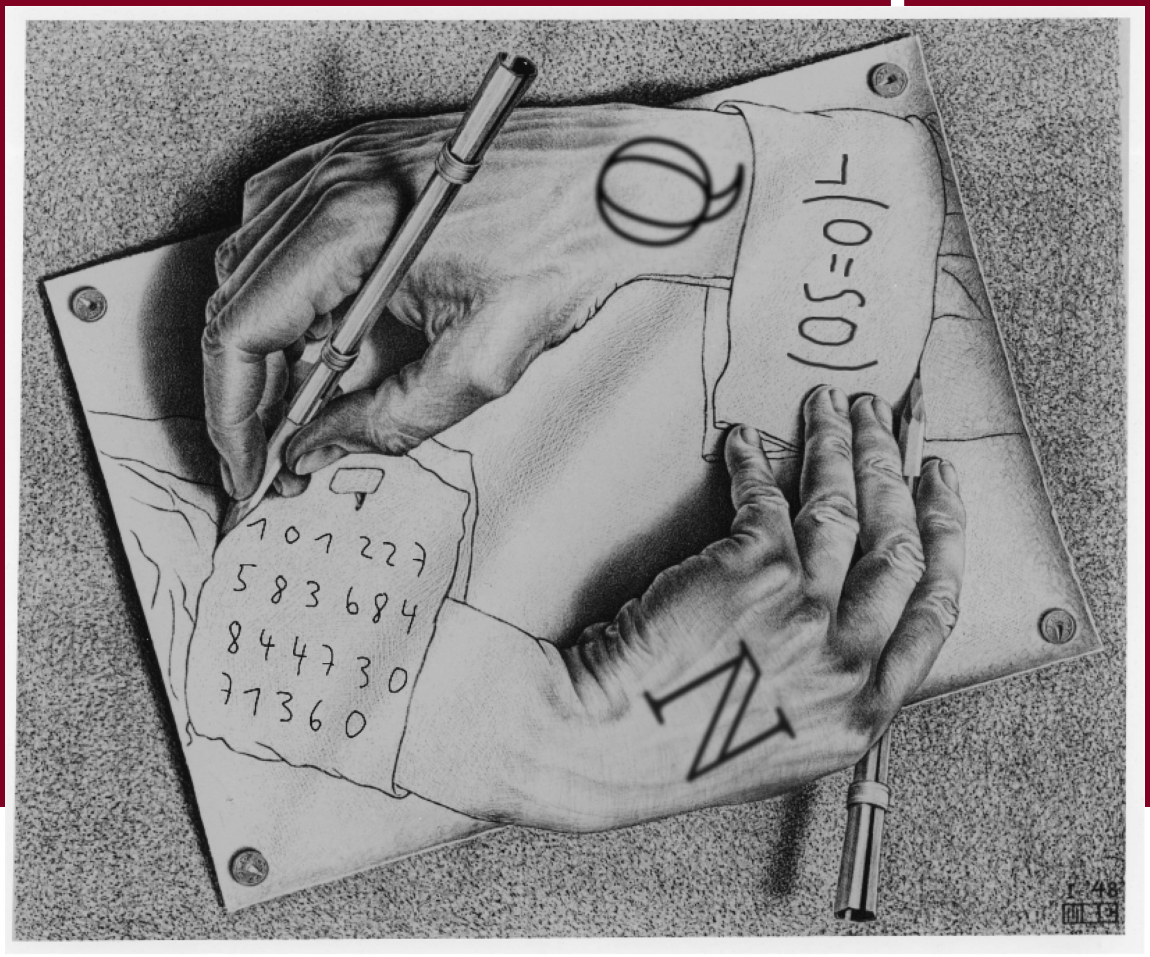
Beweisbar unbeweisbar



Formale Systeme
und
Gödels Unvollständigkeitssätze

Ali Gottschall (4a)

Maturitätsarbeit HS 2019/20
Kantonsschule Im Lee Winterthur
Betreut von Thomas Foertsch
Winterthur, 6. Januar 2020

Inhaltsverzeichnis

Abbildungsverzeichnis

Titelbild

Drawing Hands, Escher 1948, modifiziert vom Autor

1 Einleitung

1.1 Motivation

Über das Unvollständigkeitstheorem von Gödel bin ich das erste mal in einem YouTube-Video von „Numberphile“ [19] gestolpert. Das Feuer war entzündet. In zehn Minuten kann ein solches Thema aber nur seicht abgehandelt werden. Ein Jahr später kam ich in die 3. Klasse und damit begann am Lee für mich der Mathematik Schwerpunkt-Unterricht. In der ersten Lektion stellte unser Lehrer, um unser Interesse zu wecken, einige Perlen der Mathematik vor. Darunter unter anderem Cantors Diagonalargument, auf das wir auch noch zu sprechen kommen, und zu guter letzt Gödels Unvollständigkeitstheorem. Ich erinnerte mich an das Filmchen und bat um eine Vorführung des Beweises. Der Lehrer musste verständlicherweise abwinken, aus dem Stegreif könne er einen solchen Beweis nicht vorzeigen. Er meinte aber, dass es ein tolles Thema für eine Maturitätsarbeit wäre. Eine weitere Motivation kam aus dem Philosophie-Freifach. Dort *verwirrte* ich mich gerne in den Qualitäten von Wissen und Wahrheit und kam immer wieder auf Selbstbezüglichkeit und Meta-Ebenen zu sprechen. Während meinen Recherchen bin ich schnell auf das Buch „Gödel, Escher, Bach“ [12] von Douglas Hofstadter gestossen. Es behandelt das Thema im grossen Bogen mit vielen Hinweisen zur Musik von J.S. Bach und zur Kunst von M.C. Escher. Der Autor beleuchtet auch Selbstbezüglichkeit und *seltsame Schlaufen*[1], wie sie in der Molekularbiologie auftreten[2]. Ich fühlte mich darin gedanklich sehr geborgen und es gab mir erste Visionen von dem, was ich unter anderem in dieser Arbeit erreichen wollte, nämlich formale Systeme abstrakt zu programmieren, dass man selber neue erfinden und damit herumspielen kann, um Sätze zu generieren beziehungsweise abzuleiten.

1.2 Hintergründe

> Aus dem Paradies, das Cantor uns geschaffen, soll uns niemand vertreiben können.
>
> *David Hilbert*
> Über das Unendliche, 1926

Gödels Unvollständigkeitssätze gehören zu den wichtigsten Errungenschaften in der mathematischen Logik. Ihr Einfluss wird manchmal verglichen mit dem der Relativitästheorie Einsteins oder der Heisenbergschen Unschärferelation auf die Physik.

1.2.1 Russells Paradox

Bertrand Russel fand 1901 ein grosses Problem in der naiven Mengenlehre, nämlich ein Paradox. Eine Menge ist zum Beispiel die Menge aller Menschen. Diese Menge beinhaltet sich nicht selbst als Element, denn sie ist schliesslich kein Mensch. Man betrachte nun die Menge aller Mengen, die sich nicht selbst als Element enthalten. Die Menge der Menschen ist ein Element dieser Menge, aber die interessante Frage ist, ob sie sich selbst enthält. Es gibt zwei Möglichkeiten. Entweder enthält sie sich selbst als Element, was aber im Widerspruch mit ihrer Definition steht, denn dann würde

[1] Vgl. Titelbild
[2] Ein Ribosom synthetisiert DNS, die wiederum Ribosome erschafft.

sie eine Menge enthalten, die sich selbst als Element enthält. Wenn sie sich aber nicht selbst als Element enthält, ergibt das auch einen Widerspruch, denn sie müsste ja alle Mengen enthalten, die sich nicht selbst als Element enthalten und in diesem Fall ist sie schliesslich eine Menge, die sich nicht selbst enthält [13].

1.2.2 Principia Mathematica

Das obige Paradox führte zu dem Bestreben von Russell und Whitehead, die Paradoxa der Selbstreferenz in der Mathematik zu umgehen. Das Resultat war die *Principia Mathematica* (eine monumentale Abhandlung, die einen ganzen Schmöker brauchte, um zu beweisen, dass $1 + 1 = 2$ gilt), die es vermochte Russells Paradox zu tilgen. Die Mathematiker atmeten auf, jedoch zu früh. 20 Jahre später erkannte der junge Gödel nämlich, wie er trotzdem Selbstreferenz herstellen kann, und das sie, wenn das System mächtig genug ist, unumgänglich ist.

1.2.3 Hilbertprogramm

In den 1920ern stellte der Mathematiker David Hilbert die Herausforderung auf, die Mathematik zu formalisieren und axiomatisieren, um zu beweisen, dass sie konsistent und vollständig ist [22]. Er stellte eine Liste mit mathematischen Problemen auf, die noch ungelöst waren. Darunter haben drei einen Zusammenhang zu Gödels Sätzen [6].

Hilberts erstes Problem (Kontinuumshypothese) Existiert eine überabzählbare Menge, die von der Mächtigkeit her kleiner ist als die reellen Zahlen?	**Hilberts zweites Problem** Sind die arithmetischen Axiome widerspruchsfrei?	**Hilberts zehntes Problem** Man gebe ein Verfahren an, das für beliebige diophantische Gleichungen entscheidet, ob sie lösbar sind.

1.3 Definitionen

Definition 1. *Vollständigkeit*
Ein formales System ist vollständig, wenn für jede Aussage P, entweder P oder deren Verneinung $\neg P$ mithilfe der Deduktionsregeln zurück zu einem Axiom abgeleitet werden kann, d.h. bewiesen werden kann.

Definition 2. *Widerspruch*
Ein Widerspruch kommt vor, wenn gleichzeitig die Aussagen Q und $\neg Q$ wahr sind.

Definition 3. *Konsistenz*
Ein formales System ist konsistent, wenn es keine Aussage R gibt, für die R und $\neg R$ beweisbar sind, wenn es also widerspruchsfrei ist.

Gödel machte dem Hilbertprogramm den Garaus, indem er zeigte, dass es immer Sätze gibt, die nicht ableitbar sind und, dass das unterliegende Kalkül demnach unvollständig sein muss. Er garnierte es dazu mit einem Beweis, dass ein formales System seine eigene widerspruchsfreiheit nicht beweisen kann.

1.4 Übersicht

Gödel gelang es mithilfe der Gödel-Nummerierung, die zahlentheoretische Aussagen in Zahlen codiert, die Arithmetik über sich selber sprechen zu lassen, also mathematische mit meta-mathematischen Aussagen zu mischen (vgl. Titelbild).

Definition 4. *Lügner-Paradox*[3]
Dieser Satz ist falsch.

Er übersetzte das Lügner-Paradox zu einem mathematischen Äquivalent, das soviel lauten mag wie:

Definition 5. *Gödel Satz G*
Der Gödel Satz sagt von sich selber, dass er nicht abgeleitet werden kann.

Das ist der vermeintliche Gödel-Satz und er verleiht allen formalen Systemen, die genug Arithmetik durchführen können und sich der Widerspruchsfreiheit rühmen, das Attribut der Unvollständigkeit in einer Manier, die an das cantorsche Diagonalargument erinnert. Denn wäre G ein Satz von F, wäre das ein Widerspruch, da Satzheit von G mit der Ableitbarkeit von G in F übereinstimmt, doch G behauptet genau das Gegenteil. Wäre G kein Satz von F, könnte G nicht in F abgeleitet werden, was genau die Aussage von G ist. Das heisst, G ist richtig und müsste ein Satz sein, weil F ja konsistent ist; ein Widerspruch.

