S$$

$$6. (\exists x A)^{S_S}[\Phi] = W : \iff A_x(\underline{s})^{S_S}[\Phi] = W \text{ für ein } s \in S.$$

Es ist nicht allzu schwer einzusehen, daß dann

$$S \models F[\phi] \iff S_S \models F[\Phi]$$

für alle  $\mathcal{L}$ -Formeln  $F$  gilt.

Haben wir nun eine (nicht notwendigerweise endliche) Menge  $M$  von Formeln einer Sprache der ersten Stufe vorliegen, so vereinbaren wir die folgenden Redeweisen:

- $M$  heißt erfüllbar in einer Struktur  $S$ , wenn es eine Belegung  $\Phi$  über  $S$  gibt, so daß  $S \models A[\Phi]$  für alle Formeln  $A \in M$  gilt.
- $M$  heißt erfüllbar oder konsistent, wenn es eine Struktur  $S$  gibt, derart daß  $M$  erfüllbar in  $S$  ist.
- $M$  heißt allgemeingültig oder kurz gültig, wenn  $M$  in jeder Struktur gültig ist.

Logisches Folgern

Wichtig für die Mathematik ist der Begriff der *logischen Folgerung*. Schon im Altertum versuchte man die Gesetze des logischen Schließens zu erforschen. Dieses Bestreben zieht sich wie ein roter Faden nicht nur durch die Geschichte der Mathematik, sondern auch durch die Philosophie. Um eine befriedigende mathematische Präzisierung des logischen Schließens zu erhalten, bedienen wir uns der Sprache der ersten Stufe und treffen die folgende Definition:

**Definition 59** Eine Formel  $A$  folgt aus einer Formelmenge  $M$ , wenn für jede Struktur  $S$  und jede Belegung  $\Phi$  über  $S$ , die  $M$  erfüllt, auch  $S \models A[\Phi]$  gilt. Wir notieren dies durch

$$M \models A.$$

Eine Formel  $A$  ist offenbar genau dann eine Folgerung aus der leeren Menge, die von jeder Struktur erfüllt wird, wenn sie allgemeingültig ist. Wir notieren die Allgemeingültigkeit von Formeln  $A$  deshalb als

$\models A$ .

Intuitiv ist klar, daß man mit Definition 59 tatsächlich den Folgerungsbegriff erfaßt hat, wie er in der Mathematik verwendet wird. Durch diese Definition der logischen Folgerung wird sichergestellt, daß man durch logisches Schließen aus der Menge der in einer Struktur gültigen Formeln nicht herausgeführt wird.

Doch auch in der Informatik ist der Begriff der logischen Folgerung von entscheidender Bedeutung. Datenverarbeitung bedeutet ja viel mehr als die bloße numerische Berechnung von Funktionen. Datenverarbeitung soll nicht nur das Erfassen und Ordnen von Daten ermöglichen, sondern auch die Möglichkeit bieten, aus vorhandenen Datenbeständen in automatisierter Weise neue Daten zu gewinnen. Dabei liegen Daten nicht immer in der Form vor, die eine numerische Weiterverarbeitung erlaubt. Oft steht man vor der Tatsache, daß ein gewisses Datum entweder richtig oder falsch sein kann, also durch einen Wahrheitswert charakterisiert ist. Gewinnung neuer Daten bedeutet dann, daß aus den vorhandenen Daten logische Schlüsse zu ziehen sind. Maschinell kann dies nur dann möglich sein, wenn sich ein Algorithmus für das logische Schließen angeben läßt.

Nach dem *Kompaktheitssatz* der Prädikatenlogik läßt sich jede Folgerung aus einer unendlichen Formelmenge  $M$  bereits aus einer endlichen Teilmenge von  $M$  folgern. Bei den eben geschilderten Anwendungen in der Informatik kann man von vorneherein von einer endlichen Menge  $M$  ausgehen. Für eine endliche Menge  $M = \{F_1, \dots, F_n\}$  bedeutet  $M \models F$  nach Definition 59 aber, daß die Formel

$$F_1 \wedge \dots \wedge F_n \wedge \neg F$$

unerfüllbar oder, gleichwertig dazu, die Formel

$$F_1 \wedge \dots \wedge F_n \rightarrow F$$

allgemeingültig ist. Wir benötigen also ein automatisierbares Verfahren zur Erzeugung aller allgemeingültigen Formeln. Wenn wir betrachten, was die Allgemeingültigkeit einer Formel  $F$  bedeutet, nämlich

$$S \models F[\Phi]$$

für alle Strukturen  $S$  und alle Belegungen  $\Phi$  über  $S$ , so erscheint dies zunächst völlig hoffnungslos zu sein. Dennoch hatte man bereits seit dem Altertum vermutet, daß sich, ähnlich wie das numerische Rechnen, auch das logische Schließen kalkülisieren läßt. Den Beweis für diese Tatsache hat KURT GÖDEL im Jahre 1930 erbracht, indem er den *Vollständigkeitssatz* für die Prädikatenlogik erster Stufe bewies, dem zufolge ein Kalkül existiert, der alle allgemeingültigen Formeln erzeugt. Da ein Kalkül ein

Logisches Folgern in der Informatik

Logische Folgerung vs.  
Allgemeingültigkeit

## 5. Unentscheidbare Probleme

---

effektives, und damit nach der CHURCHschen These auch rekursives Verfahren zur Aufzählung aller gültigen Formel liefert, können wir den Vollständigkeitssatz in der Sprache der Berechenbarkeitstheorie wie folgt formulieren.

Der  
Vollständigkeitssatz  
der Prädikatenlogik

**Satz 60 (Vollständigkeitssatz der Prädikatenlogik)** Die Menge

$$\{\vdash \phi \mid \models \phi\}$$

der Gödelnummern allgemeingültiger Formeln ist rekursiv aufzählbar.

Dabei können wir wieder voraussetzen, daß die Gödelisierung so durchgeführt ist, daß sich  $\phi$  aus  $\vdash \phi$  in ähnlich effektiver Weise zurückgewinnen läßt, wie dies beispielsweise für partiellrekursive Funktionen und ihre Kodes der Fall war.

Für den Mathematiker wird die Freude über den Vollständigkeitssatz jedoch deutlich getrübt. Dies liegt daran, daß man sich in der Mathematik nicht so sehr für die logisch gültigen Sätze, sondern für Strukturen interessiert. So ist bereits die Theorie der natürlichen Zahlen – das ist die Menge der (Gödelnummern der) Sätze, die in der Struktur  $(\mathbb{N}, \{0, 1\}, \{+, \cdot\}, \{=\})$  der natürlichen Zahlen gelten – eine Menge, deren Komplexität die aller Mengen der arithmetischen Hierarchie übersteigt und die damit weit entfernt von jeder rekursiv aufzählbaren Menge ist. Diese *Unvollständigkeit* logischer Kalküle wurde ebenfalls erstmals von KURT GÖDEL bewiesen.

Der Vollständigkeitssatz stellt sicher, daß sich die Frage nach den Folgerungen aus einer endlichen Formelmenge zumindest positiv entscheiden läßt. Das heißt, daß sich Programme angeben lassen, die bestätigen, daß eine Formel  $A$  eine Folgerung aus einer Formelmenge  $M$  ist, aber im allgemeinen eine Antwort schuldig bleiben, wenn dies nicht der Fall ist. Die nächste Frage, die sich dann sofort anschließt, ist, ob sich Programme finden lassen, die auch in diesem Fall eine Antwort liefern? Formulieren wir diese Frage in der Sprache der Rekursionstheorie, so lautet sie:

Das Entscheidungsproblem der Prädikatenlogik

Ist die Menge  $\{\vdash \phi \mid \models \phi\}$  rekursiv?

Wir wollen skizzieren, daß die Antwort darauf eine negative ist. Die Idee des Beweisganges beruht darauf, zu jeder Registermaschine  $\mathcal{M}$  eine Formel  $F_{\mathcal{M}, \vec{n}}$  der Prädikatenlogik erster Stufe so anzugeben, daß  $\models F_{\mathcal{M}, \vec{n}}$  genau dann gilt, wenn  $\mathcal{M}$  beim Input  $\vec{n}$  stoppt. Wir wollen diese Formel entwickeln, ohne allerdings Details nachzurechnen. Der an diesen Details interessierte Leser findet sie in praktisch allen Lehrbüchern der mathematischen Logik.

Reduktion des Entscheidungsproblems auf das Halteproblem

Sei also  $\mathcal{M}$  eine  $r$ -Registermaschine. Wir haben uns als erstes zu überlegen, welche Sprache  $\mathcal{L}_{\mathcal{M}}$  wir benötigen, um die Rechnung einer  $r$ -Registermaschine zu beschreiben. Zum Zählen der Rechenschritte

benötigen wir natürliche Zahlen. Da sich natürliche Zahlen durch die Null und die Nachfolgerfunktion darstellen lassen, werden wir also eine Konstante  $O$  für die Null und ein einstelliges Funktionszeichen  $S$  zur Sprache nehmen, dessen intendierte Bedeutung die der Nachfolgerfunktion ist. Weiter benötigen wir neben der Identität = ein zweistelliges Prädikatszeichen  $<$ , dessen intendierte Bedeutung die der Ordnungsrelation auf den natürlichen Zahlen ist. Um die Konfiguration der Registermaschine im  $t$ -ten Rechenschritt zu beschreiben, führen wir ein  $r + 2$ -stelliges Prädikatszeichen  $R$  ein, mit der Intention, daß in einer  $\mathcal{L}_M$ -Struktur  $S$

$$S \models R(u, v, x_1, \dots, x_r)[t, k, m_1, \dots, m_r]$$

genau dann gilt, wenn die Registermaschine beim Input  $\vec{n}$  im  $t$ -ten Rechenschritt gerade die Konfiguration  $(k, m_1, \dots, m_r)$  erreicht hat. In einer beliebigen  $\mathcal{L}_M$ -Struktur kann das natürlich nicht gelten. Um es zu erzwingen, geben wir eine Formel  $F_{M, \vec{n}}$  so an, daß in jeder  $\mathcal{L}_M$ -Struktur  $S$ , die  $F_{M, \vec{n}}$  erfüllt, die nichtlogischen Zeichen von  $\mathcal{L}_M$  ihre intendierte Bedeutung erhalten.

Beginnen wir mit den Axiomen, die die Bedeutung von  $<$ ,  $O$  und  $S$  festlegen. Die Tatsache, daß  $<$  eine lineare Ordnung auf dem Individuenbereich ist, dessen kleinstes Element die Interpretation der Konstanten  $O$  ist, und die Basiseigenschaften der Nachfolgerfunktion  $S$  werden durch die Konjunktion der folgende Axiome beschrieben.

$$\begin{aligned} & \forall x[\neg(x < x)] \\ & \forall x \forall y \forall z[x < y \wedge y < z \rightarrow x < z] \\ & \forall x \forall y[x < y \vee y < x \vee x = y] \\ & \forall x[O = x \vee O < x] \\ & \forall x[x < Sx] \\ & \forall x \forall y[Sx = Sy \rightarrow x = y] \\ & \forall x \forall y[x < y \rightarrow (Sx < y \vee Sx = y)] \end{aligned}$$

Wir definieren  $\underline{0} := O$ ,  $\underline{n+1} := S\underline{n}$  und stellen natürliche Zahlen  $n$  durch  $\underline{n}$  dar. Zu jeder Anweisung  $a \in M$  definieren wir eine Formel  $F_a$ , die die Wirkung von  $a$  beschreibt.

Ist  $a = (m, INC_j, n)$ , so sei

$$F_a \equiv \forall x \forall z_1 \dots \forall z_r [R(x, \underline{m}, z_1, \dots, z_r) \rightarrow R(Sx, \underline{n}, z_1, \dots, Sz_j, \dots, z_r)].$$

Ist  $a = (m, DEC_j, n)$ , so sei

$$\begin{aligned} F_a \equiv & \forall x \forall z_1 \dots \forall z_r [R(x, \underline{m}, z_1, \dots, z_r) \rightarrow \\ & \{ (z_j = O \wedge R(Sx, \underline{n}, z_1, \dots, z_r)) \\ & \vee (\neg(z_j = O) \wedge \exists y[z_j = Sy \wedge R(Sx, \underline{n}, z_1, \dots, y, \dots, z_r)])\}]. \end{aligned}$$

Ist  $a = (m, BEQ_j, p, q)$ , so sei

$$F_a \equiv \forall x \forall z_1 \dots \forall z_r [R(x, \underline{m}, z_1, \dots, z_r) \rightarrow \{ (z_j = 0 \wedge R(\text{S}x, \underline{p}, z_1, \dots, z_r)) \\ \vee (\neg z_j = 0 \wedge R(\text{S}x, \underline{q}, z_1, \dots, z_r)) \}].$$

Bezeichnen wir mit  $k_S$  die Start- und mit  $k_{E(\mathcal{M})}$  die (ohne Einschränkung der Allgemeinheit einzige) Endmarke der Maschine  $\mathcal{M}$ , mit  $F_{\text{Ax}}$  die Konjunktion der oben angeführten Axiome zusammen mit den üblichen Gleichheitsaxiomen und nehmen wir  $(i_1, \dots, i_r) := \text{IN}_{\mathcal{M}}(\vec{n})$  an und setzen außerdem

$$F_{\mathcal{M}, \vec{n}} \equiv F_{\text{Ax}} \wedge R(0, \underline{k_S}, \underline{i_1}, \dots, \underline{i_r}) \wedge \bigwedge_{a \in \mathcal{M}} F_a,$$

so erhalten wir die Äquivalenz der folgenden beiden Aussagen:

(A) *Die Maschine  $\mathcal{M}$  stoppt bei Input  $\vec{n}$ , d.h. es gilt*

$$\exists t [RS_{\mathcal{M}}^t(k_S, i_1, \dots, i_r) \in E(\mathcal{M})].$$

(B) *Die Formel*

$$F_{\mathcal{M}, \vec{n}} \rightarrow \exists x \exists z_1 \dots \exists z_r R(x, \underline{k_{E(\mathcal{M})}}, z_1, \dots, z_r)$$

*ist allgemeingültig.*

Die Behauptung (A) folgt ganz offensichtlich aus der Behauptung (B), wenn wir als  $\mathcal{L}_{\mathcal{M}}$ -Struktur die natürlichen Zahlen mit der intendierten Interpretation der nichtlogischen Zeichen nehmen, insbesondere heißt dies, daß wir

$$R^{\mathbb{N}} := \{(t, k, m_1, \dots, m_r) \mid RS_{\mathcal{M}}^t(k_S, i_1, \dots, i_r) = (k, m_1, \dots, m_r)\}$$

setzen.

Für die Richtung von (A) nach (B) hat man sich zu überlegen, daß für jede  $\mathcal{L}_{\mathcal{M}}$ -Struktur  $S$ , die  $F_{\mathcal{M}, \vec{n}}$  erfüllt, die Beziehung

$$RS_{\mathcal{M}}^t(l, i_1, \dots, i_r) = (k, m_1, \dots, m_r) \Rightarrow S \models R(\underline{t}, \underline{k}, \underline{m_1}, \dots, \underline{m_r})$$

gilt. Dies folgt jedoch durch Induktion nach  $t$  aus der Konstruktion der Formel  $F_{\mathcal{M}, \vec{n}}$ .

Gäbe es nun ein Entscheidungsverfahren für die Allgemeingültigkeit von Formeln der Prädikatenlogik der ersten Stufe, so könnten wir unter Zuhilfenahme der Aussagen (A) und (B) das Halteproblem für Registermaschinen entscheiden. Insbesondere lieferte dies ein Entscheidungsverfahren für die Menge

$$K = \{e \mid \exists x T(e, e, x)\}.$$

Um  $e \in K$  zu entscheiden, konstruieren wir uns die Registermaschine für die Funktion  $\{e\}$ , die wir der Einfachheit halber auch mit  $\{e\}$  bezeichnen wollen, dann die Formel  $F_{\{e\}, e}$  und entscheiden

$$\models F_{\{e\}, e} \rightarrow \exists x \exists z_1 \dots \exists z_r R(x, k_{E(\{e\})}, z_1, \dots, z_r).$$

Nach (A) und (B) ist  $e \in K$  genau dann, wenn die Entscheidung positiv ausfällt. Da wir aber bereits wissen, daß die Menge  $K$  nicht rekursiv ist, erhalten wir den folgenden Satz, der erstmalig von ALONZO CHURCH und unabhängig davon auch von ALAN TURING in der Mitte der dreißiger Jahre bewiesen wurde:

**Satz 61** Die Allgemeingültigkeit von Formeln der Prädikatenlogik der ersten Stufe ist unentscheidbar, d.h. die Menge

Unentscheidbarkeit  
der Prädikatenlogik

$$\{[F] \mid \models F\}$$

ist nicht rekursiv.

Fassen wir die Ergebnisse dieses Abschnittes zusammen, so haben wir erfahren, daß der Begriff des logischen Schließens eng mit dem Begriff der allgemeingültigen Formel zusammenhängt. Die Menge der allgemeingültigen Formeln der Prädikatenlogik der ersten Stufe ist rekursiv aufzählbar. Beide Tatsachen zusammen zeigen, daß es zumindest prinzipiell ein Programm gibt, das die Folgerungen aus einer gegebenen Formelmenge erzeugen kann. Ebenso eröffnet dies die Möglichkeit eines Programmes, das die Allgemeingültigkeit von Formeln bestätigt. Der Beweis des Vollständigkeitssatzes – den wir hier nicht angegeben haben – läßt sich so führen, daß er darüber hinaus noch zeigt, daß der Bestätigungsalgorithmus für eine allgemeingültige Formel auch immer einen formalen Beweis für diese Formel liefert. Dies eröffnet die Möglichkeit des Baues von sogenannten *Theorembeweisern*. Die Grenzen für den Bau dieser Beweiser werden durch die Tatsache gesetzt, daß die Menge der allgemeingültigen Formeln nicht rekursiv ist. Es läßt sich daher kein Programm angeben, das uns auch dann eine vernünftige Antwort liefern kann, wenn die zu untersuchende Formel nicht allgemeingültig ist.

Abschließend soll hier noch angemerkt werden, daß sich für Teilklassen von Formeln der Prädikatenlogik durchaus Entscheidungsverfahren haben finden lassen.



---

## 6. Grundzüge der Komplexitätslehre

Bislang haben wir uns keine Gedanken darüber gemacht, ob die Funktionen, die wir als berechenbar erkannt haben, auch tatsächlich eine Chance besitzen, auf einem physikalisch realisierten Rechner berechnet werden zu können. Bei der Entwicklung der primitiv rekursiven Funktionen hatten wir jedoch bereits darauf hingewiesen, daß schon die Berechnung bestimmter primitiv rekursiver Funktionen die durch physikalische Gesetze gegebenen Grenzen sprengen würde.

Es liegt daher nahe, sich zu fragen, ob sich eine mathematische Theorie der praktisch berechenbaren Funktionen entwickeln läßt. Natürlich ist es so gut wie unmöglich, ein Komplexitätsmaß zu finden, das die Realität genau widerspiegelt und dennoch glatt genug ist, um die Entwicklung einer mathematischen Theorie zu erlauben. So werden wir immer vor der Notwendigkeit stehen, Exponentialfunktionen zu berechnen, obwohl deren Wachstum schnell alle vorstellbaren Grenzen sprengt. In der Praxis werden wir dabei nur immer darauf achten müssen, daß die Argumente nicht zu groß sind.

Anzustreben ist eine mathematische Theorie, die die Beschreibung und das Studium von Funktionen ermöglicht, die sich mit beschränkten Ressourcen berechnen lassen. Es ist klar, daß ein exponentielles Anwachsen des Ressourcenbedarfes eine Funktion praktisch unberechenbar macht. Akzeptabel erscheint es gerade noch, wenn der Ressourcenbedarf nur in polynomialem Weise vom Informationsgehalt des Inputs abhängt. Diese Klasse von Funktionen bezeichnet man allgemein als *polynomial berechenbar*. Da diese Klasse auch noch recht befriedigende Abschlußeigenschaften besitzt, hat es sich eingebürgert, in ihr die Klasse der real berechenbaren Funktionen zu sehen. Natürlich ist dies vom Standpunkt des Praktikers her nur "cum grano salis" zu sehen, da auch ein Polynom, wenn sein Grad nur genügend groß ist, alle praktisch zur Verfügung stehenden Ressourcen sprengen kann.

### 6.1 TIME- und SPACE-beschränkte Funktionen

Ist  $\mathcal{D}_{in}$  eine Datenmenge und  $T: \mathcal{D}_{in} \longrightarrow \mathcal{D}_{out}$  eine Funktion, so sagen wir, daß eine Funktion  $f$  auf einer Basismaschine  $\mathcal{B}$  in  $TIME(T)$  berechenbar ist, wenn es eine  $\mathcal{B}$ -Maschine  $\mathcal{M}$  gibt, so daß

Zeitbeschränkt  
berechenbare  
Funktionen

$$f(d) \simeq Res_{\mathcal{M}}(u)$$

## 6. Grundzüge der Komplexitätslehre

---

und

$$Rz_T(d) \leq T(|d|)$$

für alle  $d \in \mathcal{D}_{in}$  gilt, wobei

$$|\cdot| : \mathcal{D}_{in} \rightarrow \mathbb{N}$$

ein Komplexitätsmaß auf der Datenmenge  $\mathcal{D}_{in}$  ist.

Platzbeschränkt  
berechenbare  
Funktionen

Analog nennt man  $f$  eine über  $\mathcal{B}$  mit  $SPACE(S)$  berechenbare Funktion, wenn  $f$  wie oben und  $S$  analog zu  $T$  gegeben sind, und der Speicherbedarf (was darunter genau zu verstehen ist, wollen wir noch einen Augenblick zurückstellen) der Maschine  $\mathcal{M}$  während der Berechnung von  $f$  nicht größer als  $S(|d|)$  ist.

Anstelle von  $TIME(T)$ - bzw.  $SPACE(S)$ -berechenbaren Funktionen, sprechen wir auch von Funktionen, die durch  $T$  zeit- bzw. durch  $S$  platzbeschränkt berechenbar sind.

