Der Satz von Gödel [1]

Marc Oliver Hoge

27. Juli 2004

[1]Referat an der Sommerakademie Ftan 2004

Inhaltsverzeichnis

Kapitel 1

Einleitung

Die Entwicklung der Mathematik geht seit jeher in eine Richtung immer größerer Exaktheit und formaler Strenge. Dies führte zur Entwicklung mathematischer Systeme, in denen sich selbst Beweise aus starren, formalen Schritten zusammensetzen lassen und sehr mechanisch wirken. Im Prinzip können alle Beweise der Mathematik in solchen Systemen geführt werden, sodass der Verdacht naheliegt, dass alle mathematischen Wahrheiten auch in einem solchen formalen System bewiesen werden können. Doch hier widerspricht der österreichische Mathematiker Kurt Gödel (1906-1978) mit einer Arbeit aus dem Jahre 1931 energisch, in der er jedem hinreichend starken formalen System die Unvollständigkeit, also die Existenz wahrer, aber nicht beweisbarer Sätze, voraussagt. In diesem Skriptum wird ein formales System herausgegriffen, und zwar das der Zahlentheorie, und der Satz von Gödel für dieses System bewiesen.

Das zweite Kapitel beschäftigt sich mit einem von Cantors Diagonalbeweisen. Der vorgeführte Beweis spielt für den weiteren Text keine Rolle, sondern stellt nur exemplarisch eine Technik vor, die an anderer Stelle von Nutzen sein wird.

Im dritten Kapitel wird dann der Begriff des formalen Systems eingeführt und an Hand des beispielhaften FTAN-Systems erläutert.

Als nächstes führt das vierte Kapitel in das formale System TNT der Zahlentheorie ein. Über einen kleinen Ausflug in die Logik werden die nötigen Schlussregeln aufgestellt und danach die Axiome der Zahlentheorie hinzugenommen. Außerdem taucht das erste Mal das wichtige Konzept der Gödelnummerierung auf, die am Beispiel des FTAN-Systems vorgeführt wird.

Der für Gödels Satz wichtige Begriff der primitiv-rekursiven Eigenschaft wird im fünften Kapitel eingeführt, das sich ansonsten mit abbrechenden und nicht abbrechenden Befehlsschleifen in Computerprogrammen beschäftigt. Darüber hinaus wird gezeigt, dass es allgemein unmöglich ist, das Abbrechen oder Nicht-Abbrechen von potentiell unendlichen Schleifen vorherzusagen.

Schließlich widmet sich das letzte Kapitel ganz dem Beweis des Unvollständigkeitssatzes und zeigt auch, dass die Unvollständigkeit nicht einfach beseitigt werden kann, sondern prinzipieller Natur ist.

Kapitel 2

Cantors Diagonalmethode

Für zwei Beweise in diesem Skriptum (insbesondere für den Beweis von Gödels Satz) wird die Diagonalmethode wichtig sein, die als erster der deutsche Mathematiker Georg Cantor (1845-1918) verwendete. Als eine Vorübung für die kommenden Beweise soll hier der berühmteste Satz Cantors aus der Mengenlehre bewiesen werden. Dazu brauchen wir den Begriff der *Potenzmenge*. Sei A eine Menge. Die Anzahl der Elemente von A bezeichnen wir mit $|A|$. Unter der Potenzmenge $P(A)$ von A verstehen wir die Menge aller Teilmengen von A. Nimmt man zum Beispiel die Menge $B = \{1, 2, 3\}$, so ist die Potenzmenge

$$P(B) = \{ \quad \{\}, \{1\}, \{2\}, \{3\}, \{1, 2\}, \{1, 3\}, \{2, 3\}, \{1, 2, 3\} \quad \}.$$

Die Elemente der Potenzmenge sind wieder Mengen (nämlich Teilmengen von B), insbesondere ist die leere Menge $\emptyset \in P(B)$ und auch $B \in P(B)$. In unserem Beispiel sieht man, dass die Potenzmenge $P(B)$ *mächtiger* als die Menge selbst ist, was bedeutet, dass die Potenzmenge mehr Elemente hat als die ihr zugrundeliegende Menge. Man kann also für dieses Beispiel $|P(B)| > |B|$ schreiben. Cantors Satz sagt nun Folgendes aus:

Satz: *Für jede nichtleere Menge A ist die Potenzmenge mächtiger als die Menge selbst, also $|P(A)| > |A|$.*

Für den Beweis müssen zwei Fälle unterschieden werden. Nehmen wir erst einmal an, A sei eine endliche Menge, das heißt, dass $|A|$ eine endliche Zahl ist. Zu $P(A)$ gehören mindestens alle Teilmengen, die nur ein Element von A enthalten. Das sind genau $|A|$ Teilmengen. Zu $P(A)$ gehört aber auch die leere Menge, sodass $P(A)$ mindestens $(|A| + 1)$ Elemente hat, und damit gilt $|P(A)| > |A|$. Sei nun A eine unendliche Menge. Für diesen Fall führen wir einen Widerspruchsbeweis durch und nehmen an, es gelte $|P(A)| = |A|$. Nummeriert man die Elemente von A mit den Zahlen $1, 2, 3, \ldots$ durch, so lässt sich eine Teilmenge von A als Ziffernfolge von Nullen und Einsen darstellen, zum Beispiel

00111010110 .... Ist das i-te Element von A in der Teilmenge enthalten, so steht an der i-ten Stelle der Ziffernfolge eine 1 und sonst eine 0. Sind beide Mengen gleichmächtig (das sagt unsere Annahme), dann kann man jedem Element $a \in A$ ein-eindeutig (bijektiv) ein Element $p \in P(A)$ zuweisen. p ist eine Teilmenge von A. Man kann sich diese Teilmengen als Ziffernfolgen in einer unendlichen Tabelle eingetragen vorstellen:

Teilmenge, die zum Element 1 gehört:	**0**	0	1	1	1	0	1	0	...
Teilmenge, die zum Element 2 gehört:	0	**1**	1	0	0	0	0	1	...
Teilmenge, die zum Element 3 gehört:	1	0	**1**	0	1	0	1	0	...
Teilmenge, die zum Element 4 gehört:	0	1	0	**1**	1	1	1	1	...
Teilmenge, die zum Element 5 gehört:	1	1	0	1	**0**	0	1	1	...
Teilmenge, die zum Element 6 gehört:	1	0	1	1	1	**0**	1	0	...
Teilmenge, die zum Element 7 gehört:	0	1	0	1	0	0	**1**	0	...
Teilmenge, die zum Element 8 gehört:	1	0	1	0	0	0	0	**0**	...
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\ddots$

Nun kommt die Diagonalmethode: Man nimmt die Elemente der Diagonalen (fett gedruckt) und „dreht sie um“, aus einer Null mache man eine Eins und aus einer Eins eine Null. Die entstandene Null-Eins-Folge (nennen wir sie d) kann man als Teilmenge von A auffassen, aber diese Null-Eins-Folge steht nicht auf der Liste, denn

in der ersten Ziffer stimmt d nicht mit der ersten Zeile überein,
in der zweiten Ziffer stimmt d nicht mit der zweiten Zeile überein,
$\vdots$
in der n-ten Ziffer stimmt d nicht mit der n-ten Zeile überein,
$\vdots$

Es ist also eine Teilmenge gefunden worden, die sich mit keinem Element von A assoziieren lässt. Das widerspricht unserer Annahme, die damit falsch sein muss und es gilt $|P(A)| > |A|$.

Bemerkenswert bei diesem Beweis ist, dass für die Erstellung von d eine Zahl auf zwei Weisen benutzt wurde: Man dreht die n-te Ziffer der n-ten Zeile um. Einmal ist n die Nummer des Elements, mit dem wir die Teilmenge assoziieren (Zeile), und dann die Nummer des Elements in der Menge A (Ziffer). Das ist das Herz der Diagonalmethode und wird uns im Folgenden wieder begegnen.