[3] Auch bekannt als Paradox des Epimenides: Alle Kreter sind Lügner, sagte der Krete Epimenides.

2 Formale Systeme

Definition 6. *Formales System*
Ein formales System ist ein System von Symbolketten und Regeln. Die Regeln sind Vorschriften für die Umwandlung einer Symbolkette in eine andere, also Produktionen einer formalen Grammatik. Die Anwendung der Regeln kann dabei ohne Kenntnis der Bedeutung der Symbole, also rein syntaktisch erfolgen. [3]
Ein formales System lässt sich als Quadrupel von einem Alphabet und den Mengen der wohlgeformten Formeln, den Axiomen und den Folgerungsregeln auffassen:

$$F = \langle \alpha, W, \mathcal{A}, R \rangle .$$

Das Ganze mag nun noch sehr kryptisch tönen, deswegen will ich es Schritt für Schritt entschlüsseln. Anhand eines simplen Beispiels möchte ich einen ersten Eindruck eines formalen Systems geben und erste Begriffe einführen.

2.1 Das ALI System

[4]

2.1.1 Ketten und Symbole

Definition 7. *Symbole*
Symbole sind im Prinzip alle typographischen Zeichen. Dazu gehören selbstverständlich alle Buchstaben und Zahlzeichen.

Beispiele für Symbole sind `+`, `A`, χ, `$`, `8`. In dieser Arbeit sind Ketten und Symbole mit der Formatierung `Kette` dargestellt.

Definition 8. *Ketten*
Ketten sind Konkatenationen(Aneinanderreihungen) von Symbolen.

Beispiele für Ketten sind `1+3`, `HALLO`, $\chi\rho$, `5$`, `x8+`∞.

2.1.2 Alphabet

Definition 9. *Alphabet*
Ein Alphabet ist die Menge der Symbole, die in einer Kette erlaubt sind.

Folglich ist K eine Kette von einem formalen System F, wenn für jedes $s \in K$ gilt s $\in \alpha$. Das Alphabet von ALI ist sehr überschaulich. Es beschränkt sich auf die Buchstaben `A`, `L` und `I`.

$$\alpha_{ALI} = \{A, L, I\}.$$

Zum Beispiel sind `A`, `ILA`, `LAILALIALA`, `III` Ketten von ALI, weil sie nur die Buchstaben A, L und I enthalten, während die Ketten `AB`, `CDE`, `1+2` keine Ketten von ALI sind, weil sie auch andere Buchstaben enthalten.

[4] Inspiriert von Hofstadters MU Rätsel (GEB Kapitel I, s37ff)

2.1.3 Axiome

> Perfektion ist erreicht, nicht,
> wenn sich nichts mehr hinzufügen
> lässt, sondern, wenn man nichts
> mehr wegnehmen kann.
>
> *Antoine de Saint-Exupery*

Definition 10. *Axiome sind Sätze in einem System, die nicht weiter abgeleitet werden können. Sie werden als wahr definiert. Sie haben ihren Wahrheitsgehalt nur bezüglich des Systems.*

$$\mathcal{A}_{ALI} = \{I\}.$$

In *ALI* gibt es nur ein einziges Axiom und das ist `I`. Axiome sind die Grundlage für alle weiteren Sätze. Ein System kann auch über mehrere Axiome verfügen oder sogar mittels Axiomen-Schemata über unendlich viele.

2.1.4 Folgerungsregeln

Mit den 3 Folgerungsregeln von *ALI* lassen sich neue Sätze produzieren:

1. Bei einer beliebigen Kette x, füge `A` am Anfang an: $x \mapsto$ `A`x z.B.: `I` $\vdash$ `AI`, `LALI` $\vdash$ `ALALI`. Wobei die Notation $a \vdash b$ bedeutet, dass die Kette B sich aus A produzieren lässt.
2. Kommt `A` in einer Kette vor, ersetze es mit `LAL`, z.B.: `AI` $\vdash$ `LALI`, `ALALI` $\vdash$ `LALLALI`, `ALLALLI`. Man beobachte, dass die Definition rekursiv ist, denn im Produkt kommt wieder ein `A` vor.
3. Lösche zwei aufeinanderfolgende `L`s: `LL` $\mapsto \varnothing$ z.B.: `ALLI` $\vdash$ `AI`, `LLALLI` $\vdash$ `ALLI`, `LLAI`

Bei den letzen zwei Regeln ist es möglich, dass man von einem Satz mit einer Regel mehrere neue Sätze produzieren kann.

2.1.5 Produktion & Sätze

Definition 11. *Sätze*

1. *Alle Axiome sind Sätze.*
2. *Alle Ketten, die aus Sätzen (also auch Axiomen) mit einer Folgerungsregel geformt werden können, sind Sätze.*

Nach der Definition lässt sich ein Algorithmus entwerfen, der uns Sätze eines formalen Systems generiert.

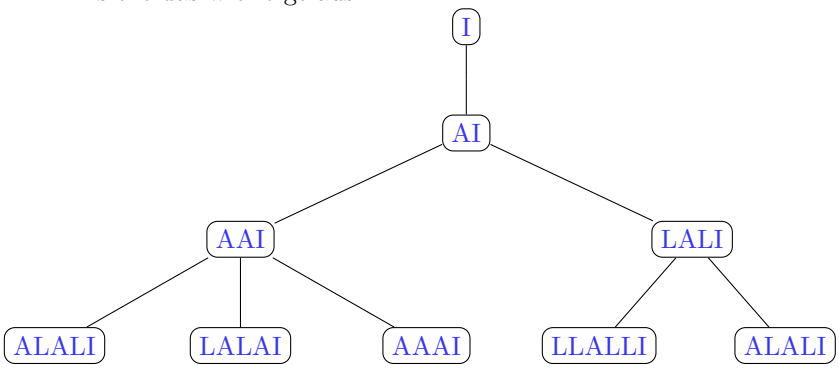
1. Wirf alle Axiome in einen Eimer.
2. Für jeden Satz im Eimer, wende alle Produktionsregeln an (wenn möglich).
3. Wirf alle neuen Sätze in einen neuen Eimer
4. Wiederhole Schritt 2 und 3.

```
from folgerungsregeln_anwenden import folgerungsregeln_anwenden

def produktion(axiome, regeln):
    eimer = axiome # Schritt 1
    while True: # Schritt 4
        if eimer:
            neuer_eimer = []
            for satz in eimer:
                neu = folgerungsregeln_anwenden(satz, regeln) # Schritt 2
                for neuer_satz in neu:
                        neuer_eimer.append(neuer_satz) # Schritt 3
                        yield neuer_satz
            eimer = neuer_eimer # der neue Eimer wird zum aktiven
        else:
            break
```

Listing 1: Produktionsalgorithmus

In *ALI* sieht das wie folgt aus:



2.1.6 Ableitung

Wenn man eine Kette hat und herausfinden will, ob sie ein Satz eines Systems ist, muss man den Satz beweisen. Das bedeutet, man muss eine Reihe von Sätzen finden, an deren Anfang ein Axiom steht und am Schluss der besagte Satz, und zwischen zweien solcher Sätze muss man immer eine Folgerungsregel legal anwenden können. Wir entwerfen wieder einen Algorithmus, der das für uns ausführt.

1. Wende den Produktionsalgorithmus an, bis die zu beweisende Kette produziert ist. Speichere dabei bei jedem neuen Satz den Elternsatz[5] mit ab. Definiere den Satz als aktuellen Satz.
2. Zurückverfolgen: Gehe zum Elternsatz vom aktuellen Satz.
3. Wenn der Elternsatz ein Axiom ist, terminiere.

[5] der Satz, aus dem der neue produziert wurde

4. Sonst wiederhole Schritt 2.

```
# importiere den Produktionsalgorithmus
from produktion import produktion

def beweise(axiome, regeln, satz):
    for theorem in produktion(axiome, regeln):
        if theorem == satz:
            return True
    return False
```

Listing 2: Ableitungsalgorithmus

Eine andere Möglichkeit wäre, mit dem zu beweisenden Satz zu beginnen und die Folgerungsregeln umgekehrt anzuwenden. Trifft man auf ein Axiom, kann man die gleiche Rückverfolgung durchspielen, um eine Ableitung zu erhalten.

2.1.7 Rätsel

Man mag sich die Frage stellen, ist `ALI` ein Satz von *ALI*? Probieren Sie es gerne selber und spielen mit dem System herum!

Blättern Sie um für die Auflösung oder springen Sie ans Ende der Arbeit und nehmen mein Programm zur Hilfe.

2.1.8 Unentscheidbarkeit

Wenn Sie nach einiger Zeit aufgegeben haben, kann ich Sie beruhigen, denn, nein, es ist nicht möglich. Aber wieso? Im System selber kann man es nicht beweisen. Man produziert nur noch mehr neue Ketten, aber man weiss ja nie, ob doch noch die richtige kommt. Der Algorithmus steckt in einer endlosen Schleife und produziert Sätze, jedoch kann er Nicht-Sätze nicht aufzählen. Um zu zeigen, dass es unmöglich ist, muss man aus dem System *springen*, um mit höheren Systemen sogenannte Meta-Aussagen zu machen. Zum Beispiel kann man beweisen, dass `ALI` unmöglich zu produzieren ist, weil der Rest, der bei der Division der Anzahl `L`s durch 2 übrig bleibt, konstant ist. Wir nennen die Anzahl `L`s n und n' die Anzahl nach Anwendung einer Regel.

1. Bei Regel 1 ist $n' = n$, weil nur `A`s hinzugefügt werden.
2. Bei Regel 2 ist $n' = n + 2$ und $n' \equiv n \mod 2$. Einfach erklärt heisst das, dass wenn es zuvor 3 `L`s und damit danach 5 `L`s gibt, weil ein `A` in `LAL` umgewandelt worden ist, bleibt der Rest bei Division mit 2 konstant.
3. Bei Regel 3 ist $n' = n - 2$ und $n' \equiv n \mod 2$ analog zu Regel 2 mit dem Unterschied, dass 2 `L`s gelöscht werden.

Weil im Axiom $AnzahlL($`I`$) \equiv 0 \mod 2$ aber $AnzahlL($`ALI`$) \equiv 1 \mod 2$ gibt, ist `ALI` unerreichbar.

2.2 Einführung Logik

2.2.1 Aussagenlogik

Die Aussagenlogik befasst sich mit den Beziehungen von Aussagen. Der Inhalt ist nicht von Belang, sondern der Wahrheitsgehalt [20]. Jede Aussage ist entweder *wahr* oder *falsch* (Bivalenz). Eine Aussage ist zum Beispiel:
`Sokrates ist ein Mensch`. In der Aussagenlogik würde man das mit einer *Aussagenvariable* ausdrücken: P. Aussagenvariablen stellen wir mit Grossbuchstaben dar, und bei Bedarf können wir unendlich viel neue erzeugen, indem wir Striche `'` anfügen. Die folgenden Aussagenvariablen sind alle wohlgeformt.
`P`, `Q`, `P'`, `R''''`

1. Jede Aussagenvariable ist eine Formel.
2. Ist P eine Formel, ist auch $\neg P$ eine Formel ($\neg$ bedeutet Negierung, wenn P wahr ist ist $\neg P$ falsch und wenn P falsch ist ist $\neg P$ wahr).
3. Sind P und Q Formeln, dann sind auch $(P \wedge Q)$, $(P \vee Q)$ und $(P \implies Q)$ Formeln.