Nichtdeterministi-  
sche  
Maschinen

Unter Komplexitätsaspekten wird auch ein Begriff wichtig, den wir bislang nicht betrachtet haben, der einer *nichtdeterministischen Maschine*. Ein Programm (bzw. eine Maschine) heißt *nichtdeterministisch*, wenn verschiedene Anweisungen die gleiche Kennmarke besitzen dürfen. Damit ist die Rechenschrittfunction einer nichtdeterministischen Maschine keine Funktion im üblichen Sinne mehr. Von einer Konfiguration  $(k, m)$  gibt es dann verschiedene Möglichkeiten zu einer Nachfolgekonfiguration zu gelangen, je nachdem, welche der Anweisungen mit Kennmarke  $k$  die Maschine ausführt. Da Programme nach wie vor als endlich vorausgesetzt werden sollen, gibt es natürlich nur endlich viele verschiedene Wahlmöglichkeiten. (Ein Beispiel eines Flußdiagrammes für ein nichtdeterministisches Programm zur Erzeugung von 0, 1-Folgen einer Länge  $\leq i$  findet sich in Abbildung 6.1 - 1.) Wir definieren dann die Rechenschrittfunction als eine mengenwertige Funktion durch

$$\begin{aligned} RS_{\mathcal{M}}(k, m) := & \{(n, \tilde{m}) \mid ((k, s, n) \in \mathcal{M} \wedge s(m) = \tilde{m})\} \cup \\ & \{(n, m) \mid (k, t, n_1, n_0) \in \mathcal{M} \wedge n = n_{t(m)}\}. \end{aligned}$$

Die Rechenschritt-  
funktion  
nichtdeterministi-  
scher  
Maschinen

Die Iteration der Rechenschrittfunction lässt sich dann nicht mehr wie für übliche Funktionen definieren, sondern wir setzen

$$RS_{\mathcal{M}}^0(k, m) := \{(k, m)\}$$

und

$$RS_{\mathcal{M}}^{i+1}(k, m) := \bigcup \{RS_{\mathcal{M}}(\tilde{k}, \tilde{m}) \mid (\tilde{k}, \tilde{m}) \in (RS_{\mathcal{M}}^i(k, m))\}.$$

Ein Rechnung ist dann erfolgreich, wenn es ein  $i$  und ein  $(\tilde{k}, \tilde{m}) \in RS_{\mathcal{M}}^i(k_{\mathcal{M}}, IN(d))$  gibt, derart daß  $\tilde{k}$  eine Endmarke des Programmes ist.

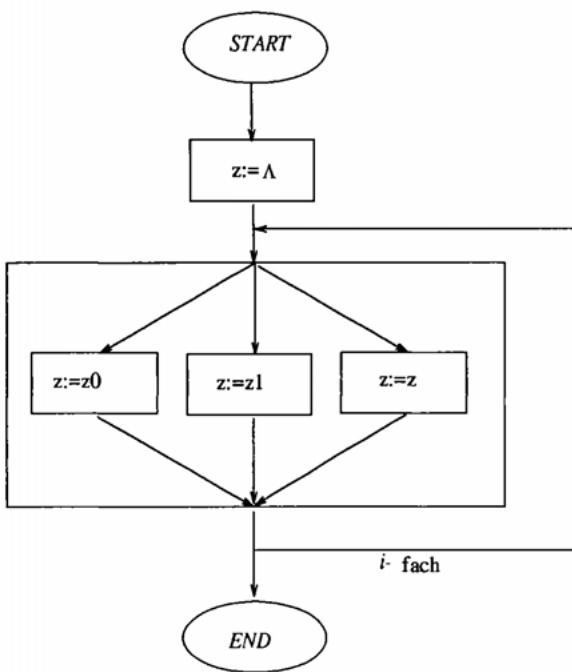


Abb. 6.1 - 1: Ein nichtdeterministisches Programm zur Erzeugung aller 0,1 Folgen einer Länge  $\leq i$

Die Rechenzeit  $Rz_{\mathcal{M}}(d)$  ist dann das kleinste solche  $i$ .

Der Berechnungsverlauf nichtdeterministischer Maschinen wird dann nicht länger linear durch die Iterationen

$$RS_{\mathcal{M}}(k_{\mathcal{M}}, \text{IN}(d)), RS_{\mathcal{M}}^2(k_{\mathcal{M}}, \text{IN}(d)), RS_{\mathcal{M}}^3(k_{\mathcal{M}}, \text{IN}(d)), \dots$$

der Rechenschrittfunktion beschrieben, sondern entspricht vielmehr einem endlich verzweigten Baum, wie er in Abbildung 6.1 - 2 angegeben ist. Eine Berechnung auf einer nichtdeterministischen Maschine terminiert, wenn einer der Fäden des Rechenbaumes eine Endkonfiguration enthält. Die Rechenzeit ist dann die Länge des kürzesten Fadens im Rechenbaum. Das genügt, wenn man sogenannte akzeptierende Maschinen betrachtet. Das sind Maschinen, bei denen es nur darauf ankommt, daß bestimmte Endzustände – eben akzeptierende Zustände – erreicht werden, ohne daß die Speicherinhalte dieser Endkonfigurationen betrachtet werden. Will man Funktionen nichtdeterministisch berechnen, so verlangt man oft, daß die Speicherinhalte aller erreichbaren Endkonfigurationen übereinstimmen. Damit vermeidet man Eindeutigkeitsprobleme für die

Berechnungsverlauf  
in nichtdeterministi-  
schen  
Maschinen

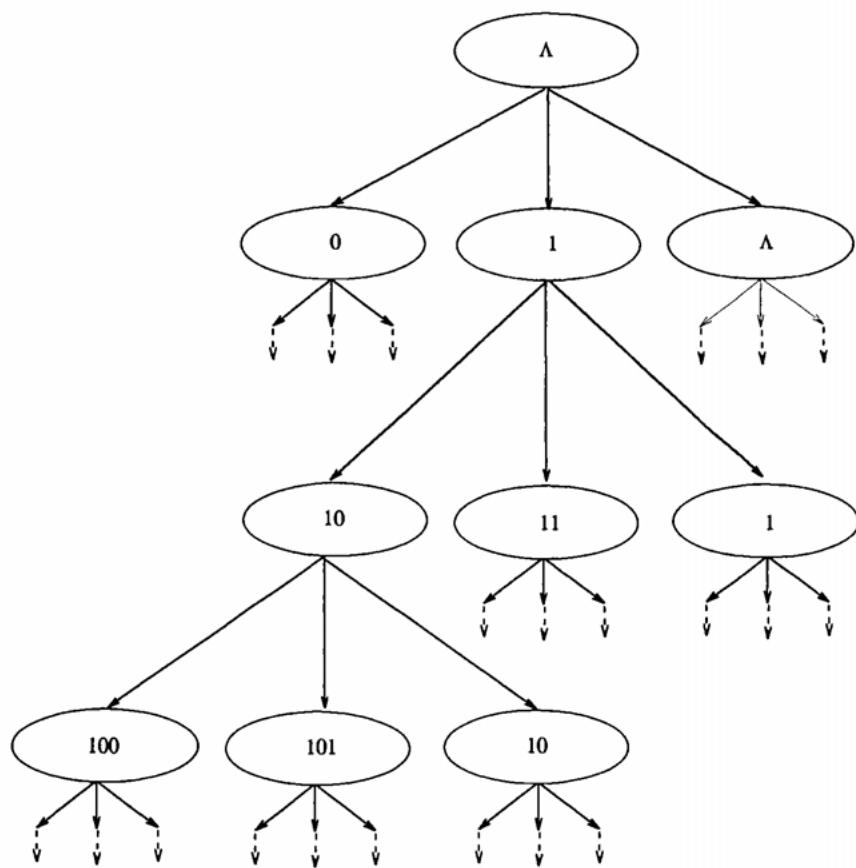


Abb. 6.1 - 2 : Ein Teil des Rechenbaumes des vorhergehenden Programmbeispiels

---

Resultatsfunktionen nichtdeterministischer Maschinen.

Man sollte sich eine nichtdeterministische Maschine als eine Maschine mit unbeschränkt vielen Prozessoren vorstellen, die parallel den Rechenbaum abarbeiten. Der erste der Prozessoren, der eine Endkonfiguration feststellt, stoppt die Rechnung der gesamten Maschine. Es liegt auf der Hand, daß der Begriff einer nichtdeterministischen Maschine bei der Betrachtung prinzipieller Berechenbarkeit nichts Neues bringt. Zu jeder nichtdeterministischen Maschine gibt es offenbar eine deterministische, die das Gleiche leistet. Unter Komplexitätsaspekten mag dies jedoch anders sein, da es intuitiv klar zu sein scheint, daß eine nichtdeterministische Maschine im allgemeinen schneller rechnen können sollte,

als eine deterministische.

Wir haben bei den eben eingeführten Begriffen noch offen gelassen, wie wir den Speicherbedarf einer Berechnung messen wollen. Dies wird im allgemeinen von der Wahl der Basismaschine abhängen. Als eine gute Wahl hat sich hier die Turingbasismaschine über einem beliebigen Alphabet herausgestellt, da sich hier der Speicherbedarf einer Rechnung einfach als die Anzahl der im Verlaufe der Rechnung besuchten Felder definieren lässt. Bei einer Basisregistermaschine wäre die Definition des Speicherbedarfes etwas aufwendiger. Wir wollen daher – in Übereinstimmung mit der gängigen Literatur – von einer Turingbasismaschine ausgehen.

Speicherbedarf einer  
Turingmaschine

Wenn wir uns von der Vorstellung leiten lassen, eine physikalisch realisierbare Maschine beschreiben zu wollen, so spricht nichts dagegen, auch Turingbasismaschinen mit mehr als nur einem Rechenband zu betrachten. Ein derartige Maschine ist physikalisch ebenso leicht zu realisieren wie eine Turingbasismaschine mit nur einem Band. Wir wollen die  $k$ -Band Turingbasismaschine nur informell beschreiben. Sie verfügt über  $k$ -Rechenbänder und jedes Rechenband ist mit einem Schreib-Lesekopf versehen, der genauso arbeitet wie der der 1-Band Maschine. Alle Schreib-Leseköpfe werden von einem gemeinsamen Programm gesteuert. Die Basisinstruktionen sind die gleichen wie bei der 1-Band Maschine, wir haben sie nur mit einem Index zu versehen, der angibt, auf welchem der Bänder sie auszuführen sind. Damit verfügen wir über den Befehlssatz  $LFT_i, RHT_i, PRT_i(z), CLR_i, BEQ_i$ , wobei  $1 \leq i \leq k$  ist und  $z$  die Symbole des zugrundeliegenden Alphabets durchläuft. Ist  $\Sigma$  das Alphabet der Turingmaschine, so bietet sich als Inputdatenmenge, die Menge  $\Sigma^*$  aller Wörter über  $\Sigma$  an. Das kanonische Komplexitätsmaß  $|u|$  für  $u \in \Sigma^*$  ist die Länge von  $u$ .

Mehrband  
Turingmaschinen

Zum Einlesen eines Wortes der Länge  $n$  benötigt eine Turingmaschine mindestens  $n$ -Felder. Um auch die Betrachtung von Funktionen zu ermöglichen, die mit weniger als linearem Speicherbedarf berechenbar sind, ist es üblich, die Turingbasismaschine mit einem Eingabeband zu versehen, von dem die Maschine nur in einer Richtung lesen und auf das es überhaupt nicht schreiben kann.

Wir haben von den Speicher- und Zeitbeschränkungsfunktionen  $S$  und  $T$  verlangt, daß sie natürliche Zahlen in natürliche Zahlen abbilden. Oft ist es jedoch nützlich, auch reellwertige Funktionen zu betrachten, die wir dann durch Verknüpfung mit der Funktion  $\lceil \cdot \rceil$  zu einer Funktion mit Werten in den natürlichen Zahlen machen, wobei  $\lceil r \rceil$  die kleinste ganze Zahl  $\geq r$  bedeutet. Da wir annehmen wollen, daß jede Turingmaschine mindestens ein Feld beschreibt, setzen wir bei Platzschranken

Allgemeine  
Konventionen über  
zeit- bzw.  
platzbeschränkende  
Funktionen

## 6. Grundzüge der Komplexitätslehre

---

$$S(n) := \max\{1, \lceil S(n) \rceil\} \quad (6.1)$$

voraus. Bei Zeitschranken können wir immer davon ausgehen, daß eine Turingmaschine mindestens  $n + 1$  Schritte benötigt, um einen Input der Komplexität  $n$  zu lesen. Daher definieren wir für Zeitschranken

$$T(n) := \max\{n + 1, \lceil T(n) \rceil\}. \quad (6.2)$$

Natürlich sind diese Definitionen richtig zu lesen. Die links definierte Funktion ist als Funktion von den natürlichen Zahlen in die natürlichen Zahlen zu verstehen, wohingegen es sich bei der auf der rechten Seite auftretenden gleichnamigen Funktion um eine reellwertige handeln darf. Es würde die Schreibweise nur unnötig komplizieren, wenn wir unterschiedliche Namen für beide Funktionen einführen würden. Bei Komplexitätsfragen gehen wir immer davon aus, daß die beschränkenden Funktionen den in (6.1) und (6.2) festgelegten Konventionen genügen. Damit macht es beispielsweise Sinn, von einer Zeitbeschränkung von  $\log_2(n)$  zu sprechen, was für  $n = 1$ , bzw.  $n = 2$  gemäß unseren Konventionen die Werte 2 und 3 liefern würde.

Die  
Komplexitätsklassen  
 $DSPACE$ ,  $NSPACE$ ,  
 $DTIME$  und  $NTIME$

Man definiert nun die Komplexitätsklassen

$$DSPACE(S) \text{ und } NSPACE(S)$$

sowie

$$DTIME(T) \text{ und } NTIME(T)$$

von Funktionen, die sich bei einem Input der Länge  $n$  deterministisch ( $DSPACE$ ,  $DTIME$ ) bzw. nichtdeterministisch ( $NSPACE$ ,  $NTIME$ ) mit durch  $S(n)$  beschränktem Speicherbedarf bzw. in durch  $T(n)$  beschränkter Zeit berechnen lassen.

Akzeptierende  
Maschinen

Wir wollen uns nun, wie es in der Komplexitätstheorie üblich ist, auf akzeptierende Maschinen konzentrieren. Eine akzeptierende Turingmaschine hat gewisse ausgezeichnete Endmarken, die *akzeptierenden Endmarken*. Eine Endkonfiguration ist akzeptierend, wenn die dazugehörige Endmarke akzeptierend ist. Ein Wort  $u$  einer Sprache  $\mathcal{L}$  über einem Alphabet  $\Sigma$  wird von einer Turingmaschine  $\mathcal{T}$  akzeptiert, wenn  $u \in \text{dom}(\text{Res}_{\mathcal{T}})$  ist und die beim Input  $u$  erreichte Endkonfiguration akzeptierend ist.

Die Beschränkung auf akzeptierende Turingmaschinen ist so zu verstehen, daß wir nur die Komplexität des Entscheidungsproblems von rekursiven Mengen betrachten, anstatt das allgemeinere (und in der Notation unhandlichere) Problem der Komplexität der Berechnung einer rekursiven Funktion zu betrachten.

Wenn wir Turingmaschinen über beliebigen Alphabeten betrachten, so ist es immer möglich, die SPACE-Komplexität einer Turingmaschine um einen beliebigen konstanten Faktor zu verringern. Alles, was man dazu zu tun hat, ist, eine Folge von Symbolen des Ursprungsalphabets durch ein einzelnes Symbol eines neuen Alphabets zu kodieren. Daraus folgt, daß es bei der Betrachtung von Komplexitätsklassen nicht auf konstante Faktoren ankommt. Dies gilt sowohl für deterministische als auch für nichtdeterministische Maschinen. Für reelle  $r > 0$  haben wir also

$$DSPACE(S) = DSPACE(r \cdot S) \quad (6.3)$$

und

$$NSPACE(S) = NSPACE(r \cdot S). \quad (6.4)$$

In ähnlicher Form läßt sich auch zeigen, daß sich die Anzahl der Arbeitsbänder einer Turingmaschine verringern läßt, ohne daß die Platzkomplexität zunimmt.

Bei den Komplexitätsbetrachtungen für rekursive Mengen kommt es also auf konstante Faktoren nicht mehr an. Man hat für diese Zusammenhänge bequeme Schreibweisen.

**Definition 62** Für eine Funktion  $g: \mathbb{N} \rightarrow \mathbb{N}$  definieren wir:

Die O-Notation

1.  $O(g)$  ist die Menge aller Funktionen  $f: \mathbb{N} \rightarrow \mathbb{N}$  für die es ein positives reelles  $r$  gibt, so daß  $f(n) < r \cdot g(n)$  für fast alle natürlichen Zahlen  $n$  gilt.

Fast alle heißt dabei "alle, bis auf endlich viele Ausnahmen".

2.  $o(g)$  ist die Menge aller Funktionen  $f: \mathbb{N} \rightarrow \mathbb{N}$ , mit  $f(n) < r \cdot g(n)$  für alle positiven reellen  $r$  und fast alle natürlichen Zahlen  $n$ .

3.  $\Omega_\infty(g)$  ist die Menge aller Funktionen  $f$  für die es ein  $r > 0$  so gibt, daß  $f(n) > r \cdot g(n)$  für unendlich viele  $n \in \mathbb{N}$  gilt.

4.  $\omega(g)$  ist die Menge aller Funktionen  $f$ , für die zu jedem positiven reellen  $r$  unendlich viele  $n \in \mathbb{N}$  mit  $f(n) > r \cdot g(n)$  existieren.

5.  $\Theta(g)$  ist die Menge aller Funktionen  $f$  für die positive reelle Zahlen  $r_1$  und  $r_2$  so existieren, daß  $r_1 \cdot g(n) \leq f(n) \leq r_2 \cdot g(n)$  für fast alle  $n \in \mathbb{N}$  gilt.

Bei der Darstellung von Größenordnungen von Funktionen bedient man sich oft der Restklassenschreibweise

$$f = h + O(g),$$

die besagt, daß die Differenzfunktion  $f - h$  ein Element der Menge  $O(g)$  ist. Man findet diese Schreibweise auch für die Nullfunktion  $h$ , was besagt, daß oft  $f = O(g)$  für  $f \in O(g)$  steht. Ähnliche Schreibwei-

## 6. Grundzüge der Komplexitätslehre

---

sen verwendet man auch für die übrigen in Definition 62 eingeführten Mengen.

Es ist leicht einzusehen, daß  $o(g) \subseteq O(g)$  ist und  $f \in o(g)$  genau dann gilt, wenn

$$\lim_{n \rightarrow \infty} \frac{g(n)}{f(n)} = 0$$

ist. Weitere einfache Beziehungen sind

$$f \in \Theta(g) \quad \text{genau dann, wenn} \quad f \in O(g) \quad \text{und} \quad g \in O(f)$$

und

$$f \in \omega(g) \quad \text{genau dann, wenn} \quad \liminf_{n \rightarrow \infty} \frac{g(n)}{f(n)} = 0,$$

wobei  $\liminf_{n \rightarrow \infty}$  den *limes inferior* bedeutet. Ist  $\mathcal{F}$  eine Familie von Funktionen, so schreibt man im allgemeinen  $O(\mathcal{F})$  für  $\cup\{O(f) \mid f \in \mathcal{F}\}$ .

Da wir endlich viele Ausnahmen in einer endlichen Tabelle mit konstantem Speicherbedarf zusammenstellen können, erhalten wir aus (6.3) und (6.4)

$$DSPACE(S) = DSPACE(O(S))$$

und

$$NSPACE(T) = NSPACE(O(T)).$$

Ähnliche Überlegungen lassen sich auch für die Zeitkomplexität durchführen. Unter der technischen Voraussetzung, daß die Zeitbeschränkungsfunktion  $T$  der Bedingung

$$\limsup_{n \rightarrow \infty} \frac{T(n)}{n} = \infty$$

genügt, was kanonischerweise meist erfüllt ist, erhalten wir die Pendants zu (6.3) und (6.4)

$$DTIME(T) = DTIME(r \cdot T) \tag{6.5}$$

und

$$NTIME(T) = NTIME(r \cdot T) \tag{6.6}$$

für alle reellen  $r > 0$ . Das heißt, wir haben auch hier

$$DTIME(T) = DTIME(O(T))$$

und

$$NTIME(T) = NTIME(O(T)).$$

Die Idee in diesem Beweis besteht darin, daß sich zu jedem Turingprogramm  $M_1$  ein Turingprogramm  $M_2$  angeben läßt, das eine vorgegebene Anzahl von Programmschritten von  $M_1$  in einem einzigen Schritt simulieren kann.

Etwas höher ist der Preis, der für eine Reduktion der Anzahl der verwendeten Bänder bezahlt werden muß. So wird eine in der Zeit  $T$  auf einer  $k$ -Band Maschine akzeptierte Sprache auf einer 1-Band Maschine erst in der Zeit  $T^2$  akzeptiert. Bei einer Reduktion auf eine 2-Band Maschine wächst die Zeit lediglich auf  $T \cdot \log(T)$ . Die gleichen Ergebnisse erhält man auch, wenn man nichtdeterministische Programme betrachtet.

