CORE
Metadata, citation and similar papers at core.ac.uk
Provided by Elektronisch archivierte Theorie - Sammelpunkt

Rekursivität

Diplomarbeit zur Erlangung des Magistergrades der Philosophie an der Grund- und Integrativwissenschaftlichen Fakultät der Universität Wien

eingereicht von
Horst Tellioğlu

Wien, am 22. November 1997

4. Dezember 1998 10:27:51 rev.1.2 korrigiert

Vielen Menschen ist ein Baum in den Kopf gepflanzt, aber das Gehirn selbst ist eher ein Kraut oder Gras als ein Baum.

Deleuze/Guattari

Inhaltsverzeichnis

Inhaltsverzeichnis ii
Inhaltsverzeichnis ii

0 Einleitung 1

1 Selbstbezüglichkeit 2
1.1 Paradoxa 3
1.1.1 Der Lügner 3
1.1.2 Klassifikation der Paradoxa nach W. V. Quine 4
1.2 Antinomien 5
1.2.1 Grellings Paradoxon 5
1.2.2 Die Antinomie von Russell 5
1.2.3 Das unmögliche Oberon-Programm 8

2 Rekursive und induktive Definitionen 10
2.1 Die mengentheoretische Sprache 12
2.2 Axiome 13
2.3 Definitionen 16
2.3.1 Relationen 16
2.3.2 Funktionen 19
2.3.3 Zahlen 21
2.3.4 n-Tupel und Strukturen 23
2.4 Induktionssatz 24
2.5 Definition durch Rekursion 25

3 Berechenbarkeit 29
3.1 Algorithmen (Effektive Prozeduren) 29
3.2 Rekursive Funktionen 32
3.2.1 Primitiv rekursive Funktionen 33
3.2.2 Die Ackermann-Funktion 43
3.2.3 μ-Rekursivität 49
3.2.4 Einschub: Induktive und rekursive Definitionen 57
3.3 Turingmaschinen 59
3.3.1 Turing-Berechenbarkeit 59
3.3.2 Äquivalenz von μ-Rekursivität und Turing-Berechenbarkeit 67
3.4 Registermaschinen 69
3.4.1 Registermaschinen-Berechenbarkeit 69
3.4.2 Das Selbstanwendungsproblem 74

4 Rekursive Algorithmen und Datenstrukturen **76**
4.1 Sich selbst aufrufen 76
4.2 Sich selbst enthalten 77

Symbolverzeichnis **84**

Stichwortverzeichnis **86**

Literatur **89**

0 Einleitung

Rekursivität, Reflexivität und Selbstreferentialität sind Begriffe, die in diesem Jahrhundert eine große Bedeutung erlangt haben. Sie prägen und verbinden Metamathematik, Logik und Informatik. Die Rekursionstheorie, als Theorie der Berechenbarkeit und Unberechenbarkeit, liefert wichtige Beiträge zur Frage, was prinzipiell algorithmisch erfassbar ist. Die Berechenbarkeit ist eine der wesentlichsten begrifflichen Neuschöpfungen mit direkten Konsequenzen für unser Verstehen des menschlichen Geistes. Ich habe versucht, mich dem Begriff Rekursivität vor allem formal zu nähern, um eine leichtere Überprüfbarkeit der Überlegungen zu ermöglichen.

Im Kapitel 1 behandle ich Selbstbezüglichkeit, Paradoxien und Antinomien. Dies scheint mir eine Voraussetzung zu sein, um im weiteren Verlauf Rekursivität davon abgrenzen zu können. Ich führe in diesem Zusammenhang ein algorithmisch nicht lösbares Problem vor. Eine erste Darstellung von Rekursivität erfolgt im Kapitel „Rekursive und induktive Definitionen“. Mit einer axiomatisierten Mengenlehre werden Mittel erarbeitet, um Strukturen definieren zu können, in denen induktive und rekursive Definitionen möglich sind.

Anschließend erläutere ich im vierten Kapitel 3 den Begriff der Berechenbarkeit, wie er durch rekursive Funktionen erfasst wird. Ich zeige die Beschränkungen der primitiven Rekursivität sowie die Äquivalenz von μ-Rekursivität zu Turing-Berechenbarkeit und Registermaschinen-Berechenbarkeit. Dies ist als Argument für die Church-Turing-These zu verstehen, die intuitive Berechenbarkeit mit Berechenbarkeit überhaupt gleichsetzt. Die μ-Rekursivität ist außerdem interessant, da, wie ich zeige, auf die Operation der primitiven Rekursion verzichtet werden kann, wenn der μ-Operator zur Verfügung steht. Registermaschinen sind ein abstraktes Modell, mit dem die heute üblichen Computer mit von-Neumann-Architektur untersucht werden können. Rekursivität als Berechenbarkeit und Selbstreferentialität treffen sich im Selbstanwendungsproblem, das eine wichtige Grenze der Leistungsfähigkeit von Registermaschinen und damit von Menschen liefert. Im letzten Kapitel bringe ich eine kleine Kritik einiger bereits als klassisch zu bezeichnenden Algorithmen und Datenstrukturen, die in der Informatik als rekursiv betrachtet werden.

1 Selbstbezüglichkeit

Rekursivität wird oft in Zusammenhang mit Paradoxien gebracht. Das verbindende Konzept scheint dabei die Selbstbezüglichkeit zu sein, da es viele Paradoxien gibt, die auf Selbstbezüglichkeit beruhen und auf den ersten Blick auch alle Rekursionen selbstreferentiell zu sein scheinen. Die gilt vor allem für rekursive Definitionen, in denen ein Ausdruck auf den ersten Blick durch sich selbst erklärt zu werden scheint. Oft wird Rekursion durch einen Satz wie den folgenden erklärt: Rekursion ist die „Definition eines Problems oder eines Verfahrens durch sich selbst“ (Dudenredaktion 1994). In der Informatik werden Prozeduren als rekursiv bezeichnet, wenn diese sich direkt oder indirekt selbst aufrufen. Analog dazu wurde versucht die Deklaration von rekursiven Datenstrukturen zu ermöglichen. „Werte eines *rekursiven* Datentyps könnten Komponenten desselben rekursiven Datentyps enthalten in Analogie zu einer Prozedur, die sich selbst aufruft. Wie Prozeduren könnten solche Typen-Definitionen direkt oder indirekt rekursiv sein.“ (Wirth 1983, p.190) Ein Objekt wird allgemein als rekursiv bezeichnet, „wenn es sich selbst als Teil enthält oder mithilfe von sich selbst definiert ist.“ (Wirth 1983, p.149) Es ist nun schwer vorstellbar, wie ein Objekt sich selbst als Teil enthalten könnte, da sich ein solches Objekt sicherlich hinsichtlich des Enthaltenseins in zwei unterscheiden würde. Auch die Aussage, dass etwas durch sich selbst oder mithilfe von sich selbst definiert wird, scheint kaum verständlich. So eine Definition würde entweder zu einem infiniten Regress oder direkt in ein Paradoxon führen. Rekursivität und Paradoxa verbindet zumindest ein besonderes Verhältnis zum Wörtchen „selbst“. Da auch in der axiomatisierten Mengenlehre, in der später die allgemeinste Form des Rekursionstheorems formuliert werden soll, Paradoxien eine entscheidende Rolle spielen, ist es notwendig, das Verhältnis von Rekursivität und Paradoxie genauer zu untersuchen. Dabei sollen auch die zitierten ungenauen Ausdrucksweisen aufgeklärt und von den eigentlichen Paradoxa unterschieden werden.

Bei Paradoxa, die auf Selbstbezüglichkeit beruhen, scheint der Referent ständig dem Denken zu entgleiten. Die dadurch ausgelöste Krise des Denkens kann bis in den Gödelschen Unvollständigkeitssatz verfolgt werden. Dieses Entgleiten des Referenten muss bei rekursiven Definitionen natürlich vermieden werden. Eine Frage ist also, ob rekursive Definitionen etwas selbstreferentielles oder paradoxes an sich haben und wie ihr Referent aussieht. Um dies zu klären, sollen im folgenden zuerst Paradoxien genauer betrachtet werden.

1.1 Paradoxa

1.1.1 Der Lügner

„Paulus, der Knecht Gottes und Apostel Jesu Christi“, hat folgenden Text geschrieben:

> Denn es gibt viele Ungehorsame, Schwätzer und Schwindler, besonders unter denen, die aus dem Judentum kommen. Diese Menschen muß man zum Schweigen bringen, denn aus übler Gewinnsucht zerstören sie ganze Familien mit ihren falschen Lehren. Einer von ihnen hat als ihr eigener Prophet gesagt: Alle Kreter sind Lügner und faule Bäuche, gefährliche Tiere. Das ist ein wahres Wort[1].

Der Antisemitismus und Religionismus des Paulus, der ihn wohl daran hinderte, das wesentliche an dieser Geschichte zu erkennen, soll hier nicht thematisiert werden. Uns interessiert hier, dass ein Kreter angeblich von sich selbst behauptet, ein Lügner zu sein. Nach mehreren Quellen hat dieser Kreter Epimenides geheißen und am Anfang des 6. Jahrhunderts vor unserer Zeitrechnung gelebt. Das logische Problem, das Epimenides durch seinen Ausspruch aufwarf, wird auch die *Paradoxie des Lügners* genannt. Sie lautet bei Paulus im Original (Diels und Kranz 1985, I.32):

Κρῆτες ἀεί ψεῦσται, κακὰ θηρία, γαστέρες ἀργαί.

Wörtlich übersetzt lautet dies:

Kreter immer Lügner, böse Tiere, faule Bäuche.

Wenn Epimenides dies als Kreter sagt, so ist er ein Lügner, falls er die Wahrheit spricht. Er lügt also, indem er die Wahrheit sagt. Falls er aber lügt, so trifft es nicht zu, dass die Kreter immer lügen, und er sagt die Wahrheit. Er sagt also die Wahrheit, indem er lügt.

Natürlich fällt bei dieser Formulierung auf, dass es nicht dasselbe ist, ein Lügner zu sein und immer zu lügen. Dies kann dann auch zur Auflösung des Paradoxons verwendet werden. Vielleicht hat Epimenides ja gerade beim Ausspruch dieses Satzes die Wahrheit gesagt. Unter der Verneinung des Satzes „Alle Kreter lügen“ wird außerdem normalerweise der Satz „Nicht alle Kreter lügen“ – also „Ein Kreter lügt nicht“ – verstanden, was ebenfalls das Paradoxon abschwächt. Wie dem auch sei, es gibt mehrere Möglichkeiten dieses ψευδόμενον schärfer zu formulieren und damit die gewünschte semantische Antinomie zu erzeugen.

Wird vom Kontext der Äußerung abgesehen, so lässt sich die Paradoxie jederzeit durch Aussprechen des folgenden Satzes reproduzieren:

[1] Der Rest des Abschnittes lautet: „Darum weise sie streng zurecht, damit ihr Glaube wieder gesund wird und sie sich nicht mehr an jüdische Fabeleien halten und an Gebote von Menschen, die sich von der Wahrheit abwenden. Für die Reinen ist alles rein; für die Unreinen und Ungläubigen aber ist nichts rein, sogar ihr Denken und ihr Gewissen sind unrein. Sie beteuern, Gott zu kennen, durch ihr Tun aber verleugnen sie ihn; es sind abscheuliche und unbelehrbare Menschen, die zu nichts Gutem taugen.“ (Bibel 1989, Tit 1.10-13)

Ich lüge jetzt.

Noch prägnanter ist folgende, angeblich von Eubulides aus dem 4. Jahrhundert vor der römischen Zeitrechnung stammende Formulierung:

$$\text{Diese Aussage ist falsch.} \tag{1.1}$$

Da die Authentizität der Epimenides zugeschriebenen Formulierung nicht endgültig geklärt ist, und auch nicht klar ist, ob Epimenides selbst seinen Ausspruch als Paradoxon verstand, könnte es sich bei dem Satz von Eubulides um die älteste bekannte Formulierung des Lügnerparadoxons handeln.

Wenn wir versuchen dieses Paradoxon zu verstehen, ohne gleich, wie Philites von Kos, am Nachdenken darüber zu sterben, so zeigt sich als erstes, dass es sich um eine *selbstreferentielle Aussage* handelt. Selbstreferenz muss jedoch nicht schlimm sein. Solche Aussagen spannen einen weiten Bogen, der von Tautologien bis zu Antinomien reicht. Also Aussagen ohne wesentliche Aussagekraft ebenso umfasst, wie solche, deren Informationsgehalt sie oszillieren lässt. Eine paradoxe jedoch nicht antinomische Form von Selbstreferenz findet sich – wenn wir von den Interpretationsschwierigkeiten absehen, die schon der Name Jahwe bereitet – im Satz „Ich bin der ich bin" (Bibel 1989, Exodus, III, 14), die eine Aussage der ersten Art darstellt. Eine Aussage der zweiten Art stellt dagegen die Paradoxie des Lügners in der letzten Formulierung dar. In ihr können wir das Oszillieren der Wahrheitswerte gut beobachten, wenn wir mit der Annahme beginnen, dass die darin enthaltene Aussage wahr ist. Wir müssen dann schließen, dass sie falsch ist. Nehmen wir hingegen an, dass sie falsch ist, so folgt daraus, dass sie wahr ist. Der Wahrheitsgehalt springt zwischen diesen binären Polen der Wahrheit und Falschheit hin und her.

1.1.2 Klassifikation der Paradoxa nach W. V. Quine

W. V. Quine versteht unter einer Paradoxie eine Konklusion, die auf den ersten Blick absurd erscheint, die sich aber argumentativ stützen lässt und auf diese Weise auflöst. Und zwar so, dass die darin ausgesprochene Behauptung entweder wahr ist (*veridical paradoxes*), falsch ist (*falsidical paradoxes*) oder keines von beiden (Quine 1962). Das heißt, als Paradoxa werden nicht nur scheinbar falsche Aussagen aufgefasst, die sich bei genauerer Betrachtung als wahr herausstellen, sondern auch wahre, die sich als falsch entpuppen. Als Beispiel für ein veridikales Paradoxon sei das des Barbiers angeführt, der alle Männer eines Dorfes rasiert, die sich nicht selbst rasieren. Der sich also genau dann selbst rasiert, wenn er es nicht tut und umgekehrt. Dieses Paradoxon ist veridikal, da es als einfache *reductio ad absurdum* aufgefasst werden kann. Es sagt nur aus, dass es so einen Barbier eben nicht gibt (Quine 1962; Kleene 1952). Es ist seltsam, aber wahr, dass es so eine Person nicht gibt. Einige von Zenons Paradoxa sind hingegen falsidikale. Nehmen wir als Beispiel Achilles und die Schildkröte. Hier ist es sowohl seltsam als auch falsch, zu sagen, dass Achilles niemals von der Schildkröte eingeholt wird. Diejenige Klasse von Paradoxien, die nur solche Aussagen enthält, die weder wahr noch falsch oder – was hier dasselbe bedeutet – auch beides zugleich sind, enthält die eigentlichen *Antinomien*.

1.2 Antinomien

1.2.1 Grellings Paradoxon

Prädikate haben die Eigenschaft, dass sie genau dann auf einen Gegenstand zutreffen, wenn es wahr ist, dass der Gegenstand die im Prädikat formulierte Eigenschaft besitzt. So trifft zum Beispiel „ist rot“ genau dann auf einen Gegenstand zu, wenn der Gegenstand rot ist. Oder der Satz „Es regnet“ ist genau dann wahr, wenn es tatsächlich regnet. Der Referent ist in den Beispielen einmal ein Gegenstand und einmal ein Ereignis.

Kurt Grelling hat 1908 folgendes Paradoxon formuliert: Wir wollen Adjektive autologisch nennen, wenn sie wahr von sich selbst – also selbstbeschreibend oder auf sich selbst zutreffend – sind. Wir wollen sie heterologisch nennen, wenn sie nicht wahr von sich selbst – also nicht selbstbeschreibend – sind. Wenn wir uns nun fragen, ob „heterologisch“ heterologisch ist, erhalten wir eine Antinomie. Der Selbstwiderspruch lässt sich nicht auflösen. Wir können in diesem Paradoxon heterologisch auch als „nicht wahr von sich selbst“ auffassen, und werden so zur wesentlichen Eigenart dieser Form von Paradoxon geführt. Denn „nicht wahr von sich selbst“ soll ja genau dann auf ein Ding zutreffen, wenn es nicht wahr von sich selbst ist. Wählen wir aber als dieses Ding genau die Phrase „nicht wahr von sich selbst“, so erhalten wir, dass „nicht wahr von sich selbst“ genau dann auf sich selbst zutrifft, wenn es nicht zutrifft. Dieselbe Art von Antinomie stellt auch der Lügner in seiner schärfsten Formulierung dar. „Dieser Satz ist falsch“ bezieht sich auf sich selbst und sagt etwas über seinen Wahrheitsgehalt aus.

In beiden Antinomien stellt sich die Frage, worauf eigentlich referiert wird, was der Referent der Aussage ist, von dem ausgesagt wird, dass er nicht wahr sei. Im Lügner ist die Frage also, worauf das Wort „dieser“ referiert. Denn „dieser Satz“ referiert ja auf den Satz „dieser Satz ist falsch“. Und zu sagen „‚Dieser Satz ist falsch‘ ist falsch“ erzeugt keine Antinomie, da der äußere Satz nicht auf sich selbst rekurriert.

W. V. Quine hat jedoch eine Methode gefunden, den Referenten eindeutig festzulegen und trotzdem eine Antinomie zu erzeugen. Dies geschieht, wenn wir den Satz so formulieren (Quine 1963, p.255):

‘yields a falsehood when appended to its own quotation’
yields a falsehood when appended to its own quotation

Dies ist wohl die beste Formulierung der Antinomie von Epimenides.

1.2.2 Die Antinomie von Russell

Eine weitere Antinomie, die nicht auf dem Konzept der Wahrheit beruht, wurde von Betrand Russell in einem Brief an Gottlob Frege formuliert und hatte katastrophale Auswirkungen auf Freges Arbeit. Außerdem macht es gewisse Vorsichtsmaßnahmen, bei der Formulierung dessen, was in einer axiomatisierten Theorie als Menge zugelassen werden soll, notwendig.

Freges fünftes Gesetz in seinem Buch „Grundgesetze der Arithmetik“ besagt, dass wir zu beliebigen Gegenständen einen Begriff bilden können, unter

den die Gegenstände dann als Umfang des Begriffes fallen. Anders formuliert sagt dieses *Komprehensionsprinzip*, dass wir beliebige Dinge, über eine Eigenschaft, die sie gemeinsam besitzen, zu einer Menge zusammenfassen können: $\bigvee_y \bigwedge_z (z \in y \leftrightarrow \varphi(z))$[2].

Dies entspricht Georg Cantors naiver Mengenvorstellung, die von ihm so beschrieben wurde:

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

Zu jeder Eigenschaft E können wir also in der üblichen Schreibweise die Menge

$$M = \{x \mid x \text{ hat die Eigenschaft } E\}$$

bilden. Das dieser Mengenvorstellung zu Grunde liegende Komprehensionsprinzip erlaubt uns jedoch auf ausgesprochen einfache Weise Mengen zu bilden, die nicht unserer naiven Mengenvorstellung entsprechen. Unter den so bildbaren Mengen befindet sich die Allmenge sowie die Russelsche Menge. Die *Allmenge*

$$A = \{x \mid x \text{ ist Menge}\}$$

enthält alle Mengen und damit sich selbst als Element. Die *Russellsche Menge*

$$R = \{x \mid x \text{ ist Menge und } x \notin x\}$$

führt auf eine Antinomie, wenn wir uns fragen, ob sich R selbst als Element enthält.

Nehmen wir an, R enthalte sich selbst, also $R \in R$. Diese Annahme besagt, dass R ein Element der Menge R ist. Das heißt, ein Element der Menge, die alle Mengen enthält, die sich *nicht* selbst enthalten. R ist also eine Menge, die sich nicht selbst als Element enthält, $R \notin R$. Dies steht im Widerspruch zur Annahme, führt jedoch noch nicht zu einer Antinomie, da es sich ja um eine reductio ad absurdum handeln könnte, die uns nur sagt, dass es so eine Menge R nicht geben kann – oder besser gesagt, dass wir eine Menge mit einer solchen Eigenschaft nicht denken wollen. Wir wollen also den Widerspruch vermeiden, indem wir die scheinbar falsche Annahme $R \in R$ fallen lassen und schließen, dass $R \notin R$. Das Ergebnis $R \notin R$, das wir ohne weitere Annahmen bewiesen haben, sagt uns, dass R kein Element der Menge ist, die alle Mengen enthält, die sich nicht selbst enthalten. Das heißt, R ist keine Menge, die sich nicht selbst enthält, also ist R – wenn wir die doppelte Negation auflösen – eine Menge, die sich selbst enthält: $R \in R$.

Da wir nun aber sowohl $R \notin R$ als auch $R \in R$ bewiesen haben, wurden wir auf eine Antinomie geführt, die nach ihrem Entdecker *Russellsche Antinomie*

[2] Vergleiche das Axiom Schema of Comprehension in Levy (1979) und das Axiom Schema of Abstraction in Suppes (1960). Die Kleinbuchstaben x und y stehen dabei für Mengen; $\phi(x)$ steht für einen mengentheoretischen Ausdruck, in dem y nicht frei vorkommt. Es handelt sich dabei um ein Axiomenschema, da ϕ variabel ist.

genannt wird. Das oben beschriebene veridikale Paradoxon des Bartscherers stellt eine popularisierte Version dieser Antinomie dar, die sich jedoch – wie wir gesehen haben – auflösen lässt. Die von Russells Antinomie ausgehende Beunruhigung ist so stark, da sie eine Beschränkung des Komprehensionsprinzip verlangt und damit eine fundamentale Annahme unseres Denkens erschüttert. Wir können plötzlich nicht mehr beliebige Dinge, über eine Eigenschaft, die sie gemeinsam besitzen, zu einer Menge zusammenfassen. Mit Frege können wir sagen, dass die so zusammengefassten Dinge den Umfang eines Begriffes bilden. Sie fallen unter einen bestimmten Begriff, wobei der Begriff eben die jeweilige Komprehension festlegt. So können wir zum Beispiel die Menge aller roten Dinge bilden, oder den Begriff Rot, unter den alle roten Dinge fallen. Jede Einschränkung des Komprehensionsprinzips scheint also auch Auswirkungen darauf zu haben, welche Begriffe wir bilden können, also die Bedingungen jeder Begriffsbildung zu bestimmen.

In der Mengenlehre gibt es zwei wesentliche Ansätze um mit diesem Problem umzugehen. Es können einerseits zu große Mengen verboten werden, wie im radikalen Ansatz des Intuitionismus, der nur konstruierbare Objekte zulässt, oder es werden zu großen Mengen nicht die gleichen Privilegien zugestanden wie kleinen (Monk 1969, p.14 und Kleene 1952, §13).

Der zweite Ansatz kann über eine Typentheorie, wie von Russell und Whitehead in ihren „Principia Mathematica“ vorgestellt, erfolgen. In einer solchen Typentheorie werden antinomische Selbstreferenzen verhindert, indem alle (mathematischen) Objekte bestimmten Stufen einer Objekthierarchie zugeordnet werden. Auf den untersten Stufen befinden sich Objekte, die keine Mengen sind, und Urelemente genannt werden können. Erst Objekte der ersten Stufe können Mengen sein und damit Objekte der nullten Stufe als Elemente enthalten. Dieses Schema wird beim Aufsteigen in der Hierarchie beibehalten. Damit kann die Russellsche Antinomie verhindert werden. Wird dieses Konzept der Hierarchisierung durch eine Typentheorie auf Sprachen übertragen, so wird damit die fundamentale Unterscheidung in *Objektsprachen* und *Metasprachen* eingeführt. Bestimmte Aussagen über Eigenschaften der Sprache dürfen nur in einer Metasprache erfolgen, deren Gegenstand dann als Objektsprache betrachtet wird. In den oben beschriebenen Paradoxa von Grelling und Epimenides führt diese Auflösung zur Unterscheidung von Wahrheit$_0$ bis Wahrheit$_n$. Typisieren wir auf diese Weise das Paradoxon von Epimenides in der Formulierung von Quine, so lautet es folgendermaßen:

‘yields a falsehood$_0$ when appended to its own quotation’
yields a falsehood$_1$ when appended to its own quotation

Das Paradoxon ist verschwunden, denn dieser Satz sagt nur, dass „‘yields a falsehood$_0$ when appended to its own quotation’ yields a falsehood$_0$ when appended to its own quotation“ falsch$_1$ ist, und das trifft nicht zu. Dieser Satz ist einfach sinnlos und die Reformulierung des Epimenidesparadoxons ist falsch$_2$. Das ist etwas umständlich, scheint aber zu funktionieren.

Ein weiterer Weg im gemäßigten zweiten Ansatz, in dem unerwünschte Mengenbildungen nicht absolut verboten werden, führt zu einer Klassenmengenlehre, in der zu große Mengen *eigentliche Klassen* genannt werden und dann der

Name Menge für die uneigentlichen Klassen reserviert wird. In der Klassenmengenlehre gibt es also eine *Allklasse* aber keine Allmenge. Aus der Russellschen Menge wird die *Russellsche Klasse*:

$$K_R = \{x \mid x \text{ ist Menge und } x \notin x\}$$

Da die Annahme, dass sich K_R selbst enthält, zu einem Widerspruch führen würde, brauchen wir nur zu schließen, dass K_R keine Menge ist. K_R ist also eine eigentliche Klasse.

Ein dritter Weg, der von Ernst Zermelo aufgezeigt wurde, besteht darin, das Komprehensionsprinzip zu einem Aussonderungsprinzip abzuschwächen, in dem Komprehensionen nur von Elementen von bereits bestehenden Mengen gebildet werden können. Das heißt, zu jeder Eigenschaft E und jeder Menge M_1 existiert dann die Menge

$$M_2 = \{x \mid x \in M_1 \text{ und } x \text{ hat die Eigenschaft } E\}$$

Da dabei die Existenz der Menge M_1 vorausgesetzt wird, ist die Formulierung der Russellschen Antinomie nicht wie im System von Frege möglich. Wenn wir nach dem Aussonderungsprinzip die Russellsche Menge

$$R = \{x \in M_1 \mid x \notin x\}$$

zu bilden versuchen, so werden wir vor die Wahl $R \notin M_1$ oder $R \in R$ gestellt und es fällt uns leicht ersteres zu wählen um die Antinomie zu vermeiden.

Die durch Russells Antinomie ausgelöste Krise im Denken ist jedoch, trotz der aufgezeigten Möglichkeiten damit umzugehen, noch nicht überstanden. Es ist zwar nicht mehr möglich die Russellsche Antinomie unmittelbar abzuleiten, das bedeutet jedoch nicht, dass dieser Widerspruch nicht auf einem anderen Wege doch noch gewonnen werden kann (Ebbinghaus 1994, p.13). Außerdem unterscheiden sich die vorgeschlagenen Wege darin, für welche Elementbeziehungen sie Klassen zulassen, und es gilt, eine optimale und konsistente Kombination von Existenzannahmen zu finden. Jede solche Einschränkung wirkt jedoch „unnatürlich“ gegenüber dem uneingeschränkten Komprehensionsprinzip (Quine 1962).

1.2.3 Das unmögliche Oberon-Programm

In Analogie zum Barbier, den es laut Quine nicht geben kann, will ich ein kleines Programm vorführen, das nicht geschrieben werden kann. Die Syntax entspricht der Sprache *Oberon*, die einer Familie von Programmiersprachen angehört, deren Entwicklung mit Algol 60 begann (Reiser und Wirth 1992; Naur 1963). Es kann über einen indirekten Beweis gezeigt werden, dass es unmöglich ist, das zum Definitionsmodul *Unschreibbar* gehörige Modul zu implementieren.

```
DEFINITION Unschreibbar ;
    PROCEDURE Haelt( prg, inp: ARRAY OF CHAR ) : BOOLEAN ;
END Unschreibbar .
```

Die Prozedur *Haelt* liefert *TRUE*, falls der Parameter *prg* ein syntaktisch korrektes Oberon-Programm enthält, dass für die Eingabe *inp* terminiert. Im Programm *prg* darf dabei als einzige Eingabeanweisung *In.String* genau einmal vorkommen. Die Prozedur *String* ist im Standardmodul *In* enthalten und liest eine Zeichenkette ein. Falls *prg* für *inp* nicht terminiert, oder mehrere Eingabeanweisungen vorkommen, gibt *Haelt* den Wert *FALSE* zurück. Nehmen wir für den Beweis an, dass *Unschreibbar* geschrieben werden könnte. Dann könnten wir mit seiner Hilfe das Modul *Entscheide* implementieren.

```
MODULE Entscheide ;
IMPORT Unschreibbar, In ;
VAR str: ARRAY 1024 OF CHAR ;

PROCEDURE Entscheide* ;
BEGIN
    In.Open ;
    In.String( str ) ;
    IF Unschreibbar.Haelt( str, str ) THEN
        WHILE TRUE DO
        END
    END
END Entscheide ;
END Entscheide .
```

Entscheide importiert die Module *In* und *Unschreibbar*. Es exportiert die Prozedur *Entscheide*. Rufen wir *Entscheide.Entscheide* auf, so wird mit *In.String* eine Zeichenkette in die Variable *str* eingelesen. Diese Zeichenkette wird an *Unschreibbar.Haelt* sowohl als Programmtext, als auch als dessen Eingabe übergeben. Das heisst, in *Entscheide* überprüft *Haelt*, ob *str* ein Programm ist, dass für seinen eigenen Text als Eingabe terminiert. Falls *Haelt* den Wert *TRUE* liefert, gerät *Entscheide* in die Endlosschleife *WHILE TRUE DO END*. Liefert *Haelt* den Wert *FALSE*, so wird *Entscheide* beendet.

Was passiert nun, wenn wir dem Programm *Entscheide* den eigenen Programmtext, also die Zeichenkette „MODULE Entscheide ; IMPORT Unschreibbar, In ; VAR str: ARRAY 1024 OF CHAR ; PROCEDURE Entscheide* ; BEGIN In.Open ; In.String(str) ; IF Unschreibbar.Haelt(str, str) THEN WHILE TRUE DO END END END Entscheide ; END Entscheide .“, übergeben? Terminiert *Entscheide* oder läuft es endlos? (1) Falls *Entscheide* terminiert, muss *Haelt* den Wert *FALSE* geliefert haben. Da die Eingabe nur eine Anweisung *In.String* enthält und syntaktisch korrekt ist, kann dies nur daran liegen, dass *Entscheide*, dessen Programmtext wir ja übergeben haben, nicht terminiert. (2) Falls *Entscheide* jedoch nicht terminiert, so muss *Haelt* den Wert *TRUE* zurückgegeben haben. Das bedeutet jedoch, dass *Entscheide* für den eigenen Programmtext terminiert.

Es ergibt sich also, dass das Programm *Entscheide* für den eigenen Programmtext als Eingabe genau dann terminiert, wenn es nicht terminiert. Ein offensichtlicher Widerspruch, der nur den Schluss zulässt, dass *Unschreibbar* nicht geschrieben werden kann.

2 Rekursive und induktive Definitionen

Kurt Gödel hat in seinem 1931 erschienenen Aufsatz „*Über formal unentscheidbare Sätze der* Principia mathematica *und verwandter Systeme I*“ eine Klasse von Funktionen definiert, die später als die *Klasse der primitiv rekursiven Funktionen* bezeichnet werden sollte. In Gödels berühmten Aufsatz spielt also nicht nur Selbstreferenz, sondern auch Rekursivität eine wichtige Rolle. Gödel schreibt (Gödel 1986, p.156–8):

> Eine zahlentheoretische Funktion $\phi(x_1, x_2, \ldots, x_n)$ heißt *rekursiv definiert aus* den zahlentheoretischen Funktionen $\psi(x_1, x_2, \ldots, x_{n-1})$ und $\mu(x_1, x_2, \ldots, x_{n+1})$, wenn für alle $x_2, \ldots, x_n, k$ folgendes gilt:
>
> $$\begin{aligned} \phi(0, x_2, \ldots, x_n) &= \psi(x_2, \ldots, x_n), \\ \phi(k+1, x_2, \ldots, x_n) &= \mu(k, \phi(k, x_2, \ldots, x_n), x_2, \ldots, x_n). \end{aligned}$$

In diesem Auszug aus Gödels Text ist natürlich noch nicht die Klasse der primitiv rekursiven Funktionen beschrieben, sondern nur, was es heißt, als Funktion rekursiv definiert zu sein. Diese besondere Form von Definition, die auch oft Definition durch Induktion[1] genannt wird – obwohl das eher eine Beweis- als eine Definitionsmethode ist – spielt nicht nur in der Rekursionstheorie eine wesentliche Rolle, und es lohnt sich genauer zu betrachten, was es heißt, rekursiv zu definieren, und welche Voraussetzungen erfüllt sein müssen, damit dies funktioniert.

Dabei will ich versuchen, genau zwischen einer induktiven Definition (*inductive definition*) und einer rekursiven Definition (*definition by induction*, *recursive definition*) zu unterschieden. Typische Beispiele für induktive Definitionen, die nur voraussetzen, dass unter dem Begriff „Nachfolger“ etwas vorgestellt werden kann, sind folgende:

Definition der natürlichen Zahlen: 1. 0 ist eine natürliche Zahl. 2. Wenn n eine natürliche Zahl ist, dann ist der Nachfolger von n eine natürliche Zahl. 3. Nichts sonst ist eine natürliche Zahl.

Definition einer Klasse P von Zeichenfolgen: 1. Die Zeichenfolge () ist in P. 2a. Wenn die Zeichenfolge E in P ist, dann auch die Zeichenfolge (E). 2b. Wenn die Zeichenfolgen E und F in P sind, dann auch die Zeichenfolgen EF. 3. Nichts sonst ist in P.

Dieser generative Aufbau von Objektklassen ausgehend von einem ersten Objekt – in den Beispielen die Zahl 0 oder die Zeichenfolge „()“ – mit Hilfe einer induktiven Definition hat ein Analogon in der rekursiven Definition einer Funktion ϕ. Dabei wird der Wert der Funktion für bestimmte erste Argumente,

[1] *definition by induction*: cf. Devlin (1993, p.51) und Kleene (1952, p.217)

also meist für die Zahl 0 festgelegt, und dann für eine beliebige Zahl y der Wert von $\phi(y')$ (y' bezeichnet dabei den Nachfolger von y) mit Hilfe von y und $\phi(y)$ ausgedrückt. Daraus folgt dann, dass $\phi(y)$ für jede natürliche Zahl y definiert ist. Das dies tatsächlich gefolgert werden kann, soll als ein Ergebnis aus den Betrachtungen in diesem Kapitel hervorgehen. Als Beispiel sei hier die Definition einer Funktion $\phi(y)$ angegeben, die leicht als eine vereinfachte Version aus dem obenstehenden Zitat von K. Gödel mit $n = 1$ zu erkennen ist:

$$\begin{aligned} \phi(0) &= q \\ \phi(y') &= \chi(y, \phi(y)) \end{aligned}$$

Die Funktion $\phi(y)$ wird durch Induktion über y definiert. Wobei q eine beliebige Konstante und χ eine zahlentheoretische Funktion mit zwei Argumenten ist. Als Beispiel sei die Funktion $pot(x, y)$, die die Potenzierung der Basis x mit den Exponenten y, also x^y berechnet, rekursiv definiert. Die Funktion pot wird dabei mit Hilfe der Funktion $mul(x, y)$, die das Produkt ihrer Argumente x und y liefert, und der Bildung der nachfolgenden Zahl y' von y definiert. Ich gebe die Definition in zwei unterschiedlichen Schreibweisen wieder, um den Übergang zu bekannteren Notationsweisen deutlich zu machen:

$$\begin{array}{lcl|lcl} pot(x,0) & = & 1 & x^0 & = & 1 \\ pot(x,y') & = & mul(x, pot(x,y)) & x^{y+1} & = & x \cdot x^y \end{array}$$

Auf der rechten Seite ist die Potenzierung und Multiplikation in üblicher Schreibweise dargestellt. Die Bildung des Nachfolgers wird dabei durch Addition von 1 notiert.[2] In der rekursiven Definition von $pot(x, y)$ spielt die Multiplikation als Rekursionsvorschrift dieselbe Rolle wie die Funktion χ in der Definition von $\phi(y)$.

Wann ist es nun möglich, eine Funktion auf diese oder ähnliche Weise rekursiv zu definieren? Ich werde in diesem Kapitel eine Version des Rekursionstheorems darstellen, die „aussagt“, dass dies in genauer zu bestimmenden axiomatisierten mengentheoretischen Systemen möglich ist. Diese Systeme sind unter anderem deshalb für uns interessant, da es relativ einfach ist, die notwendigen Grundlagen zu erarbeiten, um sie zu beschreiben und das Rekursionstheorem in ihnen in seiner allgemeinsten Gestalt zu formulieren. Außerdem spielt die Mengenlehre neben der Logik in der mathematischen Grundlagenforschung eine wichtige Rolle. Viele mathematische Begriffsbildungen lassen sich auf mengentheoretische zurückführen, wobei eine interessante Frage ist, ob dies von ontologischer Relevanz ist. Wird zum Beispiel etwas über den ontologischen Status der Zahlen ausgesagt, wenn es möglich ist, diese mengentheoretisch zu beschreiben? Ich werde jedoch von solchen ontologischen Fragestellungen absehen und rein axiomatisch-deduktiv vorgehen. Das heißt auch, dass wir zuerst einmal nicht wissen wollen, was wir uns unter Mengen vorzustellen haben, wir aber dann schrittweise immer mehr von ihren Eigenschaften angeben werden, bis wir unter anderem sicher sind, dass es sich nicht um Körbe voller Äpfel oder Birnen handeln kann.

[2] Dies kann bei der rekursiven Definition der Addition etwas verwirrend aussehen, wenn wie üblich $x + 0 = x, x + (y + 1) = (x + y) + 1$ statt $x + 0 = x, x + y' = (x + y)'$ oder $add(x, 0) = 0, add(x, y') = add(x, y)'$ geschrieben wird.

2.1 Die mengentheoretische Sprache

Naiv können Mengen nach dem bereits erwähnten Komprehensionsprinzip gebildet werden, was jedoch zu den beschriebenen Antinomien führt. Für den axiomatischen Aufbau dagegen ist eine einfache formale Sprache vorteilhaft. Wir gehen dabei davon aus, dass alle Dinge als Mengen aufgefasst werden können – wobei dies nicht als ontologische Behauptung missverstanden werden soll –, dass sich Mengen als Elemente enthalten können und dass Mengen, die die gleichen Elemente enthalten, als dieselbe Menge betrachtet werden sollen. Letzteres drückt den *extensionalen Standpunkt* aus, der sagt, dass Mengen nur durch ihre Elemente bestimmt werden. Es handelt sich also zum Beispiel, für $a = 2$, bei $\{2, 3\}$, $\{3, a\}$ und $\{2, 3, 2\}$ um verschiedene Notationen für die gleiche Menge. Die Gleichheitsbeziehung zwischen zwei Mengen wird mit dem Symbol $=$, die Elementbeziehung mit dem Symbol $\in$ ausgedrückt. Als Variablen für Mengen verwende ich hauptsächlich eventuell indizierte Kleinbuchstaben $(x, y, z, z_0, z_1, \dots)$, als metasprachliche Variablen für Ausdrücke der mengentheoretischen Sprache griechische Kleinbuchstaben $(\phi, \psi, \dots)$. Weiters verwende ich die üblichen Junktoren Negation $\neg$, Disjunktion $\vee$, Konjunktion $\wedge$, Implikation $\rightarrow$ und Äquijunktion $\leftrightarrow$, sowie die Quantoren Existenz $\bigvee_x$ und Generalisierung $\bigwedge_x$. Das heißt, es handelt sich um die übliche Sprache der Prädikatenlogik erster Stufe mit Identität, nur erweitert um die Elementbeziehung. Neben den atomaren Ausdrücken der Form $Variable = Variable$ und $Variable \in Variable$, können zusammengesetzte Ausdrücke mit Hilfe der Junktoren in der Form $(\phi\ \textit{Junktor}\ \psi)$ gebildet werden. Die Klammern gehören zum Alphabet der Sprache. Eine Ausnahme bildet die Negation, da sie einstellig ist. Mit ihr kann ein zusammengesetzter Ausdruck der Form $\neg\phi$ gebildet werden. Die Quantoren ermöglichen es, aus einem Ausdruck ϕ die Ausdrücke $\bigwedge_x \phi$ und $\bigvee_x \phi$ zu bilden. Dieser informelle Abriss sollte ausreichen. Eine genauere Darstellung ist zum Beispiel im Buch von Ebbinghaus et al. (1996) zu finden.

