Der Satz von Alonzo Church

Die Unentscheidbarkeit der Prädikatenlogik

KS SS 2020

Karl-Franzens-Universität Graz

Michael Matzer

Version vom 27.07.2020, 07:44

Vorwort

Dieser Text ist im Rahmen meiner Lehrveranstaltung über den Satz von Church im Sommersemester 2020 an der Karl-Franzens-Universität Graz entstanden. Ich danke meinen Studierenden für ein herovrragendes Semester (unter außergewöhnlichen Umständen, wir hatten das ganze Semester lang Fernlehre aufgrund der Corona-Pandemie) — ich hatte so viel Freude mit Euch!

Zunächst danke ich Sebastian Rapatz, der in diesem Semester in unserer Lehrveranstaltung war, für den Hinweis auf die Existenz des Programms P_5 in unserer Liste in Abbildung 4.6 auf S. 46. Sebastian hat das Programm „entdeckt“ und auch sofort an der richtigen Position in die Liste eingereiht. Ich danke ihm auch für die angeregte Diskussion, die gezeigt hat, dass die Zusatzprämissen bei der Eliminierung der Funktionssymbole hinreichend stark sind, sodass gültige Argumentformen ihre Gültigkeit bewahren. Weiters hat er mich dankenswerterweise auf einige Tipp- und sonstige kleinere Fehler in diesem Text hingewiesen.

David Schilhan, der ebenfalls an unserer Lehrveranstaltung in diesem Semester teilgenommen hat, danke ich für zwei weitere wesentliche Hinweise bezüglich der Liste der Abacus-Programme in Tabelle 4.6 auf S. 46: Ich hatte eine Gödelzahl falsch, und die Reihenfolge zweier Programme in der Liste stimmte ursprünglich nicht. David hat mich aber zum Glück auf diese beiden Fehler, die wahrlich nicht leicht zu finden waren, hingewiesen.

Andrea Schreiner, mit der ich einst gemeinsam studiert, und die ich in unserer Lehrveranstaltung nach Jahren wiedergesehen habe, danke ich für die Korrektur einiger Fehler im Text.

Meiner Frau Kerstin Hernler danke ich für die Inspiration zu Abschnitt 4.8, denn ihre Neugier war es, die mich dazu brachte, die Gödelisierung so zu sehen, wie ich sie dort darstelle.

Inhaltsverzeichnis

Kapitel 0

Hinführung

0.1 Positionierung der Lehrveranstaltung

Wenn Sie Logik-Lehrveranstaltungen besuchen, so werden Sie diese grob in folgende Kategorien unterteilen können:

1. **Elementare Logik.** Dies ist die Einführung in die klassische Aussagen- und Prädikatenlogik, manchmal Prädikatenlogik mit Identität. Sie ist für alle weiteren Punkte dieser Aufzählung Voraussetzung.

2. **Höhere Logik.** Gehen Sie in diese Richtung weiter, befassen Sie sich z.B. mit Prädikatenlogik 2. Stufe (die über Prädikate quantifiziert).

3. **Philosophische Logik.** Sie umfasst Logiken, die sich mit spezifisch philosophischen logischen Wörtern beschäftigen, z.B. Modallogik („notwendig“ / „möglich“) , Normenlogik („geboten“ / „erlaubt“ / „verboten“), Zeitlogik („Es war immer / manchmal / nie der Fall, dass ...“ / „Es wird immer / manchmal / nie der Fall sein, dass ...“).

4. **Nicht-klassische Logik.** In diesen Logiken bestehen irgendwelche Grundannahmen der klassischen Logik nicht, etwa das Bivalenzprinzip (seine Leugnung führt zu mehrwertigen Logiken und weiter zur *fuzzy logic*), das ontologische Gewicht des Existenzquantors (das manche sehen; dies führt zu existenzfreien Logiken) oder das *ex falso quodlibet* (dessen Leugnung zu parakonsistenten Logiken führt).

5. **Philosophie der Logik.** Sie erörtert als spezielle Wissenschaftstheorie philosophische Fragen rund um die Logik, so etwa das „Wesen“ logischer Gesetze, den Zusammenhang zwischen logischen Wörtern und logischen Konstanten (insbesondere um den Pfeil als Repräsentaten des „Wenn..., dann...“ gibt es eine rege Diskussion) oder um die Erhellung von Grundbegriffen der Logik (wie Alfred Tarski dies mit „Wahrheit“ getan hat).

6. **Metalogik.** Sie untersucht die Logik mit logischen Methoden, d.h. sie macht die Logik zu ihrer eigenen Metatheorie (wie man die Logik zur Metathteorie von vielen Wissenschaften machen kann). Ihre Zweige sind z.B. Modelltheorie, Kalkültheorie, Definitionslehre ....

Im Rahmen dieser Aufstellung findet sich diese Lehrveranstaltung eindeutig im letzten Aufzählungspunkt wieder: Wir machen Metalogik, insbesondere Kalkültheorie (und ein paar verwandte Disziplinen — mehr dazu später).

Voraussetzung für diese Lehrveranstaltung ist lediglich die Kenntnis der Elementaren Logik. Syntaktik der Prädikatenlogik und Ableitungsregeln für den Kalkül des natürlichen Schließens entsprechen exakt [Lei18], außer dass ich niemals Klammerersparnisregeln verwende, weil mir Klammern beim Lesen eher helfen, als dass sie mich stören würden (vgl. [Lei18, S. 99]).

Anmerkung: Vielleicht sehen Sie ein Problem beim letzten Punkt, Logik auf sich selbst anzuwenden. Führt das nicht in einen unendlichen Regress, fragen Sie vielleicht. — Ich sehe das nicht so, da die Logik ja nicht zu ihrer eigenen *Begründung* herangezogen wird: Es wird keine Letztbegründung der Logik durch Logik versucht. Das System genügt lediglich seinen eigenen Standards.

0.2 Logiker*innen-Träume

Aus Ihrer Einführung in die Elementare Logik wissen Sie sicherlich noch, dass die Logik diejenige Wissenschaft ist, die sicht mit der Entwicklung und der Anwendung von Methoden beschäftigt, um Argumente auf ihre Gültigkeit zu prüfen.

Manche Logiker*innen haben einen Traum — einen ganz bestimten Traum, den auch ich bis zu einem gewissen Grad teile. Dieser Traum ist folgender: Es wäre wunderbar, hätten wir eine Methode, jedes beliebige Argument auf seine Gültigkeit zu prüfen. Und zwar auf eine rein mechanische Weise, indem wir irgendeinem Verfahren das Argument als Input zukommen lassen und nach endlicher Zeit ein Output herauskommt, der (wahrheitsgemäß) lautet: „gültig“ oder „ungültig“, und das für jedes beliebige Argument.

Für Argumente in der natürlichen Sprache ist dieser Traum sehr schnell ausgeträumt. Bei allem Fortschritt des NLP (*Natural Language Processing*, ein Zweig der Künstlichen Intelligenz) sind natürliche Sprachen immer noch *viel* zu komplex und wandelbar, um eine allgemeine Methode zu finden, die für darin formulierte Argumente ein Gültigkeitsprüfung vornehmen könnte.

Also haben wir Logiker*innen uns auf Kunststprachen zurückgezogen, die einen Teil der natürlichsprachlichen Argumente erfassen und zu prüfen erlauben. Diese Kunstsprachen sind *formale Sprachen*, ganz so, wie Sie aus Ihrer Ausbildung in Elementarer Logik die Sprache der Aussagenlogik und die Sprache der Prädikatenlogik bereits kennen.

Es gibt keine formale Sprache, die die Gesamtheit aller natürlichsprachlichen Argumente erfassen könnte: dazu sind es zu viele, zu unterschiedliche. Auch wenn wir von den Schwierigkeiten des NLP absehen und eine einzige logische Universalmethode im Formalen suchen würden, so wäre diese Methode viel zu kompliziert und unhandlich, um sie zu übersehen und mit ihr operieren zu können.

Also erfinden (oder „entdecken“, wenn Sie Platonist*in sind) wir Logiker*innen viele verschiedene formale Sprachen, die jeweils einen bestimmten Aspekt der Gültigkeit von jeweils verschiedenen Klassen natürlichsprachlicher Argumente erfassen. Ein besonders einfaches logisches System ist das der Aussagenlogik; es erfasst denjenigen Aspekt der Gültigkeit von Argumenten, der auf

logischen Wörtern wie „Es ist nicht der Fall, dass ...“, „und“, „oder“ usw. gegründet ist. Kommen die Wörter „alle“, „einige“, „keine“ usw. hinzu, so machen wir Prädikatenlogik, und die Sache wird schon etwas komplizierter. (Erinnern Sie sich, dass die Syntaktik, die Semantik und der Kalkül der Prädikatenlogik, wie Sie sie in Ihrer Logikausbildung kennengelernt haben, komplizierter sind als die der Aussagenlogik?) Und es wird noch komplizierter, wenn man zur höheren Logik oder zur philosophischen Logik beispielsweise aufsteigt. Man gewinnt zwar Ausdruckskraft, aber ...

0.3 Traum zerplatzt!

Wie im echten Leben zumeist, so gibt es auch in der Logik nichts umsonst: Wenn man von irgendwelchen Logiksystemen zu ausdrucksstärkeren übergeht, mit denen sich mehr und andere Argumente formalisieren lassen, so gehen immer mehr und mehr „schöne“ (damit meine ich von vielen Logiker*innen erwünschte) Eigenschaften verloren, die die einfacheren und ausdrucksärmeren Systeme noch hatten. (Vielleicht haben Sie schon einmal von Kurt Gödels Erstem Unvollständigkeitssatz gehört, der besagt, dass wenn man ein formales System so ausdrucksstark macht, dass man damit etwas Mathematik betreiben kann, es in diesem System immer mindestens einen wahren Satz gibt, der in dem System nicht beweisbar ist. Beim Aufstieg zur Arithmetik geht also die „schöne“ Eigenschaft verloren, dass man für jeden wahren Satz auch einen Beweis im System hat, und es somit im System auf einmal unbeweisbare Wahrheiten gibt.)

Der Aufstieg von der Aussagenlogik zur Prädikatenlogik hat auch den Verlust einer „schönen“ (wünschenswerten) Eigenschaft zur Folge: In der Aussagenlogik kann man von jeder Argumentform in endlich vielen Schritten (d.h. in endlicher Zeit) feststellen, ob sie gültig ist oder nicht. Die Methode der Wahrheitstafeln ist ist ein sogenanntes solches „Entscheidungsverfahren“; auch wenn das Aufstellen einer Wahrheitstafel für eine Argumentform mit vielen Aussagenvariablen sehr lange dauert, kommt es immer zu seinem Ende. Auch der Baumkalkül ist in der Aussagenlogik, wie man beweisen kann, ein Entscheidungsverfahren (meist ein effizienteres als die Wahrheitstafeln). — In der Prädikatenlogik geht das nicht mehr! D.h. obwohl die Prädikatenlogik vollständig ist, also es für jede gültige Argumentform auch einen Gültigkeitsbeweis im System gibt, gibt es dennoch kein Verfahren, mit dem von jeder prädikatenlogischen Argumentform in endlich vielen Schritten entschieden werden kann, ob sie gültig ist oder nicht. Dies ist die *Unenscheidbarkeit der Prädikatenlogik*, wie sie genannt wird, und wie sie von Alonzo Church und Alan Turing beinahe gleichzeitig im Jahre 1936 bewiesen wurde.

Der Beweis ist kein einfacher, und er wird uns, da wir ihn Schritt für Schritt durchgehen wollen, ein Semester lang beschäftigen.

0.4 Kurzer Refresher: Einige wichtige Metatheoreme

In diesem Skriptum werden hin und wieder die Begriffe der Erfüllbarkeit bzw. Unerfüllbarkeit von Satzmengen und Sätzen fallen. Hier sind ihre Definitionen — wir werden sie sogleich zum ersten Mal brauchen:

- Eine Satzmenge $\{A_1, A_2, \ldots, A_n\}$ ist *erfüllbar* gdw. es mindestens eine Interpretation gibt, unter der alle ihre Elemente $A_1, A_2, \ldots, A_n$ wahr sind. Sonst ist die Satzmenge *unerfüllbar*.

- Eine Formel A ist *erfüllbar* gdw. die Einermenge $\{A\}$ erfüllbar ist, d.h. A unter mindestens einer Interpretation wahr ist, und *unerfülbar* sonst.

Ich folge z.B. [Nol97] darin, strikt zwischen Theoremen und Metatheoremen zu unterscheiden. Dabei sind Theoreme beweisbare Sätze im Logiksystem, d.h. die Theoreme sind logisch wahre und als solche bewiesene Formeln. Metatheoreme dagegen sind beweisbare Sätze über ein Logiksystem in der Metasprache (hier: Deutsch), d.h. Metatheoreme sind beweisbare Sätze in der deutschen Sprache, die über das Logiksystem etwas Wahres aussagen. Auch der Beweis eines Metatheorems erfolgt in der Metasprache, üblicherweise etwas durchsetzt von einigen Formeln.

Für das Verständnis des Folgenden sind folgende Metatheoreme vielleicht hilfreich. Erinnern Sie sich noch an sie, aus Ihrer Ausbildung in Elementarer Logik? (Wie es sich für Theoreme und Metatheoreme gehört, sollte auf deren Nennung eigentlich deren Beweis folgen. Ich verzichte aber darauf, denn wir wollen ja zur Unentscheidbarkeit vordringen und uns weder mit komplizierter Modelltheorie herumplagen, wie für die ersten zwei, noch uns mit einigen recht einfachen Beweisen aus der Elementaren Logik aufhalten, wie für die übrigen.)

1. Wenn $A_1, \ldots, A_n \models B$, dann $A_1, \ldots, A_n \vdash B$. (Vollständigkeit der Prädikatenlogik)

2. Wenn $A_1, \ldots, A_n \vdash B$, dann $A_1, \ldots, A_n \models B$. (Korrektheit der Prädikatenlogik)

3. $A_1, \ldots, A_n \models B$ gdw. $((A_1 \wedge A_2 \wedge \ldots \wedge A_n) \rightarrow B)$ logisch wahr ist. (Eine Argumentform ist gültig gdw. die ihr zugeordnete Implikationsformel logisch wahr ist.)

4. $A_1, \ldots, A_n \models B$ gdw. die Satzmenge $\{A_1, A_2, \ldots, A_n, \neg B\}$ unerfüllbar ist. (Der Zusammenhang zwischen logischem Folgen und (Un-)Erfüllbarkeit.)

Kapitel 1

Einleitung

1.1 Kurze Geschichte des Entscheidungsproblems

Im Jahre 1900 erstellte der berühmte Mathematiker David Hilbert eine Liste von 23 Problemen als Aufgaben für die Mathematik im kommenden zwanzigsten Jahrhundert. Das zweite Problem auf der Liste war der Nachweis der Widerspruchsfreiheit der Arithmetik mit lediglich gewissen Methoden.

Im Jahre 1920 stellte er das nach ihm benannte Programm vor, in dem es seine Leitidee war, die Vollständigkeit und Widerspruchsfreiheit aller Axiomensysteme der Mathematik innerhalb dieser Systeme nachzuweisen (vgl. [Ber16, S. 9]), um sie fortan auf logisch einwandfreien, sicheren Boden zu stellen. Im Jahre 1928, in [HA28, S. 73], kam das Entscheidungsproblem für die Prädikatenlogik prominent hinzu:

> Das Entscheidungsproblem ist gelöst, wenn man ein Verfahren kennt, das bei einem vorgelegten logischen Ausdruck durch endlich viele Operationen die Entscheidung über die Allgemeingültigkeit bzw. Erfüllbarkeit erlaubt. Die Lösung des Entscheidungsproblems ist für die Theorie aller Gebiete, deren Sätze überhaupt einer logischen Entwickelbarkeit aus endlich vielen Axiomen fähig sind, von grundsätzlicher Wichtigkeit. [...] das Entscheidungsproblem muss als das Hauptproblem der mathematischen Logik bezeichnet werden.

Der schon erwähnte Kurt Gödel führte 1931 Hilberts zweites Problem einer Lösung zu, allerdings einer negativen: Der Nachweis der Widerspruchsfreiheit der Arithmetik ist innerhalb des Systems der Arithmetik nicht zu erbringen.

Auch die Lösung des Entscheidungsproblems ist eine negative: „[E]in Verfahren [...], das bei einem vorgelegten [prädikaten-]logischen Ausdruck durch endlich viele Operationen die Entscheidung über die Allgemeingültigkeit [d.i. logische Wahrheit] bzw. Erfüllbarkeit [und damit auch die Folgerungsbeziehung] erlaubt“, gibt es nicht. Beweise dafür gibt es mittlerweile viele; die ersten waren von Alonzo Church [Chu36a] und [Chu36b] und Alan Turing [Tur36]. Damit war das Hilbert-Programm vernichtet.

Wie dies häufig bei Pionierarbeiten so ist, sind diese ersten Nachweise zwar methodisch einwandfrei, aber für eine didaktisch anspruchsvolle Darstellung

eher ungeeignet. Daher habe ich mir eine Variante des Unentscheidbarkeitsbeweises ausgesucht, die sich vom Grad der formalen Abstraktheit so weit wie möglich in Grenzen hält zugunsten ihrer Anschaulichkeit.

1.2 Was Unentscheidbarkeit ist, und was sie nicht ist

Wir werden sehen, dass die Unentscheidbarkeit der Prädikatenlogik nichts ist, was in irgendeiner Weise vom gegenwärtigen Stand der Erkenntnis abhängt: Es ist keine Unzulänglichkeit unserer derzeit bekannten Kalküle, dass sie keine Entscheidungsverfahren für die Prädikatenlogik sind. Der Satz von Church besagt vielmehr ganz allgemein, dass, egal welche Kalküle und Methoden noch entwickelt werden, und sei es in noch fernen Jahrhunderten: Keine davon wird jemals ein Entscheidungsverfahren für die Prädikatenlogik sein.

Unentscheidbarkeit bedeutet insbesondere nicht:

- **Gegenwärtige Unmöglichkeit, eine Entscheidung zu fällen, weil nötige Kenntnisse oder Daten fehlen.** Das Entscheidungsproblem für die Prädikatenlogik ist so vollständig definiert, wie man sich nur wünschen kann.
- **Die Verhinderung einer Entscheidung durch vage Begriffe.** Das Entscheidungsproblem für die Prädikatenlogik ist auch so präzise definiert, wie man sich nur wünschen kann.
- **Die Tatsache, dass bisher noch keine Entscheidung vorliegt (etwa weil noch niemand einen Beweis gefunden hat).** Vielleicht findet ja jemand bereits morgen oder auch nur irgendwann einen Beweis? Denn dann würde sich das Problem als entscheidbar (weil entschieden) herausgestellt haben.
- **Die Möglichkeit, auf beliebigem Wege (also nicht nur durch einen Kalkül) zu einer Entscheidung zu gelangen.** Bei Entscheidbarkeitsfragen steht also die Frage im Zentrum: *Was können unsere Kalküle leisten?*
- **Die Unmöglichkeit, irgendeine bestimmte gültige Folgerung abzuleiten.** Ein solcher Fall kann nicht eintreten, da die Prädikatenlogik vollständig ist.

Und, wie mittlerweile klar sein dürfte, hat das Entscheidungsproblem nichts mit dem handlungstheoretischen Entscheidungsbegriff zu tun, wie er etwa in der *Rational-Choice*-Theorie vorkommt (mittels derer, in der Sozialwissenschaft, getroffene Entscheidungen untersucht werden), oder mit Fragen nach der Moralität anstehender problematischer Entscheidungen.

1.3 Was uns erwartet

In diesem Kurs werden wir uns, immer letztlich den Nachweis der Unentscheidbarkeit der Prädikatenlogik im Blick, mit folgenden Themen näher befassen: mit

1. arithmetischen Funktionen,
2. dem infiniten Abacus nach Joachim Lambek, der arithmetische Funktionen berechnet,
3. Berechnungen des Abacus und
4. dem Halteproblem für den Abacus sowie mit
5. mit Berechnungen des Abacus assoziierten Argumentformen.

Der Schluss auf die Unentscheidbarkeit der Prädikatenlogik wird dann ein einfacher *modus tollens* sein.

Im Wesentlichen ist die Beweisstrategie dieselbe wie in [Nol97, S. 268–304].

Kapitel 2

Arithmetische Funktionen

Die Arithmetik ist dasjenige Teilgebiet der Mathematik, das sich mit den Zahlen (griech. „ἀριθμός“, *arithmós*, Zahl), insbesondere den natürlichen Zahlen, und deren Eigenschaften befasst. Was wir in diesem Kurs an Mathematik brauchen werden, wird sich ausschließlich mit den natürlichen Zahlen, in unserem Falle einschließlich der Null, beschäftigen. Doch ist Arithmetik, wenn Sie bei Mathematik etwa an die Analysis mit Integral- und Differentialrechnung und ihren überabzählbar vielen reellen Zahlen denken, weder trivial noch langweilig. Ja, es gibt sogar arithmetische Funktionen, also Funktionen, die es nur mit natürlichen Zahlen zu tun haben, die die Möglichkeiten jeder denkbaren Rechenmaschine übersteigen. Das wird uns in den nächsten Kapiteln beschäftigen; zunächst wollen wir uns allerdings noch über den Begriff der arithmetischen Funktion klarer werden.

2.1 Funktionen

Ich will hier den präzisen modenen Funktionsbegriff außen vor lassen, da er zu abstrakt ist für unsere Zwecke, und wir seine Genauigkeit auch gar nicht benötigen. Der hier vorgestellte Funktionsbegriff hat seine Probleme (insbesondere, weil ich den zugrundeliegenden Mengenbegriff ungeklärt lasse), aber in die Gebiete, wo er problematisch wird, werden wir nicht vordringen.

Ein Funktionsausdruck (also ein Ausdruck, der eine Funktion bezeichnet) der deutschen Sprache ist beispielsweise die Wortreihe ‚der Vater von . . . ‘. Je nachdem, welche Person (vertreten durch ihren Namen) man an der Stelle ‚. . . ‘ einsetzt, ergibt sich die Bezeichnung für genau eine Person als „Ergebnis“, nämlich der Vater der eingesetzten Person. Versuchen wir das der Reihe nach mit den Namen dreier Personen:

$$\begin{aligned} \text{der Vater von Michael} &= \text{Egon} \\ \text{der Vater von Kerstin} &= \text{Rudolf} \\ \text{der Vater von Reinhard} &= \text{Reinhard sen.} \end{aligned}$$

Wesentlich für Funktionen ist die Eindeutigkeit des Ergebnisses der Zuordnung; in unserem Beispiel: genau ein (biologischer) Vater für jede Person. Jeder

Person wird genau eine Person als ihr Vater zugerordnet, und das ist auch richtig so. Denn jede Person hat genau einen (biologischen) Vater; niemand hat zwei oder mehr Väter, und niemand hat gar keinen. Es ist leicht einzusehen, dass der Ausdruck ‚der Vater von . . . ' ein Funktionausdruck in diesem Sinne ist.

Nicht jeder Ausdruck, der ein/n Verwandte*n bezeichnet, ist übrigens ein Funktionsausdruck; ich hatte den (biologischen) Vater nicht ganz ohne Bedacht als unser erstes Beispiel gewählt. Der Ausdruck ‚der Bruder von . . . ' beispielsweise ist kein Funktionsausdruck, denn nicht jede Person hat genau einen Bruder. Manche Menschen haben dies zwar, aber es gibt sehr viele Menschen ohne einen einzigen Bruder, und auch sehr viele, die mehr als einen Bruder haben.

Man kann sich eine Funktion vorstellen als eine kleine Maschine, in die gewisse Dinge reingetan werden können (bei der Funktion, die Menschen auf ihre Väter abbildet, sind das eben Menschen), und aus der daraufhin gewissse Dinge rauskommen (bei derselben Funktion sind das gewisse Männer). Die Funktion, die von dem Ausdruck ‚der Vater von . . . ' bezeichnet wird, hat Menschen zu ihrem „Input" und gibt uns Männer als „Output". Je einen Menschen, der als Input zu der Funktion fungiert, nennen wir das *Argument* der Funktion, und den jeweils zugeordneten Mann nennen wir den *Wert* der Funktion für das entsprechende Argument. (Diese Verwendung des Wortes ‚Argument' hat nichts mit jener aus der Logik zu tun, der gemäß ein Argument gültig oder ungültig sein kann!)

In dem obigen Beispiel hat die Funktion ‚der Vater von . . . ' für das Argument *Michael* den Wert *Egon*, für das Argument *Kerstin* den Wert *Rudolf*, usw.

Die Menge, die die Argumente der Funktion enthält, nennen wir den *Argumentbereich* oder den *Definitionsbereich* (engl. *domain*) der Funktion; die Menge, aus der die Funktionswerte genommen werden, ist der *Wertebereich* der Funktion (engl. *range*).

Für unser obiges Beispiel bedeutet dies: Die Funktion, die von dem Ausdruck ‚der Vater von . . . ' bezeichnet wird, bildet Menschen auf Männer ab; ihr Argumentbereich / Definitionsbereich ist die Menge der Menschen, und ihr Wertebereich ist die Menge der Männer.

Noch etwas ist für Funktionen (vorerst) wichtig: Jedem Argument muss ein Wert zugeordnet werden. Es ist also wieder kein Zufall, dass ich die (biologische) Vaterschaft als Beispiel gewählt habe, denn es gibt keinen Menschen, der keinen Vater hat: Jeder Mensch hat einen Vater. Auf der anderen Seite muss nicht jedes Element des Wertebereichs Wert der Funktion für irgendein Argument sein: Nicht alle Männer sind tatsächlich (biologische) Väter, es gibt auch in diesem Sinne kinderlose Männer. — Wir sagen: die Funktion, die von dem Ausdruck ‚der Vater von . . . ' bezeichnet wird, ist eine Abbildung *von* der Menge der Menschen *in* die Menge der Männer. Das ‚von' bedeutet dabei, dass es für jedes Argument (je ein Mensch) einen Wert (einen Vater) gibt; das ‚in', dass nicht jedes Element des Wertebereichs (jeder Mann) ein Wert der Funktion für irgendein Argument (Vater von irgendjemandem) sein muss.

2.2 Einstellige arithmetische Funktionen

Wenn Sie die bisherigen Informationen aus diesem Kapitel, nämlich darüber, was Arithmetik ist, und das, was wir bereits über Funktionen wissen, zusammenfügen, so ergibt sich leicht der Sinn der Wortreihe „arithmetische Funktion":

Arithmetische Funktionen sind Funktionen, die natürliche Zahlen zum Argument haben und wiederum natürliche Zahlen als Werte. Arithmetische Funktionen sind also Funktionen *von* der Menge der natürlichen Zahlen *in* die Menge der natürlichen Zahlen. (Das ‚von' und das ‚in' sind in dem technischen Sinne gebraucht wie am Ende der letzten Abschnitts.) — Das Kommende bereits vor Augen setze ich hinzu: *Einstellige* arithmetische Funktionen sind Funktionen von der Menge der natürlichen Zahlen in die Menge der natürlichen Zahlen. Kurz schreiben wir für eine einstellige arithmetische Funktion f von der Menge der natürlichen Zahlen in die Menge der natürlichen Zahlen:

$$f\colon \mathbb{N} \to \mathbb{N}$$

Ein ausgesprochen wichtiges Beispiel für eine einstellige arithmetische Funktion ist die Nachfolgerfunktion, die jede Zahl auf ihren Nachfolger abbildet, also

$$\begin{array}{rcl} 0 & \mapsto & 1 \\ 1 & \mapsto & 2 \\ 2 & \mapsto & 3 \\ & \vdots & \end{array}$$

(Den kleinen Rechtspfeil mit dem Balken links ‚$\mapsto$' lesen wir als „... wird abgebildet auf ...".)

Wir schreiben für die Nachfolgerfunktion einen kleinen hochgestellten Strich ‚$'$' nach ihrem Argument und können so folgende Gleichungen formulieren:

$$\begin{array}{rcl} 0' & = & 1 \\ 1' & = & 2 \\ 2' & = & 3 \\ & \vdots & \end{array}$$

Die Nachfolgerfunktion ist eine einstellige Funktion von der Menge der natürlichen Zahlen in die Menge der natürlichen Zahlen. Das kann man sich leicht klarmachen: Jede natürliche Zahl hat einen Nachfolger, auch jede noch so große (um eins kann man immer weiterzählen), aber nicht jede natürlich Zahl ist Nachfolger einer natürlichen Zahl. Denn die Null ist die einzige natürliche Zahl, die nicht Nachfolger einer natürlichen Zahl ist.

Mit der Null und dem Funktionszeichen für den Nachfolger kann man jede natürliche Zahl bezeichnen: Die Eins ist der Nachfolger der Null, die Zwei ist als Nachfolger der Eins der Nachfolger des Nachfolgers der Null, usw. Unsere üblichen arabischen Ziffern werden damit zur Definitionssache:

$$\begin{array}{rcl} 1 & =_{\mathrm{df}} & 0' \\ 2 & =_{\mathrm{df}} & 1' = 0'' \\ 3 & =_{\mathrm{df}} & 2' = 1'' = 0''' \\ & \vdots & \end{array}$$

Die auf diese Weise bezeichnete Zahl erhält man durch Abzählen der kleinen Strichlein, die auf die Null folgen.

Wenn Ihnen das ein wenig trivial erscheint, so ersuche ich Sie um ein wenig Geduld, denn diese Dinge werden uns später noch ganz entscheidend beschäftigen. Und nicht zuletzt zeigt sich, dass man alle berechenbaren Funktionen der Arithmetik allein mit der Null, der Nachfolgerfunktion sowie einem Schema, das „allgemeines Rekursionsschema“ heißt, darstellen kann. Diese Idee stammt von Thoralf Skolem (nachzulesen in [Sko23]); die erste Monographie [Pét67] über rekursive Funktionen stammt von Péter Rózsa, einer ungarischen Mathematikerin. (Im Ungarischen ist es üblich, den Nachnamen voran zu schreiben, sodass es sich hier um eine Frau mit Vornamen ‚Rózsa‘ und Nachnamen ‚Péter‘ handelt.)