Die erste Frage, die sich aufdrängt, ist, ob sich vielleicht bereits alle rekursiven Sprachen mit einer platz- oder zeitbeschränkten Maschine entscheiden lassen. Die Antwort ist erwartungsgemäß ‘nein’, zumindest dann, wenn wir sinnvollerweise nur rekursive Funktionen als Zeit- bzw. Platzschranken zulassen. Dies kann vermöge eines Diagonalarguments eingesehen werden. Natürlich lassen sich auch Indizes für  $k$ -Band Turingmaschinen einführen. Wir gehen ebenso von einer Kodierung des zugrunde liegenden Alphabets aus und wollen ohne Einschränkung Worte und deren Kodes identifizieren. Mit Hilfe der CHURCHSchen These sehen wir ein, daß die Sprache

$$L(e) := \{x \mid \{e\} \text{ akzeptiert } x \text{ nicht in } T(|x|) \text{ Schritten}\},$$

rekursiv ist. Einen Entscheidungsalgorithmus erhalten wir einfach dadurch, daß wir –  $T$  war ja als rekursiv vorausgesetzt –  $T(|x|)$  berechnen und dann die durch  $e$  kodierte  $k$ -Band Turingmaschine  $T(|x|)$ -vielen Schritte laufen lassen. Wird  $x$  in dieser Zeit akzeptiert, so ist  $x \notin L(e)$ , anderenfalls ist  $x \in L(e)$ . Damit ist aber auch die Diagonalisierung

$$L_D := \{e \mid e \in L(e)\}$$

rekursiv. Die Sprache  $L_D$  ist aber nicht in  $DTIME(T)$ . Wäre sie es, so erhielten wir einen Index  $e$  einer  $k$ -Band Maschine, die  $L_D$  in  $DTIME(T)$  akzeptiert. Wäre nun

$$e \in L_D,$$

so akzeptierte  $\{e\}$  das durch  $e$  kodierte Wort in höchstens  $T(|e|)$ -vielen Schritten. Dann wäre aber  $e \notin L(e)$ , was unserer Annahme  $e \in L_D$  widerspricht. Umgekehrt folgte aus der Annahme

$$e \notin L_D,$$

Zeitkomplexität bei  
Reduktion der  
Anzahl der  
Arbeitsbänder

Eine rekursive  
Sprache, die nicht in  
 $DTIME$   
entscheidbar ist

## 6. Grundzüge der Komplexitätslehre

---

daß  $\{e\}$  das Wort  $e$  nicht innerhalb von  $T(|e|)$ -vielen Schritten akzeptiert. Also ist  $e \in L(e)$ , was  $e \notin L_D$  widerspricht. Die Annahme, daß  $L_D \in DTIME(T)$  sei, ist somit widerlegt.

Eine ähnliche Argumentation läßt sich natürlich auch für platzbeschränkte Klassen finden.

Damit haben wir das folgende Ergebnis:

**Lemma 63** *Zu jeder rekursiven Funktion  $T$  gibt es rekursive Sprachen  $L_S$  und  $L_T$ , mit  $L_S \notin \text{DSPACE}(T)$  und  $L_T \notin \text{DTIME}(T)$ .*

Gilt

$$T(n) \leq \tilde{T}(n)$$

für alle natürlichen Zahlen  $n$ , so ist ganz offensichtlich

$$\text{DTIME}(T) \subseteq \text{DTIME}(\tilde{T}).$$

Nach Lemma 63 gibt es zu rekursivem  $T$  eine Sprache  $L_T$ , die nicht in  $\text{DTIME}(T)$  liegt. Natürlich gibt es eine Turingmaschine, die  $L_T$  akzeptiert. Sei deren Laufzeit  $\hat{T}$  und  $\hat{T}$  definiert durch

$$\hat{T}(n) := \max\{T(n), \tilde{T}(n)\}.$$

Dann ist  $\hat{T}$  eine rekursive Funktion mit  $\hat{T}(n) \geq T(n)$  für alle natürlichen Zahlen  $n$ . Da  $L_T \in \text{DTIME}(\hat{T})$ , aber  $L_T \notin \text{DTIME}(T)$  ist, haben wir

$$\text{DTIME}(T) \subsetneq \text{DTIME}(\hat{T}),$$

Die  
 $\text{DTIME}$ -Hierarchie  
ist unendlich und  
nicht stationär

was zeigt, daß es sich bei der  $\text{DTIME}$ -Hierarchie um eine unendliche Hierarchie handelt, die niemals stationär wird. Analoge Überlegungen lassen sich natürlich auch wieder für die  $\text{DSPACE}$ -, die  $\text{NTIME}$ - und die  $\text{NSPACE}$ -Hierarchien anstellen. Auch hier erhalten wir unendliche, niemals stationär werdende Hierarchien.

Eine sich nun natürlich ergebende, interessante Frage ist die nach der Dichte der  $\text{DSPACE}$  bzw.  $\text{DTIME}$ -Hierarchien. Wir werden eine Hierarchie als dicht empfinden, wenn sie nur kleine Lücken hat. Dabei wollen wir unter einer Lücke zwei Funktionen  $S_1$  und  $S_2$  bzw.  $T_1$  und  $T_2$  verstehen, derart daß  $S_2$  bzw.  $T_2$  die Funktion  $S_1$  bzw.  $T_1$  majorisiert, aber dennoch  $\text{DSPACE}(S_1) = \text{DSPACE}(S_2)$  und  $\text{DTIME}(T_1) = \text{DTIME}(T_2)$  gelten. Liegt  $S_2$  nah an  $S_1$  bzw.  $T_2$  nah an  $T_1$ , so werden wir die Lücke als klein empfinden. Wir haben bereits gesehen, daß lineares Anwachsen von  $S_1$  und  $T_1$  zum Überbrücken einer Lücke nicht genügt. Es zeigt sich jedoch, daß unter gewissen Voraussetzungen an die Funktionen  $S_1$  und  $T_1$  schon recht kleine Änderungen genügen. Funktionen, die diese "gewissen Voraussetzungen" erfüllen, sind als platz- bzw. zeitkonstruierbare

Funktionen bekannt.

**Definition 64** Eine rekursive Funktion  $S$  heißt platzkonstruierbar, wenn es eine durch  $S$  SPACE-beschränkte Turingmaschine  $M$  gibt, die bei jedem Input der Länge  $n$  hält, nachdem sie genau  $S(n)$ -viele Speicherzellen beschrieben hat.

Platz- und  
zeitkonstruierbare  
Funktionen

Analog heißt eine rekursive Funktion  $T$  zeitkonstruierbar, wenn es eine durch  $T$  TIME-beschränkte Turingmaschine  $M$  gibt, die bei jedem Input der Länge  $n$  nach genau  $T(n)$ -Schritten hält.

Die hier verwendete Definition wird in der Literatur (z.B. [HOP79]) oft als vollkommen platz- bzw. zeitkonstruierbar (fully SPACE - bzw. fully TIME - constructible) bezeichnet. Für unsere Zwecke reicht aber dieser etwas engere Begriff völlig aus.

Für Funktionen, die stärker als die Identität wachsen, d.h. für Funktionen  $F$ , für die es ein  $\epsilon > 0$  mit  $F(n) \geq (1 + \epsilon) \cdot n$  gibt, erhält man die folgenden Charakterisierungen:

Eine Funktion  $F$  ist genau dann zeit- bzw. platzkonstruierbar, wenn  $F$  in  $O(F)$  Schritten bzw. mit einem Speicherbedarf von  $O(F)$  berechnet werden kann.

Der Beweis dieser Tatsache ist nicht eigentlich schwierig, benötigt aber die etwas technische Überlegung, daß aus der Zeitkonstruierbarkeit von  $F + G$  bei zeitkonstruierbarem  $G$  man auch die Zeitkonstruierbarkeit von  $F$  folgern kann, vorausgesetzt, daß  $F$  etwas stärker als  $G$  wächst.

Insbesondere läßt sich jede rekursive Funktion durch eine zeit- bzw. platzkonstruierbare Funktion majorisieren. Es gibt sehr viele Funktionen, die zeit- und platzkonstruierbar sind. Als Beispiele seien

Beispiele platz- und  
zeitkonstruierbarer  
Funktionen

$$\lambda n . n^k$$

$$\lambda n . n!$$

$$\log$$

$$\lambda n . 2^n$$

...

angegeben.

Gehen wir nun davon aus, daß wir zwei platzkonstruierbare Funktionen  $S_1$  und  $S_2$  haben, für die  $S_2 \in \omega(S_1)$  gilt, so können wir eine Sprache angeben, die in  $DSPACE(S_2)$ , aber nicht in  $DSPACE(S_1)$  liegt. Wir legen dazu eine Kodierung aller Turingmaschinen über einem geeigneten Alphabet zugrunde. Da wir uns um Platzkomplexität kümmern, dürfen wir ohne Einschränkung der Allgemeinheit von 1-Band Turingmaschinen ausgehen. Wir programmieren eine Turingmaschine  $M$ , die bei einem Input  $w$  der Länge  $n$  erst einmal  $S_2(n)$ -viele Felder markiert. Da  $S_2$  platzkonstruierbar ist, gibt es eine Turingmaschine, die dies garan-

## 6. Grundzüge der Komplexitätslehre

---

tiert und sich so auch auf  $M$  simulieren läßt. Die Maschine  $M$  sei nun so programmiert, daß immer ein nichtakzeptierender Zustand vorliegt, wenn  $M$  den markierten Bereich verläßt. Nach Beschreiben der Felder beginnt  $M$  die Maschine  $\{w\}$  zu simulieren. Die Maschine  $M$  akzeptiert  $w$  genau dann, wenn die Simulation von  $\{w\}$  gelingt, ohne mehr als  $S_2(n)$ -viele Felder zu beschreiben und  $\{w\}$  das Wort  $w$  nicht akzeptiert. Die von der Maschine  $M$  akzeptierte Sprache  $\mathcal{L}(M)$  ist offensichtlich in  $DSPACE(S_2)$ . Nehmen wir an, sie wäre auch in  $DSPACE(S_1)$ . Dann hätten wir eine Maschine  $\tilde{M}$ , die  $\mathcal{L}(M)$  mit einem durch  $S_1$  beschränkten Platzbedarf akzeptiert. Sei  $\Sigma$  das Alphabet von  $\mathcal{L}(M)$  und  $\sigma$  die Anzahl der Symbole in  $\Sigma$ . Da  $S_1$  platzkonstruierbar ist, dürfen wir ohne Einschränkung der Allgemeinheit annehmen, daß  $\tilde{M}$  bei jedem Input hält. Sei  $w$  ein Kode für  $\tilde{M}$ . Da es unendlich viele solche Kodes gibt (Satz von RICE) und  $S_2 \in \omega(S_1)$  gilt, finden wir einen Kode  $w$  einer Länge  $n$ , für den  $\lceil \log_2 \sigma \rceil \cdot S_1(n) \leq S_2(n)$  ist. Da das Alphabet  $\Sigma$  der Turingmaschine  $\tilde{M}$  aber  $\sigma$ -viele Symbole hat, benötigt  $M$  offenbar höchstens  $\lceil \log_2 \sigma \rceil \cdot S_1(n)$ -viele Speicherzellen, um  $\{w\}$  bei einem Input der Länge  $n$  zu simulieren. Damit hat  $M$  genügend Platz, um die Rechnung von  $\tilde{M}$  beim Input  $w$  zu simulieren. Da  $M$  das Wort  $w$  genau dann akzeptiert, wenn  $\tilde{M}$  dies nicht tut, ist  $\mathcal{L}(M)$  offenbar verschieden von  $\mathcal{L}(\tilde{M})$ . Dies widerspricht aber der Annahme, daß beide Sprachen übereinstimmen.

Für platzdefinierbare Funktionen erhält man somit den folgenden Hierarchiensatz:

Hierarchiensätze für  
platz- und  
zeitkonstruierbare  
Funktionen

**Satz 65** Sind  $S_1$  und  $S_2$  platzkonstruierbare Funktionen, derart daß

$$S_2 \in \omega(S_1)$$

ist, d.h. ist

$$\lim_{n \rightarrow \infty} \inf \frac{S_1(n)}{S_2(n)} = 0,$$

so gibt es eine Sprache in  $DSPACE(S_2)$ , die nicht in  $DSPACE(S_1)$  liegt.

Eine ähnliche Überlegung läßt sich auch für die Zeithierarchie anstellen. Allerdings können wir uns hier nicht mehr ohne Beschränkung der Allgemeinheit auf 1-Band Maschinen beziehen. Da das eben verwendete Diagonalargument aber eine feste Anzahl von Bändern erfordert, betrachten wir Turingmaschinen mit zwei Bändern. Das bedeutet, wie wir bereits gesehen haben, jedoch eine logarithmische Verlangsamung. Also erhalten wir für die Zeithierarchie:

**Satz 66** Sind  $T_1$  und  $T_2$  zeitkonstruierbare Funktionen mit

$$T_2 \in \omega(T_1 \cdot \log T_1),$$

d.h. haben wir

$$\liminf_{n \rightarrow \infty} \frac{T_1(n) \cdot \log T_1(n)}{T_2(n)} = 0,$$

so gibt es eine Sprache in  $\text{DTIME}(T_2)$ , die nicht in  $\text{DTIME}(T_1)$  liegt.

Die Zeithierarchie erscheint hier also weniger dicht als die Platzhierarchie.

Einige Beziehungen zwischen der Zeit- und Platzhierarchie lassen sich sehr leicht herstellen. Da eine Turingmaschine in  $T(n)$  Rechenschritten nicht mehr als  $T(n) + 1$  Felder beschreiben kann, erhalten wir sofort

$$\text{DTIME}(T) \subseteq \text{DSPACE}(T).$$

Sei nun  $M$  eine Turingmaschine über einem Alphabet mit  $\sigma$ -vielen Zeichen, die  $m$  Marken besitzt und eine Sprache  $\mathcal{L}$  mit einem durch eine Funktion  $S$  beschränkten Platz akzeptiert. Ist  $w$  ein Wort der Länge  $n$ , so ist  $z \cdot (n+2) \cdot S(n) \cdot z^{S(n)}$  offenbar eine obere Schranke für die Anzahl der möglichen Konfigurationen, die  $M$  beim Input  $w$  annehmen kann. Setzen wir voraus, daß  $S(n) \geq \log_2(n)$  ist, so finden wir eine Konstante  $c$  derart, daß  $z \cdot (n+2) \cdot S(n) \cdot z^{S(n)} \leq c^{S(n)}$  für alle  $n > 0$  ist. Zu  $M$  programmieren wir nun eine Mehrbandmaschine  $\tilde{M}$ , die auf einem zusätzlichen Band bis  $c^{S(n)}$  zählt. Auf den beiden anderen Bändern simuliere  $\tilde{M}$  die Maschine  $M$ . Falls  $M$  nach  $c^{S(n)}$  Rechenschritten noch nicht akzeptiert hat, so stoppt  $\tilde{M}$  in einem nicht akzeptierenden Zustand. Hat  $M$  nach  $c^{S(n)}$  Schritten noch nicht akzeptiert, so muß mindestens eine Konfiguration wiederholt worden sein.  $M$  wird daher das Inputwort niemals akzeptieren. Also gilt:

Ist  $\mathcal{L} \in \text{DSPACE}(S)$  und  $S(n) \geq \log_2(n)$ , so gibt es ein  $c$ , das im allgemeinen von der Sprache  $\mathcal{L}$  abhängen wird, derart, daß  $\mathcal{L} \in \text{DTIME}(c^S)$  ist.

Intuitiv ist auch klar, daß die Berechnung auf einer nichtdeterministischen Turingmaschine auf einer deterministischen nachgeahmt werden kann, indem man eine Linearisierung des Rechenbaumes abarbeitet. Der Preis hierfür ist allerdings ein exponentielles Anwachsen der Rechenzeit. Somit gilt auch die folgende Tatsache:

Ist  $\mathcal{L}$  in  $\text{NTIME}(T)$ , so gibt es ein  $c$ , (das wieder von der Sprache  $\mathcal{L}$  abhängen kann) derart, daß  $\mathcal{L}$  in  $\text{DTIME}(c^T)$  ist.

Ein weiterer wichtiger Zusammenhang zwischen deterministischer

Beziehungen zwischen der Zeit- und Platzhierarchie

## 6: Grundzüge der Komplexitätslehre

---

und nichtdeterministischer Platzkomplexität wird durch den folgenden Satz gegeben, auf dessen Beweis wir aber nicht weiter eingehen wollen:

Der Satz von  
SAVITCH

**Satz 67** (Satz von SAVITCH) *Ist eine Sprache  $\mathcal{L}$  in  $\text{NSPACE}(S)$ , und ist  $S$  eine platzkonstruierbare Funktion, so ist  $\mathcal{L}$  auch in  $\text{DSPACE}(S^2)$ .*

Die eben zitierten Ergebnisse könnten die Vermutung nahelegen, daß die nichtdeterministischen Platz- bzw. Zeithierarchien weniger dicht sein könnten als die deterministischen. Man könnte erwarten, daß die nichtdeterministische Platzhierarchie im Vergleich zur deterministischen erst mit quadratischen, die Zeithierarchie gar mit exponentiellen Schrittängen wächst. Daß dies nicht der Fall ist, zeigen die sogenannten Translationslemmata. So gilt beispielsweise für die Platzhierarchie:

Translationslemmata

**Lemma 68** *Sind  $S_1, S_2$  und  $F$  platzkonstruierbare Funktionen mit  $F(n) \geq n$  für alle  $n$  und gilt*

$$\text{NSPACE}(S_1) \subsetneqq \text{NSPACE}(S_2),$$

*so folgt auch*

$$\text{NSPACE}(S_1 \circ F) \subsetneqq \text{NSPACE}(S_2 \circ F).$$

Völlig analoge Resultate erhält man im übrigen auch für die  $\text{DSPACE}$ -,  $\text{DTIME}$ - und  $\text{NTIME}$ -Hierarchien.

Konsequenzen der  
Translationslemmata

Translationslemmata haben interessante Konsequenzen. So läßt sich beispielsweise zeigen, daß

$$\text{DTIME}(2^n) \subsetneqq \text{DTIME}(n \cdot 2^n)$$

gilt, obwohl

$$\liminf_{n \rightarrow \infty} \frac{2^n \cdot \log(2^n)}{n \cdot 2^n} = 1$$

ist, und sich das Ergebnis somit nicht unmittelbar aus Satz 66 folgern läßt. Dennoch spielt dieser zusammen mit dem Translationslemma die Schlüsselrolle in dieser Überlegung. Nehmen wir nämlich an, wir hätten

$$\text{DTIME}(n \cdot 2^n) = \text{DTIME}(2^n),$$

so folgte mit einer zweimaligen Anwendung des Translationslemmas für  $\text{DTIME}$ , wobei einmal  $F(n) := n + 2^n$  und das anderemal  $F(n) := 2^n$  gewählt sei,

$$\text{DTIME}((n + 2) \cdot 2^n \cdot 2^{2^n}) \subseteq \text{DTIME}(2^n \cdot 2^{2^n}) \subseteq \text{DTIME}(2^{2^n}).$$

Dies steht aber im Widerspruch zu Satz 66, da

$$\liminf_{n \rightarrow \infty} \frac{2^{2^n} \cdot \log 2^{2^n}}{(n + 2^n) \cdot 2^n \cdot 2^{2^n}} = 0$$

ist.

Als eine weitere interessante Anwendung erhalten wir

$$NSPACE(n^{\frac{x}{y}}) \subsetneq NSPACE(n^{\frac{x+1}{y}}).$$

Auch hier nehmen wir an, wir hätten

$$NSPACE(n^{\frac{x+1}{y}}) = NSPACE(n^{\frac{x}{y}}).$$

Mit dem Translationslemma für  $F(n) := n^{(x+i)y}$  erhielten wir dann

$$NSPACE(n^{(x+1)(x+i)}) \subseteq NSPACE(n^{x(x+i)}) \quad (6.7)$$

für alle natürlichen Zahlen  $i$ . Wegen  $x(x+i) < (x+1)(x+i-1)$  für  $i > 0$  haben wir auch

$$NSPACE(n^{x(x+i)}) \subseteq NSPACE(n^{(x+1)(x+i-1)}) \quad (6.8)$$

für alle positiven natürlichen Zahlen  $i$ . Alternierender Gebrauch von (6.7) und (6.8) liefert schließlich

$$\begin{aligned} NSPACE(n^{(x+1)(x+x)}) &\subseteq NSPACE(n^{x(x+x)}) \\ &\subseteq NSPACE(n^{(x+1)(x+x-1)}) \subseteq NSPACE(n^{x(2x-1)}) \\ &\subseteq \dots \subseteq NSPACE(n^{(x+1)x}) \subseteq NSPACE(n^{x^2}). \end{aligned}$$

Nach dem Satz von SAVITCH gilt aber

$$NSPACE(n^{x^2}) \subseteq DSPACE(n^{2x^2})$$

und mit Satz 65 erhalten wir

$$DSPACE(n^{2x^2}) \subsetneq DSPACE(n^{2x^2+2x}) \subseteq NSPACE(n^{2x^2+2x}).$$

Zusammengenommen bedeutet dies den Widerspruch

$$\begin{aligned} NSPACE(n^{(x+1)(x+x)}) &\subsetneq NSPACE(n^{x^2}) \subseteq NSPACE(n^{2x^2+2x}) \\ &= NSPACE(n^{(x+1)(x+x)}). \end{aligned}$$

Unsere Annahme war also falsch. Setzen wir  $r := \frac{x}{y}$  und  $\varepsilon := \frac{1}{y}$ , so ergibt dies den folgenden Satz:

**Satz 69** Für reelle  $r \geq 0$  und  $\varepsilon > 0$  ist

$$NSPACE(n^r) \subsetneq NSPACE(n^{r+\varepsilon}).$$

## 6. Grundzüge der Komplexitätslehre

---

Die nichtdeterministische Platzhierarchie ist also bereits im polynomia- len Bereich sehr dicht. Ähnliche Ergebnisse lassen sich auch für die nichtdeterministische Zeithierarchie erzielen. Allerdings benötigt man dort etwas andersartige Überlegungen, da man nicht mehr auf den Satz von SAVITCH zurückgreifen kann.

### 6.2 Abstrakte Komplexitätsmaße

Die Eigenschaften von Komplexitätsmaßen, die wir bisher kennengelernt haben, decken sich ziemlich genau mit den intuitiven Vorstellungen, die man sich üblicherweise von einem Komplexitätsmaß macht. Dies ist allerdings nur die Spitze des Eisberges. So hängt beispielsweise das stete Anwachsen der Platz- bzw. Zeithierarchie wesentlich davon ab, daß die Funktionen, die die Platz- bzw. Zeitbeschränkung liefern selbst platz- bzw. zeitkonstruierbar sind. Läßt man diese Voraussetzung fallen, so ändern sich die Verhältnisse dramatisch. Als Beispiel dafür darf der Lückensatz von BORODIN gelten, der besagt, daß

*zu jeder rekursiven Funktion  $r$ , die stärker als die Identität wächst, eine rekursive Funktion  $S$  so existiert, daß*

$$\text{DSPACE}(S) = \text{DSPACE}(r \circ S)$$

*ist.*

Mit Hilfe des Lückensatzes lassen sich durch geeignete Wahl von  $r$  beliebig große Lücken in der Platzhierarchie konstruieren. Ähnliche Lückensätze gelten auch für andere Komplexitätsmaße wie beispielsweise die Zeitkomplexität.