Um diesen Formalismus handhabbar zu machen, ist es notwendig, Abkürzungen einzuführen. Die so als Abkürzungen eingeführten weiteren Symbole müssen jedoch immer auf einen Ausdruck zurückgeführt werden können, der nur nach den obigen ersten Regeln gebildet wurde, und auf diese Weise wieder eliminiert werden können. Zum Beispiel kann der Ausdruck $\bigwedge_z (z \in x \rightarrow z \in y)$, der die Teilmengenbeziehung zwischen den Mengen x und y beschreibt, durch das *Prädikatsymbol* $x \subseteq y$ abgekürzt werden. Eine solche durch einen mengentheoretischen Ausdruck beschriebene Beziehung wird *Prädikat* genannt. Bei der Teilmengenbeziehung und der Elementbeziehung handelt es sich um zweistellige Prädikate. Jedes n-stellige Prädikate kann durch einen Ausdruck der Gestalt $\phi(x_1, \dots, x_n)$ definiert werden. *Operationen* werden auf dem Universum definierte Abbildungen genannt, die durch ein Prädikat beschrieben werden können. Wenn wir zum Beispiel mit dem Prädikat $\phi_T(x, y)$ die Teilmengenbeziehung bezeichnen, so können wir mit $\phi_P(x, y) := \bigwedge_z (z \in y \leftrightarrow \phi_T(z, x))$ die Potenzmengenabbildung beschreiben, die jeder Menge x des Universums, die Menge y ihrer Teilmengen z zuordnet.

2.2 Axiome

Das Existenzaxiom (Ex): *Es gibt eine Menge.*

$$\bigvee_x (x = x)$$

Das Extensionalitätsaxiom (Ext) (Extensionality axiom)**:** *Umfangsgleiche Mengen sind gleich.*

$$\bigwedge_x \bigwedge_y (\bigwedge_z (z \in x \leftrightarrow z \in y) \rightarrow x = y)$$

Das Schema der Aussonderungsaxiome (Aus) (Axiom schema of separation)**:** *Zu allen x gibt es ein y, das genau diejenigen Elemente z von x enthält, für die $\varphi(z)$ gilt.*

$$\bigwedge_x \bigvee_y \bigwedge_z (z \in y \leftrightarrow z \in x \wedge \varphi(z))$$

Da das Schema der Aussonderungsaxiome angibt, wie für jeden mengentheoretischen Ausdruck der Gestalt $\varphi(z, x_1, \ldots, x_n)$ ein Axiom gebildet werden kann, lassen sich mithilfe von **Aus** unendlich viele Axiome erzeugen. Es unterscheidet sich wesentlich vom Komprehensionsprinzip $\bigvee_y \bigwedge_z (z \in y \leftrightarrow \varphi(z))$, da die Existenz der Menge x als Bedingung eingeht. Das ist gemeint, wenn von einer Abschwächung des Komprehensionsprinzips zum Aussonderungsaxiom gesprochen wird.

Wenn wir zum Beispiel wissen, dass die Menge aller Tiere existiert, dann können wir davon die Menge alle Tiere aussondern, die Menschen sind. Wenn wir das nicht wissen, könnte es sich bei der Menge aller Menschen um eine antinomisch gebildete Menge handeln. Die Existenz der Menge, für deren Element $\varphi(x)$ gilt, muss also immer erst *bewiesen* werden.

Die nach dem Aussonderungsprinzip möglichen Mengenbildungen schließen Mengen aus, die auf die Russellsche Antinomie führen würden. So kann zwar noch die Menge $y := \{z \mid z \in x \wedge z \notin z\}$ gebildet werden. Wenn wir aber annehmen, dass $y \in y$, so folgt daraus, dass $y \notin y$. Nehmen wir das Gegenteil an, so ist ebenfalls $y \notin y$. Also ist auf jeden Fall $y \notin y$. Dies bedeutet, dass wir uns entscheiden müssen, ob wir sagen wollen, dass y kein Element von x ist, oder dass y ein Element von y ist. Da wir die Russellsche Antinomie vermeiden wollen, fällt uns die Wahl leicht, und wir sagen einfach, dass $y \notin x$ ist. Damit haben wir uns allerdings auch entschieden, die Existenz einer Allmenge abzulehnen, denn $\{z \mid z \in x \wedge z \notin z\}$ ist kein Element von x, welches x wir auch wählen. *Es gibt keine Allmenge.* Das kann so formuliert werden:

$$\bigwedge_x \{z \mid z \in x \wedge z \notin z\} \notin x$$

Als Trost dafür liefert uns das Aussonderungsprinzip die Existenz der *leeren Menge*:

Definition 2.1 $\emptyset := \{z \mid z \in x \wedge z \neq z\}$

Eine solche Definition können wir, wenn wir im Gedächtnis behalten, dass der Existenzbeweis immer erst erbracht werden muss, zu $\emptyset := \{z \mid z \neq z\}$ abkürzen. Da nach **Aus** die leere Menge existiert, dürfen wir die abgekürzte Schreibweise verwenden. Die leere Menge kann statt mit $\emptyset$ auch mit $\{\}$ bezeichnet werden.

Mir scheint der Vorschlag Mengen nicht als Gefäße, sondern als *Inhalte* aufzufassen, überlegenswert (Boolos und Jeffrey 1988). Die leere Menge ist dann nicht als leeres Gefäß zu denken, sondern vielmehr als ein Name für „kein Inhalt". So ergibt sich intuitiv, dass es nur eine leere Menge gibt, während es ja viele leere Gefäße gibt, die sich sehr wohl unterscheiden. Als welcher Inhalt ist allerdings dann $\{\emptyset\}$ zu denken? Als „kein Inhalt als Inhalt von etwas"? Jedenfalls scheint das Existensionalitätsaxiom so etwas von seiner Willkürlichkeit zu verlieren, denn einem Buch, das alle Seiten doppelt enthält, würden wir kaum mehr Inhalt zuschreiben als seiner üblicheren Variante, in der jede Seite nur einmal vorkommt.

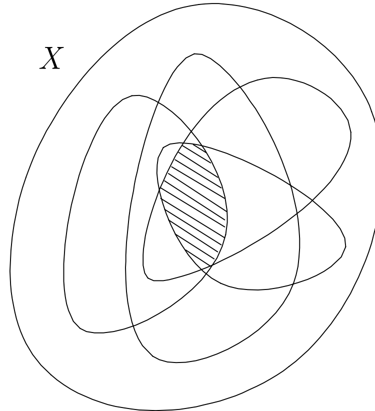
Es ergibt sich ebenfalls, dass die *Durchschnittsmenge* zweier Mengen existiert:

Definition 2.2 $x \cap y := \{z \mid y \in x \wedge z \in y\}$

Wichtig ist der Begriff des Durchschnittes aller Elemente einer Menge X, oder kurz des *Durchschnittes einer Menge* X:

Definition 2.3 $\bigcap X := \{z \mid X \neq \emptyset \wedge \bigwedge_x (x \in X \to z \in x)\}$

Falls $X = \emptyset$ legen wir fest, dass auch $\bigcap X := \emptyset$[3]. Die Menge $\{z \mid \bigwedge_x (x \in X \to z \in x)\}$ existiert nach **Aus**, da alle z, die der Bedingung $\bigwedge_x (x \in X \to z \in x)$ genügen, Elemente eines $y \in X$ sind. $\bigcap X$ enthält alle Elemente, die Element eines Elementes von X sind. Eine Zeichnung soll dies verdeutlichen:



Die Menge X enthält vier Mengen, deren gemeinsame Elemente durch die schraffierte Fläche symbolisiert sind. Die Elemente, die sich in der schraffierten

[3] In einer von Neumann-Bernays-Gödelschen Klassenmengenlehre wird $\bigcap$ so definiert, dass $\bigcap \emptyset = V$. Wobei $V = \{x \mid x = x\}$ *Universum* genannt wird. In einer Klassenmengenlehre wird unterschieden zwischen Klassen, denen das Elementsein erlaubt ist und die Mengen genannt werden, und solchen die zu groß dafür sind – die *eigentliche Klassen.* V ist eine solche eigentliche Klasse aller Mengen.

Fläche befinden, bilden den Durchschnitt von X.

Die *mengentheoretische Differenz* ist definiert als: $x \setminus y := \{z \in x \mid z \notin y\}$

Das Kleine Vereinigungsmengenaxiom ($\cup$-Ax): *Zu je zwei Mengen x und y gibt es eine Menge, die alle Elemente von x und y enthält.*

$$\bigwedge_x \bigwedge_y \bigvee_w \bigwedge_z (z \in x \wedge z \in y \rightarrow z \in w)$$

Das Große Vereinigungsmengenaxiom ($\bigcup$-Ax) (Union axiom)**:** *Zu jeder Menge X gibt es eine Menge, die alle Elemente der Elemente von X enthält.*

$$\bigwedge_X \bigvee_y \bigwedge_{x,z} (x \in X \wedge z \in x \rightarrow z \in y)$$

Potenzmengenaxiom (Pot) (Power-set axiom)**:** *Zu jeder Menge x gibt es eine Menge, die alle Teilmengen von x enthält.*

$$\bigwedge_x \bigvee_y \bigwedge_z (z \subseteq x \rightarrow z \in y)$$

Ebenso wie für den Durchschnitt einer Menge lässt sich hier eine unäre Operation definieren. Die *Potenzmenge* einer Menge: $\mathsf{Pot}(x) := \{z \mid z \subseteq x\}$. Nach **Aus** existiert die Potenzmenge jeder Menge. Die Potenzmenge von $\{\}$ ist $\{\{\}\}$, da die leere Menge Teilmenge jeder Menge ist.

Obwohl die Begriffe Abbildung und Mächtigkeit noch nicht vorgestellt wurden, nehme ich hier eine Bekanntschaft mit ihnen an, und bringe an dieser Stelle eine fundamentale Erkenntnis der Mengenlehre, die besagt, dass es zu jeder Menge eine andere mit mehr Elementen gibt:

Cantors Theorem: *Es gibt keine Abbildung von einer Menge x auf $\mathsf{Pot}(x)$.*
Das kann gleichbedeutend auch so formuliert werden:
Die Mächtigkeit einer Menge x ist kleiner als die Mächtigkeit der $\mathsf{Pot}(x)$.

Der Beweis des Theorems erfolgt mit einer Variante des Gedankens, der im Beweis von Russells Antinomie angewandt wird. Und tatsächlich gelangte Russell zu seinem Beweis durch eine Analyse des Beweises von Cantor (Ebbinghaus 1994, p.133, Levy 1979, p.87, Monk 1969, p.56).
Beweis Nehmen wir an, dass eine Funktion f die Menge z auf $\mathsf{Pot}(z)$ abbildet. Basteln wir nun die Menge $y = \{x \mid x \in z, x \notin f(x)\}$, die eine Teilmenge von z ist, $y \subseteq z$. Da jede Teilmenge von z Element von $\mathsf{Pot}(z)$ ist, muss es ein x geben, dessen Bild unter f die Menge y ist. Wir wählen gezielt dieses $x \in z$ mit $f(x) = y$. Dann gilt: $x \in y \rightarrow x \notin f(x) \rightarrow x \notin y$ und $x \notin y \rightarrow x \notin f(x) \rightarrow x \in y$.

Unendlichkeitsaxiom (Inf) (Infinity Axiom)**:**

$$\bigvee_x (\emptyset \in x \wedge \bigwedge_z (z \in x \rightarrow z \cup \{z\} \in x))$$

Mit diesem Axiom kommen wir dem Thema der Arbeit, der Rekursivität, bereits etwas näher, da Mengen, die nach *Inf* gebildet werden, einen *induktiven* Aufbau besitzen. Eine nach *Inf* gebildete Menge können wir uns so vorstellen: $\{\emptyset, \{\emptyset\}, \{\emptyset, \{\emptyset\}\}, \{\emptyset, \{\emptyset\}, \{\emptyset, \{\emptyset\}\}\}, \dots\}$. *Inf* garantiert die Existenz unendlicher Mengen.

Definition 2.4 (Induktive Mengen)

$$x \textit{ ist } \text{induktiv} \; :\leftrightarrow \emptyset \in x \wedge \bigwedge_z (z \in x \rightarrow z \cup \{z\} \in x\}$$

Besonders interessant ist die *kleinste* induktive Menge, die nur aus den Mengen besteht, die Element jeder induktiven Menge ist.

Definition 2.5 $\omega := \bigcap\{x \mid \emptyset \in x \wedge \bigwedge_z (z \in x \rightarrow z \cup \{z\} \in x)\}$

ω ist Teilmenge jeder induktiven Menge, also die kleinste Menge im Sinne von $\subseteq$. Nach **Ext** ist die so definierte Menge ω eindeutig bestimmt.

Schema der Ersetzungsaxiome (Ers) (Axiom-Schema of Replacement)**:**

$$\bigwedge_u \bigwedge_v \bigwedge_w (\psi(u,v) \wedge \psi(u,w) \rightarrow v = w) \rightarrow \bigwedge_z \bigvee_y \bigwedge_v (v \in y \leftrightarrow (\bigvee_u \in z)\psi(u,v))$$

Dabei dürfen y, z und w nicht frei in $\psi(u,v)$ vorkommen und $\psi(u,w)$ bezeichnet die Formel, die entsteht, wenn v in $\psi(u,v)$ durch w ersetzt wird. Der erste Teil der Formel formuliert die Voraussetzung, dass ψ jedem u genau ein v zuordnet, ψ also eine Operation ist. Wenn das nun so ist, dann behauptet das Axiom, dass es eine Menge y gibt, die alle Mengen v enthält, die auf solche Weise eindeutig den Mengen u aus z zugeordnet werden. Es werden also über eine Operation ψ die Elemente der Menge z durch Elemente der Menge y ersetzt. Einfach ausgedrückt sagt **Ers**, dass, falls f eine Funktion und $\mathsf{Def}(f)$ eine Menge ist, auch $\mathsf{Bild}(f)$ eine Menge ist. Allerdings werden Def, Bild und Funktionen erst später erklärt.

Das ganze funktioniert auch etwas komplizierter mit Parametern, als die wir von u und v verschiedene freie Variablen $x_1, \ldots, x_n$ in $\psi(u, v, x_1, \ldots, x_n)$ bezeichnen können. Das v ist nicht nur durch u, sondern auch durch $x_1, \ldots, x_n$ bestimmt. Sind $x_1, \ldots, x_n$ gegeben, so besteht die Menge y aus denjenigen Mengen v, für die $\bigvee_{u \in z} \psi(u, v, x_1, \ldots, x_n)$ gilt. Eine Variable kommt *frei* in einem Ausdruck vor, wenn sie nicht im Wirkungsbereich eines Quantors liegt.

Fundierungsaxiom (Fund) (Regularity axiom, Axiom of foundation)**:**
Jede nicht leere Menge besitzt ein bezüglich der Elementbeziehung minimales Element.

$$\bigwedge_y (y \neq \emptyset \rightarrow \bigvee_x (x \in y \wedge x \cap y = \emptyset))$$

Fund garantiert, dass das Mengenuniversum bezüglich $\in$ hierarchisch aufgebaut ist. Es gibt keine zirkulären Elementbeziehungen und auch keine Menge, die Element eines ihrer Elemente sein könnte:

$$\neg \bigvee_{x_0, \ldots @!, x_n} (x_0 \in x_1 \wedge \ldots \wedge x_{n-1} \in x_n \wedge x_n \in x_0)$$

2.3 Definitionen

2.3.1 Relationen

Die einzigen Dinge, die sich in unserem Universum befinden sollen, sind Mengen und bisher trifft das ja auch zu. Es ist jedoch wesentlich, dass wir auch

Beziehungen zwischen Dingen darstellen können. Die Weise, in der zwei Dinge zueinander in Beziehung stehen, kann durch ein *geordnetes Paar* dargestellt werden. So könnte zum Beispiel das geordnete Paar $\langle x, y\rangle$ interpretiert werden als Größen- oder Verwandtschaftsverhältnis. Um geordnete Paare unter die gewöhnlichen Dinge unseres Universums subsumieren zu können, beschränken wir uns darauf, sie als beliebige Objekte zu betrachten, für die wir (auf dem Mengenuniversum) eine binäre Paarungsoperation P definieren können, für die folgendes gilt:

$$\bigwedge_{x,y,u,v} (P(x,y) = P(u,v) \rightarrow x = u \wedge y = v) \tag{2.1}$$

Wir können dann $\langle x, y\rangle$ so definieren: $\langle x, y\rangle := P(x, y)$. Wir *eliminieren* also Relationen durch die Weise, in der wir geordnete Paare explizieren. Laut Quine ist sogar jede Explikation eine Elimination (Quine 1960, p.260). Zu beachten ist, dass es viele Operationen und damit Objekte des Universums gibt, die den Anforderungen von 2.1 genügen. Nach einem Vorschlag von Norbert Wiener können wir $\langle x, y\rangle$ als die Menge $\{\{x\}, \{y, \emptyset\}\}$ auffassen. Nach einem bekannteren Vorschlag von Kazimierz Kuratowski ist es ebenso möglich, $\langle x, y\rangle$ mit $\{\{x\}, \{x, y\}\}$ zu identifizieren, was wir als Definition übernehmen wollen.

Definition 2.6 (geordnetes Paar) $\langle x, y\rangle := \{\{x\}, \{x, y\}\}$.

Aus der Auffassung des geordneten Paares als Relation zwischen zwei Objekten ist also die Eigenschaft eines Objektes geworden, die als einstelliges Prädikat formuliert werden kann:

Definition 2.7 *x ist geordnetes Paar* $:\leftrightarrow \bigvee_u \bigvee_v x = \langle u, v\rangle$.

Die Frage, was ein geordnetes Paar ist, ist zwar nicht beantwortet worden, der Begriff des geordneten Paares ist aber in einer klärenden Weise durch andere Konzepte ersetzt worden. Wir haben das Problem nicht gelöst, wir haben es als unwichtig markiert und aufgelöst, indem wir geordnete Paare als bestimmte Mengen auffassen. Quine beschreibt dies so: „In the beginning there was the notion of the ordered pair, defective and perplexing but serviceable. Then men found that whatever good had been accomplished by talking of an ordered pair $\langle x, y\rangle$ could be accomplished by talking instead of the class $\{\{x\}, \{y, \emptyset\}\}$ – or, for that matter, of $\{\{x\}, \{x, y\}\}$." (Quine 1960, p.260) Allerdings ergeben sich bei diesem Vorgang ein paar zusätzliche Eigenschaften, die wir vorher nicht mit geordneten Paaren assoziiert haben, wie die, dass x und y Elemente der Elemente von $\langle x, y\rangle$ sind, oder dass $\langle x, x\rangle = \{\{x\}\}$. Für geordnete Paare gilt also $\langle x, y\rangle = \langle u, v\rangle \rightarrow x = u \wedge y = v$ und es ist wissenswert, dass

$$u \in x \wedge v \in y \rightarrow \langle u, v\rangle \in \mathsf{Pot}(\mathsf{Pot}(x \cup y))$$

Mit Hilfe von geordneten Paaren lassen sich leicht das kartesische Produkt zweier Mengen, sowie Relationen definieren:

Definition 2.8 $x \times y := \{\langle u, v\rangle \mid u \in x \wedge v \in y\}$. *$x \times y$ heißt* kartesisches Produkt *von x und y.*

Definition 2.9 *x ist eine binäre* Relation $:\leftrightarrow$ *x ist eine Menge von geordneten Paaren.*

Definition 2.10 *x ist eine binäre* Relation über $y :\leftrightarrow x \subseteq y \times y$.

Besondere Relationen sind die *leere Relation* $\emptyset$, die *Allrelation* $x \times x$ über einer Menge x, die *Identitätsrelation* $id_x := \{\langle x, y\rangle \mid x = y\}$, sowie die Elementrelation über einer Menge x: $\in_x = \{\langle u, v\rangle \mid u, v \in x \wedge u \in v\}$. Für „$u$ steht in der binären Relation r zu v", also $\langle u, v\rangle \in r$, können wir auch in Infixnotation kurz urv schreiben.

Nachdem wir geordnete Paare mit Hilfe von Prädikaten über dem Mengenuniversum als Mengen erklärt haben, können wir also auch *Relationen* über Mengen selbst als Mengen auffassen. Dies ist möglich, da für alle Ausdrücke $\varphi(u, v, x_1, \ldots, x_n)$ gilt:

$$\bigwedge_{x_1,\ldots,x_n} \bigwedge_a \bigvee_r r = \{\langle u, v\rangle \mid u \in a \wedge v \in a \wedge \varphi(u, v, x_1, \ldots, x_n)\}$$

Wenn zum Beispiel φ das Gleichheitsprädikat bezeichnet, dann gilt für $u, v \in a$, dass $u = v \leftrightarrow \langle u, v\rangle \in id_a$. id_a ist eine Relation über der Menge a. Allerdings, und das ist hinsichtlich Überlegungen zum Identitätsbegriff vielleicht aufschlussreich, kann mit einer Relation nicht das ganze Gleichheitsprädikat eingefangen werden, da diese Relation eine Allmenge wäre, die es ja nach **Aus** nicht geben kann.

Definition 2.11

(i)	$\mathsf{Def}(x)$	$:=$	$\{u \mid \bigvee_v(\langle u, v\rangle \in x)\}$. $\mathsf{Def}(x)$ *ist das* Vorfeld *von x.*
(ii)	$\mathsf{Bild}(x)$	$:=$	$\{v \mid \bigvee_u(\langle u, v\rangle \in x)\}$. $\mathsf{Bild}(x)$ *ist das* Nachfeld *von x.*
(iii)	$\mathsf{Feld}(x)$	$:=$	$\mathsf{Def}(x) \cup \mathsf{Bild}(x)$.

Für „f ist eine Relation" schreiben wir kurz $Rel\ f$.

Definition 2.12 *Eine Relation r heißt*

(i)	irreflexiv	$:\leftrightarrow$	$\bigwedge_x(\langle x, x\rangle \notin r)$
(ii)	reflexiv	$:\leftrightarrow$	$\bigwedge_x(x \in a \to \langle x, x\rangle \in r)$
(iii)	transitiv	$:\leftrightarrow$	$\bigwedge_x\bigwedge_y\bigwedge_z(\langle x, y\rangle \in r \wedge \langle y, z\rangle \in r \to \langle x, z\rangle \in r)$
(iv)	konnex (linear)	$:\leftrightarrow$	$\bigwedge_x\bigwedge_y(\langle x, y\rangle \in r \vee \langle y, x\rangle \in r \vee x = y)$
(v)	symmetrisch	$:\leftrightarrow$	$\bigwedge_x\bigwedge_y(\langle x, y\rangle \in r \wedge \langle y, x\rangle \in r)$
(vi)	antisymmetrisch (identitiv)	$:\leftrightarrow$	$\bigwedge_x\bigwedge_y(\langle x, y\rangle \in r \wedge \langle y, x\rangle \in r \to x = y)$
(vii)	asymmetrisch	$:\leftrightarrow$	$\bigwedge_x\bigwedge_y(\langle x, y\rangle \in r \wedge \neg\langle y, x\rangle \in r)$

Ist eine Relation konnex, so sind je zwei Elemente miteinander *vergleichbar* und jedes Element ist mit sich selbst vergleichbar.

Definition 2.13 (Ordnungsrelation, Äquivalenzrelation)

(i) r *ist eine* Ordnungsrelation über a $:\leftrightarrow$ *Rel* r *und* r *ist reflexiv über* a*, antisymmetrisch, transitiv und konnex über* a*. Eine Ordnungsrelation über* a *ist eine Ordnungsrelation im Sinne von* $\leq$*.*

(ii) r *ist eine* Ordnungsrelation $:\leftrightarrow$ r *ist eine Ordnungsrelation über* $\mathsf{Feld}(r)$*.*

(iii) r *ist eine* strikte Ordnungsrelation über a $:\leftrightarrow$ *Rel* r *und* r *ist irreflexiv, transitiv und konnex über* a*. Eine strikte Ordnungsrelation über* a *ist eine Ordnungsrelation im Sinne von* $<$*.*

(iv) r *ist eine* strikte Ordnungsrelation $:\leftrightarrow$ r *ist eine strikte Ordnungsrelation über* $\mathsf{Feld}(r)$*.*

(v) r *ist eine* Äquivalenzrelation über a $:\leftrightarrow$ *Rel* r *und* r *ist reflexiv über* a*, symmetrisch und transitiv.*

(vi) r *ist eine* Äquivalenzrelation $:\leftrightarrow$ r *ist eine Äquivalenzrelation über* $\mathsf{Feld}(r)$*.*

Können je zwei Elemente miteinander verglichen werden, so wird die Ordnungsrelation *total* genannt, ansonsten *partiell.* Das heißt, die oben definierten Ordnungsrelationen sind totale, da die Konnexivität jeweils als Eigenschaft angegeben wurde. Im weiteren Verlauf sollen alle angeführten Ordnungsrelationen als totale verstanden werden.

Wir sagen, dass ein x ein r-minimales Element von a ist, wenn $x \in a$ und es kein $y \in a$ gibt, mit $\langle y, x\rangle \in r$. Falls r eine Ordnungsrelation über a ist, so ist x das kleinste Element von a bezüglich r.

Mengen wird mit Hilfe von auf ihnen definierten Relationen eine relationale Struktur aufgeprägt. Binäre Relationen erzeugen dabei eine binäre Struktur.

Definition 2.14 (binäre Struktur)
x *ist eine binäre Struktur* $:\leftrightarrow x = \langle a, r\rangle$*, mit* $r \subseteq a \times a$ *(vgl. Def. 2.10)*

Die Menge a heißt *Trägermenge* von x. Ist r eine Ordnungsrelation über a, so wird $\langle a, r\rangle$ (totale) Ordnung genannt.

2.3.2 Funktionen

Ein Begriff, der scheinbar gleichberechtigt neben dem Mengenbegriff steht, ist der Funktionsbegriff. Gottlob Frege verwehrte sich in seiner Analyse von Funktionen, diese über den Begriff einer Veränderlichen zu erklären, auf die ein Teil eines Funktionsausdrucks verwiese: „Einfacher und klarer wäre der Fall wohl so darzustellen: Jeder Zahl eines x-Bereiches ist eine Zahl zugeordnet. Die Gesamtheit dieser Zahlen nenne ich den y-Bereich.“ (Frege 1904) Anschließend erklärt er aber die Abgrenzung dieser Bereiche für unwesentlich und konzentriert sich auf den Aspekt der Zuordnung zweier Zahlen, die nach unterschiedlichen Gesetzen vor sich geht. Unterschiede in den Zuordnungsgesetzen drücken Unterschiede der Funktionen aus. Die Funktionen selber können jedoch nicht bezeichnet werden. Lediglich das Zeichen der Funktion kann als *ungesättigter* Teil eines Ausdrucks auftreten. So können wir die Ergänzungsbedürftigkeit der Sinusfunktion mit $\sin(\cdot)$ andeuten, indem wir das Zeichen des Arguments, das ja auf eine Veränderliche referieren würde, weglassen. Die Argumentstelle selbst

habe ich durch „·“ markiert. Frege markierte sie durch einen etwas größeren Abstand zwischen den Klammern: sin( ).

Wir wollen von Frege übernehmen, dass es auf eine Zuordnung ankommt, und dass Funktionen nichts mit Veränderlichen zu tun haben. Die Unterscheidung der Bereiche ist jedoch wesentlich. Auch kann auf Funktionen referiert werden, da sie als besondere Relationen und damit als Mengen aufgefasst werden können.

Das Wesentliche einer Funktionen ist also ihr Definitionsbereich, ihr Wertebereich sowie die Tatsache, dass sie eine Zuordnung erfasst. Dabei wird jedes Element des Definitionsbereichs mit genau einem Element des Wertebereichs in Relation gesetzt. Für „f ist eine Funktion“ schreiben wir kurz *Fkn* f und definieren:

Definition 2.15 (Funktion)
$\mathit{Fkn}\ f :\leftrightarrow \mathit{Rel}\ f \wedge \bigwedge_x \bigwedge_y \bigwedge_z (\langle x, y\rangle \in f \wedge \langle x, z\rangle \in f \rightarrow y = z).$

Dass jedem Element der Menge x ein Element der Menge y zugeordnet wird, symbolisieren wir durch $x \longrightarrow y$ und definieren dies so:

Definition 2.16 (Zuordnung) $f : x \longrightarrow y :\leftrightarrow \mathit{Fkn}\ f \wedge \mathsf{Def}(f) = x \wedge \mathsf{Bild}(f) \subseteq y.$

Während also Frege „$f(\ )$“ als Zeichen der Funktion deutete, wollen wir eine zweistellige Operation ( ) einführen, die als Argumente eine Funktion f, sowie eine beliebige zweite Menge x hat, und diesen den Funktionswert von x unter f zuordnet. Die Argumentstellen der Operation ( ) können wir wieder durch „·“ andeuten: $\cdot(\cdot)$. Die intuitivere Schreibweise wäre wohl $\langle x, y\rangle \in f$ oder, wie für Relationen üblich, $x\ f\ y$.

Definition 2.17 *Funktionswert von x unter f:*

$$f(x) := \begin{cases} \text{das eindeutig bestimmte } y \text{ mit } \langle x, y\rangle \in f, \text{ falls } \mathit{Fkn}\ f \wedge x \in \mathsf{Def}(f); \\ \emptyset, \text{ sonst.} \end{cases}$$

Dabei ist zu beachten, dass die Verwendungsweise des Zeichens „$f(x)$“ mehrdeutig ist. Es kann darunter die Funktion selbst verstanden werden, wie im Satz „$\mathsf{Add}(x, y)$ ist symmetrisch“. Es kann aber auch der Wert der Funktion gemeint sein, wie im Satz „Die Summe $\mathsf{Add}(x, y)$ von zwei natürlichen Zahlen x und y ist grösser oder gleich x“. Es sollte deshalb nur Add geschrieben werden, wenn die Funktion gemeint ist, und $\mathsf{Add}(x, y)$, wenn der Funktionswert bezeichnet werden soll. Die ist jedoch nicht immer einfach möglich, wie das Beispiel „$f(g(x))$“ zeigt. Dieses Zeichen steht wieder einerseits für den Wert der Funktion f, wenn als Argument der Wert der Funktion g für das Argument x verwendet wird, andererseits aber auch für eine neue Funktion, die durch die Komposition (siehe unten) von g und f entsteht.

Um einer Erklärung von Rekursion näher zu kommen, wollen wir uns erinnern, dass es eines ihrer wesentlichen Merkmale ist, dass bei der Beschreibung eines Einzeldings auf ein ähnliches aber einfacher strukturiertes Einzelding rekurriert wird. Dieses Merkmal hat ihr den Namen gegeben und unterscheidet sie

auch von der Induktion, die nicht reduktionistisch, sondern generisch operiert. Ob diese Unterscheidung tatsächlich gemacht werden kann, soll noch untersucht werden. Der Vorgang der Reduktion verlangt natürlich, dass bei bestimmten basalen Entitäten abgebrochen wird. Das Fundierungsaxiom (p.16) garantiert die Existenz einer solchen basalen Entität in Form eines minimalen Elements bezüglich der $\in$-Relation. Das Rekurrieren sowie der Begriff der Vereinfachung können durch die Komposition und Restriktion von Funktionen – die ja, laut Definition nur bestimmte Mengen sind – erfasst werden, wenn es sich bei den untersuchten Dingen um Mengen handelt.

Das Hintereinanderausführen von Funktionen, das als Operation auf dem Mengenuniversum definiert werden kann, nennen wir ihre Komposition:

Definition 2.18 (Komposition)

$$g{\circ}f := \begin{cases} \{(u, g(f(u))) \mid u \in \mathsf{Def}(f)\}, & \textit{falls Fkn } f \wedge \textit{ Fkn } g \wedge \mathsf{Bild}(f) \subseteq \mathsf{Def}(g); \\ \emptyset, \textit{ sonst.} & \end{cases}$$

Gilt *Fkn* f und *Fkn* g, sowie $\mathsf{Bild}(f) \subseteq \mathsf{Def}(g)$, so kann auf $\mathsf{Def}(f)$ eine Funktion mit $(g \circ f)(x) := g(f(x))$ für alle $x \in \mathsf{Def}(f)$ definiert werden. Falls $f : a \longrightarrow b$ und $g : b \longrightarrow c$, so ist $g \circ f : a \longrightarrow c$. Um Verwirrungen zu vermeiden, ist $g \circ f$ am besten „g nach f“ zu lesen, da die Ausführung von rechts nach links erfolgt. Ähnlich wie Prädikate durch eine Relation, können wir also Operationen durch Funktionen „einfangen“, wenn wir sie nur für eine bestimmte Menge betrachten (vergleiche Ebbinghaus 1994).

Definition 2.19 (Fortsetzung)
g heißt Fortsetzung *der Funktion f* $:\leftrightarrow$

$$\mathsf{Def}(f) \subseteq \mathsf{Def}(g) \wedge \bigwedge_x (x \in \mathsf{Def}(f) \rightarrow f(x) = g(x))$$

g ist genau dann eine Fortsetzung von f, wenn $f \subseteq g$. Falls g eine Fortsetzung der Funktion f ist, so nennen wir f auch eine *Restriktion* von g. $\mathsf{Def}(f)$ sei mit y bezeichnet. Anstelle von f schreiben wir dann $g{\restriction}y$.

Definition 2.20 (Restriktion) *Restriktion von g auf den Definitionsbereich y:*

$$g{\restriction}y := \{(x, g(x)) \mid x \in y\}$$

Gleichbedeutend kann $g{\restriction}y$ auch als $g \cap (y \times \mathsf{Bild}(g))$ definiert werden. Ist g eine Funktion, so auch $g{\restriction}y$. Ein Beispiel soll die Restriktion anschaulich machen. Für $g = \{\langle v, w\rangle, \langle x, y\rangle, \langle y, z\rangle\}$ und $y = \{x, y\}$ ist $g{\restriction}y = \{\langle x, y\rangle, \langle y, z\rangle\}$.

Zum Abschluss sei noch die Menge aller Funktionen f von x nach y definiert:

Definition 2.21 ${}^x y := \{f \in \mathsf{Pot}(x \times y) \mid f : x \longrightarrow y\}$

2.3.3 Zahlen

Zahlen dienen uns einerseits dazu, etwas zu zählen, das heißt, eine Reihenfolge anzugeben, andererseits dazu eine Anzahl festzustellen. Zahlen, die im ersten

Sinn gebraucht werden, nennen wir *Ordinalzahlen*. Diejenigen, die zur Feststellung einer Anzahl verwendet werden, nennen wir *Kardinalzahlen*. Welche Mengen sollen wir nun *als* die natürlichen Zahlen wählen? Wir können damit beginnen, dass wir die mengentheoretische Zahl **7** mit irgendeiner Menge x identifizieren, die 7 Elemente hat. **7** ist das Symbol einer Konstanten der mengentheoretischen Sprache, so wie das Konstantensymbol $\emptyset$ die Menge y bezeichnet, auf die das Prädikat φ, mit $\varphi(y) := \neg\bigvee_y(x \in y)$, zutrifft. 7 dagegen ist das metasprachliche Zeichen für die Zahl 7, wenn wir an die von uns unabhängige Existenz von Zahlen glauben, oder einfach für den Begriff, der durch das Wort Sieben bezeichnet wird. Die Menge **8** als Zahl 8 soll dann natürlich um ein Element mehr als **7** haben. Es liegt nahe, die Menge **7** selbst als achtes Element in **7** aufzunehmen, um acht Elemente zu erhalten. Die Menge **8** besteht also aus den sieben Elementen der Menge **7** vereinigt mit der Menge **7** selbst. Nach diesem Vorschlag ist jede Zahl also die Menge ihrer Vorgängerinnen. Formal definieren wir die *Nachfolgerin* einer Menge als:

Definition 2.22 (Nachfolgerin von x) $\mathsf{N}(x) := x \cup \{x\}$

Wollen wir mit **0** die Menge mit null Elementen bezeichnen, so bleibt uns dafür nur die Wahl von $\emptyset$. Die ersten drei Nachfolgerinnen von **0** sind dann
$\mathbf{1} := \mathsf{N}(\mathbf{0}) = \mathsf{N}(\emptyset) = \emptyset \cup \{\emptyset\} = \{\emptyset\} = \{\mathbf{0}\}$,
$\mathbf{2} := \mathsf{N}(\mathbf{1}) = \mathsf{N}(\{\emptyset\}) = \{\emptyset\} \cup \{\{\emptyset\}\} = \{\emptyset, \{\emptyset\}\} = \{\mathbf{0}, \mathbf{1}\}$ und
$\mathbf{3} := \mathsf{N}(\mathbf{2}) = \mathsf{N}(\{\emptyset, \{\emptyset\}\}) = \{\emptyset, \{\emptyset\}\} \cup \{\{\emptyset, \{\emptyset\}\}\} = \{\emptyset, \{\emptyset\}, \{\emptyset, \{\emptyset\}\}\} = \{\mathbf{0}, \mathbf{1}, \mathbf{2}\}$

Da uns dieser Aufbau bekannt vorkommt, wählen wir die kleinste induktive Menge ω als die Menge der natürlichen Zahlen. **Inf** garantiert uns, dass die Konstruktion der Nachfolgerin unendlich oft *in einer* Menge fortgesetzt werden kann. Das heißt, **Inf** garantiert, dass die von **0** ausgehende unendliche Reihe von Nachfolgerinnen zu *einer* Menge zusammengefasst werden kann. Ohne das Unendlichkeitsaxiom wäre also nicht gewährleistet, dass diese Menge, die einen Denkschritt von einer unendlichen Reihe zu deren aktual unendlichen Gesamtheit darstellt, existiert. Ob das alles einen Sinn ergibt, sei hier beseite gestellt.

Definieren wir die Projektion des linken und rechten Elements entsprechend als

Definition 2.23 (Projektion)
(i) $\pi_l(\langle x, y\rangle) := x$
(ii) $\pi_r(\langle x, y\rangle) := y$

so kann die Nachfolgerfunktion folgendermaßen definiert werden:[4]

Definition 2.24 (Nachfolgerin) $\mathsf{N} := \{z \in \omega \times \omega \mid \pi_r(z) = \pi_l(z) \cup \{\pi_l(z)\}\}$

Als Teilmenge jeder induktiven Menge, ist ω selber induktiv. Dies besagt gerade, dass erstens $\mathbf{0} \in \omega$ und zweitens $\bigwedge_n(n \in \omega \rightarrow \mathsf{N}(n) \in \omega)$.