Eine andere einstellige arithmetische Funktion ist beispielsweise diejenige, die von der Wortreihe ‚das Quadrat von . . . ‘ bezeichnet wird: $n \mapsto n^2$. Sie bildet ab

$$\begin{aligned} 0 &\mapsto 0 \\ 1 &\mapsto 1 \\ 2 &\mapsto 4 \\ 3 &\mapsto 9 \\ 4 &\mapsto 16 \\ &\vdots \end{aligned}$$

Eine besonders einfache Funktion, die man manchmal braucht, ist die Identitätsfunktion, die jede Zahl auf sich selbst abbildet, $f(n) = n$:

$$\begin{aligned} 0 &\mapsto 0 \\ 1 &\mapsto 1 \\ 2 &\mapsto 2 \\ 3 &\mapsto 3 \\ &\vdots \end{aligned}$$

Es gibt auch konstante arithmetische Funktionen, die jedes Argument auf dieselbe Zahl abbilden, so z.B. $f(n) = 3$:

$$\begin{aligned} 0 &\mapsto 3 \\ 1 &\mapsto 3 \\ 2 &\mapsto 3 \\ 3 &\mapsto 3 \\ &\vdots \end{aligned}$$

Eine einstellige arithmetische Funktion, mit der man sehr viel anfangen kann, ist die Funktion $\mathrm{Pr}\colon \mathbb{N} \to \mathbb{N}$, die $n \geq 1$ auf die n-te Primzahl abbildet (vgl.

[Göd86, S. 162, Nr. 5]). Sie spielt eine kaum zu überschätzende Rolle in einer Erfindung Gödels, mit der wir uns später noch beschäftigen werden, vgl. Abschnitt 4.5.

2.3 Mehrstellige arithmetische Funktionen

Einstellige arithmetische Funktionen sind vielleicht zunächst schon ganz interessant, aber wir wollen auch Funktionen erfassen, die von Wortreihen wie ‚die Summe von ... und ...' oder ‚das Produkt von ... und ...' bezeichnet werden. Wie man bereits an den Wortausdrücken sieht, sind hier zwei natürliche Zahlen als Argumente im Spiel. Um dies formal in den Griff zu kriegen, müssen wir uns zunächst ein wenig mit geordneten Tupeln befassen.

„Tupel" ist ein Kunstwort, das die Verallgemeinerung der Reihe *Paar — Tripel — Quadrupel* — usw. bezeichnen soll. Für natürliche Zahlen $n \geq 2$ spricht man von n-Tupeln und meint damit die eben genannte Reihe in beliebiger Fortführung: Ein 2-Tupel ist also ein Paar, ein 3-Tupel ein Tripel, ein 4-Tupel ist ein Quadrupel; dann geht es weiter mit Quintupeln, Sextupeln, Septupeln, usw. (Die Endung ‚-tupel', die bei größeren Zahlen regelmäßig auftritt, rechtfertigt die Bezeichnung der Verallgemeinerung.)

„Geordnet" bedeutet für unsere Tupel, dass die Reihenfolge, in denen ihre Glieder stehen und genannt werden, nicht egal ist. Zwei Personen, Hans und Beate, können zu einem Paar (einem 2-Tupel) gefasst werden — genauer, zu zwei geordneten Paaren. Geordnete Tupel schreiben wir in spitzen Klammern, also können wir die beiden geordneten Paare als ‚⟨Hans, Beate⟩' und ‚⟨Beate, Hans⟩' notieren. Dass die Reihenfolge im Allgemeinen eine Rolle spielt, sehen wir daran, dass beispielsweise das zweistellige Prädikat ‚... liebt ...' auf das erste Paar angewendet eine andere Aussage ergibt als dasselbe Prädikat auf das zweite Paar. Der Satz, der im ersten Fall entsteht: „Hans liebt Beate" ist ein anderer, der im zweiten Fall entsteht: „Beate liebt Hans".

Oder, ein typischer Fall für geordnete Tupel, an dem man die Wichtigkeit der Reihenfolge einsehen kann, sind Richtungen: Das geordnete Paar von Städten ‚⟨Wien, Graz⟩' ist ein anderes als ‚⟨Graz, Wien⟩'. Die Anwendung des zweistelligen Prädikats ‚Ich fahre von ... nach ...' zeigt dies: Im ersten Fall beginnt meine Reise in Wien und endet in Graz, im zweiten Fall verhält es sich genau umgekehrt.

Übrigens war es kein Zufall, dass ich soeben Prädikate der Stelligkeit 2 auf 2-Tupel, also Paare, angewendet habe. Allgemein sind n-stellige Prädikate auf n-Tupel anzuwenden. (Das ist die formale Ausführung des Satzes, den Sie bereits aus der Elementaren Logik kennen: „Ein n-stelliges Prädikat gefolgt von n-vielen Termen ist eine Formel.") — Es gibt auch Funktionszeichen der Stelligkeit n für natürliche Zahlen $n \geq 2$, und diese sind dann entsprechend auf n-Tupel anzuwenden.

Beispielsweise die schon erwähnte Summenfunktion ist eine zweistellige arithmetische Funktion. Sie bildet zwei Zahlen auf ihre Summe ab, genauer: Sie bildet ein geordnetes Paar natürlicher Zahlen auf die Summe seiner Glieder ab,

$$\langle m, n \rangle \mapsto m + n$$

So z.B.

$$\begin{aligned}\langle 1,1\rangle &\mapsto 2\\ \langle 1,2\rangle &\mapsto 3\\ \langle 2,1\rangle &\mapsto 3\\ \langle 5,3\rangle &\mapsto 8\\ &\vdots\end{aligned}$$

Dass hier das Argument der Summenfunktion ein geordnetes Paar natürlicher Zahlen (nämlich das Paar der beiden Summanden) ist, drücken wir mit dem ‚$\times$‘ aus, wie es hier links von dem Rechtspfeil verwendet wird:

$$+\colon \mathbb{N}\times\mathbb{N}\to\mathbb{N}$$

Dabei bezeichnet die Menge links von dem Kreuz diejenige Menge, aus der das erste Paarglied genommen ist, und die Menge rechts von dem Kreuz diejenige, aus der das zweite Paarglied genommen ist. Für unsere Summenfunktion ist das beide Male die Menge der natürlichen Zahlen, denn genau zwei davon wollen wir ja schließlich addieren.

Bei der Addition ist die Reihenfolge der Paarglieder für das Ergebnis irrelevant, denn es gilt für alle $m, n \in \mathbb{N}$: $m + n = n + m$. Dass das nicht für jede zweistellige arithmetische Funktion gilt, sieht man beispielsweise an der Potenz-/Exponentialfunktion, denn es gilt nicht allgemein: $m^n = n^m$, so z.B. ist $5^2 \neq 2^5$, denn $5^2 = 25$, während $2^5 = 32$.

Das Konzept der zweistelligen arithmetischen Funktion ist verallgemeinerbar: So gibt es dreistellige, vierstellige, etc., ja, allgemein: für jede natürliche Zahl $n \geq 1$ beliebig viele n-stellige Funktionen. Aber wir werden für das, was noch kommt, keine einzige davon brauchen.

Kapitel 3

Der infinite Abacus

In diesem Kapitel beginnt unsere Auseinandersetzung mit der theoretischen Informatik, und wir werden eine ideale Maschine kennenlernen, die gewisse arithmetische Funktionen berechnen kann — und zwar kann sie, so die mittlerweile weitest verbreitete Meinung, genau jene arithmetischen Funktionen berechnen, die überhaupt berechenbar sind. Es gibt nämlich auch arithmetische Funktionen, die nicht berechenbar sind, und eine davon wird für das Weitere ganz entscheidend sein. Die Maschine, die wir nun zu studieren beginnen, ist der infinite Abacus nach Joachim Lambek, vgl. [Lam61].

3.1 Wie der infinite Abacus rechnet

Der infinite Abacus (ich werde das Adjektiv ‚infinit' im Folgenden häufig weglassen, denn mit anderen Abaci werden wir es nicht zu tun haben) — der Abacus nun ist in gewisser Weise eine Vereinfachung unserer modernen Computer, in gewisser anderer Weise ist er frei von ihren technischen Beschränkungen.

Für seine im ersten Absatz versprochene Mächtigkeit, jede berechenbare arithmetische Funktion berechnen zu können, ist die Einfachheit der Funktionsweise des infiniten Abacus vielleicht überraschend. Er kann nämlich im Wesentlichen nur drei Dinge, und zwar kann er

1. den Nachfolger einer natürlichen Zahl bestimmen,
2. die Zahl bestimmen, deren Nachfolger je eine natürliche Zahl ist (und wir erinnern uns, dass die Null nicht Nachfolger irgendeiner natürlichen Zahl ist!), sowie
3. mit dem Rechnen aufhören.

Der infinite Abacus hat im Gegensatz zu technisch realisierbaren Computern einen potenziell unendlich großen Speicher; potenziell unendlich ist seine Größe, damit die Berechnung irgendwelcher sehr komplizierter arithmetischer Funktionen nicht bereits daran scheitert, dass nicht genug Speicher vorhanden ist. Keine Berechnung braucht wirklich unendlich viel Speicher; es soll einfach für jeden möglichen Anwendungsfall genug da sein.

Der Speicher des Abacus ist unterteilt in jeweils diskrete Speicherzellen („Register"), die mit den positiven natürlichen Zahlen durchnummeriert sind.

Der zweite augenfällige Unterschied zum Speicher eines realen Computers ist der, dass jede Speicherzelle des Abacus eine beliebig große Zahl speichern kann. Während man in jeder Speicherzelle eines realen Computers lediglich eine Zahl einer gewissen Größe und keine größere speichern kann, unterliegt der Abacus auch dieser Beschränkung nicht. Der Grund ist wiederum derselbe wie der für die beliebig große Anzahl der Register: Jede noch so große Zahl, die in irgendeiner Berechnung vielleicht vorkommt, soll in jedem Register Platz haben. Die Weise, in der der infinite Abacus also frei von den technischen Beschränkungen realer Computer ist, ist die Bemessung seines Speichers. Da er also nicht real (weil gar nicht realisierbar) ist, und weil er diskrete Register als Speicherzellen nutzt, ist der Abacus eine „ideale Registermaschine“.

Die oben genannten Operationen des Abacus können auch als Befehle formuliert werden, die der Abacus ausführen kann. (Die Rede von „Befehlen“, die vom Computer „ausgeführt“ werden, ist in der Informatik so weit verbreitet, dass zumeist eigens daran erinnert werden muss, dass sie metaphorisch ist.)

Operation	Befehl
den Nachfolger der natürlichen Zahl in Register Nr. k bestimmen	„Erhöhe des Inhalt des Registers Nr. k um eins!“
die Zahl bestimmen, deren Nachfolger die natürliche Zahl in Register Nr. k ist	„Verringere den Inhalt des Registers Nr. k um eins!“
mit dem Rechnen aufhören	„Halt!“

Tabelle 3.1: Operationen als Befehle

War das jetzt sehr abstrakt? Wenn ja, hilft vielleicht folgendes Bild: Sie können sich den Speicher des Abacus vorstellen wie eine unendlich lange Reihe von Behältern, die mit ‚1‘ beginnend durchnummeriert sind. In jeden Behälter können beliebig viele Kugeln gelegt werden: entweder gar keine, oder eine, oder zwei, ... Es ist auch ein unendlich großer Vorrat an Kugeln da, um immer genug zu haben. Die ersten zwei Befehle aus Tabelle 3.1, die Registerinhalte verändern, können dann wie folgt anschaulich reformuliert werden:

Befehl	bildlich gesprochen
„Erhöhe des Inhalt des Registern Nr. k um eins!“	„Lege in Behälter Nr. k eine Kugel hinein!"
„Verringere den Inhalt des Registers Nr. k um eins!“	„Nimm aus Behälter Nr. k eine Kugel heraus!“

Tabelle 3.2: Befehle im Bild

Der Abacus führt diese Befehle nicht wahllos und in beliebiger Reihenfolge aus, denn er soll ja sinnvoll arithmetische Funktionen berechnen. Daher braucht er eine Vorschrift, die festlegt, wann welche Operation stattfindet (wann welcher Befehl auszuführen ist): ein Programm. Das Programm wird von einer Steuereinheit abgearbeitet, deren Details uns hier nicht zu interessieren brauchen. Bezüglich der Anzahl der Befehle, die in einem Programm vorkommen

können, ist der Abacus eine Vereinfachung unserer gegenwärtigen Computer: Denn diese kennen weit mehr verschiedene Befehle als unser Abacus, nämlich nicht nur drei, sondern einige hundert.

3.2 Wie der Abacus programmiert wird

Während die Programme realer Computer zumeist als Listen von Befehlen geschrieben werden, kann man Abacus-Programme besonders gut zeichnen. Abacus-Programme sind Graphen, die aus Knoten und Kanten bestehen, einer genau bestimmten Art (und zwar sind sie endliche, gerichtete, zusammenhängende, kantengefärbte Graphen).

Jeder Knoten eines Abacus-Programmgraphen (da die Rede eindeutig ist, werde ich kurz auch von „Abacus-Programmen“ oder auch nur schlicht „Programmen“ sprechen) repräsentiert einen Befehl, wie er oben in Tabelle 3.2 beschrieben ist, wobei die Knoten für „Halt!“ i.d.R. nicht gezeichnet werden. Damit gibt es zwei Arten von Befehlen, für die es Knoten gibt — ein Knoten für einen Befehl, den Inhalt eines Registers um eins zu erhöhen, sieht z.B. wie in Abbildung 3.1 aus.

Abbildung 3.1: Abacus-Befehl: Erhöhen

Die Fünf in dem Kreis bedeutet, dass der Befehl Register Nr. 5 betrifft (Behälter Nr. 5 in unserem Bild von vorhin); wir nennen die 5 auch die „Registernummer“ des Befehls. Das Pluszeichen bedeutet, dass der Inhalt des Registers Nr. 5 um eins erhöht werden soll (in unserem Bild: dass eine [weitere] Kugel in Behälter Nr. 5 hineingelegt werden soll).

Die zweite Art von Knoten in Abacus-Programmen weist an, dass der Inhalt eines ganz bestimmten Registers um eins verringert werden soll (in unserem Bild: dass eine Kugel aus einem ganz bestimmten Behälter herausgenommen werden soll). Beispielsweise sieht das für Register (Behälter) Nr. 2 so aus:

Abbildung 3.2: Abacus-Befehl: Verringern

Das Minuszeichen besagt, dass der Inhalt des Registers um eins zu verringern ist.

Abacus-Programme werden, wie bereits gesagt, als gerichtete Graphen dargestellt, und wir haben die zwei Typen von Knoten unserer Abacus-Programmgraphen soeben kennengelernt. Die (gerichteten) Kanten unserer Programmgraphen sind Pfeile, die festlegen, in welcher Reihenfolge die Knoten zu durchlaufen sind, d.h. in welcher Reihenfolge die Befehle abzuarbeiten sind.

Dabei gilt für jedes Abacus-Programm:

- Genau ein Pfeil legt fest, wo die Verarbeitung beginnt. Wir zeichnen ihn so, dass er von keinem Befehl ausgeht, aber zu genau einem Befehl hinführt. Diesen Pfeil nennen wir den *Programmeingang*.
- Es gibt keinen, einen oder mehrere Pfeile, an denen die Verarbeitung hält. Diese Pfeile führen zu einem nicht gezeichneten Befehl „Halt!“ — wir zeichnen sie so, dass sie „ins Leere“ zeigen. Wir nennen diese Pfeile die *Programmausgänge*.

Für jeden Befehl steht eindeutig fest, welcher Befehl nach seiner Verarbeitung ausgeführt werden soll. (Der Abacus hat also nie eine „Wahl“, ob vielleicht der eine oder vielleicht doch lieber ein anderer Befehl verarbeitet werden soll: Er ist eine „deterministische Maschine“.) Für ‚+‘-Befehle bedeutet dies: Von jedem ‚+‘-Befehl zeigt genau ein Pfeil weg.

Bei ‚−‘-Befehlen gibt es allerdings zwei Fälle zu berücksichtigen: Der erste ist, dass die Zahl bestimmt werden kann, deren Nachfolger der Inhalt des betreffenden Register ist. Das bedeutet, dass der Inhalt des betreffenden Registers um eins verringert werden kann; in unserem Bild: dass in dem Behälter noch mindestens eine Kugel enthalten ist, die herausgenommen werden kann. Die Null allerdings ist nicht Nachfolger irgendeiner natürlichen Zahl (darauf habe ich bereits in der Liste auf S. 19 im Vorausblick auf genau dies hier hingewiesen), sodass nicht bestimmt werden kann, Nachfolger welcher natürlichen Zahl sie ist. D.h. wenn in einem Register die Null gespeichert ist, kann der Inhalt dieses Registers nicht mehr um eins verringert werden; das passende Bild dazu ist, dass man aus einem leeren Behälter keine Kugel mehr herausnehmen kann. Daher führen von jedem ‚−‘-Befehl genau zwei Pfeile weg: Der eine ist für den „gewöhnlichen“ Fall da, dass der Inhalt des Registers um eins verringert werden konnte (weil er nicht 0 war); der andere ist mit einer kleinen ‚0‘ (Null) beschriftet und ist zu verfolgen, wenn der Inhalt des Registers nicht um eins verringert werden konnte, da er 0 war. (Diese Beschriftung dieser Pfeile — eben der Kanten — des Graphen rechtfertigt seine Bezeichnung als „kantengefärbt“: Man könnte sich statt der Beschriftung mit der Null nämlich auch die Darstellung des Pfeils in einer anderen Farbe als schwarz vorstellen.)

Etwas allgemeiner sehen unsere Knoten für die möglichen Befehle in Abacus-Programmen wie folgt aus (das ‚k‘ bezeichnet das Register, den der Befehl betrifft, ist also in jedem konkreten Programm eine positive natürliche Zahl):

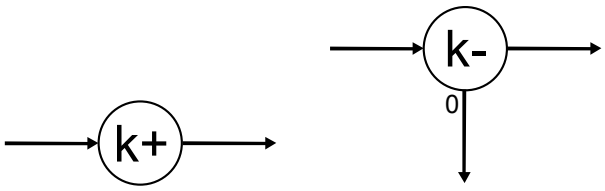


Abbildung 3.3: Die beiden Abacus-Befehle

Es ist vielleicht bemerkenswert, dass das Verfolgen des mit einer Null beschrifteten Pfeils in dem Fall, dass ein Registerinhalt nicht verringert werden kann, die einzige Verzweigungsmöglichkeit in einem Abacus-Programm ist.

3.3 Berechnungen des infiniten Abacus

Damit können wir bereits ein ganz einfaches Abacus-Programm zeichnen. Es bestimmt den Nachfolger der natürlichen Zahl, die vor Beginn der Verarbeitung im ersten Register gespeichert ist — es ist in Abbildung 3.4 zu sehen.

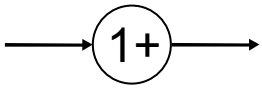


Abbildung 3.4: Abacus-Programm: die Nachfolgerfunktion

Überhaupt wollen wir eine Konvention treffen, die fortan für alle weiteren Abacus-Programme gelten soll:

Konvention 3.1. Wenn der Abacus den Wert der Funktion $y = f(x_1, x_2, \ldots, x_n)$ berechnen soll, so sind

- zu Beginn der Verarbeitung die Argumente $x_1, x_2, \ldots, x_n$ der Reihenfolge nach in den ersten n-vielen Registern gespeichert sowie
- in allen übrigen Registern der Wert 0, und
- der Funktionswert y ist, wenn der Abacus an einem Programmausgang angelangt ist, im ersten Register gespeichert.

Wir sehen, dass das Programm für die Nachfolgerfunktion in Abbildung 3.4 dieser Konvention genügt.

Nennen wir noch, der Einfachheit halber, die Argumente $x_1, x_2, \ldots, x_n$, die vor Beginn der Verarbeitung in den ersten n-vielen Registern stehen, den *Input* des jeweiligen Programms.

Konvention 3.2. Wenn in einem Abacus-Programm ein Befehl ein bestimmtes Register betrifft (ein Befehl mit einer bestimmten Registernummer vorkommt), dann gibt es in dem Programm auch für jede kleinere Registernummer mindestens einen Befehl.

D.h. die Registernummern, die in einem Programm vorkommen, beginnen stets mit 1 und haben keine „Lücken“. — Diese Konvention ist keine wesentliche Einschränkung, denn:

- Jedes Programm, in dem das erste Register nicht verändert wird, berechnet die Identitätsfunktion und ist redundant, da die Identitätsfunktion auch mit einem Programm berechnet werden kann, in dem das erste Register verändert wird; und
- in jedem Programm, das „Lücken“ in den Registernummern hat, können die Register so umnummeriert werden, dass die Registernummern eine zusammenhängende Reihe bilden.

Wir stellen uns eine Berechnung des Abacus zunächst wie folgt vor: Der Programmablauf verharrt stets auf einem Pfeil (einer Kante des Programmgraphen) und „springt“ nach einer gewissen Zeit (sagen wir, einer Sekunde) über

genau einen Befehl (Knoten des Programmgraphen) auf den Pfeil, der von dem Knoten weg zu nehmen ist. (Wir erinnern uns: Die Frage, welcher von einem Knoten wegzeigende Pfeil als nächstes zu nehmen ist, ist immer eindeutig zu beantworten; es gibt dabei nie eine wie auch immer geartete Wahlmöglichkeit.)

Einen Pfeil, auf dem die Verarbeitung nach Abarbeitung eines Befehls „verweilt“, nennen wir auch einen *Zustand* des Abacus.

Konvention 3.3. Die Zustände des Abacus bezeichnen wir jeweils mit kleinen ‚q‘ mit unteren Indizes aus den natürlichen Zahlen. Insbesondere bezeichnen wir mit ‚q_0‘ den Programmeingang.

Das Abacus-Programm für die Nachfolgerfunktion mit eingetragenen Zuständen zeigt folende Abbildung:

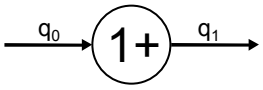


Abbildung 3.5: Das Nachfolgerprogramm, mit eingetragenen Zuständen

Dass der Abacus während einer Berechnung gerade in einem bestimmten Zustand ist, kennzeichnen wir durch einen Punkt in der Nähe des Pfeils, der ebendieser Zustand ist. So zeigt Abbildung 3.6 den Abacus mit dem Nachfolerprogramm, wie er auf dem Programmausgang verweilt: im Zustand q_1 ist.

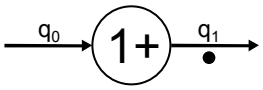


Abbildung 3.6: Der Abacus im Zustand q_1

Um den Stand der Verarbeitung des Abacus in einem gewissen Zeitpunkt zu kennen, müssen wir wissen,

- um welchen Zeitpunkt es sich handelt, d.h. wie viele Schritte der Abacus schon gemacht hat, wobei wir mit 0 zu zählen beginnen,
- in welchem Zustand der Abacus gerade steht, und
- eine Liste der Registerinhalte, beginnend mit Register Nr. 1, aufsteigend nach Registernummer.

Eine Liste mit diesen Informationen in dieser Reihenfolge nennen wir eine *Konfiguration* des Abacus. (Eine Konfiguration ist beispielsweise die Liste ‚1, q_1, 8‘: Sie beschreibt einen Abacus, der zum Zeitpunkt 1 in Zustand q_1 steht, mit der Zahl 8 im ersten Register.)

Abbildung 3.7 zeigt Schritt für Schritt die Berechnung des Nachfolgers der 7. Unter den Bildern, die zeigen, in welchem Zustand der Abacus jeweils gerade ist, steht die Konfiguration für den jeweiligen Zeitpunkt. (Wenn man ein einziges Bild mit eingetragenen Zuständen hat, sind die jeweiligen Bilder zu den Konfigurationen sogar überflüssig; sie sind hier der Anschaulichkeit wegen aber dargestellt.)

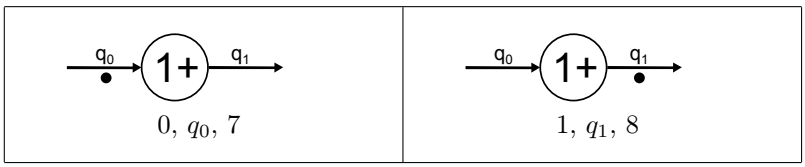


Abbildung 3.7: Berechnung eines Nachfolgers im Detail

In unserem Bild mit den Behältern und den Kugeln bedeutet dies: Zu Beginn dere Berechnung in Behälter Nr. 1 sind sieben Kugeln. Nach 0 Sekunden steht der Programmablauf am Programmeingang: Das ist der Pfeil, der ganz links in der Abbildung von keinem Knoten ausgeht: Zustand q_0. Nach einer Sekunde verarbeitet der Abacus den einzigen Befehl in dem Programm, legt eine Kugel in Behälter Nr 1 hinein und hält auch schon: Der Pfeil, der von dem Befehl wegführt, d.i. Zustand q_1, ist ein (der einzige) Programmausgang. In Behälter Nr. 1 sind, ganz gemäß unserer Konvention 3.1, nun 8 Kugeln, und 8 ist der Nachfolger von 7.

Etwas abstrakter ausgedrückt: Am Programmeingang, im Zustand q_0, ist die Zahl 7, das Argument für unsere Berechnung (der Input des Programms), gemäß Konvention 3.1 im ersten Register gespeichert. In einem Schritt (beim Übergang in den Zustand q_1) erhöht der Abacus den Inhalt des ersten Registers, sodass in Register Nr. 1 nun die Zahl 8 gespeichert ist: der Nachfolger der 7. Nach diesem Schritt hält die Berechnung im Zustand q_1 (d.i. der Programmausgang), und der korrekt berechnete Funktionswert ist gemäß Konvention 3.1 im ersten Register gespeichert.

Können wir ein Programm konstruieren, das die Summe zweier natürlicher Zahlen berechnet? — Aber gewiss! Wir brauchen nur zu überlegen: Für die Berechnung der Summe zweier Zahlen $y = x_1 + x_2$ sind gemäß Konvention 3.1 die beiden Summanden x_1 und x_2 in den ersten beiden Registern gespeichert, und am Ende der Berechnung soll deren Summe y im ersten Register stehen. Wir konstruieren in unserem Bild das Abacus-Programm so, dass es schlicht den zweiten Behälter in den ersten „entleert“ und dann hält: Dann sind (klarerweise) im ersten Behälter nachher so viele Kugeln enthalten wie zuvor in den ersten zwei Behältern zusammen. Abbildung 3.8 auf S. 25 zeigt das Programm.

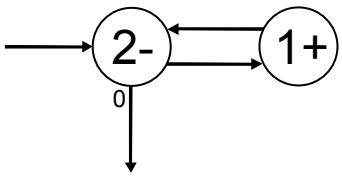


Abbildung 3.8: Abacus-Programm: die Summe zweier Zahlen

Verbleiben wir noch kurz bei diesem Programm, um genau zu verstehen, wie es funktioniert, und sehen wir uns wieder eine konkrete Berechnung an, nämlich beispielsweise die Addition von 5 und 3. Bezeichnen wir zunächst die

Zustände des Programms wieder mit kleinen ‚q‘ mit unteren Indizes, dies zeigt uns Abbildung 3.9 auf S. 26.

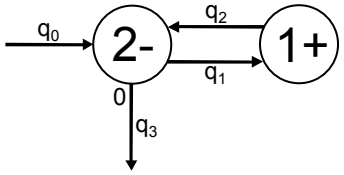


Abbildung 3.9: Das Summenprogramm, mit eingetragenen Zuständen

Abbildung 3.10 zeigt Schritt für Schritt die Berechnung der Summe von 5 und 3. Die Bilder des Programmgraphen mit den gekennzeichneten jeweiligen Zuständen des Abacus habe ich wiederum um der Anschaulichkeit willen eingefügt.

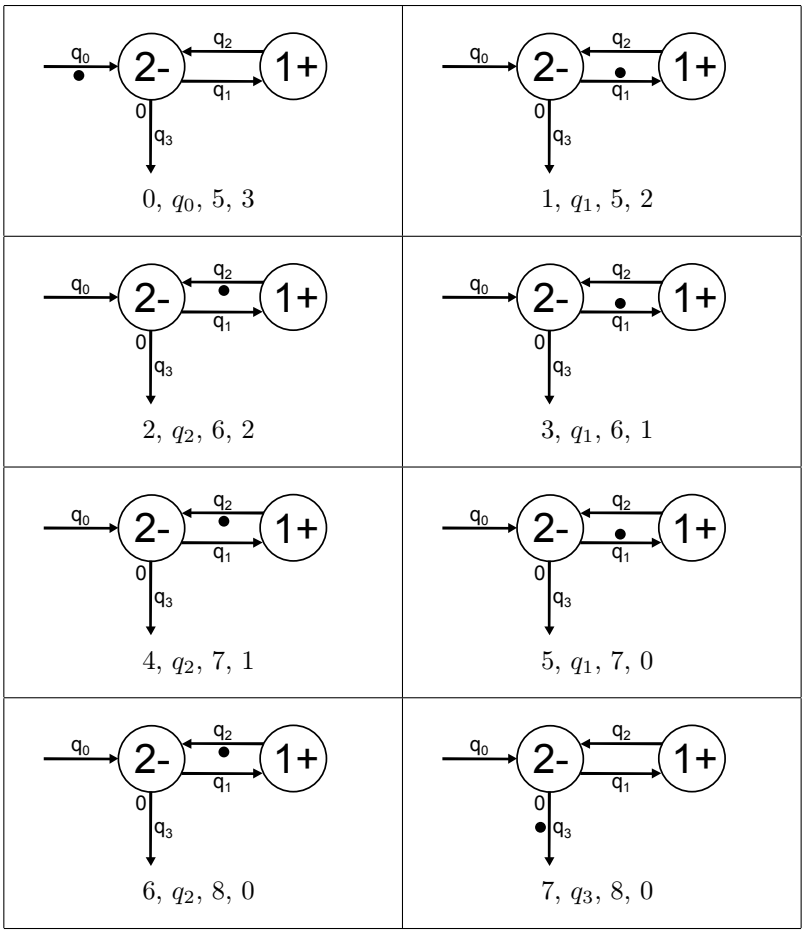


Abbildung 3.10: Berechnung einer Summe im Detail

☞ Es wäre eine reizvolle Übungsaufgabe, ein Programm zu zeichnen, das das

Produkt $y = x_1 \cdot x_2$ zweier natürlicher Zahlen berechnet.

Es gibt auch Programme, die konstante Funktionen berechnen, z.B. $f(x) = 3$. Dazu muss, egal welcher Input in Register Nr. 1 zu Beginn der Berechnung gespeichert ist, am Programmausgang die Zahl 3 im ersten Register gespeichert sein. Dazu wird der Inhalt des Register zunächst auf Null gesetzt, und dann wird sein Wert (in unserem Beispiel) dreimal um eins erhöht. In unserem Bild: Aus Register Nr. 1 werden alle Kugeln herausgenommen — egal ob welche und wenn ja, wie viele darin waren — und dann werden genau drei Kugeln hineingelegt. Abbildung 3.11 zeigt das Programm für unser Beispiel.