Die bereits mehrfach aufgetretene Tatsache, daß sich viele Eigen- schaften der Platzkomplexität ohne große Schwierigkeiten auch auf die Zeitkomplexität und auf die nichtdeterministischen Komplexitätsvarian- ten übertragen ließen, legt natürlich den Gedanken nahe, eine abstrakte Notation von Komplexitätsmaßen zu entwickeln. Ein derartiger axiomati- scher Ansatz zur Entwicklung einer Komplexitätstheorie wurde erstmalig von M. BLUM angegeben. Wir wollen kurz auf diesen Ansatz eingehen, ohne ihn allerdings allzusehr zu vertiefen. Nach BLUM ist ein abstraktes Komplexitätsmaß für partiellrekursive Funktionen durch die folgenden Axiome gegeben:

**Definition 70** Eine  $n + 1$ -stellige partiellrekursive Funktion  $\Phi$  heißt ein abstraktes Komplexitätsmaß für  $n$ -stellige partiellrekursive Funktionen, wenn gilt:

1.  $\text{dom}(\lambda \vec{x}. \Phi(e, \vec{x})) = \text{dom}(\{e\})$

Der Lückensatz

Axiomatische  
Entwicklung der  
Komplexitätstheorie

Abstrakte  
Komplexitätsmaße

### 2. $\Phi$ hat einen rekursiven Graphen.

Ist  $\Phi$  ein abstraktes Komplexitätsmaß, so nennen wir in Analogie zur Platz- bzw. Zeitkonstruierbarkeit eine Funktion  $F$   $\Phi$ -konstruierbar, wenn es ein  $e$  gibt mit  $F = \lambda \vec{n}. \Phi(e, \vec{n})$ . Ist nun  $F$  eine partiellrekursive Funktion und  $M$  eine primitiv rekursive Maschine mit Index  $e$ , die  $F$  berechnet, so ist  $\Phi(e, \vec{m}) : \simeq Rz_{\{e\}}(\vec{m})$  ganz offensichtlich eine Funktion, die die BLUMschen Axiome erfüllt. Schon im Abschnitt 2.1 hatten wir klar gestellt, daß  $\text{dom}(Rz_M) = \text{dom}(\text{Res}_M) = \text{dom}(F)$  ist. Wir müssen uns also nur noch klar machen, daß die Rechenzeitfunktion einen rekursiven Graphen hat. Um

$$Rz_{\{e\}}(\vec{m}) \simeq n$$

zu entscheiden, gewinnen wir durch Dekodierung von  $e$  die Maschine  $M$  und lassen  $M$  mit Input  $\text{IN}(\vec{m})$  genau  $n$ -viele Schritte laufen. Erreicht die Maschine dann eine Endkonfiguration  $(k_E, \vec{z})$ , so fällt die Antwort positiv aus, anderenfalls aber negativ. Offenbar liefert dies ein rekursives Entscheidungsverfahren. In ähnlicher Form läßt sich einsehen, daß die Anzahl der von einer Turingmaschine im Verlaufe einer Berechnung besuchten Felder ein Komplexitätsmaß liefert. Es ist auch nicht schwer einzusehen, daß Platz- und Zeitbedarf auch für nichtdeterministische Maschinen Komplexitätsmaße liefern, die die BLUMschen Axiome erfüllen.

Bei den BLUMschen Axiomen haben wir aus Vereinfachungsgründen angenommen, daß die Komplexität einer natürlichen Zahl  $n$  einfach  $n$  ist. Gehen wir aber von beliebigen Datenbereichen  $\mathcal{D}$  und einem Komplexitätsmaß  $|d|$  auf den Ausgangsdaten  $d \in \mathcal{D}$  aus, so haben wir das erste BLUMsche Axiom abzuändern zu

*I'.  $n \in \text{dom}(\lambda x. \Phi(e, x))$  genau dann, wenn die Maschine  $\{e\}$  bei jedem Input  $w$  der Komplexität  $|w| = n$  hält.*

Insbesondere erhalten wir damit ein abstraktes Komplexitätsmaß für akzeptierende Maschinen.

Viele Eigenschaften von Komplexitätsmaßen lassen sich bereits aus den BLUMschen Axiomen herleiten. Beispielsweise der Lückensatz. Der Lückensatz hat kuriose Folgerungen. So garantiert er die Existenz einer rekursiven Funktion  $g$ , für die

$$\text{DTIME}(g) = \text{NTIME}(g) = \text{DSPACE}(g) = \text{NSPACE}(g)$$

ist. Die Inklusionen  $\subseteq$  in der obigen Kette gelten offensichtlich. Die inverse Inklusion ergibt sich aus der Tatsache, daß zu  $\mathcal{L} \in \text{NSPACE}(g)$  eine Konstante  $c$  so existiert, daß  $\mathcal{L} \in \text{DTIME}(c^g)$  ist. Damit läßt sich einsehen, daß  $\mathcal{L}$  schon in  $\text{DTIME}(g^g)$  ist. Wenden wir nun den Lücken-

## 6. Grundzüge der Komplexitätslehre

---

satz für  $DTIME$  auf die Funktion  $r(x) = x^x$  an, so erhalten wir die Existenz einer rekursiven Funktion  $g$  mit  $DTIME(g) = DTIME(g^g)$ .

Der BLUMsche  
Beschleunigungssatz

Eine weitere merkwürdige Eigenschaft, die sich bereits aus den BLUMschen Axiomen herleiten lässt, ist der sogenannte *Beschleunigungssatz*. Dieser besagt, daß sich rekursive Funktionen konstruieren lassen, für die sich besten Programme finden lassen. In abstrakter Form läßt er sich wie folgt formulieren.

**Satz 71** (*Beschleunigungssatz von BLUM*) *Sei  $\Phi$  ein abstraktes Komplexitätsmaß und  $r$  eine rekursive Funktion. Dann gibt es eine partiellrekursive Funktion  $F$  derart, daß sich zu jedem Index  $e$  für  $F$  ein Index  $e_b$  so finden läßt, daß*

$$r(\Phi(e_b, n)) \leq \Phi(e, n)$$

*für fast alle natürlichen Zahlen  $n$  gilt.*

Der Name ‘‘Beschleunigungssatz’’ (Speed-up Theorem) wird verständlich, wenn man das abstrakte Komplexitätsmaß als Zeitkomplexität interpretiert. Dann besagt der Beschleunigungssatz, daß sich zu *jedem* Programm  $e$  für  $F$ , das  $F(n)$  in der Rechenzeit  $T(n)$  berechnet, ein Programm  $e_b$  finden läßt, das  $F(n)$  in höchstens  $T_b(n)$ -vielen Schritten berechnet, wobei fast überall  $r(T_b(n)) \leq T(n)$  ist.

Der Beschleunigungssatz beruht natürlich wieder auf einer Diagonalisierung. Man konstruiert sich eine Funktion  $F$ , die durch kurze Programme (d.h. Programme mit kleinen Indizes) nicht schnell zu berechnen ist. Je länger man jedoch das Programm macht, desto schneller läßt sich die Funktion berechnen. Da es (nach dem Satz von RICE) beliebig lange Programme für  $F$  gibt, läßt sich dessen Berechnung durch Verlängerung des Programmes beliebig beschleunigen.

Schwächen  
abstrakter  
Komplexitätsmaße

Mit Hilfe abstrakter Komplexitätsmaße läßt sich eine recht elegante Theorie entwickeln. Absolut unbefriedigend ist jedoch die Tatsache, daß sich Komplexitätsmaße angeben lassen, die der Intuition, die wir üblicherweise mit einem Komplexitätsmaß verbinden, eklatant widersprechen. So sollte die Komplexität zweier nacheinander ablaufender Programme doch mindestens so groß wie die jedes einzelnen Programmes sein. Es lassen sich jedoch Komplexitätsmaße  $\Phi$  angeben, die die BLUMschen Axiome erfüllen, dabei aber die Eigenschaft

$$\Phi([F \circ G], n) < \Phi([G], n)$$

haben. D.h., daß die Komplexität der nacheinander ablaufenden Programme kleiner werden kann, als die der einzelnen Programme.

Eine weitere Eigenschaft, die sich ebenfalls bereits aus den BLUMschen Axiomen folgern läßt, ist der Vereinigungssatz. Ist  $\Phi$  ein abstraktes

Komplexitätsmaß, so sei

$$K_\Phi(S) := \{\{e\} \mid (\forall d)(\Phi(e, d) \leq S(|d|))\}.$$

Interpretieren wir  $\Phi$  als die deterministische Platzkomplexität, so ist  $K_\Phi(S) = \text{DSPACE}(S)$ , interpretieren wir es als deterministische Zeitkomplexität, ist  $K_\Phi(S) = \text{DTIME}(S)$  und so fort.

**Satz 72 (Vereinigungssatz)** Sei  $\Phi$  ein abstraktes Komplexitätsmaß und  $\{r_i \mid i \in \mathbb{N}\}$  eine rekursiv aufzählbare Kollektion rekursiver Funktionen derart, daß für jedes  $n$  immer  $r_i(n) < r_{i+1}(n)$  gilt. Dann gibt es eine rekursive Funktion  $S$  mit

$$K_\Phi(S) = \bigcup_{i \in \mathbb{N}} K_\Phi(r_i).$$

Der Vereinigungssatz, auf dessen Beweis wir nicht näher eingehen wollen, stellt sich als nützlich für die Nomenklatur von Komplexitätsklassen heraus. So können wir die Klassen

$$P = \bigcup_{i \in \mathbb{N}} \text{DTIME}(n^i)$$

der in *polynomialer Zeit deterministisch entscheidbaren Probleme* und

$$NP = \bigcup_{i \in \mathbb{N}} \text{NTIME}(n^i)$$

In *polynomialer Zeit entscheidbare Probleme*

der in *polynomialer Zeit nichtdeterministisch entscheidbaren Probleme* definieren. Mit diesen Klassen wollen wir uns im folgenden Abschnitt befassen.

## 6.3 P- und NP- entscheidbare Probleme

Am Ende des letzten Abschnittes hatten wir die Klassen  $P$  und  $NP$  der in *polynomialer Zeit deterministisch bzw. nichtdeterministisch entscheidbaren Probleme* eingeführt. Wie wir zu Beginn des Kapitels bereits erwähnten, hat es sich eingebürgert, die Klasse  $P$  als die der praktisch entscheidbaren Probleme zu betrachten. Diese Auffassung wird empirisch dadurch gestützt, daß sich die zur Klasse  $P$  gehörenden "natürlichen" Probleme in der Regel durch Algorithmen entscheiden lassen, deren Zeitkomplexitäten sich durch Polynome niedrigen Grades beschränken lassen. Man kennt jedoch auch eine Reihe von Problemen, für die nicht-deterministische Algorithmen bekannt sind, die in *polynomialer Zeit* zum Abschluß kommen.

## 6. Grundzüge der Komplexitätslehre

---

Eine Analogie zwischen  $NP$  und  $Rel_{r.e.}$  sowie  $P$  und  $Rel_r$ .

Es besteht eine gewisse Analogie zwischen der Klasse  $P$  und den rekursiven Mengen einerseits und der Klasse  $NP$  und den rekursiv aufzählbaren Mengen andererseits. So ist  $P$  gerade die Klasse der praktisch entscheidbaren Probleme, wohingegen  $Rel_r$  die Klasse der (theoretisch) entscheidbaren Probleme darstellt. Wir hatten weiter beobachtet, daß die Klasse  $Rel_{r.e.}$  der rekursiv aufzählbaren Mengen gerade die Klasse der theoretisch positiv entscheidbaren Probleme beschreibt. Ist nun  $\mathcal{L} \in NP$ , so haben wir einen nichtdeterministischen polynomialem Algorithmus, der das Wortproblem für  $\mathcal{L}$  entscheidet. Ist  $w \in \mathcal{L}$  zu entscheiden, so denken wir uns die Berechnung so als Rechenbaum angeordnet, wie wir es in Abbildung 6.1 - 1 angedeutet haben. Ist nun  $w \in \mathcal{L}$ , so gibt es einen Pfad in dem Berechnungsbaum, in dem  $w \in \mathcal{L}$  bestätigt wird. Dieser Pfad liefert uns aber einen polynomialem Algorithmus, der  $w \in \mathcal{L}$  bestätigt. Damit haben wir in Analogie zur Klasse  $Rel_{r.a.}$  die Klasse  $NP$  als die Klasse der Probleme, die sich praktisch positiv entscheiden lassen. Allerdings sieht man sofort, daß, zumindest wie wir sie begründet haben, diese Analogie unvollständig ist. Im Falle der rekursiv aufzählbaren Menge läßt sich der Bestätigungsalgorithmus global angeben, wohingegen im Falle eines  $NP$ -Problems nur lokale Bestätigungsalgorithmen vorliegen, was heißen soll, daß der Algorithmus, der uns  $w \in \mathcal{L}$  bestätigt, von  $w$  abhängen kann.

Das Versagen der Analogie schlägt sich auch in Eigenschaften der Klassen  $P$  und  $NP$  nieder. So ließ sich ja ziemlich einfach zeigen, daß die rekursiven Relationen eine echte Teilklasse der rekursiv aufzählbaren Relationen bilden. Die COOKsche Hypothese

$$P \subsetneq NP$$

Die COOKsche Hypothese

Gründe für die Schwierigkeit der COOKschen Hypothese

hingegen ist eines der berühmtesten offenen Probleme der Komplexitätstheorie. Es gibt gewichtige Anzeichen dafür, daß dieses Problem auch sehr schwer zu lösen sein dürfte. Eines ist die Tatsache, daß es bislang allen Lösungsversuchen getrotzt hat. Ein anderes die Arbeit [BAK75] von T. BAKER, J. GILL und R. SOLOVAY, in der sie gezeigt haben, daß sich das  $P = NP?$  Problem so relativieren läßt, daß relativ zu einer Menge  $A$  einmal

$$P^A = NP^A$$

gilt, man andererseits aber auch eine Menge  $B$  angeben kann, für die

$$P^B \neq NP^B$$

ist. Da die in der Rekursionstheorie üblichen Methoden (etwa wie sie in diesem Buch dargestellt wurden) Relativierungen in der Regel überleben,

legt das zitierte Ergebnis die Vermutung nahe, daß das Problem mit der gängigen Methodik nicht zu attackieren ist.

Neben den Klassen  $P$  und  $NP$  sind auch deren Platzentsprechungen

$$PSPACE := \bigcup_{i \in \mathbb{N}} DSPACE(n^i)$$

und

$$NSPACE := \bigcup_{i \in \mathbb{N}} NSPACE(n^i)$$

von Interesse. Dabei folgt aus dem Satz von SAVITCH (Satz 67), daß

$$NSPACE = PSPACE$$

ist. Innerhalb von  $PSPACE$  haben wir die Hierarchie

$$\begin{aligned} DSPACE(\log) &\subsetneq DSPACE(\log^2) \subsetneq \dots \\ &\dots \subsetneq DSPACE(\log^i) \subsetneq \dots \subseteq PSPACE. \end{aligned}$$

Insbesondere folgt damit

$$DSPACE(\log) \subseteq P \subseteq NP \subseteq PSPACE,$$

und wir wissen, daß eine der Inklusionen echt zu sein hat, da

$$DSPACE(\log) \subsetneq PSPACE$$

ist. Offen ist jedoch, welche der Inklusionen echt ist.

Die Klasse  $P$  hat ähnliche Abschlußeigenschaften, wie die der rekursiven Relationen. So haben wir Abschluß gegenüber allen booleschen Operationen, stark beschränkter Quantifikation und der Substitution mit polynomial berechenbaren Funktionen. Da wir nicht wissen, ob  $P = NP$  ist, können wir über die Abschlußeigenschaften von  $NP$  allerdings wenig sagen. So wissen wir beispielsweise nicht, ob die Klasse  $CONP$  der Komplemente von  $NP$ -Relationen mit der Klasse  $NP$  übereinstimmt.

Abschlußeigenschaften  
der Klasse  $P$

Es ist natürlich nun von besonderem Interesse, den Begriff der Unlösbarkeitsgrade (vgl. Abschnitt 5.1) auf die Klassen  $P$  und  $NP$  zu übertragen, da es ja auch von praktischem Interesse ist, zu wissen, ob sich ein Problem durch ein deterministisches Programm in polynomia-ler Zeit lösen läßt, oder ob es einen nichtdeterministischen Algorithmus erfordert.

Die Unlösbarkeitsgrade, die sich hier anbieten, sind eine Anpassung der m-Grade, wie wir sie in Abschnitt 5.1 eingeführt haben. Allerdings macht es jetzt natürlich keinen Sinn mehr zu fordern, daß die Reduktion durch eine rekursive Funktion erfolgt. Die angebrachte Anpassung be-

## 6. Grundzüge der Komplexitätslehre

---

steht natürlich darin, daß man nur Reduktionen über Funktionen zuläßt, die selbst polynomial berechenbar sind.

**Definition 73** Eine Sprache  $A$  über einem Alphabet  $\Sigma$  heißt polynomial  $m$ -reduzibel auf eine Sprache  $B$  über  $\Sigma$ , wenn es eine polynomial zeitbeschränkte Funktion  $f : \Sigma^* \rightarrow \Sigma^*$  gibt, so daß

$$x \in A \text{ genau dann wenn } f(x) \in B$$

für alle Wörter  $x \in A$  gilt.

Wir notieren die Tatsache, daß  $A$  polynomial  $m$ -reduzibel auf  $B$  ist durch

$$A \leq_p B$$

p-Reduzibilität

Manche Autoren lassen ganz generell nur  $\text{DSPACE}(\log_2)$  beschränkte Funktionen als reduzierende Funktionen zu. Alle hier noch zu erwähnenden Ergebnisse, gelten auch für diesen Reduktionsbegriff.

Wenn wir den Begriff der polynomialen Berechenbarkeit auf zahlentheoretische Funktionen anwenden, d.h. Funktionen, die Tupel natürlicher Zahlen in natürliche Zahlen abbilden, müssen wir allerdings etwas vorsichtig sein. Wir haben davon auszugehen, daß wir natürliche Zahlen als Wörter einer geeigneten Sprache darzustellen haben. Es ist daher unrealistisch, einer natürlichen Zahl  $n$  eine Länge 1 zuzuordnen. In der Darstellung über dem Alphabet  $\{*, B\}$ , wie wir sie im Kapitel über maschinenberechenbare Funktionen benutzt haben, bekäme  $n$  die Komplexität  $n$ , was natürlich eine viel zu schlechte Kodierung darstellt. Üblicherweise stützt man sich hier auf die Tatsache, daß zu jeder natürlichen Zahl  $n$  eindeutig eine *Binärdarstellung*

$$n = \sum_{i=0}^k b_i \cdot 2^i \quad \text{mit } b_i \in \{0, 1\}$$

existiert. Auf diese Weise läßt sich jede natürliche Zahl  $n$  durch das Wort  $b_0 \dots b_k$  der Länge  $\lceil \log_2(n+1) \rceil$  über dem Alphabet  $\{0, 1\}$  darstellen. Daher wollen wir im folgenden immer von dem Komplexitätsmaß

$$|n| := \lceil \log_2(n+1) \rceil$$

für natürliche Zahlen  $n$  ausgehen. Für Tupel  $\vec{n}$  natürlicher Zahlen bedeute  $|\vec{n}|$  das Tupel  $(|n_1|, \dots, |n_k|)$ .

Die Natürlichkeit der Klasse  $P$  wird dadurch unterstrichen, daß sich die in polynomialer Zeit berechenbaren zahlentheoretischen Funktionen in kanonischer Weise maschinenunabhängig definieren lassen. Der Vollständigkeit halber wollen wir einen derartigen Zugang skizzieren, ohne dies später jedoch weiter zu vertiefen.

Unser Ziel ist es, ähnlich wie wir dies bei den primitiv rekursiven und partiellrekursiven Funktionen getan haben, die polynomial berechenbaren Funktionen als Abschluß einer Klasse von Grundfunktionen unter gewissen Operationen zu erhalten. Dazu führen wir zunächst den Begriff der *beschränkten Rekursion* ein, der eine Abschwächung der primitiven Rekursion darstellt. Seien also  $g$  eine  $n$ - und  $h$  eine  $n+2$ -stellige zahlentheoretische Funktion und  $q$  und  $p$  Polynome. Wir sagen, daß die

Ein natürliches Komplexitätsmaß für natürliche Zahlen

Ein alternativer Zugang zu polynomial berechenbaren Funktionen

$n$ -stellige Funktion  $f$  aus  $g$  und  $h$  durch beschränkte Rekursion mit Platzschranke  $q$  und Zeitschranke  $p$  definierbar ist, wenn die folgenden beiden Bedingungen erfüllt sind:

Beschränkte Rekursion

- $f(\vec{m}) = \text{Rec}(g, h)(\vec{m}, p(|\vec{m}|))$  für jedes  $n$ -Tupel  $\vec{m}$ .
- Für jedes  $k \leq |\vec{m}|$  ist  $|\text{Rec}(g, h)(\vec{m}, k)| \leq q(|\vec{m}|)$ .

Auch bei der Wahl der Grundfunktionen müssen wir etwas vorsichtiger sein. Es genügt, hier von den folgenden Funktionen auszugehen:

1. Die Nullfunktion 0,
2. die Nachfolgerfunktion  $S$ ,
3. die Rechtsverschiebungsfunktion  $\lambda n. \lfloor \frac{1}{2} \cdot n \rfloor$ ,
4. die Linksverschiebungsfunktion  $\lambda n. (2 \cdot n)$ ,
5. die charakteristische Funktion  $\chi_{\leq}$  der  $\leq$ -Relation,
6. die Fallunterscheidungsfunktion  $\text{Case}(k, l, m) := \begin{cases} l, & \text{falls } k > 0 \\ m, & \text{falls } k = 0. \end{cases}$

Die kleinste Klasse zahlentheoretischer Funktionen, die die eben angeführten Grundfunktionen umfaßt und abgeschlossen ist gegenüber Substitutionen und beschränkter Rekursion ist dann genau die Klasse der in polynomialer Zeit berechenbaren Funktionen. Die Klasse  $P$  ergibt sich dann als die Klasse der Prädikate, deren charakteristische Funktionen sich aus den Grundfunktionen durch Substitution und beschränkte Rekursion gewinnen lassen.