[4]Zur Verdeutlichung: $\pi_l(\{\{u\}, \{u, v\}\}) = u$

2.3.4 n-Tupel und Strukturen

Das Konzept des geordneten Paares lässt sich über eine rekursive Definition leicht auf beliebige n-Tupel erweitern. Geordnete Paare sind so verstanden ein Spezialfall, nämlich 2-Tupel.

Definition 2.25 (n-Tupel)
(i) $\langle x \rangle := x$
(ii) *Für* $n \geq 1$ *sei* $\langle x_0, \ldots @!, x_n \rangle := \langle \langle x_0, \ldots @!, x_{n-1} \rangle, x_n \rangle$.

Eine *algebraische Struktur* ist – als eine ebenso einfache Verallgemeinerung der binären Struktur – das 4-Tupel $\langle a, (r_i)_{i \in I_1}, (f_i)_{i \in I_2}, (c_i)_{i \in I_3}, \rangle$. Die verwirrende Familienschreibweise (vergleiche Ebbinghaus 1994, p.66-67) soll uns hier nicht weiter interessieren. Ausgesagt werden soll damit, dass eine algebraische Struktur aus einer Trägermenge a, mehreren Relationen r_i, mehreren Funktionen f_i, sowie mehreren Konstanten c_i besteht. Die bereits auf Seite 19 erwähnte binäre Struktur besteht aus der Trägermenge und nur einer Relation. Die Funktionen und Konstanten fehlen – was erlaubt ist.

Für induktive Beweise sowie rekursive Definitionen über Mengen ist es wichtig, dass diese einen bestimmten hierarchischen Aufbau besitzen. So gilt zum Beispiel für ω, dass jede ihrer Teilmengen ein kleinstes Element enthält. Wir können allgemein solche Relationen als *fundiert* bezeichnen, die einer Menge a eine Struktur aufprägen, in der jede nicht leere Teilmenge von a ein kleinstes Element bezüglich der Relation r enthält.

Definition 2.26 (fundierte Relation) (well-founded relation)
r ist fundierte Relation über a $:\leftrightarrow$

$$\bigwedge_b (\emptyset \neq b \wedge b \subseteq a \rightarrow \bigvee_y (y \in b \wedge b \cap \{x \mid \langle x, y \rangle \in r\} = \emptyset))$$

Dazu ist folgende simple Abkürzung nützlich:

Definition 2.27 (Menge der r-Vorgänger)

$$r[u] := \{v \in \mathsf{Def}(r) \mid \langle v, u \rangle \in r\}$$

$r[u]$ bezeichnet also die Menge aller bezüglich r vor u liegenden Elemente aus $\mathsf{Def}(r)$.

Definition 2.28 (fundierte Struktur) (well-founded structure)
$\langle a, r \rangle$ *ist eine fundierte Struktur* $:\leftrightarrow$
$\langle a, r \rangle$ *ist eine binäre Struktur* $\wedge \bigwedge_b (\emptyset \neq b \wedge b \subseteq a \rightarrow \bigvee_y (y \in b \wedge b \cap r[u] = \emptyset))$

Nicht leere Teilmengen von a können mehr als ein minimales Element bezüglich der Relation r haben. Wir können den Begriff der fundierten Struktur zu dem der *Wohlordnung* verschärfen, indem wir fordern, dass es nur ein solches Element in jeder nicht leeren Teilmenge der Trägermenge geben soll. Für fundierte Strukturen gilt dies nämlich im allgemeinen nicht.

Definition 2.29 (Wohlordnungsrelation)
r ist eine Wohlordnungsrelation über a $:\leftrightarrow$
r ist eine strikte Ordnungsrelation und fundierte Relation über a.

Definition 2.30 (Wohlordnung)
$\langle a, r\rangle$ *ist eine Wohlordnung* $:\leftrightarrow$
$\langle a, r\rangle$ *ist eine strikte Ordnung und eine fundierte Struktur.*

Definition 2.31 (fundierte Menge)
x *ist fundiert* $:\leftrightarrow \langle \omega, \in_x\rangle$ *ist eine fundierte Struktur.*

Für die Durchführung des Beweises, dass rekursive Definitionen auf fundierten Strukturen möglich sind, wird uns folgende Definition nützlich sein:

Definition 2.32 (Anfangsstück)
b *ist ein Anfangsstück von* a *bezüglich* $r :\leftrightarrow$

$$r \subseteq a \times a \wedge b \subseteq a \wedge \bigwedge_u (u \in b \to r[u] \subseteq b)$$

Für das leichtere Verstehen ist zu beachten, dass die Elemente der Menge $r[u]$ aus $\mathsf{Def}(r)$ sind. Das heisst, auch wenn die u in der Definition aus b gewählt werden, müssen die Elemente von $r[u]$ nicht aus b sein. Ein Beispiel: $a = \{1, 2, 3, 4\}$, $b = \{1, 2\}$, $c = \emptyset$, $d = \{2, 3\}$. Die Mengen b und c sind Anfangsstücke von a bezüglich der $<$-Relation. Die Menge d nicht, da $r[3] = \{1, 2\} \not\subseteq d$, aber $3 \in d$.

2.4 Induktionssatz

Das Prinzip des Beweisens durch mathematische Induktion ist in zwei Formen bekannt. Bei der *vollständigen Induktion* (ordinary induction) wird gezeigt, dass alle natürlichen Zahlen eine bestimmte Eigenschaft haben, indem gezeigt wird, dass erstens die Zahl 1 eine bestimmte Eigenschaft hat und dass zweitens für jede natürliche Zahl k auch $k + 1$ die Eigenschaft hat. Bei der *gewöhnlichen Induktion* (weak induction, complete induction) wird gezeigt, dass alle natürlichen Zahlen eine bestimmte Eigenschaft haben, indem gezeigt wird, dass, falls alle natürlichen Zahlen kleiner als k die Eigenschaft haben, auch k die Eigenschaft hat.

Wollen wir also eine Aussage der Form $\bigwedge_n A(n)$, mit $n \in \mathbb{N}$, durch (metasprachliche) vollständige Induktion beweisen, so zeigen wir folgendes:

1. *Induktionsanfang*: Die Behauptung gilt für ein kleinstes Element $n = 0$.
2. *Induktionsschritt*: Wenn die Behauptung für ein beliebiges n gilt (*Induktionsvoraussetzung*), dann gilt sie auch für deren Nachfolgerin $n + 1$ (*Induktionsbehauptung*).

Wir können nun einen Satz beweisen, der es uns erlaubt, die Induktion *objektsprachlich* durchzuführen.

Theorem 2.1 (vollst. Ind. in fund. Strukturen – Mengenform)
Sei $\langle a, r\rangle$ *eine fundierte Struktur, dann gilt*

$$\bigwedge_b (\bigwedge_u (u \in a \wedge r[u] \subseteq b \to u \in b) \to a \subseteq b)$$

Im Gegensatz zum oben skizzierten metasprachlichen Induktionsverfahren wird hier nicht von einem Element auf dessen Nachfolger, sondern von allen r-Vorgängern auf das Element selbst geschlossen. Da dadurch Satz 2.1 vielleicht etwas schwieriger zu lesen ist, gebe ich eine Erklärung in Worten: Es wird etwas über Mengen b ausgesagt, die dadurch bestimmt werden, dass u in b enthalten ist, falls u Element von a ist und alle Vorgänger von u Elemente von b sind. Alle Vorgänger von u sind genau dann Elemente von b, wenn die Menge der r-Vorgänger von u – die ja alle Vorgänger von u bezüglich r zu einer Menge zusammenfasst – Teilmenge von b ist, $r[u] \subseteq b$. Für alle so bestimmten Mengen b gilt nun, dass $a \subseteq b$. Das heißt, die Mengen b enthalten mindestens alle Elemente von a. Der Induktionsanfang ergibt sich dadurch, dass die Menge der r-Vorgänger von u leer ist, $r[u] = \emptyset$, falls u ein r-minimales Element von a ist.
Beweis Wäre $a \setminus b \neq \emptyset$ und $u \in a \setminus b$ ein minimales Element bezüglich r, so gälte $r[u] \subseteq b$, und damit auch $u \in b$. Das ist ein Widerspruch. □

Etwas vertrauter wirkt der Induktionssatz als Schema in Prädikatsform:

Theorem 2.2 (vollst. Ind. in fund. Strukturen – Prädikatsform)
Sei $\langle a, r\rangle$ eine fundierte Struktur

$$\rightarrow \bigwedge_u (u \in a \wedge \bigwedge_v (\langle v, u\rangle \in r \rightarrow \phi(v)) \rightarrow \phi(u)) \rightarrow \bigwedge_u (u \in a \rightarrow \phi(u))$$

In Worten: Wenn für ein beliebiges u gilt, dass $\phi(u)$ gilt, falls $\phi(v)$ für alle v gilt, die in der Relation r zu u stehen, dann gilt $\phi(u)$ für alle u. Die Induktionsannahme lautet dabei $\bigwedge_v (\langle v, u\rangle \in r \rightarrow \phi(v))$. Für Strukturen, die nicht fundiert sind, gilt ein solches Induktionsprinzip nicht. Damit ist die Fundiertheit eine Voraussetzung für das im nächsten Abschnitt formulierte Rekursionstheorem.

2.5 Definition durch Rekursion

Das hier vorgestellte Rekursionstheorem sagen etwas über die *Definierbarkeit* von bestimmten Funktionen über bestimmten Strukturen mit Hilfe von Rekursion aus. Rekursive Definitionen scheinen sich wesentlich von *expliziten Definitionen* zu unterscheiden. Bei expliziten Definitionen wird ein Definiendum durch ein Definiens erklärt, das jederzeit wieder eliminiert werden kann. Das Definiendum ist also nur eine abgekürzte Schreibweise für das Definiens. In dieser normativen Auffassung des Definitionsbegriffes, ist „eine Definition eine willkürliche Festsetzung, durch die ein neues Zeichen, für einen zusammengesetzten Ausdruck eingeführt wird, dessen Sinn aus seiner Zusammensetzung bekannt ist. Das Zeichen, das bis dahin keinen Sinn hatte, bekommt durch die Definition den Sinn jenes zusammengesetzten Ausdrucks.“ (Frege 1914, p.104–106, vergleiche auch Whitehead und Russell 1910, p.21) Erinnern wir uns an die bereits erwähnte Teilmengenbeziehung: $x \subseteq y :\leftrightarrow \bigwedge_z (z \in x \rightarrow z \in y)$. Das Symbol $\subseteq$ kommt im Definiens nicht vor und das Definiendum kann in allen bisherigen Formeln durch das Definiens ersetzt werden. Das heißt, dass durch die Einführung von $\subseteq$ keine *wesentliche* Erweiterung der mengentheoretischen Sprache stattfindet, da das Symbol jederzeit wieder eliminiert werden kann. Es ist nach dieser Auffassung theoretisch unnötig, überhaupt Definitionen einzuführen. Bei der ebenfalls

bereits erwähnten rekursiven Definition der Addition ist diese Eliminierbarkeit nicht auf den ersten Blick ersichtlich: $m + 0 = m; m + \mathsf{N}(n) = \mathsf{N}(m + n)$. Im zweiten Teil der Definition erscheint das Definiendum + auch im Definiens. Handelt es sich hier um eine von der expliziten Definition völlig verschiedenen Definitionsweise? Wird die mengentheoretische Sprache und Theorie dadurch in einer wesentlichen Weise erweitert?

Die Rekursionstheoreme zeigen unter anderem, dass dem nicht so ist, und dass sich rekursive Definitionen auf gewöhnliche zurückführen lassen. Die Addition auf $\mathbb{N}$ zum Beispiel (vergleiche Monk 1969, p.87) kann durch eine explizite Definition eingeführt werden, wenn gezeigt werden kann, dass es genau eine Funktion $f : \omega \times \omega \longrightarrow \omega$ gibt, mit

$\bigwedge_m \bigwedge_n (m, n \in \omega \wedge f(m, 0) = m \wedge f(m, \mathsf{N}(n)) = \mathsf{N}(f(m, n)))$.

Dann kann die Addition auf folgende Weise definiert werden:

$$+ := \bigcap\{f : f \in^{\omega\times\omega} \omega \wedge \bigwedge_m \bigwedge_n (m, n \in \omega \rightarrow f(m, 0) = m \wedge f(m, \mathsf{N}(n) = \mathsf{N}(f(m, n))))\}$$

Das Definiens bezeichnet die Einermenge $\{f\}$. Es gibt *genau eine* solche Funktion f und + ist genau diese Funktion, $+ := \bigcap\{f\} = f$. Den etwas komplizierten Beweis, der zeigt, dass die Funktion f existiert und dass es genau ein solches f gibt, wollen wir uns nicht ersparen.

Im Beispiel wurde die Addition auf der Menge der natürlichen Zahlen ω definiert.[5] Die allgemeinste Form von rekursiven Definitionen ist jedoch nicht über ω, sondern über fundierten Strukturen möglich. Wir wollen also zeigen, dass auf einer fundierten Struktur a die Funktion G mit Hilfe einer *Rekursionsvorschrift* F definiert werden kann: $G(x) = F(x, G \restriction r[x])$. Die Rekursionsvorschrift F führt dabei den Wert der Funktion G für x mit Hilfe einer *Rekursion* auf Werte der Funktion G für Vorgänger von x zurück. Die Rekursion gibt dabei an, auf welche Elemente zu rekurrieren ist. In unserem Fall besteht die Rekursion in der Bestimmung der Menge der r-Vorgänger. Wir erinnern uns, dass $r[x]$ eine Abkürzung für $\{y \mid \langle y, x\rangle \in r\}$ ist. Im einfachsten Fall der *primitiven Rekursion*, wird dabei $G(x)$ nur auf die direkte Vorgängerin von x zurückgeführt, also nur auf die unmittelbar vorangehende Stelle rekurriert. Bei einer *Wertverlaufsrekursion* wird auf alle bezüglich der Relation r vor x liegenden Werte zurückgegriffen.

In einer fundierten Struktur $\langle a, r\rangle$ ist ein Element x der Trägermenge a nicht eindeutig durch die Menge seiner Vorgängerinnen bestimmt, denn verschiedene Elemente von a können die gleiche Menge von r-Vorgängern besitzen. So ist zum Beispiel für alle r-minimalen Elemente u von a die Menge $r[u]$ leer. Ist dagegen $\langle a, r\rangle$ eine Ordnung, so wird jedes Element eindeutig durch seine Vorgänger festgelegt. Das r-minimale Element u, für das $r[u] = \emptyset$, ist das kleinste Element

[5] ω ist eine Ordinalzahl, und zwar die kleinste Limeszahl. Die Elemente von ω heißen natürliche Zahlen.

der Ordnung.[6] Wir können daher in der Rekursionsvorschrift F nicht einfach nur auf die Vorgängerinnen rekurrieren, sondern müssen das Element x, auf dessen Vorgängerinnen wir uns beziehen, als Parameter in die F aufnehmen.

Theorem 2.3 (Definition durch Rekursion auf fundierten Strukturen) *Sei $\langle a, r\rangle$ eine fundierte Struktur und F eine zweistellige Operation, dann gibt es genau eine Funktion G mit* $\mathsf{Def}(G) = a$, *sodass für alle $x \in a$:*

$$G(x) = F(x, G\restriction r[x])$$

Beweis[7] Es muss gezeigt werden, dass es genau eine Funktion G gibt, die die angegebenen Bedingungen erfüllt. Wir müssen also die *Eindeutigkeit* und die *Existenz* einer solchen Funktion beweisen. Die Idee hinter dem Existenzbeweis besteht darin, dass wir die Existenz von Funktionen h beweisen, die auf Anfangsstücken von a bezüglich r definiert sind, und uns so langsam der gewünschten Funktion G annähern. Das erreichen wir, indem wir zeigen, dass $\mathsf{Def}(h) = \mathsf{Def}(G)$ für eine der so konstruierten Funktionen h. Definieren wir M als die Menge der Annäherungen an die Funktion G:

$$M = \{h \mid Fkn\ h \wedge \mathsf{Def}(h) \subseteq a \wedge \bigwedge_x (x \in \mathsf{Def}(h) \rightarrow h(x) = F(x, h\restriction r[x]) \wedge r[x] \subseteq \mathsf{Def}(h))\} \quad (1)$$

Die Elemente von M wollen wir *Anfänge* nennen. Das Ziel ist, G als die gemeinsame Fortsetzung aller Anfänge zu erhalten. Es sei $G = \bigcup M$ und wir wollen zeigen, dass G eine Funktion ist. Dazu zeigen wir zuerst, dass

$$\bigwedge_f \bigwedge_g (f, g \in M \rightarrow f\restriction(\mathsf{Def}(f) \cap \mathsf{Def}(g)) = g\restriction(\mathsf{Def}(f) \cap \mathsf{Def}(g))) \quad (2)$$

Beweis von (2) Seien $f, g \in M$ und $b = \{x \mid x \in \mathsf{Def}(f) \cap \mathsf{Def}(g) \wedge f(x) \neq g(x)\}$. Wir führen einen indirekten Beweis, um zu zeigen, dass $b = \emptyset$. Nehmen wir an, dass $b \neq \emptyset$ und dass x ein r-minimales Element aus b ist. Dann gilt $b \cap r[x] = \emptyset$ und nach (1) $r[x] \subseteq \mathsf{Def}(f) \cap \mathsf{Def}(g)$. Also ist für ein beliebiges y mit $\langle y, x\rangle \in r$, $f(y) = g(y)$, da wir x ja r-minimal gewählt haben. Nach (1) gilt dann $f(x) = F(x, f\restriction r[x]) = F(x, g\restriction r[x]) = g(x)$ und das widerspricht unserer Annahme $x \in b$. Damit ist b leer, (2) bewiesen und wir müssen nur noch zeigen, dass daraus Fkn $\bigcup M$ folgt.
Beweis, dass Fkn $\bigcup M$, wenn (2) gilt Nehmen wir an, dass $\langle x, y\rangle \in \bigcup M$ und $\langle x, z\rangle \in \bigcup M$. Dann gibt es $f, g \in M$, für die gilt, dass $\langle x, y\rangle \in f$ und $\langle x, z\rangle \in g$. Also ist $x \in \mathsf{Def}(f) \cap \mathsf{Def}(g)$ und $f(x) = g(x)$ sowie $y = z$. Insgesamt haben wir also gezeigt, dass G eine Funktion ist.

Da laut (1) $\bigwedge_h (h \in M \rightarrow \mathsf{Def}(h) \subseteq a)$, ist leicht zu sehen, dass $\mathsf{Def}(G) \subseteq \mathsf{Def}(a)$. Jetzt gilt es, auch $\mathsf{Def}(a) \subseteq \mathsf{Def}(G)$ zu zeigen, denn dann gibt es ein h, das mit G auf a übereinstimmt. Nehmen wir an, dass dem nicht so ist. Dann gibt es ein r-minimales $x \in a \setminus \mathsf{Def}(G)$, sodass $(a \setminus \mathsf{Def}(G)) \cap r[x] = \emptyset$. Es sei

[6]Siehe Seite 19

[7]Ich folge der Darstellung von Monk (1969)

$b := \{h \mid h \in M \wedge y \in \mathsf{Def}(h)\}$. Wenn $\langle y, x\rangle \in r$, dann ist $y \in \mathsf{Def}(G)$ und damit $b \neq \emptyset$ weiters ist $\bigcap b \in M$. Letzteres ist zu zeigen. Dazu beweisen wir, dass für jede nicht leere Teilmenge N von M gilt, dass $\bigcap N \in M$. Klarerweise ist $\mathsf{Def}(\bigcap N) \subseteq a$. Nehmen wir an, dass $x \in \mathsf{Def}(\bigcap N)$. Dann ist $x \in \mathsf{Def}(h)$ für alle $h \in N$. Damit ist nach (1) $r[u] \subseteq \mathsf{Def}(h)$ und nach (2) auch $r[u] \subseteq \mathsf{Def}(\bigcap N)$. Sei h ein beliebiges Element von N. Dann $x \in \mathsf{Def}(h)$ und $(\bigcap N)(x) = h(x) = F(x, h{\restriction}r[x]) = F(x, (\bigcap N){\restriction}r[x])$. Damit ist gezeigt, dass $\bigcap b \in M$.

Wir definieren nun eine Funktion H mit $\mathsf{Def}(H) = r[x]$, sodass für alle y mit $\langle y, x\rangle \in r$

$$H(y) := \bigcap b \tag{3}$$

Nach **Ers** ist $\mathsf{Bild}(H)$ eine Menge, und für $k := \bigcup \mathsf{Bild}(b)$ ist $k \in M$. Letzteres müssen wir beweisen. Das können wir tun, indem wir wieder allgemein zeigen, dass für jede Teilmenge N von M gilt $\bigcup N \in M$. Nach (2) ist $\bigcup N$ eine Funktion. Nehmen wir an, dass $x \in \mathsf{Def}(\bigcup N)$. Dann gibt es ein $h \in N$, sodass $x \in \mathsf{Def}(h)$. Nach (1) ist dann $r[x] \subseteq \mathsf{Def}(h) \subseteq \mathsf{Def}(\bigcup N)$ und $(\bigcup N)(x) = h(x) = F(x, h{\restriction} r[x]) = F(x, (\bigcup N){\restriction}r[x])$. Damit ist $k \in M$.

Aus (3) folgt $r[u] \subseteq \mathsf{Def}(k)$. Nun sei $h := k \cup \{\langle x, F(x, k{\restriction}r[x])\rangle\}$. Offensichtlich ist $h \in M$. Da aber $x \in \mathsf{Def}(h)$, gilt $x \in \mathsf{Def}(G)$. Das steht im Widerspruch zu unserer Wahl von x und damit ist $\mathsf{Def}(G) = a$.

Wir haben also gezeigt, dass G auf $\mathsf{Feld}(r)$ definiert ist. Es bleibt noch zu zeigen, dass G auch die Eigenschaft $G(x) = F(x, G{\restriction}r[x])$ für alle $x \in a$. Ist $x \in a$, dann ist $x \in \mathsf{Def}(G)$ und damit auch $x \in \mathsf{Def}(h)$ für ein bestimmtes $h \in M$. Daraus folgt $G(x) = h(x) = F(x, h{\restriction}r[x]) = F(x, G{\restriction}r[x])$. Damit haben wir die Existenz der Funktion G bewiesen.

Um die Eindeutigkeit von G zu zeigen, nehmen wir an, dass H ebenfalls den Voraussetzungen des Theorems entspricht. Sei nun $b := \{x \mid x \in a \wedge G(x) \neq H(x)\}$. Wir nehmen an, dass $b \neq \emptyset$ und wählen ein r-minimales Element aus b. Dann ist $G(y) = H(y)$ falls $\langle y, x\rangle \in r$, also $G(x) = F(x, G{\restriction}r[x]) = F(x, H{\restriction} r[x] = H(x))$. Das widerspricht unserer Wahl von $x \in b$, also ist $b = \emptyset$, $G = H$ und damit G eindeutig über die Rekursionsvorschrift und Rekursion bestimmt.
□

3 Berechenbarkeit

Nachdem wir gesehen haben, dass es unter bestimmten Voraussetzungen möglich ist, Funktionen rekursiv zu definieren, will ich mich nun mit den *rekursiven Funktionen* unter dem Gesichtspunkt der *Berechenbarkeit* beschäftigen. Als solche sind sie Gegenstand der *Rekursionstheorie*, die auch *Theorie der rekursiven Funktionen* genannt wird. Rekursive Funktionen stellen eine von mehreren Möglichkeiten dar, den unscharfen Begriff der Berechenbarkeit zu formalisieren.

3.1 Algorithmen (Effektive Prozeduren)

Um von einer Berechnung sprechen zu können, ist mindestens vorausgesetzt, dass eine Menge von symbolischen Eingaben mit symbolischen Ausgaben in Beziehung gesetzt wird. Dabei soll natürlich eine bestimmte Berechnung für dieselbe Eingabe immer dieselbe Ausgabe erzielen. Die Anweisungen, mit deren Hilfe die gesuchten Werte gefunden werden können, werden *Algorithmus* oder *effektive Prozedur* genannt. Das Wort 'Algorithmus' entstand aus dem Wort 'Algorismus' – der latinisierten Form des Eigennamens des usbekischen oder persischen Mathematikers *Abu Abdallah Muhammed ibn Musa al-Hwarizmi al-Magusi*, der etwa um 780 bis 850 lebte und über das Ziffernrechnen arbeitete. Das Wort 'Algebra' stammt vom arabischen Ausdruck 'al-ǧabr' im Titel seines Buches „Al-kitāb al-muktaṣar fī ṇisāb al-ǧabr w'al-muqābala - Kurzes Buch über das Rechnen der Algebra und Almukabala". Dabei ist 'al-ǧabr' eine Form des Wortes 'ǧabar', das herstellen oder einrichten bedeutet (Krämer 1988, p.50–52).

Berechnet wird also eine Funktion, und ein Algorithmus ist die Anleitung zur Berechnung dieser Funktion. Dabei ist zu beachten, dass die effektive Prozedur von der Funktion zu unterscheiden ist, die durch sie berechnet wird, da es mehrere effektive Prozeduren zur Berechnung derselben Funktion geben kann. Die so berechenbaren Funktionen heißen dann algorithmisch berechenbar oder berechenbar durch eine effektive Prozedur. In den Worten von Hartely Rogers: „In §1.5, we formally characterized a class of partial functions, known as the *partial recursive functions*. Henceforth, when we say that a partial function is 'effective,' 'computable,' 'effectively computable,' 'recursively computable,' 'mechanically computable,' or 'algorithmic,' we shall mean that it falls within this class. The property of being a member of this class is called *recursiveness*." (Rogers 1967, p.26–27)

Einerseits sollen also manche Funktionen als unberechenbar gelten, andererseits soll etwas über die Berechenbarkeit durch jede mögliche Rechnerin ausgesagt werden. Wir stellen uns damit die grundsätzliche Frage, ob eine Funktion

berechnet werden *könnte*, wenn unendlich viel Zeit zur Verfügung stünde, die Rechengeschwindigkeit beliebig schnell wäre oder unendlich viele Blätter Papier vorhanden wären, um die Rechensymbole aufzuschreiben. Die hier untersuchte Berechenbarkeit umfasst mehr als das, was von aktuellen Menschen oder Maschinen berechnet werden kann, da wir keine realistischen Annahmen darüber machen, welche Ressourcen uns zur Verfügung stehen. Die Berechenbarkeit wird damit nicht als relativ zum Wissen oder den Fähigkeiten eines Rechners, sondern absolut verstanden. Denn „[...] der Sachverhalt, daß eine Funktion ein Berechnungsverfahren hat, ist unabhängig davon, wer dies zu welchen Zeiten weiß. Es ist eine Eigenschaft der Funktion allein." (Oberschelp 1993, p.21) Sie ist auch nicht – wie etwa Definierbarkeit oder Beweisbarkeit – relativ zu einem Formalismus. Während in der wissenschaftlichen Präzisierung vieler intuitiver Begriffe eine Verschiebung dessen eintritt, was unter den Begriff fällt, ist dies bei der Berechenbarkeit nicht der Fall. *Alles was intuitiv berechenbar ist, ist effektiv berechenbar und umgekehrt.* Das heißt, in der Analyse dessen, was effektiv berechenbar ist, erfahren wir etwas über die Grenzen dessen, was nicht nur für Menschen überhaupt algorithmisch berechenbar ist.

Bestimmte andere Beschränkungen sind jedoch wichtig, um die Plausibilität eines Berechnungsverfahrens überprüfen zu können. Diese Beschränkungen sind es, was die Standardcharakterisierungen der effektiven Prozeduren ausmachen. Dazu gehört zum Beispiel, dass wir eine endliche Anzahl von Anweisungen haben, die in diskreten Schritten ausgeführt werden sollen. Es wird sich jedoch zeigen, dass unabhängig davon, wie die Charakterisierungen der Verfahren im einzelnen aussehen, dieselbe Klasse von Funktionen mit ihnen berechnet werden kann. Ich werde drei solche unterschiedliche Verfahren zur Charakterisierung und Ausführung von Algorithmen angeben und zeigen, dass sie dieselbe Klasse von berechenbaren Funktionen erfassen. Dies liefert Evidenz für die *Church-Turing These*, die besagt, dass die Standardcharakterisierungen eine zufriedenstellende Formalisierung von dem geben, was ein Algorithmus und damit berechenbar ist. Enthalten ist in dieser Aussage, dass die verschiedenen Präzisierungen äquivalent sind und dem intuitiven Begriff von Berechenbarkeit entsprechen.

Was aber ist das gemeinsame der Charakterisierungen? Was zeichnet einen Algorithmus aus? Rogers markiert folgende Eigenschaften, als wesentlich (Rogers 1967, p.2):

1) An algorithm is given as a set of instructions of finite size.
2) There is a computing agent, usually human, which can react to the instructions and carry out the computations.
3) There are facilities for making, storing, and retrieving steps in a computation.
4) Let P be a set of instructions as in 1) and L be a computing agent as in 2). Then L reacts to P in such a way that, for any given input, the computation is carried out in a discrete stepwise fashion, without use of continuous methods or analogue devices.
5) L reacts to P in such a way that a computation is carried forward deterministically, without resort to random methods or devices, e.g., dice.

Rogers weist auch darauf hin, dass damit ziemlich gut die Eigenschaften eines heute üblichen Digitalrechners beschrieben werden: Eigenschaft 1) entspricht dem Programm, 2) den logischen Schaltkreisen, 3) den Speichermedien, 4) seiner Digitalität und 5) seinem Mechanismus.

Marvin Minsky schlägt folgenden ersten Versuch einer Definition vor: „An effective procedure is a set of rules which tell us, from moment to moment, precisely how to behave“ (Minsky 1967), wobei jedoch, um Missverständnisse bei der Interpretation der Regeln zu vermeiden, eine formale Sprache formuliert werden soll, in der die Regeln angegeben werden können. Weiters soll eine Maschine entwickelt werden, die Anweisungen in dieser formalen Sprache ausführen kann.

Sybille Krämer nennt als die vier Merkmale von algorithmisierbaren Lösungen: 1) Elementarität – die Tätigkeit wird in elementare Operationsschritte zerlegt, 2) Determiniertheit – „Das Problem wird nicht be-arbeitet, sondern ab-gearbeitet“, 3) Allgemeinheit – Algorithmen sind einerseits unabhängig vom Material ihrer Objekte, andererseits gelten Algorithmen immer für Klassen von Problemen, nie für ein einzelnes konkretes Problem, sowie 4) Endlichkeit – endliche Anzahl von Anweisungen, endliche Ausführungszeit für eine Operation und endlich viele Schritte bis zum Ergebnis (Krämer 1988). Das Kriterium der Allgemeinheit scheint die Einschränkung, dass bei Berechnungen immer *symbolische* Ein- und Ausgaben aufeinander abgebildet werden, aufzuheben. Es ist jedoch so, dass die verwendeten Objekte immer als Bedeutungsträger interpretiert werden müssen, damit vom Ausführen eines Algorithmus gesprochen werden kann. Es muss ein Isomorphismus zwischen den „deutlich voneinander abgegrenzten“ (Krämer 1988, p.160) Objekten und den wohlunterschiedenen Zuständen des abstrakten Verfahrens bestehen. Es scheint damit uns oder anderen vernünftigen Wesen überlassen, zu entscheiden, ob die Wand vor uns eine Textverarbeitung durchführt oder nicht (Searle 1990). Es ist offensichtlich, dass es eine wesentliche Rolle in der Diskussion um AI spielt, ob ein solcher Isomorphismus als hinreichend betrachtet wird, um von der Abarbeitung eines Programms sprechen zu können.

Mit der bisher gegebenen Präzisierung des Berechenbarkeitsbegriffes sind einige wichtige Fragen noch nicht klar beantwortet worden, die vor allem die physische Implementierbarkeit betreffen. So wollen wir uns etwa, da wir uns die *prinzipielle* Frage stellen, welche Funktionen berechenbar sind, keine Beschränkungen in Bezug auf die Größe der Eingaben oder Ausgaben auferlegen. Auch wenn dies natürlich in der Praxis nicht möglich ist. Ebenso könnten wir uns entscheiden, beliebig komplexe Instruktionen in der Formulierung von Algorithmen zuzulassen. Die Anzahl der Instruktionen in einer effektiven Prozedur haben wir dagegen schon als endlich festgelegt. Beliebig komplexe Instruktionen würden eine unbeschränkte Leistungsfähigkeit der Rechnerin voraussetzen, womit alle Funktionen berechenbar würden. Da wir dies nicht wollen, beschränken wir den Instruktionssatz auf einige wenige ausgesprochen primitive Operationen. Dagegen lassen wir, wie bereits erwähnt, beliebig große Speichermedien zu, um die im Algorithmus verwendeten Symbole aufzunehmen. Ebenso verlangen wir nur, dass die Ausführung nach endlich vielen Schritten abbricht, nehmen jedoch an, dass unbeschränkt viel Zeit für die Abarbeitung zur Verfügung steht.

3.2 Rekursive Funktionen

Als erste Präzisierung aller im intuitiven Sinn berechenbaren Funktionen definiere ich in diesem Abschnitt die Klasse der *primitiv rekursiven Funktionen.* Die hier vorgestellten Funktionen müssen dabei nicht überall auf ihrer Trägermenge definiert sein. Solche Funktionen heißen *partiell*. Sind sie es doch, so werden sie *total* genannt, wobei die totalen Funktionen als spezielle partielle Funktionen aufgefasst werden. Es wird sich zeigen, dass *alle primitiv rekursiven Funktionen total* sind. Meist ist die Trägermenge $\mathbb{N}^k$, was sich gut trifft, da wir bereits gesehen haben, wie die natürlichen Zahlen als Mengen verstanden werden können. Bevor ich die Klasse der primitiv rekursiven Funktionen definiere, bringe ich jedoch die Additionsfunktion als Beispiel für eine rekursiv definierte Funktion und zeige, wie die Funktionswerte für gegebene Argumente mit Hilfe ihrer Definition ausgerechnet werden können. Ich verwende dazu die bereits bekannte Nachfolgerfunktion N, mit deren Hilfe die natürlichen Zahlen als $0, \mathsf{N}(0), \mathsf{N}(\mathsf{N}(0))$ und so weiter dargestellt werden können. Die Zeichen für die natürlichen Zahlen unterscheide ich ab hier nicht mehr von ihrer mengentheoretischen Schreibweise, da wir bereits gesehen haben, wie die natürlichen Zahlen als Mengen aufgefasst werden können.

Definition 3.1 (Additionsfunktion)

$\mathsf{Add}(x, 0) = x$

$\mathsf{Add}(x, \mathsf{N}(y)) = \mathsf{N}(\mathsf{Add}(x, y))$

Die Berechnung von $\mathsf{Add}(2, 3)$ wird dann als $\mathsf{Add}(\mathsf{N}(\mathsf{N}(0)), \mathsf{N}(\mathsf{N}(\mathsf{N}(0))))$ notiert. Wenn wir den Wert von Add für die Argumente 2 und 3 berechnen, können wir gut den Vorgang des Rekurrierens beobachten. Bildlich gesprochen steigen wir die Hierarchie der natürlichen Zahlen Stufen für Stufen hinunter, bis wir bei 0 angelangt sind, um dann mit den dabei gewonnenen Werten zur Ausgangsposition zurückzukehren. Noch einmal anders gesagt, werden wir bei der Berechnung der Funktionswerte für ein Argument an den Vorgänger des Argumentes verwiesen, und das solange, bis die Rekursion terminiert. Dies ist dann der Fall, wenn die erste in der Definition der Addition verwendete Gleichung für die Berechnung des Funktionswertes benutzt werden kann. Mit dem so berechneten ersten Funktionswert berechnen wir dann Schritt für Schritt die Werte für alle Nachfolger dieses ersten Wertes, bis wir bei den ursprünglichen Argumenten angelangt sind. Die Berechnung können wir uns also als eine ab- und aufsteigende oder auch eine zurück- und vorlaufende Bewegung vorstellen.

$$
\begin{aligned}
\mathsf{Add}(\mathsf{N}(\mathsf{N}(0)), \mathsf{N}(\mathsf{N}(\mathsf{N}(0)))) &= \mathsf{N}(\mathsf{Add}(\mathsf{N}(\mathsf{N}(0)), \mathsf{N}(\mathsf{N}(0)))) \\
\mathsf{N}(\mathsf{Add}(\mathsf{N}(\mathsf{N}(0)), \mathsf{N}(\mathsf{N}(0)))) &= \mathsf{N}(\mathsf{N}(\mathsf{Add}(\mathsf{N}(\mathsf{N}(0)), \mathsf{N}(0)))) \\
\mathsf{N}(\mathsf{N}(\mathsf{Add}(\mathsf{N}(\mathsf{N}(0)), \mathsf{N}(0)))) &= \mathsf{N}(\mathsf{N}(\mathsf{N}(\mathsf{Add}(\mathsf{N}(\mathsf{N}(0)), 0)))) \\
\mathsf{N}(\mathsf{N}(\mathsf{N}(\mathsf{Add}(\mathsf{N}(\mathsf{N}(0)), 0)))) &= \mathsf{N}(\mathsf{N}(\mathsf{N}(\mathsf{N}(\mathsf{N}(0)))))
\end{aligned}
$$

Erst in der letzten Zeile kommt dabei die Gleichung $\mathsf{Add}(x, 0) = x$ zur Anwendung, die sozusagen die ‘Abbruchbedingung’ darstellt. In allen anderen Fällen wird die zweite Gleichung der Definition benutzt.

Um die Berechnung zu verdeutlichen, stelle ich die einzelnen Schritte noch einmal getrennt dar. Wir können uns zuerst die absteigende Bewegung auf folgende Weise veranschaulichen:

$$\begin{aligned}
&\mathsf{Add}(\mathsf{N}(\mathsf{N}(0)),\mathsf{N}(\mathsf{N}(\mathsf{N}(0)))) &&= \mathsf{N}(\mathsf{Add}(\mathsf{N}(\mathsf{N}(0)),\mathsf{N}(\mathsf{N}(0))))\\
&\mathsf{Add}(\mathsf{N}(\mathsf{N}(0)),\mathsf{N}(\mathsf{N}(0))) &&= \mathsf{N}(\mathsf{Add}(\mathsf{N}(\mathsf{N}(0)),\mathsf{N}(0)))\\
&\mathsf{Add}(\mathsf{N}(\mathsf{N}(0)),\mathsf{N}(0)) &&= \mathsf{N}(\mathsf{Add}(\mathsf{N}(\mathsf{N}(0)),0))\\
&\mathsf{Add}(\mathsf{N}(\mathsf{N}(0)),0) &&= \mathsf{N}(\mathsf{N}(0))
\end{aligned}$$

Nun sind wir an der Basis der Rekursion angelangt und haben einen ersten Wert gewonnen. Diesen benützen wir nun, um während des Rekurrierens aufgetretene Funktionswerte zu berechnen. Diese aufsteigende Bewegung, bei der uns die jeweils berechneten Werte zur Verfügung stehen, können wir uns so veranschaulichen:

$$\begin{aligned}
&\mathsf{Add}(\mathsf{N}(\mathsf{N}(0)),0) = \mathsf{N}(\mathsf{N}(0))\\
&\mathsf{Add}(\mathsf{N}(\mathsf{N}(0)),\mathsf{N}(0)) = \mathsf{N}(\underbrace{\mathsf{Add}(\mathsf{N}(\mathsf{N}(0)),0)}_{=\mathsf{N}(\mathsf{N}(0))}) = \mathsf{N}(\mathsf{N}(\mathsf{N}(0)))\\
&\mathsf{Add}(\mathsf{N}(\mathsf{N}(0)),\mathsf{N}(\mathsf{N}(0))) = \mathsf{N}(\underbrace{\mathsf{Add}(\mathsf{N}(\mathsf{N}(0)),\mathsf{N}(0))}_{=\mathsf{N}(\mathsf{N}(\mathsf{N}(0)))}) = \mathsf{N}(\mathsf{N}(\mathsf{N}(\mathsf{N}(0))))\\
&\mathsf{Add}(\mathsf{N}(\mathsf{N}(0)),\mathsf{N}(\mathsf{N}(\mathsf{N}(0)))) = \mathsf{N}(\underbrace{\mathsf{Add}(\mathsf{N}(\mathsf{N}(0)),\mathsf{N}(\mathsf{N}(0)))}_{=\mathsf{N}(\mathsf{N}(\mathsf{N}(\mathsf{N}(0))))}) = \mathsf{N}(\mathsf{N}(\mathsf{N}(\mathsf{N}(\mathsf{N}(0)))))
\end{aligned}$$

3.2.1 Primitiv rekursive Funktionen

Die Additionsfunktion gehört zu einer Klasse von Funktionen, die sich mit Hilfe von wenigen Arten von *Basisfunktionen* und zwei *Operationen* definieren lassen. Alle solche Funktionen wollen wir primitiv rekursiv nennen. Sie haben gemeinsam, dass sich die Werte für Argumente berechnen lassen, indem auf Werte der Funktion für *einfachere Argumente* oder auf Werte von *einfacheren Funktionen* für dieselben Argumente zurückgegriffen wird. Im Beispiel haben wir gesehen, dass für die Berechnung der Summe von 2 und 3 die Berechnung der Summe von 2 und 2 verwendet wurde. Das Argument 3 würde mit Hilfe der Nachfolgerfunktion auf das 'einfachere' Argument 2 zurückgeführt. Dieser Prozess wurde solange wiederholt, bis für das Argument 0 die einfachere Identitätsfunktion zur Anwendung kam. Als Basisfunktionen wähle ich die Nullfunktion, die Projektionsfunktion, sowie die bereits bekannte Nachfolgerfunktion. Diese Auswahl ist nicht die einzig mögliche. So verzichtet etwa Minsky (1967, p.175) auf die Projektionsfunktion.