„$\wedge$“, ist was wir in der natürlichen Sprache unter „*und*“ verstehen. $(P \wedge Q)$ ist nur wahr, wenn P und Q wahr sind.
„$\vee$“, ist was wir in der natürlichen Sprache unter „*oder*“ verstehen. $(P \vee Q)$ ist nur wahr, wenn P oder Q wahr sind.
„$\implies$“, ist was wir in der natürlichen Sprache unter „*impliziert*“ verstehen (wenn P dann Q). $(P \implies Q)$ ist wahr ausser P ist wahr und Q ist falsch.

Der Wahrheitswert von einer zusammengesetzten Aussage ist eindeutig über die Wahrheitswerte der Teilaussagen gegeben (Extensionalität).
Beispiel (Syllogismus):

Sokrates ist menschlich.	S ist Q.
Alle Menschen sind sterblich.	Wenn Q, dann P.
Also ist Sokrates sterblich.	S ist P.

2.2.2 Prädikatenlogik

Die Prädikatenlogik stellt eine Erweiterung der Aussagenlogik dar. Ein Prädikat ist sozusagen eine Leerstelle in einem Satz:
Politiker(x) ist korrupt. (x ist eine ungebundene Variable)
$\forall x$ Politiker(x) ist korrupt. (Nun ist x gebunden, weil es quantifiziert worden ist. $\forall$ bedeutet „für alle“.)

Der andere Quantor neben $\forall$, der üblicherweise benutzt wird, ist der Existenzquantor $\exists$, der besagt, dass für mindestens ein x eine Formel wahr ist.

2.3 Formalisierte Arithmetik

Um die Arithmetik genauer analysieren zu können, um schliesslich die Unvollständigkeit zu zeigen, benötigen wir vorerst eine formale Definition davon. Wir interessieren uns für den Teil der Mathematik, der sich mit natürlichen Zahlen $\mathbb{N}$ beschäftigt. Natürliche Zahlen sind die, die man beim Zählen verwendet (0, 1, 2, 3,...). Ob 0 dazugezählt wird ist Definitionssache. Umgangssprachlich wird jeweils von der „Mathematik“ gesprochen, als wäre sie eine einzige Theorie. Doch es gibt eine Vielzahl von unterschiedlichen Theorien, sogar unendlich viele. Deshalb wollen wir hier eine schwache Arithmetik als Grundlage definieren. Unsere Theorie soll unsere „normalen“ Intuitionen reflektieren. Ihr Schauplatz sind die natürlichen Zahlen. Wie lassen die sich formaliseren? Dafür definieren wir die Sukzessor-Funktion S[6]. Alles, was sie tut, ist das erhöhen einer Zahl um 1.

$$\begin{aligned} S(0) &= 1 \\ S(1) &= 2 \\ S(S(S(0))) &= 3 \\ S(n) &= n + 1 \end{aligned}$$

2.3.1 Robinsons Arithmetik

Somit brauchen wir also nur eine Konstante, in diesem Fall 0, die mit der Sukzessor-Funktion per mathematische Induktion die Unendlichkeit von $\mathbb{N}$ aufzählt. Das System stellt eine Erweiterung der Prädikatenlogik dar. Deswegen stehen uns der All ($\forall$)- und der Existenz-Quantor ($\exists$) zur Verfügung. Mit dabei sind die logischen Verbindungen *und* ($\wedge$), *oder* ($\vee$) und *folglich* ($\Longrightarrow$). Für die Gruppierung benutzen wir Klammern. Ausserdem brauchen wir eine unendliche Menge an Variablen und ein Gleichheitssymbol (=). Für die formale Defintion müssen wir nun die Axiome

[6] Sukzessor: folgendes Element.

aufstellen. Die folgenden sieben Axiome definieren die Eigenschaft des Sukzessors, der Addition und der Multiplikation [10].

Definition 12. *Robinsons Arithmetik Q*

Q1 $\neg(0 = S(x))$ *0 ist die kleinste Zahl, Verankerung*

Q2 $S(x) = S(y) \implies x = y$

Q3 $x + 0 = x$ *0 als neutrales Element*

Q4 $x + S(y) = S(x + y)$ *Assoziativität*

Q5 $x \times 0 = 0$ *absorbierendes Element*

Q6 $x \times S(y) = (x \times y) + x$ *Distributivgesetz*

Q7 $\neg(x = 0) \implies \exists y(x = S(y))$ *Wenn x grösser als Null ist, gibt es eine Zahl y, die um eins kleiner ist als x*

Mit diesem Rüstzeug sind wir schon ziemlich gut ausgestattet. Nun können wir beliebige zahlentheoretische Aussagen in Q übersetzen. Zur Illustration der Mächtigkeit möchte ich zeigen, dass man auch die Aussage, dass es unendlich viele Primzahlen gibt, darstellen kann. Eine Primzahl ist eine Zahl, die nicht gleich dem Produkt zweier Zahlen ist, ausser 1 und sich selber. So können wir die Aussage aufstellen, dass 5 eine Primzahl ist, wenn es keine zwei Zahlen grösser als 1 sind, die zusammen als Produkt 5 ergeben. Damit die Zahlen grösser als 1 sind verwenden wir einen Trick und wenden den Sukzessor zweimal auf die Variablen an, die folglich nicht kleiner als 2 sein können. Der Leserlichkeit wegen sind die Klammern des Sukzessors verschluckt worden (S statt $S()$).

$$\neg\exists a : \exists b : SSSSS0 = (SSa \cdot SSb).$$

Wie bringt man nun die Unendlichkeit hinein? Wenn es unendlich viel Primzahlen gibt, gibt es für jede Zahl eine grössere Primzahl. Die finale Aussage ist also: Für jede Zahl c gibt eine Zahl $c + Sd$, die mindestens um eins grösser ist als c, die nicht als Produkt zweier Zahlen grösser als Null dargestellt werden kann und folglich eine Primzahl ist.

$$\forall c : \exists d : \neg\exists a : \exists b : (c + Sd) = (SSa \cdot SSb).$$

Definition 13. *Berechenbare Funktion*
Eine Funktion heisst berechenbar, wenn ein Algorithmus existiert, der sie berechnen kann.

2.3.2 Primitiv rekursive Funktionen

Sie sind berechenbar und der Algorithmus terminiert in finiter Zeit. Das heisst, der Algorithmus könnte in einer Programmiersprache mit sogenannter FOR-Schlaufe (führt einen Programmblock eine bestimmte Anzahl mal aus) definiert werden.

2.3.3 Partiell-rekursive Funktionen

Formulierte man partiell-rekursive Funktionen in einer Programmiersprache und liesse das Programm laufen, könnte es möglicherweise unendlich lange laufen. Die FOR-Schlaufe genügt nicht mehr, und es braucht die WHILE-Schlaufe (ein Programmblock wird solange ausgeführt bis eine Kondition erfüllt ist, möglicherweise unbestimmt lang) [11].

Theorem 1. *Alle partiell-rekursiven Funktionen sind darstellbar in der Robinson Arithmetik Q.*

Dieses Theorem wird an dieser Stelle nicht bewiesen. Zusammengefasst wird gezeigt, dass man mit den Grundfunktionen (Konstante[7], Sukzessor[8], Projektionen[9] mithilfe Komposition und primitiver Rekursion alle primitiv-rekursiven Funktionen herstellen kann. Nimmt man den μ-Operator[10] dazu erweitert man die Menge zu allen partiell-rekursiven Funktionen.

Definition 14. *Rekursiv aufzählbare Menge*
Eine Menge von natürlichen Zahlen heisst in der Berechenbarkeitstheorie rekursiv aufzählbar, wenn es einen Algorithmus gibt, der die Elemente dieser Menge aufzählt [9].

Die Menge aller Theoreme von *ALI* ist rekursiv aufzählbar. Es lässt sich ein Algorithmus schreiben, der alle Theoreme iterativ ausspuckt (vgl. Produktionsalgorithmus). Dies ist die Ausgabe meines Programmes, wenn ich ihm befehle, 10 Theoreme zu produzieren. In der rechten Spalte sehen Sie die Ausgabe, wenn der Algorithmus ein bisschen gerattert hat, das letzte Theorem ist nämlich das 42024te, das durch die sequentielle Anwendung der Produktionsregeln ausgegeben wird.

```
AI <- Rule 0 from I                    LALALLLLALLLLLLLALLLLLLLLLLALLLLLI
AAI <- Rule 0 from AI                  LALALLALLLLLLLALLLLLLLLLLALLLLLI
LALI <- Rule 2 from AI                 LLALLLLLALLLLLALLLLLLLALLLLLLLLLLALLLLLI
LALAI <- Rule 2 from AAI               LALAALALLLLLLLLLLALLLLLI
LALLALI <- Rule 2 from AAI             LALAALLLLLALLLLLLLLLLALLLLLI
AAAI <- Rule 0 from AAI                LALLALLLALLLLALLLLLLLALLLLLLLLLLALLLLLI
LLALLI <- Rule 2 from LALI             LALALALLALLLLLALLLLLLLLALLLLLI
ALALI <- Rule 0 from LALI              LALLALLLALLLALLLLLALLLLLLLLALLLLLI
LLALLAI <- Rule 2 from LALAI           AAALALLALLLLLALLLLLLLLALLLLLI
LLALLLALI <- Rule 2 from LALAI         LALALLLALLLLLALLLLLLLLALLLLLI
```

[7] $C_i^n = (x_1, \ldots, x_n) = i$
[8] $S(x) = x + 1$
[9] $\Pi_i^n(x_1, \ldots, x_n) = x_i$ Gibt ein Argument zurück, vergleiche mit einer geometrischen Projektion eines Graphen auf eine Achse
[10] Minimierungsoperator, kann verstanden werden als WHILE Programm, dass nach oben zählt, bis die Funktion verschwindet [11].

2.3.4 Rekursive Menge

Definition 15. *Rekursive Menge*
Eine Menge heisst rekusiv, wenn sie und ihr Komplement[11] *rekursiv aufzählbar sind.*

Beispiele für Rekursive Mengen sind die geraden Zahlen sowie die Primzahlen. Durch die Menge der geraden Zahlen ist direkt die Menge aller ungeraden definiert, respektive die Zusammengesetzen Zahlen (und 1) durch die Primzahlen. Im übertragenen Sinne kann man sich das so vorstellen: Wenn eine Figur einen Hintergrund hat, und dieser Hintergrund wiederum eine Figur ist, ist das Bild sozusagen rekursiv. Betrachten sie dazu das Bild von Escher. Die Menge aller Theoreme von *ALI* ist, wie gesagt, rekursiv aufzählbar, aber nicht rekursiv, daraus kann man schliessen, dass die Menge aller Nicht-Sätze nicht rekursiv aufzählbar ist. Wir können uns nicht sicher sein, ob `ALI` ein Theorem ist oder nicht. Die einzige Möglichkeit ist, den Produktionsalgorithmus rattern zu lassen, Tee zu trinken und abzuwarten bis es produziert wird. Doch womöglich müssen wir unendlich lange warten, weil die Menge aller Theoreme unendlich gross ist und niemals erschöpft wird.