Man nennt nun eine Menge  $R$  *NP-schwer* (englisch “NP-hard”, was wohl zu der auch oft benutzten Bezeichnung *NP-hart* geführt hat), wenn

NP-schwere und NP-vollständige Probleme

$$A \leq_p R$$

für jedes  $A \in NP$  gilt.

Ist  $R \in NP$  und  $R$  darüber hinaus auch *NP-schwer*, so heißt  $R$  *NP-vollständig*.

Diese Definitionen übertragen sich sinngemäß auf die übrigen Komplexitätsklassen.

Eine weitere amüsante Analogie – die allerdings keine tiefliegenden Gründe zu haben scheint – zwischen den Klassen  $Rel_{r.a.}$  und  $NP$  ergibt sich durch die folgenden Beobachtungen: Wir haben in Abschnitt 5.3 gesehen, daß die Gültigkeit von Sätzen der Prädikatenlogik unentscheidbar ist, indem wir das Halteproblem darauf reduzieren konnten. Da das Halteproblem aber m-vollständig für die rekursiv aufzählbaren Mengen ist und nach dem Vollständigkeitssatz der Prädikatenlogik Kalküle existieren, die alle gültigen Sätze aufzählen, diese also eine rekursiv aufzählbare Menge bilden, ist das Entscheidungsproblem der Prädikatenlogik umgekehrt auch auf das Halteproblem reduzierbar und somit selbst m-vollständig für  $Rel_{r.a.}$ . Im Gegensatz zur Prädikatenlogik spielen in der Aussagenlogik nur die Junktoren  $\neg, \wedge, \vee$  (vgl. Abschnitt 5.3) eine Rolle. Zu Ehren von G. BOOLE bezeichnet man diese Junktoren – und oft auch alle Operationen, die sich aus ihnen ableiten lassen – als

## 6. Grundzüge der Komplexitätslehre

boolesche Operationen.

Boolesche Terme

Ein boolescher Term (oft auch *boolesche Formel* genannt) setzt sich aus Aussagenvariablen  $u, v, u_0, v_0, u_1, \dots$  mittels boolescher Operationen zusammen.

Boolesche Belegungen

Eine boolesche Belegung ist eine Zuordnung von einem der Wahrheitswerte **W** oder **F** zu den Aussagenvariablen. Ist

$$\Phi: \text{Aussagenvariablen} \longrightarrow \{\text{W}, \text{F}\}$$

Wahrheitstafeln

eine boolesche Belegung, so berechnet sich der Wert  $t^\Phi$  eines booleschen Ausdrucks  $t$  gemäß den in Abschnitt 5.3 angegebenen Regeln, d.h. die booleschen Operationen gehorchen den Wahrheitstafeln

$\wedge$	W	F
W	W	F
F	F	F

$\vee$	W	F
W	W	W
F	W	F

$$\neg W = F$$
$$\neg F = W,$$

die für die zweistelligen Operationen  $\wedge$  und  $\vee$  so zu lesen sind, daß das erste Argument in der linken Spalte, das zweite Argument in der obersten Reihe und der Funktionswert am Schnittpunkt der zugehörigen Reihe und Spalte steht.

Das Erfüllbarkeitsproblem für boolesche Terme

Ein boolescher Term  $t$  heißt erfüllbar, wenn sich eine Belegung  $\Phi$  so finden läßt, daß  $t^\Phi = \text{W}$  ist. Das *Erfüllbarkeitsproblem* für die Aussagenlogik besteht nun darin, zu entscheiden, ob ein vorgegebener boolescher Term erfüllbar ist. Wir wollen das Erfüllbarkeitsproblem für boolesche Terme mit *SAT* abkürzen.

Es ist natürlich wieder sofort klar, daß der Wert von  $t^\Phi$  nur von der Belegung der in  $t$  tatsächlich auftretenden Aussagenvariablen abhängt. Dies sind nur endlich viele, sagen wir  $n$  Stück. Also können höchstens  $2^n$ -viele Belegungen existieren, für die  $t^\Phi$  verschiedene Werte annehmen kann. Da sich diese alle austesten lassen, ist *SAT* entscheidbar. Die Frage kann nur noch sein, wie komplex es ist.

Das Erfüllbarkeitsproblem für boolesche Terme ist *NP*-vollständig

Ohne auf Einzelheiten eingehen zu können, die zwar nicht sehr schwierig, aber doch langwierig darzustellen wären, wollen wir hier erwähnen, daß *SAT* ein *NP*-vollständiges Problem ist. Leicht einzusehen ist die Tatsache, daß *SAT* in *NP* liegt. Für jeden booleschen Term  $t$  der Länge  $n$  und jede vorgegebene Belegung  $\Phi$  läßt sich  $t^\Phi$  in einer durch eine Funktion  $T \in O(n)$  beschränkten Zeit berechnen. Wir erhalten ein *NP* Programm, indem wir alle möglichen Belegungen erraten und dann parallel auswerten lassen.

Viel aufwendiger ist es, zu zeigen, daß SAT NP-schwer ist. Die Idee besteht darin, einem durch ein Polynom  $p$  zeitbeschränkten nichtdeterministischen Programm  $P$  und einem Input  $w$  einen booleschen Term  $t_{P,w}$  so zuzuordnen, daß  $P$  das Wort  $w$  genau dann akzeptiert, wenn  $t_{P,w}$  erfüllbar ist. Dies kann so erfolgen, daß  $t_{P,w}$  aus  $w$  in polynomialer Zeit berechnet werden kann. Damit hat man  $P \leq_p \text{SAT}$ . Das hört sich allerdings einfacher an als es ist. Man hat nämlich zu beachten, daß im Term  $t_{P,w}$  der Ablauf des nichtdeterministischen Programmes mit aussagenlogischen Mitteln axiomatisch beschrieben werden muß. Daß dies gelingt, liegt wesentlich daran, daß das Programm  $P$  ein NP-Programm ist. Details finden sich beispielsweise in [HOP79] oder [PAUL78].

Wir haben eingangs gesehen, daß das Entscheidungsproblem für die volle Prädikatenlogik m-vollständig für die rekursiv aufzählbaren Mengen war. Das Erfüllbarkeitsproblem für die Aussagenlogik ist zwar entscheidbar, aber doch NP-vollständig (und damit vermutlich praktisch unentscheidbar). Da das Erfüllbarkeitsproblem eng mit dem Entscheidbarkeitsproblem zusammenhängt (eine Formel  $F$  ist genau dann entscheidbar, wenn  $\neg F$  unerfüllbar ist), zeigt sich hier eine weitere Analogie zwischen den Klassen  $\text{Rel}_{r.c.}$  und  $\text{NP}$ .

Das Ergebnis über die NP-Vollständigkeit des Erfüllbarkeitsproblems läßt sich noch verschärfen. Dazu betrachtet man sogenannte *konjunktive Normalformen*. Um zu beschreiben, worum es sich bei einer konjunktiven Normalform handelt, wollen wir unter einem *Literal* eine Aussagenvariable  $u$  oder eine einfach negierte Aussagenvariable  $\neg u$  verstehen. Eine Disjunktion  $L_1 \vee \dots \vee L_k$  von Literalen  $L_i$  nennen wir dann eine *reine Disjunktion* der Länge  $k$ . Dual nennt man eine Konjunktion von Literalen eine *reine Konjunktion*. Eine  $k$ -*konjunktive Normalform* ist eine Konjunktion reiner Disjunktionen der Längen  $k$ . Eine *konjunktive Normalform* ist eine Konjunktion reiner Disjunktionen, deren Längen beliebig sein dürfen. Völlig dual definiert man eine *disjunktive Normalform* als eine Disjunktion reiner Konjunktionen.

Konjunktive  
Normalformen für  
boolesche Terme

Wir nennen zwei boolesche Terme  $t_1$  und  $t_2$  äquivalent und notieren dies durch

$$t_1 \equiv t_2,$$

wenn  $t_1^\Phi = t_2^\Phi$  für jede boolesche Belegung  $\Phi$  gilt. Mit den DEMORGANSCHEN Regeln

$$\neg(A \wedge B) \equiv (\neg A \vee \neg B)$$

$$\neg(A \vee B) \equiv (\neg A \wedge \neg B)$$

läßt sich leicht zeigen, daß jeder boolesche Term zu einem äquivalenten

## 6. Grundzüge der Komplexitätslehre

---

NP-Vollständigkeit  
des Erfüllbarkeits-  
problems für  
3-konjunktive  
Normalformen

Term in konjunktiver Normalform umgeformt werden kann. Enthiebt der ursprüngliche Term  $n$  Aussagenvariablen, so wird der Term in konjunktiver Normalform nicht mehr als  $2^n$ -viele Aussagenvariablen enthalten.

Der Beweis der NP-Vollständigkeit des Erfüllbarkeitsproblems kann so geführt werden, daß sich der ergebende boolesche Term  $t_{P,w}$  in konjunktiver Normalform befindet. Damit hat man gezeigt, daß bereits das Erfüllbarkeitsproblem für boolesche Terme in konjunktiver Normalform NP-vollständig ist. Nun kann man sich fragen, wie lang die reinen Disjunktionen in der konjunktiven Normalform sein müssen, damit noch NP-Vollständigkeit vorliegt. Es läßt sich zeigen, daß bereits 3-konjunktive Normalformen NP-vollständig sind. Dieses Problem erweist sich oft als recht handlich, wenn es darum geht, andere Probleme darauf zu reduzieren.

Man kennt sehr viele NP-schwere und NP-vollständige Probleme. Viele von Ihnen sind Probleme der *diskreten Mathematik* und lassen sich daher gut in graphentheoretischer Sprache formulieren. Wir werden im Anhang über Graphentheorie kurz darauf zurückkommen.

Praktische Relevanz  
von  
NP-Schwierigkeit  
und -Vollständigkeit

Die Feststellung, daß ein Problem NP-schwer ist, ist durchaus von praktischer Bedeutung. Man hat ohne Erfolg viel Energie darauf verwendet, effiziente (d.h. polynomiale) Algorithmen für Probleme zu entwickeln, die wir heute als NP-schwer oder NP-vollständig erkannt haben. (Ein Beispiel dafür ist das Problem des Handlungsreisenden.) Diese Erfahrung lehrt uns, daß es – wenigstens solange nicht  $P = NP$  durch einen effizienten Algorithmus gesichert werden kann, was wenig wahrscheinlich ist – wenig Sinn macht nach polynomialem Algorithmen für Probleme zu suchen, die als NP-schwer nachgewiesen sind. Hier ist es viel besser, nach heuristischen, der speziellen Situation angepaßten Berechnungsverfahren zu suchen. Generell wissen wir nur, daß Algorithmen mit exponentieller Laufzeit zur Verfügung stehen, was diese Probleme praktisch unlösbar macht.

Es ist allerdings nicht zwangsläufig, daß selbst wenn  $P \neq NP$  ist, jedes NP-Problem auch wirklich exponentielle Laufzeit beanspruchen muß. Es ist durchaus vorstellbar, daß es in einer Zwischenklasse, z.B.  $DTIME(\lambda n \cdot n^c \log n)$  für eine Konstante  $c$  liegen kann. Ließe sich ein NP-vollständiges Problem angeben, das in  $DTIME(\lambda n \cdot n^c \log n)$  liegt, so folgte bereits  $NP \subseteq DTIME(\lambda n \cdot n^c \log n)$ . Allerdings sind derartige Probleme bislang nicht gefunden worden.

Ein PSPACE-  
vollständiges  
Problem

Ausgehend von dem Erfüllbarkeitsproblem SAT für boolesche Formeln gelangen wir auch zu einem Problem, das sich als PSPACE-vollständig erwiesen hat. Dazu führen wir den Begriff der *quantifizierten booleschen Formel* (QBF) ein.

Wir erhalten eine QBF, indem wir Aussagenvariablen gegenüber

### 6.3. P- und NP- entscheidbare Probleme

den booleschen Operationen  $\neg$ ,  $\wedge$ ,  $\vee$  und Quantifikationen  $(\exists u)$ ,  $(\forall u)$  über Aussagenvariablen abschließen. Beispiele für QBFs sind

$$(\forall u)(u \vee \neg u) \\ (\exists u)(\forall v)(u \leftrightarrow (v \wedge \neg v)),$$

Quantifizierte  
boolesche Formeln

wobei  $u \leftrightarrow v$  die boolesche Operation  $(\neg u \vee v) \wedge (\neg v \vee u)$  abkürzt. Wie in der Prädikatenlogik sagen wir, daß eine Aussagenvariable  $u$ , die im Bereich eines Quantors  $(\forall u)$  oder  $(\exists u)$  steht, durch diesen Quantor *gebunden* wird. Nicht gebundene Aussagenvariablen bezeichnen wir als *frei*. Wir wollen wieder davon ausgehen, daß, wann immer wir eine QBF  $(\forall u)F$  oder  $(\exists u)F$  bilden, die Aussagenvariable  $u$  nicht gebunden in  $F$  auftreten darf. Eine QBF heißt *geschlossen*, wenn sie keine freien Aussagenvariablen enthält. Wie bei den booleschen Termen verstehen wir unter einer booleschen Belegung eine Zuordnung von Wahrheitswerten  $W$  und  $F$  zu den Aussagenvariablen. Die Auswertung  $EVAL(F, \Phi)$  einer QBF bei einer booleschen Belegung  $\Phi$  ist dann einer der Wahrheitswerte  $W$  oder  $F$ , der sich wie folgt berechnet:

$$EVAL(u, \Phi) := \Phi(u) \text{ für Aussagenvariablen } u.$$

$$EVAL(\neg F, \Phi) = \neg EVAL(F, \Phi).$$

$$EVAL(F_1 \wedge F_2, \Phi) = \wedge(EVAL(F_1, \Phi), EVAL(F_2, \Phi)).$$

$$EVAL(F_1 \vee F_2, \Phi) = \vee(EVAL(F_1, \Phi), EVAL(F_2, \Phi)).$$

$EVAL((\forall u)F, \Phi) = W$  genau dann, wenn  $EVAL(F, \Psi) = W$  für alle booleschen Belegungen  $\Psi$  gilt, die bis auf den Wert  $\Psi(u)$  mit  $\Phi$  übereinstimmen.

$EVAL((\exists u)F, \Phi) = W$  genau dann, wenn  $EVAL(F, \Psi) = W$  für eine boolesche Belegung  $\Psi$  gilt, die bis auf den Wert  $\Psi(u)$  mit  $\Phi$  übereinstimmt.

Dabei seien die Wahrheitsfunktionen  $\neg$ ,  $\wedge$  und  $\vee$  wie vorher in den Wahrheitstafeln angegeben definiert.

Es ist wieder offensichtlich, daß der Wahrheitswert  $EVAL(F, \Phi)$  einer QBF  $F$  bei einer booleschen Belegung  $\Phi$  nur von der Belegung der Aussagenvariablen abhängt, die tatsächlich in  $F$  auftreten. Insbesondere hat jede geschlossene QBF  $F$  einen wohlbestimmten Wahrheitswert  $EVAL(F)$ . Eine geschlossene QBF  $F$  heißt gültig, wenn  $EVAL(F) = W$  ist. Da eine QBF  $F$ , in der höchstens die Aussagenvariablen  $u_1, \dots, u_n$  frei vorkommen, genau dann erfüllbar ist, wenn die QBF  $(\exists u_1) \dots (\exists u_n)F$  gültig ist, ist das Erfüllbarkeitsproblem für QBFs gelöst, wenn man einen Algorithmus zur Berechnung von  $EVAL(F, \Phi)$  hat. Da es immer nur endlich viele (d.h. genau zwei) Möglichkeiten gibt,

## 6. Grundzüge der Komplexitätslehre

---

eine Aussagenvariable zu belegen, ist das oben angegebene Berechnungsverfahren für  $EVAL(F, \Phi)$  offenbar rekursiv. Damit läßt sich die Frage  $EVAL(F) = W?$  prinzipiell entscheiden. Es ist ebenfalls leicht einzusehen, daß der angegebene Algorithmus mit quadratischem Platzbedarf auskommt. Damit ist klar, daß das Entscheidungsproblem für QBFs in  $PSPACE$  ist. Deutlich schwieriger einzusehen ist die Tatsache, daß es auch  $PSPACE$  - schwer ist. Hier sei auf die Literatur (beispielsweise [HOP79]) verwiesen.

Ein weiteres, natürliches Problem, das sich als  $PSPACE$ -vollständig nachweisen läßt, ist das Wortproblem für kontextsensitive Sprachen, das wir in Abschnitt 5.2 als primitiv rekursiv erkannt haben. Auch hier sei auf die Literatur verwiesen. In [HOP79] finden sich auch Beispiele für  $P$ -vollständige Probleme.

Wir haben bereits bemerkt, daß bis heute nicht feststeht, ob  $NP$ -Probleme tatsächlich praktisch unentscheidbar sind. Es sind allerdings Beispiele von Problemen bekannt, die beweisbarerweise exponentielle Ressourcen erfordern und somit praktisch unentscheidbar sind.

Ein relativ altes und demgemäß gut bekanntes Beispiel ist die *Presburger Arithmetik*. Bei der Presburger Arithmetik handelt es sich um die Theorie der Struktur  $(\mathbb{N}, +)$ . Das zu entscheidende Problem ist die Frage, ob für einen Satz  $\phi$  des Sprachfragments  $\mathcal{L}(\{+\})$  der Zahlentheorie, das nur die Addition als nichtlogisches Zeichen zuläßt,

$$(\mathbb{N}, +) \models \phi$$

gilt. Mit Methoden der Modelltheorie läßt sich zeigen, daß dies tatsächlich prinzipiell entscheidbar ist. M. J. FISCHER und M. O. RABIN haben jedoch 1974 gezeigt, daß jedes Entscheidungsverfahren mindestens eine Zeit  $2^{2^{O(n)}}$  erfordert und somit ein praktisch unlösbare Problem vorliegt.

Wir wollen diesen Abriß der Komplexitätstheorie nicht beschließen, ohne auf dessen Unvollkommenheit hinzuweisen. Der an weiteren Details interessierte Leser sei auf die Literatur verwiesen, wobei es in Ermangelung umfassender Lehrbücher oft nötig sein wird, auch Originalarbeiten zu studieren.

Das Wortproblem  
für kontextsensitive  
Sprachen ist  
 $PSPACE$ -vollständig

Ein theoretisch  
entscheidbares, aber  
praktisch  
unentscheidbares  
Problem

---

# A. Grundlagen der Graphentheorie

Obwohl die diskrete Mathematik nicht unbedingt zu den mathematischen Grundlagen der Informatik zu rechnen ist, stellt sie doch mit eines ihrer wesentlichsten Hilfsmittel dar. Es ist nicht ganz einfach zu beschreiben, was das eigentliche Anliegen diskreter Mathematik ist. Der Terminus diskret ist wohl als Gegensatz zum Begriff des Kontinuums zu interpretieren, und diskrete Mathematik als der Teil der Mathematik zu interpretieren, der die Phänomene studiert, die auftreten, wenn man das Kontinuum der reellen Zahlen durch diskrete Objekte (wie z.B. Gleitkommazahlen) ersetzt. Es würde den Rahmen dieses Buches bei weitem sprengen, wollten wir hier eine Beschreibung der Methoden diskreter Mathematik beginnen. Ein wesentlicher Bestandteil der diskreten Mathematik ist aber sicherlich die *Kombinatorik*. Viele kombinatorische Probleme lassen sich bequem in der Sprache der *Graphentheorie* formulieren. Da wir im vorhergehenden Text schon mehrfach Bezug auf graphentheoretische Konzepte genommen haben, wollen wir die Grundbegriffe der Graphentheorie kurz zusammenstellen.

## A.1 Grundbegriffe

Beginnen wir mit der formalen Definition eines Graphen.

Formale Definition  
eines Graphen

**Definition 74** Ein Graph ist ein geordnetes Paar  $\mathcal{G} = (V, E)$ , wobei  $V$  eine nicht leere Menge (die Menge der Knoten von  $\mathcal{G}$ ) und  $E$  (die Menge der Kanten von  $\mathcal{G}$ ) eine Teilmenge von  $[V]^{\leq 2}$ , der Menge aller Teilmengen von  $V$  mit höchstens zwei Elementen, ist.

Ist  $v$  ein Knoten und  $e$  eine Kante mit  $v \in e$ , so sagen wir, daß der Knoten  $v$  auf der Kante  $e$  liegt oder, daß  $v$  mit  $e$  *incident* ist.

Ist  $e = \{u, v\}$ , so sagen wir, daß  $e$  die Knoten  $u$  und  $v$  verbindet. Gibt es eine Kante, die die Knoten  $u$  und  $v$  in einem Graphen  $\mathcal{G}$  verbindet, so heißen  $u$  und  $v$  in  $\mathcal{G}$  *benachbart* oder *adjacent*.

Sehr viel besser als durch diese abstrakte Definition kann man sich einen Graphen graphisch veranschaulichen. Betrachten wir das Beispiel in Abbildung A.1 - 1 so ist die Menge der Knoten

$$V = \{v_1, \dots, v_4\}$$

und die Menge der Kanten geben durch

## A. Grundlagen der Graphentheorie

---

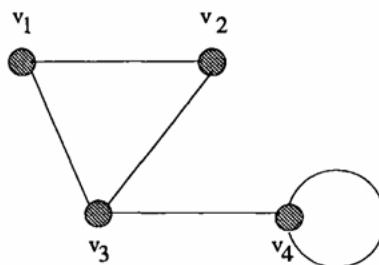


Abb. A.1 - 1: Ein ungerichteter Graph mit vier Knoten

---

$$E = \{\{v_1, v_2\}, \{v_1, v_3\}, \{v_2, v_3\}, \{v_3, v_4\}, \{v_4\}\}.$$

Gerichtete Graphen

Graphen dieser Art bezeichnet man als *ungerichtete Graphen*. Fordern wir in Definition 74 anstatt  $E \subseteq [V]^{\leq 2}$  jedoch  $E \subseteq V^2$ , so hat jede Kante die Gestalt  $(u, v)$  mit Knoten  $u, v \in V$ , wodurch eine Richtung der Kante von  $u$  nach  $v$  vorgegeben ist. Man spricht dann von einem *gerichteten Graphen*. In der graphischen Darstellung gerichteter Graphen wird die Richtung der Kanten durch *Pfeile* angegeben. Im Beispiel von Abb. A.1 - 2 ist wieder

$$V = \{v_1, \dots, v_4\}$$

und

$$E = \{(v_1, v_2), (v_1, v_3), (v_3, v_2), (v_3, v_4), (v_4, v_4)\}.$$

Die oben vereinbarten Redeweisen "v liegt auf e" und "e verbindet u

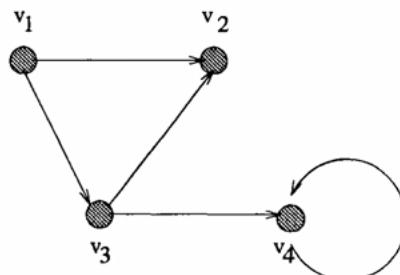


Abb. A.1 - 2: Ein gerichteter Graph mit vier Knoten

---

und v" übernehmen wir sinngemäß.