Definition 3.2 (Basisfunktionen)

(i) $\mathsf{Null}_i^n(x_1,\ldots,x_n) := 0$ *(Nullfunktion)*
(ii) $\mathsf{Proj}_i^n(x_1,\ldots,x_n) := x_i$ *(Projektions- oder Identitätsfunktion)*
(iii) $\mathsf{N}(x) := x+1$ *(Nachfolgerfunktion)*

Die n-stellige Nullfunktion bildet alle Argumente auf denselben Wert 0 ab. Ihre Stellenanzahl ist größer oder gleich Null, wobei die Zahl 0 als nullstellige Nullfunktion aufgefasst werden kann. Die Funktion Null^0 ist also die Zahl Null. Die Projektionsfunktion wählt das i-te ihrer n Argumente aus. Ihre Stellenanzahl ist größer als Null, und Proj_1^1 ist die Identitätsfunktion. Wir haben unendlich viele Basisfunktionen, da es sowohl unendlich viele verschiedene Null- als auch Projektionsfunktionen gibt ($\mathsf{Proj}_1^2, \mathsf{Proj}_2^2, \mathsf{Proj}_1^3, \dots$).

In der obigen Definition der Nachfolgerfunktion N darf das Zeichen „+“ nicht als Addition verstanden werden. Vielmehr handelt es sich um eine Notation der Zahlen, in der die Zahl 0 als „0“ geschrieben wird, die Zahl 1 als „0 + 1“, die Zahl 2 als „0 + 1 + 1“, und so weiter. Es ist dann klar, dass $\mathsf{N}(\mathsf{N}(0))$ nur eine andere Schreibweise für „0 + 1 + 1“ ist. Das Zeichen x steht also für das Zeichen einer Zahl, an das die Zeichenkette „+ 1“ angehängt wird. Es handelt sich bei dieser pseudounären Notation um ein *effektives Bezeichnungssystem*, das normierte Namen für die Elemente von $\mathbb{N}$ enthält. Andere effektive Bezeichnungssysteme sind etwa die Dezimaldarstellung, Binärdarstellung, Strichdarstellung (, |, ||, |||, ||||, ...) oder die Nachfolgerdarstellung $(0, \mathsf{N}(0), \mathsf{N}(\mathsf{N}(0)), \dots)$. Auch die Mengenschreibweise, in der wir die Nachfolgerin als $\mathsf{N}(x) := x \cup \{x\}$ definiert haben, stellt so ein Bezeichnungssystem dar. (Diese Erklärung der Nachfolgerfunktion über ein Notationssystem hat etwas doppeldeutiges. Dies fällt in der Nachfolgerdarstellung besonders auf, da hier die Funktion $\mathsf{N}(0)$ als Ergebnis die Zahl $\mathsf{N}(0)$ hat. Die Nachfolgerfunktion wirkt auch etwas überflüssig, da ihr Effekt bereits in der Notation enthalten ist.) Ein effektives Bezeichnungssystem, zusammen mit einer unendlichen Menge M, deren Elemente eindeutig durch das Bezeichnungsssystem benannt werden, heißt *effektiver Bereich*. Ein solcher effektiver Bereich ist zum Beispiel die Menge $\mathbb{N}$ zusammen mit der Strichdarstellung. Berechnet und entschieden kann in beliebigen effektiven Bereichen werden, also nicht nur im Bereich der natürlichen Zahlen.

Diese Basisfunktionen nehmen wir in die Klasse der *primitiv rekursiven Funktionen* auf. Weitere primitiv rekursive Funktionen erhalten wir durch die Operationen *normierte Einsetzung*, die auch Komposition oder Substitution genannt wird, und *primitive Rekursion*. Die Basisfunktionen sind in einer Weise definiert, die nichts rekursives an sich hat. Es scheint daher etwas willkürlich, sie rekursiv zu nennen. Sie sind jedoch Grenzfälle in der Festlegung des Begriffes der primitiv rekursiven Funktionen, die notwendig sind, um die Rekursion terminieren zu lassen. Ohne sie würde die unten erläuterte Operation der primitiven Rekursion in einen unendlichen Regress münden.

Definition 3.3 (normierte Einsetzung)
Aus einer m-stelligen Funktion f und m n-stelligen Funktionen g erhalten wir die Funktion h durch normierte Einsetzung, *wenn gilt*

$$h(x_1, \dots, x_n) = f(g_1(x_1, \dots, x_n), \dots, g_m(x_1, \dots, x_n))$$

In abgekürzter Schreibweise: $h = \mathsf{NE}[f, g_1, \dots, g_m]$

Beispiel Konstanten, wie die Zahl 4, die ja $\mathsf{N}(\mathsf{N}(\mathsf{N}(\mathsf{N}(0))))$ entspricht, erhalten wir durch normierte Einsetzungen der Nachfolgerfunktion. In der abgekürzten Schreibweise schreiben wir $4 = \mathsf{NE}[\mathsf{N}, \mathsf{NE}[\mathsf{N}, \mathsf{NE}[\mathsf{N}, \mathsf{NE}[\mathsf{N}, \mathsf{Null}^0]]]]$

Beispiel Eine einstellige Funktion $h(x) = x+2$, die für das Argument x immer $x + 2$ als Wert liefert, erhalten wir durch $\mathsf{NE}[\mathsf{N}, \mathsf{N}]$.
Beispiel Die Funktion $h = x + 3$ ergibt sich durch normierte Einsetzung von $\mathsf{NE}[\mathsf{N}, \mathsf{N}]$ in N. $h = \mathsf{NE}[\mathsf{N}, \mathsf{NE}[\mathsf{N}, \mathsf{N}]]$

Alle Funktion in den Beispielen sind primitiv rekursiv, da sie aus den Basisfunktionen nur mit Hilfe der normierten Einsetzung gewonnen wurden. Die abgekürzte Schreibweise ist das Zeichen einer Funktion, auch wenn sie etwas unüblich aussieht. Statt $h(x) = x + 3$, wie im letzten Beispiel, können wir also gleichbedeutend $\mathsf{NE}[\mathsf{N}, \mathsf{NE}[\mathsf{N}, \mathsf{N}]](x) = x + 3$ schreiben.

Die zweite Operation zur Erzeugung primitiv rekursiver Funktionen aus den Basisfunktionen heißt *primitive Rekursion*. Sie entspricht der bereits zu Beginn von Kapitel 2 zitierten Erklärung von Kurt Gödel, was es heißt, rekursiv zu definieren.

Definition 3.4 (primitive Rekursion)
Die Funktion h wird aus der Rekursionsvorschrift g und dem Rekursionsanfang f durch primitive Rekursion gewonnen, wenn gilt:
$h(x_1, \ldots, x_n, 0) = f(x)$
$h(x_1, \ldots, x_n, \mathsf{N}(y)) = g(x_1, \ldots, x_n, y, h(x_1, \ldots, x_n, y))$
In abgekürzter Schreibweise: $h = \mathsf{PR}[f, g]$

Die so gewonnene $n + 1$-stellige Funktion h hat als Argument an der Stelle $n+1$ die *Rekursionsvariable* y. Auf ihre Vorgänger wird bei der Berechnung der Funktionswerte solange rekurriert, bis sie den Wert Null erreicht. Für diesen Fall ist der Funktionswert durch den Rekursionsanfang f gegeben. Die weiteren n Argumente von h heißen *Rekursionsparameter*. Die Rekursionsvorschrift g ist $n + 2$-stellig.
Beispiel Wir können nun die in Definition 3.1 angegebene Funktion $\mathsf{Add}(x, 0) = x$, $\mathsf{Add}(x, \mathsf{N}(y)) = \mathsf{N}(\mathsf{Add}(x, y))$ aus den Basisfunktionen herleiten. Dazu müssen wir einen Rekursionsanfang f sowie eine Rekursionsvorschrift g finden, sodass für alle natürlichen Zahlen x, y und z gilt, dass $f(x) = x$ und $g(x, y, z) = \mathsf{N}(z)$. Für f eignet sich offensichtlich die Funktion Proj_1^1 und für g die normierte Einsetzung der dritten dreistelligen Projektionsfunktion in die Nachfolgerfunktion: $\mathsf{NE}[\mathsf{N}, \mathsf{Proj}_3^3]$. Add ist primitiv rekursiv, da sie aus den Basisfunktionen mit Hilfe der normierten Einsetzung sowie primitiver Rekursion gewonnen werden kann:
$\mathsf{Add}(x, 0) = \mathsf{Proj}_1^1(x)$
$\mathsf{Add}(x, \mathsf{N}(y)) = \mathsf{NE}[\mathsf{N}, \mathsf{Proj}_3^3](x, y, \mathsf{Add}(x, y))$
In abgekürzter Schreibweise: $\mathsf{Add} = \mathsf{PR}[\mathsf{Proj}_1^1, \mathsf{NE}[\mathsf{N}, \mathsf{Proj}_3^3]]$

Die Klasse der primitiv rekursiven Funktionen lässt sich nun durch eine induktive Definition angeben:

Definition 3.5 (Klasse der primitiv rekursiven Funktionen)
(i) *Die Basisfunktionen* Null_i^n, Proj_i^n *und* N *sind primitiv rekursiv.*
(ii) *Wenn $f, g_1, \ldots, g_m$ primitiv rekursiv sind und $h = \mathsf{NE}[f, g_1, \ldots, g_m]$, so ist auch h primitiv rekursiv.*
(iii) *Wenn f und g primitiv rekursiv sind, und $h = \mathsf{PR}[f, g]$, so ist auch h primitiv rekursiv.*
(iv) *Es gibt keine weiteren primitiv rekursiven Funktionen.*

Beispiel Als nächste Funktion können wir die Multiplikation definieren. Die Rekursionsgleichungen lauten:
$\mathsf{Mul}(x, 0) = 0$
$\mathsf{Mul}(x, \mathsf{N}(y)) = \mathsf{Add}(x, \mathsf{Mul}(x, y))$
Dies ist leicht zu sehen, da für die Multiplikation folgendes gilt:

$$\begin{aligned} x \cdot 0 &= 0 \\ x \cdot 1 &= x \\ x \cdot 2 &= x + x \\ x \cdot 3 &= x + \underbrace{x + x}_{=x \cdot 2} \\ x \cdot 4 &= x + \underbrace{x + x + x}_{=x \cdot 3} \\ &\vdots \\ x \cdot (y+1) &= x + (x \cdot y) \end{aligned}$$

Die erste und letzte dieser Gleichungen reichen aus, um die Werte der Multiplikationsfunktion für alle Argumente aus $\mathbb{N}$ berechnen zu können. Wir können daher in der üblichen informalen Infixnotation die Multiplikation als
$x \cdot 0 = 0$
$x \cdot (y + 1) = x + (x \cdot y)$
definieren. In der formalen Schreibweise mit Funktionssymbolen erhalten wir:

Definition 3.6 (Multiplikationsfunktion)
$\mathsf{Mul}(x, 0) = 0$
$\mathsf{Mul}(x, \mathsf{N}(y)) = \mathsf{Add}(x, \mathsf{Mul}(x, y))$

Um sie mit Hilfe von primitiver Rekursion (nach Def. 3.4) ausdrücken zu können, benötigen wir also eine Funktion g mit $g(x, y, z) = \mathsf{Add}(x, z)$, sowie eine Funktion f mit $f(x) = 0$. Die Funktionen sind leicht gefunden mit $f(x) = \mathsf{Null}^1_1(x)$ und $g(x, y, z) = \mathsf{NE}[\mathsf{Add}, \mathsf{Proj}^3_1, \mathsf{Proj}^3_3](x, y, z)$. Insgesamt ergibt sich also

$$\mathsf{Mul} = \mathsf{PR}[\mathsf{Null}^1_1, \mathsf{NE}[\mathsf{Add}, \mathsf{Proj}^3_1, \mathsf{Proj}^3_3]]$$

Um zu zeigen, dass es sich bei Mul um eine primitiv rekursive Funktion handelt, müssen wir sie der Definition 3.5 entsprechend herleiten. Aus dieser Definition folgt, dass eine Funktion f genau dann primitiv rekursiv ist, wenn es eine endliche Folge von Funktionen $f_1, \ldots, f_n$ gibt, sodass $f_n = f$ und für alle $j \leq n$ f_j entweder eine Basisfunktion nach Regel (i) ist, oder direkt aus den Funktionen f_i, mit $i \leq j$, durch die Regeln (ii) oder (iii) ableitbar ist. In den Worten von Kurt Gödel (1931, p.158), wobei im Zitat rekursiv als primitiv rekursiv zu lesen ist:

> Eine zahlentheoretische Funktion ϕ heißt *rekursiv*, wenn es eine endliche Reihe von zahlentheoretischen Funktionen $\phi_1, \ldots, \phi_n$ gibt, welche mit ϕ endet und die Eigenschaft hat, daß jede Funktion ϕ_k der Reihe entweder aus zwei der vorhergehenden rekursiv definiert ist oder aus irgend welchen der vorhergehenden durch Einsetzung entsteht oder schließlich eine Konstante oder die Nachfolgerunktion $x + 1$ ist.

Eine Weise, die Ableitbarkeit einer Funktion zu zeigen, besteht demnach darin, die einzelnen Schritte der Ableitung in einer Tabelle anzugeben. Jede Zeile entspricht dabei einem Ableitungsschritt und es wird in der ersten Spalte der Tabelle der neue Name der hergeleiteten Funktion festgelegt. In der zweiten Spalte wird – aus Platzgründen manchmal in abgekürzter Schreibweise – die hergeleitete Funktion angegeben. Schließlich wird in der dritten die Regel, die bei der Herleitung der entsprechenden Zeile zur Anwendung kam, zusammen mit den Funktionen, auf die sie angewandt wurde, angegeben. Wird eine Basisfunktion angeschrieben, so bezeichne ich die Regel mit BF.

f_1	Proj^1_1	BF
f_2	N	BF
f_3	Proj^3_3	BF
f_4	$\mathsf{NE}[\mathsf{N}, \mathsf{Proj}^3_3]$	NE f_2, f_3
Addition:		
$f_5 = \mathsf{Add}$	$\mathsf{PR}[\mathsf{Proj}^1_1, \mathsf{NE}[\mathsf{N}, \mathsf{Proj}^3_3]]$	PRf_1, f_4
f_6	Null^1_1	BF
f_7	Proj^3_1	BF
f_8	$\mathsf{NE}[\mathsf{Add}, \mathsf{Proj}^3_1, \mathsf{Proj}^3_3]$	NE f_5, f_7, f_3
Multiplikation:		
$f_9 = \mathsf{Mul}$	$\mathsf{PR}[\mathsf{Null}^1_1, \mathsf{NE}[\mathsf{Add}, \mathsf{Proj}^3_1, \mathsf{Proj}^3_3]]$	PRf_6, f_8
f_{10}	$\mathsf{NE}[\mathsf{N}, \mathsf{Null}^1_1]$	NE f_2, f_6
f_{11}	$\mathsf{NE}[\mathsf{Mul}, \mathsf{Proj}^3_1, \mathsf{Proj}^3_3]$	NE f_9, f_7, f_3
Potenzierung:		
$f_{12} = \mathsf{Pot}$	$\mathsf{PR}[\mathsf{NE}[\mathsf{N}, \mathsf{Null}^1_1], \mathsf{NE}[\mathsf{Mul}, \mathsf{Proj}^3_1, \mathsf{Proj}^3_3]]$	PRf_{10}, f_{11}
f_{13}	$\mathsf{NE}[\mathsf{Pot}, \mathsf{Proj}^3_1, \mathsf{Proj}^3_3]$	NE f_{12}, f_7, f_3
Hyperpotenz:		
$f_{14} = \mathsf{Hyp}$	$\mathsf{PR}[\mathsf{NE}[\mathsf{N}, \mathsf{Null}^1_1], \mathsf{NE}[\mathsf{Pot}, \mathsf{Proj}^3_1, \mathsf{Proj}^3_3]]$	PRf_{10}, f_{13}
f_{15}	Proj^2_1	BF
f_{16}	Proj^2_2	BF
f_{17}	Null^0	BF
mod. Vorgängerin:		
$f_{18} = \mathsf{Vor}$	$\mathsf{PR}[\mathsf{Null}^0, \mathsf{Proj}^2_1]$	PRf_{15}, f_{17}
f_{19}	$\mathsf{NE}[\mathsf{Vor}, \mathsf{Proj}^3_3]$	NE f_3, f_{18}
mod. Differenz:		
$f_{20} = \mathsf{Diff}$	$\mathsf{PR}[\mathsf{Proj}^1_1, \mathsf{NE}[\mathsf{Vor}, \mathsf{Proj}^3_3]]$	PRf_1, f_{19}
f_{21}	$\mathsf{NE}[\mathsf{Diff}, \mathsf{Proj}^2_1, \mathsf{Proj}^2_2]$	NE f_{20}, f_{15}, f_{16}
f_{22}	$\mathsf{NE}[\mathsf{Diff}, \mathsf{Proj}^2_2, \mathsf{Proj}^2_1]$	NE f_{20}, f_{16}, f_{15}
Abstand:		
$f_{23} = \mathsf{Abst}$	$\mathsf{NE}[\mathsf{Add}, f_{21}, f_{22}]$	NE f_5, f_{21}, f_{22}
Signumfunktionen:		
$f_{24} = \overline{\mathsf{sg}}$	$\mathsf{NE}[\mathsf{Diff}, \mathsf{NE}[\mathsf{N}, \mathsf{Null}^1_1], \mathsf{Proj}^1_1]$	NE f_{20}, f_{10}, f_1
$f_{25} = \mathsf{sg}$	$\mathsf{NE}[\mathsf{Diff}, \mathsf{NE}[\mathsf{N}, \mathsf{Null}^1_1], f_{24}]$	NE f_{20}, f_{10}, f_{24}

Für Add und Mul werde ich auch die üblichen Infixsymbole $+$ und $\cdot$ verwenden.

Wir haben damit gezeigt, dass die Funktionen f_1 bis f_{25} primitiv rekursiv sind. Eine solche Darstellung kann gut zur Berechnung eines Definitionswertes

benutzt werden. Ich illustriere das an zwei Beispielen. Die Berechnung für $f_5 = \mathsf{Add}$ von $f_5(2,3)$, sowie die Berechnung der modifizierten Differenz von 2 und 3 lautet:

$$\begin{aligned} f_5(3,2) &= f_4(3,1,f_5(3,1)) \\ &= f_4(3,1,f_4(3,0,f_5(3,0))) \\ &= f_4(3,1,f_4(3,0,f_1(3))) \\ &= f_4(3,1,f_4(3,0,3)) \\ &= f_4(3,1,f_2(f_3(3,0,3))) \\ &= f_4(3,1,f_2(3)) \\ &= f_4(3,1,4) \\ &= f_2(f_3(3,1,4)) \\ &= f_2(4) \\ &= 5 \end{aligned}$$

$$\begin{aligned} \mathsf{Diff}(2,3) &= \mathsf{Diff}(2,\mathsf{N}(2)) \\ &= \mathsf{Vor}(\mathsf{Diff}(2,\mathsf{N}(1))) \\ &= \mathsf{Vor}(\mathsf{Vor}(\mathsf{Diff}(2,\mathsf{N}(0)))) \\ &= \mathsf{Vor}(\mathsf{Vor}(\mathsf{Vor}(\mathsf{Diff}(2,0)))) \\ &= \mathsf{Vor}(\mathsf{Vor}(\mathsf{Vor}(2))) \\ &= \mathsf{Vor}(\mathsf{Vor}(1)) \\ &= \mathsf{Vor}(0) \\ &= 0 \end{aligned}$$

Jetzt etwas mehr zu den einzelnen Funktionen. Die Funktion f_{14} definiert die Hyperpotenz, die in üblicher Schreibweise die Rekursionsgleichung $\mathsf{Hyp}(x,0) = 1$, $\mathsf{Hyp}(x,\mathsf{N}(y)) = x^{\mathsf{Hyp}(x,y)}$ hat.

$$\mathsf{Hyp}(x,y) = \underbrace{x^{x^{x^{x}}}}_{\text{y-mal}}$$

Also ist zum Beispiel $\mathsf{Hyp}(2,5) = 2^{2^{2^{2^2}}} = 2^{2^{2^4}} = 2^{2^{16}} = 2^{65536} \approx 2 \cdot 10^{19728}$. Mit der Funktion f_{15} wird die modifizierte Vorgängerin definiert. Sie tut, was wir uns wahrscheinlich von ihr erwarten, mit der Ausnahme, dass die Vorgängerin von 0 gleich 0 ist: $\mathsf{Vor}(0) = 0$, $\mathsf{Vor}(\mathsf{N}(y)) = y$. Die Funktion f_{20} ist die modifizierte Differenz. Ihre Rekursionsgleichung lautet:

$$\mathsf{Diff}(x,0) = x,\ \mathsf{Diff}(x,\mathsf{N}(y)) = \mathsf{Vor}(\mathsf{Diff}(x,y))$$

Die Vorzeichenfunktionen, deren Rekursionsgleichungen

$$\mathsf{sg}(0) = 0,\ \mathsf{sg}(\mathsf{N}(x)) = 1 \quad \text{und} \quad \overline{\mathsf{sg}}(0) = 1, \overline{\mathsf{sg}}(\mathsf{N}(x)) = 0$$

lauten, sind definiert als:

$$\mathsf{sg}(x) := \begin{cases} 0, & \text{wenn } x = 0 \\ 1, & \text{wenn } x \neq 0 \end{cases} \qquad \overline{\mathsf{sg}}(x) := \begin{cases} 1, & \text{wenn } x = 0 \\ 0, & \text{wenn } x \neq 0 \end{cases}$$

$\mathsf{sg}(x)$ kann auch als $\overline{\mathsf{sg}}(\overline{\mathsf{sg}}(x))$ oder $\mathsf{Diff}(1,\mathsf{Diff}(1,x))$ definiert werden.

Jede *einstellige* Funktion f kann mehrmals hintereinander ausgeführt werden. Wird sie 0-mal wiederholt, so entspricht sie der Identitätsfunktion. Wird sie $n+1$-mal durchgeführt, so entspricht das dem Einsetzen der n-ten Wiederholung von f in f:

Definition 3.7 (Iteration)
f sei eine einstellige totale Funktion. Die ihr zugeordnete zweistellige Funktion $f^{[n]}$, die Iterierte von f, ist definiert durch:

$f^{[0]}(x) = x$
$f^{[n+1]}(x) = f(f^{[n]}(x))$

Definition 3.8 (charakteristische Funktion) *Jede n-stellige Relation R bestimmt die charakteristische Funktion χ_R, die definiert ist durch:*

$$\chi_R(x_1, \ldots, x_n) := \begin{cases} 1, & \textit{wenn } \langle x_1, \ldots, x_n \rangle \in R \\ 0, & \textit{wenn } \langle x_1, \ldots, x_n \rangle \notin R \end{cases}$$

Definition 3.9 (primitiv rekursive Relation)
Es sei $R \subseteq \mathbb{N}^n (k \geq 1)$. R ist primitiv rekursiv $:\leftrightarrow \chi_R$ ist primitiv rekursiv.

Diese Definition geht ebenfalls auf Gödel (1931, p.158) zurück, dessen obiges Zitat fortsetzt mit:

> Eine Relation zwischen natürlichen Zahlen $R(x_1, \ldots, x_n)$ heißt *rekursiv*, wenn es eine rekursive Funktion $\phi(x_1, \ldots, x_n)$ gibt, so daß für alle $x_1, x_2, \ldots, x_n$
> $$R(x_1, \ldots, x_n) \sim [\phi(x_1, \ldots, x_n) = 0].$$

Das Zeichen $\sim$ bedeutet die Äquivalenz und entspricht dem Zeichen $\leftrightarrow$ in der hier verwendeten Schreibweise. In der gödelschen Definition ist also – im Gegensatz zur hier verwendeten – die charakteristische Funktion $\phi(x_1, \ldots, x_n) = 0$, wenn $\langle x_1, \ldots, x_n \rangle \in R$, und $\phi(x_1, \ldots, x_n) = 1$, wenn $\langle x_1, \ldots, x_n \rangle \notin R$.

Theorem 3.1 (Vergleichsrelationen)
Die Relationen $=, \neq, <, >, \leq, \geq$ sind primitiv rekursiv.

Beweis Die charakteristischen Funktionen der Vergleichsrelationen sind definiert durch:

$$\begin{aligned}
\chi_=(x, y) &= \overline{\mathsf{sg}}(\mathsf{Abst}(x, y)) \\
\chi_\neq(x, y) &= \mathsf{sg}(\mathsf{Abst}(x, y)) \\
\chi_<(x, y) &= \mathsf{sg}(\mathsf{Diff}(y, x)) \\
\chi_>(x, y) &= \mathsf{sg}(\mathsf{Diff}(x, y)) \\
\chi_\leq(x, y) &= \mathsf{sg}(\mathsf{Diff}(\mathsf{Add}(y, 1), x)) \\
\chi_\geq(x, y) &= \mathsf{sg}(\mathsf{Diff}(\mathsf{Add}(x, 1), y))
\end{aligned}$$ □

Als nächstes können wir zeigen, dass die Verbindungen von Vergleichsrelationen, die auch Bedingungen genannt werden, mit Hilfe von aussagenlogischen Junktoren – wie sie in der mengentheoretischen Sprache zur Anwendung kamen – primitiv rekursiv sind.

Theorem 3.2 (aussagenlogische Junktoren)
Seien Q und R n-stellige Relationen und
$R_1 = \{\langle x_1, \ldots, x_n \rangle \mid \langle x_1, \ldots, x_n \rangle \notin Q\}$
$R_2 = \{\langle x_1, \ldots, x_n \rangle \mid \langle x_1, \ldots, x_n \rangle \in Q \wedge \langle x_1, \ldots, x_n \rangle \in R\}$
$R_3 = \{\langle x_1, \ldots, x_n \rangle \mid \langle x_1, \ldots, x_n \rangle \in Q \vee \langle x_1, \ldots, x_n \rangle \in R\}$.
Wenn Q und R primitiv rekursiv sind, dann auch R_1, R_2, R_3.

Wenn also Q eine Vergleichsrelation mit der charakteristischen Funktion χ_Q ist, dann ist $\mathsf{NE}[\overline{\mathsf{sg}}, \chi_Q]$ die charakteristische Funktion der Relation R_1, die alle

Tupeln enthält, die nicht in Q enthalten sind, und die somit die Negation von Q darstellt.

Beweis Es genügt, primitiv rekursive Definitionen für die charakteristischen Funktionen anzugeben:

$$\begin{aligned}
\chi_{R_1}(x_1,\ldots,x_n) &= \overline{\mathsf{sg}}(\chi_Q(x_1,\ldots,x_n))\\
\chi_{R_2}(x_1,\ldots,x_n) &= \mathsf{Mul}(\chi_Q(x_1,\ldots,x_n),\chi_R(x_1,\ldots,x_n))\\
\chi_{R_3}(x_1,\ldots,x_n) &= \mathsf{Add}(\chi_Q(x_1,\ldots,x_n),\chi_R(x_1,\ldots,x_n))
\end{aligned}$$ □

Alle weiteren Junktoren lassen sich auf $\neg$, $\wedge$ und $\vee$ zurückführen, wobei bereits die Definition von $\neg$ und $\wedge$ ausreicht. Die Verbindung $Q \vee R$ entspricht $\neg(\neg Q \wedge \neg R)$, und damit können wir $\chi_{R_3}(x_1,\ldots,x_n)$ als

$$\overline{\mathsf{sg}}(\mathsf{Mul}(\overline{\mathsf{sg}}(\chi_Q(x_1,\ldots,x_n)),\overline{\mathsf{sg}}(\chi_R(x_1,\ldots,x_n))))$$

erklären.

Ich habe bereits mehrfach Definitionen mit Hilfe von Fallunterscheidungen durchgeführt. Solche Definitionen hatten stets folgende Form:

$$f(x_1,\ldots,x_n) := \begin{cases} g_1(x_1,\ldots,x_n), & \text{wenn } \langle x_1,\ldots,x_n\rangle \in Q_1\\ & \vdots \\ g_r(x_1,\ldots,x_n), & \text{wenn } \langle x_1,\ldots,x_n\rangle \in Q_r \end{cases}$$

Die einzelnen Fallunterscheidungen schließen sich dabei wechselseitig aus und erschöpfen zusammen alle möglichen Fälle, was sich als

$$\bigwedge_{\langle x_1,\ldots,x_n\rangle\in\mathbb{N}^k} \Big(\bigvee_{1\le i\le r} (\langle x_1,\ldots,x_n\rangle \in Q_i) \wedge \bigwedge_{1\le i<j\le r} (\neg(\langle x_1,\ldots,x_n\rangle \in Q_i \wedge \langle x_1,\ldots,x_n\rangle \in Q_j))\Big)$$

formalisieren lässt. Da dies etwas schwierig zu lesen ist, führe ich $\mathfrak{x}$ als Abkürzungen für $\langle x_1,\ldots,x_n\rangle$ und $Q\,x_1\ldots x_n$ sowie $Q\,\mathfrak{x}$ als Abkürzung für $\mathfrak{x}\in Q$ ein. Die letzte Formel lautet dann:

$$\bigwedge_{\mathfrak{x}\in\mathbb{N}^k}\Big(\bigvee_{1\le i\le r} Q_i\,\mathfrak{x} \wedge \bigwedge_{1\le i<j\le r} \neg(Q_i\,\mathfrak{x}\wedge Q_j\,\mathfrak{x})\Big)$$

Theorem 3.3 (Definition durch Fallunterscheidung)
Wenn $Q_1,\ldots,Q_r,g_1,\ldots,g_r$ primitiv rekursiv sind, dann ist auch f primitiv rekursiv.

Beweis Da nur eine der charakteristischen Funktionen der Relationen Q_1 bis Q_r den Wert 1 annehmen wird, alle anderen aber den Wert 0, müssen wir nur den Funktionswert von g_1 bis g_r mit dem Wert der entsprechenden charakteristischen Funktion multiplizieren und die Ergebnisse anschließend addieren. Alle, bis auf einen Addenden, werden ja den Wert 0 haben.

$$f(\mathfrak{x}) := \mathsf{Add}(\mathsf{Mul}(g_1(\mathfrak{x}),\chi_{Q_1}(\mathfrak{x})),\ldots,\mathsf{Mul}(g_r(\mathfrak{x}),\chi_{Q_r}(\mathfrak{x})))$$ □

Die zweistelligen Funktionen Min und Max, welche die größere oder kleinere von zwei Zahlen liefern, lassen sich durch Fallunterscheidung definieren und sind damit primitiv rekursiv.

$$\mathsf{Min}(x,y) := \begin{cases} x, \text{ wenn } x \geq y \\ y \text{ sonst} \end{cases} \qquad \mathsf{Min}(x,y) := \begin{cases} x, \text{ wenn } x \leq y \\ y \text{ sonst} \end{cases}$$

Die Bedingung im Sonstfall muss nicht angegeben werden, da sie einfach das Komplement der ersten Bedingung bildet. Dass die Komplementbildung primitiv rekursiv ist, haben wir bereits in Theorem 3.2 gesehen.

Theorem 3.4 (Beschränkte Summation und Multiplikation)

$$g(x_1,\ldots,x_n,y) := \mathsf{Add}(f(x_1,\ldots,x_n,0),\ldots,f(x_1,\ldots,x_n,y)) = \sum_{i=0}^{y} f(x_1,\ldots,x_n,i)$$

$$h(x_1,\ldots,x_n,y) := \mathsf{Mul}(f(x_1,\ldots,x_n,0),\ldots,f(x_1,\ldots,x_n,y)) = \prod_{i=0}^{y} f(x_1,\ldots,x_n,i)$$

Beweis Die Funktionen g und h sind primitiv rekursiv, wenn f es ist, da sie durch primitive Rekursion gewonnen werden können:

$$\begin{aligned} g(x_1,\ldots,x_n,0) &= f(x_1,\ldots,x_n,0) \\ g(x_1,\ldots,x_n,y+1) &= g(x_1,\ldots,x_n,y) + f(x_1,\ldots,x_n,y+1) \end{aligned}$$

$$\begin{aligned} h(x_1,\ldots,x_n,0) &= f(x_1,\ldots,x_n,0) \\ h(x_1,\ldots,x_n,y+1) &= g(x_1,\ldots,x_n,y) \cdot f(x_1,\ldots,x_n,y+1) \end{aligned}$$

□

Hier ist schön zu sehen, wie sich die mit den Symbolen $\sum$ und $\prod$ ausgedrückte Iteration als Rekursion formulieren lässt.

Während die aussagenlogischen Junktoren primitiv rekursiv sind, gilt dies für die prädikatenlogischen Quantoren nur eingeschränkt. Das liegt daran, dass sonst alle prädikatenlogisch formulierbaren Bedingungen primitiv rekursiv wären. Das sind sie jedoch nicht. Es ist aber möglich, eingeschränkte Versionen des Allquantors und Existenzquantors zu definieren, die primitiv rekursiv sind. Diese Form der Quantifizierung wird *beschränkte Quantifizierung* (bounded quantification) genannt.

Theorem 3.5 (beschränkte Quantifizierung)
Q sei eine $n+1$-stellige Relation.

$$\begin{aligned} R_1 &= \{\langle x_1,\ldots,x_n,z\rangle \mid \bigvee_{y\leq z} \langle x_1,\ldots,x_n,y\rangle \in Q\} \\ R_2 &= \{\langle x_1,\ldots,x_n,z\rangle \mid \bigwedge_{y\leq z} \langle x_1,\ldots,x_n,y\rangle \in Q\} \end{aligned}$$

Wenn Q primitiv rekursiv ist, dann auch R_1 und R_2.

Beweis Für die Relation R_1 gilt:

$$\begin{aligned}\langle x_1,\ldots,x_n,z\rangle &\leftrightarrow \bigvee_{y\le z}\langle x_1,\ldots,x_n,y\rangle\in Q\\ &\leftrightarrow \bigvee_{y\le z}\chi_Q(x_1,\ldots,x_n,y)=1\\ &\leftrightarrow \textstyle\sum_{y=0}^{z}\chi_Q(x_1,\ldots,x_n,y)>0\\ &\leftrightarrow \mathsf{sg}\Big(\textstyle\sum_{y=0}^{z}\chi_Q(x_1,\ldots,x_n,y)\Big)=1\end{aligned}$$

Also ist $\chi_{R_1}=\mathsf{sg}\big(\sum_{y=0}^{z}\chi_Q(x_1,\ldots,x_n,y)\big)$ und damit primitiv rekursiv. Für die Relation R_2 gilt:

$$\begin{aligned}\langle x_1,\ldots,x_n,z\rangle\in R_2 &\leftrightarrow \bigwedge_{y\le z}\langle x_1,\ldots,x_n,y\rangle\in Q\\ &\leftrightarrow \bigwedge_{y\le z}\chi_Q(x_1,\ldots,x_n,y)=1\\ &\leftrightarrow \textstyle\prod_{y=0}^{z}\chi_Q(x_1,\ldots,x_n,y)>0\\ &\leftrightarrow \textstyle\prod_{y=0}^{z}\chi_Q(x_1,\ldots,x_n,y)=1\end{aligned}$$

Damit ist $\chi_{R_2}=\prod_{y=0}^{z}\chi_Q(x_1,\ldots,x_n,y)$ und nach Satz 3.4 primitiv rekursiv.□

In primitiv rekursive Relationen können primitiv rekursive Funktionen eingesetzt werden. Die resultierende Relation ist wieder primitiv rekursiv.

Theorem 3.6 (Einsetzung von Funktionen in Relationen)
Q sei eine n-stellige Relation, $g_1,\ldots,g_n$ seien k-stellige totale Funktionen und $R=\{\langle x_1,\ldots,x_k\rangle \mid \langle g_1(x_1,\ldots,x_k),\ldots,g_n(x_1,\ldots,x_k)\rangle\in Q\}$
Wenn Q, $g_1,\ldots,g_n$ primitiv rekursiv sind, dann auch R.

Beweis $\chi_R(x_1,\ldots,x_k)=\chi_Q(g_1(x_1,\ldots,x_k),\ldots,g_n(x_1,\ldots,x_k))$ □

Theorem 3.7
Alle primitiv rekursiven Funktionen sind total und im intuitiven Sinne berechenbar.

Dies lässt sich durch Induktion über den Aufbau beweisen. Dabei wird gezeigt, dass die Basisfunktionen total und berechenbar sind und diese Eigenschaft bei NE und PR erhalten bleibt. Den Beweis führe ich hier nicht an, er kann zum Beispiel in Davis (1958) nachgelesen werden.

Aus Theorem 3.7 scheint sich sofort zu ergeben, dass nicht alle berechenbaren Funktionen primitiv rekursiv sind, da es ja auch partielle Funktionen gibt, die berechenbar sind. Bei totalen Funktionen umfasst der Definitionsbereich die gesamten natürlichen Zahlen. Bei partiellen Funktionen ist das nicht der Fall, so zum Beispiel bei der für alle Argumente undefinierten Funktion e, deren Definitionsbereich leer ist. Sie ist, als überall undefinierte Funktion, der Grenzfall unter den partiellen Funktionen, wird jedoch als berechenbar angesehen. Berechenbarkeit heißt, dass es ein Berechnungsverfahren gibt, das in endlich vielen Schritten ein Ergebnis liefert, *wenn die Funktion für das zu berechnende Argument definiert ist.* Falls die Funktion für ein Argument nicht definiert ist, läuft die Berechnung endlos weiter und wir erhalten nie ein Ergebnis. Das

ist für die Funktion e bei allen Argumenten der Fall. Somit ist sie berechenbar aber nicht primitiv rekursiv. Wir könnten uns nun helfen, indem wir für Stellen, an denen eine partielle Funktion nicht definiert ist, einfach einen Wert festlegen. Zum Beispiel könnten wir für Definitionslücken den Wert 0 festlegen und alle anderen Funktionswerte um 1 erhöhen. Die Funktion e würde durch dieses Verfahren in die primitiv rekursive Funktion Null umgewandelt. Auch für viele andere partielle Funktionen würde dieses Verfahren funktionieren – ob für alle, weiß ich nicht. Es gibt jedoch auch totale Funktionen, die nicht primitiv rekursiv sind, und somit wäre dieser Umweg ziemlich sinnlos. Eine dieser nicht primitiv rekursiven totalen Funktionen werde ich nun genauer vorstellen.