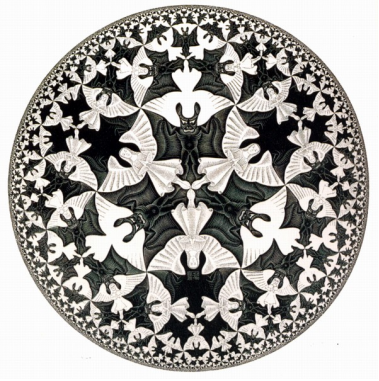


Abbildung 1: Circle Limit IV (Heaven and Hell), Escher 1960

2.4 Repräsentierbarkeit

Die Repräsentierbarkeit ist informell gesagt ein Konzept, um eine Isomorphie zwischen Bedeutung und Symbolen zu schaffen. Die Symbole an sich tragen keine intrinsische Bedeutung, aber sie können Eigenschaften repräsentieren. Beispielsweise herrscht eine Isomorphie zwischen dem Symbol `∧` und dem *und*, das uns allgemein bekannt ist.

Definition 16. *Stark repräsentierbar*
Eine Menge M heisst stark repräsentierbar, wenn es eine Formel $A(x)$ im System F gibt, für die gilt:

$$n \in M \rightarrow F \vdash A(n);$$
$$n \notin M \rightarrow F \vdash \neg A(n).$$

Die Menge der Primzahlen $\{2, 3, 5, 7, \ldots\}$ lässt sich stark repräsentieren:

$$P(x) = \neg\exists a : \exists b : x = (SSa \cdot SSb)^{12} \tag{1}$$
$$\neg P(x) = \exists a : \exists b : x = (SSa \cdot SSb) \tag{2}$$

Definition 17. *Schwach repräsentierbar*
Eine Menge M heisst schwach repräsentierbar, wenn es eine Formel $A(x)$ im System F gibt, für die gilt:

$$n \in M \rightarrow F \vdash A(n).$$

[11] Menge aller Elemente, die nicht in einer Menge sind (sozusagen das Gegenteil)

Die *kritische Masse* eines Systemes ist erreicht, sobald alle primitiv-rekursiven Begriffe darin repräsentierbar sind.

Theorem 2. *Repräsentierbarkeits Theorem*
In einem konsistenten System, das Q enthält, gilt [17]:

1. *Eine Menge ist stark repräsentierbar, genau dann wenn sie rekursiv ist.*
2. *Eine Menge ist schwach repräsentierbar, genau dann wenn sie rekursiv aufzählbar ist.*

3 Unvollständigkeitssätze

3.1 Arithmetisierung & Gödel Codierung

3.1.1 Arithmetisierung von ALI

Da *ALI* nur über drei Symbole verfügt, stellt die Codierung kein Problem dar: Wir weisen `A`, `L` und `I` respektive die Zahlen 1, 2, und 3 zu. So ist die 2123 die zugewiesene Zahl für `LALI`. Die erste Regel lässt sich nun auf der Ebene von $\mathbb{N}$ beschreiben[13]. Die Notation $\lfloor x \rfloor$ bedeutet, dass x auf die nächstkleinste Ganzzahl abgerundet wird.

$$x \overset{f}{\mapsto} x + 10^{\lfloor \log_{10}(x)+1 \rfloor}.$$

Wenden wir die arithmetisierte erste Regel auf die zu `LALI` assozierte Zahl an: $f(2123) = 12123$. Übersetzen wir das Ergebnis zurück, bekommen wir `ALALI` und sehen, dass das die korrekte Anwendung der ersten Regel war.

$$regel1(\ulcorner x \urcorner) = \ulcorner Ax \urcorner.$$

Für die anderen Regeln funktioniert das Prinzip analog.
Gödel hatte noch keinen Zugriff zu einem Computer, deswegen musste er die Berechenbarkeit über die Zuordnung von Formeln zu Zahlen belegen. Diese Zuordnung muss natürlich eindeutig sein. Um das zu garantieren, nahm sich Gödel den Fundamentalsatz der Arithmetik zu Hilfe.

3.1.2 Grundlagen

Theorem 3. *Fundamentalsatz der Arithmetik Für jede natürliche Zahl existiert eine eindeutige Primfaktorzerlegung.[7]*

$$z = p_1 \cdot p_2 \cdot \ldots \cdot p_n.$$

Beweis. Für die Existenz der Primfaktorzerlegung.
Für $0, 1 \in \mathbb{N}$ ist jeweils nichts zu zeigen.
Angenommen es gibt nicht für jede Zahl eine Primfaktorzerlegung, muss es eine kleinste Zahl k geben, für die das nicht der Fall ist. Die Zahl k kann keine Primzahl sein, weil sie dann die eigene Primfaktorzerlegung wäre. Folglich gibt es zwei Teiler $a, b \in \mathbb{N}$, dass $a \cdot b = k$, wobei $1 < a, b < n$. Weil n die kleinste Zahl ist, für die keine Primfaktorzerlegung existiert, muss es Primfaktorzerlegungen für $a = \prod p_i$ und $b = \prod q_j$ geben. Dann ist aber $\prod p_i \cdot \prod q_j = k$ eine Primfaktorzerlegung für k, was ein Widerspruch zur Annahme ist. □

[13] Hier werden schon höhere Strukturen wie die Potenzierung oder der Logarithmus benutzt. Diese könnte man tiefer herunterbrechen, das wird Ihnen aber an dieser Stelle erspart.

Beweis. Für die Eindeutigkeit der Primfaktorzerlegung.
Wenn die Primfaktorzerlegung nicht eindeutig ist, muss es eine kleinste Zahl k existieren, für die es mindestens zwei Zerlegungen $\prod p_i, \prod q_j$ gibt. Davon können keine zwei Faktoren $p_u = q_v$ gleich sein, weil es sonst für die Zahl $\frac{n}{p_u}$ auch zwei verschiedene Zerlegungen gäbe. $\frac{n}{p_i} < n$, Widerspruch zur Annahme. Da ein beliebiger Faktor p_u das Produkt $n = \prod q_j$ teilt, teilt er nach dem Lemma von Euklid auch einen geeignet gewählten Faktor dieses Produkts. p_u kann aber keinen Faktor q_v teilen, weil sie nicht gleich sind und alles Primzahlen sind. Ein Widerspruch. □

3.1.3 Zuordnung

Wir ordnen allen primitiven[14] Symbolen eine Zahl zu [16].

primitives symbol $s \to \#(s)$			
0	1	(	6
S	2	)	7
$+$	3	$\neg$	8
$\times$	4	$\to$	9
$=$	5	$\forall$	10
		x_i	$11 + i$

Weil es unendlich viele Variablen gibt definieren wir die $i$te Varible x_i als $11 + i$. Die erste wird so der 11 zugeordnet, die zweite der 12 usw.

3.1.4 Übersetzung

Wie man von einer Formel A zu ihrer Gödel-Nummer $\ulcorner A \urcorner$[15] kommt, lässt sich am besten an einem Beispiel illustrieren. Wir wollen die Gödel Zahl von der Kette $\neg(0 = S0)$ [16] finden.

1. Der erste Schritt ist ganz simpel. Wir suchen zu jedem Symbol s in der Formel die Codezahl $\#(x)$.

2. Jetzt kommt der Trick. Für jede Zahl z_i, die wir so erhalten haben, potenzieren wir die $i$te Primzahl p_i zu ihr.

3. Schliesslich multiplizieren wir alles zusammen.

[14] Alle anderen Symbole können aus der Kombination der primitiven definiert werden.
[15] Beachten Sie den Unterschied, die Notation $\#(x)$ steht für die assozierte Zahl, währen $\ulcorner x \urcorner$ die Gödel Nummer zur Formel x bedeutet.
[16] 0 ist nicht gleich 1

$$\neg(0 = S0)$$
$$8, 6, 1, 5, 2, 1, 7$$
$$2^8 \cdot 3^6 \cdot 5^1 \cdot 7^5 \cdot 11^2 \cdot 13^1 \cdot 17^7$$
$$256 \cdot 729 \cdot 5 \cdot 16807 \cdot 121 \cdot 13 \cdot 410338673$$
$$10122758368484473071360$$
$$\ulcorner \neg(0 = S0) \urcorner = 10122758368484473071360$$

Das ganze Theater lässt sich auch umkehren.

$$Göd(x) = \ulcorner x \urcorner$$
$$Göd^{-1}(\ulcorner x \urcorner) = x = Göd^{-1}(Göd(x))$$

Dank der Eindeutigkeit der Primfaktorzerlegung ist das ein leichtes Vorhaben[17]. Wir zerlegen eine Zahl in alle Faktoren und zählen dann, wieviel mal die i te Primzahl darin vorkommt. Wenn die 3 zwei mal Faktor ist, bedeutet das, dass das zweite Symbol der Formel hinter der Codenummer 2 Verborgen ist, in unserem Fall S. Wir wollen zum Beispiel wissen, für welche Formel die Gödelzahl 180 steht.

$$180 \tag{3}$$
$$2 \cdot 90 = 2 \cdot 9 \cdot 10 = 2 \cdot 2 \cdot 3 \cdot 3 \cdot \cdot 5 \tag{4}$$
$$2^2 \cdot 3^2 \cdot 5^1 \tag{5}$$
$$\texttt{SS0} \tag{6}$$
$$\ulcorner \texttt{SS0} \urcorner = 180 \tag{7}$$

Wofür das Ganze? Wir haben nun eine Abbildung von Formeln, die über $\mathbb{N}$ sprechen, auf $\mathbb{N}$. Metaphorisch gesehen können jetzt Zahlen über Zahlen sprechen. Das besondere ist, dass Operationen arithmetisch imitiert werden können.

$$neg(\ulcorner A \urcorner) = \ulcorner \neg A \urcorner.$$

Die Syntax von Q wird reflektiert, in dem es eine Funktion neg gibt, die mit dem Argument der Gödelzahl einer Formel die Gödelzahl derjengen Formel zuweist, die die Negation der ursprünglichen Formel darstellt (vgl. erste Regel ALI).
Es ist auch möglich, Operationen zu definieren, die Variablen in Formeln substituieren, oder zu testen, ob eine Gödelzahl ein Axiom ist. Dabei kommen, wie im ersten Beispiel zu sehen ist, schnell immense Zahlen zustande. Doch die praktische Anwendung war auch nie das Ziel, es geht nämlich nur um das Prinzip. Im Appendix kann man begutachten, wie man die folgende Relation in einer Programmiersprache implementieren könnte. Das erspart uns unglaublich viel mühselige und unsinnige Arbeit, wenn wir die Funktion arithmetisch definieren wollten.