Kapitel 3

Formale Systeme

3.1 Definition des Begriffs

Unter einem formalen System *versteht man eine Menge von Axiomen, Ableitungsregeln und Sätzen, wobei Sätze entstehen, wenn man von Axiomen ausgehend beliebig oft die Ableitungsregeln anwendet, sofern die Anwendung der gewählten Ableitungsregel in jedem Einzelfall erlaubt ist. Die Anwendung keiner Ableitungsregel ist immer erlaubt.*

Axiome verstehen wir hier als Zeichenketten und Ableitungsregeln als Anweisungen, wie man Zeichenketten manipulieren darf. Aus der Definition folgt, dass Axiome automatisch Sätze sind.
Diese abstrakte Definition eines formalen Systems ist eher undurchsichtig und es wird vieles klarer werden, wenn wir uns einmal ein bestimmtes formales System näher anschauen.

3.2 Das FTAN-System

Bei diesem formalen System werden nur vier Buchstaben des Alphabets benutzt, nämlich **F**, **T**, **A** und **N**. Ketten sehen daher etwa so aus:

FTN
FTAAAAA
ATN

Außerdem gibt es folgende zwei Schlussregeln im **FTAN**-System:

Regel I: Hat man **FT**x, wobei x für eine beliebige Kette steht, kann man **FT**xx hinzufügen.

Regel II: Kommt in einer Kette **NNN** vor, kann man eine neue Kette mit **A** anstelle von **NNN** bilden.

Als Axiom (also erste Kette, mit der man starten kann,) gibt es hier die Kette

FTN. Nun kann man vom Axiom ausgehend die Regeln anwenden und jede Kette, die sich so erzeugen lässt, ist dann ein Satz des FTAN-Systems. Schreibt man diese Manipulationen in eine Tabelle so heißt das *Ableitung*. In der nun folgenden Ableitung wird gezeigt, dass **FTAN** ein Satz ist.

1)	**FTN**	(Axiom)
2)	**FTNN**	(Regel I)
3)	**FTNNNN**	(Regel I)
4)	**FTAN**	(Regel II)

Entscheidend bei einer Ableitung ist, dass jeder Schritt samt angewendeter Regel aufgeschrieben wird, ohne Teile wegzulassen oder gar Regeln zu verletzen. Als nächste Frage könnte man stellen, ob zum Beispiel **FTA** ein Satz ist. Dies ist nicht sofort zu sehen, da Regel I die Ketten verlängert, Regel II hingegend die Ketten verkürzt. Es könnte sehr viele Möglichkeiten geben, auf **FTA** zu kommen. Man könnte einen Computer programmieren, systematisch die Regeln anzuwenden, und hoffen, dass irgendwann die Kette **FTA** auftaucht. Unter Umständen dauert die Ausführung des Programms sehr lange, vielleicht kommt sie sogar nie zu einem Ende.

Am Beispiel des FTAN-Systems sieht man recht gut, was ein formales System ausmacht: Man kann völlig mechanisch Regeln befolgen, ohne über das System hinauszuschauen (dazu sogar einen Computer verwenden), aber es gibt auch Probleme (wie zum Beispiel die Erzeugbarkeit von **FTA**), die einen Blick – oder besser – einen Gedanken über das FTAN-System hinaus erfordern.

Auf der anderen Seite ist das FTAN-System auch ein schlechtes Beispiel, denn die vorkommenden Ketten und Sätze können nichts aussagen; die Satzheit von **FTAN** ist nicht mehr als ein Wortspiel mit einem gewissen Fleckchen in den Schweizer Alpen. Interessant werden formale Systeme, wenn man die Sätze des Systems zum Beispiel als Aussagesätze interpretieren kann und darüber hinaus jeder Satz des Systems eine Wahrheit und jeder Nicht-Satz eine Unwahrheit darstellt. So kann dann aus dem System heraus (durch Befolgung von starren Regeln) beurteilt werden, ob eine „im System darstellbare“ Aussage wahr oder falsch ist. Ein Beispiel für ein solches System stellt TNT dar, dem das nächste Kapitel gewidmet ist.

Kapitel 4

Theoria Numerorum Typographica

Das Ziel in diesem Kapitel ist es, die Zahlentheorie (also den Zweig der Mathematik, der sich mit den natürlichen Zahlen beschäftigt,) in ein formales System zu übersetzen, sodass zahlentheoretische Beweise als Ableitungen in diesem System darstellbar sind. Dieses formale System nennen wir Theoria Numerorum Typographica (lat., typographische Zahlentheorie, kurz: TNT).
Da mathematische Beweise logische Schlüsse sind, muss TNT Grundzüge der Logik enthalten, nur dann kann TNT ein geeigneter Kandidat für unser Vorhaben werden. Im nächsten Abschnitt wird daher ein formales System konstruiert, das zunächst die Aussagenlogik einfängt. Schrittweise wird dieses System mit Axiomen und Regeln dann so erweitert, dass es zu einem formalen System wird, in dem zahlentheoretische Beweisführungen möglich sind.

4.1 Aussagenlogik

Die Aussagenlogik befasst sich mit logischen Schlüssen wie diesem hier:

Wenn es regnet, ist die Straße nass.
Es regnet.
Also ist die Straße nass.

Bezeichnet man den Satz „Es regnet.“ mit P und „Die Straße ist nass.“ mit Q, dann lässt sich der obige Schluss durch

$$P \to Q \quad , \quad P \quad , \quad \text{also } Q$$

formalisieren. Das Zeichen $\to$ heißt dabei so etwas wie „Wenn ..., dann ...“ oder „impliziert“. Man braucht außerdem das „oder“, für das wir $\vee$ schreiben, und das „und“, was durch $\wedge$ repräsentiert wird. Die Kette

$$(P \wedge Q) \to (P \vee Q)$$

bedeutet dann etwa: „Die Tatsache, dass P und Q beide wahr sind, impliziert, dass P oder Q wahr ist."
Zur Sicherheit betone ich noch explizit, dass sämtliche Großbuchstaben in der Aussagenlogik stets Platzhalter für einen beliebigen (deutschen) Satz sind. Nun wissen wir also, wie Ketten der Aussagenlogik aussehen können – was fehlt sind nun die Regeln und Axiome. Zu jedem Zeichen ($\wedge$, $\vee$, $\rightarrow$) gibt es jeweils eine Einführungs- und Beseitigungsregel, die intuitiv einleuchten sollte, da für diese Regeln keine weiteren Begründungen angeführt werden können. In der Definition der Regeln werden griechische Buchstaben benutzt. Damit soll angedeutet werden, dass für ein Φ beispielsweise nicht nur eine einzelne (deutsche) Aussage P eingesetzt werden kann, sondern auch komplexere aussagenlogische Konstruktionen wie $(P \wedge Q) \rightarrow (P \vee Q)$.

$\wedge$-Einführung: *Hat man die Ketten Φ und Ψ zur Verfügung, kann man zu $(\Phi \wedge \Psi)$ übergehen.*

$\wedge$-Beseitigung: *Hat man die Kette $(\Phi \wedge \Psi)$ zur Verfügung, kann man zu Φ oder auch Ψ übergehen.*

$\vee$-Einführung: *Hat man die Kette Φ, kann man zu $(\Phi \vee \Psi)$ übergehen, wobei Ψ eine beliebige Kette ist.*

$\vee$-Beseitigung: *Hat man die Ketten $(\Phi \vee \Psi)$ und $(\Phi \rightarrow \Theta)$ und $(\Psi \rightarrow \Theta)$ zur Verfügung, kann man zu Θ übergehen.*

$\rightarrow$-Einführung: *Hat man unter der Voraussetzung Φ die Kette Ψ hergeleitet, kann man zu $(\Phi \rightarrow \Psi)$ übergehen.*

$\rightarrow$-Beseitigung: *Hat man die Ketten $(\Phi \rightarrow \Psi)$ und Φ zur Verfügung, kann man zu Ψ übergehen.*

Über ein Zeichen habe ich bisher noch nichts gesagt, nämlich über die Repräsentation des umgangssprachlichen „nicht", das man als $\neg$ schreibt. Die Regeln dazu lauten:

$\neg$-Einführung: *Hat man unter der Voraussetzung Φ einen Widerspruch (also etwas wie $(\Theta \wedge (\neg\Theta))$) hergeleitet, kann man zu $\neg(\Phi)$ übergehen.*

$\neg$-Beseitigung: *Hat man die Kette $\neg(\neg(\Phi))$ zur Verfügung, kann man zu Φ übergehen.*

Ich werde auf diese Regeln nicht besonders eingehen, da dies nicht weiter erhellend und auch nicht unbedingt nötig für unser Vorhaben ist. Stattdessen schauen wir uns eine Ableitung für $(P \wedge Q) \rightarrow (P \vee (R \wedge Q))$ an.