3.2.2 Die Ackermann-Funktion

Nachdem wir jetzt gesehen haben, dass viele Funktionen, Relationen, Junktoren und Quantoren primitiv rekursiv sind, können wir hoffen, dass alle berechenbaren Funktionen zur Klasse der primitiv rekursiven Funktionen gehören. Wilhelm Ackermann hat jedoch 1928 gezeigt, dass es eine berechenbare Funktion gibt, die nicht primitiv rekursiv ist. Wenn wir uns die Nachfolgerfunktion, sowie die Ableitung der Addition, Multiplikation, Potenz und Hyperpotenz ansehen, so ist darin ein fortsetzbares Muster erkennbar. Um dies zu verdeutlichen, schreibe ich $f_0(x, y)$ für $\mathsf{N}(y)$, $f_1(x, y)$ für $\mathsf{Add}(x, y)$, $f_2(x, y)$ für $\mathsf{Mul}(x, y)$ und $f_3(x, y)$ für $\mathsf{Pot}(x, y)$. Erinnern wir uns, wie die Rekursionsgleichungen von Add, Mul, Pot und Hyp lauten:

$$\begin{aligned} x + 0 &= x & x \cdot 0 &= 0 \\ x + \mathsf{N}(y) &= \mathsf{N}(x + y) & x \cdot \mathsf{N}(y) &= (x \cdot y) + x \end{aligned}$$

$$\begin{aligned} x^0 &= 1 & \mathsf{Hyp}(x, 0) &= 1 \\ x^{\mathsf{N}(y)} &= x^y \cdot x & \mathsf{Hyp}(x, \mathsf{N}(y)) &= x^{\mathsf{Hyp}(x,y)} \end{aligned}$$

Es ist leicht zu sehen, dass in allen Fällen die Rekursionsvariable y bis 0 durchlaufen wird und für diesen Wert ein Anfangswert festgelegt wird. Außerdem wird in jeder Definition die jeweils links davon stehende Funktion verwendet. Links von der Additionsfunktion ist die Basisfunktion N zu denken. Die in der jeweils zweiten Zeile der Rekursionsgleichungen ausgedrückte *Schleife* hat für $y > 0$ folgende allgemeine Form:

$$f_i(x, y) = f_{i-1}(x, f_i(x, y - 1))$$

Den Index i können wir als Funktionsargument interpretieren:

$$f(i, x, y) = f(i - 1, x, f(i, x, y - 1))$$

Da das x unverändert bleibt, können wir es weglassen $f(i, y) = f(i - 1, f(i, y - 1))$ und statt i wieder x schreiben $f(x, y) = f(x - 1, f(x, y - 1))$. Wenn wir jetzt noch geeignete Anfangswerte festlegen, erhalten wir eine zweistellige Funktion, die von Rósza Péter und Raphael M. Robinson in diese vereinfachte Form gebracht wurde.

Theorem 3.8 (Ackermann-Funktion)
Es gibt eine totale zweistellige Funktion Ack, *sodass für alle* x, y *gilt:*

$\mathsf{Ack}(0, y) = y + 1$
$\mathsf{Ack}(x + 1, 0) = \mathsf{Ack}(x, 1)$
$\mathsf{Ack}(x + 1, y + 1) = \mathsf{Ack}(x, \mathsf{Ack}(x + 1, y))$

Beweis Die Gleichungen der Ackermann-Funktion stellen eine gleichzeitige Induktion über zwei Variablen dar. Es liegt also nahe, den Beweis, dass Ack total ist, durch Induktion über x und y zu führen. Da Ack zweistellig ist, können die Funktionswerte für die Argumente x und y anschaulich in eine Matrix eingetragen werden:

x \ y →	0	1	2	3	...
0	1	2	3	4	
1	2	3	4	5	
2	3	5	7	9	
3	5	13	29	61	
⋮					

Wir wollen durch Induktion über x zeigen, dass $\mathsf{Ack}(x, y)$ für alle y definiert ist. Für $x = 0$ folgt dies sofort aus der ersten Rekursionsgleichung, da diese die Werte von Ack für die gesamte erste Zeile festlegt. Nehmen wir an, dass die Funktion bereits für alle y bis zur Zeile x festgelegt ist. Durch Induktion über y wollen wir zeigen, dass Ack dann auch in der Zeile $x + 1$ für alle y festgelegt ist. Im Induktionsanfang müssen wir zeigen, dass die Behauptung für $y = 0$ gilt. Dies ist unmittelbar aus der zweiten Rekursionsgleichung und der Induktionsvoraussetzung für x zu sehen, mit der wir ja angenommen haben, dass die Funktion bis zur Zeile x festgelegt ist. Als Induktionsvoraussetzung für die Induktion über y nehmen wir jetzt an, dass die Funktion in der Zeile $x + 1$ bereits bis zur Spalte y festgelegt ist, die Behauptung also für ein beliebiges y gilt. Wir müssen zeigen, dass sie dann für $x + 1$ und $y + 1$ festgelegt ist. Das ist aber unmittelbar aus der dritten Rekursionsgleichung zu sehen. Wenn die Funktion bereits bis zur Spalte y festgelegt ist, wird der Funktionswert für die Spalte $y + 1$ mit dieser Gleichung aus bereits bekannten Werten berechnet. □

Die Ackermann-Funktion wächst sehr stark:

$$\mathsf{Ack}(4, 0) = 13$$
$$\mathsf{Ack}(4, 1) = 65533$$
$$\mathsf{Ack}(4, 1) \approx 2{,}0035 \cdot 10^{19728}$$
$$\mathsf{Ack}(5, 0) = 65533$$
$$\mathsf{Ack}(6, 0) = \mathsf{Hyp}(2, 65536) - 3 = \underbrace{2^{2^{\cdot^{\cdot^{\cdot^{2}}}}}}_{65536\text{-mal}} - 3$$

$\mathsf{Ack}(6, 0)$ ist so gross, dass sie im Dezimalsystem nicht mehr angeschrieben werden kann. Auf den derzeit üblichen Computern dauert die Berechnung von

Ack(4, 1) sehr lange. Die Berechnung von Ack(4, 2) ist in den Programmiersprachen Oberon oder Java, die im Buch von Goslin et al. (1996) beschrieben ist, bereits unmöglich. Hier ein rekursiv formuliertes Oberon-Programm zur Berechnung von Ack:

```
MODULE AckTest ;
IMPORT In, Out ;

PROCEDURE Ack( x, y : LONGINT ) : LONGINT ;
BEGIN
    IF (x = 0) THEN
        RETURN y + 1 ;
    ELSE
        IF (y = 0) THEN
            RETURN Ack( x - 1, 1 ) ;
        ELSE
            RETURN Ack( x - 1, Ack( x, y - 1 )) ;
        END
    END
END Ack ;

PROCEDURE Do* ;
VAR x, y : LONGINT ;
BEGIN
    In.Open ; In.LongInt( x ) ; In.LongInt( y ) ;
    Out.Open ; Out.Int( Ack( x, y ), 0 ) ; Out.Ln ;
END Do ;

END AckTest .
```

Um den Berechnungsvorgang zu verdeutlichen, können wir das Programm mit Ausgabeanweisungen versehen und nach *Java* portieren, um die Zeichenkettenmanipulation zu erleichtern:

```
public class AckTestPrint
    {
    static long Ack( long x, long y, String head, String tail )
        {
        System.out.print( head + "Ack( " + x + ", " + y + " )" + tail ) ;
        if ( x == 0 )
            return y + 1 ;
        else
            System.out.print( "\n" ) ;
            if ( y == 0 )
                return (Ack( x - 1, 1, head, tail)) ;
            else
                {
```

```
            long res = Ack( x, y - 1, head + "Ack( " + (x-1) + ", ",
                tail + " )" ) ;
            System.out.print( "\n" + head ) ;
            res = Ack( x - 1, res, "", "" ) ;
            System.out.print( tail ) ;
            return res ;
            }
        }

    public static void main ( String[] argv ) throws NumberFormatException
        {
        long x = Long.parseLong( argv[0] ) ;
        long y = Long.parseLong( argv[1] ) ;
        long res = Ack( x, y, "", "") ;
        System.out.println( "\n" + res ) ;
        }
    }
```

Für den Aufruf *AckTestPrint 2 1* erhalten wir zum Beispiel folgendes Ergebnis:

```
Ack( 2, 1 )
Ack( 1, Ack( 2, 0 ))
Ack( 1, Ack( 1, 1 ))
Ack( 1, Ack( 0, Ack( 1, 0 )))
Ack( 1, Ack( 0, Ack( 0, 1 )))
Ack( 1, Ack( 0, 2 ))
Ack( 1, 3 )
Ack( 0, Ack( 1, 2 ))
Ack( 0, Ack( 0, Ack( 1, 1 )))
Ack( 0, Ack( 0, Ack( 0, Ack( 1, 0 ))))
Ack( 0, Ack( 0, Ack( 0, Ack( 0, 1 ))))
Ack( 0, Ack( 0, Ack( 0, 2 )))
Ack( 0, Ack( 0, 3 ))
Ack( 0, 4 )
5
```

Theorem 3.9 *Die Ackermann-Funktion ist berechenbar.*

Das Berechnungsverfahren ist leicht aus dem Beweis von Theorem 3.8 zu ersehen. Auch das Java-Programm wirkt überzeugend, wenn wir uns vorstellen, dass es auf einer virtuellen Maschine mit beliebig viel Speicher läuft. Die Ackermann-Funktion ist also intuitiv berechenbar. Um zeigen zu können, dass sie nicht primitiv rekursiv ist, brauchen wir ein paar Hilfssätze.

Lemma 3.10

(i) $\mathsf{Ack}(1, y) = y + 2$
(ii) $\mathsf{Ack}(2, y) = 2y + 3$
(iii) $y < \mathsf{Ack}(x, y)$
(iv) $y_1 < y_2 \rightarrow \mathsf{Ack}(x, y_1) < \mathsf{Ack}(x, y_2)$

(v) $\mathsf{Ack}(x, y+1) \leq \mathsf{Ack}(x+1, y)$
(vi) $x_1 < x_2 \rightarrow \mathsf{Ack}(x_1, y) < \mathsf{Ack}(x_2, y)$
(vii) $\bigwedge_{x_1,x_2} \bigvee_{d} \bigwedge_{y} \mathsf{Ack}(x_1, y) + \mathsf{Ack}(x_2, y) < \mathsf{Ack}(d, y)$
(viii) $\bigwedge_{x_1,\ldots,x_k} \bigvee_{d} \bigwedge_{y} \sum_{i=1}^{k} \mathsf{Ack}(x_i, y) < \mathsf{Ack}(d, y)$

Die Beweise dieser Hilfstheoreme können unter anderem in Oberschelp (1993) oder Smith (1996) nachgelesen werden.

Lemma 3.11 *Zu jeder n-stelligen primitiv rekursiven Funktion f gibt es eine Zahl c, sodass gilt:* $\bigwedge_{x_1,\ldots,x_n} f(x_1, \ldots, x_n) < \mathsf{Ack}(c, x_1 + \cdots + x_n)$

Mit Hilfe dieses Lemmas lässt sich das folgende Theorem beweisen:

Theorem 3.12
Die Ackermann-Funktion ist nicht primitiv rekursiv.

Beweis Wäre Ack primitiv rekursiv, dann auch die Funktion $g(x) := \mathsf{Ack}(x, x)$. Es gäbe dann nach Lemma 3.11 eine Zahl c, sodass $\bigwedge_{x} g(x) < \mathsf{Ack}(c, x)$. Als x kann auch c selbst gewählt werden. Damit ergäbe sich die falsche Aussage $g(c) < \mathsf{Ack}(c, c) = g(c)$. Ein Widerspruch. □

Der Beweis von Lemma 3.11 ist noch nachzutragen. Er wird durch Induktion über den Aufbau der Funktion f geführt. Das heißt, es wird gezeigt, dass die im Lemma 3.11 ausgedrückte Eigenschaft für die Basisfunktionen gilt und erhalten bleibt, wenn auf beliebige primitiv rekursive Funktionen mit dieser Eigenschaft die Operationen NE oder PR angewendet wird.

Beweis
Induktionsanfang
Wir zeigen, dass die Behauptung von Lemma 3.11 für die Basisfunktionen gilt.

(i) $\mathsf{Null}_i^n(x_1, \ldots, x_n) := 0 < x_1 + \cdots + x_n + 1 = \mathsf{Ack}(0, x_1 + \cdots + x_n)$, da laut Definition $\mathsf{Ack}(0, y) = y + 1$ ist. Wir wählen für c den Wert 0.
(ii) Derselbe Wert für c ergibt sich auf die gleiche Weise für die Projektionsfunktion $\mathsf{Proj}_i^n(x_1, \ldots, x_n) := x_i < x_1 + \cdots + x_n + 1 = \mathsf{Ack}(0, x_1 + \cdots + x_n)$.
(iii) Und nach Lemma 3.10.(i) erhalten wir $c = 1$ für die Nachfolgerfunktion $\mathsf{N}(x) := x + 1 < x + 2 = \mathsf{Ack}(1, x)$.

Induktionsschritt
(i) Für die normierte Einsetzung NE nehmen wir an, dass $f(\mathfrak{x}) = \mathsf{NE}[h, g_1, \ldots, g_r]$ und $c_0, c_1, \ldots, c_r$ Zahlen sind und die Induktionsvoraussetzungen für die Funktionen g und h durch folgende Ungleichungen ausgedrückt werden:

$$h(y_1, \ldots, y_r) < \mathsf{Ack}(c_0, y_1, \ldots, y_r)$$
$$g_i(\mathfrak{x}) < \mathsf{Ack}(c_i, x_1 + \cdots + x_n) \qquad \text{für } i = 1, \ldots, r$$

Es gilt dann:

$$\begin{aligned}
f(\mathfrak{x}) &= h(g_1(\mathfrak{x}),\dots,g_r(\mathfrak{x})) \\
&< \mathsf{Ack}(c_0, g_1(\mathfrak{x})+\dots+g_r(\mathfrak{x})) && \text{Ind.-Vor. für } h\\
&< \mathsf{Ack}(c_0, \sum_{j=1}^{r}\mathsf{Ack}(c_j,\sum_{i=1}^{n}x_i)) && \text{3.10.(iv) und Ind.-Vor. für } g\\
&< \mathsf{Ack}(c_0, \mathsf{Ack}(d,\sum_{i=1}^{n}x_i)) && \text{3.10.(viii) und 3.10.(iv)}\\
&\le \mathsf{Ack}(c_0+d, \mathsf{Ack}(c_0+d+1,\sum_{i=1}^{n}x_i)) && \text{3.10.(iv) und 3.10.(vi)}\\
&= \mathsf{Ack}(c_0+d+1, \sum_{i=1}^{n}x_i+1) && \text{Theorem 3.8}\\
&\le \mathsf{Ack}(c_0+d+2, \sum_{i=1}^{n}x_i) && \text{3.10.(v)}
\end{aligned}$$

(ii) Für die primitive Rekursion PR nehmen wir an, dass $f = \mathsf{PR}[g,h]$ und c_0, c_1 Zahlen sind, mit den Induktionsvoraussetzungen

$$\begin{aligned}
g(\mathfrak{x}) &< \mathsf{Ack}(c_0, \sum_{i=1}^{n}x_i)\\
h(\mathfrak{x},y,z) &< \mathsf{Ack}(c_1, \sum_{i=1}^{n}+y+z)
\end{aligned}$$

Es gilt dann für ein gut gewähltes c_2 :

$$\begin{aligned}
\sum_{i=1}^{n}x_i + g(\mathfrak{x}) &< \mathsf{Ack}(0,\sum_{i=1}^{n}x_i) + \mathsf{Ack}(c_0,\sum_{i=1}^{n}x_i)\\
&< \mathsf{Ack}(c_2,\sum_{i=1}^{n}x_i) && \text{3.10.(iii), 3.10.(vii)}\\
\sum_{i=1}^{n}x_i + 0 + f(\mathfrak{x},0) &< \mathsf{Ack}(c_2,\sum_{i=1}^{n}x_i+0) && \text{Def. 3.4} \qquad (\star)
\end{aligned}$$

Nach Theorem 3.8 gilt auch:

$$\sum_{i=1}^{n}x_i + y + 1 \le \sum_{i=1}^{n}x_i + (y+1) + z = \mathsf{Ack}(0,\sum_{i=1}^{n}x_i+y+z)$$

und damit nach der Induktionsvoraussetzung für h:

$$\sum_{i=1}^{n}x_i + y + 1 + h(\sum_{i=1}^{n}x_i, y, z) \le \mathsf{Ack}(0,\sum_{i=1}^{n}x_i+y+z) + \mathsf{Ack}(c_1,\sum_{i=1}^{n}x_i+y+z)$$

Also nach Lemma 3.10.(vii) und eine geeignete Zahl c_3

$$\sum_{i=1}^{n} x_i + y + 1 + h(\sum_{i=1}^{n} x_i, y, z) \leq \mathsf{Ack}(c_3, \sum_{i=1}^{n} x_i + y + z) \qquad (\star\star)$$

Sei nun $c = \mathfrak{Max}(c_2, c_3)$. Durch Induktion über y können wir zeigen, dass

$$\sum_{i=1}^{n} x_i + y + f(\sum_{i=1}^{n} x_i, y) < \mathsf{Ack}(c+1, \sum_{i=1}^{n} x_i + y) \qquad (\star\star\star)$$

Beweis von $(\star\star\star)$ Im Induktionsanfang müssen wir zeigen, dass diese Behauptung für $y = 0$ gilt. Mit $c_2 < c + 1$, ist dies aus $(\star)$ ersichtlich. Im Induktionsschritt müssen wir nun zeigen, dass die Behauptung für $y + 1$ gilt, wenn sie für y gilt.

$$\begin{aligned}
&\sum_{i=1}^{n} x_i + (y+1) + f(\sum_{i=1}^{n} x_i, y+1) \\
&\qquad = \sum_{i=1}^{n} x_i + (y+1) + h(\sum_{i=1}^{n} x_i, y, f(\sum_{i=1}^{n} x_i, y)) \\
&\qquad < \mathsf{Ack}(c_3, \sum_{i=1}^{n} x_i + y + f(\sum_{i=1}^{n} x_i, y)) && \text{nach } (\star\star) \\
&\qquad < \mathsf{Ack}(c_3, \mathsf{Ack}(c+1, \sum_{i=1}^{n} x_i + y)) && \text{3.10.(iv) und Ind.-Vor.} \\
&\qquad < \mathsf{Ack}(c, \mathsf{Ack}(c+1, \sum_{i=1}^{n} x_i + y)) && \text{3.10.(vi)} \\
&\qquad = \mathsf{Ack}(c+1, \sum_{i=1}^{n} x_i + y + 1)
\end{aligned}$$

Damit ist $(\star\star\star)$ bewiesen und es folgt daraus:

$$f(\sum_{i=1}^{n} x_i, y) < \mathsf{Ack}(c+1, \sum_{i=1}^{n} x_i + y) \qquad (\star\star\star\star)$$

Damit ist der Induktionsschritt für PR vollzogen, und wir haben insgesamt bewiesen, dass Lemma 3.11 gilt. □

3.2.3 μ-Rekursivität

Um eine Klasse von berechenbaren Funktionen zu erhalten, die auch Ack enthält, nehmen wir nun eine neue Operation in die Erzeugungsregeln der Definition 3.5 (Seite 35) auf. Es handelt sich um die Operation der *Minimisierung*, die meist mit Hilfe des μ-Operators bezeichnet wird. Der μ-Operator liefert das kleinste Element einer Menge und zwar liefert er, angewandt auf einen Ausdruck, der eine Menge beschreibt, den Namen des kleinsten Elementes dieser Menge. Mit $Qx_1 \ldots x_n$ als Abkürzung für $\langle x_1, \ldots, x_n \rangle \in Q$, lautet die Definition des μ-Operators:

Definition 3.10 (μ-Operator) *Q sei eine $n+1$-stellige Relation und $\mathfrak{x} \in \mathbb{N}^n$.*

$$\mu_y[Qx_1 \dots x_n y] := \begin{cases} \text{das kleinste } y \text{ mit } Qx_1 \dots x_n y, \text{ wenn } \bigvee_y Qx_1 \dots x_n y \\ \text{undefiniert, sonst} \end{cases}$$

Durch Anwendung des μ-Operators entsteht aus einer $n+1$-stelligen Relation eine n-stellige partielle rekursive Funktion, die $\langle x_1, \dots, x_n \rangle$ auf $\mu_y[Qx_1 \dots x_n y]$ abbildet. Der μ-Operator kann auch auf $n+1$-stellige partielle Funktionen angewandt werden, und liefert dann n-stellige partielle Funktionen.

Definition 3.11
Sei g eine $n+1$-stellige partielle Funktion.

$$\mu[g] := \begin{cases} \mu_y(g(\mathfrak{x}, y) = 0 \wedge \bigwedge_{z \leq y} (\langle \mathfrak{x}, z \rangle \in \mathsf{Def}(g))), \text{ falls so ein } y \text{ existiert} \\ \text{undefiniert, sonst} \end{cases}$$

Wenn g intuitiv partiell berechenbar ist, so ist es auch $\mu[g]$.

Mit dem μ-Operator als neuen Erzeugungsprozess kann die Klasse der primitiv rekursiven Funktionen zur Klasse der μ-rekursiven Funktionen erweitert werden.

Definition 3.12 (Klasse der μ-rekursiven Funktionen)

(i) Die Basisfunktionen Null_i^n, Proj_i^n und N sind partiell μ-rekursiv.

(ii) Wenn $f, g_1, \dots, g_m$ partiell μ-rekursiv sind und $h = \mathsf{NE}[f, g_1, \dots, g_m]$, so ist auch h partiell μ-rekursiv.

(iii) Wenn f und g partiell μ-rekursiv sind und $h = \mathsf{PR}[f, g]$, so ist auch h partiell μ-rekursiv.

(iv) Wenn g partiell μ-rekursiv ist und $f = \mu[g]$, so ist auch f partiell μ-rekursiv.

(v) Es gibt keine weiteren partiell μ-rekursiven Funktionen.

f ist μ-rekursiv $:\leftrightarrow$ f ist total und partiell μ-rekursiv.
R ist μ-rekursiv $:\leftrightarrow$ χ_R ist μ-rekursiv.

Alles was primitiv rekursiv ist, ist natürlich auch μ-rekursiv. Umgekehrt gilt das nicht. Die Klasse der partiell μ-rekursiven Funktionen umfasst sogar alle intuitiv berechenbaren Funktionen, und damit auch die Ackermann-Funktion.

Theorem 3.13 Ack *ist μ-rekursiv*

Mit Hilfe des μ-Operators können alle berechenbaren Funktionen sogar ohne die Operation der primitiven Rekursion erzeugt werden. Auf PR können wir verzichten, wenn wir etwas andere Basisfunktionen wählen. Die Operation PR ist also – ähnlich wie die rekursive Definition in der Mengenlehre – kein wesentliches und damit ein reduzierbares Mittel. Um das zu zeigen, benötigen wir die Faktorisierung und Gödelisierung, die auch Arithmetisierung genannt wird, als Werkzeuge. Sie sollen als nächstes dargestellt werden.

Faktorisierung und Gödelisierung

Unsere Aufgabe ist es, beliebige Zahlentupel so in eine einzelne Zahl zu kodieren, dass sie später wieder daraus zurückgewonnen werden können. Um dies

zu erreichen, gibt es mehrere Möglichkeiten. Die von Gödel (1931) verwendete Methode beruht auf der Primfaktorzerlegung. Nach dieser kann eine Kodierungsfunktion von $\mathbb{N}^n \longrightarrow \mathbb{N}$ definiert werden, sodass jedem Tupel $\langle x_1, \ldots, x_n \rangle$ die Zahl $\prod_{i=1}^{n} \mathsf{Pr}(i)^{x_i}$ zugewiesen wird. Die Funktion Pr liefert dabei die Folge der Primzahlen.

Definition 3.13 (Primzahl) *Eine Zahl p heißt* Primzahl*, wenn sie größer als 1 ist, und außer 1 und p keinen Teiler hat.*

Es ist $\mathsf{Pr}(1) = 2, \mathsf{Pr}(2) = 3, \mathsf{Pr}(3) = 5$ und so fort. Eine einzelne Komponente des Tupels kann dann leicht über eine Komponentenfunktion

$$k(x,i) := \begin{cases} 0, \text{ falls } x = 0 \vee i = 0 \\ \text{das größte } m \text{ , für das } x \text{ durch } \mathsf{Pr}(i)^m \text{ teilbar ist} \end{cases}$$

zurückgewonnen werden.
Beispiel Das Tupel $\langle 6, 4, 1 \rangle$ wird, mit $2^6 \cdot 3^4 \cdot 5^1$, auf die Zahl $25920 = x$ abgebildet, und es ergibt sich $k(x, 1) = 6$, $k(x, 2) = 4$ und $k(x, 3) = 1$. Für alle weiteren i ist $k(x, i) = 0$.

Ich werde jedoch hier eine Kodierungsfunktion verwenden, die ohne die Funktion zur Berechnung der Primzahlen auskommt: $\tau(x, y) := \frac{1}{2}(x+y)(x+y+1) + x$. Diese Funktion ist bijektiv von $\mathbb{N}^n \longrightarrow \mathbb{N}$ und numeriert die Argumente auf folgende Weise durch:

$x \downarrow$ \ $y \rightarrow$	0	1	2	3	...
0	0	1	3	6	
1	2	4	7	11	
2	5	8	12	17	
3	9	13	18	24	
⋮					

Die entsprechenden Komponentenfunktionen lauten

$$k_1(z) := \mu_{x \leq z}(\bigvee_{y \leq z} z = \tau(x,y))$$

und

$$k_2(z) := \mu_{y \leq z}(\bigvee_{x \leq z} z = \tau(x,y)).$$

Sie liefern die erste oder zweite Komponente, die mithilfe von τ in eine Zahl kodierte wurde. In ihnen kommt eine beschränkte Variante des μ-Operators zur Anwendung. Er ist für eine $n+1$-stellige Relation R definiert als $\mu_{y \leq z}[R\mathfrak{x}\,y] := \mu_y[R\mathfrak{x}\,y \vee y \geq z]$.

Mithilfe der zweistelligen Kodierungsfunktion τ können rekursiv die n-stelligen Kodierungsfunktionen τ^n definiert werden. τ^n ist eine Bijektion von $\mathbb{N}^n$ auf $\mathbb{N}$.

Definition 3.14 (n-stellige Kodierungsfunktionen)

$\tau^1(x) := x$
$\tau^{n+1}(x_1, \ldots, x_n, x_{n+1}) := \tau(\tau^n(x_1, \ldots, x_n), x_{n+1})$

Die n-stelligen Komponentenfunktionen k_i^n, die die i-te Komponente eines n-Tupels liefern, können über die Komponentenfunktionen k_1 und k_2 definiert werden. Dabei wird eine Komponente durch *Iteration* der einstelligen Komponentenfunktionen gewonnen.

Definition 3.15 (n-stellige Komponentenfunktionen)

$$k_i^n(z) := \begin{cases} z, & \textit{falls } 1 = i = n \\ k_1^{[n-1]}(z), & \textit{falls } 1 = i < n \\ k_2(k_1^{[k-i]}(z)), & \textit{falls } 1 < i \leq n \\ 0, & \textit{sonst} \end{cases}$$

Die auf Seite 34 erwähnten effektiven Bereiche können, mit Hilfe von injektiven Funktionen, auf die natürlichen Zahlen abgebildet werden. Dieser Vorgang wird *Gödelisierung* oder *Arithmetisierung* genannt. Die injektive Funktion von Elementen eines effektiven Bereichs B auf $\mathbb{N}$ bezeichnen wir mit $\ulcorner\urcorner$. Sie liefert für Elemente $\alpha \in B$ die *Gödelnummer* $\ulcorner\alpha\urcorner$. Die Gödelnummer von Zahlenfolgen oder n-Tupel kann mit Hilfe der n-stelligen Kodierungsfunktion festgelegt werden.

Definition 3.16 (Gödelnummer von Zahlenfolgen)
$\ulcorner\mathfrak{x}\urcorner = \ulcorner\langle x_0, \ldots, x_{n-1}\rangle\urcorner := \tau(n-1, \tau^n(x_0, \ldots, x_{n-1})) + 1$

Die Gödelisierung werde ich erst später zur Bildung der Gödelnummer eines Registermaschinenprogramms benötigen.

μ-Rekursion ohne PR

Nach diesen Vorbereitungen können wir nun zeigen, dass die Klasse der partiell μ-rekursiven Funktionen auch ohne PR definiert werden kann.

Definition 3.17 (Klasse der μ-rekursiven Funktionen)

(i) *Die Basisfunktionen* Null_0^0, Proj_i^n, N, Add, Mul *und* Diff *sind partiell μ-rekursiv.*

(ii) *Wenn $f, g_1, \ldots, g_m$ partiell μ- rekursiv sind und $h = \mathsf{NE}[f, g_1, \ldots, g_m]$, so ist auch h partiell μ-rekursiv.*

(iii) *Wenn g partiell μ-rekursiv ist und $f = \mu[g]$, so ist auch f partiell μ-rekursiv.*

(iv) *Es gibt keine weiteren partiell μ-rekursiven Funktionen.*

Es muss also gezeigt werden, dass durch PR keine weiteren Funktionen gewonnen werden können. Außer der Hinzunahme weiterer Basisfunktionen und dem Weglassen von PR, unterscheidet sich diese Definition von der Definition 3.12 dadurch, dass es nur eine Nullfunktion gibt. Alle übrigen können jedoch über die modifizierte Differenz und die Projektionsfunktionen gewonnen werden: $\mathsf{Null}_i^n := \mathsf{Diff}(\mathsf{Proj}_i^n(\mathfrak{x}), \mathsf{Proj}_i^n(\mathfrak{x}))$. Wie im Abschnitt 3.2.1 für primitiv rekursive Funktionen, gebe ich nun Ableitungen für wichtige partiell μ-rekursive Funktionen an.

f_1	N	BF	
f_2	Proj_1^1	BF	
f_3	Null_0^0	BF	
f_4	Diff	BF	
$f_5 = \overline{\mathsf{sg}}$	$\mathsf{NE}[\mathsf{Diff}, \mathsf{NE}[\mathsf{N}, \mathsf{Null}_0^0], f_1]$	NE f_0, f_1, f_3	
	Vorzeichen: $\overline{\mathsf{sg}}(x) := \mathsf{Diff}(1, x)$		
f_6	Add	BF	
f_7	Proj_1^2	BF	
f_8	Proj_2^2	BF	
f_9	$\mathsf{NE}[\mathsf{Diff}, f_7, f_8]$	NE f_7, f_8	
f_{10}	$\mathsf{NE}[\mathsf{Diff}, f_8, f_7]$	NE f_7, f_8	
$f_{11} = \mathsf{Abst}$	$\mathsf{NE}[\mathsf{Add}, f_9, f_{10}]$	NE f_5, f_6, f_7	
	Abstand: $\mathsf{Abst}(x, y) = \mathsf{Diff}(x, y) + \mathsf{Diff}(y, x)$		
f_{12}	Mul	BF	
f_{13}	Proj_3^3	BF	
f_{14}	Proj_2^3	BF	
f_{15}	Proj_1^3	BF	
$f_{16} = 1$	$\mathsf{NE}[\mathsf{N}, \mathsf{Null}_0^0]$	NE f_0, f_3	1
f_{17}	$\mathsf{NE}[\mathsf{Add}, \mathsf{Proj}_3^3, 1]$	NE f_6, f_{13}, f_{16}	$z + 1$
f_{18}	$\mathsf{NE}[\mathsf{Mul}, f_{17}, \mathsf{Proj}_2^3]$	NE f_{12}, f_{14}, f_{17}	$y \cdot (z + 1)$
f_{19}	$\mathsf{NE}[\mathsf{Diff}, f_{18}, \mathsf{Proj}_1^3]$	NE f_5, f_{15}, f_{18}	$\mathsf{Diff}(y \cdot (z + 1), x)$
f_{20}	$\mathsf{NE}[\overline{\mathsf{sg}}, f_{19}]$	NE f_5, f_{19}	$y \cdot \mathsf{Diff}(y \cdot (z + 1), x)$
$f_{21} = \mathsf{Div}$	$\mu[f_{20}]$	μ f_{20}	
	ganzzahlige Division: $\mathsf{Div}(x, y) = \mu[y \cdot \mathsf{Diff}(y \cdot (z + 1), x)](x, y, z)$		
f_{22}	$\mathsf{NE}[\mathsf{Div}, \mathsf{Proj}_1^2, \mathsf{Proj}_2^2]$	NE f_7, f_8, f_{21}	$\mathsf{Div}(x, y)$
f_{23}	$\mathsf{NE}[\mathsf{Mul}, \mathsf{Proj}_2^2, f_{22}]$	NE f_8, f_{12}, f_{22}	$y \cdot \mathsf{Div}(x, y)$
$f_{24} = \mathsf{Mod}$	$\mathsf{NE}[\mathsf{Diff}, \mathsf{Proj}_1^2, f_{23}]$	NE f_4, f_7, f_{23}	
	Restfunktion: $\mathsf{Mod}(x, y) = \mathsf{Diff}(x, y \cdot \mathsf{Div}(x, y))$		

Die ganzzahlige Division $\mathsf{Div}(x, y)$ können wir berechnen, indem wir das größte z suchen, für das $y \cdot z \leq x$. Da wir aber nur den Operator des kleinsten Elements zur Verfügung haben, suchen wir statt dessen das kleinste z, für das $y \cdot (z+1) > x$. Als Sonderfall müssen wir $y = 0$ beachten. Es ergibt sich also: $\mu_z[y \cdot (z+1) > x \vee y = 0]$. Das ist gleichbedeutend mit $\mu_z[y \cdot \overline{\mathsf{sg}}(\mathsf{Diff}(y \cdot (z + 1), x)) = 0]$, da $\overline{\mathsf{sg}}(\mathsf{Diff}(y \cdot (z + 1), x))$ für alle z mit $\neg(y \cdot (z + 1) > x)$, also $y \cdot (z + 1) \leq x$, den Wert 1 ergibt, ist nur für $y = 0$ die Bedingung erfüllt. Erst für ein z mit $y \cdot (z + 1) > x$ ergibt $\overline{\mathsf{sg}}(\mathsf{Diff}(y \cdot (z + 1), x))$ den Wert 0 und erfüllt damit die Bedingung $y \cdot \overline{\mathsf{sg}}(\mathsf{Diff}(y \cdot (z+1), x)) = 0$. Wenn wir das kleinste solche z wählen, haben wir den gewünschten Quotienten.

Es ist leicht einsehbar, dass die Kodierungsfunktion τ partiell μ-rekursiv ist, da sie nur mit Hilfe der Basisfunktionen Mul, Add und Div definiert ist. Für die Komponentenfunktionen k_1 und k_2 ergibt sich das über die partielle μ-Rekursivität der beschränkten Quantifizierung, Summation und Multiplikation.

Resultate aus der Zahlentheorie

Wir benötigen nun einige Definitionen und Resultate aus der Zahlentheorie.

Definition 3.18 (Teiler) *Eine Zahl a heißt* teilbar *durch eine Zahl b, und b ist ein* Teiler *von a, wenn $b \neq 0$ und $\bigvee_{c}(a = b \cdot c)$. In Symbolen $b \mid a$. Wenn b kein Teiler von a ist, schreiben wir $b \nmid a$.*

Definition 3.19 (teilerfremd, relativ prim) *Zwei Zahlen a und b heißen* teilerfremd *oder* relativ prim*, wenn sie außer 1 keinen Teiler gemeinsam haben.*

Lemma 3.14 *Wenn $c \mid b$, dann $c \mid a \cdot b$.*

Beweis Wenn $b = c \cdot k$, dann ist $ab = c \cdot (k \cdot a)$. □

Lemma 3.15 $\bigwedge_{a,b,c} [(b > c) \wedge (a \mid b) \wedge (a \mid c)] \rightarrow a \mid (b - c)$

Beweis Aus $a \mid b$ folgt $b = a \cdot x$ und aus $a \mid c$ folgt $c = a \cdot y$. Also ist $b - c = a \cdot x - a \cdot y = a \cdot (x - y)$ und damit $a \mid b - c$. □

Wenn a, b, c nicht aus $\mathbb{N}$, sondern aus der Menge $\mathbb{Z}$ der ganzen Zahlen sind, so lässt sich auf dieselbe Weise zeigen, dass viel allgemeiner

$$\bigwedge_{a,b,c} ((a \mid b) \wedge (a \mid c)) \rightarrow a \mid (b - c)$$

und auch

$$\bigwedge_{a,b,c} ((a \mid b) \wedge (a \mid c)) \rightarrow a \mid (b + c)$$

gilt.

Lemma 3.16 *Jede Zahl $n > 1$ hat mindestens eine Primzahl als Teiler.*

Beweis Jede natürliche Zahl n kann nur endlich viele Teiler haben. Sie besitzt also einen kleinsten Teiler $p > 1$. Die Zahl p muss aber eine Primzahl sein. Wäre sie das nicht, dann hätte sie einen Teiler t mit $1 < t < p$ und damit wäre t auch ein Teiler von n. (Aus $t \mid p$ folgt $t \cdot x = p$ und aus $p \mid n$ folgt $p \cdot y = n$. Also ist $t \cdot x \cdot y = n$ und daraus folgt $t \mid n$). Das steht im Widerspurch zur Annahme, dass p der kleinste Teiler von n ist. □

Lemma 3.17 *Für zwei teilerfremde Zahlen a und b gibts es stets zwei Zahlen x und y, sodass $1 = a \cdot x + b \cdot y$.*

Aus dem Euklidischen Algorithmus ergibt sich, dass der größte gemeinsame Teiler d von zwei Zahlen a und b immer als $d = k \cdot a + l \cdot b$ geschrieben werden kann. Daraus ergibt sich für zwei teilerfremde Zahlen Lemma 3.17. Siehe zum Beispiel Courant und Robbins (1941) oder Davis (1958).