[17] Praktisch gesehen nicht, die Verschlüsselungstechnik RSA basiert zum Beispiel darauf, dass grosse Zahlen nicht verlässlich schnell faktorisiert werden können.

```
from folgerungsregeln_anwenden import folgerungsregeln_anwenden

def val(beweis, satz):
    """ Stellt die Liste von Sätzen `beweis` einen legitimen Beweis
    für `satz` dar? """
    last = None
    for i, step in enumerate(reversed(beweis)):
        if last is not None:
            options = folgerungsregeln_anwenden(last, regeln)
            if verbose: print(options)
        if i == 0: # Der Beweis muss in einem Axiom münden
            if step not in [aaxiom for a in axioms]:
                return False
        elif step not in options:
            return False # Der Beweisschritt ist falsch
        last = step
    options = folgerungsregeln_anwenden(last, regeln)
    if not satz in options:
        return False
    return True # Valider Beweis
```

Listing 3: Algorithmus für Beweistest

Die Relation val_F besagt, dass $\ulcorner x \urcorner$ die Gödelzahl der Ableitung der zu beweisenden Formel mit Gödelzahl $\ulcorner y \urcorner$ ist.

$$val_F(\ulcorner x \urcorner, \ulcorner y \urcorner).$$

Bis anhin sind alle Operationen stark repräsentierbar und entscheidbar. Für die nächste Eigenschaft gilt das nicht mehr.

$$bew_F(y) := \exists a : val_F(a, \ulcorner y \urcorner).$$

y Diese Aussage bedeutet, dass eine Ableitung a für die Formel y existiert, was widerum heisst, dass x ableitbar beziehungsweise beweisbar in der Theorie F ist.

$$F \vdash y \Rightarrow F \vdash bew_F(\ulcorner y \urcorner) \ .$$

3.2 Diagonalisierungslemma

Das Diagonalisierungslemma ist das Herz und Hirn des ersten Unvollständigkeitstheorems. Salopp gesagt, wird so Selbstreferenz erzeugt. Man kann sich vorstellen, dass der Satz D die Eigenschaft A hat, weil immer wenn D in F ableitbar ist, $A(\ulcorner D \urcorner)$ wahr, und umgekehrt falsch ist, wenn D nicht ableitbar ist [18]. Die Selbstbezüglichkeit ist eine schöne Heuristik für die bessere Vorstellung, aber eigentlich haben wir nur eine Äquivalenz von zwei logischen Formeln.

Theorem 4. *Diagonalisierungslemma Für jedes einstellige Prädikat A im formalen System F existiert ein Satz D, der äquivalent ist zum Prädikat mit $\ulcorner D \urcorner$ der Gödelzahl von D eingesetzt.*

$$F \vdash D \Leftrightarrow A(\ulcorner D \urcorner).$$

Beweis. Wir benutzen die arithmetisierte Substitutionsfunktion *subst*, die in einem Prädikat alle Vorkomnisse einer ungebundenen[18] Variable x, durch einen Term n ersetzt.

$$subst(\ulcorner A(x) \urcorner, n) = \ulcorner A(n) \urcorner.$$

Die Formel $S(x, y, z)$ repräsentiert diese Operation als eine Relation stark.

$$S(x, y, z) \Leftrightarrow x = \ulcorner A(x) \urcorner \wedge y = n \wedge z = \ulcorner A(n) \urcorner.$$

Eine spezielle Untergruppe der Substitution ist die Quinierung[19]. Dabei wird eine Formel in sich selbst eingesetzt.

$$quin(\ulcorner A(x) \urcorner) = \ulcorner A(\ulcorner A(x) \urcorner) \urcorner = subst(\ulcorner A(x) \urcorner, \ulcorner A(x) \urcorner).$$

Die Quinierung wird wie die Substitution durch die Formel $Q(x, y) = S(x, x, y)$ stark repräsentiert.

$$B(x) := \exists y(A(y) \wedge Q(x, y)).$$

$$k = \ulcorner B(x) \urcorner.$$

Wir geben B *sich selbst* als Argument:

$$D := B(k) = \exists y(A(y) \wedge Q(k, y)).$$

$$D = B(\ulcorner B(x) \urcorner) = \exists y(A(y) \wedge y = quin(\ulcorner B(x) \urcorner) = \ulcorner B(\ulcorner B(x) \urcorner) \urcorner.$$

$$F \vdash \forall y(Q(k, y) \Leftrightarrow y = \ulcorner B(k) \urcorner).$$

Aus rot und blau, blau $\Leftrightarrow$ grün folgt rot und grün:

$$F \vdash D \Leftrightarrow \exists y(A(y)) \wedge y = \ulcorner B(k) \urcorner).$$

Wir fassen rot und grün zusammen:

$$F \vdash D \Leftrightarrow A(\ulcorner B(k) \urcorner).$$

Aus der Definition von D folgt:

$$F \vdash D \Leftrightarrow A(\ulcorner D \urcorner).$$

□

[18] ungebunden heisst, sie ist nicht durch einen Quantor quantifiziert
[19] Benannt nach dem Logiker Willard Van Orman Quine

3.3 Erster Unvollständigkeitssatz

Nun sind wir genügend vorbereitet, um uns den ersten Unvollständigkeitssatz vorzuknüpfen. Unser Ziel ist nach wie vor zu zeigen, dass es einen Satz G_F gibt, der im formalen System F nicht beweisbar ist und für dessen Verneinung $\neg G_F$ das gleiche gilt. Dazu greifen wir auf das Prädikat bew_F zurück. Jedoch negieren wir es, damit es bedeutet, dass es keine valide Beweisfolge y gibt, die die fehlerlose Ableitung von x beschreibt. Kurz gesagt: x ist nicht beweisbar in F.

$$A := \neg bew_F(\ulcorner x \urcorner) = \neg \exists y(val(\ulcorner y \urcorner, \ulcorner x \urcorner)).$$

Wir diagonalisieren A :

$$F \vdash G_F \Leftrightarrow \neg bew_F(\ulcorner G_F \urcorner).$$

Es gibt zwei Optionen:

I. $F \vdash G_F$
Aus der schwachen Repräsentierbarkeit von *bew* folgt:

$$F \vdash G_F \Leftrightarrow \neg bew_F(\ulcorner G_F \urcorner) \Leftrightarrow F \vdash \neg G_F.$$

Das ist wie zu sagen „Rauchen ist gesund und Rauchen ist nicht gesund.“, also Unsinn, wenn Konsistenz angenommen wird. Folglich ist F entweder inkonsistent oder $F \nvdash G_F$.

II. $F \vdash \neg G_F$
Wir nehmen an, F sei 1-konsistent[20].

$$F \vdash \neg G_F \Leftrightarrow F \vdash \forall x(\neg val(x, \ulcorner G_F \urcorner)).$$

F darf also $\exists x : \neg val(x, \ulcorner G_F \urcorner)$ nicht beweisen, weil das ein Widerspruch zur Annahme wäre. Aus der Äquivalenz von G_F folgt aber auch:

$$F \vdash \neg G_F \Leftrightarrow \neg\neg bew_F(\ulcorner G_F \urcorner) \Leftrightarrow bew_F(\ulcorner G_F \urcorner).$$

Schon wieder ein Widerspruch! Demnach muss die Annahme falsch sein. Wenn $F \vdash \neg G_F$ ist F 1-inkonsistent. Soll F 1-konsistent sein, gilt $F \nvdash G_F$

Theorem 5. *Gödels erster Unvollständigkeitssatz*

i. *F ist konsistent* $\Rightarrow F \nvdash G_F$

ii. *F ist 1-konsistent* $\Rightarrow F \nvdash \neg G_F$
Für jedes formale System F, das konsistent und mindestens so stark wie de Robinson Arithmetik Q ist, gibt es einen Satz G_F, der in F nicht entalten ist und für dessen Negation $\neg G_F$ das gleiche gilt und F demnach unvollständig ist.

[20] 1-konsistent, wenn nur sogennante $\sum_1^0$ Formeln (mit einem einzelnen Existenzquantor) verwendet werden, sonst muss die ω-Konsistenz angenommen werden. Ein System, das $A(0)$ bis $A(n)$ beweist, aber die allgemeine Aussage $\neg\forall x : A(x)$ beweist, heisst ω-unvollständig [15]

3.4 Ex contradictione sequitur quodlibet

[21] Bis jetzt hiess es immer, *wenn ein System konsistent ist, dann gilt . . .* . Was wäre denn an einem inkonsistenten System so schlimm? Nichts, es ist nur langweilig, denn alle Sätze sind in dem System wahr. Ein inkonsistentes System muss einen Widerspruch enthalten, zum Beispiel P *alle Zitronen sind gelb* und $\neg P$ *nicht alle Zitronen sind gelb*. Mit der ersten Annahme können wir beliebig viele Disjunktionen aufstellen. $P \vee Q$ Alle Zitronen sind gelb oder Gott existiert. Der Syllogismus aus der zweiten Annahme und Disjunktion folgert, dass Q wahr sein muss. Gott existiert, weil nicht alle Zitronen gelb sind. Das zeigt, dass wir völlig absurde Sätze beweisen können, weil für Q beliebiges eingestzt werden kann [8].

3.5 Bemerkungen

Häufig wird von *dem Gödelsatz* gesprochen. Doch die Brillianz des Diagonalisierungslemma ist, dass es sich immer wieder anwenden lässt. Gödel hat nicht *einen* unbeweisbaren Satz gefunden, sondern er hat eine Konstruktionsanleitung gegeben. Käme ein gerissener Kopf darauf, ein neues System F' zu definieren, dass wie F ist und G_F als Axiom hat, könnte man einen neuen Satz $G_{F'}$ finden, der F' widerum unvollständig macht. Selbstverständlich lässt sich das beliebig mal wiederholen.

3.5.1 Euklidische & Hyerbolische Geometrie

Der alte Grieche Euklid definierte die *Standardgeometrie* mit fünf Axiomen. Das Parallelenaxiom war lange umstritten und man forschte, ob es überhaupt nötig ist und nicht aus den anderen Axiomen ableitbar ist. Es hat sich herausgestellt, dass das nicht möglich ist. Wenn man die ersten vier Axiome und die Negierung des Parallelenaxioms nimmt, erhält man etwas sehr kurioses, nämlich die sogennante hyperbolische Geometrie. Die Abbildung 2 gibt einem eine Vorstellung von ihr. Alle Fische haben die Gleiche Fläche!

Abbildung 2: Circle Limit III, Escher 1959

3.5.2 Vollständige Systeme

Es wurde gezeigt, dass die Euklidische Geometrie mit dem Parallelenaxiom vollständig ist. Ebenfalls vollständig ist die klassische Aussagenlogik. Ein weiteres Beispiel ist die Presburger Arithmetik. Sie enthält nur die Addition und nicht die Multiplikation. Dafür ist sie aber entscheidbar [21]. Das heisst, es existiert ein Algorithmus, der für jeden Satz entscheiden kann, ob er in der Presburger Arithmetik wahr ist. Die Mengenlehre[22], die ein integraler Bestandteil höherer Theorien ist, ist aber schon zu komplex. Sie enthält die natürlichen Zahlen zwar nicht im klassischen Sinne, aber sie lassen sich in das

[21] Aus Widerspruch folgt Beliebiges
[22] Es gibt auch viele verschiedene Mengenlehren, hier ist die Zermelo-Frenkel Mengenlehre gemeint.