Manchmal betrachtet man auch *gemischte Graphen*, in denen sowohl gerichtete als auch ungerichtete Kanten vorkommen dürfen.

Einen noch allgemeineren Zugang stellt der Begriff des *Multigraphen* dar. Ein Multigraph lässt sich als ein Tripel  $(V, E, \phi)$  definieren, wobei  $V$  und  $E$  disjunkte Mengen sind und

$$\phi: E \longrightarrow \text{Pow}(V)$$

eine Abbildung von der Kantenmenge  $E$  in die Potenzmenge der Knotenmenge  $V$  ist, so daß  $|\phi(e)| \leq 2$  ist. Jeder Kante  $e$  wird also eine Knotenmenge  $\phi(e) = \{u, v\}$  zugeordnet. Man verwendet wieder die Redeweise, daß  $e$  die Knoten  $u$  und  $v$  verbindet. Der Unterschied zur vorherigen Definition besteht darin, daß nun zwei Knoten durch mehr als nur eine Kante verbunden werden können. Solche Situationen können auftreten, wenn man beispielsweise versucht, Straßenkarten als Graphen darzustellen, in dem Plätze oder andere markante Punkte als Knoten auftreten. Die diese Punkte verbindenden Straßenzüge werden durch Kanten repräsentiert. Da sich zwischen zwei Plätzen in der Regel eine Vielzahl von verbindenden Straßenzügen finden lassen, würde es hier sinnvoll sein, Multigraphen zu betrachten. In einem Multigraphen heißen zwei Kanten  $e_1 \neq e_2$  mit  $\phi(e_1) = \phi(e_2)$  ein *Zweieck*.

Betrachten wir  $\mathcal{G} = (V, E, \phi)$  mit  $\phi: E \longrightarrow V \times V$ , so ist durch  $\phi(e) = (v_1, v_2)$  wieder eine Richtung von  $v_1$  nach  $v_2$  festgelegt und wir erhalten einen gerichteten Multigraphen.

Man kann natürlich auch gemischte Multigraphen betrachten. Im obigen Beispiel wäre dies der Fall, wenn wir die Richtung eventueller Einbahnstraßen im Straßennetz berücksichtigen wollten.

Wir haben in unserer Definition nicht ausgeschlossen, daß Knoten in einem Graphen zu keiner Kante gehören. Wir sprechen dann von isolierten Knoten. Im folgenden wollen wir jedoch immer stillschweigend voraussetzen, daß wir Graphen ohne isolierte Knoten vorliegen haben.

Ein Graph heißt *endlich*, wenn seine Kantenmenge endlich ist. Wir werden im folgenden immer nur von endlichen Graphen sprechen.

Eine Folge

$$P = (v_1, e_1, v_2, e_2, v_3, \dots, v_n, e_n, v_{n+1}), \quad (\text{A.1})$$

wobei  $v_1, \dots, v_{n+1}$  Knoten und  $e_1, \dots, e_n$  Kanten sind, für die immer  $(\phi)(e_i) = \{v_i, v_{i+1}\}$  – bzw.  $(\phi)(e_i) = (v_i, v_{i+1})$  im Falle eines gerichteten (Multi)Graphen – gilt, heißt ein *Kantenzug* oder *Pfad*. Der Knoten  $v_1$  heißt der *Ausgangspunkt*,  $v_{n+1}$  der *Endpunkt* des Pfades. Wir sagen, daß der Pfad  $P$  die Knoten  $v_1$  und  $v_n$  verbindet. Der Kantenzug heißt *geschlossen* wenn  $v_1 = v_n$  ist. Die *Länge*  $\lg(P)$  des Kantenzuges

Kantenzüge und  
Pfade

## A. Grundlagen der Graphentheorie

---

$$P = (v_1, e_1, v_2, e_2, v_3, \dots, v_n, e_n, v_{n+1})$$

ist die Zahl  $n$ .

Einfache  
Kantenzüge und  
Wege

Wir sprechen von einem *einfachen Kantenzug*, wenn  $e_i \neq e_j$  für  $i \neq j$  ist und von einem *Weg*, wenn  $v_i \neq v_j$  für  $i \neq j$  ist. In einem einfachen Kantenzug wird also keine Kante zweimal durchlaufen, ein Weg besucht jeden Knoten höchstens einmal.

Kreise

Ist  $v_1 = v_{n+1}$  und  $v_2, \dots, v_n$  ein Weg, so sprechen wir von einem *Kreis* oder *Zykel*. Jeder geschlossene Kantenzug, lässt sich zu einem Kreis verkürzen.

Lokale Kantenzahl  
und Ordnung eines  
Graphen

Eine Kante  $e$ , die den gleichen Knoten verbindet, wird *Schlinge* genannt.

Die *lokale Kantenzahl*  $\text{grad}_{\mathcal{G}}(v)$  eines Knotens  $v$  in einem Graphen  $\mathcal{G}$  ist die Anzahl (Kardinalzahl) aller Kanten mit denen  $v$  inzidiert. Dabei werden i.a. Schlingen doppelt gezählt. Die *Ordnung* eines Graphen  $\mathcal{G} = (V, E)$  ist dann durch

$$\alpha(\mathcal{G}) := \sum_{v \in V} \text{grad}_{\mathcal{G}}(v)$$

definiert.

Ist  $\mathcal{G}$  ein schlingenloser Graph ohne Zweiecke, so erhalten wir für die Anzahl  $\kappa(\mathcal{G})$  der Kanten in  $\mathcal{G}$  offenbar

$$\kappa(\mathcal{G}) = \frac{1}{2} \cdot \alpha(\mathcal{G}).$$

Zusammenhangs-  
komponenten und  
zusammenhängende  
Graphen

Zwei Knoten  $v_1$  und  $v_2$  heißen *zusammenhängend*, wenn es einen Weg mit Anfangspunkt  $v_1$  und Endpunkt  $v_2$  gibt. Man notiert dies manchmal als  $v_1 \sim v_2$ . Ganz offensichtlich ist durch  $\sim$  eine Äquivalenzrelation auf den Knoten eines Graphen gegeben. Die Äquivalenzklassen von  $\sim$  in einem Graphen  $\mathcal{G}$  heißen die *Zusammenhangskomponenten* von  $\mathcal{G}$ . Hat  $\mathcal{G}$  nur eine einzige Zusammenhangskomponente, so sprechen wir von einem *zusammenhängenden Graphen*. Es ist leicht einzusehen, daß sich ein zusammenhängender Graph nicht als disjunkte Vereinigung zweier nicht leerer Graphen darstellen läßt.

Sind  $u$  und  $v$  zwei zusammenhängende Knoten, so definieren wir deren Abstand

$$d_{\mathcal{G}}(u, v) := \min\{\lg(P) \mid P \text{ verbindet } u \text{ mit } v\}.$$

Vollständige  
Graphen

Ergänzen wir diese Definition um  $d_{\mathcal{G}}(u, v) := \infty$ , falls  $u$  und  $v$  nicht zusammenhängen, so definiert der Abstand  $d_{\mathcal{G}}$  eine Metrik auf  $\mathcal{G}$ .

Ein Graph  $\mathcal{G} = (V, E)$  heißt *vollständig*, wenn  $E = [V]^{\leq 2}$  ist, d.h. wenn jeder Knoten mit jedem Knoten durch eine Kante verbunden ist.

Man spricht auch von vollständigen gerichteten Graphen, von denen man verlangt, daß jede Kante mit jeder durch einen Pfeil verbunden ist (d.h., zu den Knoten  $u$  und  $v$  müssen die Kanten  $(u, v)$  und  $(v, u)$  existieren).

In einem vollständigen Graphen  $\mathcal{G}$  mit  $\alpha(\mathcal{G})$ -vielen Knoten, gilt

$$\text{grad}_{\mathcal{G}}(v) = \alpha(\mathcal{G}) - 1.$$

Daher erhalten wir für die Kantenzahl

$$\kappa(\mathcal{G}) = \frac{1}{2} \cdot \alpha(\mathcal{G})(\alpha(\mathcal{G}) - 1).$$

Graphen, in denen alle Knoten den gleichen Grad haben, nennt man *regulär*. Wie wir eben eingesehen haben, sind vollständige Graphen regulär. Andere Beispiele für reguläre Graphen sind die Oberflächengraphen regulärer Polyeder. Deshalb nennt man einen vollständigen Graphen der Ordnung  $n$  oft auch einen  *$n$ -Simplex* und bezeichnet ihn mit  $S(n)$ .

Ist  $(u, v)$  eine Kante in dem gerichteten Graphen  $\mathcal{G}$ , so heißt “ $u$  ein *unmittelbarer Vorgänger* von  $v$ ” oder dual “ $v$  ein *unmittelbarer Nachfolger* von  $u$ ”.

Ein wichtiger Spezialfall eines gerichteten Graphen ist ein *Baum*. Ein Baum ist ein gerichteter Graph  $B$ , der den folgenden Bedingungen genügt:

- $B$  besitzt einen ausgezeichneten Knoten – die *Wurzel des Baumes* –, der keine unmittelbaren Vorgänger besitzt und von dem aus ein Weg zu jedem Knoten des Baumes führt.
- Mit Ausnahme der Wurzel hat jeder Knoten des Baumes genau einen unmittelbaren Vorgänger.

Oft ist es auch nützlich, eine Ordnung auf den unmittelbaren Nachfolgern eines Knotens in einem Baum zu haben. Dazu nimmt man an, daß eine Ordnung auf der Menge  $V$  der Knoten des Baumes vorgegeben ist, die dann in kanonischer Weise die unmittelbaren Nachfolger jedes Knotens ordnet. Solche Bäume heißen *geordnet*.

Wir können sofort einsehen, daß ein Baum keine Kreise enthalten kann. Nehmen wir nämlich an, wir hätten einen Kreis  $v = v_1 \rightarrow \dots \rightarrow v_n = v$ . Dann hat jeder der Knoten  $v_i$  einen unmittelbaren Vorgänger und kann daher nicht die Wurzel sein. Auf der anderen Seite gibt es einen Pfad von der Wurzel  $w$  zu jedem der  $v_i$ . Also muß mindestens einer der Knoten unter den  $v_i$  mehr als einen unmittelbaren Vorgänger haben, was nach der Definition eines Baumes nicht möglich ist.

Vergessen wir die Richtung der Kanten in einem Baum, so gibt es zu je zwei Knoten  $v_1$  und  $v_2$  in dem Baum einen Kantenzug mit Anfangsknoten  $v_1$  und Endknoten  $v_2$ , der über die Wurzel führt. Es ist leicht einzusehen, daß jeder solche Kantenzug sich zu einem Weg verkürzen

## A. Grundlagen der Graphentheorie

---

lässt. Also ist ein Baum – als ungerichteter Graph betrachtet – zusammenhängend. Man definiert daher manchmal etwas allgemeiner einen Baum als einen zusammenhängenden Graphen ohne Kreise.

### A.2 Subgraphen und Homomorphismen

Sei  $\mathcal{G} = (V, E)$  ein Graph.  $\mathcal{G}' = (V', E')$  heißt ein *Teilgraph* von  $\mathcal{G}$ , wenn  $V' \subseteq V$  und  $E' \subseteq E$  ist. Offenbar ist der leere Graph  $\emptyset = (\emptyset, \emptyset)$  ein Teilgraph jedes Graphen.

Entsteht der Teilgraph  $\mathcal{G}'$  eines Graphen  $\mathcal{G}$  durch Restriktion der Nachbarschaftsrelation, d.h. gilt

$$E' = \{\{x, y\} \in E \mid x, y \in V'\},$$

so spricht man von einem *Unter- oder Subgraphen*.

Eine Teilmenge  $V_0 \subseteq V$  der Knotenmenge von  $\mathcal{G}$  spannt somit den Untergraphen

$$\mathcal{G}(V_0) := (V_0, \{\{x, y\} \in E \mid x, y \in V_0\})$$

auf. Seien  $\mathcal{G}_1 = (V_1, E_1)$  und  $\mathcal{G}_2 = (V_2, E_2)$  Graphen. Eine Abbildung

$$F: V_1 \longrightarrow V_2,$$

mit der Eigenschaft, daß

1. für alle  $\{x, y\} \in E_1$ , die eine Kante von  $\mathcal{G}_1$  sind,  $F(x) = F(y)$  oder  $\{F(x), F(y)\} \in E_2$  gilt,

heit ein *Homomorphismus* der Graphen  $\mathcal{G}_1$  und  $\mathcal{G}_2$ . Wir notieren dies oft auch als

$$F: \mathcal{G}_1 \longrightarrow \mathcal{G}_2.$$

Von grerem Interesse sind jedoch *Epimorphismen* von Graphen, bei denen es sich um Homomorphismen  $F$  handelt, die den folgenden zusätzlichen Bedingungen genügen:

2.  $F$  ist surjektiv

3. Zu jeder Kante  $\{x_2, y_2\} \in E_2$  gibt es eine Kante  $\{x_1, y_1\} \in E_1$  mit  $F(x_1) = x_2$  und  $F(y_1) = y_2$ .

*Isomorphismen* von Graphen sind injektive Epimorphismen.

Der Homomorphiesatz der universellen Algebra überträgt sich natürlich auch auf Epimorphismen von Graphen. Um dies zu verdeutlichen, betrachten wir eine Partition  $P$  der Knoten eines Graphen  $\mathcal{G}$ . Für einen Knoten  $x$  bezeichne  $P_x$  das Element der Partition, das  $x$  enthält. Alle Knoten in einer Partition bilden eine Äquivalenzklasse. Zwei Äquiva-

Homomorphismen  
und Epimorphismen  
von Graphen

lenzklassen betrachten wir als benachbart, wenn sie disjunkt sind und mindestens eine Kante zwischen beiden verläuft, d.h. wenn  $P_x \neq P_y$  ist und es  $x_0 \in P_x$  sowie  $y_0 \in P_y$  so gibt, daß  $\{x_0, y_0\}$  eine Kante ist. Auf diese Weise erhalten wir den Faktorgraphen

$$\mathcal{G}/P$$

zusammen mit dem kanonischen Epimorphismus

$$\pi: \mathcal{G} \longrightarrow \mathcal{G}/P$$

$$\pi(x) := P_x.$$

Ist nun  $F: \mathcal{G}_1 \longrightarrow \mathcal{G}_2$  ein Epimorphismus der Graphen, so liefert die Menge  $P := \{F^{-1}(z) \mid z \in V_2\}$  eine Partition des Graphen  $\mathcal{G}_1$ . Es ist dann sofort nachzurechnen, daß die Abbildung

$$\tilde{F}: \mathcal{G}/P \longrightarrow \mathcal{G}_2$$

$$\tilde{F}(F^{-1}(z)) := z$$

wohldefiniert und ein Isomorphismus ist, für den

$$F = \tilde{F} \circ \pi$$

gilt.

Ein Epimorphismus

$$F: \mathcal{G}_1 \longrightarrow \mathcal{G}_2$$

heißt *kontraktierend*, wenn alle Urbilder  $F^{-1}(z)$  der Knoten von  $\mathcal{G}_2$  zusammenhängende Untergraphen von  $\mathcal{G}_1$  aufspannen. Gibt es einen Subgraphen  $\mathcal{G}_0 \subseteq \mathcal{G}_1$  und einen kontraktierenden Epimorphismus von  $\mathcal{G}_0$  auf einen Graphen  $\mathcal{G}_2$ , so sagen wir, daß ein *c-Homomorphismus* aus  $\mathcal{G}_1$  in  $\mathcal{G}_2$  existiert. Dies notiert man oft als

$$\mathcal{G}_1 \succ \mathcal{G}_2.$$

Die Relation  $\succ$  ist eine Halబordnung, d.h. reflexive, transitive und antisymmetrische Relation, auf den (Isomorphieklassen von) endlichen Graphen. Anschaulich bedeutet  $\mathcal{G} \succ \mathcal{G}'$ , daß sich  $\mathcal{G}$  in zusammenhängende Subgraphen zerlegen läßt, die sich bijektiv auf die Knoten in  $\mathcal{G}'$  abbilden lassen, wobei zusammenhängende Subgraphen, die durch mindestens eine Kante verbunden sind, auf in  $\mathcal{G}'$  benachbarte Knoten abgebildet werden.

Von Interesse sind auch noch Graphen, die durch Identifikation von Knoten entstehen. Dazu gehen wir von einer Teilmenge  $U \subseteq V$  der Knotenmenge eines Graphen  $\mathcal{G}$  aus und lassen diese auf einen Punkt

## A. Grundlagen der Graphentheorie

---

zusammenschrumpfen, indem wir alle Knoten, die in  $U$  lagen, aus  $\mathcal{G}$  entfernen und einen neuen Punkt hinzufügen, den wir mit den verbliebenen Knoten von  $\mathcal{G}$  verbinden, die bereits früher einen Nachbarn in  $U$  hatten. Wir notieren den neu entstandenen Graphen auch mit

$$\mathcal{G}/H,$$

Verwechslungen mit dem Restklassengraphen werden durch den jeweiligen Kontext praktisch immer ausgeschlossen. Wählen wir für  $U$  speziell eine Kante  $e = \{x, y\}$ , so sagen wir, daß der neue Graph

$$\mathcal{G}/e$$

durch *Kantenkontraktion* entstanden ist. Man kann nun feststellen, daß für endliche Graphen

$$\mathcal{G} \succ \mathcal{G}'$$

immer genau dann gilt, wenn sich  $\mathcal{G}$  durch sukzessive Kantenkontraktionen in einem Graphen überführen läßt, der einen zu  $\mathcal{G}'$  isomorphen Teilgraphen besitzt.

## A.3 Pfadprobleme

Das Königsberger  
Brückenproblem

EULER-Graphen

Eines der Ausgangsprobleme, das zur Entwicklung der Graphentheorie führte, war das Königsberger Brückenproblem (vgl. Abb. A.3 - 1). Das Problem besteht darin, eine Spaziergang durch Königsberg zu finden, der genau einmal über jede der sieben Pregelbrücken führt und zum Ausgangspunkt zurückkehrt. Schon EULER bemerkte, daß sich dieses Problem auf ein Pfadproblem in dem in Abbildung A.3 - 2 dargestellten Multigraphen reduzieren läßt. Die Knoten in den Graphen sind die durch den Fluß Pregel abgegrenzten Stadtteile Königsbergs, die Kanten die Wege über die Brücken. Gesucht ist nun ein geschlossener Pfad, der *alle* Kanten enthält. EULER hat bereits die Bedingungen untersucht, denen ein Multigraph zu gehorchen hat, damit er Pfade enthält, die über alle Kanten führen. Man nennt solche Pfade daher heute EULER-Pfade. Man überlegt sich relativ leicht, daß ein Graph  $\mathcal{G}$  genau dann ein EULER-Graph ist, d.h. ein Graph mit einem EULER-Pfad, wenn er den folgenden Bedingungen genügt:

1.  $\mathcal{G}$  ist zusammenhängend.
2. Für jeden Knoten  $v$  in  $\mathcal{G}$  ist  $\text{grad}_{\mathcal{G}}(v)$  gerade.

Damit folgt sofort, daß das Königsberger Brückenproblem unlösbar ist.

Man kann nun das Brückenproblem dahingehend erweitern, daß man

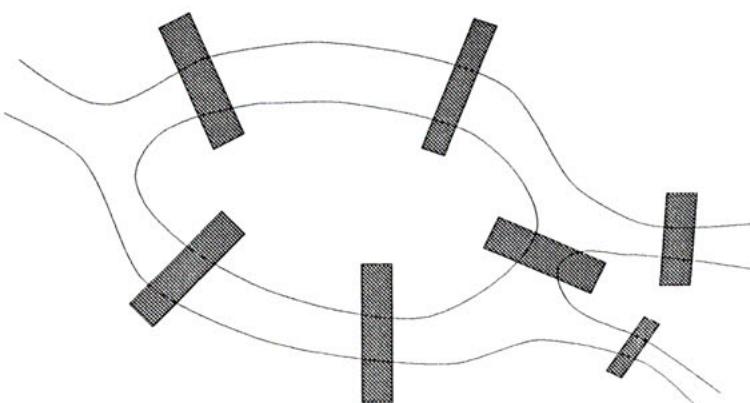


Abb. A.3 - 1 : Das Königsberger Brückenproblem

für Graphen die Frage nach der kleinsten Zahl stellt, die man benötigt, um einen zusammenhängenden Graphen mit disjunkten Pfaden zu überdecken. In einem EULER-Graphen ist diese Zahl offensichtlich 1. Besitzt ein Graph  $\mathcal{G}$  keinen EULER-Pfad, so existieren  $k$  Knoten ungeraden Grades. Man überlegt sich leicht, daß  $k$  gerade zu sein hat. Darüber hinaus muß jeder Knoten ungeraden Grades einer der Endpunkte eines Pfades in der Überdeckung sein. Also existieren mindestens  $\frac{k}{2}$ -viele Pfade in der Überdeckung und man überlegt sich, daß  $\frac{k}{2}$ -viele Pfade auch ausreichen, um  $\mathcal{G}$  zu überdecken. Diese Fragestellung kann man auf gerichtete Multigraphen ausdehnen. Hier zeigt sich, daß ein EULER-Graph natürlich wieder zusammenhängend zu sein hat und darüber hinaus jeder Knoten  $v$  von  $\mathcal{G}$  genausoviele eingehende wie ausgehende Kanten besitzen muß.

Man kann dem Problem des EULER-Graphen sogar eine gewisse praktische Bedeutung unterlegen. Das Problem tritt beispielsweise auf, wenn es darum geht, eine Führungslinie in einem Museum so zu legen, daß jedes Exponat besucht wird, ohne daß Wege doppelt zurückgelegt werden müssen.