Lemma 3.18 *Wenn p eine Primzahl ist und $p \mid a \cdot b$, dann $p \mid a$ oder $p \mid b$.*

Beweis Nehmen wir an, dass p kein Teiler von a ist. Dann sind p und a teilerfremd, denn wenn sie eine von 1 verschiedene Zahl als gemeinsamen Faktor hätten, könnte das nur p sein, da p eine Primzahl ist. Nach Lemma 3.17 gibt es dann zwei Zahlen x und y, sodass $1 = a \cdot x + p \cdot y$. Also $b = a \cdot b \cdot x + p \cdot b \cdot y$. Da aber $p \mid a \cdot b$ ist, gilt für ein passendes k, dass $a \cdot b = p \cdot k$. Das ergibt $b = a \cdot b \cdot x + p \cdot b \cdot y = p \cdot k \cdot x + p \cdot b \cdot y = p \cdot (k \cdot x + b \cdot y)$ und damit $p \mid b$. □

Aus dem Lemma 3.18 folgt als Verallgemeinerung unmittelbar:

Lemma 3.19 *Wenn p eine Primzahl ist, und $p \mid a_1 \cdot a_2 \cdot a_3 \cdots a_n$, dann $p \mid a_i$, $1 \leq i \leq n$.*

Lemma 3.20 *Wenn v durch $1, 2, \ldots, n$ teilbar ist, dann sind die Zahlen $1 + v \cdot (i + 1)$, mit $i = 0, 1, 2, \ldots, n$ paarweise teilerfremd.*

Beweis Sei $m_i = 1 + v \cdot (i + 1)$. Da v durch $1, 2, \ldots, n$ teilbar ist, muss jeder von 1 verschiedene Teiler t von m_i größer als n sein. Dies deshalb, da aus $t \mid v$ nach Lemma 3.14 folgt, dass $t \mid v \cdot (i + 1)$. Wenn aber $t \neq 1$ und $t \mid v \cdot (i + 1)$, dann gilt sicher $t \nmid v \cdot (i + 1) + 1$. Da aber alle Zahlen von 1 bis n Teiler von v sind, muss $t > n$ sein. Nehmen wir nun an, dass $d \mid m_i$ und $d \mid m_j$, mit $i > j$. Es gilt dann, wieder nach Lemma 3.14, auch $d \mid (j + 1) \cdot m_i$ und $d \mid (i + 1) \cdot m_j$. Nach Lemma 3.15 folgt daraus $d \mid (i + 1) \cdot m_j - (j + 1) \cdot m_i$. Wenn wir das ausrechnen, ergibt sich mit

$$
\begin{aligned}
(i + 1) \cdot m_j - (j + 1) \cdot m_i &= (i + 1) \cdot (1 + v \cdot (j + 1)) - (j + 1) \cdot (1 + v \cdot (i + 1)) \\
&= (j + 1) + v \cdot (i + 1) \cdot (j + 1) - (i + 1) - \\
&\quad\; v \cdot (j + 1) \cdot (i + 1) \\
&= j + 1 - i - 1 \\
&= j - i,
\end{aligned}
$$

dass $d \mid j - i$. Da $i - j \leq n$ ist, muss $d = 1$ sein. □

Theorem 3.21 (Chinesischer Restsatz) *Seien $a_1, a_2, \ldots, a_k$ beliebige Zahlen und $m_1, m_2, \ldots, m_k$ seien paarweise teilerfremd. Dann gibt es eine Zahl x, sodass*

$$x \equiv a_i \pmod{m_i} \text{ für } i = 1, 2, \ldots, k.$$

Beweis Der Beweis erfolgt durch Induktion über k.

Für $k = 1$ haben wir zu zeigen, dass $x \equiv a_1 \pmod{m_1}$ für ein x gilt. Dazu müssen wir nur $x = a_1$ wählen.

Nehmen wir nun an, dass die Behauptung für $k = n$ gilt. Wir müssen zeigen, dass sie dann auch für $k = n + 1$ gilt. Wir wollen also zeigen, dass es eine Zahl x gibt, die das folgende Kongruenzsystem erfüllt.

$$
\begin{aligned}
x &\equiv a_1 \pmod{m_1}, \\
x &\equiv a_2 \pmod{m_2}, \\
&\vdots \\
x &\equiv a_n \pmod{m_n}, \\
x &\equiv a_{n+1} \pmod{m_{n+1}}.
\end{aligned}
$$

Nach der Induktionsannahme gibt es eine Zahl x_0, die die ersten n dieser Kongruenzen erfüllt.

Die Zahlen $m_1 \cdot m_2 \cdots m_n$ und m_{n+1} sind teilerfremd. Denn hätten sie einen gemeinsamen Teiler $t > 1$, dann hätten sie laut Lemma 3.16 eine Primzahl p als Teiler gemeinsam. Aber nach Lemma 3.19, folgt aus $p \mid m_1 \cdot m_2 \cdots m_n$, dass $p \mid m_j$, $i \leq j \leq n$. Dann hätten aber m_{n+1} und m_i einen gemeinsamen Teiler

$p > 1$. Das steht im Widerspruch zu unserer Annahme, dass alle m_i teilerfremd sind. Daraus folgt nach Lemma 3.17, dass es zwei Zahlen r und s gibt, sodass $r \cdot m_1 \cdot m_2 \cdots m_n + s \cdot m_{n+1} = 1$. Also ist $r \cdot (a_{n+1} - x_0) \cdot m_1 \cdot m_2 \cdots m_n + s \cdot (a_{n+1} - x_0) \cdot m_{n+1} = a_{n+1} - x_0$. Mit $q = r \cdot (a_{n+1} - x_0)$ und $p = -s \cdot (a_{n+1} - x_0)$ ergibt sich $x_0 + q \cdot m_1 \cdot m_2 \cdots m_n = a_{n+1} + p \cdot m_{n+1}$. Wählen wir nun ein t, sodass $t \cdot m_{n+1} + q \geq 0$, dann ist $x_0 + (t \cdot m_{n+1} + q) \cdot m_1 \cdot m_2 \cdots m_n \equiv a_{n+1} \pmod{m_{n+1}}$. Laut Induktionsannahme gilt aber, dass $x_0 \equiv a_i \pmod{m_i}$ für $i = 1, 2, \ldots, n$, und damit $x_0 + (t \cdot m_{n+1} + q) \cdot m_1 \cdot m_2 \cdots m_n \equiv a_i \pmod{m_i}$ für $i = 1, 2, \ldots, n$.
□

Mithilfe des Chinesischen Restsatzes lässt sich nun der nächste Hilfssatz zeigen:

Theorem 3.22 *Sei $a_0, a_1, \ldots, a_n$ eine endliche Folge natürlicher Zahlen. Dann gibt es natürliche Zahlen u und v, sodass $\mathsf{Mod}(u, v \cdot (i+1) + 1) = a_i$ für $i = 0, 1, \ldots, n$.*

Beweis Sei A die größte der a_i und sei $v = 2 \cdot A \cdot n!$. Weiters sei $m_i = 1 + v \cdot (i+1)$. Dann sind nach Lemma 3.20 die m_i paarweise relativ prim und $a_i < v < m_i$. Nach dem Chinesischen Restsatz (Theorem 3.21) gibt es nun eine Zahl u, sodass

$$u \equiv a_i \pmod{m_i} \qquad i = 0, 1, \ldots, n.$$

Also

$$\mathsf{Mod}(u, m_i) = \mathsf{Mod}(a_i, m_i) \qquad i = 0, 1, \ldots, n.$$

Da aber $a_i < m_i$, ist $\mathsf{Mod}(a_i, m_i) = a_i$ und damit

$$\mathsf{Mod}(u, 1 + v \cdot (i+1)) = \mathsf{Mod}(u, m_i) = \mathsf{Mod}(a_i, m_i) = a_i.$$ □

Lemma 3.23 *Es gibt eine partiell μ-rekursive Funktion S, sodass es für jede endliche Folge $a_1, \ldots, a_n$ natürlicher Zahlen eine Zahl w gibt, mit $S(i, w) = a_i$ für $i = 0, 1, \ldots, n$.*

Beweis Wir definieren $S(i, w) := \mathsf{Mod}(k_1(w), 1 + k_2(w) \cdot (i+1))$. S ist dann offensichtlich partiell μ-rekursiv. Seien nun Zahlen $a_0, a_1, \ldots, a_n$ gegeben. Nach Lemma 3.22 gibt es dann zwei Zahlen u und v mit $\mathsf{Mod}(u, 1 + v \cdot (i+1)) = a_i$ für $i = 0, 1, \ldots, n$.

Also $S(i, w) = \mathsf{Mod}(k_1(w), k_2(w) \cdot (i+1) + 1) = \mathsf{Mod}(u, 1 + v \cdot (i+1)) = a_i$ für $i = 0, 1, \ldots, n$. □

Elimination der Operation PR

Theorem 3.24 *Wenn g und h partiell μ-rekursive Funktionen sind und $f = \mathsf{PR}[g, h]$, dann ist auch f partiell μ-rekursiv.*

Beweis Die Funktion f ist also definiert als:

$$f(\mathfrak{x}, 0) = g(\mathfrak{x})$$
$$f(\mathfrak{x}, y+1) = h(\mathfrak{x}, y, f(\mathfrak{x}, y))$$

Für die Folge $f(\mathfrak{x}, 0), f(\mathfrak{x}, 1), \ldots, f(\mathfrak{x}, y)$ gibt es eine Zahl w mit $f(\mathfrak{x}, i) = S(i, w)$ für $i = 0, 1, \ldots, y$. Für dieses w gilt, dass

$$S(0, w) = g(\mathfrak{x}) \wedge \bigwedge_i (0 \leq 1 < y \to S(i+1, w) = h(\mathfrak{x}, i, S(i, w)))$$

Definieren wir nun die partiell μ-rekursive Hilfsfunktion H als

$$\begin{aligned} H(\mathfrak{x}, y, w) &= \mu_i[i = y \vee S(i+1, w) \neq h(\mathfrak{x}, i, S(i, w))] \\ &= \mu_i[\mathsf{Abst}(i, y) \cdot \overline{\mathsf{sg}}(\mathsf{Abst}(S(i+1, w), h(\mathfrak{x}, i, S(i, w)))) = 0] \end{aligned}$$

dann gilt, dass

$H(\mathfrak{x}, y, w) \leq y$ und

$H(\mathfrak{x}, y, w) < y \rightarrow \bigvee_i (0 \leq i < y \wedge S(i+1, w) \neq h(\mathfrak{x}, i, S(i, w)))$

Also ist

$H(\mathfrak{x}, y, w) = y \leftrightarrow \bigwedge_i (0 \leq i < y \rightarrow S(i+1, w) = h(\mathfrak{x}, i, S(i, w)))$

Es gilt also

$S(0, w) = g(\mathfrak{x}) \wedge H(\mathfrak{x}, y, w) = y.$

Über den μ-Operator erhalten wir das kleinste solche w und damit für f die Darstellung:

$$\begin{aligned} f(\mathfrak{x}, y) &= S(y, \mu_w[S(0, w) = g(\mathfrak{x}) \wedge H(\mathfrak{x}, y, w) = y]) \\ &= S(y, \mu_w[\mathsf{Abst}(S(0, w), g(\mathfrak{x})) + \mathsf{Abst}(H(\mathfrak{x}, y, w), y) = 0]) \end{aligned}$$

Also ist auch $f(\mathfrak{x}, y)$ partiell μ-rekursiv. Damit ist Theorem 3.24 bewiesen. □

Die primitive Rekursion liefert uns keine Funktionen, die nicht schon in der Klasse der partiell μ-rekursiven Funktionen enthalten wären. Die Operation der primitiven Rekursion kann also, bei geeigneter Wahl der Basisfunktionen, durch die Operation der Minimisierung ersetzt werden. Wir kommen auf diese Weise auch ohne die primitive Rekursion aus, da uns der μ-Operator eine Möglichkeit gibt einen bestimmten Zahlenbereich auf der Suche nach einem kleinsten Element zu *durchlaufen*. Die primitive Rekursion durchläuft einen Zahlenbereich im Rekurrieren auf jeweils kleinere Elemente. Der μ-Operator ist dabei sogar, wie wir an der Funktion Ack gesehen haben, mächtiger als PR.

3.2.4 Einschub: Induktive und rekursive Definitionen

Die in Definition 3.5 verwendete Definitionsweise wird von Kleene (1952, §§6, 43, 53) *induktive Definition* genannt und von *rekursiver Definition*, die auch Definition durch Induktion genannt wird, unterschieden. Die induktive Definition bildet die Basis für die im Kapitel 2.4 vorgestellten Beweisverfahren der mathematischen Induktion, denn für die durch induktive Definition generierten Objekte einer Klasse – hier Funktionen – können mittels eines Induktionsbeweises Theoreme bewiesen werden. Induktive Definitionen erlauben also, das Beweisverfahren der mathematischen Induktion auf durch sie generierte Objekte anzuwenden. In einer ähnlichen Weise erlauben induktive Definitionen die rekursive Definition von Funktionen auf der durch die induktive Definition erzeugten Trägermenge. Bei der rekursiven Definition einer Funktion wird ihr Wert für alle Werte einer Induktionsvariable y angegeben. Zuerst also der Wert der Funktion h für das Argument 0. Anschließend wird für ein beliebiges y, $h(\mathsf{N}(y))$ mit Hilfe von $h(y)$ ausgedrückt. Daraus lässt sich ersehen, dass h für jeden Wert der Induktionsvariable y definiert ist. Die rekursive Definition erlaubt uns den Wert von $h(y)$ zu berechnen, während wir die y durchlaufen. Diese

terminologische Unterscheidung zwischen induktiver und rekursiver Definition wird jedoch nicht von allen Autoren gemacht.

Marvin L. Minsky zum Beispiel nennt die von S. C. Kleene induktiv genannte Definitionsweise rekursiv (Minsky 1967, p.73). Eine solche in der Terminologie von Kleene induktive Definition besteht laut Minsky aus drei Teilen: einem Anfang, einer Rekursion und einer Einschränkung. Im Anfang wird festgelegt, dass bestimmte Objekte der zu definierende Klasse angehören. In der Rekursion werden eine oder mehrere Regeln angegeben, die festlegen, welche zusätzlichen Objekte der Klasse angehören, wenn sich bestimmte Objekte bereits in der Klasse befinden. In der Einschränkung wird festgestellt, dass keine Objekte, die sich nicht bereits durch Anfangs- und Rekursionsregel in der Klasse befinden, der Klasse angehören. Als Beispiel bringe ich hier die Klasse K der regulären Ausdrücke, wie sie von M. L. Minsky definiert wird:

Base: Any letter symbol a, b, c, ... is an expression in K.

Recursion: If E and F are expressions in K, then so is (EF).
If $E_1, E_2, \ldots, E_n$ are expressions in K, then so is
$(E_1 \vee E_2 \vee \ldots \vee E_k)$.
If E is an expression in K, then so is $E*$.[1]

Restriction: Only expressions generated by the above rules are in K.

Diese Form der Induktion wird auch *strukturelle Induktion* oder *Induktion über den Aufbau* genannt. Strukturelle Induktion legt die Elemente einer Menge wie im obigen Beispiel auf zwei grundsätzliche Weisen fest. Zu einer Menge S gehören (a) alle Elemente von einer oder mehreren vordefinierten Basismengen und (b) alle Elemente, die aus bereits in S enthaltenen Elementen, mit Hilfe von genau festgelegten Erzeugungsregeln, gebildet werden können. Das bedeutet, $S := \bigcup_{i \in \mathbb{N}} S_i$, wobei die S_i induktiv definiert sind. S_0 ist die Menge aller nach Regel (a) erhaltenen Elemente und jedes S_{i+1} ist die Menge aller Objekte, die aus einem oder mehreren Elementen von S_j, mit $j \in \{1, \ldots, i\}$, mit den Erzeugungsregeln von (b) gewonnen werden können.

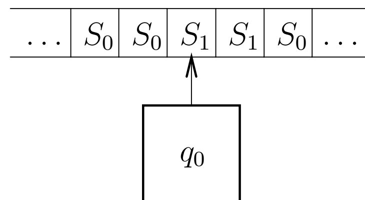
Zu dieser Form der Induktion gibt es einen entsprechenden *Beweis durch Induktion über den Aufbau.* Wenn wir zeigen wollen, dass alle Elemente von S eine bestimmte Eigenschaft P haben, dann müssen wir nur zeigen, dass erstens für alle Elemente der Basismengen P gilt und dass zweitens P, falls P auf alle Elemente in S_j zutrifft, auch auf alle Elemente in S_{i+1}, die ja durch die Erzeugungsregeln aus Elementen von S_j gewonnen werden, zutrifft.

[1] „$E*$“ ist zu lesen als „eine beliebige Anzahl von Vorkommnissen von E“.

3.3 Turingmaschinen

3.3.1 Turing-Berechenbarkeit

Eine weitere Möglichkeit, den Begriff der Berechenbarkeit zu fassen, wurde von Alan M. Turing im Jahr 1936 vorgestellt. Dieser Erklärungsversuch geht nicht von Funktionen aus, sondern von einem abstrakten Maschinenmodell, das eine idealisierte menschliche Rechnerin darstellt. Entsprechend unseren Annahmen über effektive Prozeduren ist auch für sie die Rechenzeit unbeschränkt und das Schreibmaterial unerschöpfbar. Die Anzahl der Rechensymbole ist jedoch endlich, ebenso die Anzahl der inneren Zustände. Die Turingmaschine ist nun eine mathematische Beschreibung einer auf diese Weise als unbeschränkte Rechenmaschine gedachte Rechnerin. Sie besteht aus einem unendlichen in Feldern eingeteilten Band, einem beweglichen Schreiblesekopf und einer damit verbundenen Kontrolleinheit mit endlich vielen Zuständen q_i.



Die Turingmaschine befindet sich stets in einem bestimmten Zustand q_i, kann ein Zeichen S_j vom Band lesen oder eines darauf schreiben, den Schreiblesekopf ein Feld nach links oder rechts bewegen, und befindet sich anschliessend in einem neuen Zustand q_l. Die Übergänge zwischen den Zuständen sind durch eine endliche Anzahl von Anweisungen geregelt. Sie legen das Programm der Turingmaschine fest, das aus einer endlichen Zahl von Quadrupeln der Form <aktueller Zustand, gelesenes Symbol, zu schreibendes Symbol oder Bewegungsrichtung, neuer Zustand> besteht, und Maschinentafel genannt wird. Auf jedem Feld des Bandes befindet sich eines der für die Turingmaschine erlaubten Zeichen.

Definition 3.20 (Wort) *Ein Wort ist eine endliche (auch leere) Folge der Zeichen:* $q_1, q_2, \dots; S_0, S_1, S_2, \dots; r, l$.

Die Zeichen werden unterschieden in Drucksymbole $(S_0, S_1, \dots)$, Zustände $(q_0, q_1, \dots)$ und Bewegungsrichtungen (r, l).

Definition 3.21 (Quadrupel)
Ein Quadrupel ist ein Wort in einer der drei folgenden Formen:
$\langle q_i, S_j, S_k, q_l \rangle, \langle q_i, S_j, r, q_l \rangle, \langle q_i, S_j, l, q_l \rangle$.

Definition 3.22 (Turingmaschine)
Eine Turingmaschine TM ist eine endliche (nichtleere) Menge von Quadrupel, sodass für keine zwei Quadrupel die ersten zwei Komponenten q_i *und* S_j *gleich sind.*

Jedes solche Quadrupel der Maschinentafel wird von der Turingmaschine nun in diskreten Schritten als Anweisung interpretiert. Die Anweisung $\langle q_i, S_j, -, - \rangle$

wird ausgeführt, wenn sich die Maschine im Zustand q_i befindet und gerade das Zeichen S_j vom Band liest. Es wird dann der durch die dritte Komponente angegebene Vorgang ausgeführt. Steht an dieser Stelle S_k, so wird dieses Zeichen auf das Band geschrieben. Steht dort das Zeichen r oder l, so wird der Schreiblesekopf nach rechts oder links bewegt. Anschließend befindet sich die Maschine im, durch die vierte Komponente angegeben, Folgezustand. Die Maschine liest in diesem neuen Zustand wieder ein Zeichen vom Band und führt die für diesen Zustand in der Maschinentafel angegebene Anweisung aus. Findet sich dort keine entsprechende Anweisung, so stoppt die Maschine und die Berechnung ist beendet.

Eine Turingmaschine kommt mit nur zwei Zeichen S_0 und S_1 aus, für die ich als Abkürzung auch entsprechend 0 oder 1 schreibe. Wenn nicht anders angegeben, nehme ich für alle folgenden Maschinen an, dass sie nur mit den Symbolen S_0 und S_1 operieren.

Beispiel 1 Die Turingmaschine $\{\langle q_0, 0, 1, q_1\rangle, \langle q_1, 1, 0, q_0\rangle\}$, läuft endlos, wenn sie im Zustand q_0 gestartet wird und schreibt dabei abwechselnd 1 und 0 auf das Band. Das erste Quadrupel besagt, dass sie das Zeichen 1 schreiben soll, wenn sie sich im Zustand q_0 befindet und das Zeichen 0 liest, und anschließend in den Zustand q_1 übergehen soll. Das zweite Quadrupel besagt, dass sie nun 0 schreiben und wieder in den Zustand q_0 zurückkehren soll.

Vereinfachend können wir die Mengen- und Tupelklammern, sowie die Beistriche innerhalb der Quadrupeln weglassen.

Beispiel 2 Die Turingmaschine

$$\begin{array}{c} q_0\ 0\ 1\ q_0 \\ q_0\ 1\ r\ q_1 \\ q_1\ 0\ 1\ q_1 \\ q_1\ 1\ r\ q_2 \\ q_2\ 0\ 1\ q_2, \end{array}$$

schreibt dreimal das Zeichen 1 auf das Band.

Definition 3.23 (Zustände) *Die Zustände einer Turingmaschine TM sind alle in den Quadrupeln von TM vorkommenden Zustände q_i.*

Definition 3.24 (Alphabet) *Das Alphabet einer Turingmaschine TM sind alle in den Quadrupeln vorkommenden Zeichen S_i.*

Definition 3.25 (Bandinschrift) *Ein Wort, das ausschließlich aus den Zeichen S_i besteht, heißt Bandinschrift.*

Definition 3.26 (Konfiguration) *Eine Konfiguration ist ein Wort, das genau ein Zeichen q_i enthält und in dem weder r noch l vorkommen. Das Zeichen q_i darf dabei nicht am Anfang des Wortes stehen.*

Beispiel Für die Turingmaschine TM mit dem Alphabet $\{S_0, S_1\}$ und den Zuständen q_0 ist „00011011100“ eine Bandinschrift. „$0q_0001110$“ ist eine Konfiguration.

Definition 3.27 (beob. Symbol, aktueller Zustand, Bandinschrift)
Für eine Konfiguration $\alpha = S_{i_0}S_{i_1}\dots S_{i_n}q_jS_{i_{n+1}}\dots S_{i_m}$ von TM , heißt

(i) S_{i_n} *das beobachtete Symbol von* TM *in* α,
(ii) q_j *der aktuelle Zustand von* TM *in* α,
(iii) $S_{i_0}S_{i_1}\dots S_{i_m}$ *die Bandinschrift von* TM *in* α.

Die in der Definition der Konfiguration angegebene Einschränkung, dass q_i nicht am Anfang eines Wortes stehen darf, soll also vermeiden, dass es im aktuellen Zustand kein beobachtetes Symbol gäbe.

Da auch das Fehlen eines Symbols als Symbol aufgefasst werden muss, enthalten alle Felder des Bandes Symbole. Wir können das Leersymbol mit S_0 bezeichnen. Wir stellen uns nun aber das Band nicht unendlich lang vor, sondern nur so lang wie eine bestimmte Bandinschrift. Droht der Schreiblesekopf das Band zu verlassen, so denken wir uns, dass automatisch ein neues Feld, mit dem Symbol S_0 als Inhalt, angeklebt wird.

Wir können jetzt die eingangs beschriebenen Verhaltensweisen von TM formalisieren, indem wir festlegen, wie aus einer Konfiguration die unmittelbare Folgekonfiguration erhalten wird. Dies entspricht einem Rechenschritt von TM .

Definition 3.28 (unmittelbare Folgekonfiguration) *Sei* TM *eine Turingmaschine und* α *und* β *Konfigurationen. Wir schreiben dann,* $\alpha \xrightarrow{\mathsf{TM}} \beta$, *wenn* β *die unmittelbare Folgekonfiguration von* α *für die Turingmaschine* TM *ist. Das bedeutet, dass eine der folgenden Alternativen zutrifft:*

(i) $\bigvee_P \bigvee_Q (\alpha = Pq_iS_jQ \land \beta = Pq_lS_kQ \land q_iS_jS_kq_l \in \mathsf{TM})$
(ii) $\bigvee_P \bigvee_Q (\alpha = Pq_iS_kQ \land \beta = PS_jq_lS_kQ \land q_iS_jrq_l \in \mathsf{TM})$
(iii) $\bigvee_P (\alpha = Pq_iS_j \land \beta = PS_jq_lS_0 \land q_iS_jrq_l \in \mathsf{TM})$
(iv) $\bigvee_P \bigvee_Q (\alpha = PS_kq_iS_jQ \land \beta = Pq_lS_kS_jQ \land q_iS_jlq_l \in \mathsf{TM})$
(v) $\bigvee_Q (\alpha = q_iS_jQ \land \beta = q_lS_0S_jQ \land q_iS_jlq_l \in \mathsf{TM})$

Daraus ergibt sich sofort:

Theorem 3.25
(i) $\alpha \xrightarrow{\mathsf{TM}} \beta \land \alpha \xrightarrow{\mathsf{TM}} \gamma \to \beta = \gamma$
(ii) $\alpha \xrightarrow{\mathsf{TM}} \beta \land \mathsf{TM} \subseteq \mathsf{TM}' \to \alpha \xrightarrow{\mathsf{TM}'} \beta$

Eine Konfiguration, die über mehrere Zwischenschritte erreicht wird, nennen wir einfach Folgekonfiguration.

Definition 3.29 (Folgekonfiguration)
$\alpha \overset{\mathsf{TM}}{\twoheadrightarrow} \beta :\leftrightarrow$ *Es gibt eine Folge* $\langle \alpha_0, \alpha_1, \dots, \alpha_p \rangle$ *von Konfigurationen mit* $\alpha = \alpha_0$, $\alpha_i \xrightarrow{\mathsf{TM}} \alpha_{i+1}$ *für* $i = 0, \dots, p-1$ *und* $\alpha_p = \beta$.

Definition 3.30 (Endkonfiguration)
α *ist eine Endkonfiguration von* TM $:\leftrightarrow \neg \bigvee_\beta \alpha \xrightarrow{\mathsf{TM}} \beta$

Definition 3.31 (Berechnung, Resultat)
Eine endliche Folge $\alpha_0, \alpha_1, \dots, \alpha_p$ *von Konfigurationen ist eine Berechnung der Turingmaschine* TM *genau dann, wenn* $\alpha_i \xrightarrow{\mathsf{TM}} \alpha_{i+1}$ *für* $1 \le i < p$ *und* α_p *eine*

Endkonfiguration von TM *ist. Wir nennen* α_p *das Resultat der Berechnung von* α *durch* TM *und schreiben* $\alpha_p = Res_{\mathsf{TM}}(\alpha_1)$.

Beispiel TM sei gegeben durch $\{\langle q_0, 0, r, q_0\rangle\}$. Diese Maschine läuft auf dem Band nach rechts, bis sie das erste von 0 verschiedene Symbol findet:

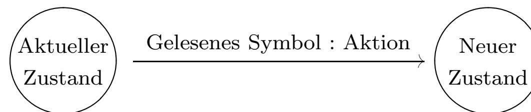
$$Res_{\mathsf{TM}}(010q_000011) = 0100001q_01$$

Ist das beobachtete Symbol in der Anfangskonfiguration ein 1, so stoppt die Maschine sofort.

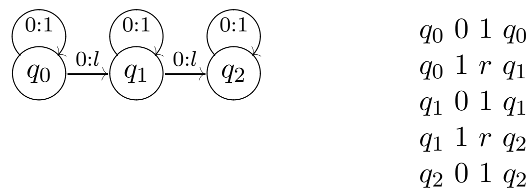
$$Res_{\mathsf{TM}}(01q_0000011) = 01q_0000011$$

Findet die Maschine keine von 0 verschiedenen Symbole, so läuft sie endlos. Wir sagen dann auch, dass die Berechnung divergiert und damit ist $Res_{\mathsf{TM}}(0q_0)$ nicht definiert.

Turingmaschinen können übersichtlicher durch *Flussdiagramme* dargestellt werden. Jeder Kreis entspricht dabei einem Zustand der Maschine. Pfeile markieren Übergänge zwischen zwei Zuständen.



Durch das folgende Flussdiagramm und die Liste von Quadrupeln werden also dieselbe Turingmaschine beschrieben:



q_0 0 1 q_0
q_0 1 r q_1
q_1 0 1 q_1
q_1 1 r q_2
q_2 0 1 q_2

Wir wollen nun partielle n-stellige Funktionen von Turingmaschinen berechnen lassen. Dazu müssen Bandinschriften als Darstellung von Zahlenfolgen dienen. Ich wähle eine unäre Darstellung, bei der jede Zahl durch eine entsprechende Anzahl von Symbolen S_1, gefolgt von einem Leersymbol S_0, repräsentiert wird:

Bandinschrift	Zahl
001100	2
0	0
11100110111110	$\langle 3, 2, 5\rangle$
000	$\langle 0, 0, 0\rangle$

Definition 3.32 (Unäre Zahlendarstellung)
Eine natürliche Zahl x *wird durch* x *aufeinander folgende Felder mit dem Symbol* S_1 *und einem anschließenden Feld mit dem Symbol* S_0 *dargestellt. Die Darstellung von* x *wird mit* $\overline{x}$ *bezeichnet. Es ist also* $\overline{x} := \underbrace{\underline{S_1 S_1 S_1 \ldots S_1 S_1 S_1}}_{x-mal\ S_1}$.
Ein n*-Tupel wird dargestellt durch die aufeinanderfolgenden Darstellungen von* $x_1, \ldots, x_n$*, also* $\overline{\mathfrak{x}} := \overline{x_1}, \ldots, \overline{x_n}$.

Definition 3.33 (Turing-Berechenbarkeit)
X und Y seien zwei beliebige Wörter. Eine n-stellige partielle Funktion φ heißt partiell Turing-berechenbar :↔ Es gibt eine Turingmaschine TM, sodass für jedes $\mathfrak{x} \in \mathbb{N}^n$ gilt: $\varphi(\mathfrak{x})$ ist genau dann definiert, wenn $Res_{\mathsf{TM}}(0\,\overline{\mathfrak{x}}\,q_0) = X\underbrace{S_1S_1\ldots S_1S_1}_{\varphi(x)-mal\ S_1}S_iq_eY$, wobei X nicht mit dem Symbol S_1 endet.
Eine partiell Turing-berechenbare Funktion, die total ist, heißt Turing-berechenbar.

Ist die Berechnung beendet, so befindet sich links vom beobachteten Symbol der Funktionswert $\varphi(\mathfrak{x})$. Das heißt, im Endzustand wird ein beliebiges Symbol beobachtet, dass nichts mit dem Resultat zu tun hat.

Definition 3.34 (Normierte Turing-Berechenbarkeit)
Sei φ eine partielle n-stellige Funktion und TM eine Turingmaschine.
TM berechnet φ normiert :↔ Für jedes $\mathfrak{x} \in \mathbb{N}^n$ gilt: $\varphi(\mathfrak{x})$ ist genau dann definiert, wenn $Res_{TM}(0\,\overline{\mathfrak{x}}\,q_0)$ definiert ist, und in diesem Fall ist $Res_{TM}(0\,\overline{\mathfrak{x}}\,q_0) = 0\,\mathfrak{x}\,\overline{\varphi(\mathfrak{x})}\,q_e$.
Ferner läuft die Maschine TM niemals nach links, wenn sie im Verlauf einer Rechnung von $0\,\overline{\mathfrak{x}}\,q_0$ aus zu einer Konfiguration $0\,q_i\,X$ kommt.

Hier wird im Endzustand das Symbol beobachtet, das die Darstellung des Funktionswertes abschließt. Alle Zwischenrechnungen werden gelöscht.

Jede Funktion, die normiert Turing-berechenbar ist, ist auch Turing-berechenbar. Die Umkehrung gilt nicht.

Theorem 3.26 *Die Funktion* Add *ist normiert Turing-berechenbar.*

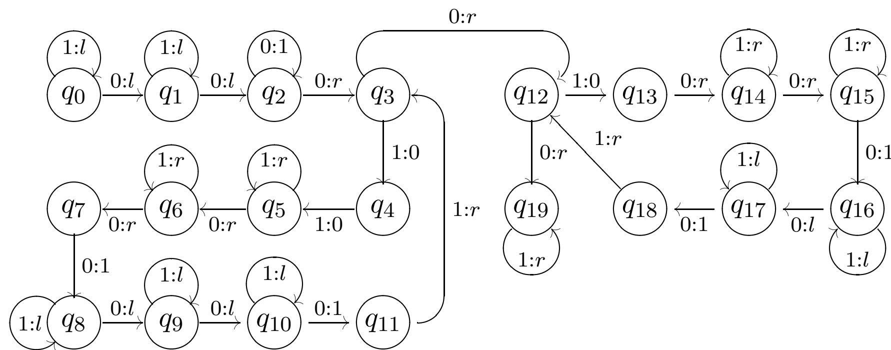
Sehen wir uns zum Beispiel an, wie eine Maschine bei der Berechnung von $\mathsf{Add}(1,3)$ aus der Anfangskonfiguration $0101110q_0$ in die Endkonfiguration $010111011110q_0$ übergeht.

$$\begin{array}{llllllllll}
\alpha_1 = 0 & 1 & 0 & 1 & 1 & 1 & 0 & q_0 & & \\
\alpha_2 = 0 & 1 & 0 & 1 & 1 & 1 & q_1 & 0 & & \\
\alpha_3 = 0 & 1 & 0 & 1 & 1 & q_1 & 1 & 0 & & \\
\alpha_4 = 0 & 1 & 0 & 1 & q_1 & 1 & 1 & 0 & & \\
\alpha_5 = 0 & 1 & 0 & q_1 & 1 & 1 & 1 & 0 & & \\
\alpha_6 = 0 & 1 & q_2 & 0 & 1 & 1 & 1 & 0 & & \\
\alpha_7 = 0 & q_2 & 1 & 0 & 1 & 1 & 1 & 0 & & \\
\alpha_8 = 0 & 1 & q_3 & 0 & 1 & 1 & 1 & 0 & & \\
\alpha_9 = 0 & 0 & q_4 & 0 & 1 & 1 & 1 & 0 & & \\
\alpha_{10} = 0 & 0 & 0 & q_5 & 1 & 1 & 1 & 0 & & \\
\alpha_{11} = 0 & 0 & 0 & 1 & q_6 & 1 & 1 & 0 & & \\
\alpha_{12} = 0 & 0 & 0 & 1 & 1 & q_6 & 1 & 0 & & \\
\alpha_{13} = 0 & 0 & 0 & 1 & 1 & 1 & q_6 & 0 & & \\
\alpha_{14} = 0 & 0 & 0 & 1 & 1 & 1 & 0 & q_6 & & \\
\alpha_{15} = 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & q_7 & \\
\alpha_{16} = 0 & 0 & 0 & 1 & 1 & 1 & 0 & 1 & q_8 & \\
\alpha_{17} = 0 & 0 & 0 & 1 & 1 & 1 & 0 & q_8 & 1 & \\
\alpha_{18} = 0 & 0 & 0 & 1 & 1 & 1 & q_9 & 0 & 1 & \\
\alpha_{19} = 0 & 0 & 0 & 1 & 1 & q_9 & 1 & 0 & 1 & \\
\alpha_{20} = 0 & 0 & 0 & 1 & q_9 & 1 & 1 & 0 & 1 & \\
\alpha_{21} = 0 & 0 & 0 & q_9 & 1 & 1 & 1 & 0 & 1 &
\end{array}$$

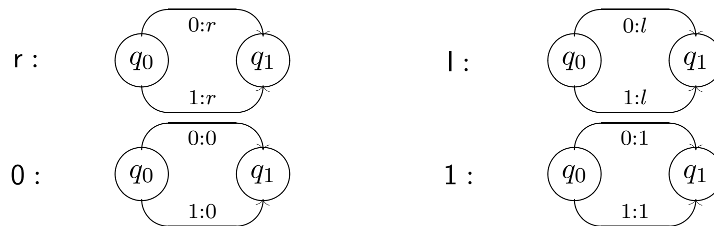
$$\begin{array}{lllllllllll}
\alpha_{22} = 0 & 0 & q_{10} & 0 & 1 & 1 & 1 & 0 & 1 & & \\
\alpha_{23} = 0 & 1 & q_{11} & 0 & 1 & 1 & 1 & 0 & 1 & & \\
\alpha_{24} = 0 & 1 & 0 & q_3 & 1 & 1 & 1 & 0 & 1 & & \\
\alpha_{25} = 0 & 1 & 0 & 1 & q_{12} & 1 & 1 & 0 & 1 & & \\
\alpha_{26} = 0 & 1 & 0 & 0 & q_{13} & 1 & 1 & 0 & 1 & & \\
\alpha_{27} = 0 & 1 & 0 & 0 & 1 & q_{14} & 1 & 0 & 1 & & \\
\alpha_{28} = 0 & 1 & 0 & 0 & 1 & 1 & q_{14} & 0 & 1 & & \\
\alpha_{29} = 0 & 1 & 0 & 0 & 1 & 1 & 0 & q_{14} & 1 & & \\
\alpha_{30} = 0 & 1 & 0 & 0 & 1 & 1 & 0 & 1 & q_{15} & & \\
\alpha_{31} = 0 & 1 & 0 & 0 & 1 & 1 & 0 & 1 & 0 & q_{15} & \\
\alpha_{32} = 0 & 1 & 0 & 0 & 1 & 1 & 0 & 1 & 1 & q_{16} & \\
\alpha_{33} = 0 & 1 & 0 & 0 & 1 & 1 & 0 & 1 & q_{16} & 1 & \\
\alpha_{34} = 0 & 1 & 0 & 0 & 1 & 1 & 0 & q_{16} & 1 & 1 & \\
\alpha_{35} = 0 & 1 & 0 & 0 & 1 & 1 & q_{17} & 0 & 1 & 1 & \\
\alpha_{36} = 0 & 1 & 0 & 0 & 1 & q_{17} & 1 & 0 & 1 & 1 & \\
\alpha_{37} = 0 & 1 & 0 & 0 & q_{17} & 1 & 1 & 0 & 1 & 1 & \\
\alpha_{38} = 0 & 1 & 0 & 1 & q_{18} & 1 & 1 & 0 & 1 & 1 & \\
\alpha_{39} = 0 & 1 & 0 & 1 & 1 & q_{12} & 1 & 0 & 1 & 1 & \\
\alpha_{40} = 0 & 1 & 0 & 1 & 0 & q_{13} & 1 & 0 & 1 & 1 & \\
\alpha_{41} = 0 & 1 & 0 & 1 & 0 & 1 & q_{14} & 0 & 1 & 1 & \\
\alpha_{42} = 0 & 1 & 0 & 1 & 0 & 1 & 0 & q_{14} & 1 & 1 &
\end{array}$$

$\alpha_{43} = 0\ 1\ 0\ 1\ 0\ 1\ 0\ 1\ q_{15}\ 1$
$\alpha_{44} = 0\ 1\ 0\ 1\ 0\ 1\ 0\ 1\ 1\ q_{15}$
$\alpha_{45} = 0\ 1\ 0\ 1\ 0\ 1\ 0\ 1\ 1\ 0\ q_{15}$
$\alpha_{46} = 0\ 1\ 0\ 1\ 0\ 1\ 0\ 1\ 1\ 1\ q_{16}$
$\alpha_{47} = 0\ 1\ 0\ 1\ 0\ 1\ 0\ 1\ 1\ q_{16}\ 1$
$\alpha_{48} = 0\ 1\ 0\ 1\ 0\ 1\ 0\ 1\ q_{16}\ 1\ 1$
$\alpha_{49} = 0\ 1\ 0\ 1\ 0\ 1\ 0\ q_{16}\ 1\ 1\ 1$
$\alpha_{50} = 0\ 1\ 0\ 1\ 0\ 1\ q_{17}\ 0\ 1\ 1\ 1$
$\alpha_{51} = 0\ 1\ 0\ 1\ 0\ q_{17}\ 1\ 0\ 1\ 1\ 1$
$\alpha_{52} = 0\ 1\ 0\ 1\ 1\ q_{17}\ 1\ 0\ 1\ 1\ 1$
$\alpha_{53} = 0\ 1\ 0\ 1\ 1\ 1\ q_{12}\ 0\ 1\ 1\ 1$
$\alpha_{54} = 0\ 1\ 0\ 1\ 1\ 0\ q_{13}\ 0\ 1\ 1\ 1$
$\alpha_{55} = 0\ 1\ 0\ 1\ 1\ 0\ 0\ q_{14}\ 1\ 1\ 1$
$\alpha_{56} = 0\ 1\ 0\ 1\ 1\ 0\ 0\ 1\ q_{14}\ 1\ 1$
$\alpha_{57} = 0\ 1\ 0\ 1\ 1\ 0\ 0\ 1\ 1\ q_{15}\ 1$
$\alpha_{58} = 0\ 1\ 0\ 1\ 1\ 0\ 0\ 1\ 1\ 1\ q_{15}$
$\alpha_{59} = 0\ 1\ 0\ 1\ 1\ 0\ 0\ 1\ 1\ 1\ 0\ q_{15}$
$\alpha_{60} = 0\ 1\ 0\ 1\ 1\ 0\ 0\ 1\ 1\ 1\ 1\ q_{16}$
$\alpha_{61} = 0\ 1\ 0\ 1\ 1\ 0\ 0\ 1\ 1\ 1\ q_{16}\ 1$
$\alpha_{62} = 0\ 1\ 0\ 1\ 1\ 0\ 0\ 1\ 1\ q_{16}\ 1\ 1$
$\alpha_{63} = 0\ 1\ 0\ 1\ 1\ 0\ 0\ 1\ q_{16}\ 1\ 1\ 1$
$\alpha_{64} = 0\ 1\ 0\ 1\ 1\ 0\ 0\ q_{16}\ 1\ 1\ 1\ 1$
$\alpha_{65} = 0\ 1\ 0\ 1\ 1\ 0\ q_{17}\ 0\ 1\ 1\ 1\ 1$
$\alpha_{66} = 0\ 1\ 0\ 1\ 1\ 1\ q_{18}\ 0\ 1\ 1\ 1\ 1$
$\alpha_{67} = 0\ 1\ 0\ 1\ 1\ 1\ 0\ q_{12}\ 1\ 1\ 1\ 1$
$\alpha_{68} = 0\ 1\ 0\ 1\ 1\ 1\ 0\ 1\ q_{19}\ 1\ 1\ 1$
$\alpha_{69} = 0\ 1\ 0\ 1\ 1\ 1\ 0\ 1\ 1\ q_{19}\ 1\ 1$
$\alpha_{70} = 0\ 1\ 0\ 1\ 1\ 1\ 0\ 1\ 1\ 1\ q_{19}\ 1$
$\alpha_{71} = 0\ 1\ 0\ 1\ 1\ 1\ 0\ 1\ 1\ 1\ 1\ q_{19}$
$\alpha_{72} = 0\ 1\ 0\ 1\ 1\ 1\ 0\ 1\ 1\ 1\ 1\ 0\ q_{19}$

Die Berechnung könnte von der folgenden Turingmaschine durchgeführt worden sein:



Wir wollen nun einige einfachste Maschinen konstruieren, um dann aus ihnen durch Kopplung komplexere zusammengesetzte Maschinen zu erhalten.