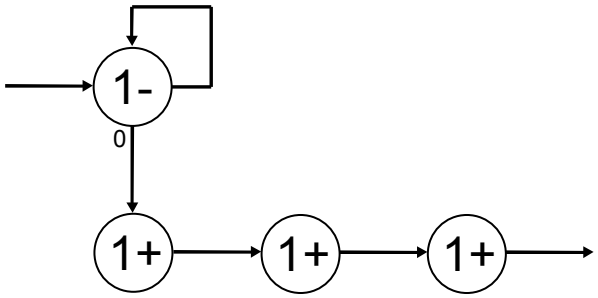


Abbildung 3.11: Abacus-Programm für eine konstante Funktion

3.4 Was der Abacus kann: die Church-Turing-These

Zunächst mag es scheinen, als ob der infinite Abacus eine ziemlich zahnlose Angelegenheit wäre: Er kann zwei Operationen mit Nachfolgern natürlicher Zahlen ausführen, in genau einem Fall im Programmablauf verzweigen und halten. Im Gegensatz zu unseren modernen Computern, mit denen wir im Internet surfen, Videos anschauen, uns sozial vernetzen, die zur Authentifizierung unsere Gesichter erkennen und die mit *Natural Language Processing* bereits einfachere gesprochene Sätze verarbeiten können, erscheint das sehr wenig. Doch der Eindruck täuscht, denn es zeigt sich, dass es keine berechenbare arithmetische Funktion gibt, die der Abacus nicht berechnen kann. Der Abacus ist vielmehr geradezu eine beispielhafte Präzisierung unseres Begriffs der Berechenbarkeit einer arithmetischen Funktion. Nach den beiden Pionieren, die diesen Begriff als erste präzisierten, heißt die Präzisierung

Die Church-Turing-These: Eine arithmetische Funktion ist berechenbar gdw. sie abacus-berechenbar ist.

Die Church-Turing-These ist eine Begriffsexplikation: Sie schärft unseren zunächst unscharfen Begriff der berechenbaren arithmetischen Funktion, und sie ist seine erste exakte Explikation. Die Mathematik*innen hatten selbstverständlich bereits jahrtausendelang gerechnet, ohne dass jedoch eine präzise Fassung dieses Begriffs notwendig geworden wäre, vgl. [Ber16, S. 11].

Abacus-Programme sind Anweisungen zur schrittweisen „Erzeugung“ von Funktionswerten. In präzisen Schritten formulierte Rechenanleitungen heißen nach dem arabischen Mathematiker Al-Chwarizmi (auch „Al-Khawarizmi“ u.a., um 800 n. Chr., vgl. z.B. [Her61, S. 28] oder [Nol97, S. 129, Fn. 3]) *Algorithmen*.

Es gibt viele, von ihrem Charakter her ganz unterschiedliche Explikationen des Begriffs der Berechenbarkeit; sie alle sind Typen von Algorithmen, weil sie schrittweise verfahren und jeder Schritt exakt definiert ist:

- **Abacus-Berechenbarkeit** (Joachim Lambek). Vgl. [Lam61]. Sie ist das bevorzugte Modell in diesem Kurs.
- **Turing-Berechenbarkeit** (Alan Turing). Vgl. [Tur36]. Unsere modernen Computer sind am ehesten endliche universelle Turing-Maschinen.
- **Definierbarkeit im Lambda-Kalkül** (Alonzo Church). Die exakte Formalisierung der Anwendung eines Funktionsausdrucks auf einen Argumentausdruck, vgl. [Chu36a]. Sehr abstrakt und daher zunächst etwas unanschaulich, vgl. [Ber16, S. 72].
- **Rekursivität** (Kurt Gödel, Stephen Cole Kleene). Gödel formuliert im Gang seines berühmten Beweises 46 rekursive Funktionen, vgl. [Göd86]. („Rekursiv“ nennt Gödel hier lediglich die primitiv-rekursiven Funktionen, die eine echte Unterklasse der allgemein-rekursiven Funktionen sind.)
- ... und noch einige andere.

Es zeigt sich allerdings, dass diese sehr verschiedenen Explikationen des Begriffs der berechenbaren arithmetischen Funktion sämtlich äquivalent sind. D.h. jede Funktion, die abacus-berechebar ist, ist auch Turing-berechenbar, ist auch im Lambda-Kalkül definierbar, ist auch rekursiv, etc. (Ein Beweis der Äquivalenz mehrerer solcher Algorithmentypen in Form einer zyklischen Inklusion findet sich in [BJ87, S. 52–96]; ein frühes Dokument der Entdeckung der Äquivalenz von Lambda-Definierbarkeit und Rekursivität ist [Chu36a, S. 355, Theoreme XVIf.].) Wir wollen, gemäß den ersten beiden Pionieren dieses Begriffs der Berechenbarkeit, sie *Church-Turing-Berechenbarkeit* nennen.

Bis heute wurde keine berechenbare Funktion gefunden, die nicht gemäß den genannten Algorithmentypen berechenbar wäre — und es wurde teilweise gezielt nach einer solchen gesucht. Die Vielzahl an sämtlich äquivalenten Algorithmentypen einerseits und das bisherige Scheitern der Versuche, eine berechenbare, aber nicht Church-Turing berechenbare arithmetische Funktion zu finden andererseits weisen darauf hin, dass hier etwas für Berechenbarkeit überhaupt Wesentliches gefunden wurde. So z.B. Hans Hermes in [Her61, S. 19, Hervorh. i. Orig.]:

> „Sehr bemerkenswert ist die Tatsache, daß man *rein mathematisch beweisen kann, daß die von sehr verschiedenen Ausgangspunkten herrührenden vorgeschlagenen Präzisierungen des Begriffs eines Algorithmus äquivalent sind.* Dies ist zumindest ein Hinweis darauf, daß es sich um einen fundamentalen Begriff handelt.“

Und Alonzo Church, über die Äquivalenz der Rekursivität und der Lambda-Definierbarkeit, in [Chu36a, S. 346, Fn. 3]:

> „The fact, however, that two such widely different and (in the opinion of the author) equally natural definitions of effective calculability turn out to be equivalent adds to the strength of the reasons adduced below for believing that they constitute as general a characterization of this notion as is consistent with the usual intuitive understanding of it.“

Das bedeutet: Auf dem Abacus können Sie jedes Computerprogramm laufen lassen! Wenn Sie geeignete Peripheriegeräte und Ein-/Ausgabemethoden vorsehen, z.B. einen Bildschirm, eine Tastatur, ein Zeigegerät, Netzwerkhardware etc., können Sie im Prinzip Abacus-Programme schreiben, um im Internet zu surfen, Ihren Lieblingsfilm abzuspielen, sich sozial zu vernetzen oder Ihre interessantesten Texte zu typographieren. Diese Abacus-Programme wären aufgrund des sehr kleinen Befehlssatzes des Abacus zwar unüberschaubar groß, und wenn wir die Annahme von der Dauer eines Schritts von einer Sekunde (s.o., S. 23) beibehalten wollen, unsäglich langsam in der Verarbeitung, aber Komplexitätsfragen sind bei diesen Erörterungen prinzipieller Möglichkeiten außen vor.

Anmerkung: Es macht keinen Sinn, nach einem Beweis für die Church-Turing-These zu fragen. Denn in ihr kommt der unscharfe Begriff der Berechenbarkeit vor, sodass ein präziser Beweis unmöglich ist — und es ist gerade dieser unscharfe Begriff, der als Abacus-Berechenbarkeit durch die Church-Turing-These präzisiert wird. So ist sie kein Metatheorem, sondern eine Begriffsexplikation, bei der wir nicht nach der Wahrheit, sondern nach ihrer Adäquatheit fragen sollten: Ist es angemessen, unseren zunächst vagen Begriff der Berechenbarkeit als Abacus-Berechenbarkeit zu verstehen? Wie aus den voraufgehende Ausführungen hoffentlich ein Stück weit hervorgegangen ist, sprechen sehr gute Gründe für die Adäquatheit der Church-Turing-These.

Kapitel 4

Das Halteproblem für den infiniten Abacus

Die Abacus-Programme, die wir bisher gesehen haben, haben sich sämtlich in einer ganz bestimmten Weise „manierlich“ benommen: Die Verarbeitung ist für jede natürliche Zahl als Argument, bzw. für jedes Paar natürlicher Zahlen beim Summenprogramm, irgendwann an einem Programmausgang angelangt. Da Programmausgänge dem Befehl „Halt!“ entsprechen, sagen wir: Die bisherigen Programme *halten* für alle Argumente.

4.1 Programme, die nicht immer halten

Doch was ist mit folgendem Programm?

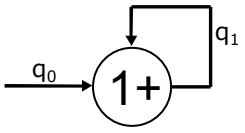


Abbildung 4.1: Ein Abacus-Programm, das nie hält

Dieses Programm hat zwei Zustände, den Programmeingang q_0 und einen weiteren Zustand q_1. Ist es einmal im Zustand q_1, und das ist nach dem ersten Zustandsübergang auch der Fall, dann verlässt es diesen Zustand nie wieder: Jeder weitere Zustandsübergang ist ein Übergang von q_1 nach q_1, bei dem der Inhalt des ersten Registers um eins erhöht wird. Und das geht „ewig“ so! D.h. dieses Programm hält für keine einzige natürliche Zahl, mit der als Argument in Register Nr. 1 eine Verarbeitung beginnt. (Das sieht man übrigens auch daran, dass das Programm keinen einzigen Programmausgang hat.)

Programme, die für alle Argumente halten (wie alle im voraufgehenden Kapitel) und Programme, die für kein einziges Argument halten (wie das aus Abbildung 4.1), sind die beiden Extremfälle, zwischen denen es auch Programme gibt, die für manche Argumente halten und für andere nicht. Ein recht einfaches Programm, das für manche Argumente hält und für andere nicht, ist das

folgende:

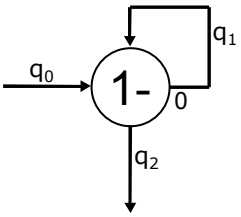


Abbildung 4.2: Ein Abacus-Programm, das hält für ...

☞ Können Sie sich vorstellen, für welche Argumente es hält, und für welche nicht? (Die Auflösung folgt sehr bald.)

Es sind also beipielsweise folgende Aussagen formulierbar:

1. Für das Nachfolgerprogramm:
   - Das Nachfolgerprogramm *hält* für den Input 0.
   - Das Nachfolgerprogramm *hält* für den Input 1.
   - Das Nachfolgerprogramm *hält* für den Input 2.
   - ...
2. Für das Programm aus Abbildung 4.1:
   - Das Programm *hält nicht* für den Input 0.
   - Das Programm *hält nicht* für den Input 1.
   - Das Programm *hält nicht* für den Input 2.
   - ...
3. Für das Programm aus Abbildung 4.2:
   - Das Programm *hält nicht* für den Input 0.
   - Das Programm *hält* für den Input 1.
   - Das Programm *hält* für den Input 2.
   - ...

Wir haben also eine zweistellige Relation vorliegen, zwischen je einem Programm einerseits und einer natürlichen Zahl, dem Input der Verarbeitung, andererseits:

Halt(..., ...): Das Programm ... hält für den Input ....

Die obigen Aussagen sind mittels des Prädikats ‚Halt' und geeigneter Individuenkonstanten für die Programme wie folgt repräsentierbar:

1. Für das Nachfolgerprogramm, wenn es durch die Individuenkonstante a repräsentiert wird:

- Halt$(a, 0)$
- Halt$(a, 1)$
- Halt$(a, 2)$
- ...

2. Für das Programm aus Abbildung 4.1, wenn es durch die Individuenkonstante b repräsentiert wird:
   - $\neg$Halt$(b, 0)$
   - $\neg$Halt$(b, 1)$
   - $\neg$Halt$(b, 2)$
   - ...
3. Für das Programm aus Abbildung 4.2, wenn es durch die Individuenkonstante c repräsentiert wird:
   - $\neg$Halt$(c, 0)$
   - Halt$(c, 1)$
   - Halt$(c, 2)$
   - ...

An dieser Stelle kann man fragen, ob die Frage, ob das Prädikat ‚Halt' auf ein geordnetes Paar aus einem Programm und einer Zahl zutrifft oder nicht, durch irgendeinen Algorithmus entscheidbar ist. Diese Frage ist das *Halteproblem für den infiniten Abacus*. Und da wir als einen (scheinbar universellen) Typ von Algorithmus den infiniten Abacus kennen, ist die Frage so formulierbar: Kann der infinite Abacus das Halteproblem für den infiniten Abacus berechnen? Die Frage ist also, ob das Halteproblem für den infiniten Abacus selbst abacusberechenbar ist.

Bevor wir uns an die Beantwortung dieser Frage machen, müssen wir noch zwei Dinge erledigen:

1. Da an der ersten Prädikatstelle von ‚Halt' ein Programm vorkommt, und der infinite Abacus nur Zahlen als Input verarbeiten kann, müssen wir uns überlegen, ob und wie man Abacus-Programme durch natürliche Zahlen darstellen („codieren") kann.
2. Mit dem Prädikat ‚Halt' werden Sätze gebildet, die entweder wahr oder falsch sind; der Abacus kann als Funktionswerte allerdings nur Zahlen liefern und keine Wahrheitswerte. Es ist also auch zu überlegen, ob und wie Wahrheitswerte durch natürliche Zahlen codierbar sind.

Wir werden das zweite Problem zuerst angehen.

4.2 Charakteristische Funktionen von Prädikaten

Man kann das Zutreffen von Prädikaten auf logische Individuen durch Funktionen modellieren: und zwar durch Funktionen, die genau dann Null sind („verschwinden", wie der/die Mathematiker*in sagt) für irgendwelche Argumente,

wenn das in Frage stehende Prädikat auf die Argumente zutrifft. Eine solche Funktion nennen wir die *charakteristische Funktion* des Prädikats.

Ein besonders einfaches Beispiel ist die Identitätsfunktion, die so aufgefasst eine charakteristische Funktion des Prädikats ‚... ist identisch mit 0' ist: Denn die Identitätsfunktion verschwindet für das Argument 0 und nur für dieses.

Charakteristische Funktion des Prädikats ‚... ist eine gerade Zahl' ist der Divisionsrest bei Teilung durch 2: Dieser ist 0 für gerade Zahlen und 1 für ungerade Zahlen. Damit verschwindet er gdw. die in Frage kommende Zahl gerade ist und ist somit charakteristische Funktion der Eigenschaft, eine gerade Zahl zu sein.

Für sehr viele interessante Prädikate sind die charakteristischen Funktionen berechenbar, für manche andere nicht. Klarerweise berechenbar ist die Identitätsfunktion. (☞ Können Sie sich ein Abacus-Programm vorstellen, das sie berechnet?) Auch die charakteristische Funktion der Eigenschaft, eine gerade Zahl zu sein, ist berechenbar; ein Programm dazu zeigt folgende Abbildung:

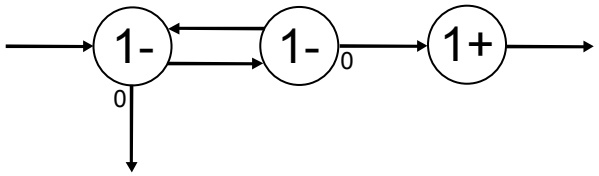


Abbildung 4.3: Abacus-Programm: gerade Zahl

Es hält mit 0 im ersten Register (dem Wert der charakteristischen Funktion, der verschwindet; am linken Programmausgang) gdw. sein Input gerade war, und mit 1 in Register Nr. 1 (und zwar am rechten Programmausgang) sonst (also wenn der Input ungerade war).

Wir können also definieren:

Definition 4.1. Sei P^n ein n-stelliges Prädikat ($n \geq 1$). Dann ist die n-stellige Funktion f_p *charakteristische Funktion von* P^n, wenn gilt:

$$f_P(t_1, \ldots, t_n) = \begin{cases} 0 & \text{falls } P^n(t_1, \ldots, t_n) \\ m \neq 0 & \text{falls } \neg P^n(t_1, \ldots, t_n) \end{cases}$$

Die Definition besagt: Die charakteristische Funktion f_P des Prädikats P ist auf genau dieselben Terme anzuwenden wie das Prädikat P, und der Wert von f_P ist 0, wenn das Prädikat P auf die Terme zutrifft, und eine von 0 verschiedene (natürliche) Zahl m sonst.

Damit haben wir eines der beiden Probleme vom Ende des voraufgehenden Abschnitts gelöst: Wir können jetzt Wahrheitswerte, die Prädikationen haben, durch die Werte arithmetischer Funktionen modellieren.

4.3 Wie viele Abacus-Programme gibt es?

Das andere Problem am Ende von Abschnitt 4.1 (s.o., S. 31) war, dass für eine etwaige (Abacus-)Berechnung des Halteproblems (genauer: der charakteristischen

Funktion des Halteproblems) für den infiniten Abacus keine Programme als Input für die Berechnung angegeben werden können, sondern nur natürliche Zahlen. Es ist also nach einer Methode zu suchen, wie Abacus-Programmen eindeutig irgendwelche natürlichen Zahlen zugeordnet werden können. Eine Frage, die sich auf dem Wege dieser Suche ergeben wird, ist: Wie viele Abacus-Programme gibt es? Und eine weitere wird sein: Wie viele berechenbare arithmetische Funktionen (beliebiger Stelligkeit) gibt es?

Recht leicht fällt die versuchsweise Antwort auf beide Fragen: „unendlich viele“. Denn es gibt alleine schon unendliche viele einstellige arithmetische Funktionen $\mathbb{N} \to \mathbb{N}$ — würden wir versuchen, sie in eine Liste zu schreiben, könnte deren Anfang etwa so aussehen:

1. $n \mapsto n$
2. $n \mapsto n + 1$
3. $n \mapsto n + 2$
4. $n \mapsto n + 3$

Diese Liste ließe sich beliebig fortführen, denn eine solche Funktion gibt es klarerweise für jede natürliche Zahl als zweiten, konstanten Summanden. Und wir haben in unserer Liste noch keine einzige Funktion, die eine Multiplikation enthält; wir haben nicht die Quadratfunktion und nicht die Funktion, die $n \in \mathbb{N}$ auf die n-te Primzahl abbildet.

Alleine schon konstante Funktionen gibt es unendlich viele:

1. $n \mapsto 0$
2. $n \mapsto 1$
3. $n \mapsto 2$
4. $n \mapsto 3$
5. ...

Auch von den zweistelligen arithmetischen Funktionen $\mathbb{N} \times \mathbb{N} \to \mathbb{N}$ kommt noch keine einzige auf unserer Liste vor. Von diesen gibt es wieder unendlich viele; der Anfang einer Liste der zweistelligen arithmetischen Funktionen könnte wie folgt aussgehen:

1. $(m, n) \mapsto m + n$
2. $(m, n) \mapsto m + n + 1$
3. $(m, n) \mapsto m + n + 2$
4. $(m, n) \mapsto m + n + 3$

Wieder ließe sich die Liste unendlich fortführen, mit beliebig großen Zahlen als dritten Summanden. Und wieder würde keine einzige Multiplikation vorkommen, keine Potenz-/Exponentialfunktion und was es sonst noch an zweistelligen arithmetischen Funktionen gibt.

Unser erster Verdacht könnte also sein: Es gibt weit mehr berechenbare arithmetische Funktionen als natürliche Zahlen.

Doch es kommt noch schlimmer: Denn zu jeder berechenbaren arithmetischen Funktion gibt es unendlich viele Programme, die sie berechnen! (Und nicht nur, wie man anfangs vielleicht meinen könnte, eines pro Funktion.)

Das ergibt sich aus folgender Überlegung: Wenn ein Programm einen Programmausgang hat, so kann man stets noch zwei Befehle an diesen anhängen, von denen der erste den Inhalt von Register Nr. 1 um eins erhöht und der zweite diesen sofort wieder um eins verringert. Dann ergeben sich zwar neue Programmausgänge, an denen aber der Inhalt des ersten Registers im Vergleich zum urprünglichen Programm unverändert ist. Ein so verändertes Programm berechnet offensichtlich dieselbe arithmetische Funktion wie das ursprüngliche Programm.

So berechnet auch das folgende Programm die Nachfolgerfunktion:

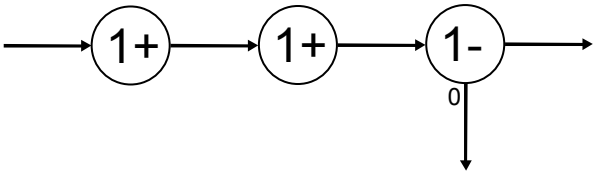


Abbildung 4.4: Nachfolgerprogramm (zugegeben, etwas blöd)

Lassen Sie sich nicht davon irritieren, dass Ihnen dieses Programm im Vergleich zu dem in Abbildung 3.4 auf S 23 vielleicht unnötig kompliziert oder gar unsinnig erscheint! Es berechnet, gemäß Konvention 3.1, die Nachfolgerfunktion für natürliche Zahlen. Und es ist von dem Programm in Abbildung 3.4 eindeutig verschieden: Das Programm hier hat drei Befehle und fünf Pfeile, während das Programm dort einen Befehl und zwei Pfeile hatte. (Man kann sich übrigens ganz leicht klarmachen, dass der mit ‚0' beschriftete Programmausgang nie erreicht wird.)

Und dieses Verfahren ist, wenn man es einmal hat, beliebig iterierbar (d.h. wiederholt anwendbar, auf sein eigenes Ergebnis), für jedes beliebige Programm mit mindestens einem Programmausgang. Auch das folgende Programm berechnet nämlich die Nachfolgerfunktion für natürliche Zahlen:

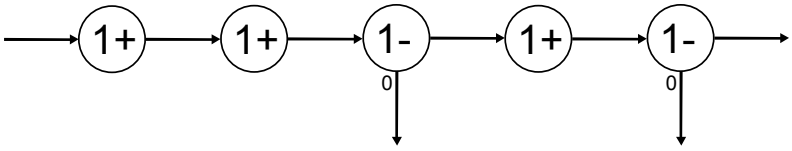


Abbildung 4.5: Nachfolgerprogramm (jetzt schon besonders blöd)

Wiederum ist es ein Programm für die Nachfolgerfunktion, und auch dieses ist von den beiden bisher gezeigten verschieden: Es hat noch mehr Befehle und Pfeile als alle beiden vorherigen. (Dass wohl niemand dieses Programm zur Berechnung des Nachfolgers einer natürlichen Zahl verwenden wird, da es ja das

wesentlich einfachere aus Abbildung 3.4 gibt, ist hier nicht von Belang. Es geht rein um die Existenz dieses Programms und darum, dass es die Nachfolgerfunktion berechnet.)

Fassen wir also zusammen. Es gibt

- unendlich viele berechenbare arithmetische Funktionen, und
- für jede berechenbare arithmetische Funktion unendlich viele Programme, die sie berechnen.

Dabei gibt es mindestens so viele berechenbare arithmetische Funktionen wie natürliche Zahlen. Es scheinen sogar weit mehr Funktionen zu sein als Zahlen. Und es gibt mindestens so viele Abacus-Programme wie berechenbare arithmetische Funktionen. Es scheinen also nochmals mehr Abacus-Programme zu sein als berechenbare arithmetische Funktionen, und damit erst recht mehr Programme als natürliche Zahlen. Die Suche nach einer Methode, jedem Abacus-Programm eine natürliche Zahl zuzuordnen, erscheint daher hoffnungslos! Jedoch, der Schein trügt ...

Schreiben wir F für die Menge der berechenbaren arithmetischen Funktionen und A für die Menge der Abacus-Programme (und verwenden wir wie üblich die senkrechten Striche um ein Mengensymbol als Funktionszeichen, das die Menge auf ihre Mächtigkeit, d.i. die Anzahl ihrer Elemente, abbildet), so lässt sich das Resultat der bisherigen Überlegungen kurz wie folgt anschreiben:

$$|\mathbb{N}| \leq |F| \leq |A|$$

Es wird aber im Folgenden gezeigt werden, dass

$$|A| \leq |\mathbb{N}|$$

ist, und somit kann wegen

$$|\mathbb{N}| \leq |F| \leq |A| \leq |\mathbb{N}|$$

(da dieselbe Mächtigkeit links und rechts am Ende der Kette vorkommt), nur sein, dass

$$|\mathbb{N}| = |F| = |A| = |\mathbb{N}|$$

D.h. es gibt gleich viele berechenbare arithmetische Funktionen und sogar gleich viele Abacus-Programme wie natürliche Zahlen! Unser erster Verdacht, dass es schon weit mehr berechenbare arithmetische Funktionen als natürliche Zahlen gibt, hat also getäuscht.

Im übernächsten Abschnitt werden wir zeigen, dass $|A| \leq |\mathbb{N}|$, ja noch stärker, dass $|A| = |\mathbb{N}|$, und zwar indem wir jedem Abacus-Programm eine natürliche Zahl zuordnen. Damit wird sich das erste Problem vom Ende des Abschnitts 4.1 (s.o., S. 31) erledigt haben, dass die charakteristische Funktion des Halteproblems natürliche Zahlen als Argumente braucht anstatt von Programm(graph)en.

4.4 Exkurs: „Mehr, als es natürliche Zahlen gibt“?

Vielleicht sind Sie, liebe/r Leser*in, am Ende des vorigen Abschnitts stutzig geworden: Ich hatte die Ungleichung

$$|\mathbb{N}| \leq |F| \leq |A|$$

angeschrieben und damit in Aussicht gestellt, dass es möglicherweise *mehr* berechenbare arithmetische Funktionen und auch *mehr* Abacus-Programme als natürliche Zahlen gibt. Natürliche Zahlen gibt es doch schon unendlich viele — wie kann es „mehr als unendlich viele“ geben; anders gefragt: Warum kollabiert das ‚$\leq$‘ nicht schon aufgrund der Unendlichkeit nicht zur Gleichheit, da es „mehr als unendlich viele“ ohnehin nicht geben kann?

Dass es durchaus guten Sinn machen kann, von einer Mächtigkeit zu sprechen, die größer ist als die der natürlichen Zahlen — zu sagen, dass es von manchen Entitäten „mehr gebe als natürliche Zahlen“ (meine Worte, M.M.) — das hat uns Georg Cantor in [Can32] vorgeführt. Dort ist sein berühmtes, nach ihm benanntes „Diagonalverfahren“ beschrieben. Da ich in diesem Text auch an einer anderen Stelle einen Beweis von der allgemeinen Struktur des Diagonalverfahrens anwenden werde, erläutere ich es an dieser Stelle kurz. (Ein hervorragender zusammenfassender Text über viele Beweise und Paradoxien mit dieser Struktur ist [Jac04].)

Cantor zeigt bei seinem Verfahren, dass es von den unendlichen Folgen, die aus einem Vorrat von mindestens zwei verschiedenen Zeichen bestehen, mehr gibt, als es natürliche Zahlen gibt: dass die Mächtigkeit der Menge der unendlichen Folgen aus einem Vorrat von mindestens zwei verschiedenen Zeichen größer ist als die Mächtigkeit der Menge der natürlichen Zahlen. Wir sagen, von den natürlichen Zahlen gebe es „abzählbar unendlich viele“, und von den Folgen der genannten Sorte gebe es „überabzählbar unendlich viele“. Überraschenderweise gibt es die Größer-Kleiner-Relation also auch in der Unendlichkeit: bei Mächtigkeiten unendlicher Mengen!

Machen wir die Beobachtung, dass die Dezimaldarstellungen der reellen Zahlen (die Menge der reellen Zahlen bezeichnen wir wie üblich mit ‚$\mathbb{R}$‘) unendliche Folgen sind, die aus einem Zeichenvorrat von mehr als zwei Zeichen gebildet werden, nämlich den Ziffern von 0 bis 9. (Reelle Zahlen, die eine endliche Dezimaldarstellung haben, können durch das explizite Anschreiben einer Periode von 0 in unendliche Folgen von Ziffern verwandelt werden.) Beschränken wir uns für das Weitere sogar nur auf die rellen Zahlen im Intervall $[0, 1)$, d.h. auf all jene, deren Dezimaldarstellung mit ‚0,‘ (*Null Komma*) beginnt.

Versuchen wir, eine unendlich lange, mit 1, 2, 3, ... (also mit den natürlichen Zahlen) nummerierte Liste der reellen Zahlen in diesem Intervall anzuschreiben. Gelingt es uns, alle rellen Zahlen in dem Intervall auf der Liste unterzubringen, so würden wir gezeigt haben: Reelle Zahlen (in besagtem Intervall) und natürliche Zahlen gibt es gleich viele. Es wird aber mindestens eine reelle Zahl übrigbleiben, die nicht auf unserer Liste vorkommen kann.

Diese tentative Liste braucht gar keine besondere Ordnung zu haben, und wir brauchen auch nur die ersten paar Dezimalstellen der ersten paar Zahlen zu kennen, um das Prinzip zu demonstrieren. Wir werden sehr bald sehen: Reelle Zahlen, alleine schon im Intervall $[0, 1)$, gibt es mehr, als es natürliche Zahlen

gibt. (Und wir werden kein Prinzip von der Art, wie es im folgenden Abschnitt für die Abacus-Programme vorgezeigt werden wird, zu Hilfe nehmen können: Es geht sich einfach nicht aus!)

Unsere Liste der reellen Zahlen könnte beispielsweise so beginnen (ich führe die ersten fünf Listenpositionen an):

1. 0,73528...
2. 0,41035...
3. 0,60982...
4. 0,15346...
5. 0,07808...

Dass es „mehr“ reelle Zahlen, alleine schon im Intervall $[0, 1) \subset \mathbb{R}$, gibt als natürliche Zahlen, werde ich dadurch vorführen, dass ich für unsere Liste gleich eine Zahl konstruiere, die an keiner Position in ihr vorkommen kann: nicht an der ersten Position, nicht an der zweiten Position, nicht an der dritten, ...

Dazu sehe ich mir zunächst nur die erste Dezimalstelle der ersten Zahl an: Sie ist 7. Wenn ich meine neue Zahl so konstruiere, dass an ihrer ersten Dezimalstelle eine *andere* Ziffer vorkommt als die 7, dann unterscheidet sich meine neue Zahl bereits ganz sicher von der ersten Zahl auf unserer Liste: nämlich zumindest in ihrer ersten Dezimalstelle.

Dann sehe ich mit die zweite Dezimalstelle der zweiten Zahl an: Sie ist die 1. Wenn ich meine neue Zahl so konstruiere, dass an ihrer zweiten Dezimalstelle eine *andere* Ziffer vorkommt als die 1, dann unterscheidet sich meine neue Zahl auch ganz sicher von der zweiten Zahl auf unserer Liste: nämlich zumindest in ihrer zweiten Dezimalstelle.