1)	$(P \wedge Q)$	Voraussetzung
2)	P	$\wedge$ Beseitigung
3)	$(P \vee (R \wedge Q))$	$\vee$ Einführung
4)	$(P \wedge Q) \rightarrow (P \vee (R \wedge Q))$	$\rightarrow$ Einführung

Es stellt kein Problem dar, dass in dieser Ableitung kein Axiom benutzt wurde – ganz im Gegenteil! Das hier verwendete formale System der Aussagenlogik ist axiomenfrei, sodass es gar keine Axiome gibt, die hätten verwendet werden können.

Im System der Aussagenlogik kann jedoch noch keine zahlentheoretische Aussage dargestellt werden wie beispielsweise „5 ist eine Primzahl“. Um das zu leisten, müssen wir zuerst Zahlzeichen, Variable und Quantoren einführen.

4.2 Zahlzeichen

Um eine Zahl wie Fünf auszudrücken, könnte man sich naiv einfach dem Zeichen 5 bedienen. Dies wird sich allerdings später als unpraktisch herausstellen, wenn wir die Gödelnummerierung einführen, daher schreiben wir für die Null das Zahlzeichen 0, für die Zahl Eins aber S0 (S für Sukzessor, Nachfolger) und entsprechend die Fünf SSSSS0. Diese auf den ersten Blick merkwürdig anmutenden S-Kolonnen mit einer 0 am Ende heißen *Zahlzeichen*.
Um mathematische Aussagen zu formulieren, brauchen wir außerdem das Gleichheitszeichen =, das Pluszeichen +, das Minuszeichen −, das Produktzeichen · und das Quotientenzeichen : , wobei alle diese Zeichen ihre gewohnte Bedeutung haben sollen. Als nächstes werden wir die P's und Q's in der Aussagenlogik durch mathematische Aussagen wie

$$\mathrm{SS0} + \mathrm{SSS0} = \mathrm{SSSSS0}$$
oder
$$\mathrm{SSS0} \cdot \mathrm{SSSS0} = \mathrm{SSSSSSSSSSSS0}$$

ersetzen. Man erhält dann Ketten wie

$$(\mathrm{SS0} + \mathrm{SSS0} = \mathrm{SSSSS0} \rightarrow \mathrm{S0} = \mathrm{S0}) \quad ,$$

was bedeutet: *Wenn* 2 *und* 3 *zusammen* 5 *ist, dann ist* 1 *gleich* 1.

Für das neue Zeichen S gibt es auch jeweils eine Einführungs- und Beseitigungsregel. Seien r und t Zahlzeichen, dann gilt:

S-Einführung: *Ist* r=t *ein Satz von TNT, dann auch* Sr=St.

S-Beseitigung: *Ist* Sr=St *ein Satz von TNT, dann auch* r=t.

Noch immer lässt sich aber nicht ausdrücken, dass 5 eine Primzahl ist. Es ist also eine zusätzliche Erweiterung notwendig.

4.3 Quantorenlogik und TNT-Axiome

Was wir noch brauchen sind Ausdrücke für „Es existiert mindestens ein ...“ und „Für alle ...“. Symbole hierfür sind die Quantoren (der Existenzquantor ∃

und der Allquantor ∀). Der Satz „Es gibt eine Zahl a, die zu 3 addiert 5 ergibt“ übersetzt sich folgendermaßen:

$$\exists a{:}a{+}SSS0{=}SSSSS0$$

Das dazu etwas merkwürdig klingende, falsche Gegenstück „Alle Zahlen ergeben 5, wenn man sie zu 3 addiert“ schreibt sich ganz analog

$$\forall a{:}a{+}SSS0{=}SSSSS0 \quad .$$

Der Ausdruck

$$a{+}SSS0{=}SSSSS0$$

für sich alleine jedoch macht keinen Sinn. Man sagt, die Variable a ist in der Formel *frei* oder *nicht quantifiziert*. Interpretiert sagt diese Kette a+3=5, was weder wahr noch falsch ist, weil keine Aussage über a gemacht wurde.

Nun haben wir zwei neue Zeichen (∃ und ∀) eingeführt, und natürlich gibt es auch für diese beiden Einführungs- und Beseitigungsregeln, die ich hier aber nicht angeben will. Die Regeln sind nicht weiter schwierig, nur mühselig aufzuschreiben und wiederum wenig erhellend. Erwähnen möchte ich allerdings noch zwei Regeln, die das Gleichheitszeichen betreffen. Seien dafür r, s und t beliebige Zahlzeichen.

Gleichheit: *Ist r=s ein Satz, dann ist auch s=r ein Satz.*

Transitivität: *Sind r=s und s=t Sätze, dann ist auch r=t ein Satz.*

Nun haben wir alle Regeln beieinander. Um einen Zusammenhang zwischen dem formalen System TNT, das bis hierher geschaffen wurde, und der Zahlentheorie herzustellen, müssen noch die folgenden fünf Axiome hinzugefügt werden:

Axiom 1: ∀a:¬Sa=0
Axiom 2: ∀a:(a+0)=a
Axiom 3: ∀a:∀a':(a+Sa')=S(a+a')
Axiom 4: ∀a:a·0=0
Axiom 5: ∀a:∀a':(a·Sa')=((a·a')+a)

Mit diesen Regeln und Axiomen ist TNT nun schon sehr schlagfertig. Die Tatsache, dass 5 eine Primzahl ist, kann man jetzt in TNT-Formeln aufschreiben:

$$(*) \qquad \neg\exists a{:}\exists a'{:}(SSa{\cdot}SSa'){=}SSSSS0$$

Eine „wörtliche“ Übersetzung wäre: Es gibt keine Zahlen a und a', sodass das Produkt von (a+2) und (a'+2) gleich 5 ist. Nach kurzem Nachdenken wird man merken, dass dies nichts anderes bedeutet, als dass 5 prim ist.
Man beachte, dass ich für eine zweite Variable a' nicht etwa b geschrieben habe. Das

hätte ich tun können, aber auch hier wird sich später herausstellen, dass man besser damit fährt, nur Variablen a, a', a", a"', ... zu benutzen.
Die Frage ist natürlich nun, ob (∗) auch ein Satz von TNT ist (also ob sich (∗) aus den Axiomen von TNT mit den Ableitungsregeln von TNT ableiten lässt) und ob Ketten wie

¬∃a:∃a':(SSa·SSa')=SSSS0

sich als Nicht-Sätze herausstellen. Dies ist in der Tat der Fall und darüber hinaus kann TNT eine sehr große Klasse von zahlentheoretischen Aussagen nicht nur *darstellen*, d.h. man kann eine TNT-Formel aufstellen, die die gewünschte Aussage beinhaltet, sondern TNT kann diese Klasse von Aussagen sogar *repräsentieren*, d.h. die entsprechenden TNT-Formeln sind genau dann Sätze von TNT, wenn die dazugehörenden Aussagen wahr sind. (Mehr dazu im nächsten Kapitel.)
Ein Sachverhalt sollte aber noch besondere Aufmerksamkeit verdienen: Die Schlussregeln von TNT sind sehr stark an der Weise orientiert, in der wir täglich logische Schlüsse ziehen. Beschäftigt man sich eine kurze Zeit mit den Regeln (und auch den Axiomen), so sind sie einfach einleuchtend und es ist unmittelbar klar, dass jede Aussage, die unter Anwendung der Schlussregeln aus den Axiomen heraus gewonnen werden kann, auch wahr sein muss. Folgenden Sachverhalt sollten wir daher explizit festhalten:

Jeder Satz von TNT stellt eine Wahrheit dar.