Vokabular der Mengenlehre übersetzten:

$$\{\varnothing\} = 1$$
$$\{\varnothing, \{\varnothing\}\} = 2$$
$$\{\varnothing, \{\varnothing, \{\varnothing\}\}\} = 3$$
$$\vdots$$
$$\{\varnothing, \overline{n-1}\} = n$$

3.6 Zweiter Unvollständigkeitssatz

„Jedes hinreichend mächtige konsistente formale System kann die eigene Konsistenz nicht beweisen."[4] Wenn ein System konsistent sein will, darf es keine einzige Inkonsistenz enthalten, da ansonsten alles wahr wäre (ex contradictione sequitur quodlibet). $S0 = 0$ darf zum Beispiel auf keinen Fall ein Satz sein, da $\neg S0 = 0$ bestimmt ein Satz ist und sie zusammen inkonsistent wären.

$$\bot := S0 = 0.$$

Das Symbol $\bot$ steht für Widerspruch. Das Prädikat $Kons(X)$ besagt, dass es im System X keinen Beweis für eine inkonsistente Formel gibt.

$$Kons(F) = \neg bew_F(\ulcorner \bot \urcorner).$$

Aus dem ersten Unvollständigkeitssatz folgt, dass ein konsistentes System sein Gödelsatz nicht enthalten kann.

$$Kons(F) \Rightarrow F \nvdash G_F.$$

Es wird angenommen, F könne seine eigene Konsistenz beweisen.

$$F \vdash Kons(F).$$

Via des Modus Ponens[23] fassen wir die oberen zwei Formeln zusammen:

$$F \vdash F \nvdash G_F.$$

Da der Gödel Satz soviel sagt wie *Ich kann nicht abgeleitet werden.* ist $F \nvdash G_F$ äquivalent zu G_F. Wir haben folglich:

$$F \vdash G_F.$$

Das steht im Konflikt mit dem ersten Unvollständigkeitssatz. Entweder ist das System inkonsistent oder die Annahme war falsch und die eigene Konsistenz kann nicht bewiesen werden. □

Theorem 6. *Gödels zweiter Unvollständigkeitssatz*
Ein konsistentes formales System F, das grundlegende Arithmetik durchführen kann, kann seine eigene Konsistenz nicht beweisen.

$$F \nvdash Kons(F).$$

An dieser Stelle könnte man völlig berechtigt einwenden, dass doch $Kons$ anders ausgedrückt werden könnte. Um diese Kritik zu befriedigen, wird der Beweis ziemlich technisch und mühsam, deshalb wird an diesem Punkt nicht genauer darauf eingegangen.

[23] Aus P und $P \to Q$ folgt Q

4 Ausblick

4.1 Beispiele unentscheidbarer Sätze

Eine Formel, die man mit Gödels Methode erlangen würde, um die Unvollständigkeit zu zeigen, wäre unglaublich lange und kompliziert. Ein menschlicher Mathemtiker könnte damit wohl wenig anfangen. Deshalb möchte ich zwei Beispiele von Sätzen geben, die in manchen Theorien unbeweisbar sind. Man kann sich vorstellen, dass auch die nicht gerade simpel sind und ihre eigene Maturitätsarbeit verdient hätten.

4.1.1 Satz von Goodstein

Der Satz von Goodstein besagt, dass alle Goodstein-Folgen mit 0 enden [5]. Um Goodstein-Folgen zu verstehen, müssen wir das Konzept des Stellenwertsystems hervorrufen: Die Zahl 42 im Dezimalsystem bedeutet 4 Zehner und 2 Einer.

$$42 = 4 \cdot 10^1 + 4 \cdot 10^0.$$

Stellt man 42 mit der Basis 2 dar gibt das:

$$42 = 32 + 8 + 2 = 1 \cdot 2^5 + 1 \cdot 2^3 + 1 \cdot 2^1.$$

Wenn man diese Dastellung iterativ auf die Exponenten anwendet, erhält man die sogennate *hereditary base representation*.

$$42 = 2^{(2^2+1)} + 2^{2+1} + 2.$$

Das erste Element einer Goodstein-Folge $G(x)$ ist immer x. Das zweite Element ist erhält man, indem man x in der hereditary representation bezüglich der Basis 2 schreibt und dann alle 2 mit 3 ersetzt und 1 subtrahiert. Das $i$te Element erhält man, indem man das vorherige Element bezüglich zur Basis i hereditary repräsentiert und alle i mit $i+1$ ersetzt und 1 subtrahiert. Hier als Beispiel die Goodstein-Folge $G(3)$:

Basis	Schritt	Wert	Neu
2	$2^1 + 1$	3	$2^1 + 1$
3	$3^1 + 1 - 1$	3	3^1
4	$4^1 - 1$	3	$3 \cdot 4^0$
5	$3 \cdot 5^0 - 1$	2	$2 \cdot 5^0$
6	$2 \cdot 6^0 - 1$	1	$1 \cdot 6^0$
7	$1 \cdot 7^0 - 1$	**0**	

Nun stellte Goodstein die Hypothese auf, dass alle Goodstein-Folgen mit 0 enden. Im Falle von $G(4)$ kann das schon sehr lange dauern[24]. Nimmt man die stärkere Mengenlehre dazu, kann beweisen, dass es in der Tat so ist. Innerhalb der Arithmetik ist der Satz aber unentscheidbar!

[24] Dazu habe ich auch einen kleinen Skript geschrieben: Siehe Appendix/beigelegter USB-Stick.

4.1.2 Kontinuumshypothese

Georg Cantor zeigte zuerst, dass die Natürlichen Zahlen $\mathbb{N}$ die gleiche Grössenordnung wie die rationalen Zahlen $\mathbb{Q}$[25] haben, indem er eine Bijektion $\mathbb{N} \mapsto \mathbb{Q}$ aufstellte, die alle rationalen Zahlen im *Zickzack* ablief. Später zeigte er, dass die reellen Zahlen $\mathbb{R}$ nicht abzählbar sind [1]. Jeder Versuch, eine endgültige Liste aller reellen Zahlen aufzuschreiben, ist zum Scheitern verurteilt: Man betrachte die rote diagonale im Bild. Ist die erste Zahle eine 0, nehmen wir eine 1, wenn sie eine 1 ist, eine 0. So kann man eine neue Zahl konstruieren, die verschieden ist von jeder bisherigen Zahl, weil sie an mindestens einer Stelle unterschielich sind. Das heisst es ist unmöglich eine Liste mit allen reellen Zahlen aufzuzählen, was gleichbedeutend dazu ist, dass $\mathbb{R}$ eine höhere Grössenordnung als $\mathbb{N}$ besitzt. Die ursprüngliche Frage lautete: Gibt es eine überabzählbare[26] Menge Zahlen, kleiner als $\mathbb{R}$ und grösser als $\mathbb{N}$. Das ist die grosse Kontinuumshypothese [2].

$$2^{\aleph_0} = \aleph_1.$$

Das spezielle daran ist, dass die Mengenlehre Widerspruchsfrei mit der Kontinuumshypothese aber auch mit der Verneinung davon ist. Bis anhin konnte man auch nicht mithilfe stärkeren Systemen die Kontinuumshypothese beweisen. Es gibt keinen Hinweis, ob man sie als wahr oder falsch auffassen soll und zweifelt sogar den Satz vom ausgeschlossenen Dritten[27] an.

$$\begin{array}{ll}
s_1 & = 0\,0\,0\,0\,0\,0\,0\,0\,0\,0\,0\ldots \\
s_2 & = 1\,1\,1\,1\,1\,1\,1\,1\,1\,1\,1\ldots \\
s_3 & = 0\,1\,0\,1\,0\,1\,0\,1\,0\,1\,0\ldots \\
s_4 & = 1\,0\,1\,0\,1\,0\,1\,0\,1\,0\,1\ldots \\
s_5 & = 1\,1\,0\,1\,0\,1\,1\,0\,1\,0\,1\ldots \\
s_6 & = 0\,0\,1\,1\,0\,1\,1\,0\,1\,1\,0\ldots \\
s_7 & = 1\,0\,0\,0\,1\,0\,0\,0\,1\,0\,0\ldots \\
s_8 & = 0\,0\,1\,1\,0\,0\,1\,1\,0\,0\,1\ldots \\
s_9 & = 1\,1\,0\,0\,1\,1\,0\,0\,1\,1\,0\ldots \\
s_{10} & = 1\,1\,0\,1\,1\,1\,0\,0\,1\,0\,1\ldots \\
s_{11} & = 1\,1\,0\,1\,0\,1\,0\,0\,1\,0\,0\ldots \\
\vdots & \quad \vdots\;\vdots\;\vdots\;\vdots\;\vdots\;\vdots\;\vdots\;\vdots\;\vdots\;\vdots\;\ddots \\
\hline
s & = 1\,0\,1\,1\,1\,0\,1\,0\,0\,1\,1\ldots
\end{array}$$

Abbildung 3: Cantors Diagonalargument [1]

4.2 Tarskis Undefinierbarkeit von Wahrheit

Eng verbunden mit den Unvollständigkeitssätzen ist die Undefinierbarkeit der Wahrheit.

Theorem 7. *Undefinierbarkeitssatz [17]*
Der Begriff der Wahrheit in einem konsistenten System kann nicht in dem System selbst definiert werden.

Beweis. Es gibt eine Funktion *wahr*, für die gilt:

$$F \vdash wahr(\ulcorner A \urcorner) \Leftrightarrow A.$$

Aus der Diagonlisierung von $\neg wahr$ folgt:

$$F \vdash L \Leftrightarrow \neg wahr(\ulcorner L \urcorner).$$

Man setze die Defintion von *wahr* ein:

$$F \vdash L \Leftrightarrow \neg L.$$

Aus diesem Widerspruch folgt, dass sich die Funktion *wahr* nicht in dem System selbst definieren lässt. □

[25] Nicht zu verwechseln mit der Robinson Arithmetik Q.
[26] Es existiert keine Bijektion zu $\mathbb{N}$.
[27] Das Grundprinzip, dass eine Aussage entweder wahr oder falsch ist.

5 Programm: Typographische formale Systeme

5.1 Einführung

Die Motivation für die Programmierung dieses Programmes lag darin, die abstrakten formalen Systeme besser zu verstehen. Dabei ist es nicht auf ein spezifisches System berschränkt, denn es lässt alle typographischen formale Systeme simulieren. Indem man seine eigenen Systeme erfinden und damit herumspielen kann, bekommt man hoffentlich eine bessere Vorstellung von der Theorie dahinter.