Dieses Vorgehen mache ich nun zur Regel: Ich schreite zunächst die *Diagonale* der Liste der Zahlen ab.

1. 0,$\underline{7}$3528...
2. 0,4$\underline{1}$035...
3. 0,60$\underline{9}$82...
4. 0,153$\underline{4}$6...
5. 0,0780$\underline{8}$...

Ich finde die Ziffern 7, 1, 9, 4, 8, ... Dann konstruiere ich meine neue Zahl so, dass sie mit ‚0,‘ (*Null Komma*) beginnt, und dann an ihrer ersten Dezimalstelle *nicht* die 7 vorkommt, an ihrer zweiten Dezimalstelle *nicht* die 1, an ihrer dritten Dezimalstelle *nicht* die 9, usw. (Um unangenehme Überraschungen zu vermeiden, die die Mehrdeutigkeit der Dezimaldarstellung eventuell mit sich bringt, vermeide ich stets die Verwendung der Ziffer 9.)

Beispielsweise könne meine neue Zahl so beginnen: 0,53672.... — Nennen wir eine auf die beschriebene Weise zu einer Liste von Dezimalzahlen konstruierte Zahl die *Diagonalzahl*.

Damit weicht, für jede natürlich Zahl i, die Diagonalzahl an ihrer i-ten Dezimalstelle von der Zahl an der i-ten Listenposition ab. Somit kann die Diagonalzahl an keiner Position in der Liste vorkommen.

Damit gilt: Eine mit 1, 2, 3, ... nummerierte Liste der reellen Zahlen, alleine schon im Intervall $[0, 1) \subset \mathbb{R}$, kann niemals vollständig sein. Denn sofort ist zu jeder solchen Liste die Diagonalzahl konstruierbar, die zwar eine reelle Zahl im

Intervall $[0, 1)$ ist, jedoch gerade aufgrund ihrer Konstruktionsweise auf der Liste nicht vorkommen kann.

(Und es würde gar nichts helfen, wenn wir zu irgendeiner Liste von Dezimalzahlen die Diagonalzahl konstruieren würden und die Sache dann dadurch zu retten versuchten, dass wir die Diagonalzahl an irgendeiner Position in unsere ursprüngliche Liste einfügten: Denn dann würde eine neue Liste von Dezimalzahlen entstehen, zu der gleich wieder, auf die soeben beschriebene Weise, eine neue Diagonalzahl konstruierbar wäre ....)

Die Menge der reellen Zahlen hat also tatsächlich eine größere Mächtigkeit als die Menge der natürlichen Zahlen, obwohl diese bereits unendlich viele Elemente hat. Es gilt also: $|\mathbb{N}| < |\mathbb{R}|$. — Die Rede am Ende des letzten Abschnitts davon, dass es möglicherweise „mehr“ Abacus-Programme als natürliche Zahlen gibt, d.h. dass auch die in $|\mathbb{N}| \leq |A|$ enthaltene Möglichkeit $|\mathbb{N}| < |A|$ wahr sein könnte, erscheint jetzt nicht mehr so absurd wie vielleicht noch am Anfang dieses Abschnitts.

Halten wir einige wesentliche Merkmake unserer Methode, die Diagonalzahl zu konstruieren, fest. (Sie werden uns am Ende dieses Kapitels erneut begegnen, s.u., S. 53.)

1. Wir sind die Diagonale unserer Liste abgeschritten, d.h. wir haben von der Zahl an der Listenposition i die i-te Dezimalstelle betrachtet (klarerweise ist $i \in \mathbb{N}^+$). Bezeichnen wir die Zahl an der Listenposition i mit a_i und ihre j-te Dezimalstelle mit a_{ij} (auch $j \in \mathbb{N}^+$), so haben wir uns besonders für die Folge $a_{11}, a_{22}, a_{33}, \ldots, a_{ii}, \ldots$ interessiert.

2. Wir haben unsere Diagonalzahl, nennen wir sie d und ihre i-te Dezimalstelle d_i, so konstruiert, dass $d_i \neq a_{ii}$ für alle $i \in \mathbb{N}^+$.

3. Wir haben, um des indirekten Beweises willen, angenommen, dass unsere Diagonalzahl an irgendeiner Listenposition $j \in \mathbb{N}^+$ vorkommen könnte,

   - d.h. es gibt ein $j \in \mathbb{N}^+$, sodass $d = a_j$.
   - Es gilt aber für alle $i \in \mathbb{N}^+$: $d_i \neq a_{ii}$, gemäß Punkt (2).
   - Diese beiden Behauptungen zusammengenommen bedeuten: Es gibt ein $j \in \mathbb{N}^+$, sodass für alle $i \in \mathbb{N}^+$ gilt: $a_{ji} \neq a_{ii}$. (D.h. Die i-te Stelle der Zahl an der Listenposition j — unserer Diagonalzahl — ist verschieden von der i-ten Stelle der Zahl an der Listenposotion i.)
   - Daraus folgt aber: Es gibt ein $j \in \mathbb{N}^+$, sodass $a_{jj} \neq a_{jj}$. Das ist ein Widerspruch!

4. Also gibt es kein $j \in \mathbb{N}^+$, sodass unsere Diagonalzahl an Listenposition j vorkommen könnte, d.h. dass es gibt kein $j \in \mathbb{N}^+$, sodass $d = a_j$. Die Diagonalzahl kommt mithin auf unserer Liste nicht vor.

Punkte (1) und (3) sind die *Selbstanwendung*: In Nr. (1) wird sie vorbereitet, insofern, als für unsere Betrachtung jeweils die i-te Dezimalstelle der i-ten Zahl (für $i \in \mathbb{N}^+$) herausgefiltert wird.

Punkt (2) ist die Bildung eines *Komplements*: Wir konstruieren unsere Diagonalzahl so, dass sie von dem in Punkt (1) Herausgefilterten in jeder Stelle abweicht.

Punkt (3) schließlich fügt die Punkte (1) und (2) in der „komplementierenden Selbstanwendung“ (meine Übersetzung von Jacquettes „*self-non-application*“, [Jac04, S. 70], M.M.) zusammen. Das ist nach [Jac04, S. 67–72] das doppelte Merkmal jeder Diagonalisierung: „I. There must be a *self-application*, whereby the designation of an item in an array involves a term that applies specifically to itself.“ [Jac04, S. 67, Hervorh. i. Orig.] — sowie: „II. There must be an alteration, *denial*, *internal negation* or *imposition of the complement of a term, predicate* or *proposition*, in a self-applicational expression satisfying requirement I.“ [Jac04, S. 70, Hervorh. i. Orig.]

In Punkt (3) haben wir für den indirekten Beweis angenommen, dass wir unsere Diagonalzahl auf der Liste an irgendeiner Position j unterbringen könnten, also dass $d_i = a_j$. Die Diagonalzahl d war nun so konstruiert, das für alle $i \in \mathbb{N}^+$ gilt: $d_i \neq a_{ii}$. Diese beiden Annahmen zusammen ergeben die logisch falsche Quantifikation

$$\exists j \forall i \, a_{ji} \neq a_{ii}$$

denn aus ihr folgt unmittelbar

$$\exists j \, a_{jj} \neq a_{jj}$$

D.h. welches $j \in \mathbb{N}^+$ auch immer die Listenposition unserer Diagonalzahl wäre, dann wäre die Ziffer an ihrer j-ten Dezimalstelle verschieden von sich selbst. (Und das kann natürlich nicht sein.)

Die obige logisch falsche Quantifikation ist die Negation einer Instanz von Thomsons Theorem (vgl. [Tho62, S. 104]). Das Theorem ist an der angegebenen Stelle zwar wortsprachlich formuliert ist, kann aber offenkundig formalisiert werden zu:

Theorem 4.1 (Thomson)**.** $\neg\exists y \forall x (R(y,x) \leftrightarrow \neg R(x,x))$

Beweis.

1.	$\neg\neg\exists y \forall x (R(y,x) \leftrightarrow \neg R(x,x))$	(IB-Annahme)
2.	$\exists y \forall x (R(y,x) \leftrightarrow \neg R(x,x))$	1., (DN2)
3.	$\forall x (R(a,x) \leftrightarrow \neg R(x,x))$	(EB-Annahme)
4.	$(R(a,a) \leftrightarrow \neg R(a,a))$	3., (UB)
5.	$(R(a,a) \rightarrow \neg R(a,a))$	4., (ÄQ-ELIM1)
6.	$(\neg R(a,a) \rightarrow R(a,a))$	4., (ÄQ-ELIM2)
7.	$R(a,a)$	(FU-Annahme 1)
8.	$\neg R(a,a)$	7., 5., (MP)
9.	$(F(b) \wedge \neg F(b))$	7., 8., (ECQ)
10.	$\neg R(a,a)$	(FU-Annahme 2)
11.	$R(a,a)$	10., 6., (MP)
12.	$(F(b) \wedge \neg F(b))$	11., 10., (ECQ)
13.	$(F(b) \wedge \neg F(b))$	7.–12., (FU)
14.	$(F(b) \wedge \neg F(b))$	3.–13., (EB)
15.	$\neg\exists y \forall x (R(y,x) \leftrightarrow \neg R(x,x))$	1.–14., (IB)

□

Ein bekanntes Beispiel ist das Barbier-Paradoxon, das sich ergibt, wenn wir die Relation ‚$R(\ldots,\ldots)$‘ lesen als ‚… rasiert …‘: Es ist logisch ausgeschlossen, dass es einen (nämlich y) gibt, der all jene und nur jene x rasiert, die sich

nicht selbst rasieren. — Und eine Instanz von Thomsons Theorem, nämlich mit ‚$a_{yx} = a_{xx}$‘ für die Relation ‚$R(y, x)$‘, ist:

$$\neg\exists y\forall x\, a_{yx} \neq a_{xx}$$

Denn die Setzung ergibt $\neg\exists y\forall x(a_{yx} = a_{xx} \leftrightarrow a_{xx} \neq a_{xx})$. Das rechte Glied der Äquivalenzformel ist als negierte Selbstidentität logisch falsch, und eine Äquivalenzformel mit einem logisch falschen Glied ist äquivalent mit der Negation ihres anderen Gliedes.

4.5 Gödelisierung der Abacus-Programme

Jetzt wird eine Methode vorgestellt, wie jedem Abacus-Programmgraphen eine natürliche Zahl zugeordnet wird. Dabei ist die vorgezeigte Methode lediglich eine von vielen; ich habe bei der Ausarbeitung meiner Methode Wert darauf gelegt, dass man aus der zugeordneten natürlichen Zahl möglichst einfach das Programm wiedergewinnen kann, zu dem sie gehört.

Diese Methode ist eine Erfindung (oder „Entdeckung“, wenn Sie Platonist*in sind) Kurt Gödels, der sie erstmals in [Göd86] anwandte, und heißt nach ihm „Gödelisierung“. Wir machen uns also an die Gödelisierung unserer Abacus-Programmgraphen; die natürliche Zahl, die einem Programm zugeordnet wird (man sagt auch: die ein Programm „codiert“), heißt seine „Gödelzahl“.

Wir konstruieren zu einem Programm seine Gödelzahl, indem wir zuerst die Knoten (Befehle) des Programms codieren, und danach seine Kanten (Pfeile).

Dazu nummerieren wir als erstes die Befehle des Programms mit ‚1‘ beginnend durch, wobei dem Befehl unmittelbar nach dem Programmeingang die ‚1‘ zugeordnet wird. (Bei der Nummerierung der anderen Befehle haben wir etwas Freiheit, in welcher Reihenfolge wir ihre Nummern vergeben.) Diese Zahl heiße „die laufende Nummer des Befehls“.

1. Codierung der Befehle (Knoten). Für jeden Befehl erhalten wir seine Kennziffer, indem wir unmittelbar aufeinander folgend schreiben:
   (a) so viele ‚1‘-en, wie die laufende Nummer des Befehls angibt; danach
   (b) so viele ‚2‘-en, wie die Registernummer des Befehls angibt; und schließlich
   (c) eine ‚3‘, falls es sich um einen ‚+‘-Befehl handelt, und ‚33‘, falls es sich um einen ‚−‘-Befehl handelt.
2. Codierung der Pfeile (Kanten). Für jeden Pfeil erhalten wir seine Kennziffer, indem wir (genau einmal) unmittelbar aufeinander folgend schreiben:
   (a) eine ‚4‘ als Kennzeichen dafür, dass es sich um die Codierung eines Pfeils handelt; dann
   (b) eine ‚0‘, falls es sich um einen mit ‚0‘ beschrifteten Pfeil handelt und gar nichts sonst; dann
   (c) so viele ‚1‘-en, wie die laufende Nummer des Befehls angibt, von dem der Pfeil ausgeht oder gar nichts, wenn der Pfeil der Programmeingang ist; dann

   (d) eine ‚8' als Trennzeichen; und schließlich
   (e) so viele ‚1'-en, wie die laufende Nummer des Befehls angibt, zu dem der Pfeil hinfürt oder gar nichts, wenn der Pfeil ein Programmausgang ist.
3. Die so erhaltenen Ziffern ordnen wir der Größe nach, und
4. schreiben sie in dieser Reihenfolge jeweils mit einer ‚9' als Trennzeichen dazwischen an.
5. Die Zahl, deren Dezimaldarstellung auf diese Weise gewonnen wurde, ist die Gödelzahl des Abacus-Programms.

Es wird sicherlich gut sein, wenn wir uns dazu Beispiele ansehen. Ermitteln wir die Gödelzahl des Programms für die Nachfolgerfunktion aus Abbildung 3.4 auf S. 23:

- Der einzige Befehl des Programms erhält die laufende Nummer 1.
- Dieser Befehl hat also die laufende Nummer 1, wir beginnen zu schreiben: *eine* ‚1'.
- Er betrifft Register Nr. 1, also schreiben wir weiter: *eine* ‚2'. Wir haben inzwischen ‚12'.
- Er erhöht den Inhalt des Registers, das er betrifft, also schreiben wir weiter ‚3' und haben als Befehlskennziffer des einzigen Befehls in unserem Programm: ‚123'.
- Wir codieren die beiden Pfeile in unserem Programm; wir beginnen mit dem Programmeingang.
- Wir schreiben, da es sich um eine Pfeilkennziffer handelt, ein ‚4'.
- Der Pfeil ist ein Programmeingang, also schreiben wir zunächst gar nichts und bleiben bei ‚4'.
- Wir schreiben eine ‚8' als Trennzeichen und haben inzwischen ‚48'.
- Der Pfeil führt zu dem Befehl mit der laufenden Nummer 1, also schreiben wir eine ‚1'und haben ‚481' als Kennziffer des Programmeingangs.
- Wir codieren den Programmausgang: Wir schreiben zunächst, da es sich um eine Pfeilkennziffer handelt, ein ‚4'.
- Der Pfeil geht dem Befehl mit der laufenden Nummer 1 aus, also schreiben wir eine ‚1'und kommen zu ‚41'.
- Wir schreiben eine ‚8' als Trennzeichen und haben inzwischen ‚418'.
- Der Pfeil ist ein Programmausgang, also schreiben wir gar nichts und haben ‚418' als Kennziffer des Programmausgangs.
- Wir haben nun die Kennziffern 123, 481 und 418.
- Wir ordnen sie der Größe nach: 123, 418, 481.

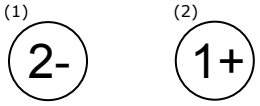
- Wir schreiben sie, jeweils mit einer ‚9' als Trennzeichen dazwischen an: 12394189481. Das ist die Gödelzahl des Nachfolgerprogramms.

Umgekehrt kann man von jeder natürlichen Zahl eindeutig in endlich vielen Schritten feststellen, ob sie Gödelzahl eines Abacus-Programms ist, und wenn ja, das Programm eindeutig aus ihr rekonstruieren.

Ist beispielsweise die Zahl 1239401181594181 Gödelzahl eines Abacus-Programms? — Nein, denn in ihr kommt die Ziffer ‚5' vor, und diese kommt gemäß unserer Gödelisierungsvorschrift in keiner Gödelzahl eines Abacus-Programmgraphen vor.

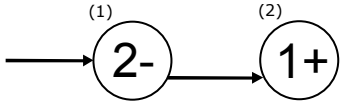
Wie steht es mit der Zahl 48194018911239122339411819411811?

- Trennen wir sie zunächst an den ‚9'-en auf. Wir bekommen die Kennziffern ‚481', ‚4018', ‚1123', ‚12233', ‚41181', ‚41811'.
- Davon sind Befehlskennziffern: ‚12233', ‚1123'.
- Die übrigen, die mit einer ‚4' beginnen, sind Pfeilkennziffern: ‚481', ‚4018', ‚41181', ‚41811'. Es handelt sich also, wenn es sich überhaupt um ein Profgramm handelt, um ein Programm mit zwei Befehlen und vier Pfeilen.
- Der Befehl mit der Kennziffer ‚12233' hat die laufende Nummer 1 (er folgt also unmittelbar auf den Programmeingang), er betrift Register Nr. 2, und er verringert dessen Inhalt um 1.
- Der andere Befehl, mit der Kennziffer ‚1123' hat die laufende Nummer 2, er betrifft Register Nr 1, und er erhöht dessen Inhalt um 1. Wir können also zeichnen:

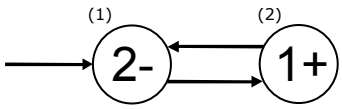


- Der Pfeil mit der Kennziffer ‚481' ist nicht beschriftet; er ist der Programmeingang, und er führt zu dem Befehl mit der laufenden Nummer 1.

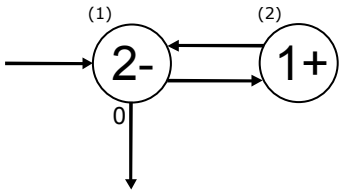
- Der Pfeil mit der Kennziffer ‚41811' ist nicht beschriftet, und er führt von dem Befehl mit der laufenden Nummer 1 zu dem Befehl mit der laufenden Nummer 2.



- Der Pfeil mit der Kennziffer ‚41181' ist nicht beschriftet, und er führt von dem Befehl mit der laufenden Nummer 2 zu dem Befehl mit der laufenden Nummer 1.



- Der Pfeil mit der Kennziffer ‚4018' ist mit ‚0' beschriftet; er ist der Programmausgang, und er führt weg von dem Befehl mit der laufenden Nummer 1.



Die Zahl 48194018911239122339411819418 11 codiert also ein Abacus-Programm — und zwar kein anderes als unser Summenprogramm aus Abbildung 3.8 auf S. 25.

Wir können nun die Abacus-Programme der Größe ihrer Gödelzahlen nach anordnen (vgl. [Chu36a, S. 353, Theorem V]): P_0 (Programm Nr. 0) ist das Programm mit der kleinsten Gödelzahl, P_1 (Programm Nr. 1) ist das Programm mit der zweitkleinsten Gödelzahl, P_2 (Programm Nr. 2) hat die nächstgrößere Gödelzahl, usw. Damit haben wir jedem Abacus-Programm eine natürliche Zahl zugeordnet. (Gleichzeitig haben wir, vielleicht überraschenderweise, gezeigt, dass es gleich viele Abacus-Programme wie natürliche Zahlen gibt, und daher auch höchstens so viele berechenbare arithmetische Funktionen.)

Die ersten paar Abacus-Programme, geordnet nach ihren Gödelzahlen, zeigt Abbildung 4.6 auf S. 46. (Hätten wir Konvention 3.2 auf S. 23 nicht, so gäbe es ein Programm, das nur den Inhalt von Register Nr. 2 erhöht und dann hält, also ein recht uninteressantes Programm für die Identitätsfunktion, zwischen P_0 und P_1.)

Jetzt sind Aussagen über Programme — und das ist der zweite Teil einer Gödelisierung — als Aussagen über Zahlen möglich. Ändern wir unser Prädikat ‚Halt' ein wenig ab, sodass jetzt auch an seiner ersten Stelle natürliche Zahlen vorkommen können:

Halt(. . . , . . .): Das Programm Nr. . . . hält für den Input . . . .

So ist z.B. die Aussage ‚Programm Nr. 0 (also das mit der kleinsten Gödelzahl 12394189481) hält für den Input 7' jetzt mit unserem Prädikat ‚Halt' formulierbar als

$$\mathrm{Halt}(0, 7)$$

Und die Aussage ‚Programm Nr. 4 (also das mit der fünftkleinsten Gödelzahl 418948191233940181) hält nicht für den Input 0':

$$\neg\mathrm{Halt}(4, 0)$$

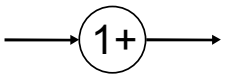
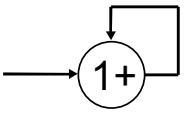
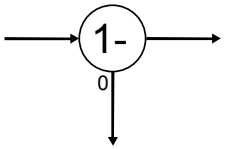
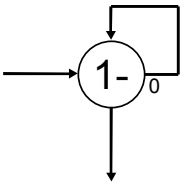
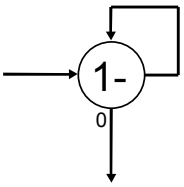
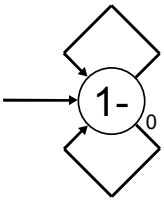
P_0	 12394189481	Nachfolger
P_1	 123948194181	vgl. Abb. 4.1
P_2	 12339418948194018	Vorgänger
P_3	 4189481912339401 81	vgl. Abb. 4.2
P_4	 4819123394018941 81	$x \mapsto 0$
P_5	 4819123394181940181	hält nie

Abbildung 4.6: Die Abacus-Programme mit den kleinsten Gödelzahlen

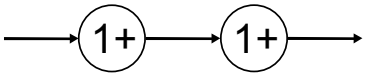
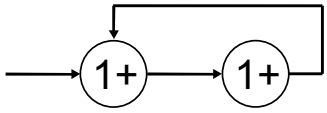
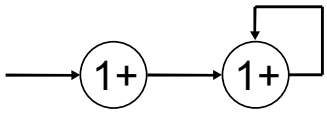
P_6	 1239481911239411894181 1	$n \mapsto n+2$
P_7	 1239481911239411819418 11	hält auch nie
P_7	 1239481911239418119411811	hält auch nie

Abbildung 4.7: Noch ein paar Abacus-Programme mit „kleinen“ Gödelzahlen

Allgemein sind jetzt Aussagen der Art ‚Programm Nr. p hält für den Input i‘ formalisierbar als

$$\mathrm{Halt}(p, i)$$

Wenn wir die Wahrheitswerte, die diese Prädikationen haben, gemäß Definition 4.1 in Abschnitt 4.2 (s.o., S. 34) durch natürliche Zahlen modellieren, so bekommen wir die charakteristische Funktion des Halteproblems:

$$\mathrm{halt}(p, i) = \begin{cases} 0 & \text{falls } \mathrm{Halt}(p, i) \\ 1 & \text{falls } \neg\mathrm{Halt}(p, i) \end{cases}$$

Diese Funktion verschwindet gdw. das Programm Nr. p für den Input i hält. Sie ist eine arithmetische Funktion, d.h. sie hat natürliche Zahlen als Argumente und als Werte: Wir haben das Halteproblem arithmetisiert — Gödelisierung ist Arithmetisierung der Metatheorie — und können damit unsere Frage aus Abschnitt 4.1 von S. 33 so formulieren:

Ist die Funktion ‚halt**‘ abacus-berechenbar?**

Ihre Beantwortung wird uns im nächsten Abschnitt beschäftigen.

4.6 Das Halteproblem ist nicht berechenbar

In diesem Abschnitt wird die Frage nach der (Abacus-)Berechenbarkeit des Halteproblems ihre Beantwortung erfahren, und zwar ihre negative: Das Halteproblem für den Abacus ist nämlich nicht berechenbar. Um dies zu beweisen, werden wir indirekt vorgehen, d.h. zunächst das Gegenteil unserer zu beweisenden Annahme annehmen und diese auf einen Widerspruch zurückführen. Wenn das

gelungen ist, so werden wir wissen: Das Gegenteil unserer Annahme kann nicht wahr sein, da aus ihr ein Widerspruch folgt — also muss sie wahr sein.

Die Annahme, die wir auf einen Widerspruch zurückführen werden, ist:

Das Halteproblem ist berechenbar. (Das wird sich am Ende als falsch herausstellen.)

Nehmen wir dies einmal an. Dann ist das Halteproblem, aufgrund der Church-Turing-These (s.o., S. 27), auch abacus-berechenbar. Dann aber gibt es ein Abacus-Programm, das die Funktion ‚halt' berechnet. Nennen wir es P_{halt}.

Die Interna dieses Programms P_{halt} brauchen uns gar nicht weiter zu interessieren, wir müssen nur wissen:

- **Es hat genau einen Programmeingang.** Das muss so sein, denn jedes Abacus-Programm hat genau einen Programmeingang. Dort stehen die beiden Argumente für die Berechnung des Funktionswertes in den ersten beiden Registern, gemäß Konvention 3.1.
- **Es hat mindestens einen Programmausgang** — oder auch mehrere. Da das Programm eine zweistellige arithmetische Funktion berechnet, muss es gemäß Konvention 3.1 irgendwann mit dem Funktionswert im ersten Register an irgendeinem Programmausgang halten.

Das Schema dieses Programms zeigt folgende Abbildung (achten Sie auf den einen Programmeingang links oben und die Programmausgänge unten):

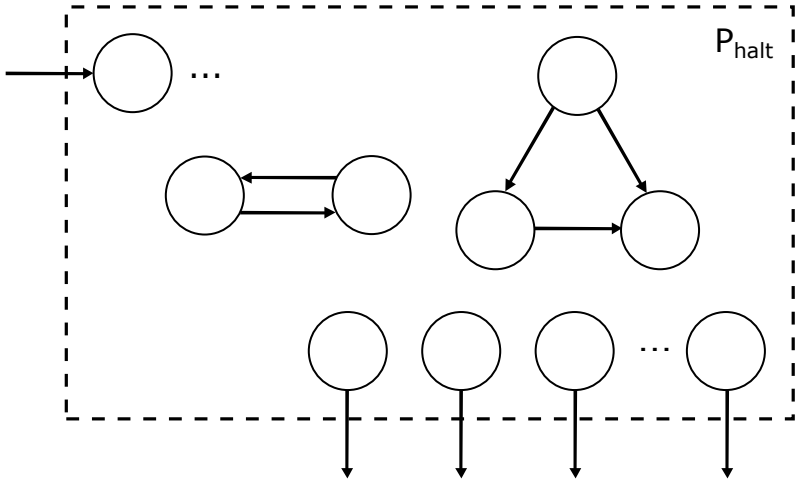


Abbildung 4.8: Halt!?

Jetzt konstruieren wir aus diesem Programm ein neues, und zwar wenden wir zwei Tricks an:

Wir machen zuerst aus unserer zweistelligen Funktion ‚halt' eine einstellige Funktion, und zwar dadurch, dass das erste und das zweite Argument der Funktion stets identisch gesetzt werden. Wir definieren also die Funktion $\text{halt}^1(n) =_{\text{df}} \text{halt}(n, n)$ für jede positive natürliche Zahl n. Das dazugehörige Abacus-Programm konstruieren wir, indem wir dem Programm P_{halt} ein anderen Programm „vorschalten", und zwar kopiert dieses das erste Register in das zweite. Abbildung 4.9 zeigt dieses „Kopierprogramm".

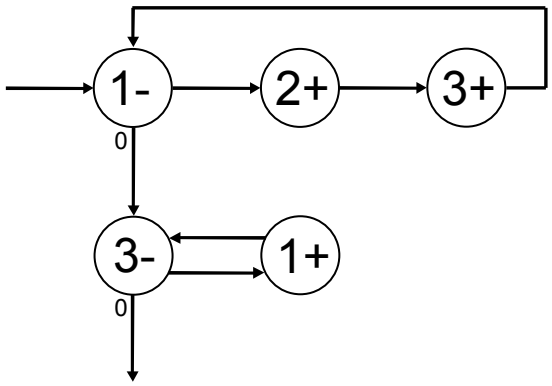


Abbildung 4.9: Register Nr. 1 kopieren

Es funktioniert wie folgt: Zunächst wird der Inhalt von Register Nr. 1 in die beiden Register Nr. 2 und 3 übernommen. Register Nr. 1 ist am Ende dieser Operation leer. Daher wird der Inhalt von Register Nr. 3 wieder in Register Nr. 1 übernommen, und Register Nr. 3 ist dann leer. In den beiden Registern Nr. 1 und 2 steht nun dieselbe Zahl, nämlich die, mit der in Register Nr. 1 die Verarbeitung begonnen hat.

Durch Vorschaltung dieses „Kopierprogramms“ resultiert das Programm P^1_{halt}, vgl. Abbildung 4.10. Was berechnet P^1_{halt}? Es berechnet die Funktion $\text{halt}^1(n)$ für alle positiven natürlichen Zahlen n, d.i. nichts anderes als die Funktion $\text{halt}(n, n)$. Wir erinnern uns:

$$\text{halt}(n,n) = \begin{cases} 0 & \text{falls Halt}(n,n) \\ 1 & \text{falls } \neg\text{Halt}(n,n) \end{cases}$$

Das bedeutet, Programm P^1_{halt} berechnet, ob Programm Nr. n für die Zahl n hält oder nicht: ob Programm Nr. n für seine eigene Positionsnummer in der Liste der Abacus-Programme hält oder nicht.

Wir schalten dem Programm P^1_{halt} — und das ist der zweite Trick — noch ein kleines Programm nach, und zwar das Programm P_3, das uns in Abbildung 4.2 auf S. 32 zuerst begegnet ist. (In einer Veranstaltung im Jahre 2005 hat ein scharfsinniger Geist das Programm P_3 in dieser Funktion einmal den „Paradoxator“ getauft — es wird sich gleich zeigen, warum.) Jeder Pfeil, der bei P^1_{halt} (und auch bei unserem ursprünglichen Programm P_{halt}) ein Programmausgang war, zeigt jetzt auf den durch diese Operation neu hinzugekommenen Befehl. Die Zustände, die diesen Pfeilen (den alten Programmausgängen, wo nunmehr das nachgeschaltete kleine Programm anschließt) entsprechen, wollen wir unter der Bezeichnung „der kritische Punkt“ zusammenfassen.

So sind wir wieder zu einem neuen Programm gekommen; nennen wir es P^*_{halt}. Abbildung 4.11 zeigt es.

Betrachten wir nun die Fälle, in denen P^*_{halt} hält.