Die Umkehrung („Jede Wahrheit ist als Satz in TNT vorhanden.“) wird von Gödels Satz später quasi zerschmettert werden, aber an der Wahrheit jedes TNT-Satzes lässt sich nicht zweifeln.

4.4 Gödelnummerierung des FTAN-Systems

Wir werden jetzt zum ersten Mal eine Gödelnummerierung durchführen und kehren dazu zu unserem alten Beispiel, dem FTAN-System, zurück. Die Idee der Gödelnummern ist neben Cantors Diagonalmethode der entscheidende Schritt zum Beweis von Gödels Unvollständigkeitssatz. Mit der Nummerierung kann man nun zeigen, dass TNT in gewisser Weise das FTAN-System enthält.

Sätze des FTAN-Systems bestehen aus den Buchstaben **F**, **T**, **A** und **N**. Diese Buchstaben erhalten nun die folgenden Nummern:

F	⇔	9
T	⇔	8
A	⇔	0
N	⇔	1

Mit der Kette **FTAN** assoziiert man dann die Zahl 9801 oder anders ausgedrückt: Mit dieser Übersetzung sind die Zahl 9801 und die Kette **FTAN** gleichbedeutend. Die geniale Idee von Gödel war nun, auch die Schlussregeln zu arithmetisieren, also zu Rechenanweisungen für natürliche Zahlen zu machen. Zur Erinnerung noch einmal die Regeln des FTAN-Systems:

Regel I: Hat man **FT**x, wobei x für eine beliebige Kette steht, kann man **FT**xx hinzufügen.

Regel II: Kommt in einer Kette **NNN** vor, kann man eine neue Kette mit **A** anstelle von **NNN** bilden.

Übersetzt man nun die FTAN-Ketten in natürliche Zahlen und operiert mit diesen, schreiben sich diese Regeln äquivalent als

Regel I: Sei $r > 0$ und $n < 10^r$. Hat man eine Zahl, die sich als $98 \cdot 10^r + n$ schreiben lässt, dann kann man die Zahl $98 \cdot 10^{2r} + n \cdot 10^r + n$ hinzufügen.

Regel II: Sei $r > 3$, $n < 10^{r-3}$ und m beliebig. Hat man eine Zahl, die sich als $m \cdot 10^r + 111 \cdot 10^{r-3} + n$ schreiben lässt, dann kann man die Zahl $m \cdot 10^{r-2} + 0 \cdot 10^{r-3} + n$ hinzufügen.

Im letzten Kapitel hatten wir die Kette **FTAN** aus dem Axiom **FTN** abgeleitet. Jetzt können wir analog auch die Zahl 9801 aus dem Axiom 981 ableiten:

1)	981	Axiom
2)	9811	Regel I mit $r = 1$ und $n = 1$
3)	981111	Regel I mit $r = 2$ und $n = 11$
4)	9801	Regel II mit $r = 4$, $n = 1$ und $m = 98$

Was haben wir nun gewonnen? Wir haben ein typographisches System, ein System, in dem Zeichen durch gewisse Regeln manipuliert werden, in ein arithmetisches System verwandelt, in dem Rechenoperationen auf natürlichen Zahlen entsprechende (isomorphe) Veränderungen durchführen. Dies ist ein gewaltiger Schritt, denn wenn wir zurück zu der Frage kommen, ob **FTA** ein Satz des FTAN-Systems ist, können wir jetzt gleichbedeutend fragen, ob 980 eine Satz-Zahl des „9801-Systems“ ist. Dies ist ein Problem aus der Zahlentheorie, in dem es um gewisse Eigenschaften der natürlichen Zahl 980 geht, und wir dürfen davon ausgehen, dass es eine TNT-Formel gibt, die eben diese Eigenschaft von 980 ausdrückt. Diese Kette ist sehr, sehr lang und wir begnügen uns damit, dass man diese Kette finden *kann*, und kürzen sie mit

FTAN-ZAHL[SSS...SSS0]

ab (es sind 980 S's). Wenn dies ein Satz von TNT ist, dann ist auch **FTA** ein Satz des FTAN-Systems. Es ist also gezeigt, dass man alle Überlegungen über das FTAN-System auch in TNT durchführen kann. Schaut man sich typographische Systeme etwas allgemeiner an, so kann man sogar zeigen, dass sich

Manipulationen von Buchstabenketten immer mit einer Gödelnummerierung und Addition, Multiplikation und Zehnerpotenzen darstellen lassen, sodass *jedes formale System in TNT hineintransportiert werden kann.*
Nun ist es auch möglich, einen kleinen Vorgeschmack auf Gödels Satz zu geben: Man kann jedes formale System mit Gödelnummern in TNT einbetten – der nächste Schritt ist nun, TNT selbst zu „gödelisieren“, sodass man innerhalb einer TNT-Kette Aussagen über *andere* TNT-Ketten machen kann. Dies wird uns erlauben, eine Kette zu konstruieren, die über sich sagt, sie sei selbst kein Satz von TNT. Wie wir auch noch sehen werden, ist es die Existenz dieser Kette, die die Unvollständigkeit von TNT besiegelt.
Wir können aber noch nicht sofort auf Gödels Satz kommen, da wir für den Beweis noch die Begriffe der primitiv-rekursiven, allgemein-rekursiven und partiellen Funktion brauchen. Dazu machen wir im nächsten Kapitel einen Ausflug in die Welt der Computerprogrammierung.

Kapitel 5

Berechenbare Funktionen

In diesem Kapitel werden wir uns damit beschäftigen, wie ein Computer mit natürlichen Zahlen operieren kann und welche Funktionen maschinell berechnet werden können und welche eventuell nicht. Unter einer *Funktion* verstehen wir im Folgenden einfach ein Computerprogramm, das mit einem Inputparameter (einer natürlichen Zahl) einen Output (ebenfalls eine natürliche Zahl) berechnet und diesen am Ende der Prozedur ausgibt, falls die Prozedur zu einem Ende kommt.
Entscheidend ist, dass Computer nur in wohldefinierten Schritten rechnen können, wobei ein Rechenschritt einer der folgenden vier Operationen ist:

- Addition zweier Zahlen,
- Multiplikation zweier Zahlen,
- Prüfen, ob zwei Zahlen gleich sind, und
- Ermitteln der größeren zweier Zahlen.

Neben den Funktionen sind auch Programme wichtig, die testen, ob die Input-Zahl eine gewisse Eigenschaft hat oder nicht (zum Beispiel, ob es sich um eine Primzahl handelt). Ein solches Programm heißt *Test* und gibt als Output ein *Ja* oder *Nein* aus. Interessant ist nun die Frage, wieviele Rechenschritte zur Berechnung einer bestimmten Funktion oder Durchführung eines bestimmten Tests nötig sind.