5.2 Reguläre Ausdrücke

Ein formales System heisst typographisch, wenn die Produktionsregeln nur auf Basis der Symbolketten operieren. *ALI* ist ein solches System, da die Regelanwendungen nur den vorigen Satz benötigen. Die Robison Arithmetik lässt sich leider nicht typographisch beschreiben, denn bei manchen Regeln muss man wissen, ob eine Variable ungebunden ist oder manchmal wird auch das Konzept der Satzheit anderer Formeln verwendet, das sich nicht syntaktisch darstellen lässt.
Um die Eigenschaften der Regeln einzufangen, kann man reguläre Ausdrücke benutzen. Reguläre Ausdrücke kurz RE[28] sind eine Art *Minisprache* zur Mustererkennung. Man definiert also eine Regel mit einem Muster und mit einer Substitutionskette. Der Algorithmus sucht einen Satz für alle Übereinstimmungen mit dem Muster ab und ersetzt dann alle Treffer mit der Substitutionskette. Im Fall der Regeln I und II von *ALI* ist das trivial:

```
[[rule]] # I
pattern = "^(.*I)$"
sub = "A\\1"
```

```
[[rule] # II
pattern = "LL"
sub = ""
```

```
[[rule] # III
pattern = "A"
sub = "LAL"
```

Für die erste Regel benötigen wir Features von RE. Beginnen wir von hinten und vorne. `^`und `$` sind Sonderzeichen und stehen für den Anfang beziehungsweise das Ende eines Wortes. Sie sind in diesem Fall eine Sicherheitsmassnahme, damit bestimmt das ganze Wort mit dem Muster übereinpasst. Die Klammern stellen eine Gruppierung dar, auf die später zugegriffen werden kann. Im Innern bleibt noch `.`, das für einen beliebiges Symbol steht und `*`, das für eine beliebig ofte Wiederholung des vorgestellten Objektes steht. Kurz gesagt, stimmt die Gruppe mit dem gesamten Eingabewort *matcht*. Im Substitutionsausdruck wird auf die Gruppe mit `\1` zurückgegriffen (1 weil die erste und einzige Gruppe) und fügt zusätzlich ein *A* am Anfang ein.
Zusätzlich zu `*` steht einem `+` und `?` zur Verfügung. `+` steht für eine bis unendlich viele Wiederholungen und `?` für keine oder eine Wiederholung. Diese Quantoren lassen sich auch auf Gruppen anwenden. Möchte man ein Sonderzeichen literal benutzen, muss man ihm ein \vorstellen. Da aber \in der benutzten Programmiersprache auch eine besonder Bedeutung hat, muss man sie mit einem weiteren Backslash umgehen. Für ein wörtliches muss man \\\\schreiben.

5.3 Konfigurationsformat

Am einfachsten ist es die Defintion eines formalen Systemes in einer Textdatei am besten im gleichen Ordner abzuspeichern. Man benenne die Datei mit einem aussagekräftigen Namen mit .toml als

[28] Engl.: regular expressions

Suffix. Man kann sich beliebig viele Axiome definieren, dafür schreibt man `[[axiom]]` und auf die nächste Zeile.

```
[[axiom]]
axiom = "Axiom Definition"
wildcards = {symbol = "RE"[, ..., ]}
```

Wenn man ein Axiomenschema definieren will, soll man auf einer weiteren Zeile `wildcards` als kommagetrennte Liste mit Definition. Im Beispiel kommt zweimal x vor, das als der gleiche RE spezifiziert ist. Dieses Axiomenschema prudziert folglich: `p-g-`, `-p-g--`, `--g-g---`, `...` Um eine Regel zu definieren schreibe man `[[rule]]` und spezifiziere dann

```
[[rule]]
pattern = "Musterausdruck"
sub = "Substitutionsausdruck"
```

5.4 Beispiel pg.toml

[12]

```
alphabet = ["-", "p", "g"]
[[axiom]]
axiom = "xp-gx-"
wildcards = {x = "-*"}
[[rule]]
pattern = "(-*)p(-*)g(-*)"
sub = "\\1p\\2-g\\3-"
```

Finden Sie heraus, welche Isomorphie dieses System darstellt?

5.5 Beispiel ali.toml

```
alphabet = ["A", "L", "I"]
[[axiom]]
axiom = "I"
wildcards = {}
[[rule]]
pattern = "^(.*I)$"
sub = "A\\1"
[[rule]]
pattern = "LL"
sub = ""
[[rule]]
pattern = "A"
sub = "LAL"
```

5.6 Installation

Diese Sektion setzt gewisse Grundkenntnisse voraus.
Die verwendete Programmiersprache ist Python 3. Das hat kein besonderer Grund, ausser dass ich mich darin am besten auskenne. Für die Ausführung benötigt man einen Python Interpreter[29]. Falls Sie Git[30] Versionskontrolle zur Verfügung haben, können Sie das Github repository `TODO` klonen oder als zip-file herunterladen. Öffnen Sie ein Terminal in dem Ordner und installieren Sie die Abhängigkeiten:

```
$ pip install -r requirements.txt
```

5.7 Befehle

Starten Sie eine Interaktive Python Sitzung:

```
$ python -i FormalSystem.py ali.toml
```

Ersetzen Sie `ali.toml` bei Bedarf mit einem Pfad zu Ihrer eigenen Definitionsdatei. Das System ist hinter der Variable f zugänglich. Prüfen Sie mit den folgenden Befehlen, ob alles in Ordnung ist:

```
f.axioms                    f.rules                    f.theorems
```

Die folgenden Funktionen sind implementiert:

- `f.step_production(r)`

  Wendet den Produktionsalgorithmus r Schritte mal an und fügt die Resultate in `f.theorems` ein.

```
>>> f.step_production(3)
Step 0: 1 theorems produced
AI <- Rule 0 from I
Step 1: 2 theorems produced
LALI <- Rule 2 from AI
AAI <- Rule 0 from AI
Step 2: 4 theorems produced
ALALI <- Rule 0 from LALI
LLALLI <- Rule 2 from LALI
AAAI <- Rule 0 from AAI
LALAI <- Rule 2 from AAI
```

- `f.produce(n)`

  Produziert und speichert n neue Theoreme.

```
>>> f.produce(8)
AALALI <- Rule 0 from ALALI
ALLALLI <- Rule 2 from ALALI
LALLALI <- Rule 2 from ALALI
ALLI <- Rule 1 from LLALLI
LLAI <- Rule 1 from LLALLI
LLLALLLI <- Rule 2 from LLALLI
AAAAI <- Rule 0 from AAAI
LALAAI <- Rule 2 from AAAI
```

[29] https://www.python.org/
[30] https://git-scm.com/

- `f.tree_produce(r)`

  Wie `step_production`, beginnt aber jedes mal von vorne und druckt die Theoreme in einer Baum-Hierarchie.

```
>>> f.tree_produce(3)
I
  AI
    LALI
      LLALLI
      ALALI
    AAI
      AAAI
      ALALI
      LALAI
# nach f.produce(100)
```

- `f.derive(theorem)`

  Versucht `theorem` zu beweisen. Benutzt nur schon produzierte Theoreme in `f.theorems`.

```
>>> f.derive("ALLALLI")
ALLALLI
LLALLI <- Rule 2 from LALI 0
LALI <- Rule 2 from AI 2
AI <- Rule 0 from I 2
I <- Rule None from AXIOM 0
AXIOM

>>> f.derive("ALI")
Exception: 'ALI' is not a theorem or not yet produced
```

- `f.proof(theorem)`

  Im Gegensatz zu `derive`, produziert diese Funktion so lange neue Theoreme, bis der Beweis möglich ist, oder läuft womöglich unendlich lange.

```
>>> f.proof("ALLALLALLAALALI")
[...]
19120 Theoreme produziert
[...]
LALLLI <- Rule 1 from LLLALLLI 0
LLLALLLI <- Rule 2 from LLALLI 1
LLALLI <- Rule 2 from LALI 2
LALI <- Rule 2 from AI 2
AI <- Rule 0 from I 2
I <- Rule None from AXIOM 0
AXIOM
```

- `f.is_proof(proof)`

  Testet ob die Liste `proof` ein korrekter Beweis darstellt.

```
>>> f.is_proof(["ALALI", "AAI", "AI", "I"], "LALLALI")
{'AI'}
{'AAI', 'LALI'}
{'LALAI', 'AAAI', 'ALALI'}
{'AALALI', 'ALLALLI', 'LALLALI'}
Derivation is correct

>>> f.is_proof(["LLALLI", "LALI", "AI", "I"], "ALI")
{'AI'}
{'AAI', 'LALI'}
{'ALALI', 'LLALLI'}
{'ALLI', 'ALLALLI', 'LLLALLLI', 'LLAI'}
Last step of derivation is incorrect
```

Schlusswort

In dieser Arbeit erhielt ich einen schönen Einblick in die Grundlagen der Mathematik. Bereits sind neue Interessen geweckt worden, so freue ich mich besonders auf Alan Turings Entscheidungsproblem, dass ich bestimmt im Studium antreffen werde. Des weiteren bin ich fasziniert von Computer gestützten Beweisprogrammen. Der erste Unvollständigkeitssatz wurde zum Beispiel formal von einem Computer verifiziert [14]. Wie das funktioniert interessiert mich brennend, aber in diese Arbeit hat es leider nicht mehr gereicht.

Danksagungen

> Hä ich han gmeint Mathe isch logisch.
>
> Kat

Ich bedanke mich bei ...

meiner Mutter Katrin, die mir beim arbeiten delikaten Chai Latte zubereitet hat.

meinem Vater Edi, der eingesteht nichts davon zu verstehen, aber sich die Mühe gemacht hat, alles durchzulesen und sich seine eigene Danksagung zu schreiben.

meinem Betreuer Thomas Foertsch, der Vertrauen in mich hatte und wertvolle Rückmeldungen gab.

Lukas für sein genaues Auge.

Leo für die scharfen Yakisoba.

Dominik für den Kaffee.

der Crew5000 für ihr Werk.

zahlreichen Anonymen, die im Internet ihr Wissen teilen.