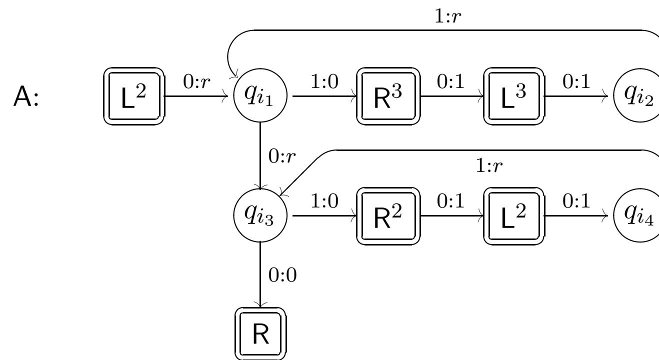


Zwei Maschinen TM_1 und TM_2 können aneinander gekoppelt werden. Dabei müssen die Zustände der zweiten Maschine so umbenannt werden, dass die Maschinen anschließend keinen Zustand mehr gemeinsam haben. Für einen Endzustand von TM_1 muss dann nur ein Kopplungsquadrupel definiert werden, dass diese Maschine in einen Zustand von TM_2 überführt. Eine Maschine kann mithilfe eines Kopplungsquadrupels auch rückgekoppelt werden.

Besteht die Maschine R aus den Quadrupeln $\langle q_0, 0, r, q_1\rangle, \langle q_0, 1, r, q_1\rangle$, so dient $\langle q_1, 1, 1, q_0\rangle$ als Rückkopplungsquadrupel. Entsprechendes gilt für die Maschine L. Die Maschine R geht zunächst einen Schritt nach rechts – egal welches Symbol sich im Zustand q_0 auf dem Band befindet –, und läuft dann solange nach rechts, bis sie auf das Symbol 0 stößt. Die Maschine L macht dasselbe, nur läuft sie dabei nach links.

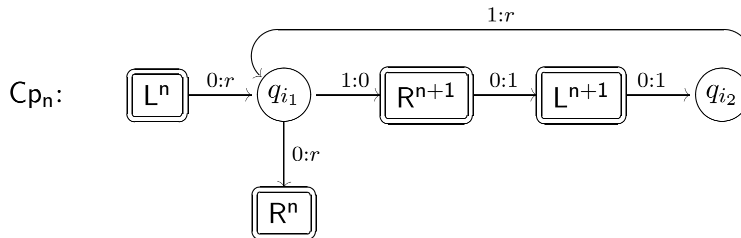
Diese Maschine läuft über zwei durch „0" getrennte Blöcke von Einsen nach rechts. Wir können allgemein TM^k für eine Maschine schreiben, die aus k hintereinandergekoppelten Kopien von TM entstanden ist.

Aus den bisher vorgestellten Komponenten können wir nun eine zweite – im Aufbau etwas kompliziertere – Additionsmaschine A bauen:

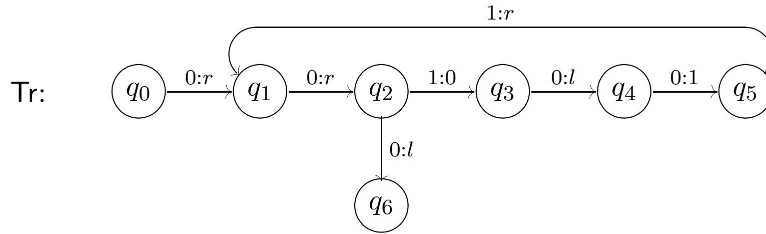


Bei der Berechnung von $0\,\overline{\mathfrak{x}\,\mathfrak{y}}\,q_a \xrightarrow{\mathsf{A}} 0\,\mathfrak{x}\,\mathfrak{y}\,\overline{(\mathfrak{x}+\mathfrak{y})}\,q_e$ bewegt sich die Maschine zuerst zwei Einserblöcke nach links, das heißt, sie positioniert sich an den Anfang des ersten Arguments der Additionsfunktion, und geht dann einen Schritt nach rechts. Findet sie dort die erste Eins von $\overline{\mathfrak{x}\,\mathfrak{y}}$, so geht sie im Diagramm rechts weiter und überträgt das erste Argument ans Ende des Bandes. Findet sie dort eine Null, so wurde $\overline{\mathfrak{x}}$ erfolgreich übertragen und der Kopf ist zwischen $\overline{\mathfrak{x}}$ und $\overline{\mathfrak{y}}$ positioniert. Die Abarbeitung wird im Diagramm mit der unteren *Schleife* fortgesetzt, die $\overline{\mathfrak{y}}$ an das Bandende überträgt.

Eine weitere nützliche Maschine ist die n-Kopiermaschine $\mathsf{Cp_n}$, die die Darstellung der n-ten Zahl links vom Kopf an das Bandende kopiert. Zu Beginn der Kopieroperation muss dabei der Kopf rechts von der letzten Zahlendarstellung stehen.

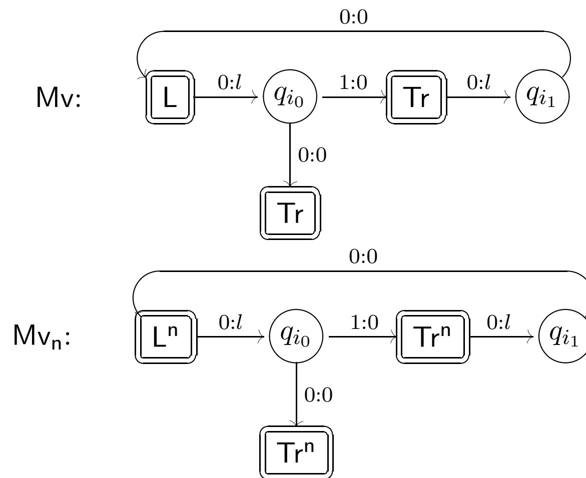


Die Translationsmaschine verschiebt einen Einserblock um ein Feld nach links: $X\,0\,q_a\,0\,\overline{\mathfrak{x}}\,Y \xrightarrow{\mathsf{Tr}} X\,0\,\overline{\mathfrak{x}}\,q_e\,0\,Y$.

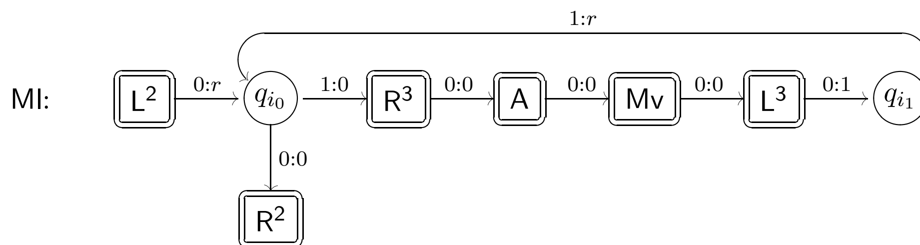


Die Verschiebemaschine Mv verschiebt die Darstellung einer Zahl $\overline{\mathfrak{y}}$ nach links und löscht dabei eine dort befindliche weitere Zahlendarstellung $\overline{\mathfrak{x}}$: $X\,0\,\overline{\mathfrak{x}}\,\overline{\mathfrak{y}}\,q_a \xrightarrow{\mathsf{Mv}} 0\,\overline{\mathfrak{y}}\,q_e$. Die allgemeine Verschiebemaschine $\mathsf{Mv_n}$ verschiebt n-Zahlendarstellungen nach links und löscht dabei ein dort befindliches $\overline{\mathfrak{x}}$:

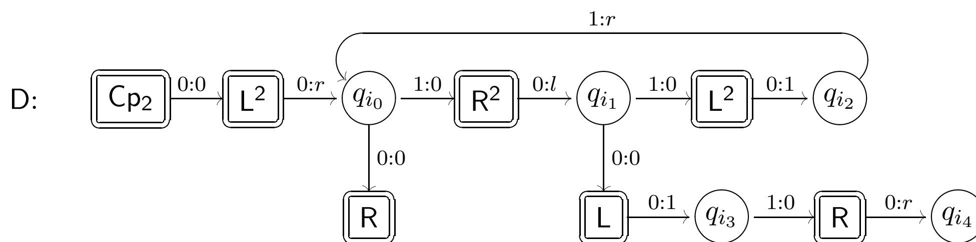
$$X\,0\,\overline{\mathfrak{x}\,\mathfrak{y}_\mathfrak{n}\,\mathfrak{y}_{\mathfrak{n}-1}\,\ldots\,\mathfrak{y}_1}\,q_a \xrightarrow{\mathsf{Mv_n}} 0\,\overline{\mathfrak{y}_\mathfrak{n}\,\mathfrak{y}_{\mathfrak{n}-1}\,\ldots\,\mathfrak{y}_1}\,q_e.$$



Die Multiplikationsmaschine Ml kann auf folgende Weise gebaut werden:



Die Maschine D zur Berechnung der modifizierten Differenz:



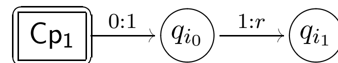
3.3.2 Äquivalenz von μ-Rekursivität und Turing-Berechenbarkeit

Mit den bisher konstruierten Turingmaschinen können wir nun auf einfache Weise zeigen, dass die Begriffe μ-Rekursivität und Turing-Berechenbarkeit denselben Umfang haben.

Theorem 3.27
Eine Funktion φ ist partiell μ-rekursiv $\leftrightarrow$ φ ist partiell Turing-berechenbar.

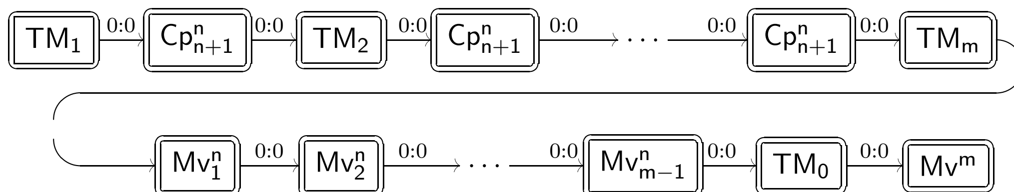
Beweis Ich zeige hier nur die eine Hälfte des Bikonditionals, nämlich dass jede partiell μ-rekursive Funktion auch partiell Turing-berechenbar ist. Der Beweis erfolgt durch Induktion über den Aufbau der partiell μ-rekursiven Funktionen nach Definition 3.17.

Die Basisfunktionen Add, Mul und Diff werden von den Turingmaschinen A, Ml und D berechnet. Für Null_0^0 ist die Maschine r ausreichend. Die Funktion Proj_i^n wird durch $\mathsf{K_{n+1-i}}$ und N durch die Maschine



berechnet.

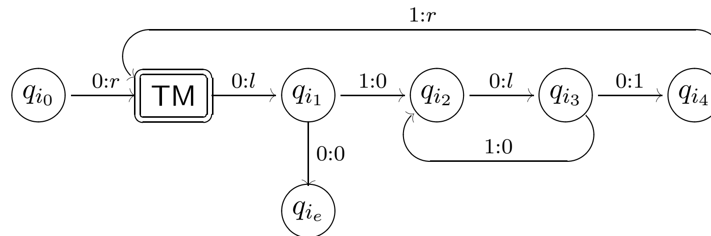
Als nächstes müssen wir zeigen, dass die Turing-Berechenbarkeit bei normierter Einsetzung erhalten bleibt. Nach Definition 3.3 erhalten wir die Funktion h durch normierte Einsetzung aus den Funktionen f und $g_1, \ldots, g_m$, wenn gilt $h(x_1, \ldots, x_n) = f(g_1(x_1, \ldots, x_n), \ldots, g_m(x_1, \ldots, x_n))$. Wir benötigen also eine Turingmaschine TM_0 zur Berechnung von f und m Turingmaschinen $\mathsf{TM}_i (i = 1, \ldots, m)$ zur Berechung der g_i. Sind diese vorhanden, so berechnet die folgende Maschine die Funktion h:



Bei der Berechung von h treten folgende Konfigurationen auf:

$$
\begin{array}{l}
0\,\overline{\mathfrak{x}}\,q_0 \xrightarrow{\mathsf{TM}_1} \\
0\,\overline{\mathfrak{x}\,g_1(\mathfrak{x})}\,q_{i_1} \xrightarrow{\mathsf{Cp}^{\mathsf{n}}_{\mathsf{n+1}}} \\
0\,\overline{\mathfrak{x}\,g_1(\mathfrak{x})\,\mathfrak{x}}\,q_{i_2} \xrightarrow{\mathsf{TM}_2} \\
0\,\overline{\mathfrak{x}\,g_1(\mathfrak{x})\,\mathfrak{x}\,g_2(\mathfrak{x})}\,q_{i_3} \xrightarrow{\mathsf{Cp}^{\mathsf{n}}_{\mathsf{n+1}}} \\
0\,\overline{\mathfrak{x}\,g_1(\mathfrak{x})\,\mathfrak{x}\,g_2(\mathfrak{x})\,\mathfrak{x}}\,q_{i_4} \xrightarrow{\mathsf{TM}_3} \\
\vdots \\
0\,\overline{\mathfrak{x}\,g_1(\mathfrak{x})\,\mathfrak{x}\,g_2(\mathfrak{x})\,\mathfrak{x}}\,\ldots\,g_{m-1}(\mathfrak{x})\,\mathfrak{x}\,g_m(\mathfrak{x})\,q_{i_5} \xrightarrow{\mathsf{Mv}^{\mathsf{n}}_{1}} \\
0\,\overline{\mathfrak{x}\,g_1(\mathfrak{x})\,\mathfrak{x}\,g_2(\mathfrak{x})\,\mathfrak{x}}\,\ldots\,\overline{g_{m-1}(\mathfrak{x})\,g_m(\mathfrak{x})}\,q_{i_6} \xrightarrow{\mathsf{Mv}^{\mathsf{n}}_{2}} \\
\vdots \\
0\,\overline{g_1(\mathfrak{x})\,g_2(\mathfrak{x})\,g_2(\mathfrak{x})}\,\ldots\,\overline{g_{m-1}(\mathfrak{x})\,g_m(\mathfrak{x})}\,q_{i_7} \xrightarrow{\mathsf{TM}_0} \\
0\,\overline{g_1(\mathfrak{x})\,g_2(\mathfrak{x})\,g_2(\mathfrak{x})}\,\ldots\,\overline{g_{m-1}(\mathfrak{x})\,g_m(\mathfrak{x})\,f(\mathfrak{x})}\,q_{i_8} \\
\xrightarrow{\mathsf{Mv}^{\mathsf{m}}} \\
0\,\overline{\mathfrak{x}\,f(\mathfrak{x})}\,q_{i_9}
\end{array}
$$

Es bleibt noch zu zeigen, dass der μ-Operator die Turing-Berechenbarkeit erhält. Es sei $f = \mu[g]$ und TM berechne die Funktion g normiert. Die folgende Maschine berechnet dann f normiert:

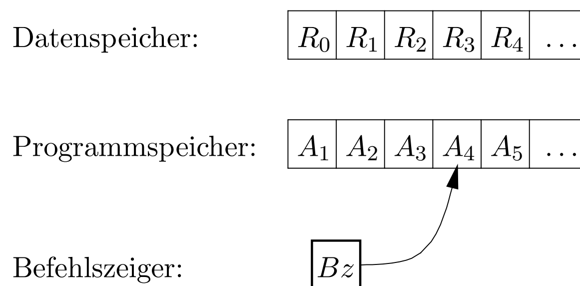


Der μ-Operator sucht das kleinste y, für das $g(\mathfrak{x}, y) = 0$ ist. Die Turingmaschine berechnet dazu die Funktionswerte von $g(\mathfrak{x}, 0)$, $g(\mathfrak{x}, 1)$, $g(\mathfrak{x}, 2), \ldots$ und stoppt, wenn sich der Wert 0 ergeben hat. Tritt 0 nicht als Funktionswert auf, läuft sie endlos weiter. $\square$

3.4 Registermaschinen

3.4.1 Registermaschinen-Berechenbarkeit

Registermaschinen sind ein weiteres Modell, um den Begriff der Berechenbarkeit zu erfassen. Sie sind um nichts weniger abstrakt als Turingmaschinen, entsprechen aber mehr den heute üblichen Digitalrechnern mit von-Neumann-Architektur. In der Literatur kommen sie in den verschiedensten Varianten vor. Ich werde hier eine Registermaschine vorstellen, die aus zwei unendlich großen Speichern besteht, von denen einer als Datenspeicher und der andere als Programmspeicher dient. Die Speicher sind in durchnummerierte Register eingeteilt. Jedes Register R_0, R_1, R_2, … des Datenspeichers kann eine beliebig große Zahlendarstellung aufnehmen. Jedes Register des Programmspeichers enthält eine Anweisung. Außerdem gibt es ein Sonderregister, das die Adresse der gerade aktuellen Anweisung enthält. Dieses Register wird auch Befehlszeiger genannt. Da alle Register der Maschine in beliebiger Reihenfolge angesprochen werden können, wird sie auch Random Access Machine genannt.



Für die Registermaschine definiere ich einen Satz von fünf Basisanweisungen. Es ist zwar möglich mit wenigeren Anweisungen auszukommen (Minsky 1967; Smith 1996), auf diese Weise entspricht unsere Maschine aber mehr den heute üblichen Digitalrechnern. Die Befehle unserer Registermaschine RM sind in Tabelle 3.3 aufgelistet (vergleiche Oberschelp 1993).

Anweisung	Wirkung auf Speicher	Wirkung auf Bz
inc(i)	Inkrementiere das Register R_i um 1	Bz wird um eins erhöht
dec(i)	Dekrementiere das Register R_i um 1	Bz wird um eins erhöht
test(i)	Register R_i wird auf 0 überprüft	Falls $R_i = 0$, wird Bz um 1, andernfalls um 2 inkrementiert.
label $\mathcal{L}$	keine	Bz wird um 1 inkrementiert
goto $\mathcal{L}$	keine	Bz wird auf die Adresse von $\mathcal{L}$ gesetzt

Tabelle 3.3: Registermaschinenbefehle

Syntax Wird die Registermaschine in Gang gesetzt, so führt sie die jeweils durch den Befehlszeiger bestimmte Anweisung aus. Dabei wird der Datenspeicher der Anweisung entsprechend verändert und der Befehlszeiger auf eine neue Adresse gesetzt. Diese Veränderung der Register als Wirkung der Anweisungen kann als die Semantik der Maschinen bezeichnet werden. Wir könnten nun den Programmspeicher mit beliebigen Anweisungen füllen und diese von der Maschine ausführen lassen. Werden aber bei der Erstellung von Programmen keine Regeln beachtet, so führt das in den meisten Fällen zu „unsinnigen" Programmen. Da dies meistens unerwünscht ist, lege ich eine Syntax für die Sprache der Registermaschinen fest. Dazu definiere ich zuerst die Bausteine der RM-Sprache.

Definition 3.35

(i) *Das* Alphabet *besteht aus den Buchstaben* ‘***a***’, . . . ,‘***z***’, *den Ziffern* ‘***0***’, . . . ,‘***9***’, *und den Sonderzeichen* ‘***(***’,‘***)***’,‘***;***’ *und* ‘ ’.

(ii) *Die* reservierten Wörter *sind* ***inc***, ***dec***, ***test***, ***label*** *und* ***goto***.

(iii) Bezeichner *sind Zeichenfolgen aus Buchstaben und Ziffern, die mit einem Buchstaben beginnen.*

(iv) Variablen *und* Marken *sind beliebige Bezeichner.*

(v) *Wenn* $\mathcal{A}$ *eine Variable und* $\mathfrak{L}$ *eine Marke ist, so sind* ***inc(***$\mathcal{A}$***)*** , ***dec(***$\mathcal{A}$***)*** , ***test(***$\mathcal{A}$***)*** , ***label*** $\mathfrak{L}$ *und* ***goto*** $\mathfrak{L}$ Anweisungen.

Variablen müssen vor der Ausführung des Programmes durch Adressen im Datenspeicher und Marken durch Adressen im Befehlsspeicher ersetzt werden. Diese Aufgabe kann von einem automatischen Übersetzer übernommen werden. Die Sprachdefinition wird nun mit der Definition von RM-Programmen vervollständigt.

Definition 3.36 (RM-Programm)

Sei $\mathcal{A}$ *eine Variable,* $\mathcal{L}_1$ *und* $\mathcal{L}_2$ *Marken.*

(i) *Die leere Zeichenkette* ϵ, *sowie die Anweisungen* ***inc(***$\mathcal{A}$***)***, ***dec(***$\mathcal{A}$***)*** *sind RM-Programme.*

(ii) *Falls* Γ *und* Δ *RM-Programme sind, dann auch* Γ ***;*** Δ.

(iii) *Wenn* Ψ *ein RM-Programm ist, dann auch die Zeichenkette* ***label*** $\mathcal{L}_1$ ***;*** ***test*** $\mathcal{A}$ ***;*** ***goto*** $\mathcal{L}_2$ ***;*** Ψ ***;*** ***goto*** $\mathcal{L}_1$ ***;*** ***label*** $\mathcal{L}_2$.

Die Trennung von zwei Anweisungen durch ein Semikolon soll der leichteren Lesbarkeit dienen. Ebenso die Schreibweise $\mathcal{A}$>***0*** für die Anweisung ***test(***$\mathcal{A}$***)*** . Weiters ist dann die folgende alternative Schreibweise des in der letzten Definition angegebenen Programms sehr hilfreich:

Definition 3.37 (*while*-Schleife)

while $\mathcal{A}$>***0*** ***do*** Ψ ***end*** $\equiv$ ***label*** $\mathcal{L}_1$ ***;*** ***test*** $\mathcal{A}$ ***;*** ***goto*** $\mathcal{L}_2$ ***;*** Ψ ***;*** ***goto*** $\mathcal{L}_1$ ***;*** ***label*** $\mathcal{L}_2$.

Der links vom Zeichen ‘$\equiv$’ stehende Text soll dabei beim Laden des Programmes durch den rechts davon stehenden ersetzt werden. Dabei muss dafür gesorgt werden, dass die Marken eindeutig sind, also nicht mit Marken in anderen Programmteilen kollidieren.

Beispiel Zur Veranschaulichung ein Beispiel, bei dem das Programm Ψ aus der Anweisung die Variable a zu dekrementieren, besteht: ***while*** a>***0*** ***do*** ***dec(***a***)*** ***end*** .

Laut Definition 3.36.*(ii)* ist dann auch ***while** a>**0** **do dec(**a**)** **end*** ; ***inc(**a**)*** ein Programm.

Semantik Die Bedeutung eines Registermaschinenprogramms besteht in der Wirkung auf den Zustand der Maschine. Ein Zustand ist dabei gegeben durch die Gesamtheit der Registerinhalte. Wenn x_i den Inhalt des Registers R_i bezeichnet, so entspricht ein Zustand der Abbildung $v(i) = x_i$. Ein Zustand kann auch als Tupel $v = \langle x_1, x_2, x_3, \dots \rangle$ angegeben werden. Bei der Abarbeitung eines Programmes ändert sich also der jeweilige Zustand und der Inhalt des Befehlszeigers. Da der Befehlszeiger bestimmt, welcher Programmteil noch auszuführen bleibt, kann statt dem Wert des Befehlsregisters auch das noch auszuführende Restprogramm angegeben werden.

Definition 3.38 (Ausführung eines RM-Programms)
Eine Ausführung eines Registermaschinenprogramms Ψ ist eine alternierende Folge $v_0\, \Psi_1\, v_1 \dots v_{n-1}\, \Psi_n\, v_n\, \Psi_{n+1} \dots$ von Zuständen und Programmen, sodass $\Psi_1 = \Psi$ und für $n \geq 1$ gilt, der Definition 3.36 entsprechend:

(i) *Falls $\Psi_n = \epsilon$,* terminiert *die Ausführung im Endzustand v_{n-1}.*

(ii) *Falls $\Psi_n =$ **inc(**i**)** ; Γ, wird der Inhalt von R_i inkrementiert und dann Γ ausgeführt:*

$$v_n(j) = \begin{cases} v_{n-1}(j) + 1, & \textit{falls } i = j \\ v_{n-1}(j), & \textit{sonst} \end{cases} \quad \textit{und } \Psi_{n+1} = \Gamma.$$

(iii) *Falls $\Psi_n =$ **dec(**i**)** ; Γ, wird der Inhalt von R_i dekrementiert und dann Γ ausgeführt:*

$$v_n(j) = \begin{cases} v_{n-1}(j) - 1, & \textit{falls } i = j \textit{ und } v_{n-1}(i) \neq 0 \\ v_{n-1}(j), & \textit{sonst} \end{cases} \quad \textit{und } \Psi_{n+1} = \Gamma.$$

(iv) *Falls $\Psi_n =$ **while** i>**0** **do** Σ **end** ; Γ:*

$$v_n = v_{n-1} \textit{ und } \Psi_{n+1} = \begin{cases} \Sigma; \Psi_n, & \textit{falls } v_{n-1}(i) \neq 0 \\ \Gamma, & \textit{sonst} \end{cases}$$

Mit Registermaschinen können nun partielle n-stellige Funktionen berechnet werden. Dazu setzen wir zuerst alle Register auf 0, laden das Programm in den Befehlsspeicher und laden die Argumente der zu berechnenden Funktion in die ersten n Datenregister $R_1, \dots, R_n$. Nach der Programmausführung soll sich das Ergebnis der Berechnung im Register R_1 befinden.

Definition 3.39 *Sie Ψ ein Registermaschinenprogramm. Für jedes $n \geq 0$ ist die durch Ψ berechnete n-stellige partielle Funktion f^n_Ψ definiert durch:*

$$f^n_\Psi(x_1, \dots, x_n) := v(1),$$

falls die Ausführung von Ψ mit Anfangszustand $\langle x_1, \dots, x_n, 0, 0, \dots \rangle$ im Endzustand v terminiert.

Beispiel Wird das bereits im letzten Beispiel vorgestellte Programm

$$\Sigma = \textbf{\textit{while}}\ a{>}\textbf{\textit{0}}\ \ \textbf{\textit{do dec(}}a\textbf{\textit{)}}\ \ \textbf{\textit{end}}$$

in eine Maschine mit dem Anfangszustand $v_0 = \langle 1, 0, 0, \dots \rangle$ geladen und wird dabei die Variable a durch die Adresse 1 ersetzt, so ergibt sich nach Definition 3.38 folgende Ausführung:

$$\begin{aligned} v_0 &= \langle 1,0,0,\dots\rangle & \Psi_1 &= \Sigma \\ v_1 = v_0 &= \langle 1,0,0,\dots\rangle & \Psi_2 &= \textbf{\textit{dec(1)}}\ \ ;\Psi \\ v_2 &= \langle 0,0,0,\dots\rangle & \Psi_3 &= \Psi \\ v_3 = v_2 &= \langle 0,0,0,\dots\rangle & \Psi_4 &= \epsilon. \end{aligned}$$

Definition 3.40 (RM-Berechenbarkeit)
Eine partielle Funktion f heißt RM-berechenbar, falls ein Registermaschinenprogramm Ψ existiert, sodass $f = f^n_\Psi$.

Makros und Prozeduren Da es für die Berechnung nicht wichtig ist, auf welchen Registern sie durchgeführt wird, solange sich die Register nicht überschneiden, können wir in Programmen *Variablennamen* statt Registeradressen verwenden. Dabei dienen als Varablennamen beliebige Bezeichner, die vor der Programmausführung wieder durch Registeradressen ersetzt werden. Auf ähnliche Weise können wir ganze Zeichenketten durch andere ersetzen, wie wir in Definition 3.37 für die ***while***-Schleife gesehen haben. Solche abgekürzte Schreibweisen für Anweisungsfolgen heißen allgemein *Makros*. Sie müssen beim Laden eines Programmes in die Registermaschine, wieder auf die ursprünglichen Anweisungsfolgen *expandiert* werden. Unser erster Makro ist eine Abkürzung für Programme, durch die Variablen Werte zugewiesen werden.

Makro 3.1 (Zuweisung)

(i) $x \Leftarrow \mathfrak{k} \ \equiv\ $ ***while*** $x{>}$***0*** ***do dec(****x****)*** ***end ;***
$\underbrace{\textbf{\textit{inc(}}x\textbf{\textit{)}}\ ;\ \dots\ ;\textbf{\textit{inc(}}x\textbf{\textit{)}}}_{\mathfrak{k}-mal}$

(ii) $y \Leftarrow x \ \equiv\ y \Leftarrow$ ***0*** ***;*** $u \Leftarrow$ ***0*** ***;***
while $x{>}$***0*** ***do inc(****u****)*** ***; dec(****x****)*** ***end ;***
while $u{>}$***0*** ***do inc(****x****)*** ***; inc(****y****)*** ***; dec(****u****)*** ***end***

falls x und y verschiedene Register bezeichnen. Falls sie dasselbe Register bezeichnen, wird der Makro $y \Leftarrow x$ zu ϵ expandiert.

Daraus ergibt sich unmittelbar, dass die Nullfunktion und Projektionsfunktion RM-berechenbar sind.

Als nächstes stelle ich die Translationsmakros vor, mit deren Hilfe eine Funktion f^n_Ψ auf beliebigen Registern berechnet werden kann. Wenn Ψ die Funktion $f(x_1, \dots, x_n)$ normiert berechnet, so werden dabei die Argumente $x_1, \dots, x_n$ ja zuerst in die Eingaberegister $R_1, \dots, R_n$ geladen. Das Resultat steht nach der Berechnung in R_1. Um die Berechnung auf anderen Registern durchzuführen,

müssen alle in Ψ vorkommenden Register R_i durch davon verschiedene Register U_i ersetzt werden. Das durch diese Ersetzung entstehende Programm Ψ' berechnet f auf den Eingaberegistern $U_1, \ldots, U_n$, die wir auf die Werte $x_1, \ldots, x_n$ setzen. Das in U_1 stehende Ergebnis können wir anschließend einem Ausgaberegister z zuweisen. Wenn wir dem so entstehenden Programm mit den Eingaberegistern $x_1, \ldots, x_n$ einen beliebigen Bezeichner $\mathcal{F}$ als Namen geben, können wir den folgenden, an die übliche Funktionsschreibweise in Programmiersprachen angelehnten, Translationsmakro definieren:

Makro 3.2 (Translation)
$z \Leftarrow \mathcal{F}(x_1, \ldots, x_n) \equiv U_1 \Leftarrow x_1$; ... ; $U_n \Leftarrow x_n$; Ψ' ; $z \Leftarrow U_1$

Wenn streng darauf geachtet wird, dass alle Bezeichner voneinander verschiedene Adressen bedeuten, können Programmteile ohne negative Nebeneffekte zusammengefügt werden. Der Translationsmakro ermöglicht dabei, dass Programme auf beliebigen Registern berechnet werden können. Bereits vorhandene Programme können also zur Definition neuer Makros herangezogen werden.

Lemma 3.28 *Die Addition $x + y$ ist RM-berechenbar.*

Beweis $\Psi =$ ***while test(2) do inc(1) ; dec(2) end***. □
Die Addition f^2_Ψ können wir mit Hilfe des Makros 3.2 als *z⇐Addiere(x, y)* schreiben.

```
z⇐Addiere(x, y) ≡ u⇐x ; v⇐y ;
                  while v>0 do
                      inc(u) ; dec(v)
                  end ;
                  z⇐u
```

Die Nachfolgerfunktion ist dann mit *x⇐Addiere(x, 1)* RM-berechenbar.

Lemma 3.29 *Die Multiplikation $x \cdot y$ ist RM-berechenbar.*

Beweis
```
w⇐0 ; u⇐x ; v⇐y
while u>0 do
    w⇐Addiere(w, v) ; dec(u)
end ;
z⇐w
```
□

Lemma 3.30 *Die modifizierte Differenz ist RM-berechenbar.*

Beweis
```
u⇐x ; v⇐y ;
while v>0 do
    dec(u) ; dec(v)
end ;
z⇐u
```
□

Dafür kann wieder mit Hilfe des Makros 3.2 *z⇐Minus(x, y)* geschrieben werden.

Lemma 3.31 (Normierte Einsetzung)
Ist $h = \mathsf{NE}[f, g_1, \ldots, g_m]$ und sind die Funktionen f und $g_1, \ldots, g_m$ RM-berechenbar, so ist auch $h = h^n_\Psi$ RM-berechenbar.

Beweis $\Psi = U_1 \Leftarrow \mathcal{G}(y_1,\ldots,y_n)$; ... ; $U_m \Leftarrow \mathcal{G}(y_1,\ldots,y_n)$; $z \Leftarrow \mathcal{F}(U_1,\ldots,U_m)$ □

Lemma 3.32 (μ-Operator)
Ist $f = \mu[g]$ und ist g RM-berechenbar, so ist auch f RM-berechenbar.

Beweis

```
y⇐0 ; v⇐1 ;
while v>0 do
    v⇐ G(x1, ..., xk, y) ; inc(y)
end ;
z⇐Minus(y, 1)
```

□

Aus dem bisher Gesagten und Definition 3.17 auf Seite 52 ergibt sich durch Induktion über den Aufbau unmittelbar:

Theorem 3.33 *Alle μ-rekursiven Funktionen sind RM-berechenbar.*

3.4.2 Das Selbstanwendungsproblem

Ich greife nun das Thema der Selbstbezüglichkeit wieder auf und zeige eine wichtige Grenze der Leistungsfähigkeit von Registermaschinen. Nach der Church-Turing-These wird damit allgemein der Begriff der Berechenbarkeit abgegrenzt. Das Resultat wird durch die Anwendung eines Registermaschinenprogramms auf sich selbst erreicht. Damit dies möglich ist, müssen wir zuerst RM-Programme als Zahlen darstellen können, da Registermaschinen ja nur mit Zahlen operieren. Wir haben bereits in Definition 3.16 eine Möglichkeit gesehen, wie die Gödelnummer von Zahlenfolgen berechnet werden kann. Da die RM-Programme aus Folgen von Zeichen bestehen, müssen wir nur jedem Zeichen des Alphabets eine Zahl zuordnen, um anschließend die Gödelnummer der so erhaltenen Zahlenfolge bilden zu können. Die einem Zeichen zugeordnete Zahl können wir ebenfalls Gödelnummer nennen. Eine Zuordnung kann zum Beispiel so aussehen:

a	***b***	***c***	...	***z***	***0***	***1***	...	***9***	***(***	***)***	***;***	***‘ ’***
1	2	3	...	26	27	28	...	36	37	38	39	40

Die Gödelnummer eines Symbols σ wird mit $\ulcorner\sigma\urcorner$ bezeichnet und die Gödelnummer einer Zeichenkette $\langle\sigma_0,\sigma_1,\ldots,\sigma_{n-1}\rangle$ wird gebildet als

$$\ulcorner\langle\ulcorner\sigma_0\urcorner,\ulcorner\sigma_1\urcorner,\ldots,\ulcorner\sigma_{n-1}\urcorner\rangle\urcorner.$$

Wir können nun versuchen ein Registermaschinenprogramm Σ zu schreiben, das untersucht, ob ein Programm Ψ für eine bestimmte Eingabe terminiert. Das Programm soll als Ergebnis den Wert 1 liefern, falls das zu untersuchende Programm Ψ für eine bestimmte Eingabe ψ terminiert, sonst soll es den Wert 0 als Resultat haben. Σ berechnet also eine Funktion s, die definiert ist als

$$s(\psi) := \begin{cases} 1, & \text{falls } \psi = \ulcorner\Psi\urcorner \land f_\Psi(\psi) \text{ ist definiert} \\ 0, & \text{sonst} \end{cases} \tag{3.1}$$

Der „Trick“ besteht also darin, ein Programm zu untersuchen, dem *seine eigene Gödelnummer* als Argument übergeben wurde. Ich zeige also das Analogon des unmöglichen Oberon-Programms für Registermaschinen. Natürlich ergibt sich auch hier:

Theorem 3.34
Die Funktion s ist nicht Registermaschinen-berechenbar.

Beweis Nehmen wir an, dass s doch RM-berechenbar sei, also $s = s^1_\Sigma$. Dann ist

$$\overline{s}(p) := \begin{cases} 1, & \text{falls } s(p) = 0 \\ \text{undefiniert}, & \text{sonst} \end{cases} \tag{3.2}$$

ebenfalls durch ein Programm $\overline{\Sigma}$ berechenbar. Zum Beispiel durch das Programm:

```
Σ̄ = x⇐s(p) ; y⇐1 ;
      while x>0  do
          while y>0  do
              inc(y)
          end
      end ;
      z⇐1
```

Falls p nun die Gödelnummer des Programms $\overline{\Sigma}$ ist, also $p = \ulcorner\overline{\Sigma}\urcorner$, gilt:

$$\overline{s}(p) = 1 \leftrightarrow \overline{s}(p) \text{ ist nicht definiert.}$$

Nach der Definition von $\overline{s}$ ist $\overline{s}(p) = 1$, wenn $s(p) = 0$. Letzteres ist aber nach 3.1 dann der Fall, wenn $f_{\overline{\Sigma}}$ undefiniert ist, da p die Gödelnummer des Programms $\overline{\Sigma}$ ist. Wieder nach Definition 3.1 ist aber $f_{\overline{\Sigma}}$ gerade die Funktion $\overline{s}$ und damit ist $\overline{s}(p)$ undefiniert.