5.1 BlooP-Programme

Es gibt grundsätzlich zwei Sorten von Schleifen in einer Programmiersprache: solche, die nach einer bestimmten Anzahl von Durchläufen abbrechen (nennen wir sie *BlooPs* für bounded loop), und solche, die erst abbrechen, wenn eine bestimmte Bedingung erfüllt ist, und potentiell nie abrechen, wenn eben diese

Bedingung nie erfüllt ist. Die letzteren Schleifen heißen dann *FlooPs* für free loop. Es ist noch wichtig zu betonen, dass für BlooPs die Anzahl der Durchläufe erst dann feststehen muss, wenn der erste Durchgang beginnt. Bevor etwa der Input bekannt ist, braucht auch die Durchlaufsanzahl noch nicht festzustehen. Es soll nun gezeigt werden, dass man den Test, ob eine Zahl eine Primzahl ist, so programmieren kann, dass er nur BlooP-Schleifen enthält. Dafür müssen wir ein wenig Programmcode aufschreiben und ich benutze dabei eine intuitive Programmiersprache (ich nenne sie EasyRead), die man LeichtLesen kann und die nicht vieler Erklärungen bedarf. Zuerst braucht man ein Programm, dem man zwei Input-Zahlen `N` und `M` eingibt und darauf als Output den Rest der Division von `N` durch `M` erhält. Der Code sieht etwa so aus:

```
PROZEDEDUR-DEFINITION REST[N,M]
BLOCK 0: ANFANG
         A = 1
         BlooP-SCHLEIFE HÖCHSTENS N MAL
         BLOCK 1: ANFANG
                  WENN (N - (M · A)) < M DANN BEENDE BlooP-SCHLEIFE
                  A = A + 1
         BLOCK 1: ENDE
         OUTPUT = N - (M · A)
BLOCK 0: ENDE
```

Diese Prozedur `REST[N,M]` gibt den Rest der Division von `N` durch `M` aus und benutzt dabei nur eine BlooP-Schleife. Jetzt können wir auch den Primzahlentest aufschreiben:

```
PROZEDEDUR-DEFINITION PRIM?[N]
BLOCK 0: ANFANG
         WENN N < 2 DANN BEENDE BLOCK 0
         B = 2
         BlooP-SCHLEIFE HÖCHSTENS (N-2) MAL
         BLOCK 1: ANFANG
                  WENN REST[N,B] = 0 DANN BEENDE BLOCK 0
                  B = B + 1
         BLOCK 1: ENDE
         OUTPUT = Ja
BLOCK 0: ENDE
```

Nun haben wir einen Primzahl-Test, der in voraussagbar vielen Schritten ermittelt, ob `N` eine Primzahl ist. Innerhalb der BlooP-Schleife von `PRIM?` wird die Prozedur `REST` aufgerufen, die selbst auch eine BlooP-Schleife enthält. Das ändert aber nichts an der Tatsache, dass es eine obere Schranke für die Anzahl der überhaupt durchlaufenen Schleifen gibt – hier ist es das Produkt aus der

maximalen Durchlaufszahl beider Schleifen. Ein solches Programm wie `PRIM?`, das nur BlooP-Schleifen enthält, nennt man *primitiv-rekursiv*. Entsprechend heißt auch die Primitivität einer Zahl eine *primitiv-rekursive Eigenschaft*, weil man ein Programm schreiben kann, das eine Zahl auf diese Eigenschaft prüft und dabei nur BlooPs verwendet.
Ein berühmtes ungelöstes Problem in der Zahlentheorie ist die Goldbach-Vermutung, nach der sich jede gerade Zahl, die größer ist als 4, als Summe zweier ungerader Primzahlen schreiben lässt. Beispiele:

$$\begin{aligned} 6 &= 3+3 \\ 8 &= 3+5 \\ 10 &= 3+7=5+5 \\ 12 &= 5+7 \\ &\vdots \end{aligned}$$

Sprechen wir einer Zahl n die Goldbach-Eigenschaft zu und meinen damit, dass es zwei ungerade Primzahlen gibt, deren Summe n ergibt, dann ist diese Eigenschaft ebenfalls primitiv-rekursiv. Es gibt nur endlich viele Primzahlen, die kleiner sind als n, und man kann in voraussagbar vielen Schritten alle Summen von je zwei dieser Primzahlen ausrechnen und prüfen, ob n eine dieser Summen ist. Das Programm dazu schreibe ich hier nicht auf.

5.2 FlooP-Programme

Verändert man das Goldbach-Problem ein bisschen und nennt es vielleicht „Silberbach“-Problem, indem man sich fragt, ob es gerade Zahlen gibt, die sich als *Differenz* von zwei ungeraden Primzahlen schreiben lassen, dann stellt man fest, dass die Silberbach-Eigenschaft der Zahl n nicht mehr primitiv-rekursiv ist. Man hat hier unendlich viele Primzahlen, deren Differenzen man berechnen und mit n vergleichen muss. Dies ist mit BlooP-Schleifen nicht mehr zu leisten.

Die Programme, die mindestens eine FlooP-Schleife enthalten, laufen Gefahr, nie zu einem Ende zu kommen. Die „guten“ Vertreter unter ihnen, also die Funktionen, die schließlich für jeden Input doch zur Ruhe kommende FlooP-Schleifen enthalten, heißen *allgemein-rekursive Funktionen*. Sollte das Programm jedoch nie zu einem Ende kommen und somit auch keinen Output liefern, heißt die Funktion *partiell*. Ein Beispiel für eine allgemein-rekursive Funktion wäre ein Programm, dem man eine natürliche Zahl n eingibt und das als Output die kleinste Primzahl liefert, die größer ist als n.

5.3 Kurz zu TNT

An dieser Stelle kann man einen Moment innehalten und sich noch einmal der Frage zuwenden, wie „gut“ TNT nun eigentlich ist. Es wurde schon erwähnt, dass TNT eine große Klasse zahlentheoretischer Aussagen darstellen kann, wobei *darstellen* nur hieß, dass man eine TNT-Kette aufschreiben kann, die als diese Aussage zu interpretieren ist. In der Tat ist es so, dass TNT praktisch alle zahlentheoretischen Aussagen darstellen kann – besonders faszinierend ist dies jedoch nicht, denn damit ist nichts darüber gesagt, ob TNT *vollständig* ist, also alle Aussagen auch repräsentiert, d.h. TNT-Satzheit gleichzeitig die Wahrheit der zugehörigen Aussage und Nicht-Satzheit die Unwahrheit der Aussage bedeutet.
Wie der Beweis von Gödels Satz zeigen wird, kann TNT Vollständigkeit nie erreichen, aber immerhin gilt Folgendes: **TNT kann alle primitiv-rekursiven Eigenschaften repräsentieren**. In gewissem Sinne sind das die „harmlosesten“, da ihre Tests (und TNT-Ableitungen) endlich bleiben. Wie wir noch sehen werden, ist die Tatsache der Repräsentierbarkeit aller primitiv-rekursiven Prädikate gerade eine der Voraussetzungen für Gödels Beweis. Jedes formale System, das primitiv-rekursive Prädikate repräsentieren kann, ist nach Gödel unvollständig – und jedes formale System, das es nicht kann, sowieso.
TNT kann sogar allgemein-rekursive Prädikate repräsentieren, das ist jedoch nicht weiter wichtig, denn könnte TNT das nicht, wäre es eben deshalb unvollständig. Gödels Beweis zeigt aber sogar, dass TNT (und jedes andere ebenbürtige System) *prinzipiell* unvollständig ist.

5.4 Der Halte-Test

Nun entfernen wir uns für einen Moment von Gödels Satz und beweisen einen interessanten Satz aus der theoretischen Informatik, der sich mit BlooPs und FlooPs beschäftigt.
Eine wichtige Frage in diesem Zusammenhang ist sicherlich, ob man einem Programm, das FlooP-Schleifen enthält, irgendwie ansehen kann, ob es für jeden Input endet oder nicht. Ein solcher Test muss primitiv-rekursiv sein, denn sonst wäre alle Bemühung nutzlos, da nicht klar wäre, ob der Test selbst jemals endet. Wir suchen also ein mit BlooP-Schleifen arbeitendes Programm, das ein FlooP-Programm als Input erhält und bestimmen kann, ob das Input-Programm immer abbricht.
Das Problem, ein ganzes Programm in ein anderes als Input einzugeben, lässt sich mit einer Gödelnummerierung leicht erledigen: In EasyRead gibt es folgende Zeichen:

```
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
1 2 3 4 5 6 7 8 9 0 ( ) [ ] + - · : = < > ? .
```

Diese Zeichen kann man mit 901, 902, 903, ... durchnummerieren, sodass man statt dem Programmcode auch eine sehr, sehr große natürlich Zahl schreiben kann. Diese Gödelnummer des zu testenden Programms bekommt dann der Halte-Test als Input und auf diese Weise kann der Halte-Test ebenfalls in EasyRead programmiert werden.
Im Folgenden wird gezeigt, dass ein solcher primitiv-rekursiver Halte-Test nicht existiert. Das ist fast beruhigend, denn die Vorstellung eines solchen Tests ist mehr als gewöhnungsbedürftig. Man denke an die Möglichkeiten: Wir hatten über die Silberbach-Eigenschaft gesprochen und das FlooP-Programm, das für eine gerade Zahl n feststellt, ob sie sich als Differenz zweier ungerader Primzahlen schreiben lässt. Hätten wir einen Halte-Test, könnten wir ihn auf dieses Programm anwenden, und falls die Antwort `Ja` wäre, hieße das, dass das Silberbach-Programm für jedes n endet. Es wäre damit bewiesen, dass alle Zahlen die Silberbach-Eigenschaft haben.