Zunächst wird für irgendeine natürliche Zahl die Funktion $\text{halt}(n, n)$ berechnet. Ihr Funktionswert ist 0 gdw. das Programm Nr. n für den Input n hält. Wenn also die Berechnung des Funktionswertes fertig ist — am „kritischen

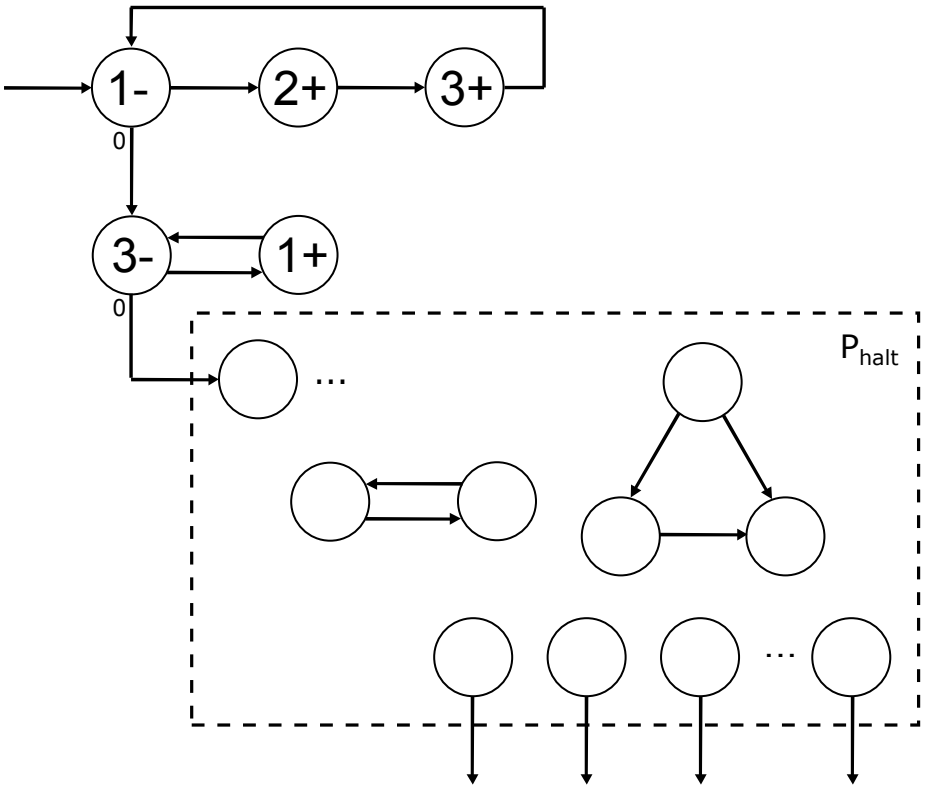


Abbildung 4.10: Halten für die eigene Listenposition

Punkt“ — beginnt das nachgeschaltete kleine Programm zu arbeiten: Wenn die Berechnung von $\text{halt}(n, n)$ den Wert 0 ergeben hat, so bleibt es auf dem Nullpfeil „hängen“, und der Programmausgang wird nie erreicht. Wenn die Berechnung von $\text{halt}(n, n)$ dagegen den Wert 1 ergeben hat, so geht es weiter zum Programmausgang, und P^*_{halt} hält. P^*_{halt} hält also für den Input n gdw. $\text{halt}(n, n) = 1$ ist, d.h. gdw. $\neg\text{Halt}(n, n)$, und d.h. P^*_{halt} hält gdw. Programm Nr. n für den Input n nicht hält. (Durch Anhängen des Programms P_3, des „Paradoxators“, *tut* das neue Programm P^*_{halt} das Gegenteil dessen, was die Berechnung von $\text{halt}(n, n)$ ergeben hat.)

Die Funktionsweise des Programms P^*_{halt} lässt sich also wie folgt zusammenfassen:

- Es testet das Programm auf der Listenposition n, ob es für die Zahl n als Input hält oder nicht. (D.h. es testet das Programm auf der Listenposition n auf Halten *für seine eigene Listenposition* als Input.)
- Dann *tut* P^*_{halt} das Gegenteil dessen, was der gerade durchgeführte Test ergeben hat. (Dafür sorgt der „Paradoxator“.)

Wenn P^*_{halt} ein Abacus-Programm ist, so ist es gödelisierbar und hat folglich eine Gödelzahl. Daher kommt es irgendwo in unserer nach Gödelzahlen aufstei-

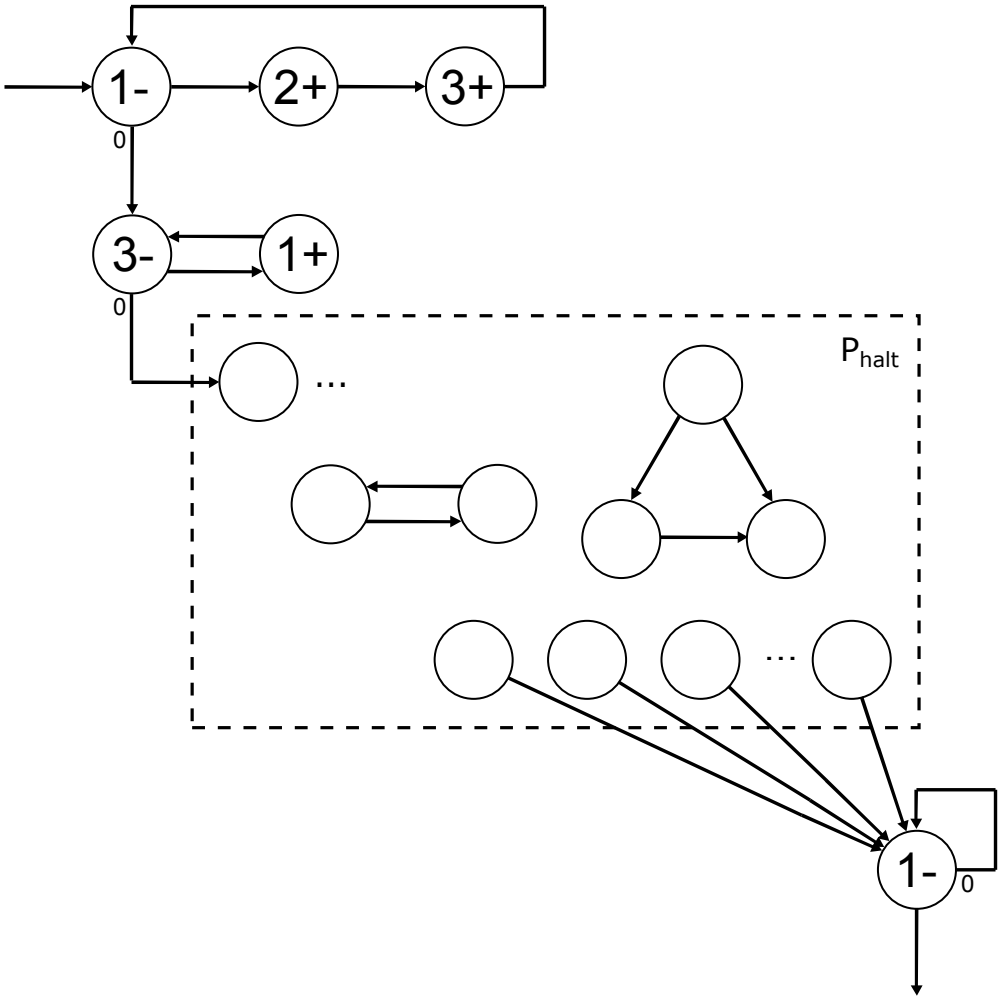


Abbildung 4.11: Dieses Programm gibt es nicht!

gend geordneten Liste der Programme (deren Anfang Abbildung 4.6 auf S. 46 zeigt) vor; nennen wir seine Nummer in der Liste p^*_{halt}.

Was würde passieren, wenn wir P^*_{halt} mit dem Input p^*_{halt} laufen ließen? Es gibt nur zwei Möglichkeiten:

1. P^*_{halt} hält für p^*_{halt}. Dann ist $\text{halt}(p^*_{\text{halt}}, p^*_{\text{halt}}) = 0$, und am kritischen Punkt ist folglich 0 im ersten Register. Dann aber hält P^*_{halt} *nicht*, wie oben beschrieben.

2. Oder aber, P^*_{halt} hält nicht für p^*_{halt}. Dann ist $\text{halt}(p^*_{\text{halt}}, p^*_{\text{halt}}) = 1$, und am kritischen Punkt ist folglich 1 im ersten Register. Dann aber *hält* P^*_{halt}, wie oben beschrieben.

Wir haben also: P^*_{halt} *hält* für p^*_{halt} gdw. P^*_{halt} *nicht* hält für p^*_{halt}.

Das ist ein Widerspruch! (P^*_{halt} verhält sich für seine eigene Listenposition als Input paradox.) Folglich muss unsere Annahme, dass das Halteproblem berechenbar ist, falsch sein, und es muss gelten:

Metatheorem 4.1. Das Halteproblem für den infiniten Abacus ist nicht berechenbar.

Hier ist der Beweis, in konziser Form noch einmal angeschrieben:

Beweis.

1. * Die Funktion ‚halt' ist berechenbar. (Annahme)

2. Dann ist die Funktion ‚halt' abacus-berechenbar.
(aus 1, Church-Turing-These)

3. Dann gibt es ein Program P_{halt}, das die Funktion ‚halt' berechnet.
(aus 2.)

4. Dieses Programm hält für alle Zahlenpaare als Input. (aus 3.)

5. Es hat also mindestens einen Programmausgang. (aus 4.)

6. Dem Programm kann das Kopierprogramm aus Abbildung 4.9 auf S.49 vorgeschaltet werden. Das neue Programm P^1_{halt} berechnet nun für jede natürliche Zahl n den Funktionswert $\text{halt}(n, n)$.
(aus 5., Programmverkettung)

7. Dem Programm kann das Programm P_3, das für die Null als einzige Zahl nicht hält (vgl. Abbildung 4.2 auf S. 32 u. Abbildung 4.6 auf S. 46) nachgeschaltet werden. Dadurch entsteht ein neues Programm, P^*_{halt}, das hält für den Input n gdw. Programm P_n nicht hält für den Input n.
(aus 6., Programmverkettung)

8. Das Programm P^*_{halt} hat die Gödelzahl p^*_{halt}. (aus 7., Gödelisierung)

9. Das Programm P^*_{halt} hält für den Input p^*_{halt} gdw. es für den Input p^*_{halt} nicht hält. (aus 7., 8., Diagonalisierung)

10. Also ist das Halteproblem nicht abacus-berechenbar.
(aus 1.–9., indirekter Beweis)

11. Also ist das Halteproblem nicht berechenbar.
(aus 10., Church-Turing-These)

□

In Abschnitt 1.3 hatte ich Ihnen angekündigt, dass „[d]er Schluss auf die Unentscheidbarkeit der Prädikatenlogik [...] ein einfacher *modus tollens* sein" würde. Dieser Schluss ist mit dem bisher Erarbeiteten formulierbar — im Prinzip, weil wir jetzt wissen, was es bedeutet, dass das Halteproblem für den infiniten Abacus nicht berechenbar ist. Der Schluss geht nämlich so:

1. Wenn die Prädikatenlogik entscheidbar ist, dann ist das Halteproblem für den infiniten Abacus berechenbar.
2. Das Halteproblem für den infiniten Abacus ist nicht berechenbar.

Die Prädikatenlogik ist nicht entscheidbar.

Prämisse 2 (die mit der Verneinung darin) ist Metatheorem 4.1; Prämisse 1 (den Konditionalsatz) werden wir im nächten Kapitel beweisen.

Anmerkung: Wenn jetzt unter Umständen der Eindruck entstanden ist, dass es eine überaus wünschenswerte Eigenschaft jedes Programms ist, dass es hält, so ist dem keineswegs so. Betriebssysteme beispielsweise sollen rechnen, solange der Computer läuft. Auch bei Textverarbeitungen beispielsweise wären wir wohl sehr unzufrieden, wenn sie ihre Bereitschaft, unsere Texte aufzunehmen, irgendwann durch ihr Halten einstellten. Es ärgert uns (oder zumindest mich, den Autor dieses Texts) sehr, wenn meine Musik- und Videostreaming-Apps plötzlich einen Programmausgang erreichen und aufhören, ihre Inhalte wiederzugeben. Bei Prozesssteuerungen ist es ebenso: Beim digitalen Raumthermostaten wäre das Halten seines internen Programms vielleicht ärgerlich; bei einer industriellen Anwendung kann durch das Halten der Prozesssteuerung enormer Schaden entstehen ...

4.7 Der Nichtberechenbarkeitsbeweis als Diagonalbeweis

Der Beweis, dass das Halteproblem für den infiniten Abacus nicht berechenbar ist, weist ebenso alle wesentlichen Merkmale eines Diagonalbeweises auf, wie der in Abschnitt 4.4 (s.o., S. 38) über die Überabzählbarkeit der Menge der reellen Zahlen. Denn auch hier tritt die „komplementierende Selbstanwendung" auf; die Schritte sind ganz analog zu der Liste auf S. 40.

1. Zunächst wird die Selbstanwendung vorbereitet: Durch die Definition der einstelligen Funktion $\text{halt}^1(n) = \text{halt}(n, n)$ und des Programms P^1_{halt}, das sie berechnen soll, werden nur diejenigen Fälle herausgefiltert, in denen das Halten eines Programms für seine eigene Listenposition (in der nach Gödelzahlen aufsteigend geordneten Liste der Abacus-Programme) als Input getestet werden soll.

2. Durch das Anhängen des „Paradoxators“ an das Programm P^1_{halt} entsteht das Programm P^*_{halt}, das für den Input i hält gdw. P^1_{halt} errechnet hat, dass Programm Nr. i für den Input i nicht hält. (Das ist die Komplementbildung.)

3. Die Annahme für den indirekten Beweis war, dass P^*_{halt} ein Abacus-Programm ist, d.h. wenn P^*_{halt} an der Position p^*_{halt} in unserer Liste der Abacus-Programme vorkommt, gilt zusammen mit seinem definierten Verhalten für alle $i \in \mathbb{N}$: $\text{halt}(p^*_{\text{halt}}, i) = 1 - \text{halt}(i, i)$, also $\text{halt}(p^*_{\text{halt}}, i) \neq \text{halt}(i, i)$. (Das ist wieder eine Instanz der Negation des Thomson-Theorems.) — Daraus folgt, wenn das Programm P^*_{halt} bezüglich Halten für seine eigene Listenposition getestet wird: $\text{halt}(p^*_{\text{halt}}, p^*_{\text{halt}}) = 1 - \text{halt}(p^*_{\text{halt}}, p^*_{\text{halt}})$, also $\text{halt}(p^*_{\text{halt}}, p^*_{\text{halt}}) \neq \text{halt}(p^*_{\text{halt}}, p^*_{\text{halt}})$. Das ist offensichtlich ein Widerspruch.

4. Daher gibt es keine natürliche Zahl p^*_{halt}, die die Listenposition unseres Programms P^*_{halt} sein könnte.

(Es ging dann weiter: Da auf unserer Liste alle Abacus-Programme vorkommen, kann P^*_{halt} kein Abacus-Programm sein. Das Kopierprogramm und das Programm P_3, also der „Paradoxator“, sind aber unproblematisch; ebenso die Programmverkettungen: Also muss aus der Nichtexistenz des Programms P^*_{halt} die Nichtexistenz eines Programms P_{halt} folgen, d.h. dass es kein Abacus-Programm gibt, das das Halteproblem für den infiniten Abacus berechnet.)

Auch diese, zunächst nicht als zweidimensionales Feld angeschriebene Diagonalisierung kann man aber auf diese Form bringen ([Jac04, S. 55–60, 73–101]: „*matrix-topological*“ vs. „*non-matrix-topological diagonalization*“) — „Non-matrix-diagonal constructions can also be given a literal or topological interpretation.“ [Jac04, S. 90]

Schreiben wir zunächst eine Tabelle, in der jede Zeile einem Programm entspricht, und jede Spalte einer natürlichen Zahl als Input. Dann finden wir in Zeile i in Spalte j den Wert der Funktion $\text{halt}(i, j)$, d.h. 0 gdw. das Programm mit der Listenposition i für den Input j hält und 1 sonst.

Nehmen wir nun das Programm P^*_{halt} in unsere Liste in irgendeiner Zeile auf, so finden wir, wenn wir seine Zeile ausfüllen, zunächst: Das Programm P^*_{halt} tut das Gegenteil dessen, was der Test des Programms an der Listenposition i für den Input i bezüglich Halten ergibt.

Die Testergebnisse für das Programm an der Listenposition i für den Input i finden wir auf der Diagonale unserer Tabelle. Die Zeile für P^*_{halt} beginnen wir also auszufüllen, indem wir unter die Diagonale in dieser Zeile jeweils die Ziffer setzen, die auf der Diagonale *nicht* vorkommt. Die Vorschrift für das sukzessive Ausfüllen der Zeile für P^*_{halt} könnte also wie folgt formuliert werden:

- Um in der Zeile für P^*_{halt} die Spalten $i = 0, 1, 2, \ldots$ auszufüllen:
- Sieh in der Tabelle in Zeile i, Spalte i nach, welche Ziffer du dort vorfindest.
- Schreibe in der Zeile für P^*_{halt} in Spalte i (also genau unter diese Ziffer) diejenige der beiden Ziffern 0 oder 1, die du auf der Diagonale *nicht* vorgefunden hast.

Beispielsweise sehen wir für das Ausfüllen der Spalte Nr. 0 in der Zeile für P^*_{halt} auf der Diagonale in Zeile 0, Spalte 0 nach: Dort steht eine 0 (da Programm P_0 für den Input 0 hält) — wir schreiben also in der Zeile für P^*_{halt} in Spalte 0 *nicht* eine 0, sondern eine 1. Für das Ausfüllen der nächsten Spalte Nr. 1 in der Zeile für P^*_{halt} sehen wir auf der Diagonale in Zeile 1, Spalte 1 nach: Dort steht eine 1 (da Programm P_1 für den Input 1 nicht hält) — wir schreiben also in der Zeile für P^*_{halt} in Spalte 1 *nicht* eine 1, sondern eine 0. So schreiten wir eine Zeitlang fort, aber nicht für immer, denn ...

Wenn wir auf diese Weise beim Ausfüllen der Zeile bei der Listenposition von P^*_{halt} angelangt sind, also der Zahl p^*_{halt} — das ist genau die Stelle wo die Diagonale die Zeile für P^*_{halt} kreuzt —, so finden wir, dass wir dieses Feld nicht ausfüllen können: Denn sobald wir eine 0 schreiben wollen, müssten wir eigentlich eine 1 dort hinschreiben, und sobald wir eine 1 schreiben wollen, müssten wir eigentlich eine 0 dort hinschreiben. Die folgende Tabelle 4.1 zeigt dies. (Auf der Diagonale finden wir die Folge ‚$0, 1, 0, 0, 0, \ldots$', daher beginnt die Zeile für P^*_{halt} mit ‚$1, 0, 1, 1, 1$'. Es ist das Feld mit dem doppelt unterstrichenen Fragezeichen rechts unten, das nicht ausgefüllt werden kann.)

	0	1	2	3	4	...	p^*_{halt}
P_0	$\underline{0}$	0	0	0	0		0
P_1	1	$\underline{1}$	1	1	1		1
P_2	0	0	$\underline{0}$	0	0		0
P_3	1	0	0	$\underline{0}$	0		0
P_4	0	0	0	0	$\underline{0}$		0
...						$\ddots$	
P^*_{halt}	1	0	1	1	1		$\underline{?}$

Tabelle 4.1: Matrix-topologische Darstellung

4.8 Gödelisierung heute: über Dateien

Gödelisierungen waren zum Zeitpunkt ihrer Erfindung / Entdeckung fast nur von theoretischem Interesse; mittlerweile sind sie durch die massenhafte Verbreitung von Computern ausgesprochen üblich. Zumindest kann man sehr vieles, was in einem Computer gespeichert ist, im Lichte der Gödelisierung betrachten.

Wenn wir nämlich eine Computerdatei von der ersten bis zur letzten Binärziffer (vom ersten bis zum letzten „bit“: *binary digit*) als die Binärdarstellung einer natürlichen Zahl auffassen, dann sind die meisten Dateien als Gödelzahlen interpretierbar: Eine Musikdatei ist die Gödelzahl eines ca. vier Minuten langen Schallereignisses, und der Player rekonstruiert Schallereignisse aus ihren Gödelzahlen. Eine Bilddatei ist die Gödelzahl eines Bildes, und der Viewer rekonstruiert Bilder aus ihren Gödelzahlen. (Eine 10 Megabyte große Datei kann damit als eine einzige natürliche Zahl mit ca. 20.000.000 Dezimalstellen aufgefasst werden.) Und eine App (eine ausführbare Datei) ist die Gödelzahl eines Algorithmus, im Prinzip ganz ähnlich unseren Gödelzahlen für Abacus-Programme .... — Es ist genau das, was unsere modernen Computer so vielseitig macht: dass sie, wie bereits auf S. 28 angemerkt, „endliche universelle Turing-Maschinen“

sind, d.h. sie können, gegeben seine Gödelzahl, jeden beliebigen Algorithmus abarbeiten. In moderner und vielen von uns höchstwahrscheinlich näherliegender Sprache ausgedrückt bedeutet dies nichts anderes, als dass auf einem modernen Computer jede beliebige App ausgeführt werden kann und somit läuft, die für ihn gemacht ist. Wir brauchen nicht einen Computer für ökonomische Berechnungen, einen für wissenschaftliche Berechnungen, einen anderen für die Textverarbeitung und noch einen weiteren zum Spielen. Es ist ein einziger Computer, der alle diese Dinge im Prinzip kann und auch ohne Weiteres tut, wenn wir die richtige App starten.

Manipulationen von Computerdaten können auf diese Weise als die Berechnung einer Gödelzahl aus einer anderen Gödelzahl verstanden werden. Filtern wir z.B. eine Musikdatei durch einen *Loudness*-Filter, so wird aus der Gödelzahl des ursprünglichen Schallereignisses die Gödelzahl eines neuen Schallereignisses berechnet, das zwar bei Wiedergabe sehr ähnlich klingt wie das ursprüngliche, aber im Vergleich zu diesem angehobene Bässe und Höhen hat, damit es in der Wahrnehmung lauter erscheint. Wenn wir mit einem Bildbearbeitungsprogramm auf ein Foto den (Pseudo-)Solarisationseffekt anwenden, dann errechnet unser Computer aus der Gödelzahl des ursprünglichen Bildes die Gödelzahl eines neuen Bildes, auf dem dieselben Konturen erkennbar sind, dessen Farben aber vom ursprünglichen Bild stark abweichende Falschfarben sind.

Die Festsetzung oder Feststellung, welche Zahlen als Gödelzahlen überhaupt in Frage kommen und welche nicht (so z.B. die Bemerkung oben, S. 44, dass gemäß unserer Gödelisierungsvorschrift die Ziffer ‚5' in keiner Gödelzahl eines Abacus-Programms vorkommen kann), heißt heutzutage „Dateiformat". Damit wird es auch zu einer praktischen Notwendigkeit, dass die Eigenschaft einer natürlichen Zahl, Gödelzahl eines bestimmten Gebildes zu sein (natürlich immer relativ zu einer gegebenen Gödelisierungsvorschrift), entscheidbar ist, vgl. [Chu36a, S 354, Theorem VI]: Denn wir wollen, wann immer wir eine Computerdatei öffnen, nach endlicher Zeit die Rückmeldung haben, ob ihr Format in Ordnung ist oder nicht.

Und ziemlich genau das, was wir mit einer „Gödelisierungsvorschrift" gemacht haben, nennt man in der Informatik, wenn man es automatisiert, einen *Codec*: eine Zusammensetzung aus dem Wortpaar COder / DECoder, das genau beschreibt, was solche Algorithmen tun. Sie codieren nämlich, also sie ordnen irgendwelchen Gebilden Gödelzahlen zu; und sie decodieren, d.h. sie gewinnen aus Gödelzahlen die ursprünglichen Gebilde wieder.

Man kann also sagen, Gödel habe in seinem bahnbrechenden Text [Göd86] unter anderem auch noch das Dateiformat und so gut wie alles, was daran hängt, erfunden — und das sogar fünf Jahre, bevor Turing seine Idee mit den nach ihm benannten Maschinen hatte! (Insbesondere mit der universellen Turing-Maschine, deren Implementierung im Endlichen, weil im Materiellen, unsere modernen Computer sind. Der Umgang mit ihnen ohne Gödels Erfindung / Entdeckung, ohne Dateien, wäre undenkbar.)

Kapitel 5

Mit Abacus-Berechnungen assoziierte Argumentformen

Es wird Zeit, dass wir vom Abacus, seinen Berechnungen sowie dem Halteproblem die Brücke zurück zur Prädikatenlogik schlagen. Dies werden wir mittels ganz bestimmter Argumentformen tun, die jeweils zu einer Berechnung des Abacus (s.o., S. 23) gehören.

5.1 Berechnungen als Folgen

Erinnern wir uns: Wenn wir ein Abacus-Programm (mit eingetragenen Zuständen, also kleinen ‚q' mit unteren Indizes bei den Pfeilen) und die Anfangskonfiguration haben, so ist die Berechnung durch die Folge der Kofigurationen vollständig beschrieben.

Rufen wir uns nochmal das Abacus-Programm, das die Nachfolgerfunktion berechnet, aus Abbildung 3.5 auf S. 24 in Erinnerung: Sein Programmeingang ist der Zustand q_0, sein Programmausgang der Zustand q_1. Seine Berechnungen brauchen stets nur ein Register, und zwar das erste.

Jede seiner Berechnungen beginnt im Zustand q_0 und endet im nächsten Zeitpunkt im Zustand q_1, wobei sich beim Übergang die im ersten Register gespeicherte Zahl um eins vergrößert. — Das ist übrigens immer so: Beim Übergang von einem Zustand in einen anderen verändert sich höchstens ein Register des Abacus, und zwar höchstens um eins.

Eine Konfiguration eines Abacus war eine Liste aus natürlichen Zahlen und kleinen ‚q' mit unteren Indizes (für die Zustände des Abacus), die den Abacus zusammen mit seinem Programm in je einem Zeitpunkt seiner Verarbeitung vollständig beschrieb: Einer solchen Liste ist nämlich zu entnehmen,

- im wievielten Schritt die Verarbeitung gerade steht,
- in welchem Zustand sich der Abacus in diesem Schritt gerade befindet,
- und welche Register in diesem Schritt welche Inhalte haben.

Jede Konfiguration hat nur endlich viele Glieder, da in jedem Abacus-Programm nur endlich viele Register verändert werden können (weil Abacus-Programme endliche Graphen sind). Genauer: Wenn der Abacus seinem Programm gemäß n-viele Register nutzt, dann hat eine Konfiguration die Länge $n + 2$: das erste Glied ist die Nummer des Schritts, das zweite ist der Zustand, in dem sich der Abacus gerade befindet, und dann folgen ab dem dritten Glied genau n-viele Registerinhalte.

Berechnungen des Abacus sind nun vollständig beschreibbar als Folgen von Konfigurationen. Beispielsweise beschreibt die Folge $,0, q_0, 7 — 1, q_1, 8‘$ die Berechnung des Nachfolgerprogramms für den Input 7.

Schreiben wir die Konfigurationen des Abacus als geordnete Tupel (s.o., S. 17), so sind Abacus-Berechnungen als Folgen von Tupeln darstellbar. Die Folge aus dem voraufgehenden Absatz sieht auf diese Weise angeschrieben beispielsweise wie folgt aus:

$$\langle 0, q_0, 7\rangle, \langle 1, q_1, 8\rangle$$

Diese Folge lässt sich dem Programm gemäß nicht fortführen, da der Zustand q_1 ein Programmausgang (in diesem Fall: der einzige Programmausgang) des Nachfolgerprogramms ist.

Unser Summenprogramm aus Abbildung 3.9 auf S. 26 nutzt zwei Register des Abacus, daher sind seine Konfigurationen Quadrupel. Die Bildunterschriften in Abbildung 3.10 auf S. 26, auf dieselbe Art wie vorhin in eine Folge gefasst, zeigen die Berechnung der Summe $5 + 3$:

$$\langle 0, q_0, 5, 3\rangle, \langle 1, q_1, 5, 2\rangle, \langle 2, q_2, 6, 2\rangle, \langle 3, q_1, 6, 1\rangle,$$
$$\langle 4, q_2, 7, 1\rangle, \langle 5, q_1, 7, 0\rangle, \langle 6, q_2, 8, 0\rangle, \langle 7, q_3, 8, 0\rangle$$

Wiederum kann die Folge nach ihrem letzten Element dem Programm gemäß nicht fortgeführt werden, da der Zustand q_3 ein Programmausgang ist, an dem die Verarbeitung endet.

Das Programm aus Abbildung 4.2 auf S. 32 (Programm P_3 aus Abbildung 4.6 auf S. 46, unserer Liste der Abacus-Programme nach Gödelzahlen) erlaubt z.B. folgende Berechnung:

$$\langle 0, q_0, 3\rangle, \langle 1, q_2, 2\rangle$$

Mit dem zweiten Folgeglied hält die Berechnung wiederum, die Folge lässt sich gemäß dem Programm nicht weiter fortsetzen.

Doch dasselbe Programm erlaubt auch folgende Berechnung:

$$\langle 0, q_0, 0\rangle, \langle 1, q_1, 0\rangle$$

Der Zustand q_1 ist kein Programmausgang, die Folge lässt sich gemäß dem Programm fortsetzen:

$$\langle 2, q_1, 0\rangle, \langle 3, q_1, 0\rangle, \langle 4, q_1, 0\rangle \ldots$$

Dass sich die Folge potenziell unendlich lange fortsetzen lässt, bedeutet nichts anderes, als dass diese Berechnung nicht hält und daher nie bei einem Programmausgang ankommt.