Die natürlichere Fragestellung hier wäre allerdings die folgende: Läßt sich ein Weg finden, der zu seinem Ausgangspunkt (dem Ein- und Ausgang) zurückführt und jedes Exponat genau einmal besucht. In Terminen der Graphentheorie formuliert wäre dies die Frage nach der Existenz eines Kreises, der alle Knoten des Graphen enthält. Solche Kreise nennt man HAMILTONSche Kreise. Ein Graph, der einen HAMILTONSchen Kreis zuläßt, heißt ein HAMILTONScher Graph. Diese Definitionen übertragen sich auch auf gerichtete Graphen.

HAMILTONSche  
Kreise und -Graphen

Für das Vorliegen eines HAMILTONSchen Graphen gibt es kein so ein-

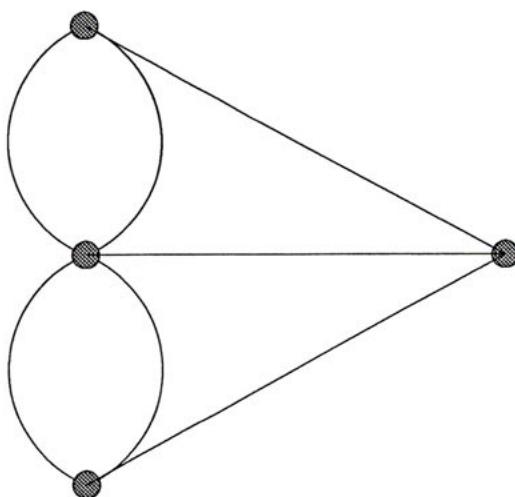


Abb. A.3 - 2: Multigraph des Königsberger Brückenproblems

---

faches Kriterium wie für das eines EULERSchen Graphen. Es ist natürlich klar, daß auch ein HAMILTONscher Graph zusammenhängend sein muß. Darüber hinaus sind aber keine effizienten Algorithmen für das Auffinden eines HAMILTONschen Kreises bekannt. Der Grund dafür ist die *NP*-Vollständigkeit dieses Problems. Das HAMILTONsche Problem wird oft in anschaulicher Weise dargestellt. Eine Form ist das Partyproblem, das die Frage stellt, unter welchen Bedingungen es möglich ist, die Gäste einer Party so um einen runden Tisch zu plazieren, daß jeder zwischen zwei Freunden zu sitzen kommt. Der Graph, der hier zu betrachten ist, hat die Gäste als Knoten und eine Kante zwischen je zwei Gästen, die miteinander befreundet sind.

Es gibt jedoch noch viel einfacher anmutende Graphenprobleme, die sich als *NP*-vollständig herausstellen. Ein Beispiel dafür ist die Frage nach der Existenz einer Knotenüberdeckung. Dabei heißt eine Menge  $A \subseteq V$  eine Knotenüberdeckung eines Graphen  $\mathcal{G} = (V, E)$ , wenn für jede Kante  $\{v_1, v_2\} \in E$  schon  $v_1 \in A$  oder  $v_2 \in A$  gilt. Auch dieses Problem ist *NP*-vollständig.

Eine weiteres Problem besteht darin, den Kern eines gerichteten Graphen  $\mathcal{G} = (V, E)$  zu bestimmen. Dabei heißt eine Menge  $K \subseteq V$  ein *Kern* des Graphen, wenn die Knoten des Kernes  $K$  paarweise nichtbenachbart sind und jeder Knoten  $v$  des Graphen entweder selbst im Kern liegt oder es einen Pfeil von einem Knoten des Kernes nach  $v$  gibt. Auch hier handelt es sich um ein *NP*-vollständiges Problem.

Neben diesen hier angeführten Beispielen lassen sich noch eine ganze Legion *NP*-vollständiger Probleme finden, die sich zum Teil als Übersetzungen praktisch auftretender Probleme in die Sprache der Graphentheorie ergeben. Vergleiche dazu beispielsweise [HOP79].

### A.4 Markierte und gefärbte Graphen

Eine wichtige Rolle spielen *markierte* und *gefärbte Graphen*. Dabei können sowohl Knoten als auch Kanten markiert werden. Eine *Knoten-*

Knotenmarkierungen

*markierung* für einen Graphen  $\mathcal{G} = (V, E)$  ist eine Abbildung

$$L_V: V \longrightarrow \mathcal{M}_V,$$

wobei wir  $\mathcal{M}_V$  die Menge der *Knotenmarken* nennen. Analog ist eine *Kantenmarkierung* eine Abbildung

Kantenmarkierungen

$$L_E: E \longrightarrow \mathcal{M}_E$$

von den Kanten in die Menge der *Kantenmarken*. Ein *markierter Graph* ist dann einer, der kanten- und/oder knotenmarkiert ist.

Wir haben markierte Graphen bereits in der Form von Flußdiagrammen für Maschinenprogramme kennengelernt. Ein Flußdiagramm kann als ein markierter gerichteter Graph aufgefaßt werden, dessen Knotenmarken die Programmarken sind und dessen Kanten mit Befehlen markiert sind.

Markierte Graphen sind allgegenwärtig. So lassen sich formale Beweise, wie sie durch Kalküle der Prädikatenlogik gegeben werden, als markierte (gerichtete) Bäume auffassen. Die bewiesene Formel erscheint dabei an der Wurzel des Baumes, die Axiome an den Spitzen. Ebenso handelt es sich bei Netzplänen um markierte Graphen. Auch viele Optimierungsprobleme lassen sich durch markierte Graphen darstellen. Ein wohlbekanntes Optimierungsproblem in einem markierten Graphen ist das *Problem des Handelsreisenden*. Anschaulich geht es in diesem Problem darum, daß ein Handelsreisender alle Städte seines Handelsgebietes bereisen und zu seinem Ausgangsort zurückkehren soll, wobei die Kosten der Reise möglichst gering zu halten sind. Graphentheoretisch geht man dazu von einem vollständigen kantenmarkierten Graphen aus. Die Markierungen in  $\mathcal{M}_E \subseteq \mathbb{R}$  bezeichnet man der besseren Anschaulichkeit wegen oft als *Kosten*. Das Problem des Handelsreisenden besteht nun darin, einen HAMILTONSchen Kreis in dem Graphen so zu finden, daß die Summe der Kosten der durchlaufenen Kanten minimal wird. Es ist nicht sonderlich überraschend, daß dieses Problem *NP*-vollständig ist.

Das Problem des Handelsreisenden

Eine Knotenmarkierung  $L_V$  eines Graphen heißt eine *Färbung*,

## A. Grundlagen der Graphentheorie

---

wenn für zwei adjazente Knoten  $u, v$  immer  $L_V(u) \neq L_V(v)$  gilt. Anschaulich hat man sich eine Färbung so vorzustellen, daß  $\mathcal{M}_V$  eine Menge von Farben ist und die Knoten des Graphen so zu färben sind, daß benachbarte Knoten immer verschiedene Farben erhalten. Die *chromatische Zahl* eines Graphen ist definiert durch

$$\kappa(\mathcal{G}) := \min\{|F| \mid (\exists \phi)[\phi: V_{\mathcal{G}} \longrightarrow F \text{ ist Färbung}]\},$$

wobei  $|F|$  die Kardinalität der Menge  $F$  der Farben bezeichnet. Ein berühmtes Problem in diesem Zusammenhang war die Frage nach der chromatischen Zahl *planarer Graphen*. Planare Graphen sind Graphen, die sich in der Ebene zeichnen lassen, ohne daß sich dabei Kanten "kreuzen". (Exakter müßte man den Begriff eines topologischen Graphen einführen. Dies sind Graphen, deren Kanten homöomorphe Bilder des Einheitsintervalls  $[0, 1]$  – sogenannte Jordan Bögen – sind. Planare Graphen sind dann die topologischen Graphen des  $\mathbb{R}^2$ .) Es ist relativ leicht zu zeigen, daß die chromatische Zahl planarer Graphen kleiner oder gleich 5 sein muß. Andererseits läßt sich der 4-Simplex  $S(4)$ , der planar ist, mit nur vier Farben färben, woraus folgt, daß für einen beliebigen planaren Graphen  $\mathcal{G}$  zumindest

$$4 \leq \kappa(\mathcal{G}) \leq 5$$

gilt. Die Frage war nun, ob man mit vier Farben immer auskommen kann. Daher ist dieses Problem auch als das *Vierfarbenproblem* bekannt. Da man jeden planaren Graphen als eine Landkarte auffassen kann, deren Länder die Knoten des Graphen sind, wobei Länder benachbarter Knoten eine gemeinsame Grenze erhalten, läßt sich das Vierfarbenproblem anschaulich gut beschreiben. Die Aufgabe besteht darin, eine Landkarte so zu färben, daß benachbarte Länder immer verschiedenen Farben erhalten. Die Frage ist, ob sich dies immer mit vier Farben bewerkstelligen läßt. Neben seiner Schwierigkeit ist sicher auch seine anschauliche und allgemeinverständliche Formulierbarkeit ein Grund für die Berühmtheit des Vierfarbenproblems. Es wurde erstmals im Jahre 1850 von F. GUTHRIE explizit erwähnt und hat über hundert Jahre allen intensiven Beweisversuchen (sowohl von Fachleuten als auch von Laien) getrotzt. Erst im Jahre 1976 gelang K. APPEL, W. HAKEN indexHAKEN, W. und J. KOCH nach Vorarbeiten von H. HEESCH der Beweis, daß vier Farben ausreichen. Interessant an diesem Beweis ist die Tatsache, daß es sich um den ersten mathematischen Beweis handelt, der so komplexe Berechnungen erfordert, daß er nur mit Hilfe von Computern geführt werden kann.

Färbungsprobleme sind aber auch im Sinne der Komplexitätstheorie schwierig. So handelt es sich bereits bei der Entscheidung, ob ein vorgegebener Graph eine chromatische Zahl kleiner oder gleich drei hat,

um ein *NP*-vollständiges Problem. Daß sich nichtdeterministisch in polynomialer Zeit alle Dreifärbungen ausprobieren lassen, ist offensichtlich. Andererseits läßt sich das Erfüllbarkeitsproblem für boolesche Terme in konjunktiver Normalform auf das Dreifärbungsproblem reduzieren. Dazu beobachtet man nur, daß die in Abbildung A.4 - 1 angegebenen

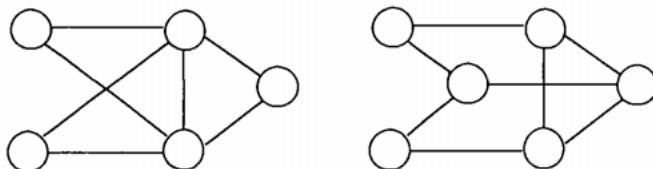


Abb. A.4 - 1 : Ein schwaches  $\wedge$ - und  $\vee$ - Gatter in dreifärbbaren Graphen

Graphen bei der Verfügbarkeit von höchstens drei Farben wie  $\wedge$ - und  $\vee$ -Gatter in dem Sinne arbeiten, daß der rechts stehende Outputknoten genau dann mit der Farbe  $W$  gefärbt werden kann, wenn beide bzw. mindestens einer der links stehenden Inputknoten mit  $W$  gefärbt war. Jedem booleschen Term  $t$  in konjunktiver Normalform entspricht dann ein Graph  $\mathcal{G}_t$ , der sich in polynomialer Zeit aus  $t$  berechnen läßt, so daß  $t$  genau dann erfüllbar ist, wenn der Graph  $\mathcal{G}_t$  dreifärbbar ist.



---

## B. Literaturempfehlungen

Bei einem Titel wie *Mathematische Grundlagen der Informatik* besteht immer die Gefahr, die Erwartungen der Leser zu enttäuschen, die eigentlich ein Buch über *mathematische Werkzeuge der Informatik* erwartet haben. Dem Leser, dem es so ergeht, sei die Lektüre des Buches [OBE76] von W. OBERSCHELP und D. WILLE empfohlen, in dem eine knapp gehaltene Einführung in die mathematischen Werkzeuge des Informatikers in gut lesbarer Weise gegeben wird.

Der vorliegende Band befaßte sich mit der mathematischen Grundlage der Informatik, der Berechenbarkeitstheorie. Wer mehr über die Ursprünge der Berechenbarkeitstheorie erfahren möchte, dem sei die Lektüre der Originalarbeiten empfohlen, wie sie beispielsweise in der Anthologie von M. DAVIS [DAV64] zusammengestellt sind. Man wird erstaunt sein, wie modern viele der dort nachgedruckten Artikel bereits wirken.

Lehrbücher über Berechenbarkeitstheorie, die meist weit über das hinausgehen, was hier zusammengestellt wurde, gibt es in großer Zahl. Eine einfach zu lesende Einführung findet man in H. HERMES Buch [HERM61]. Das klassische Werk ist [ROG67] von H. ROGERS JR., in dem (in englischer Sprache) eine profunde Einführung in die Theorie der Berechenbarkeit gegeben wird, die bis zur Theorie der analytischen Hierarchie führt. Die Theorie wird dort unter mathematischen Aspekten entwickelt. Das Studium dieses Buches erfordert allerdings intensive Mitarbeit des Lesers. Dazu findet man eine große Zahl sehr schöner Übungsaufgaben in allen Schwierigkeitsgraden.

Modernere Darstellungen der Theorie findet man beispielsweise in den Büchern von M. LERMAN [LER83] und R. SOARE [SOA85]. Beide Bücher behandeln die Theorie umfassend, stellen aber ebenfalls hohe Ansprüche an die Mitarbeit des Lesers. Besondere Betonung liegt auf der Theorie der Unlösbarkeitsgrade, die wir nur am Rande erwähnt haben.

Ein komplette Darstellung des heutigen Standes der Rekursions-theorie beabsichtigt P. ODDIFREDDI. Der bereits erschienene erste Band [ODD89] stellt auf ca. 600 Seiten alle wesentlichen Ergebnisse der sogenannten klassischen Rekursionstheorie dar.

Wer Interesse an der Theorie der rekursionstheoretischen Hierarchien hat, die hier nur kurz gestreift wurde, sei auf das Buch von P. HINMAN [HINM78] verwiesen, das diese Theorie erschöpfend behandelt.

Eine speziell für Informatiker geschriebene Einführung in die Berechenbarkeitstheorie findet sich in U. SCHÖNINGS Buch [SCHÖ92]. Der

## B. Literaturempfehlungen

---

informatische Aspekt der Berechenbarkeitstheorie steht auch im Mittelpunkt des Buches [SCHN74] von C. P. SCHNORR. Dort findet sich auch ein abstraktes Maschinenkonzept, das dem ähnelt, das wir in dem vorliegenden Buch vorgestellt haben.

Die Zahl der Originalarbeiten im Bereich der Berechenbarkeitstheorie, auch im Hinblick auf die Querverbindungen zur Informatik ist so groß, daß wir darauf verzichten müssen, einzelne Arbeiten herauszustellen. Eine ausführliche Bibliographie findet sich beispielsweise in [ODD89].

Neben den Klassikern [CUR58] von H. B. CURRY, R. FEYS and W. CRAIG sowie [CUR72] von H. B. CURRY, J. R. HINDLEY und J. P. SELDIN ist das moderne Standardwerk für den  $\lambda$ -Kalkül H. BARENDRREGTS Buch [BARD81]. Der dort dargestellte Stoff geht natürlich weit über das hinaus, was wir hier anreißen konnten.

Informationen über  $\lambda$ -Kalküle mit Typen bieten das Buch von J. Y. GIRAD, Y. LAFONT und P. TAYLOR [GIR89] sowie der Artikel [BARD92] von H. BARENDRREGT. Dort finden sich auch weiterführende Literaturhinweise.

Unlösbare Probleme werden ausführlich in [HOP79] behandelt. Die Unentscheidbarkeit der Prädikatenlogik findet man in praktisch jedem Lehrbuch der Mathematischen Logik. Man vergleiche hierzu die Literaturhinweise in Band 1.1 dieser Serie.

Das Standardwerk zur Einführung in die Komplexitätstheorie ist noch immer das Buch [HOP79] von J.E. HOPCROFT und J. D. ULLMAN, obwohl dessen Schwerpunkt mehr in der Theorie der formalen Sprachen liegt. Ein neueres Werk ist [BAL88] von J. L. BALCÁZAR, J. DÍAZ und J. GABARRÓ, von dem in der Zwischenzeit auch der zweite Band erschienen ist, der speziellere Fragestellungen behandelt. Immer noch lesenswert ist die Einführung [PAUL78] von W. PAUL. Als Führer zu den Originalarbeiten kann das Literaturverzeichnis in [BAL88] dienen.

Eine knappe Einführung in den für den Informatiker relevanten Teil der Graphentheorie wird in [OBE76] gegeben. Hier ist auch das Buch [DÖR73] von W. DÖRFER und J. MÜHLBACHER zu nennen. Ein klassisches Werk ist [ORE62] von O. ORE. Eine etwas neuere Darstellung der Graphentheorie in deutscher Sprache ist [HAL89] von R. HALIN.

# Literaturverzeichnis

- [APP77a] K. Appel, W. Haken: Every planar graph is four-colorable. Part I: Discharging. *Illinois Journal of Mathematics* 21 (1977) pp. 218 - 314
- [APP77B] K. Appel, W. Haken, J. Koch: Every planar graph is four-colorable. Part II: Reducibility. *Illinois Journal of Mathematics* 21 (1977) pp. 491 - 567
- [BAK75] T. Baker, J. Gill, R. Solovay: Relativizations of the  $P =? NP$  question. *SIAM Journal on Computing* 4 (1975) pp. 431 - 442
- [BAL88] J. L. Balcázar, J. Díaz, J. Gabarró: Structural Complexity I. Springer, Berlin - Heidelberg - New York 1988
- [BARD81] H. P. Barendregt: The Lambda Calculus. North-Holland, Amsterdam - New York 1981
- [BARD92] H. P. Barendregt: Lambda Calculi with Types. In: S. Abramsky et al.: *Handbook of Logic in Computer Science*, Vol. 2. Oxford University Press, Oxford 1992
- [BARW75] J. Barwise: Admissible Sets and Structures. Springer, Berlin - Heidelberg 1975
- [BLUM67] M. Blum: A machine-independent theory of the complexity of recursive functions. *Journal of the Association for Computing Machinery* 14 (1967) pp. 322 - 336
- [BÖRG85] E. Börger: Berechenbarkeit, Komplexität, Logik. Vieweg, Braunschweig 1985
- [BÖRG89] E. Börger: Computability. Complexity, Logic. North Holland, Amsterdam - New York -Oxford - Tokyo 1989
- [BOR72] A. Borodin: Computational complexity and the existence of complexity gaps. *Journal of the Association for Computing Machinery (ACM)* 19 (1972) pp. 158 - 174

- [CHUR36a] A. Church: An unsolvable problem of elementary number theory. *The American Journal of Mathematics* 58 (1936) pp. 345 - 363
- [CHUR36b] A. Church: A note on the Entscheidungsproblem. *Journal of Symbolic Logic* 1 (1936) pp. 40 - 41
- [CUR58] H. B. Curry, R. Feys, W. Craig: Combinatory Logic, Vol. I. North-Holland, Amsterdam - New York 1958
- [CUR72] H. B. Curry, J. R. Hindley, J. P. Seldin: Combinatory Logic, Vol. II. North-Holland, Amsterdam - New York 1972
- [DAV64] M. Davis: The Undecidable. Raven Press, New York 1965
- [DÖR73] W. Dörfler, J. Mühlbacher: Graphentheorie für den Informatiker. Sammlung Göschen Bd. 6016. De Gruyter, Berlin - New York 1973
- [FISCH74] M. J. Fischer, M. O. Rabin: Super exponential complexity of Presburger arithmetic. In: R.M. Karp (ed.): Complexity of Computation. Proceedings of a Symposium in Applied Mathematics of the AMS and SIAM. Vol. 7. American Mathematical Society, Providence, Rhode Island 1973, pp. 27 - 41
- [FRIED57] R. M. Friedberg: Two recursively enumerable sets of incomparable degree of unsolvability. *Proceedings of the National Academy of Sciences of the United States of America* 43 (1957) pp. 236 - 238
- [GÖD30] K. Gödel: Die Vollständigkeit der Axiome des logischen Funktionenkalküls. *Monatshefte für Mathematik und Physik* 37 (1930) pp. 394 - 360
- [GÖD31] K. Gödel: Über formal unentscheidbare Sätze der "Principia Mathematica." *Monatshefte für Mathematik und Physik* 38 (1931) pp. 173 - 198
- [GÖD33] K. Gödel: Zum Entscheidungsproblem des logischen Funktionenkalküls. *Monatshefte für Mathematik und Physik* 40 (1933) pp. 433 - 443
- [GÖD58] K. Gödel: Über eine bisher noch nicht benutzte Erweiterung des finiten Standpunktes. *Dialectica* 12 (1958) pp. 280 - 287
- [GIR71] J.-Y. Girard: Une extension de l'interprétation de Gödel et son application à l'élimination des coupures dans

- l'analyse et la théorie des types. In: J. E. Fenstadt, (ed.): Proceedings of the Second Scandinavian Logic Colloquium. North-Holland, Amsterdam - New York 1971 pp. 63 -92
- [GIR89] J.-Y. Girard, Y. Lafont, P. Taylor: Proofs and Types. Cambridge University Press, Cambridge 1989
- [HAL89] R. Halin: Graphentheorie. Wissenschaftliche Buchgesellschaft, Darmstadt 1989
- [HERM61] H. Hermes: Aufzählbarkeit, Entscheidbarkeit, Berechenbarkeit: Einführung in die Theorie der rekursiven Funktionen. Springer, Heidelberg - New York 1961
- [HIL26] D. Hilbert: Über das Unendliche. Mathematische Annalen 95 (1926) pp. 161 - 190
- [HIN72] J. R. Hindley, B. Lercher, J. P. Seldin: Introduction to Combinatory Logic. Cambridge University Press, Cambridge 1972
- [HINM78] P. G. Hinman: Recursion-theoretic Hierarchies. Springer, Heidelberg - New York 1978
- [HOP79] J. E. Hopcroft, J. D. Ullman: Introduction to Automata Theory, Languages and Computation. Addison-Wesley, Reading, Mass. 1979
- [KLEE36] S. C. Kleene: General recursive functions of natural numbers. Mathematische Annalen 112 (1936) pp. 727 - 742
- [KLEE43] S. C. Kleene: Recursive predicates and quantifiers. Transactions of the American Mathematical Society 53 (1943) pp. 41 - 73
- [KREO91] H.-J. Kreowski: Logische Grundlagen der Informatik. Handbuch der Informatik, Bd. 1.1. Oldenbourg, München 1991
- [LER83] M. Lerman: Degrees of Unsolvability. Springer, Berlin - Heidelberg - New York 1983
- [MUCH57] A. A. Muchnik: Negative solution of the reducibility problem of Post (Russisch). Uspekhi Matematicheskikh Nauk (Fortschritte der Mathematik)12/2 (74) (1957) pp. 215-216
- [OBE76] W. Oberschelp, D. Wille: Mathematischer Einführungskurs für Informatiker. Teubner, Stuttgart 1976