5.5 EasyRead und die Church'sche These

Bevor wir die Unmöglichkeit des Halte-Tests zeigen, müssen noch zwei Bemerkungen gemacht werden. Zum einen haben wir uns inzwischen an EasyRead mit seinen FlooPs und BlooPs gewöhnt – könnte nicht eine andere Programmiersprache für das Halte-Problem geeigneter sein?
Die Antwort lautet schlicht: Nein. Man kann mathematisch exakt zeigen, dass eine Programmiersprache, die FlooPs und BlooPs umfasst, äquivalent zu allen im Moment gängigen Sprachen ist und darüber hinaus keine leistungsfähigere Programmiersprache existiert. Mit anderen Worten: EasyRead ist alles, was wir brauchen.

Zum anderen wartet aber eine Frage auf uns, die es etwas mehr in sich hat: Kann ein Mensch irgendetwas ausrechnen, das ein idelaer Computer nicht berechnen kann? Wieder antworten wir: Nein. Dieses „Nein“ ist die Church'sche These und leider kann man hier keinen mathematischen Beweis erwarten, sondern nur ein paar Überlegungen dazu. Was wir Menschen ausrechnen, berechnen wir mit Hilfe der vier Grundrechenarten und vielleicht (Zehner-) Potenzen – diese Mittel stehen dem Computer ebenfalls zur Verfügung. Wenn wir komplizierte Berechnungen durchführen, tun wir das in Schritten. Wir können die Church-These herunterkochen auf die Aussage: Die Schritte, die wir im Gehirn zur Berechnung komplizierter Aufgaben durchlaufen, sind auf einer gewissen Ebene isomorph zu den Schritten, die ein Computerprogramm zur Lösung desselben Problems abarbeitet. Akzeptiert man diesen Gedanken, dann muss man wohl auch der Church-These zustimmen, die man zusammen mit der obigen

Argumentation auch so wiedergeben kann:

> Es gibt nichts, was ein Mensch ausrechnen kann, ein EasyRead-Programm aber nicht.

5.6 Beweis: Halte-Test existiert nicht

Diese Tatsache soll mit einem Widerspruchsbeweis gezeigt werden. Wir nehmen also erst einmal an, es gäbe einen primitiv-rekursiven Halte-Test, und führen diese Aussage auf einen Widerspruch. Los geht's:

Zunächst einmal stellen wir uns vor, wir hätten den Vorrat aller EasyRead-Programme vor uns. Nun führen wir vier Filteroperationen aus:

1. Entferne alle Programme, die mehr als einen Input-Parameter haben.
2. Entferne alle Programme, die keinen Input-Parameter haben.
3. Entferne alle Programme, die nicht eine natürliche Zahl als einzigen Output erzeugen.
4. Entferne alle nicht endenden Programme.

Es ist klar, dass wir die vierte Filteroperation nur auf Grund der Annahme machen können, dass wir einen Halte-Test haben.
Zurück bleiben also alle Programme, die Funktionen eines Input-Parameters sind und nach endlicher Zeit eine Output-Zahl ausgeben. Diese Programme nummerieren wir jetzt durch, indem wir sie erst der Länge (Anzahl der im Code verwendeten Zeichen) nach sortieren und dann Programme gleicher Länge alphabetisch anordnen. Es reicht, wenn man einsieht, dass es prinzipiell möglich ist, alle Programme mit einer Indexnummer zu versehen; man braucht die Liste niemals aufzuschreiben. Der Output des `M`-ten Programms zum Input `N` wird mit

```
Programm{Nr. M}[N]
```

bezeichnet. Jetzt sind wir an der ersten Stelle in diesem Text, an der Cantors Diagonalmethode eingesetzt wird. Wir definieren jetzt die Funktion `DIAG` durch

```
DIAG[N] = 1 + Programm{Nr. N}[N]   .
```

Diese Gleichung soll bedeuten, dass ein Mensch, der den Wert von `DIAG` zum Input `N` berechnen soll, zuerst das Programm mit der Indexnummer `N` und dem Input `N` starten lässt und anschließend zum Output 1 dazuaddiert. Wie bei der ersten Begegnung mit Cantor wird bei `DIAG` die Zahl `N` auf zwei verschiedene Weisen benutzt: als Nummer das Programms und als Input für das Programm. Nun aber das Problem: Das Programm `DIAG` hat keine Indexnummer! Nehmen wir an, es hätte die Indexnummer `X`, dann wäre

```
DIAG[N] = Programm{Nr. X}[N]
```

für alle natürlichen Zahlen `N`. Berechnet man nun `DIAG[X]`, so ist nach der Definition von `DIAG`

```
DIAG[X] = 1 + Programm{Nr. X}[X]   ,
```

da `DIAG` aber die Indexnummer `X` hat, ist auch

```
DIAG[X] = Programm{Nr. X}[X]   .
```

Zusammenfassend gilt also dann

```
Programm{Nr. X}[X] = 1 + Programm{Nr. X}[X]   ,
```

also ein offensichtlicher Widerspruch. Die Annahme, `DIAG` hätte eine Indexnummer, ist also falsch. Die Tatsache, dass `DIAG` keine Indexnummer hat, obwohl es eine Funktion einer natürlichen Zahl ist, die ein Mensch berechnen kann (Anleitung siehe oben), heißt, dass man `DIAG` in EasyRead nicht programmieren kann. Dies ist aber ein Widerspruch zur Church-These aus dem letzten Abschnitt und damit müssen wir eine weitere Annahme revidieren. Die einzige noch im Raum stehende Annahme ist jedoch die Existenz des Halte-Tests, sodass es den Schluss erzwingt:

Es existiert kein primitiv-rekursiver Halte-Test.

Kapitel 6

Der Satz von Gödel

6.1 Gödelnummerierung von TNT

In Kapitel 4 wurde das FTAN-System in TNT eingebettet, indem jedes Zeichen des FTAN-Systems eine ein-eindeutige Gödelnummer bekommen hat und auch die Ableitungsregeln von FTAN in zahlentheoretische Operationen auf den Gödel-Nummern der Ketten des FTAN-Systems übersetzt wurden. Man konnte dann in TNT die Kette

FTAN-ZAHL[a]

bilden, die interpretiert bedeutet: Die zur Gödel-Nummer a gehörende Kette ist ein Satz des FTAN-Systems. Nun gehen wir einen Schritt weiter und werden den Zeichen von TNT selbst dreistellige Gödel-Nummern zuweisen:

Symbol	*Gödel-Nummer*	*Gedächtnisstütze*
0	666	Die Zahl des Teufels für die geheimnisvolle Null
S	123	Nachfolger: 1, 2, 3, ...
=	111	Wenn auf der Seite liegend, visuelle Ähnlichkeit
+	112	$1 + 1 = 2$
·	236	$2 \cdot 3 = 6$
(	888	(ist auf der Tastatur die 8
)	999	) ist auf der Tastatur die 9
a	262	
’	163	a prime for a prime
∧	161	
∨	616	
→	633	6 impliziert 3 und 3 (?)
¬	223	2 und 2 sind *nicht* 3
∃	333	visuelle Ähnlichkeit
∀	626	
:	636	zwei Punkte, zwei Sechsen

Die TNT-Kette

$$\forall a\colon \neg\ Sa = 0$$

erhält also nun die riesige Zahl 626,262,636,223,123,262,111,666. Es ist hier nützlich, im Sinne der amerikanischen Konvention für die Niederschreibung großer Zahlen, immer drei Ziffern mit einem Komma abzutrennen.
Als nächstes käme jetzt die Übersetzung der Schlussregeln von TNT in arithmetische Operationen an die Reihe, aber diesen Schritt möchte ich mir sparen. An Hand des FTAN-Systems wurde deutlich, dass dies machbar, aber schon für das einfache FTAN-System nicht sofort ersichtlich war. Auch TNT ist ein formales System und daher ist eine Arithmetisierung der Schlussregeln mit Addition, Multiplikation und Zehnerpotenzen zu bewerkstelligen. Das Wissen um die Existenz dieser arithmetischen Schlussregeln für TNT soll nun an dieser Stelle genügen.