5.2 Konfigurationsbeschreibungen

Die Konfigurationen des Abacus, die wir bisher kennengelernt haben, lassen sich verbalisieren mit Hilfe des deutschen Prädikats: „Zum Zeitpunkt ... steht der Abacus im Zustand ... mit der Zahl ... im ersten Register, der Zahl ... im zweiten Register, ...“

Beispielsweise kann mit Hilfe dieses deutschen Prädikats die Konfiguration ‚$\langle 3, q_1, 0\rangle$‘ gelesen werden als: „Zum Zeitpunkt 3 steht der Abacus im Zustand q_1 mit der Zahl 0 im ersten Register.“

Und die Konfiguration ‚$\langle 2, q_2, 6, 2\rangle$‘ kann gelesen werden als: „Zum Zeitpunkt 2 steht der Abacus im Zustand q_2 mit der Zahl 6 im ersten Register und der Zahl 2 im zweiten Register.“

Haben wir noch Individuenkonstanten für die natürlichen Zahlen (denn Ziffern sind nicht Bestandteil unserer prädikatenlogischen Sprache), dann steht einer Formalisierung solcher Aussagen über Konfigurationen des Abacus nicht mehr viel im Wege. Vereinbaren wir daher folgenden Formalisierungsschlüssel:

C^{n+2}: Zum Zeitpunkt ... steht der Abacus im Zustand ... mit der Zahl ... im ersten Register, der Zahl ... im zweiten Register, ..., der Zahl ... im n-ten Register. [für $n \geq 1$]
$f(\ldots)$: der Nachfolger von ...
q: der Anfangszustand q_0 des Abacus
q_i: der Zustand q_i des Abacus [für $i \geq 1$]
a: die natürliche Zahl 0
a_i: die natürliche Zahl i [für $i \geq 1$]

Die Individuenkonstanten ‚a‘ und ‚q‘ repräsentieren also die Zahl 0 bzw. den Zustand q_0, und für alle übrigen natürlichen Zahlen, die in den Bezeichnern vorkommen, sind diese identisch mit dem unteren Index der jeweiligen Individuenkonstanten. — Mit Hilfe dieses Schlüssels formalisierte Prädikationen über Konfigurationen des Abacus werden wir *Konfigurationsbeschreibungen* des Abacus nennen.

Wenn Sie sich den Formalisierungsschlüssel genau angesehen haben, dann werden Sie festgestellt haben, dass er ein Funktionssymbol enthält, nämlich das ‚f‘. Wir machen hier also zunächst Prädikatenlogik mit Funktionssymbolen und wollen das kleine ‚f‘ als Zeichen für den Nachfolger einer natürlichen Zahl verwenden. Syntaktisch verhält es sich so, dass es unseren Begriff des Terms erweitert: Wenn t ein Term ist, dann ist auch $f(t)$ ein Term.

Oben genannte Konfiguration des Abacus aus dem voraufgehenden Abschnitt können wir also formalisieren als die Konfigurationsbeschreibung:

$$C(a_3, q_1, a)$$

Und das zweite Beispiel aus diesem Abschnitt, ‚$\langle 2, q_2, 6, 2\rangle$‘, ist formalisierbar als:

$$C(a_2, q_2, a_6, a_2)$$

Die Berechnung des Summenprogramms, die wir auch bereits kennen, und die im letzten Abschnitt zuletzt beispielhaft angeführt wurde, ist formalisierbar als die Folge von Prädikationen:

$$C(a, q, a_5, a_3), C(a_1, q_1, a_5, a_2), C(a_2, q_2, a_6, a_2), C(a_3, q_1, a_6, a_1),$$

$$C(a_4, q_2, a_7, a_1), C(a_5, q_1, a_7, a), C(a_6, q_2, a_8, a), C(a_7, q_3, a_8, a)$$

5.3 Programmbeschreibungen

Mit den Konfigurationsbeschreibungen können wir auch die Programme unseres Abacus beschreiben, und zwar als als allgemeine Aussagen über Übergänge zwischen Konfigurationen: als alle möglichen Übergänge zwischen je zwei Zuständen, gemäß dem Programm.

Betrachten wir unser bekanntes Programm für die Nachfolgerfunktion, so sehen wir, dass jede seiner Berechnungen genau einen Schritt braucht, der der Übergang vom Zustand q_0 in den Zustand q_1 ist.

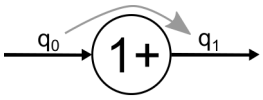


Abbildung 5.1: Der Zustandsübergang des Nachfolgerprogramms

Zum Zeitpunkt 0 steht der Abacus am Programmeingang q_0 mit (sagen wir) 5 im ersten Register. Dann steht der Abacus einen Schritt später, also zum Zeitpunkt 1, am Programmausgang q_1 mit 6 im ersten Register. Immer findet der Übergang vom Zustand q_0 in den Zustand q_1 in genau einem Schritt statt, und immer ist der Inhalt von Register Nr. 1 danach um 1 größer als davor. Dies können wir in die allgemeine Aussage fassen:

„Wenn der Abacus zum Zeitpunkt x im Zustand q_0 mit der Zahl y im ersten Register ist, dann ist er zum Zeitpunkt $x + 1$ im Zustand q_1 mit der Zahl $y + 1$ im ersten Register."

Diese Aussage können wir gemäß dem Formalisierungsschlüssel aus dem vorhergehenden Abschnitt formalisieren als (vergessen wir nicht, dass wir das einstellige Funktionssymbol ‚f' als die Nachfolgerfunktion vereinbart hatten, s.o., S. 59):

$$\forall x \forall y_1 (C(x, q, y_1) \rightarrow C(f(x), q_1, f(y_1))$$

Da das Nachfolgerprogramm genau einen Zustandsübergang hat, ist diese Formel auch schon seine Programmbeschreibung. Das Prädikat ‚C' ist hier dreistellig, da die Kofigurationsbeschreibungen des Abacus im Falle des Nachfolgerprogramms drei Terme brauchen: einen für den Zeitpunkt, einen für den Zustand, und einen weiteren für das einzige Register (Nr. 1), das der Abacus zur Berechnung der Nachfolgerfunktion nutzt. (Das Prädikat ‚C' hat immer zwei Stellen mehr als zur Berechnung Register erforderlich sind, da Zeitpunkt und Zustand immer dazukommen; s.o., S. 58, über die Länge der Konfigurationen. Konfigurationsbeschreibungen haben dieselbe Länge wie die Konfigurationen, die sie beschreiben.)

Eine Programmbeschreibung ist eine Beschreibung aller möglichen Zustandsübergänge des Abacus, d.h. aller möglichen Übergänge von einem Pfeil (Zustand des Abacus) über einen Befehl zum nächsten Pfeil (Zustand des Abacus).

Jeder ‚+'-Befehl hat nur einen von ihm ausgehenden Pfeil. Da von jedem in ihn eigehenden Pfeil ein Übergang über den Befehl in diesen einzigen ausgehenden Pfeil möglich ist, gibt es pro ‚+'-Befehl genau so viele Zustandsübergänge (und damit Formeln in der Programmeschreibung), wie Pfeile in den ‚+'-Befehl eingehen.

Jeder ‚−'-Befehl hat zwei von ihm ausgehende Pfeile: einen ohne Beschriftung und einen mit ‚0' beschrifteten. Da von jedem in den Befehl eingehenden Befehl jeweils ein Übergang über den Befehl in jeden der ausgehenden Pfeile möglich ist, gibt es pro ‚−'-Befehl zweimal so viele Zustandsübergänge (und damit Formeln in der Programmbeschreibung), wie Pfeile in den ‚−'-Befehl eingehen.

Ein Blick auf das Programm, das den Vorgänger einer natürlichen Zahl berechnet (und für die 0 wiederum 0 als Funktionswert ausgibt; Programm P_2 aus Abbildung 4.6 auf S. 46), zeigt, was gemeint ist. Es gibt in diesem Programm genau einen ‚−'-Befehl, und demgemäß zwei Zustandsübergänge: einen von q_0 nach q_1 (der erfolgt gdw. der Input nicht 0 ist), und einen von q_0 nach q_2 (der erfolgt gdw. der Input 0 ist).

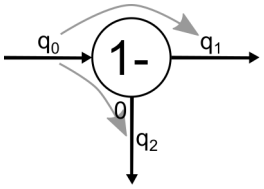


Abbildung 5.2: Zustandsübergänge des Vorgängerprogramms

Der erste Zustandsübergang ist beschreibbar als: „Wenn der Abacus zum Zeitpunkt x im Zustand q_0 mit der Zahl $y+1$ im ersten Register ist, dann ist er zum Zeitpunkt $x+1$ im Zustand q_1 mit der Zahl y im ersten Register." (y, die Zahl in Register Nr. 1 nach dem Befehl, ist klarerweise um 1 weniger als $y+1$, die Zahl ebendort vor dem Befehl, sodass der Satz die Verringerung des Inhalts von Register Nr. 1 ausdrückt.)

Der zweite Zustandsübergang ist beschreibbar als: „Wenn der Abacus zum Zeitpunkt x im Zustand q_0 mit der Zahl 0 im ersten Register ist, dann ist er zum Zeitpunkt $x+1$ im Zustand q_2 mit der Zahl 0 im ersten Register."

Diese beiden Beschreibungen sind formalisierbar als:

$$\forall x \forall y_1 (C(x, q, f(y_1)) \rightarrow C(f(x), q_1, y_1))$$
$$\forall x (C(x, q, a) \rightarrow C(f(x), q_2, a))$$

Das Programm, das für den Input 0 einzig nicht hält aus Abbildung 4.2 auf S. 32, Programm P_3 aus unserer Liste auf S. 46, besteht aus einem ‚−'-Befehl,

in den diesmal zwei Pfeile hineinführen: der Programmeingang q_0 sowie der Zustand q_1. Dementsprechend gibt es für diesen einen Befehl vier Zustandsübergänge:

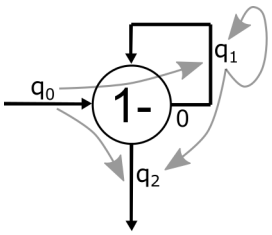


Abbildung 5.3: Zustandsübergänge des Programms P_3, vgl. Abb. 4.2

Die Programmbeschreibung sind somit folgende vier Formeln:

$$\forall x \forall y_1 (C(x, q, f(y_1)) \rightarrow C(f(x), q_2, y_1))$$
$$\forall x (C(x, q, a) \rightarrow C(f(x), q_1, a))$$
$$\forall x \forall y_1 (C(x, q_1, f(y_1)) \rightarrow C(f(x), q_2, y_1))$$
$$\forall x (C(x, q_1, a) \rightarrow C(f(x), q_1, a))$$

Die dritte dieser Formeln beschreibt einen Zustandsübergang, der niemals stattfindet. (☞ Warum nicht?) Wir wollen die Formel aber in der Programmbeschreibung belassen, denn sie stört nicht, und wir wollen uns solche semantischen Überlegungen über unsere Programme um der Einheitlichkeit des Vorgehens willen, und nicht zuletzt auch wegen der Irrtumssicherheit, ersparen.

Unser letztes Beispiel sei das Summenprogramm. Es hat einen ‚−'-Befehl, in den zwei Pfeile eingehen — also zunächst vier Zustandsübergänge — und einen ‚+'-Befehl, in den ein Pfeil eingeht — also noch ein Zustandsübergang mehr: Es hat insgesamt fünf Zustandsübergänge.

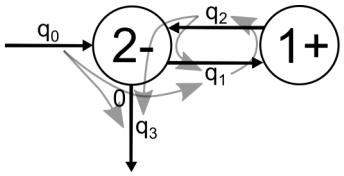


Abbildung 5.4: Zustandsübergänge des Summenprogramms

Sie sind formalisierbar als:

$$\forall x \forall y_1 \forall y_2 (C(x, q, y_1, f(y_2)) \rightarrow C(f(x), q_1, y_1, y_2))$$
$$\forall x \forall y_1 (C(x, q, y_1, a) \rightarrow C(f(x), q_3, y_1, a))$$
$$\forall x \forall y_1 \forall y_2 (C(x, q_1, y_1, y_2) \rightarrow C(f(x), q_2, f(y_1), y_2))$$
$$\forall x \forall y_1 \forall y_2 (C(x, q_2, y_1, f(y_2)) \rightarrow C(f(x), q_1, y_1, y_2))$$

$$\forall x \forall y_1 (C(x, q_2, y_1, a) \to C(f(x), q_3, y_1, a))$$

— und so haben wir die Programmbeschreibung unseres Summenprogramms bekommen.

5.4 Beschreibungen der Programmausgänge

Wir können in unserer prädikatenlogischen Sprache auch ausdrücken, dass ein Abacus-Programm hält, also einen Programmausgang erreicht. Dazu formulieren wir Sätze, die mit Hilfe unseres Prädikats ‚C' ausdrücken: Der Abacus kommt zu irgendeinem Zeitpunkt zum ersten Programmausgang mit irgendwelchen beliebigen Zahlen in seinen Registern, oder der Abacus kommt zu irgendeinem Zeitpunkt zum zweiten Programausgang mit irgendwelchen beliebigen Zahlen in seinen Registern, oder .... — Jeder Teilsatz (die Teilsätze sind durch das Wort ‚oder' getrennt) ist durch eine Existenzformel zu formalisieren; haben wir mehrere Teilsätze (also Existenzformeln), so ist deren Oder-Verknüpfung zu bilden: ihre Disjunktion.

Unser Nachfolgerprogramm beispielsweise hat den Programmausgang q_1; für die Beschreibung dieses Umstands alleine ist uns egal, zu welchem Zeitpunkt und mit welcher Zahl im ersten Register er dort ankommt — uns interessiert lediglich, dass der Zustand q_1 ein Programmausgang ist und irgendwann mit irgendwelchen Registerinhalten erreicht werden kann. Formalisieren wir dies, so erhalten wir die Existenzformel

$$\exists x \exists y_1 C(x, q_1, y_1)$$

Sie beschreibt vollständig den einzigen Programmausgang und damit die möglichen Haltezustände des Nachfolgerprogramms.

Dass unser Vorgängerprogramm P_2 die Programmausgänge q_1 und q_2 hat, formalisieren wir als

$$(\exists x \exists y_1 C(x, q_1, y_1) \vee \exists x \exists y_1 C(x, q_2, y_1))$$

(Wieder ist uns egal, zu welchem Zeitpunkt der Abacus in q_1 oder q_2 hält und mit welcher Zahl im ersten Register.)

☞ Können Sie formalisieren, dass das Programm P_3 als (einzigen) Programmausgang den Zustand q_2 hat?

Dass das Summenprogramm den Programmausgang q_3 hat, formalisieren wir als

$$\exists x \exists y_1 \exists y_2 C(x, q_3, y_1, y_2)$$

Die Formeln, die die Programmausgänge beschreiben, sind also existenzielle Generalisierungen der Konfigurationsbeschreibungen, die an den Programmausgängen auftreten, bezüglich Zeitpunkt und Registerinhalten.

5.5 Syntaktik und Semantik assoziierter Argumentformen

Jetzt haben wir alle Bestandteile für unsere mit Berechnungen assoziierten Argumentformen. Wir können sie nun zusammenbauen — mit einem ganz bestimmten Ziel vor Augen, nämlich: Eine mit einer Berechnung assoziierte Argumentform soll gültig sein gdw. die Berechnung hält. Wir werden diese Eigenschaft dieser Argumentformen ganz wesentlich für unseren Unentscheidbarkeitsbeweis brauchen.

Eine mit einer Berechnung (die klarerweise gemäß einem bestimmten Programm verläuft) assoziierte Argumentform besteht aus

- einer Beschreibung der Anfangskonfiguration des Abacus und
- der Programmbeschreibung als Prämissen, sowie
- der Beschreibung der Programmausgänge als Konklusion.

Sehen wir noch vor: Ist der letzte Punkt nicht erfüllbar, weil das Programm keinen einzigen Ausgang hat, so sei die Konklusion der logisch falsche Satz ‚$\exists x(F(x) \wedge \neg F(x))$‘.

Das Ziel, das wir bei der Konstruktion von mit Berechnungen assoziierten Argumentformen vor Augen haben, ist also dieses: Eine mit einer Berechnung assoziierte Argumentform ist gültig gdw. die entsprechende Berechnung hält. Da dies nicht augenfällig ist, wollen wir es beweisen.

Methodologische Zwischenbemerkung: Bei dem Beweis des Lemmas 5.1 bedienen wir uns der vollständigen Induktion. Da ich für diese Lehrveranstaltung ausschließlich die Elementare Logik voraussetze, möchte ich dieses überaus nützliche Beweisverfahren kurz erläutern.

Die vollständige Induktion ist in gewisser Weise das Spiegelbild der bereits erwähnten rekursiven Funktionen und funktioniert folgendermaßen: Wenn man zeigen will, dass ein Prädikat — nennen wir es P — für alle natürlichen Zahlen gilt, dann kann man nicht unendlich viele Beweise für $P(0)$, $P(1)$, $P(2)$ usw., also für das Zutreffen des Prädikats auf jede einzelne natüriche Zahl, führen. Man bedient sich vielmehr zweier getrennter Beweisschritte, „Indukionsanfang“, manchmal auch „(Induktions-)Basis“ einerseits und „Induktionsschritt“ andererseits genannt, um die Gesamtheit der natürlichen Zahlen abzudecken und das Zutreffen von P auf alle natürlichen Zahlen zu zeigen:

1. Im Induktionsanfang zeigt man, dass

   $$P(0)$$

   D.h. man zeigt, dass das Prädikat P auf die Zahl 0 zutrifft.

2. Im Induktionsschritt zeigt man (meist mit einem konditionalen Beweis), dass für alle natürlichen Zahlen n gilt:

   $$\text{Wenn } P(n), \text{ dann } P(n').$$

D.h. man zeigt: Wenn das Prädikat P auf eine natürliche Zahl n zutrifft, dann trifft es auch auf ihren Nachfolger n' zu. Die ‚Wenn'-Komponente dieses Konditionalsatzes, wenn sie für einen konditionalen Beweis angenommen wird, nennt man „Induktionshypothese".

So kann man sich mit Hilfe von lauter sukzessiven Anwendungen des *modus ponens* von einer natürliche Zahl zur nächsten durcharbeiten, potenziell durch die gesamte Reihe der natürlichen Zahlen: $P(0)$ gilt laut Induktionsanfang, und gemäß Induktionsschritt gilt: Wenn $P(0)$, dann $P(1)$. Also $P(1)$. — Soeben haben wir gezeigt: $P(1)$, und gemäß Induktionsschritt gilt: Wenn $P(1)$, dann $P(2)$. Also $P(2)$. — Soeben haben wir gezeigt: $P(2)$, und gemäß Induktionsschritt gilt: Wenn $P(2)$, dann $P(3)$. Also $P(3)$. — Da dieses Verfahren immer weiter zu führen ist, hat die vollständige Induktion gezeigt: Für alle natürlichen Zahlen n gilt: $P(n)$.

Für das Folgende sollten wir nicht vergessen, dass wir die Zeitpunkte in unseren Berechnungen mit 0 zu zählen beginnen, sodass eine Berechnung der Länge 1 die Konfiguration zum Zeitpunkt 0 umfasst; eine Berechnung der Länge 2 umfasst die Zeitpunkte 0 und 1 — allgemein: Eine Berechnung der Länge n ($n \geq 1$) besteht aus den Konfigurationen zu den Zeitpunkten 0, 1, ..., $n-1$.

Lemma 5.1. Für alle $x \in \mathbb{N}$: Wenn die Länge einer Berechnung größer oder gleich $x+1$ ist, dann folgt aus den Prämissen der mit ihr assoziierten Argumentform eine Beschreibung der Konfiguration des Abacus zum Zeitpunkt x.

Beweis. Wir beweisen durch vollständige Induktion, zunächst für $x = 0$:

Induktionsanfang, $x = 0$: Eine Konfigurationsbeschreibung für den Zeitpunkt 0 ist in den Prämissen enthalten und folgt aus ihnen aufgrund der Reflexivität des logischen Folgens.

Induktionsschritt: Unsere Induktionshypothese lautet: „Wenn die Länge einer Berechnung größer oder gleich $x+1$ ist, dann folgt aus den Prämissen eine Beschreibung der Konfiguration des Abacus zum Zeitpunkt x."

Zu zeigen ist: Wenn die Länge einer Berechnung größer oder gleich $x+2$ ist, dann folgt aus den Prämissen eine Beschreibung der Konfiguration des Abacus zum Zeitpunkt $x+1$.

1. * Wenn die Länge einer Berechnung größer oder gleich $x+1$ ist, dann folgt aus den Prämissen eine Beschreibung der Konfiguration des Abacus zum Zeitpunkt x. (Induktionshypothese)
2. * Die Länge der Berechnung ist größer oder gleich $x+2$. (Annahme)
3. Die Länge der Berechnung ist größer oder gleich $x+1$. (aus 2.)
4. Aus den Prämissen folgt eine Beschreibung der Konfiguration des Abacus zum Zeitpunkt x. (aus 1., 3., MP)
5. Sei $\mathfrak{I}$ eine Interpretation, unter der alle Prämissen wahr sind. (Def. $\mathfrak{I}$)
6. Die Beschreibung der Konfiguration des Abacus zum Zeitpunkt x ist wahr unter $\mathfrak{I}$. (aus 4., 5., Semantik)

7. Zum Zeitpunkt x steht der Abacus nicht auf einem Programmausgang, d.h. die Folge der Konfigurationen lässt sich dem Programm gemäß um mindestens ein weiteres Glied fortführen. (aus 2.)

8. Unter den Prämissen (in der Programmbeschreibung) gibt es eine Allformel, die so spezialisierbar ist, dass das Antecedens der erhaltenen Implikationsformel die Beschreibung der Konfiguration des Abacus zum Zeitpunkt x ist. (aus 7., Programmbeschreibung)

9. Da diese Implikationsformel die Spezialisierung einer unter $\mathfrak{I}$ wahren Allformel ist, ist sie wahr unter $\mathfrak{I}$. (aus 8., Semantik)

10. Das Konsequens dieser Implikationsformel ist wahr unter $\mathfrak{I}$. (aus 6., 9., Semantik)

11. Das Konsequens dieser Implikationsformel beschreibt die Konfiguration des Abacus zum Zeitpunkt $x + 1$. (aus 8., Programmbeschreibung)

12. Aus den Prämissen folgt die Beschreibung der Konfiguration des Abacus zum Zeitpunkt $x + 1$. (aus 10., 11.)

13. Wenn die Länge einer Berechnung größer oder gleichj $x + 2$ ist, dann folgt aus den Prämissen die Beschreibung der Konfiguration des Abacus zum Zeitpunkt $x + 1$. (aus 2.–12., KB)

14. Wenn gilt: Wenn die Länge einer Berechnung größer oder gleich $x + 1$ ist, dann folgt aus den Prämissen eine Beschreibung der Konfiguration des Abacus zum Zeitpunkt x — dann gilt: Wenn die Länge einer Berechnung größer oder gleichj $x+2$ ist, dann folgt aus den Prämissen die Beschreibung der Konfiguration des Abacus zum Zeitpunkt $x + 1$. (aus 2.–12., KB).

□

Beispiele für dieses Lemma, nämlich für das Folgen von Beschreibungen von Konfigurationen zu irgendwelchen Zeitpunkten, werden wir etwas weiter unten sehen, und zwar wenn wir Ableitungen mit mit Berechnungen assoziierten Argumentformen machen werden.

Metatheorem 5.1. Die mit einer Berechnung assoziierte Argumentform ist gültig gdw. die Berechnung hält.

Beweis. Dies ist ein ‚genau dann, wenn'-Satz; in einem solchen Fall geht man häufig so vor, dass man das ‚Wenn-Dann' in jede Richtung einzeln beweist.

1. Die eine Richtung ist einfach: Wenn die mit einer Berechnung assoziierte Argumentform gültig ist, dann hält die Berechnung. Sei nämlich $\mathfrak{I}$ eine Interpretation, unter der alle Prämissen wahr sind. Dann ist (aufgrund der Gültigkeit der assoziierten Argumentform) auch die Konklusion der Argumentform wahr unter $\mathfrak{I}$, und die Konklusion besagt, dass die Berechnung hält.

2. Für die andere Richtung hilft uns Lemma 5.1: Wenn die Berechnung hält, dann ist die mit ihr assoziierte Argumentform gültig. Denn gemäß dem Lemma folgt aus den Prämissen der Argumentform eine Beschreibung

jeder Konfiguration, die in der Berechnung auftritt — also auch eine Beschreibung der letzten Konfiguration in der Berechnung. In dieser Konfiguration ist der Abacus an einem Programmausgang. Die Konklusion der Argumentform beschreibt definitionsgemäß alle Programmausgänge, also folgt sie aus den Prämissen.

□

Die Berechnung des Nachfolgers der 7 mit unserem Nachfolgerprogramm beispielsweise hat, wie oben gezeigt (s.o., S. 58), zwei Glieder:

$$\langle 0, q_0, 7\rangle, \langle 1, q_1, 8\rangle$$

Die mit dieser Berechnung assoziierte Argumentform hat also folgende Komponenten:

1. $C(a, q, a_7)$ — dies ist die Beschreibung der Anfangskonfiguration,
2. $\forall x \forall y_1(C(x, q, y_1) \rightarrow C(f(x), q_1, f(y_1)))$ — dies ist die Programmbeschreibung, sowie
3. $\exists x \exists y_1 C(x, q_1, y_1)$ — dies ist die Beschreibung des einzigen Programmausgangs.

Die mit der Berechnung des Nachfolgers der 7 assoziierte Argumentform sieht nun wie folgt aus:

1.	$C(a, q, a_7)$
2.	$\forall x \forall y_1(C(x, q, y_1) \rightarrow C(f(x), q_1, f(y_1)))$
	$\exists x \exists y_1 C(x, q_1, y_1)$

Die Berechnung des Nachfolgers der 7 mit unserem Nachfolgerprogramm hält (klarerweise) — und die mit ihr assoziierte Argumentform ist auch gültig.

Sehen wir uns die Ableitung der Konklusion aus den Prämissen an:

1. $C(a, q, a_7)$	(P1)
2. $\forall x \forall y_1(C(x, q, y_1) \rightarrow C(f(x), q_1, f(y_1)))$	(P2)
3. $\forall y_1(C(a, q, y_1) \rightarrow C(f(a), q_1, f(y_1)))$	2., (UB)
4. $(C(a, q, a_7) \rightarrow C(f(a), q_1, f(a_7)))$	3., (UB)
5. $C(f(a), q_1, f(a_7))$	1., 4., (MP)
6. $\exists y_1 C(f(a), q_1, y_1)$	5., (EE)
7. $\exists x \exists y_1 C(x, q_1, y_1)$	6., (EE)

(Substitutionen in die Argumentstellen von Funktionssymbolen bei der universellen Beseitigung und existenzielle Einführung über Terme mit Funktionssymbolen bieten keine Überraschungen.)

Zeile (1) ist, gemäß der Definition der mit einer Berechnung assoziierten Argumentform, eine Beschreibung der Konfiguration des Abacus zum Zeitpunkt 0. Aus den Prämissen der Argumentform folgt, gemäß Lemma 5.1, auch jede Beschreibung jeder Konfiguration, die zu einem späteren Zeitpunkt in der Berechnung auftritt: Eine Beschreibung der Konfiguration zum Zeitpunkt 1 wurde in

Zeile (5) abgeleitet. Der atomare Satz in Zeile (5) ist, relativ zu unserem Formalisierungsschlüssel, verbalisierbar als: „Zum Zeitpunkt 1 steht der Abacus im Zustand q_1 mit der 8 im ersten Register."

Sehen wir uns das Programm P_4 aus Abbildung 4.6 auf S. 46 an:

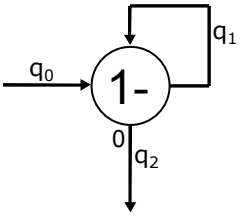


Abbildung 5.5: Konst. Funktion 0, mit eingetragenen Zuständen

Eine Berechnung dieses Programms, beispielsweise für den Input 3, sieht so aus:

$$\langle 0, q_0, 3\rangle, \langle 1, q_1, 2\rangle, \langle 2, q_1, 1\rangle, \langle 3, q_1, 0\rangle, \langle 4, q_2, 0\rangle$$

Die mit dieser Berechnung assoziierte Argumentform lautet:

1.	$C(a, q, a_3)$
2.	$\forall x \forall y_1 (C(x, q, f(y_1)) \rightarrow C(f(x), q_1, y_1))$
3.	$\forall x \forall y_1 (C(x, q, a) \rightarrow C(f(x), q_2, a))$
4.	$\forall x \forall y_1 (C(x, q_1, f(y_1)) \rightarrow C(f(x), q_1, y_1))$
5.	$\forall x \forall y_1 (C(x, q_1, a) \rightarrow C(f(x), q_2, a))$
	$\exists x \exists y_1 C(x, q_2, y_1)$

Diese Formalisierung der mit dieser Berechnung assoziierten Argumentform ist ungültig! Das liegt an unserer Formalisierung — unter einer Interpretation beispielsweise, unter der der atomare Satz ‚$C(a, q, a_3)$' wahr ist, und unter der alle anderen atomaren Sätze falsch sind, sind alle Prämissen der Argumentform wahr, und ihre Konklusion ist falsch.

Versuchen wir die Ableitung der Konklusion aus den Prämissen, so sehen wir, wo wir auf Schwierigkeiten stoßen:

1. $C(a, q, a_3)$	(P1)
2. $\forall x \forall y_1 (C(x, q, f(y_1)) \rightarrow C(f(x), q_1, y_1))$	(P2)
3. $\forall x \forall y_1 (C(x, q, a) \rightarrow C(f(x), q_2, a))$	(P3)
4. $\forall x \forall y_1 (C(x, q_1, f(y_1)) \rightarrow C(f(x), q_1, y_1))$	(P4)
5. $\forall x \forall y_1 (C(x, q_1, a) \rightarrow C(f(x), q_2, a))$	(P5)
6. $\forall y_1 (C(a, q, f(y_1)) \rightarrow C(f(a), q_1, y_1))$	2., (UB)
7. $(C(a, q, f(a_2)) \rightarrow C(f(a), q_1, a_2))$	6., (UB)

In Zeile (6) bietet sich ausschließlich Formel (2) zur Spezialisierung durch (UB) an. In Zeile (7) aber sehen wir, dass wir nicht mit *modus ponens* auf die Beschreibung der Konfiguration zum Zeitpunkt 1 schließen können, da die Beschreibung unserer Anfangskonfiguration die Individuenkonstante ‚a_3' für die

Zahl 3 verwendet, die Spezialisierung der zweiten Prämisse allerdings den komplexen Term ‚$f(a_2)$‘. Unser Logiksystem ist so sensibel, dass es diese Identität nicht von vorneherein „sieht“; anders ausgedrückt: In die Semantik der Prädikatenlogik ist keine Information über die Nachfolgerfunktion betreffend die natürlichen Zahlen „eingebaut“. Abhilfe könnten wir auf zwei Arten schaffen:

- Entweder führen wir eine lange Konjunktionsformel ein, bis zur größten in unserer Formalisierung vorkommenden natürlichen Zahl, die die Nachfolgerfunktion mit den Individuenkonstanten für die natürlichen Zahlen in Verbindung bringt, in unserem Falle:

$$(a_1 = f(a) \wedge (a_2 = f(a_1) \wedge a_3 = f(a_2)))$$

- Oder wir eliminieren die Individuenkonstanten für die natürlichen Zahlen und schreiben von vorneherein ‚$f(a)$‘ statt ‚a_1‘, ‚$f(f(a))$‘ statt ‚a_2‘, ‚$f(f(f(a)))$‘ statt ‚a_3‘ usw., wieder bis zur größten in unserer Formalisierung vorkommenden natürlichen Zahl.