Aus diesem Widerspruch schließen wir, dass es p und damit auch $\overline{\Sigma}$ gar nicht geben kann. Wenn wir uns aber das Beispielprogramm $\overline{\Sigma}$ ansehen, so kann das nur daran liegen, dass s nicht berechenbar ist. □

4 Rekursive Algorithmen und Datenstrukturen

Nachdem ich nun die prinzipielle Funktionsweise von abstrakten Registermaschinen vorgestellt habe, werde ich auf heute üblichen konkreten Registermaschinen ausführbare Formalismen auf rekursive Phänomene hin untersuchen. Ich beschränke mich dabei auf Formalismen, die als prozedurale Programmiersprachen klassifiziert werden. Die prominentesten Sprachen dieser Familie sind wohl C++ und Pascal. Zur Formulierung der Beispiele verwende ich hier die Sprache Oberon, die sich in den verwendeten Sprachkonstrukten nur wenig von Pascal unterscheidet.

4.1 Sich selbst aufrufen

Wie bereits auf Seite 2 erwähnt, werden in der Literatur Prozeduren meist dann als rekursiv bezeichnet, wenn sie sich selbst aufrufen: „A function that calls itself is said to be *recursive*.“ (Wand 1980) Das Kriterium „selbstaufrufend“ reicht nicht aus, um Rekursion von Iteration zu unterscheiden. Zuerst einmal bedeutet es ja nur, dass innerhalb eines bestimmten Programmabschnittes die Abarbeitung desselben Programmabschnitt veranlasst wird.

Auch die für Registermaschinen definierten ***while***-Schleifen sind selbstaufrufend. Wenn wir Definition 3.37 betrachten, so ist klar zu erkennen, dass nach dem wiederholt auszuführenden Programm Ψ mit der Anweisung ***goto*** $\mathcal{L}_1$ der Befehlszeiger an den Anfang dieses Programmabschnittes gesetzt wird. Auch wenn dies in der Literatur nicht so formuliert wird, ist klar ersichtlich: das Programm *ruft sich selbst auf*, wobei zu untersuchen wäre, ob es sich nicht hier nur um eine Metapher handelt. Prozeduren sind im wesentlichen nur mit Namen versehene Programmabschnitte. „A *procedure definition* is a declaration that, in its simplest form associates an identifier with a statement. The identifier is the *procedure name*, and the statement is the *procedure body*.“ (Aho et al. 1986, p.389) Da auch ein „Selbstaufruf“ von Prozeduren durch entsprechendes Setzen des Befehlszeigers erreicht wird, muss diese Eigenschaft auch while-Schleifen zugestanden werden. While-Schleifen werden nicht nur in meiner abstrakten RM-Maschine über Sprungbefehle und Marken realisiert. Das ist, wie zum Beispiel Holub (1990, p.643) oder Aho et al. (1986, p.493) vorführen, die unumgängliche Technik bei der Erstellung von Maschinencode.

Der wichtige Unterschied zwischen Prozeduren und while-Schleifen liegt darin, dass bei einem Prozeduraufruf die Abarbeitung eines Teilprogrammes unterbrochen wird, um später möglicherweise an derselben Stelle wieder fortge-

setzt zu werden. Dazu muss nicht nur die Rücksprungadresse gemerkt werden, sondern es müssen auch die aktuellen Parameter und die Werte der eventuell vorhandenen lokalen Variablen des Prozeduraufrufs protokolliert werden. Geschieht dies nicht, so wären Selbstaufrufe sinnlos, da die jeweiligen Werte verlorengehen würden. Diese Protokollierung ermöglicht also, dass mehrere Instanzen derselben Prozedur gleichzeitig im Speicher liegen. Damit ist natürlich nicht gemeint, dass die Anweisungen des Prozedurkörpers mehrfach vorhanden sind. Gemeint ist, dass die zu protokollierende Information mithilfe eines *Activation Records* gemerkt wird. Eine zur Speicherung von Activation Records und damit von Prozeduraufrufen hervorragend geeignete Datenstruktur ist das *push-down stack*. Solche Stacks besitzen eine last-in first-out (LIFO) Organisation. Bildlich gesprochen ist es nur erlaubt, ein Element oben auf das Stack zu legen, oder das zuletzt darauf gelegte Element zu entfernen. Bei jeder Aktivierung einer Prozedur wird also ein Activation Record auf das Stack gelegt und beim Verlassen einer Prozedur wieder von dort geholt. Damit sind geordnete und sinnvolle Selbstaufrufe möglich. While-Schleifen dagegen instantiieren sich nicht gleichzeitig. Jeder Schleifendurchlauf wird beendet, bevor der nächste begonnen wird. Eine der wenigen Textstellen, an der die Rekursivität von Prozeduren nicht über Selbstaufrufe definiert wird, ist: „A procedure is *recursive* if a new activation can begin before an earlier activation of the same procedure has ended.“ (Aho, Sethi und Ullman 1986, p.493)

Ein klassisches Beispiel für eine rekursive formulierte Prozedur ist die Berechnung der Faktoriellen einer Zahl, deren Rekursionsgleichungen wie folgt lauten:

$$0! = 1$$
$$n! = n \cdot (n-1)!$$

Das entsprechende Oberon-Programm lässt sich so formulieren:

```
PROCEDURE factorial(n: INTEGER) : LONGINT ;
BEGIN
    IF n = 0 THEN
        RETURN 1
    ELSE
        RETURN n * factorial(n - 1)
END factorial ;
```

Dieses Programm ist elegant für abstrakte Registermaschinen. Für konkrete empfiehlt sich die iterative Version:

```
factorial := 1
WHILE n > 0 DO
    factorial := factorial * n ;
    n := n - 1
END ;
```

4.2 Sich selbst enthalten

Unter einem *Datentyp* wird in der Informatik die Menge von Werten verstanden, die eine Variable annehmen kann oder denen eine Konstante oder das Resultat

einer Prozedur angehört. Es wird zwischen einfachen und strukturierten Datentypen unterschieden. Variable mit einfachen Datentypen enthalten zu einem Zeitpunkt immer genau einen Wert. Variable, die einen strukturierten Datentyp besitzen, können mehrere Werte eines anderen Typs gleichzeitig aufnehmen. Handelt es sich dabei um gleichartige Typen, so wird die Datenstruktur meist *Array* genannt. Sind Typen aus unterschiedlichen anderen Typen zusammengesetzt, so heißen die Datenstrukturen meist *Record* oder *Structure*. Der Begriff Record wird im Deutschen manchmal mit dem Wort Verbund bezeichnet. Die in einen Record zusammengefassten Typen werden Felder oder Komponenten genannt. In objektorientierten Programmiersprachen, in denen es keine Records gibt, übernehmen *Klassen* deren Funktion. Ein Recordtyp entspricht einer Tabellendefinition in einer relationalen Datenbank. Eine Zeile in einer Tabelle entspricht einer Variable vom Typ Record und eine Spalte entspricht einem Recordfeld. Die entsprechenden formalen Bezeichnungen aus dem relationalen Modell lauten Relation für Tabelle, Tupel für Zeile und Attribut für Spalte (Date 1990, p.250). Die Entsprechungen in objektorientierter Terminologie lauten Klasse, Objekt und Member, wobei diese natürlich begriffliche Erweiterungen der prozeduralen Gegenstücke sind.

Da Werte immer als Bitfolgen dargestellt werden, handelt es sich bei Typen um unterschiedliche Interpretationen von Speicherinhalten. Erschwert eine Sprache verschiedene Interpretationen, so wird sie als streng typisiert bezeichnet. Datentypen prägen also den Bitfolgen im Speicher eine abstrakte Struktur auf: „Für die Rechenanlage ist der Speicher eine homogene Masse von Bits ohne sichtbare Struktur. Es ist jedoch einzig und allein diese abstrakte Struktur, die es dem menschlichen Programmierer ermöglicht, einen Sinn in der monotonen Welt eines Computerspeichers zu finden." (Wirth 1983) Ein wichtiges Mittel bei dieser Sinnsuche ist der Entwurf und das Finden von Programm- und Datenstrukturen. Bei diesem Versuch, Ordnung in die Welt zu bringen, meint Niklaus Wirth, durch eine Gestaltanalyse, Analogien zwischen Programm- und Datenstrukturen entdeckt zu haben. Als Gestalttypen listet er unter anderen die „Wiederholung um eine unbekannte Anzahl" und die Rekursion auf. Zum ersten Typ gehören die while-Schleife und sequentielle Datentypen, zum zweiten die Prozeduren und rekursive Datentypen (Wirth 1983, p.197).

Es ist wichtig zwischen Datentypen, die im Programmtext festgelegt werden, und Datenstrukturen, die während der Ausführung des Programmes erzeugt werden, zu unterscheiden. Wenn ich hier über Eigenschaften von Typdefinitionen spreche, so sind damit syntaktische Merkmale der Programmiersprache gemeint. Aussagen über Eigenschaften von Datenstrukturen und Variablen dagegen sind Zuschreibungen von Merkmalen an den Speicherinhalt einer Registermaschine während der Laufzeit. Der Speicherinhalt wird dabei zuerst auf eine bestimmte Weise interpretiert. Wir sagen, dass dieses oder jenes Programm läuft. Variablen, die während der gesamten Laufzeit eines Programmes bestehen, heißen statisch. Dynamische Variablen dagegen werden während der Ausführung erzeugt und zerstört. Dynamisch werden aber auch Daten*strukturen* genannt, die vom Programm während der Laufzeit angelegt und verändert werden. Diese Unterscheidung genauer zu treffen, ist für die weitere Untersuchung jedoch nicht notwendig, wenn wir vereinbaren, eine Variable mit Recordtyp statisch zu nennen. Dafür spricht, dass Records ihre Struktur nicht verändern.

Eine typische Typdefinition sieht wie folgt aus:

```
TYPE genealogy =  RECORD
                      name: String;
                      mother, father: String
                  END
```

Dabei nehme ich an, dass bereits ein Typ String zur Speicherung von Zeichenketten definiert ist.

Es könnten nun rekursive Typdefinitionen zugelassen werden. Bei solchen wird als Typ eines Recordfelds der Typ des Records selbst festgelegt:

```
TYPE genealogy =  RECORD
                      mother, father: genealogy
                  END
```

Diese Definition ist offensichtlich zirkulär und damit unsinnig. Ein Verbot solcher Typdefinitionen wird meist als Anmerkung zur Sprachdefinition ausgesprochen, denn die formale Spezifikation der Syntax erfolgt oft in Metasprachen – wie dem Backus-Naur-Formalismus –, die eine Obermenge der legalen Programme einer Sprache festlegen, und in denen rekursive Typdeklarationen den Normalfall darstellen. Diese Eigenschaft ist bereits im von Emil Post entwickelten System der Post-Produktionen angelegt, aus dem der Backus-Naur-Formalismus entwickelt wurde (Wexelblat 1981, p.162). Das heisst also, dass zum Beispiel in Oberon die Typdeklaration

R = RECORD x: R END;

verboten ist, obwohl sie laut Syntaxdefinition korrekt ist. Wäre sie erlaubt, so würde sie zu einer endlosen Verschachtelung von Records vom Typ *R* führen und außerdem kein Feld enthalten, in dem etwas gespeichert werden könnte. Wir benötigen demnach einen Mechanismus, um die Rekursion terminieren zu lassen. Die Sprache Pascal stellt diesen in Form von *varianten Records* zur Verfügung. In solchen wird ein Feld des Records dafür verwendet zu entscheiden, welche zusätzlichen Felder in einer Variable dieses Record-Typs tatsächlich vorhanden sein sollen. Damit ist ein Abbruch der Verschachtelung möglich. Variantenrecords werden meist zu den statischen Datentypen gezählt, da sich die Anzahl der möglichen Felder, also ihre „innere" Struktur, während der Laufzeit nicht verändern kann. Unterschiedlich ist jedoch, welche Felder bei einer Variable vom Typ Variantenrecord jeweils aktualisiert sind und damit wieviel Speicherbereich sie während der Laufzeit einnehmen. Sie können deshalb auch zu den dynamischen Datenstrukturen gezählt werden, was vor allem Sinn macht, wenn rekursive Typdefinitionen erlaubt sind.

Mithilfe eines Variantenrecords könnten wir Genealogien zum Beispiel auf folgende Weise speichern:

```
TYPE genealogy =  RECORD
                      CASE known: BOOLEAN OF
                          TRUE:   (name: String; mother, father: genealogy) ;
                          FALSE:  ()
                  END
```

Eine Variable vom Typ *genealogy* hat also eine geschachtelte Struktur, bei der auf jeder Schachtelungsebene Variationen einer Datenstruktur auftreten, die entweder einen Namen und zwei Genealogien aufnimmt oder leer ist. Es entsteht also eine endliche Schachtelung von im wesentlichen gleichen Datenstrukturen, die sich durch den gespeicherten Namen und die in ihnen enthaltenen Genealogien unterscheiden. Mit einem Namen sind immer auch zwei weitere Genealogien gespeichert, die allerdings leer sein können. Ist dies der Fall, kommt die Verschachtelung zu einem Ende. Evolutionäre Variationen können so nicht erfasst werden, wohl aber biblische Genealogien, auch wenn diese meist etwas asymmetrisch ausfallen:

(Mehujaël, (Irad, (Henoch, (Kain, (Adam, (), ()), (Eva, (), ())), ()), ()))

Die Genealogie Kains enthält diejenigen von Eva und Adam. Henochs Mutter ist unbekannt, müsste aber eine Schwester seines Vaters oder seine Großmutter Eva sein.

Obwohl also rekursive Typdefinitionen als syntaktisch einfache Erweiterung der erlaubten Typdefinitionen für Records gesehen werden können, sind sie in den meisten prozeduralen Sprachen verboten, da sie sich als nicht sehr nützlich erwiesen haben und außerdem auf den gängigen Rechnerarchitekturen schwierig zu implementieren sind (Meyer 1990). Der erlaubte Weg zu rekursiven Datenstrukturen führt über dynamische Datenstrukturen, denen der Speicher erst zur Laufzeit zugeordnet wird. Dynamische Datenstrukturen bestehen aus Komponenten, die durch Verweise aufeinander zusammengehalten werden, und die während der Abarbeitung eines Programmes erzeugt und zerstört werden. Die Komponenten solcher Strukturen werden meist Knoten genannt, die Verweise oder Adressen von Knoten Zeiger. Knoten sind meist Variablen mit Recordtyp und Zeiger sind Variablen, die die Adresse von anderen Variablen enthalten. Zeigervariablen haben also den besonderen Typ Zeiger. Sie enthalten einen speziell dafür reservierten Wert, wenn sie auf keinen Knoten verweisen. Meist wird dafür die Adresse 0 verwendet. Mit Zeiger als Teil eines Knotens können auf diese Weise beliebig verkettete Strukturen erzeugt werden. Sie werden üblicherweise als Rechtecke, die in Felder geteilt und mit Pfeilen verbunden sind, dargestellt.

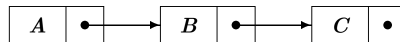


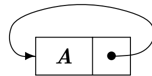
Abbildung 4.1: Verkettete Liste

Die Grafik zeigt drei Variable von Typ *Node*, die zu einer Liste verkettet wurden. Die Variable, deren Feld *data* den Wert A enthält, hat im Feld *next* die Adresse der Variable gespeichert, die im Feld *data* den Wert B enthält. Die Variable mit dem Wert C im *data*-Feld enthält keine Adresse. Die entsprechende Typdefinition in Oberon lautet:

```
TYPE
   NodePointer = POINTER TO Node ;
   Node =    RECORD
                data: String ; next : NodePointer
             END ;
```

Wird eine Variable vom Typ *NodePointer* definiert, so kann sie die Adresse eines Records vom Typ *Node* aufnehmen. Ein Record vom Typ *Node* enthält die Felder *data* und *next*. Da das Feld *next* wieder vom Typ NodePointer ist, können Variablen vom Typ *Node* Verweise auf sich selbst oder anderen Variablen gleichen Typs enthalten. Es ist jedoch nicht unmittelbar einsichtig, warum Typdeklaration in der obigen Form als selbstreferentiell oder „offensichtlich rekursiv“ (Reiser und Wirth 1992) bezeichnet werden sollen. Wäre sie das, dann müsste sie ja wohl zirkulär sein. Der Witz ist, dass wir nichts über den Typ eines Objekts im Speicher wissen müssen, um einen Zeiger darauf definieren zu können. Alle Adressen sind gleich, egal was sich an ihnen befindet.

Werden mit der obigen Typdefinition Datenstrukturen im Speicher erzeugt, so lässt sich schon sinnvoller darüber diskutieren, ob sie rekursiv oder selbstreferentiell sind. Doch auch hier ist Selbstreferentialität wohl nur in dem besonderen Fall gegeben, in welchem eine Variable vom Typ *Node* ihre eigene Adresse im Feld *next* enthält.



Enhält sie die Adresse einer anderen Variable vom gleichen Typ, so kann die Datenstruktur als rekursiv bezeichnet werden, wenn sie eine bestimmte Gestalt hat – wenn also unter Rekursivität eine Art Schachtelung verstanden wird – oder wenn sie induktiv definiert wird. Für Listen gilt offensichtlich letzteres, da jede Liste aus einem Listenkopf und einer Restliste besteht. Da die Restliste auch eine Liste ist, gilt für sie dasselbe. Die Schachtelung bricht ab, wenn die Restliste leer ist. Listen werden also als rekursive Strukturen aufgefasst, da sie auf einfache Weise induktiv definiert werden können. Die Programmiersprache LISP (List Processor) wurde speziell zur Verarbeitung solcher Listen von der Artificial Intelligence Group am M.I.T. entworfen. Hierzu wurde eine Klasse von S-Ausdrücken (symbolic expressions) definiert, die mithilfe der Sonderzeichen ‘$\cdot$’, ‘***(***’, ‘***)***’ und einer unendlichen Menge von atomaren Symbolen gebildet werden. S-Ausdrücke sind nun wie folgt induktiv definiert (McCarthy 1960, p.187):

Definition 4.1 (S-expressions)

(i) *Atomic symbols are S-expressions.*

(ii) *If* e_1 *and* e_2 *are S-expression, so is* $(e_1 \cdot e_2)$.

Beispiele ***AB***, ***(A · B)***, ***((AB · C) · D)***

In der Beschreibung „An S-expression is then simply an ordered pair, the terms of which may be atomic symbols or simpler S-expressions“ kommt die Rekursivität der Struktur gut zum Ausdruck. Eine Liste atomarer Symbole $\langle\lambda_1, \lambda_2, \ldots, \lambda_n\rangle$ kann nun mit dem S-Ausdruck $(\lambda_1 \cdot (\lambda_2 \cdot (\ldots (\lambda_n \cdot \textbf{\textit{NIL}}) \ldots)))$ dargestellt werden. Der Bezeichner ***NIL*** wird als reserviertes Symbol zur Kennzeichnung des Listenendes eingeführt. Die S-Ausdrücke können auf unterschiedliche Weise im Speicher realisiert werden. Die Liste in Abbildung 4.1 kann allerdings nicht als Darstellung des S-Ausdrucks ***(A · (B · C))*** dienen, da aus ihr nicht ersichtlich ist, welche Teillisten verkettet wurden. Sie könnte ja auch

als $((\boldsymbol{A} \cdot \boldsymbol{B}) \cdot \boldsymbol{C})$ gelesen werden. Die Typdefinition von *Node* und *NodePointer* reichen allerdings zur Erzeugung linearer Listen aus, die auf folgende Weise induktiv definiert werden können:

Definition 4.2 (Lineare Liste vom Typ T)

(i) *Ein Knoten vom Typ T ist eine lineare Liste vom Typ T.*

(ii) *Die Verkettung eines Knotens vom Typ T mit einer Liste vom Typ T ist eine Liste vom Typ T.*

S-Ausdrücke dagegen haben die Gestalt von *binären Bäumen*, da jeder Knoten entweder ein atomares Symbol – und damit ein Endknoten – ist, oder zwei weitere S-Ausdrücke enthält. Wir kommen damit zu einer Obsession der Informatik, der *Baumstruktur*, der folgende Definition zu Grunde liegt:

Definition 4.3 (Baum vom Typ T)

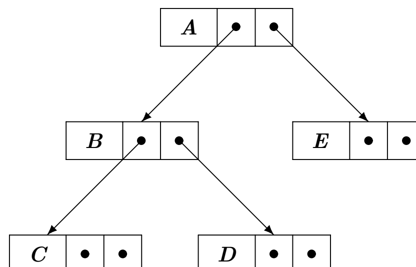
(i) *Ein Knoten vom Typ T ist ein Baum vom Typ T.*

(ii) *Ein Knoten vom Typ T verkettet mit einer beliebigen Anzahl von Bäumen vom Typ T ist ein Baum vom Typ T.*

Ein binärer Baum vom Typ T ist ein Baum, bei dem jeder Knoten mit maximal zwei Bäumen verkettet sein darf. Dazu benötigen wir eine Typdefinition, die die Speicherung von zwei Zeigern auf weitere Knoten erlaubt.

```
TYPE
   BinaryTreeNodePointer = POINTER TO BinaryTreeNode ;
   BinaryTreeNode =   RECORD
                          data: String ;
                          left, right : BinaryTreeNodePointer
                      END ;
```

Diese Baumform nimmt eine zentrale Stellung ein, doch auch die lineare Liste kann als Baum aufgefasst werden, bei dem jeder Knoten nur *einen* Teilbaum besitzt. Das wird manchmal mit drastischen Worten ausgedrückt: „Die Sequenz (Liste) heißt deshalb auch *entarteter Baum* (degenerate tree).“ (Wirth 1983, p.219) Meist wird als weitere Beschränkung auferlegt, dass Bäume sich nicht selbst als Teilbäume enthalten dürfen. Dies gilt auch für die Darstellung von S-Ausdrücken durch Listenstrukturen (McCarthy 1960, p.191–192). Um das Bild der Hierarchisierung zu vervollständigen, wird gesagt, dass Bäume mit der Wurzel nach oben dargestellt werden.



Dabei erinnern diese Darstellungen doch bei weitem mehr an Wurzelsysteme, als an Bäume, deren Stamm oben liegt und deren Äste nach unten wachsen. Doch

auch Wurzeln sind ein schwacher Trost, denn „darauf läuft auch das Prinzip der Wurzeln und Bäume hinaus, oder der Ausweg und die Lösungsmöglichkeit der Nebenwurzeln: auf die Struktur der Macht.“ (Deleuze und Guattari 1992)

Wenn wir also alle möglichen Baumstrukturen als rekursiv bezeichnen, weil sie sich induktiv definieren lassen, dann scheinen fast alle irgendwie geordneten Strukturen diese Bezeichnung zu verdienen, also auch Listen und verzweigte Wurzelsysteme. Es scheint dann kaum möglich, dem Gewicht der das Denken dominierenden n-ären Baumstrukturen zu entkommen. „And this mysterious thing called ‘recursion’ is nothing more than induction applied to programming.“ (Wand 1980) Auf diese Weise wird das Konzept der Selbstähnlichkeit unterschlagen und Rekursivität wird zum Synonym von Induktion, so wie im Titel Rekursionstheorie rekursiv nur mehr berechenbar bedeutet. Am anderen Extrem dieser Sichtweise ist „the whole world built out of recursion.“ (Hofstadter 1979, p.142) Natürlich mit dem Ausweg, dass hinreichend komplexe Systeme mächtig genug sind, um aus prädeterminierten Mustern auszubrechen. „And isn’t this one of the defining properties of intelligence?“ (Hofstadter 1979, p.152) Ein möglicher radikaler Gegenentwurf könnten die von Gilles Deleuze und Félix Guattari beschriebenen Rhizome sein. Sie sind Gefüge mit einer weder induktiven noch rekursiven Struktur. Induktiv nicht, denn das Rhizom „ist nicht das Eine, das zu zwei wird, oder etwa direkt zu drei, vier oder fünf, etc. Es ist kein Mannigfaltiges, das sich aus der Eins herleitet und dem man die Eins hinzuaddieren kann (n+1). Es besteht nicht aus Einheiten, sondern aus Dimensionen, oder vielmehr aus beweglichen Richtungen.“ (Deleuze und Guattari 1992, p.36) Rekursiv wohl auch nicht, denn „Das Mannigfaltige *muss gemacht werden*, aber nicht dadurch, dass man immer wieder eine höhere Dimension hinzufügt, sondern vielmehr schlicht und einfach in allen Dimensionen, über die man verfügt, immer $n-1$ (das Eine ist nur dann ein Teil des Mannigfaltigen, wenn es davon abgezogen wird). Wenn eine Mannigfaltigkeit gebildet werden soll, muss man das Einzelne abziehen, immer in $n-1$ Dimensionen schreiben. Man könnte ein solches System Rhizom nennen.“ (Deleuze und Guattari 1992, p.16) Allerdings stellt sich hier nicht nur die Frage nach der Machbarkeit, sondern auch die nach der Denkmöglichkeit.

Symbolverzeichnis

$\bigcap$ 14
$\in$ 12
$\neg$ 12
$\subseteq$ 12
$\wedge$ 12
$\bigvee_x$ 12
$\bigwedge_x$ 12
$\leftrightarrow$ 12
$\vee$ 12
$\rightarrow$ 12
$\emptyset$ 13, 14
$\cap$ 14
$\bigcup$ 15
$\cup$ 15
Pot 15
$x \setminus y$ 15
Ers 16
Fund 16
ω 16
$\in_x$ 18
Bild 18
Def 18
Feld 18
urv 18
Rel 18
Fkn 20
${}^x y$ 21
$f \upharpoonright y$ 21
N 22, 33
$r[u]$ 23
$\mathbb{N}$ 32
Add 32
Null_i^n 33
Proj_i^n 33
NE 34
PR 35
Mul 36
Abst 37
Diff 37
Hyp 37
Vor 37
sg 37
$Q\,\mathfrak{x}$ 40
$\mathfrak{x}$ 40
Max 41
Min 41
$\bigvee_{y \leq z}$ 41
$\bigwedge_{y \leq z}$ 41
Ack 44
μ 49, 51
Pr 51
τ 51
$\ulcorner \alpha \urcorner$ 52
$b \mid a$ 54
$b \nmid a$ 54
$\xrightarrow{\mathsf{TM}}$ 61
$\overset{\mathsf{TM}}{\twoheadrightarrow}$ 61
Res_{TM} 62
$\overline{x}$ 62
$\overline{\mathfrak{x}}$ 62
0 64
1 64
I 64
r 64
A 65
$\mathsf{Cp_n}$ 65
TM^k 65
L 65
R 65
D 66
MI 66
$\mathsf{Mv_n}$ 66
Mv 66
dec 69
goto 69
inc 69
label 69

test 69
$\equiv$ 70
while 70

Stichwortverzeichnis

- Additionsfunktion . . . 32
- al-Hwarizmi . . . 29
- Algorithmus . . . 29
- Allmenge . . . 13
- Alphabet . . . 60
- Anfang . . . 27
- Anfangsstück . . . 24
- Antinomie
  - Quine . . . 5
  - Russell . . . 5, 15
- Arithmetisierung . . . 52
- Array . . . 78
- Axiom
  - Aussonderung . . . 13
  - Ersetzung . . . 16
  - Existenz . . . 13
  - Extensionalität . . . 13
  - Fundierung . . . 16
  - Potenzmengen . . . 15
  - Unendlichkeit . . . 15
  - Vereinigung großes . . . 15
  - Vereinigung kleines . . . 15

- Bandinschrift . . . 60
- Barbier . . . 4, 8
- Baum . . . 82
- Baumstruktur . . . 82
- Berechenbarkeit . . . 29
- Bereich
  - effektiver . . . 34
- Bezeichnungssystem
  - effektives . . . 34

- Chinesischer Restsatz . . . 55
- Church-Turing These . . . 30
- Churchsche These . . . 30

- Datenstruktur
  - dynamische- . . . 78
  - statische- . . . 78
  - strukturiert . . . 78
- Datentyp . . . 77
- Definition
  - explizite . . . 25
  - rekursive . . . 27
- Differenz . . . 15
  - modifizierte . . . 37
- Durchschnitt
  - einer Menge . . . 14
  - zweier Mengen . . . 14

- Einschränkung . . . 21
- Einsetzung
  - normierte . . . 34
- Element
  - einer Menge . . . 12
  - kleinstes . . . 19
- Extension . . . 12

- Faktorielle . . . 77
- Fallunterscheidung . . . 40
- Fortsetzung . . . 21
- Funktion . . . 19, *20*
  - μ-rekursive . . . *50*, *52*
  - Abstands- . . . 37
  - bei Frege . . . 19
  - charakteristische . . . 39
  - Fortsetzung . . . 21
  - iterierte . . . 38
  - Komposition . . . 21
  - partielle . . . 32
  - primitiv rekursive . . . 32, *35*
  - rekursive . . . 29
  - Restriktion . . . 21
  - Signums- . . . 37
  - totale . . . 32
  - Wert . . . 20

- Gödelisierung . . . 52
- Gödelnummer . . . 52

- Hyperpotenz . . . 37

Induktion
- über den Aufbau . . . 58
- strukturelle . . . 58

Iteration . . . 38

Java . . . 44
Junktor . . . 12

kartesisches Produkt . . . 17
Kodierungsfunktion . . . 50
- n-stellige . . . 51

Komponentenfunktion . . . 51
- n-stellige . . . 52

Komprehensionsprinzip . . . 6
Konfiguration . . . 60

Liste
- lineare . . . 82

Makro . . . 72
Maximum . . . 40
Menge
- fundierte . . . 24
- induktive . . . 16
- kleinste induktive . . . 16
- Teil- . . . 12
- Träger- . . . 19

Minimisierung . . . 49
Minimum . . . 40
μ-Operator . . . 49
- beschränkter . . . 51

Multiplikation
- beschränkte . . . 41

Multiplikationsfunktion . . . 36

Nachfeld . . . 18
Nachfolgerin
- einer Menge . . . 22

Normierte Einsetzung . . . 34

Oberon . . . 8
Operation . . . 12
Ordnung . . . 19

Paradoxon
- Epimenides . . . 3
- falsidikal . . . 4
- Grelling . . . 5
- Lügner . . . 3
- veridikal . . . 4

Parameter . . . 16
Primitive Rekursion . . . 35
Primzahl . . . 50
Prozedur . . . 76
- effektive . . . 29

Quadrupel . . . 59
Quantifizierung
- beschränkte . . . 41

Quantor . . . 12

r-minimal . . . 19
Record . . . 78
- Varianten- . . . 79

Registermaschine . . . 69
- Programm einer . . . 70

Rekursion . . . 26
- Definition durch . . . 25
- primitive . . . 26, 35
- Vorschrift . . . 26
- Wertverlaufs- . . . 26

Rekursions-
- anfang . . . 35
- parameter . . . 35
- theorie . . . 29
- variable . . . 35
- vorschrift . . . 35

Rekursionsvorschrift . . . 26
Relation . . . 16
- $\in_x$ . . . 18
- Äquivalenz- . . . 19
- über einer Menge . . . 18
- All- . . . 18
- antisymmetrisch (identitiv . . 18
- asymmetrisch . . . 18
- binäre . . . 18
- Identität . . . 18
- irreflexiv . . . 18
- konnex . . . 18
- leere . . . 18
- linear . . . 18
- Ordnungs- . . . 19
- primitiv rekursive . . . 39
- reflexiv . . . 18
- symmetrisch . . . 18
- transitiv . . . 18

  Wohlordnungs- 23
relativ prim 53
Restriktion 21

S-Ausdrücke 81
S-expressions 81
Structure 78
Summation
  beschränkte 41
Symbol
  atomares 81

teilbar 53
Teiler 53
teilerfremd 53
Teilmenge 12
Trägermenge 19
Turing-Berechenbarkeit 62
  normierte 63
Turingmaschine 59
  Alphabet 60
  Bandinschrift 60
  Konfiguration 60
  Quadrupel 59
  Wort 59
  Zustände 60

unär 62

Variable
  freie 16
Vorfeld 18
Vorgänger 37
  Menge der r-Vorgänger 23

while-Schleife 70
Wohlordnung 24
Wort 59

Zahlen 21
Zahlendarstellung
  unäre 62
Zustände 60

Literatur

Aho, Alfred V., R. Sethi und J. D. Ullman (1986). *Compilers, principles, techniques, and tools*. Reading, MA: Addison-Wesley Publishing Company.

Bartlett, Steven J. (1992). *Reflexivity: A Source-Book in Self-Reference*. Amsterdam, London, New York, Tokyo: North-Holland.

Bibel (1989). *Altes und Neues Testament, Einheitsübersetzung*. Freiburg, Basel, Wien: Herder.

Boolos, George S. und R. C. Jeffrey (1988). *Computability and Logic* (third edition). Cambridge: Cambridge University Press.

Courant, Richard und H. Robbins (1941). *What is Mathematics?* (second edition, revised by Ian Steward, 1991). New York: Oxford University Press.

Date, C. J. (1990). *An Introduction to Database Systems* (fifth edition), Volume I. Reading, MA: Addison-Wesley Publishing Company.

Davis, Martin (1958). *Computability and Unsolvability* (3rd edition with a new appendix, 1982). New York: Dover. Reprint. Originally published by McGraw-Hill Book Company.

Deleuze, Gilles und F. Guattari (1992). *Tausend Plateaus, Kapitalismus und Schizophrenie*. Berlin: Merve. Übersetzt von Gabriele Ricke und Ronald Voullié, Originalausgabe 1980.

Devlin, Keith (1993). *The Joy of Sets: Fundamentals of Contemporary Set Theory* (second edition, completely rewritten). New York: Springer-Verlag.

Diels, Hermann und W. Kranz (1985). *Die Fragmente der Vorsokratiker* (6., verbesserte Auflage). Zürich, Hildesheim: Weidmann.

Dudenredaktion, Wissenschaftlicher Rat der (Ed.) (1994). *Duden, Das große Fremdwörterbuch*. Mannheim, Leipzig, Wien, Zürich: Dudenverlag.

Ebbinghaus, Heinz-Dieter (1994). *Einführung in die Mengenlehre* (3., vollständig überarbeitete und erweiterte Auflage). Mannheim, Leibzig, Wien, Zürich: BI-Wissenschaftsverlag.

Ebbinghaus, Heinz-Dieter, J. Flum und W. Thomas (1996). *Einführung in die mathematische Logik* (4. Auflage). Heidelberg, Berlin, Oxford: Spektrum Akademischer Verlag GmbH.

Frege, Gottlob (1904). Was ist eine Funktion? In G. Patzig (Ed.), *Funktion, Begriff, Bedeutung: fünf logische Studien* (1994), pp. 81–90. Vandenhoeck und Ruprecht.

Frege, Gottlob (1914). Logik in der Mathematik. In (Frege 1990), pp.92–165.

Frege, Gottlob (1990). *Schriften zur Logik und Sprachphilosophie, Aus dem Nachlaß* (dritte, bibliographisch neu bearbeitete Auflage). Hamburg: Meiner.

Gödel, Kurt (1931). Über formal unentscheidbare Sätze der *Principia mathematica* und verwandter Systeme I. *Monatsh. Math. Phys. 38*, pp.173–198.

Gödel, Kurt (1986). Über formal unentscheidbare Sätze der *Principia mathematica* und verwandter Systeme I. In *Collected Works: Volume I, Publications 1929–1936*. Oxford, New York: Oxford University Press.

Goslin, James, B. Joy und G. Steele (1996). *The Java Language Specification*. Reading, MA: Addison-Wesley Publishing Company.

Hofstadter, Douglas R. (1979). *Gödel, Escher, Bach: an Eternal Golden Braid*. London, New York: Pinguin Books.

Holub, Allen I. (1990). *Compiler Design in C*. Englewood Cliffs, New Jersey: Prentice-Hall International, Inc.

Kleene, Stephen Cole (1952). *Introduction to Metamathematics* (tenth impression 1991). Amsterdam, New York, Groningen: North Holland, Wolters-Noordhoff.

Krämer, Sybille (1988). *Symbolische Maschinen: die Idee der Formalisierung in geschichtlichem Abriss*. Darmstadt: Wissenschaftliche Buchgesellschaft.

Levy, Azriel (1979). *Basic Set Theory*. Ω Perspectives in Mathematical Logic. Berlin, Heidelberg, New York: Springer Verlag.

McCarthy, John (1960). Recursive Functions of Symbolic Expressions and Their Computation by Machine, Part I. *Communications of the ACM 3*(12), pp.184–195.

Meyer, Bertrand (1990). *Introduction to the theory of programming languages*. Hertfordshire: Prentice-Hall International.

Minsky, Marvin L. (1967). *Computation: Finite and Infinite Machines*. Englewood Cliffs, N.J.: Prentice-Hall.

Monk, J. Donald (1969). *Introduction to Set Theory*. New York: McGraw-Hill Book Company.

Naur, Peter (Ed.) (1963). Revised report on the algorithmic language Algol 60. *Communication of the ACM 6*(1), pp.1–17.

Oberschelp, Arnold (1993). *Rekursionstheorie*. Mannheim, Leibzig, Wien, Zürich: BI-Wissenschaftsverlag.

Post, Emil L. (1943). Formal Reductions of the General Combinatorial Decision Problem. *American Journal of Mathematics LXV*, pp.197–215.

Quine, Willard van Orman (1960). *Word and Object*. Cambridge, Massachusetts: MIT Press.

Quine, Willard van Orman (1962). Paradox. *Scientific American 20*(4), pp.84–96. wiederabgedruckt in und zitiert nach (Bartlett 1992).

Quine, Willard van Orman (1963). *Set Theory and its Logic*. Cambridge, Massachusetts: Harvard University Press.

Reiser, Martin und N. Wirth (1992). *Programming in Oberon: steps beyond Pascal and Modula*. New York, New York: ACM Press, Addison-Wesley Publishing Company.

Rogers, Hartley Jr. (1967). *Theory of Recursive Functions and Effective Computability* (1987, first MIT Press paperback edition). Cambridge, London: MIT Press.

Searle, John R. (1990). Is the brain a digital computer? *Proceedings and Addresses of the American Philosophical Association 64*(3), pp.21–37.

Smith, Einar (1996). *Elementare Berechenbarkeitstheorie*. Berlin; Heidelberg; New York: Springer Verlag.

Stroustrup, Bjarne (1997). *The C++ Programming Language* (Third edition). Reading, Massachusetts: Addison-Wesley Publishing Company.

Suppes, Patrick (1960). *Axiomatic Set Theory*. The University Series in Undergraduate Mathematics. Princeton, New Jersey: D. Van Nostrand Company, Inc.

Wand, Mitchell (1980). *Induction, recursion, and programming*. Amsterdam, The Netherlands: Elsevier Science Publishing.

Wexelblat, Richard L. (Ed.) (1981). *History of Programming Languages*. New York, London: Academic Press.

Whitehead, Alfred North und B. Russell (1910). *Principia mathematica*. Cambridge.

Wirth, Niklaus (1983). *Algorithmen und Datenstrukturen* (3. überarbeitete Auflage; erste Auflage 1975). Stuttgart: Teubner.