6.2 Beweispaare

Ziel dieses Abschnitts ist es, eine TNT-Kette aufzustellen, die so etwas aussagt wie „**1+1=2** ist ein Satz von TNT“. Dies lässt sich am besten mit Beweispaaren erledigen:

Zwei Zahlen m und n bilden ein TNT-Beweispaar*, wenn m die Gödelnummer einer gültigen TNT-Ableitung ist, deren letzte Zeile die Kette mit der Gödelnummer n ist.*

An dieser Stelle ist es nützlich, noch ein letztes Mal an das FTAN-System zu denken und sich den Begriff des FTAN-Beweispaares vorzustellen. Wir wissen nämlich schon, dass die Zahlen

$$m = 98198119811119801 \text{ und } n = 9801$$

ein FTAN-Beweispaar für die Kette **FTAN** bilden, denn in Kapitel 4 wurde 9801 (die Gödelnummer von **FTAN**) folgendermaßen abgeleitet:

1)	981	Axiom
2)	9811	Regel I mit $r = 1$ und $n = 1$
3)	981111	Regel I mit $r = 2$ und $n = 11$
4)	9801	Regel II mit $r = 4$, $n = 1$ und $m = 98$

Durch Vergleich sieht man, dass m die Gödelnummer der Ableitung und n die letzte Zeile ist.
Hat man nun zwei Zahlen gegeben, zum Beispiel $s = 981981198111980$ und $t = 980$, was hat man dann zu tun, um zu entscheiden, ob diese Zahlen ein FTAN-Beweispaar bilden? Man muss die Ableitung noch einmal explizit (sei es

in Gödelnummern oder in Zeichen) ausschreiben und prüfen, ob jeder Schritt mit den Regeln verträglich ist, und natürlich muss auch geprüft werden, ob die unterste Zeile der Ableitung mit der durch n gegebenen Kette übereinstimmt. Man wird feststellen, dass s und t hier kein Beweispaar bilden, aber der wichtigere Gedanke ist der: Um die Beweispaar-Eigenschaft zu prüfen, sind voraussagbar viele Schritte notwendig – die Eigenschaft, ein Beweispaar zu bilden ist primitiv-rekursiv. Und da alle primitiv-rekursiven Eigenschaften in TNT repräsentiert sind, ist auch die FTAN-Beweispaar-Eigenschaft repräsentiert.
Damit ist garantiert, dass es in TNT eine Formel gibt, die besagt, dass etwa die Zahlen a und a' ein FTAN-Beweispaar bilden. Wir kürzen diese Formel mit

FTAN-BEWEISPAAR[a,a']

ab.
Das FTAN-System ist verglichen mit TNT ein sehr einfaches System mit nur einem einzigen Axiom und zwei Schlussregeln. Ableitungen in TNT sind viel komplexer als in FTAN und die Gödelnummern der Ableitungen viel unübersichtlicher. Trotzdem genügen auch in TNT endlich viele Schritte, um dort zu prüfen, ob zwei Zahlen ein Beweispaar bilden, und der Vorgang ist dem im FTAN-System analog. Entsprechend ist also auch die Eigenschaft des TNT-Beweispaares primitiv-rekursiv und somit gibt es auch eine TNT-Formel, die von den Zahlen a und a' aussagt, dass sie ein TNT-Beweispaar bilden:

TNT-BEWEISPAAR[a,a']

Jetzt hat man eine Möglichkeit auszudrücken, dass zum Beispiel die Kette **0=0** ein Satz von TNT ist, denn man kann gleichbedeutend sagen, dass es zur Gödelnummer 666,111,666 dieser Kette eine zweite Zahl gibt, sodass diese beiden ein TNT-Beweispaar bilden. In Zeichen:

∃a: **TNT-BEWEISPAAR**[a,SSS...SSS0]

(Es sind 666,111,666 S's.) Am Anfang dieses Abschnitts wollten wir eine Formel für die Aussage „**1+1=2** ist ein Satz von TNT“ finden – es ist genau diese, nur mit 123,666,112,123,666,111,123,123,666 S's.

6.3 Substitution

Der Schlüssel zum Beweis von Gödels Satz wird eine Kette sein, die von sich selbst aussagt, sie sei kein Satz von TNT. Wir haben aus dem letzten Abschnit eine Formel, die für die Kette mit der Gödelnummer a' aussagt, diese sei kein Satz von TNT:

¬∃a: **TNT-BEWEISPAAR**[a,a']

In diese Formel für a' nun die Gödelnummer ihrer selbst einzusetzen hilft leider nicht weiter – es wäre dem Versuch ähnlich, einen (deutschen) Satz innerhalb seiner selbst zu zitieren, und das ist schwierig, denn man muss dann das Zitat zitieren und das Zitat des Zitats ebenfalls usw.
Doch es gibt (für TNT fatalerweise) einen anderen Weg. Betrachten wir einmal die Formel

$$\text{a} = \text{S0} \quad ,$$

in der die Variable a frei vorkommt und die daher keine Aussage darstellt. Die Gödelnummer der Formel ist 262,111,123,666. Um aus der Formel eine Aussage zu machen, können wir die Variable a durch ein bestimmtes Zahlzeichen, etwa SSS0, austauschen. Die Formel lautet dann

$$\text{SSS0} = \text{S0}$$

und drückt eine Unwahrheit aus, was uns aber jetzt nicht stört. Mit der Gödelnummer passiert ähnliches: Die Ziffern 262 werden gegen 123,123,123,666 ausgetauscht und die Gödelnummer der neuen Formel lautet 123,123,123,666,111,123,666.

An dieser Stellle wieder ein kleiner Test: Gegeben sind die Zahlen

$$\text{b} = 262{,}111{,}123{,}262 \ ; \ \text{c} = 1 \ ; \ \text{d} = 123{,}666{,}111{,}123{,}123{,}666$$

und die Aufgabe besteht darin festzustellen, ob d die Gödelnummer der Formel ist, die entsteht, wenn man in der zu b gehörenden Formel die freie Varable durch das Zahlzeichen für c ersetzt. Das ist leicht nachzuprüfen. b steht für die Formel

$$\text{a} = \text{Sa}$$

und d für

$$\text{S0} = \text{SS0} \quad .$$

Offensichtlich wurde a durch S0 ersetzt – also das Zahlzeichen für 1. Wahrscheinlich ist bereits klar, worauf ich hinauswill: Der Prüfungsprozess, ob drei Zahlen in diesem „Substitutions"-Verhältnis stehen, ist primitiv-rekursiv. Und damit existiert in TNT mit Sicherheit eine Kette

$$\textbf{SUBST}[\text{a,a',a"}] \quad ,$$

die aussagt, dass a" die Gödelnummer einer Kette ist, die man bekommt, wenn man in der Kette, die zu a gehört, alle Vorkommnisse der freien Variablen durch das Zahlzeichen für a' ersetzt.
Wir sind schon fast am Ziel und können die Schlüsselkette, die die Unvollständigkeit von TNT besiegeln wird, bald hinschreiben. Helfen wird dabei auch noch ein „alter Bekannter" – Cantors Diagonalmethode, bei der eine Zahl auf zwei verschiedene Weise gebraucht wird. Betrachten wir dazu einmal die Formel

SUBST[a,a,a"] .