Beide Wege machen die Formalisierung der assoziierten Argumentform in diesem Falle (und auch in allen anderen Fällen) gültig. — Folgend ist die Ableitung mit der Eliminierung der Individuenkonstanten, also nach der zweiten Methode, vorgezeigt.

1. $C(a, q, f(f(f(a))))$	(P1)
2. $\forall x \forall y_1 (C(x, q, f(y_1)) \rightarrow C(f(x), q_1, y_1))$	(P2)
3. $\forall x \forall y_1 (C(x, q, a) \rightarrow C(f(x), q_2, a))$	(P3)
4. $\forall x \forall y_1 (C(x, q_1, f(y_1)) \rightarrow C(f(x), q_1, y_1))$	(P4)
5. $\forall x (C(x, q_1, a) \rightarrow C(f(x), q_2, a))$	(P5)
6. $\forall y_1 (C(a, q, f(y_1)) \rightarrow C(f(a), q_1, y_1))$	2., (UB)
7. $(C(a, q, f(f(f(a)))) \rightarrow C(f(a), q_1, f(f(a))))$	6., (UB)
8. $C(f(a), q_1, f(f(a)))$	1., 7., (MP)
9. $\forall y_1 (C(f(a), q_1, f(y_1)) \rightarrow C(f(f(a)), q_1, y_1))$	4., (UB)
10. $(C(f(a), q_1, f(f(a))) \rightarrow C(f(f(a)), q_1, f(a)))$	9., (UB)
11. $C(f(f(a)), q_1, f(a))$	8., 10., (MP)
12. $\forall y_1 (C(f(f(a)), q_1, f(y_1)) \rightarrow C(f(f(f(a))), q_1, y_1))$	4., (UB)
13. $(C(f(f(a)), q_1, f(a)) \rightarrow C(f(f(f(a))), q_1, a))$	12., (UB)
14. $C(f(f(f(a))), q_1, a)$	11., 13., (MP)
15. $(C(f(f(f(a))), q_1, a) \rightarrow C(f(f(f(f(a)))), q_2, a))$	5., (UB)
16. $C(f(f(f(f(a)))), q_2, a)$	14., 15., (MP)
17. $\exists y_1 C(f(f(f(f(a)))), q_2, y_1)$	16., (EE)
18. $\exists x \exists y_1 C(x, q_2, y_1)$	17., (EE)

Die Substitutionen in dieser nicht ganz einfachen Ableitung sind, übersichtlich dargestellt, folgende:

Zeile	Substitution
6	$[a/x]$
7	$[f(f(a))/y_1]$
9	$[f(a)/x]$
10	$[f(a)/y_1]$
12	$[f(f(a))/x]$
13	$[a/y_1]$
15	$[f(f(f(a)))/x]$
17	$[a/y]$
18	$[f(f(f(f(a))))/x]$

Zeile (1) beschreibt die Konfiguration des Abacus zum Zeitpunkt 0; das ist die erste Prämisse der mit der Berechnung assoziierten Argumentform. Definitionsgemäß beschreibt sie die Konfiguration im Anfangszustand: Zum Zeitpunkt 0 steht der Abacus im Zustand q_0 mit der Zahl 3 im ersten Register. Zeilen (8), (11) und (14) beschreiben die Konfigurationen des Abacus zu den Zeitpunkten 1, 2 und 3. (☞ Können Sie sie entsprechend verbalisieren?) Zeile (16) beschreibt die Konfiguration im Zustand q_2, am Programmausgang: $\langle 4, q_2, 0\rangle$, d.h. zum Zeitpunkt 4 steht der Abacus im Zustand q_2 mit der Zahl 0 im ersten Register. Zeilen (1), (8), (11), (14) und (16) sind wiederum die Beschreibungen der Konfigurationen des Abacus zu den Zeitpunkten 0, 1, 2, 3 und 4, die gemäß Lemma 5.1 aus den Prämissen der mit der Berechnung assoziierten Argumentform folgen und daher auch daraus ableitbar sind. Die Ableitung der Konklusion aus den Prämissen der assoziierten Argumentform durchläuft also die Beschreibungen aller Konfigurationen, die in der Berechnung auftreten. — Zeile (18) ist die existenzielle Generalisierung der letzten Konfigurationsbeschreibung bezüglich Zeitpunkt und Inhalt des ersten Registers.

Wir haben jetzt eigentlich schon die Unentscheidbarkeit der Prädikatenlogik mit Funktionssymbolen gezeigt. Denn man kann mit dem bisher Gewonnenen argumentieren: Wenn die Prädikatenlogik entscheidbar ist — wenn es also ein Entscheidungsverfahren für die Prädikatenlogik gibt, mit dem man die Gültigkeit oder Ungültigkeit jeder Argumentform rein mechanisch in endlich vielen Schritten bestimmen kann — dann kann man (oder, wir wissen schon: könnte man) zu jeder Abacus-Berechnung die mit ihr assoziierte Argumentform aufstellen, diese mit dem Entscheidungsverfahren testen, und man wüsste somit von jeder Abacus-Berechnung, ob diese hält oder nicht. Das heißt aber nichts anderes als: Wenn die Prädikatenlogik (mit Funktionssymbolen) entscheidbar ist, dann ist das Halteproblem für den infiniten Abacus berechenbar.

Wenn Sie in dem letzten Satz des voraufgehenden Absatzes die zweite Prämisse unseres *modus tollens* auf S. 53 wiedererkannt haben, so liegen Sie völlig richtig. Wir werden das soeben Gesagte auch stringent beweisen (als Metatheorem 5.2 auf S. 77). Zuvor ersuche ich Sie, liebe/r Leser*in, noch um etwas Geduld, denn wir wollen unser schönes Resultat noch möglichst verallgemeinern und zeigen: Was für die Prädikatenlogik mit Funktionssymbolen gilt, gilt auch für prädikatenlogische Systeme ohne Funktionssymbole (und ohne Identität).

Wir wollen also als Nächstes zeigen, dass sich die assoziierten Argumentformen, die hier mit Funktionssymbolen gebildet wurden, sich in Argumentformen

ohne Funktionssymbole (und ohne Identität) überführen lassen, die sich bezüglich ihrer Gültigkeit ebenso verhalten. Damit wäre, gäbe es ein Entscheidungsverfahren für prädikatenlogische Argumentformen ohne Funktionssymbole (und ohne Identität), auch unter Zuhilfenahme dieser etwas einfacheren Systeme das Halteproblem berechenbar. — Machen wir uns also an die Eliminierung der Funktionssymbole.

5.6 Eliminierung der Funktionssymbole

Da wir den Beweis der Unentscheidbarkeit für die Prädikatenlogik so allgemein wie möglich führen und nicht auf die Prädikatenlogik mit Funktionssymbolen beschränkt bleiben wollen, werden wir in diesem Abschnitt die Funktionssymbole eliminieren. Dies ist ohne Einschränkung der Ausdrucksstärke des Logiksystems möglich, d.h. alle Argumente, die in Prädikatenlogik mit Funktionssymbolen formalisierbar sind, behalten ihre Eigenschaft der (Un-)Gültigkeit bei, wenn man sie überführt in Argumentformen ohne Funktionssymbole — die Sache wird nur etwas umständlicher. Wir werden in einem Zwischenschritt Prädikatenlogik ohne Funktionssymbole, dafür aber mit Identität machen, und dann in einem weiteren Schritt die Identität eliminieren. Doch verfahren wir langsam, der Reihe nach!

Führen wir zunächst eine zweistellige Relation ein, und zwar:

$N(\ldots,\ldots)$: … ist Nachfolger von ….

Die Relation ‚N‘ ist zunächst eine gewöhnliche zweistellige Relation, mit der man über beliebige Dinge aussagen kann, dass das eine ein Nachfolger des anderen sei. Wir müssen sie erst so einschränken, dass sie sich wie eine Funktion verhält; technisch gesehen sind Funktionen nämlich spezielle Relationen: Jede Funktion ist eine Relation, aber nicht jede Relation ist eine Funktion. (Ich hatte das, weil wir es eigentlich nicht gebraucht haben, übergangen.) In die Semantik der Funktionssymbole und in die Kalkülregeln sind nämlich in der Prädikatenlogik mit Funktionssymbolen einige wesentliche Eigenschaften von Funktionen, die wir in Abschnitt 2.1 kennengelernt haben, fix „eingebaut“, so z.B. dass eine Funktion für jedes Argument auch einen Wert hat (s.u., Theorem 5.1), und dass eine Funktion für ein Argument stets genau einen Wert hat und nicht mehrere (s.u., Theorem 5.2). (☞ Ist Ihnen aufgefallen, dass hingegen bei ‚N‘ oben „ist Nachfolger“ ohne bestimmten Artikel steht?)

Theorem 5.1. $\forall x \exists y\, y = f(x)$

Beweis.

1. $\forall x\, x = x$	(REF)
2. $f(a) = f(a)$	1., (UB)
3. $\exists y\, y = f(a)$	2., (EE)
4. $\forall x \exists y\, y = f(x)$	3., (UE)

□

Theorem 5.2. $\forall x \forall y \forall z((y = f(x) \wedge z = f(x)) \rightarrow y = z)$

Beweis.

1.	$\forall x \forall y((x = y \wedge x = c) \rightarrow y = c)$	(SUB)
2.	$\forall y((f(a) = y \wedge f(a) = c) \rightarrow y = c)$	1., (UB)
3.	$((f(a) = b \wedge f(a) = c) \rightarrow b = c)$	2., (UB)
4.	$\mid (b = f(a) \wedge c = f(a))$	(KB-Annahme)
5.	$\mid b = f(a)$	4., (SIMP1)
6.	$\mid \forall x \forall y((x = y \wedge x = x) \rightarrow y = x)$	(SUB)
7.	$\mid \forall y((b = y \wedge b = b) \rightarrow y = b)$	6., (UB)
8.	$\mid ((b = f(a) \wedge b = b) \rightarrow f(a) = b)$	7., (UB)
9.	$\mid \forall x\, x = x$	(REF)
10.	$\mid b = b$	9., (UB)
11.	$\mid (b = f(a) \wedge b = b)$	5., 10., (KON)
12.	$\mid f(a) = b$	11., 8., (MP)
13.	$\mid c = f(a)$	4., (SIMP2)
14.	$\mid \forall y((c = y \wedge c = c) \rightarrow y = c)$	6., (UB)
15.	$\mid ((c = f(a) \wedge c = c) \rightarrow f(a) = c)$	14., (UB)
16.	$\mid c = c$	9., (UB)
17.	$\mid (c = f(a) \wedge c = c)$	13., 16., (KON)
18.	$\mid f(a) = c$	17., 15., (MP)
19.	$\mid (f(a) = b \wedge f(a) = c)$	12., 18., (KON)
20.	$\mid b = c$	19., 3., (MP)
21.	$((b = f(a) \wedge c = f(a)) \rightarrow b = c)$	4.–20., (KB)
22.	$\forall z((b = f(a) \wedge z = f(a)) \rightarrow b = z)$	21., (UE)
23.	$\forall y \forall z((y = f(a) \wedge z = f(a)) \rightarrow y = z)$	22., (UE)
24.	$\forall x \forall y \forall z((y = f(x) \wedge z = f(x)) \rightarrow y = z)$	23., (UE)

□

Anmerkung: Theorem 5.2 ist eine Instanz der Drittengleichheit der Identität, näherhin ihrer Rechtskomparativität. (Einen Spezialfall der Linkskomparativität haben wir bereits in Zeile (3); wir verallgemeinern jedoch nicht diesen, sondern zeigen zweimal die Kommutativität der Identität, vgl. Nr. (3) u. (21).)

Legen wir also für unsere Nachfolgerrelation ‚N' die genannten und soeben auch bewiesenen allgemeinen Eigenschaften von Funktionen (Punkte (1) und (2) in der folgenden Liste) und einige spezielle Eigenschaften der Nachfolgerfunktion für natürliche Zahlen (die restlichen Punkte in der Liste) fest.

1. **Jede natürliche Zahl hat mindestens einen Nachfolger.** Dies ist die Eigenschaft von Funktionen, für jedes Argument mindestens einen Wert zu haben. Formal:

$$\forall x \exists y N(y, x)$$

2. **Der Nachfolger jeder natürlichen Zahl ist eindeutig.** Dies ist die Eigenschaft von Funktionen, für jedes Argument höchstens einen Wert zu haben. Formal:

$$\forall x \forall y \forall z((N(y, x) \wedge N(z, x)) \rightarrow y = z)$$

(Die ersten beiden Eigenschaften zusammen drücken aus, dass Funktionen für jedes Argument mindestens einen und höchstens einen, d.h. genau einen Wert haben.)

3. **Die Null ist nicht Nachfolger irgendeiner natürlichen Zahl.** Formal:

$$\neg\exists x N(a, x)$$

4. **Die Zahl, deren Nachfolger eine natürliche Zahl ist, ist eindeutig.** D.h. jede natürliche Zahl ist Nachfolger höchstens einer natürlichen Zahl. Formal:

$$\forall x \forall y \forall z((N(x, y) \wedge N(x, z)) \rightarrow y = z)$$

5. Schließlich, wenn wir wieder unsere Individuenkonstanten a mit unteren Indizes verwenden müssen, müssen wir angeben, wie diese **Individuenkonstanten in der Nachfolgerrelation** stehen. Daher machen wir die Nachfolgerrelation explizit, bis zur größten in unserer assoziierten Argumentform vorkommenden natürlichen Zahl:

$$(N(a_1, a) \wedge (N(a_2, a_1) \wedge (N(a_3, a_2) \wedge \ldots)))$$

Manchmal tragen nicht alle diese Formeln zur Gültigkeit einer mit einer Berechnung assoziierten Argumentform bei, aber wir wollen vereinbaren, dass wir jede davon zu den Prämissen jeder assoziierten Argumentform hinzufügen wollen. (Wenn es beim einen oder anderen Mal zu viele Prämissen sind, so werden dadurch schon mit weniger Prämissen gültige Argumentformen in keinem Falle wieder ungültig, aufgrund der Monotonie des logischen Folgens.)

Wir müssen auch unsere Programmbeschreibungen für die neue Nachfolgerrelation unschreiben, da uns ja nicht mehr das (zugegeben, bequemere) Funktionssymbol ‚f‘ zur Verfügung steht.

Die Programmbeschreibung unseres Nachfolgerprogramms bestand im letzten Abschnitt aus der einen Zeile:

$$\forall x \forall y_1(C(x, q, y_1) \rightarrow C(f(x), q_1, f(y_1)))$$

Wir müssen, um die Nachfolgerfunktion ‚f‘ mit der Nachfolgerrelation ‚N‘ auszudrücken, für jeden Funktionswert eine neue Variable einführen — für ‚$f(x)$‘ nehme ich hier ‚x_1‘, und für ‚$f(y_1)$‘ nehme ich ‚y_2‘. Dann müssen Bedingungen hinzukommen wie „Wenn x_1 Nachfolger von x ist, und wenn y_2 Nachfolger von y_1 ist, dann …“, sodass die Zeile der Programmbeschreibung wird zu:

$$\forall x \forall y_1 \forall x_1 \forall y_2(((N(x_1, x) \wedge N(y_2, y_1)) \wedge C(x, q, y_1)) \rightarrow C(x_1, q_1, y_2))$$

Allgemein gilt für die Eliminierung der Funktionssymbole aus der Programmbeschreibung:

- Jede Zeile einer Programmbeschreibung, die den Übergang über einen ‚+'-Befehl beschreibt, zwei neue Variablen enthalten: eine für den Nachfolger des Zeitpunkts, und eine für den Nachfolger des Registerinhalts, der beim Zustandsübergang verändert wird. (Letztere Variable wird im Nachglied der allquantifizierten Implikationsformel vorkommen.)

- Jede Zeile einer Programmbeschreibung, die den Übergang über einen ‚−'-Befehl beschreibt, wobei der Registerinhalt verringert wird, wird zwei neue Variablen enthalten: eine für den Nachfolger des Zeitpunkts, und eine für den Nachfolger des Registerinhalts, der bei dem Zustandsübergang verändert wird. (Letztere Variable wird im Vorderglied der allquantifizierten Implikationsformel vorkommen.)

- Jede Zeile einer Programmbeschreibung, die den Übergang über einen ‚−'-Befehl beschreibt, wobei der Registerinhalt nicht verringert werden kann, weil er schon 0 ist, wird eine neue Variable enthalten: für den Nachfolger des Zeitpunkts. (Eine zweite neue Variable brauchen wir in diesem Fall nicht, da sich bei einem solchen Zustandsübergang kein Registerinhalt verändert.)

Schreibt man die Formeln der Programmbeschreibung mit Funktionssymbolen auf diese Weise in Formeln ohne Funktionssymbole um und nimmt die eben genannten Formeln (1) bis (5) zu den Prämissen der assoziierten Argumentform dazu, so ergibt sich eine neue Argumentform. Diese neue Argumentform ist nun in einem prädikatenlogischen System ohne Funktionssymbole formalisierbar, aber von ihr gilt: Sie ist gültig gdw. die ursprüngliche Argumentform (mit Funktionssymbolen) gültig ist.

Sehen wir uns als ein Beispiel, in dem alle drei Fälle vorkommen, die mit der uns mittlerweile gut bekannten Berechnung der Summe von 5 und 3 mit unserem Summenprogramm assoziierte Argumentform an. In der hier nochmals engeführten Programmbeschreibung mit Funktionssymbolen ist die Nummerierung an die gleich folgende Argumentform angeglichen, sodass die Zeilennummern hier den Prämissennummern dort entsprechen. Dabei ist Zeile 4 ein Beispiel für den ersten Punkt, Zeilen 2 und 5 sind Beispiele für den zweiten Punkt, und Zeilen 3 und 6 sind Beispiele für den dritten Punkt.

2. $\forall x \forall y_1 \forall y_2 (C(x, q, y_1, f(y_2)) \rightarrow C(f(x), q_1, y_1, y_2))$
3. $\forall x \forall y_1 (C(x, q, y_1, a) \rightarrow C(f(x), q_3, y_1, a))$
4. $\forall x \forall y_1 \forall y_2 (C(x, q_1, y_1, y_2) \rightarrow C(f(x), q_2, f(y_1), y_2))$
5. $\forall x \forall y_1 \forall y_2 (C(x, q_2, y_1, f(y_2)) \rightarrow C(f(x), q_1, y_1, y_2))$
6. $\forall x \forall y_1 (C(x, q_2, y_1, a) \rightarrow C(f(x), q_3, y_1, a))$

Durch die Eliminierung der Funktionssymbole entsteht eine etwas kompliziertere Argumentform, deren Komponenten ich deshalb zunächst ein wenig erläutern möchte.

- Prämisse 1 ist die Beschreibung der Anfangskonfiguration; wir können sie lesen als „Zum Zeitpunkt 0 steht der Abacus im Zustand q_0 (d.i. der Programmeingang) mit der 5 im ersten und der 3 im zweiten Register."

- Prämissen 2 bis 6 sind die Programmbeschreibung, umgeschrieben mit der Nachfolgerrelation ‚N' anstatt der Nachfolgerfunktion ‚f'. Dabei wurde immer die Variable ‚x_1' verwendet für den Nachfolger des Zeitpunkts und die Variable ‚y_3' für das Register, dessen Inhalt sich verändert.

- Prämissen 7 bis 11 sind die Zusatzprämissen, die durch die Eliminierung der Funktionssymbole hinzugekommen sind. Dabei legen Prämissen 7 und 8 die Nachfolgerrelation als Funktion fest, Prämissen 9 und 10 sind speziell für die Nachfolgerfunktion, und Prämisse 11 ordnet die Individuenkonstanten bis zur größten in der Argumentform vorkommenden natürlichen Zahl (d.i. die 5, daher ‚a' bis ‚a_5') in die Nachfolgerrelation ein.

- Die Konklusion hat sich nicht verändert, da in ihr kein Funktionssymbol vorgekommen ist, das zu eliminieren gewesen wäre.

Die mit der Berechnung der Summe von 5 und 3 mit unserem Summenprogramm assoziierte Argumentform ist nach der Eliminierung der Funktionssymbole somit diese:

1. $C(a, q, a_5, a_3)$
2. $\forall x \forall y_1 \forall y_2 \forall x_1 \forall y_3 (((N(x_1, x) \wedge N(y_3, y_2)) \wedge C(x, q, y_1, y_3)) \rightarrow C(x_1, q_1, y_1, y_2))$
3. $\forall x \forall y_1 \forall x_1 ((N(x_1, x) \wedge C(x, q, y_1, a)) \rightarrow C(x_1, q_3, y_1, a))$
4. $\forall x \forall y_1 \forall y_2 \forall x_1 \forall y_3 (((N(x_1, x) \wedge N(y_3, y_1)) \wedge C(x, q_1, y_1, y_2)) \rightarrow C(x_1, q_2, y_3, y_2))$
5. $\forall x \forall y_1 \forall y_2 \forall x_1 \forall y_3 (((N(x_1, x) \wedge N(y_3, y_2)) \wedge C(x, q_2, y_1, y_3)) \rightarrow C(x_1, q_1, y_1, y_2))$
6. $\forall x \forall y_1 \forall x_1 ((N(x_1, x) \wedge C(x, q_2, y_1, a)) \rightarrow C(x_1, q_3, y_1, a))$
7. $\forall x \exists y N(y, x)$
8. $\forall x \forall y \forall z ((N(y, x) \wedge N(z, x)) \rightarrow y = z)$
9. $\neg \exists x N(a, x)$
10. $\forall x \forall y \forall z ((N(x, y) \wedge N(x, z)) \rightarrow y = z)$
11. $(N(a_1, a) \wedge (N(a_2, a_1) \wedge (N(a_3, a_2) \wedge (N(a_4, a_3) \wedge N(a_5, a_4)))))$

$\exists x \exists y_1 \exists y_2 C(x, q_3, y_1, y_2)$

Diese Argumentform ist gültig. (Wir ersparen uns die Ableitung der Prämissen aus der Konklusion. ☞ Wenn Sie lange Ableitungen mögen, können Sie das natürlich gerne versuchen.)

5.7 Eliminierung der Identität

Wenn Sie aufmerksam bis hierher gelesen haben, ist Ihnen bestimmt aufgefallen: Wir haben uns durch die Eliminierung der Funktionssymbole zunächst ein neues Symbol „eingehandelt", das nicht in jedem prädikatenlogischen System vorkommt, nämlich das Identitätszeichen ‚$=$'. Um möglichst sparsam und gerade dadurch möglichst allgemein zu sein, und die Unentscheidbarkeit der Prädikatenlogik ohne Funktionssymbole und ohne Indentität zu zeigen, werden wir die Identität auch noch eliminieren. Dabei folgen wir Willard Van Orman Quine in [Qui70, S. 63].

Die Grundidee ist die alte Leibnizsche Vorstellung von der Ununterscheidbarkeit des Identischen: Wenn x und y identisch sind, dann ist eine Aussage über x wahr gdw. die entsprechende Aussage über y wahr ist. Etwas technischer ausgedrückt: Wenn x und y identisch sind, dann sind x und y an allen Stellen aller Prädikate *salva veritate* (d.h. unter Erhaltung des Wahrheitswertes) austauschbar.

Als *ad hoc*-Lösung, wenn die vorkommenden Prädikate bekannt sind, ist damit in einem Logiksystem ohne Identität ein „brauchbares Faksimile“ ([Qui70, S. 63], Übers. M.M.) des Identitätsprädikats definierbar. Wir wählen als Surrogat für das Identitätsprädikat irgendein zweistelliges Prädikat aus, z.B. ‚I‘, und dann formulieren wir eine Allformel, die für alle x und für alle y $I(x, y)$ definiert als: Wenn wir eine Prädikation mit irgendeinem Prädikat, das in unserer Argumentform vorkommt, haben,

- in der ‚x‘ vorkommt, und
- in der wir genau ein Vorkommnis von ‚x‘ durch ‚y‘ ersetzen, sowie
- an allen anderen Stellen des Prädikats exakt dieselben Terme belassen,

dann bekommen wir eine neue Prädikation, die denselben Wahrheitswert hat wie die ursprüngliche (also die mit ‚x‘ darin).

Für ein einstelliges Prädikat ‚F‘ (wir haben es zwar nicht in unserem Formalisierungsschlüssel, aber dies ist einer der einfachsten Fälle) würde diese Allformel wie folgt aussehen:

$$\forall x \forall y (I(x, y) \leftrightarrow (F(x) \leftrightarrow F(y)))$$

Für ein zweistelliges Prädikat, hier das ‚N‘, das wir aus unseren mit Berechnungen assoziierten Argumentformen kennen:

$$\forall x \forall y (I(x, y) \leftrightarrow \forall z ((N(x, z) \leftrightarrow N(y, z)) \wedge (N(z, x) \leftrightarrow N(z, y))))$$

Die Formel besagt, dass identische x und y sowohl an der ersten Stelle von N als auch an der zweiten Stelle von N *salva veritate* austauschbar sind — und zwar für alle z, die an der jeweils anderen Stelle von N vorkommen und beim Austausch nicht verändert werden.

Da man in einer endlich langen Formel — und andere kennen wir hier nicht — natürlich nur Aussagen über einen solchen Austausch *salva veritate* an den Stellen endlich vieler Prädikate machen kann, wird klar, warum ich diese Lösung vorhin als „*ad hoc*“ bezeichnet habe: Denn man muss die Austauschbarkeit der Identischen *salva veritate* für all jene Prädikate formulieren, die in der Argumentform vorkommen, ohne sie für alle unendlich vielen Prädikate zugleich formulieren zu können.

Die Formel mit dem zweistelligen Prädikat ‚N‘ weist also zwar schon den Weg Richtung Eliminierung der Identität aus unseren mit Berechnungen assoziierten Argumentformen, aber wir sind damit noch nicht fertig: Denn es gibt in unseren assoziierten Argumentformen noch das zumindest dreistellige Prädikat ‚C‘. Auch für dieses müssen wir natürlich in der Definition unseres Identitäts-Faksimiles Sorge tragen — für dreistelliges ‚C sieht unsere Allformel so aus (ich habe sie, um der besseren Lesbarkeit willen, mehrzeilig aufgeteilt):

$$\begin{aligned}
&\forall x \forall y (I(x,y) \leftrightarrow \\
&\quad (\forall z((N(x,z) \leftrightarrow N(y,z)) \wedge (N(z,x) \leftrightarrow N(z,y)))) \wedge \\
&\quad \forall z_1 \forall z_2 (((C(x,z_1,z_2) \leftrightarrow C(x,z_1,z_2)) \wedge \\
&\qquad (C(z_1,x,z_2) \leftrightarrow C(z_1,y,z_2))) \wedge \\
&\qquad (C(z_1,z_2,x) \leftrightarrow C(z_1,z_2,y))))
\end{aligned}$$

Die Idee ist, wie Quine [Qui70, S. 63] schreibt, „die Erschöpfung der Kombinationen“ (Übers. M.M.): Die Austauschbarkeit *salva veritate* für jedes Prädikat an jeder seiner Stellen in die Allformel aufzunehmen. Dann verhält sich ‚I‘ in Bezug auf die Prädikate, für die es definiert ist, wie Identität.

Nehmen wir jeweils eine solche Formel zu unseren (mittlerweile von den Funktionssymbolen befreiten) Argumentformen hinzu und verwenden das Prädikat ‚I‘ anstelle von ‚=‘, so ist die resultierende Argumentform ohne Identität gültig gdw. die Argumentform mit Identität gültig ist.

5.8 Zurück zum Halteproblem

Schlagen wir nun die Brücke zurück zum Halteproblem für den infiniten Abacus. Mit Metatheorem 5.1 kann man nämlich zeigen:

Metatheorem 5.2. Wenn die Prädikatenlogik entscheidbar ist, dann ist das Halteproblem für den infiniten Abacus berechenbar.

Beweis. Wir beweisen zunächst durch Konstruktion: Man konstruiere folgenden Algorithmus für ein Abacus-Programm und die Beschreibung der Anfangskonfiguration einer Berechnung:

1. Stelle zu dem Abacus-Programm und der Berechnung die mit dieser assoziierte Argumentform auf.
2. Entscheide, ob die Argumentform gültig ist, oder nicht.
3. Liefere als Resultat zurück:
   (a) „Die Berechnung hält“, falls die Argumentform gültig ist, oder
   (b) „Die Berechnung hält nicht“, falls die Argumentform ungültig ist.

Nr. (1) ist algorithmisch kein Problem: Die Beschreibung der Anfangskonfiguration ist ein einfacher atomarer Satz, und in Abschnitt 5.3 haben wir informell eine Vorschrift angegeben, wie man die Beschreibung eines Programms rein mechanisch in endlich vielen Schritten aus seinem Graphen gewinnen kann. Es liegt also an Nr. (2): Wenn es für Nr. (2) einen Algorithmus gibt, d.h. die Prädikatenlogik entscheidbar ist, dann ist das Halteproblem für den infiniten Abacus (auf dem Umweg der soeben gezeigten Konstruktion) berechenbar. □

Metatheorem 5.2 ist die zweite Prämisse unseres *modus tollens* auf S. 53. Somit sind beide Prämissen unseres „Kernarguments“ etabliert.

Und zwar gilt Metatheorem 5.2 zunächst für prädikatenlogische Systeme mit Funktionssymbolen. Denn wir haben gezeigt: Wenn die Prädikatenlogik mit Funktionssymbolen entscheidbar ist, dann ist das Halteproblem für den infiniten Abacus berechenbar. Dann haben wir die Funktionssymbole eliminiert und

dafür die Identität „eingekauft“, wobei die gültigen Argumentformen gültig und die ungültigen ungültig geblieben sind. Dadurch war gezeigt: Wenn die Prädikatenlogik (ohne Funktionssymbole) mit Identität entscheidbar ist, dann ist das Halteproblem für den infiniten Abacus berechenbar.

Dann haben wir auch die Identität eliminiert (sodass wiederum alle Argumentformen ihren Status der (Un-)Gültigkeit beibehalten haben) und sind bei dem prädikatenlogischen Standardsystem mit den wenigsten Zeichen angekommen: der Prädikatenlogik ohne Identität und ohne Funktionssymbole. Und somit ist die erste Prämisse unseres „Kernarguments“, dem *modus tollens* auf S. 53, nämlich der Konditionalsatz in möglichst allgemeiner Form etabliert:

Wenn die Prädikatenlogik (ohne Funktionssymbole und ohne Identität) entscheidbar ist, dann ist das Halteproblem für den infiniten Abacus berechenbar.