## Literaturverzeichnis

---

- [ODD89] P. Odifreddi: Classical Recursion Theory. North-Holland, Amsterdam - New York 1989
- [ORE62] O. Ore: Theory of Graphs. American Mathematical Society, Providence, Rhode Island 1962
- [PAUL78] W. Paul: Komplexitätstheorie. Teubner, Stuttgart 1978
- [PET34] R. Péter: Über den Zusammenhang der verschiedenen Begriffe der rekursiven Funktionen. Mathematische Annalen 110 (1934) pp. 612 - 632
- [PET35] R. Péter: Konstruktion nichtrekursiver Funktionen. Mathematische Annalen 111 (1935) pp. 42 - 60
- [POST36] E. Post: Finite combinatory processes. Formulation I. The Journal of Symbolic Logic 1 (1936) pp. 103 - 105
- [POST41] E. Post: Absolutely unsolvable problems and relatively undecidable propositions. Account of an anticipation. In: [DAV64] pp. 340 - 433
- [POST44] E. Post: Recursively enumerable sets of positive integers and their decision problems. Bulletin of the American Mathematical Society 50 (1944) pp. 284 - 316
- [POST47] E. Post: Recursive unsolvability of a problem of Thue. The Journal of Symbolic Logic 12 (1947) pp. 1 - 11
- [PRES29] M. Presburger: Über die Vollständigkeit eines gewissen Systems der Arithmetik ganzer Zahlen, in welcher die Addition als einzige Operation hervortritt. In: F. Leja (ed.): Comptes Rendus de 1. Congrès des Mathématiciens des Pays Slaves. Warschau 1929 pp. 92 - 101, 395
- [RICE53] H. G. Rice: Classes of recursively enumerable sets and their decision problem. Transaction of the American Mathematical Society 74 (1953) pp. 385 - 366
- [RICE56] H. G. Rice: On completely recursively enumerable classes and their key arrays. The Journal of Symbolic Logic 21 (1956) pp. 304 - 308
- [ROG67] H. Rogers jr.: Theory of Recursive Functions and Effective Computability. McGraw Hill, New York 1967
- [SACH70] H. Sachs: Einführung in die Theorie der endlichen Graphen, I, II. Teubner, Stuttgart 1970, 1972

- [SAV70] W. J. Savitch: Relationship between nondeterministic and deterministic tape complexity. *Journal of Computer and System Sciences* 4 (1970) pp. 177 - 192
- [SCHÜ77] K. Schütte: *Proof Theory*. Springer, Heidelberg - New York 1977
- [SCHÖ92] U. Schöning: *Theoretische Informatik kurzgefaßt*. Bibliographisches Institut - Wissenschaftsverlag, Mannheim - Leipzig - Wien - Zürich 1992
- [SCHN74] C. P. Schnorr: *Rekursive Funktionen und ihre Komplexität*. Teubner, Stuttgart 1974
- [SHOE67] J. R. Shoenfield: *Mathematical Logic*. Addison Wesley, Reading, Mass. 1967
- [SOA85] R. Soare: *Recursively Enumerable Sets and Degrees: A Study of Computable Functions and Computably Generated Sets*. Springer, Heidelberg - New York 1985
- [THUE14] A. Thue: Probleme über die Veränderung von Zeichenreihen nach gegebenen Regeln. *Videnskaps Selskapet i Kristiania. Skrifter Utgit. 1 Mathematisk-Naturvedenskapelig Klasse* 10 (1914) p. 34
- [TUR36] A. M. Turing: On computable numbers with an application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society*, ser.2, 42 (1936) pp. 230 - 265
- [TUR39] A. M. Turing: Systems of logic based on ordinals. *Proceedings of the London Mathematical Society*, ser. 2, 45 (1939) pp. 162 - 228
- [TUT84] W. T. Tutte: *Graph Theory*. Cambridge University Press, Cambridge 1984

# Register

## Symbole

$A \equiv_T B$ , 116	$\mathbf{F}_{\text{tw}}^{(k,l)}$ , 28
$A \leq_p$ , 158	$\mathbf{F}$ , 101
$A^*$ , 26	IF THEN ELSE , 101
$A^+$ , 26	$\mathcal{C}(P)$ , 44
$B \leq_T A$ , 116	$NP$ , 155
$K$	$NSPACE(S)$ , 142
	$NTIME(T)$ , 142
	NULL, 102
Grundkombinator, 93	$\mathbf{Num}$ , 106
Halteproblem, 76	$\Omega_\infty(g)$ , 143
$K(C)$ , 22	$\Omega$ , 95
$M \models A$ , 130	$Ind(\mathbf{P})$ , 65
$O(g)$ , 143	$\mathbf{P}$ , 57
$R_1 \leq_1 R_2$ , 117	$\mathbf{P}_{\text{r-RM}}^{(k,l)}$ , 29
$R_1 \leq_m R_2$ , 117	$\mathbf{P}_{\text{RM}}$ , 30
Grundkombinator, 93	$\mathbf{P}_{\text{tw}}^{(k,l)}$ , 28
$T^n$ , 68	$\mathbf{PR}$ , 39
$U$ , 68	$Ind(Pri)$ , 54
$U =_\beta V$ , 95	$\Phi^n$ , 69
$U \rightarrow V$ , 92	$\Pi^n$ , 101
$U \rightarrow_n V$ , 92	$\Pi_j^n$ , 101
$U \rightarrow_1^* V$ , 98	$P$ , 155
$U \rightarrow^* V$ , 98	$RS_C(k, z)$ , 22
$W_e^n$ , 73	$Rec(g, h)$ , 38
$[e]$ , 54	$Rz_C(d)$ , 25
$\mathbf{Ap}$ , 106	$Sb$ , 99
$DSPACE(S)$ , 142	$\Sigma^*$ , 118
$DTIME(T)$ , 142	$\Sigma^+$ , 118
$\mathbf{F}$	$Sub(g, h_1, \dots, h_m)$ , 38
Klasse der rekursiven Funktionen, 57	$\Theta(g)$ , 143
System polymorpher Funktionale, 113	$T$ , 101
$\mathbf{F}_{\text{r-RM}}^{(k,l)}$ , 30	$\alpha(\mathcal{G})$ , 168

---

$\beta$ -abgeschlossen,	106	$\sharp P$ ,	127
$S \models F$ ,	129	$\sharp f$ ,	127
$\chi_P$ ,	44	$\sigma \rightarrow \tau$ ,	108, 112
$(x)_i$ ,	50	$\underline{n}$ ,	37
$[M]$ ,	99	$\{e\}^n(x_1, \dots, x_n)$ ,	69
$[F]$ ,	53	$a \dashv b$ ,	29
$\text{grad}_G(v)$ ,	168	$f(x) \simeq u$ ,	24
$BEQ$ ,	27	$f(z) \simeq g(z)$ ,	24
$BEQ(z)$ ,	25	$f = h + O(g)$ ,	143
$BEQ_j$ ,	29	$f: Q \longrightarrow_p Z$ ,	23
$CLR$ ,	25	$k$ -konjunktive Normalform,	161
$DEC_j$ ,	29	$l(w)$ ,	26
$INC_j$ ,	29	$m$ -reduzibel,	117
$LFT$ ,	25	$n$ -Simplex,	169
$PRT$ ,	27	$o(g)$ ,	143
$PRT(z)$ ,	25	$t^S[\Phi]$ ,	128
$Rel_{r.a.}$ ,	73	$u \xrightarrow[n]{R} v$ ,	119
$Rel_{r.}$ ,	73	$u \xrightarrow[1]{R} v$ ,	119
$RHT$ ,	25	$x \frown y$ ,	50
$\lambda x . Z$ ,	91	$RS_c^n(k, z)$ ,	22
$\lambda x^\sigma . T$ ,	109	$\mathcal{L}_G$ ,	120
$SPACE(S)$ ,	138	$\mathcal{O}(P_1, \dots, P_n)$ ,	47
$TIME(T)$ ,	137	$Res_c(d)$ ,	25
$QBF$ ,	162	$T$ ,	111
$G_f$ ,	39, 79	1-Reduzibilität,	117
$\kappa(\mathcal{G})$ ,	168		
$lh(x)$ ,	50		
$\models A$ ,	131		
$\mu P$ ,	56		
$\mu f$ ,	56		
$\mu z \leq y .$ ,	46		
$\mu z . P(\vec{x}, z)$ ,	57		
$\mu z . [f(\vec{x}, z) \simeq 0]$ ,	57		
$\mu_b(P)$ ,	45		
$\omega(g)$ ,	143		
$\overline{f}$ ,	51		
$\phi_n$ ,	55		
$\langle R \rangle$ ,	71		
$\langle f \rangle$ ,	60		
$\langle x_1, \dots, x_n \rangle$ ,	50		
$sg$ ,	43		
$\overline{sg}$ ,	43		

### Stichworte

- Abbildung , *siehe* Funktion  
abgeleitetes Wort, 120  
Abschluß einer abstrakten Basismaschine, 33  
abstrakte Komplexitätsmaße, 152  
abstrakte Basismaschine, 19  
Ackermannsche Funktion, 82  
adjazente Knoten, 165  
allgemeingültige Formelmengen, 130  
Alphabet, 26  
Anweisungen für Basismaschinen, 20  
Arbeitsfeld, 25  
arithmetische Differenz, 43  
arithmetische Hierarchie, 75
- Basismaschine, 19  
Baum, 169  
geordneter, 169  
Belegung freier Variablen, 128  
benachbarte Knoten, 165  
Berechnungsprädikat, 66  
Beschleunigungssatz, 154  
beschränkte Quantifikation, 46  
beschränkte Rekursion, 158  
beschränkter Suchoperator, 45  
beweisbar rekursive Funktion  
    der Arithmetik zweiter Stufe, 113  
    der PEANO Arithmetik, 112  
Beweistheorie, 83  
Bild (range) einer Funktion, 49  
Binärdarstellung natürlicher Zahlen, 158  
boolesche Belegung, 160  
boolesche Operationen, 45, 160
- c-Homomorphismus, 171  
charakteristische Funktion, 44
- Chomsky Grammatik, 120  
chromatische Zahl, 176  
Churchsche These, 64  
Cooksche Hypothese, 156
- Definitionsbereich einer partiellen Funktion, 23  
Dekodierungsfunktion , 48  
direkter Ableitungsschritt, 119  
diskrete Mathematik, 162
- einfacher Kantenzug, 168  
Endkonfiguration, 23  
endliche Produkte, 43  
endliche Typen, 108  
Endmarke, 23  
Entscheidbarkeit, 44  
Entscheidungsproblem, 12, 44  
Epimorphismus  
    kontraktierender, 171  
    von Graphen, 170  
erfüllbare Formeln, 129  
erfüllbar in einer Struktur, 130  
erfüllbare Formelmengen, 130  
Erfüllbarkeitsproblem, 160  
erweiterte boolesche Operationen, 47  
Euler-Pfade, 172  
extensionale Auffassung von Funktionen, 39  
extensionale Gleichheit von Funktionen, 39
- Fakultätsfunktion, 43  
Fallunterscheidungsfunktionen, 43  
Flußdiagramme, 20  
Formeln  
    boolesche, 160  
    der Prädikatenlogik, 127  
    quantifizierte boolesche, 162

- freie Variablen, 127  
 freie Aussagenvariable, 163  
**Funktion**  
      $\mu$ -partiellrekursive, 57  
     geeignet universelle, 80  
     maschinenberechenbare, 25, 28,  
         30  
     partielle, 23  
     partiellrekursive, 57, 64  
     platzbeschränkt berechenbare,  
         138  
     platzkonstruierbare, 147  
     polynomiale berechenbare, 137  
     primitiv rekursive, 39  
     rekursive, 57  
     totale, 23  
     turingberechenbare, 28  
     universelle, 69, 78  
     vektorwertige primitiv rekur-  
         sive, 60  
     vektorwertige rekursive, 60  
     zeitbeschränkt berechenbare,  
         138  
     zeitkonstruierbare, 147  
**Funktionale endlicher Typen**, 111  
**funktionale Prozedur**, 32  
**Funktionsterme**  
     partiell rekursive, 57  
     primitiv rekursive, 40  
**Funktionszeichen**, 127  
**Färbung von Graphen**, 175  
  
 gebundene Aussagenvariablen, 163  
 gebundene Variablen, 127  
 geschlossene  $\lambda$ -Terme, 93  
 geschlossene Formeln und -Terme,  
     127  
 geschlossene quantifizierte boo-  
     lesche Formel, 163  
 geschlossener Kantenzug, 167  
**Grammatik**, 120  
 kontextsensitive, 124  
  
 kontextfreie, 125  
 reguläre, 125  
 Typ 0, 124  
 Typ 1, 124  
 Typ 2, 125  
 Typ 3, 125  
**Graph**  
     einer partiellen Funktion, 79  
     einer Funktion, 39  
     endlicher, 167  
     gefärbter, 175  
     gemischter, 167  
     gerichteter, 166  
     markierter, 175  
     planarer, 176  
     regulärer, 169  
     vollständiger, 168  
     zusammenhängender, 168  
**Graphentheorie**, 165  
**Grundfunktionen**, 38  
**Grundkombinatoren**, 93  
**Grundoperationen**, 39  
 Gödelnummer eines Funktionster-  
     mes, 53  
 gültige Formelmengen, 130  
  
 Halteproblem, 78  
 Hamiltonsche Kreise, 173  
 Hamiltonscher Graph, 173  
**Homomorphismus**  
     von Basismaschinen, 30  
     von Graphen, 170  
  
**Index**  
     einer partiellrekursiven Funk-  
         tion, 68  
     einer primitiv rekursiven Funk-  
         tion, 54  
 Individuenkonstanten, 127  
 intensionale Auffassung von Funk-  
     tionen, 39  
 intensionale Gleichheit, 40

## Register

---

- Interpretationen von Termen und Formeln, 128  
inzidente Knoten, 165  
Isomorphismen, 170  
Junktoren, 127  
Kanten, 165  
Kantenkontraktion, 172  
Kantenmarken, 175  
Kantenmarkierung, 175  
Kantenzug, 167  
Kennmarke, 20  
Kern eines gerichteten Graphen, 174  
Knoten, 165  
Knotenmarken, 175  
Knotenmarkierung, 175  
Kodierung, 48  
Kodierungsfunktion, 48  
Kombinationen, 93  
kombinatorische Logik, 90  
Kompaktheitssatz, 131  
Komplement eines Prädikats, 44  
Konfigurationsraum, 22  
Konkatenation von Kodes, 50  
konsistente Formelmengen, 130  
konstante Funktionen, 38  
Kontraktion  
    einer Relation, 71  
    einer Funktion, 60  
Konversion von  $\lambda$ -Termen, 92  
Kreis in einem Graphen, 168  
  
limes inferior, 144  
Literal, 161  
logische Folgerung, 130  
logische Zeichen, 127  
lokale Kantenzahl, 168  
Länge  
    eines Kantenzuges, 167  
    eines Wortes, 26  
  
Längenfunktion einer Kodierung, 49  
  
m-Grad, 117  
m-vollständige Relationen, 117  
many-one reduzibel, 117  
Marken, 20  
Maschenlemma, 96  
Maschine  
    abstrakte, 21  
    deterministische, 22  
    nichtdeterministische, 138  
    universelle, 69  
Modifikationsanweisungen, 20  
Modifikationsinstruktionen, 19  
Multigraphen, 167  
  
Nachfolgerfunktion, 37  
natürliche Zahl, 37  
nichtlogische Zeichen, 127  
Normalform  
    disjunktive, 161  
    eines  $\lambda$ -Terms, 95  
    konjunktive, 161  
Normalformensatz von KLEENE, 68  
  
one-one-Reduzibilität, 117  
Ordnung eines Graphen, 168  
  
p-Reduzibilität, 158  
Parser, 126  
partiellrekursiv, *siehe* Funktion  
partiellrekursiv in einer Funktion, 115  
partiellrekursive Funktionale, 81  
Pfad, 167  
Pfeile, 166  
polymorphe Funktionale, 113  
polymorphe Typen, 112  
polynomial  $m$ -reduzibel, 158  
positive boolesche Operationen, 73

- 
- positive Entscheidbarkeit, 71  
 Presburger Arithmetik, 164  
 primitiv rekursive Basismaschinen, 60  
 primitiv rekursive Kodierung, 49  
 primitive Rekursion, 38  
*Principia Mathematica*, 12  
 Problem des Handelsreisenden, 175  
 Probleme  
     *NP*-harte, 159  
     *NP*-schwere, 159  
     *NP*-vollständige, 159  
     unlösbare, 115  
     polynomial deterministisch entscheidbare, 155  
     polynomial nichtdeterministisch entscheidbare, 155  
     unentscheidbare, 115  
 Programm, 20  
     deterministisches, 22  
     nichtdeterministisches, 138  
 Projektion, mengentheoretische, 24  
 Projektionsfunktion, 38  
 Prädikate, *siehe* Relationen  
 Prädikatszeichen, 127  
 Quantoren, 127  
 Quelle einer (partiellen) Funktion, 23  
 Rechenschrittfunktion, 22  
 Rechenzeitfunktion  
     innere, 23  
     äußere, 25  
 Redex eines  $\lambda$ -Terms, 94  
 Reduktion, 92  
 Reduktionsrelation, 92  
 Regelsystem, 119  
 Registermaschine, 29  
 reine Disjunktion, 161  
 reine Konjunktion, 161  
 rekursiv aufzählbare Menge, 71  
 rekursive Trennbarkeit, 106  
 relationale Prozedur, 32  
 Relationen, 44  
      $\Pi_n^-$ ,  $\Pi_n^0$ -, 75  
      $\Sigma_n^-$ ,  $\Sigma_n^0$ -, 75  
     arithmetische, 75  
     Abschlußeigenschaften von, 77  
     primitiv rekursive, 44  
     rekursiv aufzählbare, 71  
     rekursive, 70  
     semirekursive, 71  
 relative Rekursivität, 115  
 Resultatsfunktion  
     innere, 24  
     äußere, 25  
 Schlinge, 168  
 Semantik, 128  
 Semi-THUE-System, 119  
 Signatur einer Sprache, 127  
 Simulation von Maschinen, 33  
 simultane Rekursion, 51  
 Sprache, 118, 120  
     kontextfreie, 125  
     kontextsensitive, 124  
     reguläre, 125  
 Sprungmarke, 20  
 Startkonfiguration, 22  
 Startmarke, 20  
 Startsymbol, 120  
 Steuereinheit, 20, 21  
 Struktur, 128  
 Subgraphen, 170  
 Substitution in Prädikaten, 45  
 Substitution von Funktionen, 38  
 Suchoperator, 56  
 Summation, 43  
 Super-Exponentiation, 42  
 Syntax, 127  
 Sätze der Prädikatenlogik, 127, 129  
 Teilgraph, 170

## Register

---

- Terme  
boolesche, 160  
der Prädikatenlogik, 127  
des  $\lambda$ -Kalküls, 91  
äquivalente boolesche, 161  
terminales Alphabet, 120  
Testanweisungen, 20  
Testinstruktionen, 20  
Theorembeweiser, 135  
Theorie der formalen Sprachen,  
126  
Träger einer Struktur, 128  
Turingmaschine, 25  
Turingreduzibilität, 116  
turingäquivalent, 116  
Typenvariablen, 112
- Übergangsfunktion, 22  
ungerichtete Graphen, 166  
universell, *siehe* Funktion  
Unlösbarkeitsgrade, 115, 116  
unmittelbarer Vorgänger eines Kno-  
tens, 169  
unmittelbarer Nachfolger, 169  
Untergraph, 170  
Unvollständigkeit logischer Kalküle,  
132
- Variable, 127, 160  
Verkettung von Wörtern, 26  
Vierfarbenproblem, 176  
Vollständigkeitssatz der Prädika-  
tenlogik, 131  
Vorgängerfunktion, 43
- Wertverlauf einer Funktion, 51  
Wertverlaufsrekursion, 51, 52  
Wortproblem, 118  
    für kontextsensitive Sprachen,  
    125  
Wurzel eines Baumes, 169  
Wörter, 26, 118

**Namen**

- ACKERMANN, W., 13, 82  
AL-KHWARIZMI, 11  
APPEL, K., 176  
BAKER, T., 156  
BOOLE, G., 159  
CHURCH, A., 13, 90, 135  
CURRY, H. B., 90  
DAVIS, M., 12  
FISCHER, M. J., 164  
FRIEDBERG, R., 116  
GILL, J., 156  
GIRARD, J.-Y., 113  
GÖDEL, K., 12, 13, 108, 111, 132  
HEESCH, H., 176  
HERBRAND, J., 13  
HILBERT, D., 13  
HOWARD, W., 111  
KLEENE, S. C., 13, 14  
KOCH, J., 176  
LEIBNIZ, G.W., 11  
ULLUS, R., 11  
LÖWENHEIM, L., 12  
MUCHNIK, A. A., 116  
PÉTER, R., 13  
POST, E., 14, 116  
PRESBURGER, M., 13  
RABIN, M. O., 164  
ROSSER, J. B., 13  
RUSSEL, B., 12  
SKOLEM, T., 13  
SOLOVAY, R., 156  
TAIT, W., 98  
TARSKI, A., 13  
TURING, A. M., 14, 25, 135  
V. NEUMANN, J., 15, 16  
WHITEHEAD, A.N., 12