Hier wird die freie Variable in der Kette, deren Gödelnummer a ist, durch das Zahlzeichen für a ersetzt. a wird also als Gödelnummer einer Kette und als substituiertes Zahlzeichen verwendet. Beispiel:

a = 0	⇔	262,111,666
SSS...SSS0 = 0	⇔	123,123,...,123,123,666,111,666

Die Formel a=0 enthält die freie Variable a und hat die Gödelnummer 262,111,666. Das a ist jetzt durch das Zahlzeichen SSS...SSS0 (262,111,666 S's) zu ersetzen. In der Gödelnummer der Formel wird entsprechend 262 durch 123,...,123,666 (262,111,666 Wiederholungen von 123) ersetzt. Dies ist oben geschehen.

Die Technik, in eine Formel ihre eigene Gödelnummer zu substituieren, werden wir nach dem Philosophen Willard von Orman Quine, der eine ähnliche Operation in natürlichen Sprachen erfand, als *Quinieren* bezeichnen, die primitiv-rekursive arithmetische Operation dazu als *Arithmoquinieren*. Für die Aussage *a" ist die Arithmoquinierung von a* schreiben wir anstelle von **SUBST**[a,a,a"] kürzer

ARITHMOQUINE[a,a"]

und wir sind fast am Ziel.

6.4 TNT gibt sich geschlagen

Nennen wir folgende Formel F:

¬∃a:∃a': (**TNT-BEWEISPAAR**[a,a'] ∧ **ARITHMOQUINE**[a",a'])

Formel F hat natürlich eine Gödelnummer, die wir u nennen. u ist eine sehr, sehr große Zahl, die hauptsächlich davon abhängt, wie die Formeln für **TNT-BEWEISPAAR** und **ARITHMOQUINE** aussehen, was wir nicht untersucht haben. Ein paar Bruchstücke von u kann man aber angeben:

$$u = 223, 333, 262, 636, 333, 262, 163, 636, 888, ..., 161, ..., 999$$

In F ist die Variable a" frei, sodass F für sich alleine gar nichts bedeutet. Die freie Variable kann man aber beseitigen, wenn man F quiniert, also für a" das Zahlzeichen für u einsetzt. Man bekommt

¬∃a:∃a': (**TNT-BEWEISPAAR**[a,a'] ∧ **ARITHMOQUINE**[SS...SS0,a'])

mit u-mal dem Buchstaben S im Argument von **ARITHMOQUINE**. Dies ist Gödels Kette G. Bevor wir G interpretieren, stellen wir fest, dass die Gödelnummer von G die *Arithmoquinierung von u* ist. Nun zur Frage, was G aussagt. Sehr roh übersetzt besagt G etwa:

„Es gibt keine Zahlen a und a', sodass beide ein TNT-Beweispaar bilden und a' die Arithmoquinierung von u ist."

u ist jedoch die Gödelnummer einer Kette (nämlich F), die eine freie Variable besitzt, deshalb gibt es mit Sicherheit eine Zahl a', die die Arithmoquinierung von u ist. Das Problem liegt also bei a, und G sagt wirklich:

„Es gibt keine Zahl a, die mit der Arithmoquinierung von u ein Beweispaar bildet."

Deutlicher formuliert heißt das nichts anderes als:

„Die Formel, deren Gödelnummer die Arithmoquinierung von u ist, ist kein Satz von TNT."

Allerdings ist die Formel, deren Gödelnummer die Arithmoquinierung von u ist, nach obiger Überlegung gerade G selbst. Man bringt die Aussage von G also auf den Punkt, wenn man G so übersetzt:

„G ist kein Satz von TNT."

Warum besiegelt die Existenz dieser TNT-Formel dessen Unvollständigkeit? Nun, es stellt sich zuerst die Frage, ob G ein Satz von TNT ist oder nicht. Nehmen wir an, G sei ein Satz von TNT. Dann, so haben wir in Kapitel 4 argumentiert, stellt er eine Wahrheit dar. Seine Aussage ist aber, dass er selbst kein Satz von TNT ist – wir haben also einen Widerspruch und müssen die Annahme verwerfen, G sei ein Satz von TNT. **G ist also kein Satz von TNT**, aber dann ist G wahr, denn seine eigene Nicht-Satzheit sagt G ja aus. Es ist also gezeigt, dass es mindestens eine Kette gibt, die kein TNT-Satz, aber wahr ist, **und damit ist TNT unvollständig**.

Auf den ersten Blick mag es als nicht so schlimm erscheinen, dass eine furchtbar lange TNT-Formel existiert, die zu kompliziert ist, als dass es nützlich wäre, sie hier aufzuschreiben, und die *kein Satz* von TNT ist, aber eine Wahrheit ausdrückt. Man darf jedoch nicht vergessen, was die Formel auf der untersten, sozusagen „wörtlichen", Ebene bedeutet: Es ist eine zahlentheoretische Aussage über natürliche Zahlen! Zugegebenermaßen ist sie etwas komplexer als $7 \cdot 4 = 28$, aber doch eigentlich nichts grundlegend Verschiedenes – und diese Aussage lässt sich (trotz ihrer Richtigkeit) in TNT nicht ableiten.

6.5 Die Wiederholbarkeit von Gödels Beweis

Gerade wurde gezeigt, dass Gödels Kette G kein Satz von TNT ist, aber eine Wahrheit ausspricht. Die Verneinung ¬G ist damit falsch, und weil falsche Aussagen niemals Sätze von TNT sein können, auch kein Satz von TNT. Es gibt also in TNT eine *unentscheidbare Aussage*, da weder G noch ¬G Sätze von TNT sind.
Man könnte sich vorstellen, dieses Problem einfach zu beheben, indem man entweder G oder ¬G als weiteres TNT-Axiom hinzufügt (beide Varianten sind denkbar). Das scheint die beiden Probleme, nämlich die Existenz eines unentscheidbaren Satzes und die Unvollständigkeit des Systems, auf einen Schlag zu lösen – aber der Schein trügt.
Nimmt man etwa G als zusätzliches Axiom in TNT auf, dann hat man ein neues System, das man mit TNT+G bezeichnen kann. TNT+G ist stark genug, dass es primitiv-rekursive Prädikate repräsentieren kann (daran hat sich durch die Hinzunahme eines weiteren Axioms nichts geändert). Es gibt also in TNT+G eine Formel, die die Eigenschaft des TNT+G-Beweispaares repräsentiert, und man sieht schon, wohin das führt – nämlich zu einer Formel G', die von sich sagt „Ich bin kein Satz von TNT+G“ und damit die Unvollständigkeit von TNT+G besiegelt. Die Aufnahme dieser Formel als neues Axiom führt zu TNT+G+G', einem System, in dem es eine Formel G" gibt, die ...

An dieser Stelle wird die Tragweite von Gödels Beweis besonders deutlich und man kann sie mit einem Satz zusammenfassen:

Jedes formale System, das stark genug ist, alle primitiv-rekursiven Prädikate zu repräsentieren, ist prinzipiell unvollständig.

Literaturverzeichnis

[1] **Douglas R. Hofstadter**, Göder Escher Bach, Kett-Cotta 1991

[2] **Kurt Gödel**, Über formal unentscheidbare Sätze der Pricipia Mathematica und verwandter Systeme I, Monatshefte für Mathematik und Physik, Nr. 38, 1931

[3] **Joachim Biskup**, Vorlesung „Grundzüge der Informatik“, ls6-www.informatik.uni-dortmund.de/issi/teaching/lectures/01ws/Grundzuege/Skript/Book.ps

[4] **John Nolt, Dennis Rohatyn, Achille Varzi**, Logic, Schaum's Outline Series, McGraw-Hill, 1998