Kapitel 6

Zusammenfassung und Diskussion

6.1 Zusammenfassung

Wir sind nun fertig: Wir haben den Satz von Alonzo Church bewiesen. Die Prädikatenlogik (ohne Funktionssymbole und ohne Identität) ist unentscheidbar.

Unser Kernargument war folgender *modus tollens* (wir sind ihm auf S. 53 erstmals begegnet):

1. Wenn die Prädikatenlogik entscheidbar ist, dann ist das Halteproblem für den infiniten Abacus berechenbar.
2. Das Halteproblem für den infiniten Abacus ist nicht berechenbar.

Die Prädikatenlogik ist nicht entscheidbar.

Prämisse 1 (der Konditionalsatz) ist Metatheorem 5.2 von S. 77, für die wir in Kapitel 5 argumentiert haben; Prämisse 2 (die mit der Verneinung darin) ist Metatheorem 4.1 von S. 52, dessen Beweis Kapitel 4 gewidmet war. Das Argument ist klarerweise gültig (da es ein *modus tollens* ist, dessen Gültigkeit sie bereits aus Ihrer Ausbildung aus Elementarer Logik kennen), und da seine Prämissen sämtlich wahr sind, ist es auch stichhaltig. Somit ist seine Konklusion wahr, und es gilt:

> Es gibt kein Entscheidungsverfahren für die Prädikatenlogik, d.h. keinen Algorithmus, mittels dessen von jeder prädikatenlogischen Argumentform rein mechanisch in endlich vielen Schritten feststellbar wäre, ob sie gültig ist oder nicht.

6.2 Betroffene Logiksysteme

Auf welche Logiksysteme trifft der Satz von Church nun genauer zu? Es ist aus dem voraufgehenden Kapitel klar, dass er zutrifft auf

- Prädikatenlogik mit Funktionssymbolen,

- Prädikatenlogik mit Identität (ohne Funktionssymbole), sowie
- Prädikatenlogik ohne Funktionssymbole und ohne Identität.

Allgemein sind zumindest all jene Logiksysteme unentscheidbar, in deren Sprache mit Abacus-Berechnungen assoziierte Argumentformen repräsentierbar (formalisierbar) sind. Das sind insbesondere konservative Erweiterungen der Prädikatenlogik, also Logiksysteme, in denen alle in prädikatenlogischen Standardsystemen (wie wir sie verwendet haben) (un-)gültigen Argumentformen ihren Status der (Un-)Gültigkeit beibehalten.

Nicht betroffen von dieser Variante des Unentscheidbarkeitsbeweises sind zumindest all jene Logiksysteme, in deren Sprache mit Abacus-Berechnungen assoziierte Argumentformen nicht so feinkörnig repräsentierbar (formalisierbar) sind, dass ihre Gültigkeit mit dem Halten der jeweiligen Berechnung koinzidiert. Dies sind beispielsweise:

- die Aussagenlogik, sowie
- die sog. „monadische Prädikatenlogik“, d.h. jenes Fragment der Prädikatenlogik, das ausschließlich einstellige Prädikate verwendet.

Für die Aussagenlogik kennen wir Entscheidungsverfahren, z.B. die Methode der Wahrheitstafeln (die zwar sehr langwierig werden kann, aber immer nach endlich vielen Schritten zu einem Ende kommt), die semantischen Tableaux nach Evert Willem Beth oder der mit dieser Methode verwandte Baumkalkül. Daraus erhellt, nebenbei bemerkt, auch: Die Aussagenlogik ist entscheidbar, die Prädikatenlogik nicht — also ist Prädikatenlogik nicht auf Aussagenlogik reduzierbar; insbesondere sind Quantifikationen nicht als laufende Konjunktionen bzw. Disjunktionen aufzufassen (vgl. [Jac04, 112f.]).

Eine gewisse Überraschung mag es sein, dass es auch für die monadische Prädikatenlogik Entscheidungsverfahren gibt (eines davon kann beispielsweise nachgelesen werden in [Bos97, S. 115–126]). Das bedeutet, analog zu dem Argument am Ende des voraufgehenden Abschnitts: Die monadische Prädikatenlogik ist entscheidbar, die polyadische Prädikatenlogik nicht („polyadisch“ bedeutet: mit mehrstelligen Prädikaten) — also ist polyadische Prädikatenlogik nicht auf monadische Prädikatenlogik reduzierbar. Das ist ein starker Hinweis darauf, dass es für die voll entfaltete Prädikatenlogik irgendwie „wesentlich“ ist, dass sie auch mehrstellige Prädikate kennt.

Kalmár László hat im Jahre 1936 bewiesen, dass in einem gewissen Sinne sogar ein einziges zweistelliges Prädikat ausreicht, um nicht nur die Unentscheidbarkeit zu bekommen, sondern auch die ganze Ausdruckskraft der polyadischen Prädikatenlogik bereits voll zu entfalten, vgl. [Kal36]: Zu jeder Formel gibt es nämlich eine erfüllbarkeitsäquivalente Formel von einer gewissen Form und mit nur einem einzigen zweistelligen und keinem einzigen mehrstelligen Prädikat. — „Erfüllbarkeitsäquivalent“ bedeutet dabei: Zwei Formeln sind *erfüllbarkeitsäquivalent* gdw. die eine erfüllbar ist gdw. die andere erfüllbar ist. Damit ist zu jeder Argumentform eine prädikatenlogische Formel mit höchstens einem zweistelligen und keinem einzigen mehrstelligen Prädikat konstruierbar, die erfüllbar ist gdw. die Argumentform ungültig ist (vgl. Punkt (4) auf S. 8).

Die Bücher [Nol97] und [Bos97] führen noch einige weitere entscheidbare Fragmente der Prädikatenlogik an. Die zu ihrer Zeit (1997, laut Rückentext)

umfassendste Sammlung entscheidbarer wie unentscheidbarer Fragmente der Prädikatenlogik ist:

Börger, Egon / Grädel, Erich u. Gurevich, Yuri: *The Classical Decision Problem*, Berlin u.a. 1997, Nachdr. 2001.

6.3 Unentscheidbar? Semi-entscheidbar!

Wir haben in diesem Kurs gesehen: Die Prädikatenlogik ist unentscheidbar, d.h. es gibt keinen Algorithmus, mittels dessen man auf rein mechanische Art und Weise in endlich vielen Schritten feststellen könnte, ob eine Argumentform gültig ist oder nicht. Aber es gibt dennoch eine Möglichkeit der (in gewisser Weise) systematischen Suche nach Ableitungen von irgendwelchen Konklusionen aus Prämissen, die, wenn man ihr lange genug Zeit lässt, keine einzige gültige Argumentform auslässt. Wie passt das zur Unentscheidbarkeit der Prädikatenlogik?

Um das zu ergründen, erweitern wir zunächst unseren bereits bestens bekannten infiniten Abacus um ein Peripheriegerät, nämlich ein Ausgabegerät, und zwar einen Drucker. Auf diesem Drucker kann unser Abacus Argumentformen ausdrucken.

Dann stellen wir fest, dass Ableitungen, recht ähnlich wie unsere Abacus-Programme in Abschnitt 4.5, gödelisierbar sind: Jeder Ableitung kann man jeweils eine natürliche Zahl zuordnen, sodass

- verschiedene Ableitungen verschiedene Gödelzahlen haben,
- man von jeder natürlichen Zahl in endlich vielen Schritten feststellen kann, ob sie Gödelzahl einer Ableitung ist oder nicht, und
- jede Ableitung aus ihrer Gödelzahl in endlich vielen Schritten rekonstruierbar ist.

Eine Gödelisierung der Ableitungen im formalen System der *Principia Mathematica* war eine der wesentlichen Leistungen des hier schon öfters erwähnten Kurt Gödel in [Göd86], wo er das Verfahren erfunden / entdeckt hat. [Göd86, S. 186, Nr. 45] ist bei Gödel das rekursive Prädikat, das auf die Gödelzahlen aller Beweise (im System der *Principia Mathematica* sind das Ableitungen nur aus Axiomen) und nur auf diese zutrifft. — Unter „Ableitung“ ist hier übrigens stets eine fehlerfreie und vollständig den Kalkülregeln entsprechende Ableitung zu verstehen, nicht etwa ein Gebilde, das einer Ableitung vielleicht sogar täuschend ähnlich sieht, aber in irgendwelchen Schritten nicht hundertprozentig den vereinbarten Regeln entspricht. Gebilde der letzteren Art interessieren hier überhaupt nicht, sodass wir nicht einmal einen Namen für sie brauchen (im Gegensatz beispielsweise zu einer Logik-Übung für Anfänger*innen, wo es Aufgabe des oder der Lehrenden ist, zwischen Gebilden der ersten Art und Gebilden der zweiten Art zu unterscheiden und diesen Unterschied auch zu lehren).

Jetzt fügen wir diese beiden Dinge — den Abacus mit dem angeschlossenen Drucker und unser Wissen über die Gödelisierung von Ableitungen — zusammen und programmieren unseren Abacus mit dem Drucker wie folgt:

1. Wir schreiben ein Programm P_{GTA} („GTA“ für Gödel-Test-Ableitung), das testet, ob eine natürliche Zahl Gödelzahl einer Ableitung ist.

2. Wenn das der Fall ist, rekonstruiert das Programm P_{GTA} die Ableitung aus der Gödelzahl und druckt die Argumentform, die durch die Ableitung als gültig erwiesen ist, aus. (Von der Ableitung zur Argumentform zu kommen, ist überhaupt kein Problem: Prämissen sind all jene Annahmen, die nicht beseitigt werden, und die Konklusion ist immer die Formel in der letzten Zeile der Ableitung.)

3. Wir implementieren einen Zähler in einer äußeren Schleife, sodass das Programm P_{GTA} nacheinander die natürlichen Zahlen 0, 1, 2, 3, ... testet, immer weiter, eine Zahl nach der anderen. (Unser Programm läuft also potenziell unendlich lange.)

Das hier vorgestellte Programm hält nicht (übrigens ein weiteres Beispiel für ein Programm, bei dem ein Halten nicht erwünscht ist, vgl. die Anmerkung auf S. 53), sondern es druckt jedes Mal, wenn es die Gödelzahl einer Ableitung findet, die dazugehörige Argumentform aus. Einen solchen Abacus mit Drucker zusammen mit seinem Programm wollen wir einen *Aufzähler* nennen — also haben wir hier einen Aufzähler für die gültigen Argumentformen der Prädikatenlogik vorliegen.

Da nun jede Ableitung eine Gödelzahl hat, wird unser Aufzähler jede gültige Argumentform irgendwann einmal ausdrucken! Diese Eigenschaft der Menge der gültigen Argumentformen der Prädikatenlogik wird manchmal so ausgedrückt: „Die Menge der gültigen Argumentformen der Prädikatenlogik ist *rekursiv aufzählbar*." (Wenn wir uns erinnern, dass Abacus-Programme genau dasselbe leisten wie rekursive Funktionen, dann macht diese Bezeichnung durchaus auch für einen Abacus Sinn.) — Wie geht das aber mit der Unentscheidbarkeit der Prädikatenlogik zusammen?

Das geht insofern, als die Menge der gültigen Argumentformen der Prädikatenlogik zwar rekursiv aufzählbar ist, aber die Menge der ungültigen Argumentformen der Prädikatenlogik dies nicht ist. Wir können nämlich kein Programm für unseren Abacus mit Drucker konstruieren, das alle ungültigen Argumentformen der Prädikatenlogik ausdruckt. Anders ausgedrückt: Es gibt keinen Aufzähler für die ungültigen Argumentformen der Prädikatenlogik.

Wir könnten zwar versuchen, einen Aufzähler für die ungültigen Argumentformen der Prädikatenlogik wie folgt zu konstruieren: Wir drucken jede Argumentform aus, die unser Aufzähler für die gültigen Argumentformen nicht ausdruckt. — Aber dabei geraten wir sofort in eine fatale Schwierigkeit: Wenn eine Argumentform nicht gedruckt wurde, wissen wir nicht:

- Ist die Argumentform tatsächlich ungültig, und wird sie nie auf die Liste der gültigen Argumentformen gedruckt — oder

- ist die Argumentform *doch* gültig, nur ist ihre Gödelzahl größer als die bisher durch das Programm P_{GTA} getesteten Zahlen? (D.h. ist die Argumentform gültig, aber sie wurde *noch* nicht gedruckt?)

D.h. wenn wir nicht wissen, ob eine Argumentform gültig ist oder nicht, dann wissen wir nicht, wie lange wir warten müssen, ob sie doch noch gedruckt wird und damit gültig ist, oder wann wir aufhören können zu warten, weil sie sicher nie gedruckt wird. Obwohl wir auf jede positive Antwort, also dass eine Argumentform gültig ist, immer nur endliche Zeit warten müssen, ist das

nicht der Fall für negative Antworten (also dass eine Argumentform ungültig ist): Um auf die negative Antwort bezüglich einer bestimmten Argumentform schließen zu können, müssten wir die unendliche Zeit abwarten, bis alle gültigen Argumentformen gedruckt sind, was natürlich unmöglich ist — und von einem Entscheidungsverfahren erwarten wir sowohl die positive als auch die negative Antwort nach endlicher Zeit.

Wenn eine Argumentform also nicht gedruckt wurde, dann gibt es keine natürliche Zahl, für die gilt: Nach dem Test dieser Zahl wird die Argumentform sicher nicht mehr gedruckt, also kommt sie nicht auf der Liste der gültigen Argumentformen vor, also ist sie ungültig. Das bedeutet:

1. Es gibt keine obere Schranke für die Gödelzahl einer Ableitung zu einer jeweilgen Argumentform.
2. Wenn die Ableitungen nach der Größe ihrer Gödelzahlen aufsteigend geordnet werden, so hat diese Liste keinerlei Systematik.

Nr. (2) ist eine Folgerung aus Nr. (1), denn wenn die Liste eine Systematik hätte, so wäre eine obere Schranke für die Gödelzahl einer Ableitung bestimmbar. Und wer einige Ableitungen gültiger prädikatenlogischer Argumentformen durchgeführt hat, wird auch die Wahrheit der beiden Sätze durch Erfahrung bestätigt wissen. Denn es gibt alle vier der folgenden Fälle:

- einfache Argumentformen mit kurzen Ableitungen, die also eine recht kleine Gödelzahl ihrer Ableitung erwarten lassen,
- einfache Argumentformen mit (überraschend) langen Ableitungen, die also eine große Gödelzahl ihrer Ableitung erwarten lassen,
- komplizierte Argumentformen mit (überraschend) kurzen Ableitungen, die also eine recht kleine Gödelzahl ihrer Ableitung erwarten lassen, und schließlich auch
- komplizierte Argumentformen mit langen Ableitungen, die also eine große Gödelzahl ihrer Ableitung erwarten lassen.

Kurz gesagt: Aus der Argumentform alleine kann in keiner Weise auf die Einfachheit oder Kompliziertheit, und damit die Größenordnung der Gödelzahl, der Ableitung der Konklusion aus den Prämissen geschlossen werden.

Wir haben also gesehen:

- Die Menge der gültigen prädikatenlogischen Argumentformen ist rekursiv aufzählbar.
- Die Menge der *un*gültigen prädikatenlogischen Argumentformen (das Komplement der Menge der gültigen prädikatenlogischen Argumentformen) ist *nicht* rekursiv aufzählbar.

Definition 6.1. Eine Menge ist *semi-entscheidbar* gdw. sie rekursiv aufzählbar ist, ihr Komplement aber nicht rekursiv aufzählbar ist.

Definition 6.2. Eine Menge ist *rekursiv* gdw. sowohl sie als auch ihr Komplement rekursiv aufzählbar sind.

Anmerkung: „Rekursiv“ bedeutet nicht dasselbe wie „rekursiv aufzählbar“, das zusätzliche Wort macht einen wesentlichen Unterschied! Denn eine Menge kann rekursiv aufzählbar sein, ohne rekursiv zu sein (nämlich genau dann, wenn zwar sie, nicht aber ihr Komplement rekursiv aufzählbar ist.)

Metatheorem 6.1. Eine Menge ist rekursiv gdw. sie entscheidbar ist.

Beweis. Erste Richtung: „Wenn eine Menge rekursiv ist, dann ist sie entscheidbar.“

1. * Sei M eine rekursive Menge.
2. Dann ist sowohl M als auch auch ihr Komplement $\overline{M}$ rekursiv aufzählbar.
3. Dann tritt jedes in Frage kommende Element nach endlicher Zeit entweder in der Aufzählung von M oder in der Aufzählung von $\overline{M}$ auf.
   Denn dann können wir zwei Aufzähler konstruieren, einen für M und einen für $\overline{M}$. Diese lassen wir abwechselnd je ein Element von M und ein Element von $\overline{M}$ drucken: Zu irgendeinem Zeitpunkt wird das in Frage kommende Element auf einer der beiden Listen auftauchen.
4. Von jedem in Frage kommenden Element ist so in endlicher Zeit feststellbar, ob es zu M oder zu $\overline{M}$ (also nicht zu M) gehört.
5. Also ist M entscheidbar.

Zweite Richtung: „Wenn eine Menge entscheidbar ist, dann ist sie rekursiv.“

1. * Sei M eine entscheidbare Menge.
2. Also ist von jedem in Frage kommenden Element in endlich vielen Schritten feststellbar, ob es zu M gehört oder zu ihrem Komplement $\overline{M}$.
3. Also sind sowohl M als auch $\overline{M}$ rekursiv aufzählbar.
   Denn wir brauchen nur die in Frage kommenden Elemente der Reihe nach durchzugehen: Wenn wir feststellen, dass ein Element zu M gehört, dann schreiben wir es auf eine Liste. So haben wir einen Aufzähler für M. Und wenn wir die in Frage kommenden Elemente der Reihe nach durchgehen und feststellen, dass ein Element nicht zu M gehört (sondern zu $\overline{M}$), dann schreiben wir es auf eine (andere) Liste. So haben wir auch einen Aufzähler für $\overline{M}$.
4. Also ist M rekursiv.

□

Das bedeutet nun für die Prädikatenlogik: Die Menge der gültigen Argumentformen ist nicht entscheidbar, also auch nicht rekursiv. Die gültigen Argumentformen der Prädikatenlogik sind allerdings rekursiv aufzählbar, während die ungültigen dies nicht sind. Die Prädikatenlogik ist also semi-entscheidbar.

6.4 Unentscheidbarkeit als ein weiteres Halteproblem

Wir haben zwar gesehen, dass die Prädikatenlogik semi-entscheidbar ist in dem Sinne, dass die gültigen Argumentformen rekursiv aufzählbar sind und die ungültigen nicht, aber auch unser Aufzähler für die gültigen Argumentformen erlaubt keine Lösung des Entscheidungsproblems. Denn wenn wir nicht wissen, ob eine bestimmte Argumentform gültig ist oder nicht, dann können wir von unserem Aufzähler höchstens nach endlich vielen Schritten erfahren, dass sie gültig ist — aber wenn die Argumentform zu einem bestimmten Zeitpunkt nicht gedruckt wurde, dann wissen wir nicht, ob sie in Zukunft noch gedruckt werden wird und somit gültig ist, oder ob sie nie gedruckt werden wird, weil sie ungültig ist.

Ändern wir unseren Aufzähler für die gültigen Argumentformen der Prädikatenlogik ein wenig ab, sodass er als Input die Gödelzahl einer Argumentform verarbeitet, und dass er unmittelbar, nachdem er die dieser Gödelzahl entsprechende Argumentform gedruckt hat, hält. (D.h. Er druckt nicht immer potenziell unendlich lang eine Argumentform nach der anderen aus, sondern er hält, unmittelbar nachdem er die eine Argumentform, nach der er „gefragt“ wurde, als gültig erwiesen hat — wenn das jemals der Fall ist.) Dann haben wir einen Algorithmus, der hält gdw. die in Frage kommende Argumentform gültig ist. Die Frage nach der Gültigkeit bzw. Ungültigkeit einer Argumentform wurde hier wiederum in ein Halteproblem übergeführt — ein weiteres, wie aus der Unentscheidbarkeit der Prädikatenlogik unmittelbar folgt, algorithmisch unlösbares Halteproblem.

Denn wenn dieses Halteproblem algorithmisch lösbar wäre, dann könnten wir in jedem Fall feststellen: Entweder, der modifizierte Aufzähler hält, also ist die Argumentform, nach deren Ableitung wir mit seiner Hilfe suchen, gültig. Oder, der modifizierte Aufzähler hält nicht, also ist die Argumentform, nach deren Ableitung wir mit seiner Hilfe suchen, ungültig. — Das wäre aber wiederum ein Entscheidungsverfahren für die Prädikatenlogik, das es gemäß dem Satz von Church gerade nicht geben kann.

Die im voraufgehenden Absatz formulierte Argumentation ist eine durchaus gängige, wenn es um Unentscheidbarkeit geht: Man reduziert ein Problem auf ein anderes, bekanntermaßen unentscheidbares Problem und zeigt somit, dass das ursprüngliche Problem unentscheidbar ist. So hatten wir ursprünglich das Entscheidungsproblem für die Prädikatenlogik auf das Halteproblem für den infiniten Abacus reduziert, für das es keine algorithmische Lösung gibt, und auf genau analoge Weise haben wir gerade gezeigt: Wenn das Halteproblem für den modifizierten Aufzähler lösbar ist, dann ist die Prädikatenlogik entscheidbar. Wir wussten aber schon: Die Prädikatenlogik ist aber nicht entscheidbar. So konnten wir schließen: Also ist das Halteproblem für den modifizierten Aufzähler prädikatenlogischer Argumentformen algorithmisch unlösbar.

6.5 Kommentare zum Diagonalverfahren und zum indirekten Beweis

Unser Unentscheidbarkeitsbeweis enthielt in seinem Kern einen Diagonalbeweis (vgl. Kapitel 4 und insbes. Abschnitt 4.7), und Diagonalbeweise haben immer wieder Skeptiker*innen auf den Plan gerufen, so z.B. den polnischen Philosophen und Logiker Leon Gumański (1921–2014). Gumański im Speziellen war in dieser Hinsicht besonders radikal: Er zweifelte nicht nur an der Unentscheidbarkeit der Prädikatenlogik und den dazugehörogen Beweisen, sondern er war der Auffassung und vermeinte sogar bewiesen zu haben, dass die Prädikatenlogik ganz entgegen Church und allen anderen *doch* entscheidbar sei! Zu diesem Behufe stellte er mindestens ein vermeintliches Entscheidungsverfahren vor, vgl. z.B. [Gum02a]. (Gumański hatte allerdings zumindest in dem Punkt Unrecht, dass sein vermeintliches Entscheidungsverfahren gar keines ist, weil es nicht korrekt ist: Es weist nämlich mindestens eine gültige Argumentform als ungültig aus.)

Diagonalbeweise sind — und das ist immer wieder ein spezieller Angriffspunkt für Skepsis gewesen — stets indirekte Beweise: Eine komplementierende Selbstanwendung führt zu einer Instanz der Negation des Thomson-Theorems, die als Instanz eines logisch falschen Satzes selbst logisch falsch sein muss. Dann wird geschlossen: Wenn aus der Annahme, die Diagonalfolge gehöre zu der ursprünglichen Gesamtheit, dieser logisch falsche Satz folgt, dann kann die Diagonalfolge eben nicht zu der ursprünglichen Gesamtheit gehören. (Die Diagonalzahl kann nicht nicht Element der tentativen Liste der reellen Zahlen sein, vgl. Abschnitt 4.4; das Programm P^*_{halt} kann nicht zu den Abacus-Programmen gehören, vgl. Abschnitte 4.6 u. 4.7 ....) So konnten u.a. Vertreter*innen einer skeptischen Einstellung in der Philosophie der Formalwissenschaften, die das Verfahren des indirekten Beweises ablehnen (etwa Intuitionist*innen), eine Zeitlang an genau diesem Punkt einhaken, solange alle Unentscheidbarkeitsbeweise für die Prädikatenlogik Diagonalbeweise waren — in dem Sinne, dass nur Unentscheidbarkeitsbeweise mit dieser Struktur bekannt waren.

Denn wer — aus welchem Grund auch immer — einem Beweisverfahren misstraut, muss natürlich nicht alle Sätze für wahr halten, die mittels des in Frage stehenden Verfahrens bewiesen (der/die Skeptiker*in würde hier auf Anführungszeichen bestehen: „bewiesen“) wurden. Tatsächlich gab es immer wieder Personen, die den Satz vor Church wegen des Diagonalverfahrens (im Allgemeinen, oder im Speziellen, weil es ein indirekter Beweis ist) anzweifelten oder zumindest nicht für schlüssig bewiesen hielten. So der bereits erwähnte Gumański: „[D]espite its distinguished credentials the [diagonal] method is unreliable and all the purported proofs in which it is employed — though not necessarily their theses — ought to be doomed to oblivion.“ [Gum02b, S. 29]. Der bereits erwähnte Kalmár László veröffentlichte allerdings im Jahre 1956 einen direkten Beweis für die Unentscheidbarkeit der Prädikatenlogik mit Identität:

Kalmár, László: *Ein direkter Beweis für die allgemein-rekursive Unlösbarkeit des Entscheidungsproblems des Prädikatenkalküls der ersten Stufe mit Identität*, in: Zeitschrift für mathematische Logik und Grundlagen der Mathematik 2 (1956), S. 1–14.

— und dass Identität eliminierter ist, ohne dass die Entscheidbarkeit des prädikatenlogischen Systems zurückgewonnen würde, haben wir bereits in Ab-

schnitt 5.7 gesehen. Spätestens seit dem Zeitpunkt der Veröffentlichung dieses Textes ist also Skepsis bezüglich der Unentscheidbarkeit der Prädikatenlogik zumindest aus dem einen Grund, dass Unentscheidbarkeitsbeweise stets Diagonalbeweise seien, nicht mehr angebracht.

Abbildungsverzeichnis

Tabellenverzeichnis

Literaturverzeichnis

[Ber16] Bernhardt, Chris: *Turing's Vision. The Birth of Computer Science*, Cambridge (MA) u.a. 2016.

[BJ87] Boolos, George S. u. Jeffrey, Richard C.: *Computability and Logic*, Cambridge u.a., 2. Aufl., Nachdr. 1987.

[Bos97] Bostock, David: *Intermediate Logic*, Oxford 1997, Nachdr. 2002.

[Can32] Cantor, Georg: *Über eine elementare Frage der Mannigfaltigkeitslehre*, in: ders., *Gesammelte Abhandlungen mathematischen und philosophischen Inhalts mit Erläuterungen und Anmerkungen* [...], hg. von E. Zermelo, Berlin 1932, S. 278–281.

[Chu36a] Church, Alonzo: *An Unsolvable Problem of Elementary Number Theory*, in: American Journal of Mathematics 58 (1936), S. 345–363.

[Chu36b] Church, Alonzo: *A note on the Entscheidungsproblem*, in: The Journal of Symbolic Logic 1 (1936), S. 40f. u. ders.: *Correction to A note on the Entscheidungsproblem*, ebda., S. 101f.

[Göd86] Gödel, Kurt: *Über formal unentscheidbare Sätze der Principia Mathematica und verwandter Systeme I*, in: ders., *Collected Works*, Bd. 1, hg. von S. Feferman u.a., Oxford u.a. 1986, S. 144–194.

[Gum02a] Gumański, Leon: *The Decidability of the First-Order Functional Calculus*, in: Mathesis Universalis NS 11 (2002), online verfügbar unter `http://www.calculemus.org/MathUniversalis/NS/11/leon_gum.doc`, Zugriff am 14.12.2006.

[Gum02b] Gumański, Leon: *Remarks on Cantor's Diagonal Method and Some Related Topics*, in: Mathesis Universalis NS 11 (2002), online verfügbar unter `http://www.calculemus.org/MathUniversalis/NS/11/leon_gum.doc`, Zugriff am 14.12.2006.

[HA28] Hilbert, David u. Ackermann, Wilhelm: *Grundzüge der theoretischen Logik*, [1. Aufl.] Berlin u.a. 1928.

[Her61] Hermes, Hans: *Aufzählbarkeit, Entscheidbarkeit, Berechenbarkeit. Einführung in die Theorie der rekursiven Funktionen*, Berlin u.a. 1961 (Die Grundlehren der mathematischen Wissenschaften in Einzeldarstellungen mit besonderer Berücksichtigung der Anwendungsgebiete 109).

[Jac04] Jacquette, Dale: *Diagonalization in Logic and Mathematics*, in: *Handbook of Philosophical Logic. 2^{nd} Edition*, Bd. 11, hg. von D. M. Gabbay u. F. Guenther, Dordrecht u.a. 2004, S. 55–147.

[Kal36] Kalmár, László: *Zurückführung des Entscheidungsproblems auf den Fall von Formeln mit einer einzigen, binären, Funktionsvariablen*, in: Compositio Mathematica 4 (1937), 137–144.

[Lam61] Lambek, Joachim: *How to program an infinite abacus*, in: Canadian Mathematical Bulletin 4 (1961), S. 295–302, u. ders., *Correction to my paper „How to Program an Infinite Abacus“*, in: Canadian Mathematical Bulletin 5 (1962), S. 297.

[Lei18] Leitgeb, Hannes: *Logik I. Eine Einführung in die klassische Aussagen- und Prädikatenlogik*, Stand: 22.10.2018, online verfügbar unter `https://www.mcmp.philosophie.uni-muenchen.de/people/faculty/hannes_leitgeb/logik_1_wise20182019/logikskript.pdf`, Zugriff am 30.10.2018.

[Nol97] Nolt, John: *Logics*, Belmont (CA) u.a. 1997.

[Pét67] Péter, Rózsa: *Recursive Functions*, New York u.a., 3. Aufl. 1967.

[Qui70] Quine, Willard Van Orman: *Philosophy of Logic* (Foundations of Philosophy Series), Englewood Cliffs (NJ) 1970.

[Sko23] Skolem, Thoralf: *Begründung der elementaren Arithmetik durch die rekurrierende Denkweise ohne Anwendung scheinbarer Veräenderlichen mit unendlichem Ausdehnungsbereich*, Oslo 1923.

[Tho62] Thomson, James F.: *On Some Paradoxes*, in: *Analytical Philosophy. First Series*, hg. von R. J. Butler, Oxford 1962, Nachdr. 1966, S. 104–119.

[Tur36] Turing, Alan M.: *On Computable Numbers, with an Application to the Entscheidungsproblem*, in: Proceedings of the London Mathematical Society s2-42/Nr. 1, S. 230–265, u. ders.: *On Computable Numbers, with an Application to the Entscheidungsproblem: A correction*, in: Proceedings of the London Mathematical Society s2-43/Nr. 6, S. 544–546.