A FormalSystem.py

```
import regex as re # externes regex Modul wird benötigt für sub.endpos support
import toml # Parser für Konfigurationsdateien
import reprlib # Stelle lange Listen benutzerfreundlich dar
import sys

from iteration import triangle_iteration # Importiere Dreiecksiteration-Modul


class Axiom:

    def __init__(self, axiom, wildcards=None):
        self.axiom = axiom
        self.wildcards = wildcards
        if self.is_axiom_schema:
            self.formattable_string = self._make_formattable_string()
            self.regex = self._make_regex()


    def _make_regex(self):
        """ returnt einen regex string, der verwendet werden kann, um
        herauszufinden, ob Sätze dem Muster Axiomschemata entsprechen, d.h.
        Axiome sind. """
        regex = self.axiom
        # starte mit der Wildcard, die zuerst vorkommt
        for i, (k, v) in enumerate(sorted(self.wildcards.items(), key=lambda q:
            self.axiom.find(q[0]))):
            regex = regex.replace(k, f"({v})", 1)
            # alle Instanzen der gleichen Wildcard müssen das gleiche matchen
            regex = regex.replace(k, f"\\{i+1}")
        return regex


    def _make_formattable_string(self):
        """ returnt ein formatierbarer string, der für die Produktion aus
        Axiomschemata verwendet werden kann """
        formattable_string = self.axiom
        self.chars = []
        self.starts = []
        self.stops = []
        self.names = []
        for wildcard, re in self.wildcards.items():
            char, specifier = re[0].strip(), re[1:].strip()
            formattable_string = formattable_string.replace(wildcard,"{" + wildcard + "}")
            self.names.append(wildcard)
```

```
            self.chars.append(char)
            # Die unterstützten Regex quantifizierer
            if specifier == "*":
                self.starts.append(0)
                self.stops.append(-1)
            elif specifier == "+":
                self.start.append(1)
                self.stops.append(-1)
            elif specifier == "?":
                self.start.append(0)
                self.stops.append(1)
            else:
                raise ValueError(f"Unsupported expression: {wildcard}: {value}")
        return formattable_string


    @property
    def is_axiom_schema(self):
        return len(self.wildcards) > 0


    def produce_axioms(self):
        """ iteriere mit der Dreiecksiteration über die Wildcards """
        assert self.is_axiom_schema
        for x in triangle_iteration(self.starts, self.stops):
            yield Theorem(self.formattable_string.format(
                **{name: c * i for c, i, name
                    in zip(self.chars, x, self.names)}), None, None)


    def __repr__(self):
        return f"Axiom({self.axiom})"



class Theorem:

    def __init__(self, theorem, parent, rule):
        self.theorem = theorem
        self.parent = parent
        self.rule = rule


    def __repr__(self):
        if self.parent:
            p = self.parent.theorem
```

```
        else:
            p = "AXIOM"
        return f"{self.theorem} <- Rule {self.rule} from {p}"


    def __hash__(self):
        """ wird benötig, damit Theoreme in der set Data-Struktur gespeichert
        werden können und mittels Hashtable O(1) Zugriff gestattet. """
        return hash(self.theorem)



class Rule:
    def __init__(self, pattern, sub):
        self.pattern = pattern
        self.sub = sub
        self.regex = re.compile(pattern)
        self.formatted_rule = f"{self.pattern} => {self.sub}"


    def __repr__(self):
        return f"Rule({self.formatted_rule})"



class FormalSystem:

    def __init__(self, axioms, rules):
        self.theorems = dict()
        self._generators = [] # Axiomgeneratoren
        self._active = [] # Aktiver Eimer
        self._add_axioms(axioms) # initialisiere die Axiome
        self._add_rules(rules) # initialisiere die Regeln
        self._axioms = axioms # speichere den Anfanszustand ab,
        self._rules = rules # damit zurückgesetzt werden kann


    def _reset(self, verbose=True):
        """ Setzte das System zurück: löscht alle produzierten Sätze (und Axiome)"""
        self.theorems = dict()
        self._generators = []
        self._active = []
        self._add_axioms(self._axioms)
        self._add_rules(self._rules)
        if verbose:
            print("Resetted")
```

```
def _add_axioms(self, axioms):
    self.axioms = []
    i = 0
    for ax in axioms:
        axiom = Axiom(ax["axiom"], ax["wildcards"])
        self.axioms.append(axiom)
        if not axiom.is_axiom_schema:
            self.theorems.update({ ax["axiom"]: Theorem(ax["axiom"], None, None) })
            self._active.append(Theorem(axiom.axiom, None, None))
        else:
            self._generators.append(axiom.produce_axioms())
            new_ax = next(self._generators[i]) # generiere erstes Axiom
            i += 1
            self.theorems.update({new_ax.theorem: new_ax})
            self._active.append(new_ax)


def _add_rules(self, rules):
    self.rules = list()
    for rule in rules:
        self.rules.append(Rule(rule["pattern"], rule["sub"]))


def __repr__(self):
    return f"FormalSystem({self.axioms, self.rules})"


@classmethod
def from_file(cls, path):
    parsed = toml.load(path)
    return cls(parsed["axiom"], parsed["rule"])


def apply_rules(self, sequence, as_theorems=True) -> set:
    """ Wendet alle Regeln möglichst viel mal an
    wenn as_theorems wird eine Menge von Theorem Objekten returnt, sonst
    strings"""
    news = set()
    for i, rule in enumerate(self.rules):
        if rule.regex.search(sequence):
            for _ in range(len(sequence)+1):
                res = rule.regex.sub(rule.sub, sequence, pos=_-1, count=1)
                if res != sequence:
                    if as_theorems:
```

```
                        news.add(Theorem(res, None, i))
                    else:
                        news.add(res)
        return news


    def _step_production(self, verbose=True):
        """ Generator, der pro Schritt wenn möglich ein neues Axiom produziert
        und alle Regeln auf alle aktiven Sätze anwendet"""
        nexts = list()
        for gen in self._generators:
            new_axiom = next(gen)
            if verbose: print("New axiom added")
            self._active.append(new_axiom)
            yield new_axiom
        if self._active:
            for a in self._active:
                seq = a.theorem
                news = self.apply_rules(seq)
                for new_theorem in news:
                    new_theorem.parent = a
                    if new_theorem.theorem not in self.theorems:
                        nexts.append(new_theorem)
                        yield new_theorem
            self._active = nexts


    def step_production(self, n=1, verbose=True):
        for i in range(n):
            if verbose:
                print(f"Step {i}: {len(self.theorems)} theorems produced")
            for new_theorem in self._step_production(verbose):
                self.theorems.update({new_theorem.theorem: new_theorem})
                if verbose:
                    print(new_theorem)


    def _produce(self, verbose=True):
        nexts = list()
        while self._active:
            for gen in self._generators:
                new_axiom = next(gen)
                if new_axiom.theorem not in self.theorems:
                    print("New axiom added")
                    self._active.append(new_axiom)
                    yield new_axiom
```

```
            for a in self._active:
                seq = a.theorem
                news = self.apply_rules(seq)
                for theorem in news:
                    if theorem.theorem not in self.theorems:
                        theorem.parent = a
                        nexts.append(theorem)
                        yield theorem
            self._active = nexts


    def produce(self, n, verbose=True):
        """ produziere n neue Theoreme """
        gen = self._produce(verbose)
        for _ in range(n):
            new_theorem = next(gen)
            self.theorems.update({new_theorem.theorem: new_theorem})
            if verbose:
                print(new_theorem)


    def proof(self, sentence, verbose=True):
        try:
            return self.derive(sentence, verbose)
        except:
            if verbose: print("Not yet produced")
            gen = self._produce()
            while True:
                for new_theorem in self._step_production(1):
                    self.theorems.update({new_theorem.theorem: new_theorem})
                    if new_theorem.theorem == sentence:
                        return self.derive(sentence, verbose)
                print(f"{len(self.theorems)} Theoreme produziert, aber {sentence} nicht gefunden :(


    def _derive(self, theorem, production, verbose=True):
        if theorem not in self.theorems:
            raise Exception(f"'{theorem}' is not a theorem or not yet produced")
        parent = self.theorems[theorem].parent
        rule = self.theorems[theorem].rule
        if verbose:
            if parent is None:
                print("AXIOM")
            else:
                print(parent, rule)
        if parent is None:
```

```
            return production
        production.append([parent, rule])
        parent = parent.theorem
        self._derive(parent, production)


    def derive(self, theorem, verbose=True):
        if verbose:
            print(f"Starting to derive from \n{theorem}")
        return self._derive(theorem, [theorem])


    def is_proof(self, derivation, formula, verbose=True):
        if verbose:
            print(f"Starting to check derivation {derivation} of {formula}")
        last = None
        for i, step in enumerate(reversed(derivation)):
            if last is not None:
                options = self.apply_rules(last, as_theorems=False)
                if verbose: print(options)
            if i == 0:
                if step not in [a.axiom for a in self.axioms]:
                    for ax in self.axioms:
                        if ax.is_axiom_schema:
                            if re.match(ax.regex, step):
                                print("Matched schema")
                                continue
                    if verbose:
                        print(f"Derivation {derivation} does not end with Axiom")
                    return False
            elif step not in options:
                if verbose:
                    print(f"Sentence {step} not producable out of {last}")
                return False
            last = step
        options = self.apply_rules(last, as_theorems=False)
        if verbose: print(options)
        if not formula in options:
            if verbose:
                print("Last step of derivation is incorrect")
            return False
        if verbose:
            print("Derivation is correct")
        return True
```

```
    def _tree_produce(self, theorem, depth, n, ref, verbose):
        if verbose:
            print("  "*depth+theorem)
        if depth >= n:
            return
        seen = set()
        for i in self.apply_rules(theorem):
            i = i.theorem
            if i in seen: continue
            seen.add(i)
            ref[i] = dict()
            ref = ref[i]
            self._tree_produce(i, depth+1,n, ref, verbose)


    def tree_produce(self, n, verbose=True):
        """ make production depth first by recursion depth (DFS)"""
        # only for finite axioms
        root = dict()
        ref = root
        for a in self.axioms:
            assert not a.is_axiom_schema
            a = a.axiom
            ref[a] = dict()
            ref = ref[a]
            self._tree_produce(a, 0, n, ref, verbose)
        return root


def tests():
    f = FormalSystem.from_file("ali.toml")
    f.step_production(3)
    input()
    f.is_proof(["ALALI", "AAI", "AI", "I"], "LALLALI")
    input()
    f.is_proof(["LLALLI", "LALI", "AI", "I"], "ALI")
    input()
    f.tree_produce(3, True)
    input()
    f.proof("I")
    input()
    f.produce(20)
    f.proof("ALLI")
    input()
    f.proof("ALLALLI")
    input()
```

```
    #f.proof("ALI")
    f = FormalSystem.from_file("pg.toml")
    f.step_production(3)
    input()
    f.is_proof(["---p-g----", "--p-g---", "-p-g--"], "---p--g-----")
    input()
    f.is_proof(["--p--g---"], "--p-g---")
    input()
    f.tree_produce(3, True)
    input()
    f.proof("--p-g---")
    input()
    f.proof("--p---g-----")


if __name__ == "__main__":
    if len(sys.argv) < 2:
        print("""Geben sie den Pfad zur Definitionsdatei als Argument an! Z.B.:
python -i FormalSystem.py ali.toml""")
        sys.exit()
    if len(sys.argv) > 2:
        print(""" Zu viele Argumente! Geben sie nur einen Pfad an.""")
        sys.exit()
    f = FormalSystem.from_file(sys.argv[1])
    print(f"""System {sys.argv[1]} erfolgreich geladen. Es ist zugänglich unter
            der variable f""")
```

B iteration.py

```
"""
Die Dreiecksiteration wird benützt, um neue Axiome zu produzieren, wenn das
Axiomschema über mehrere Wildcards verfügt. Wenn es nur eine Wildcard hat,
kann linear darüber iteriert werden:
z.B: -*
 , -, --, ---, ----, -----, ...
Wenn es 2 Wildcards hat und die gleiche Technik wie oben würde andgewendet, würde das wie
oben für die erste passieren, während die zweite unverändert bliebe, weil sie
erst drankäme, wenn die unendlich lange dauernde erste Iteration fertig wäre.
Deshalb brauchen wir die Dreiecksiteration, die gleichmässig ist.
für -*,-*
-,
 ,-
--,
-,-
 ,--
"""
```

```
def triangle_iteration(start, stop):
    """
    start: Liste mit n Ganzzahlen, die den Anfangszustand beschreiben.
    stop:  Liste mit n Ganzzahlen, die den Endzustand beschreiben.
    -1 in stop, wenn die Iteration unendlich weitergehen soll.
    Beispiel:
    triangle_iteration([2, 0, 1], [3, 2, 3])
    2 0 1
    2 1 1
    3 0 1
    2 0 2
    . . .
    2 2 3
    3 2 3
    """

    def increase_at_index(p, i):
        """ helper function """
        new = list(p)
        new[i] += 1
        if stop[i] > 0 and new[i] > stop[i]:
            return None
        return tuple(new)

    n = len(start)
    if n != len(stop):
        raise Exception("start und end müssen gleich lange sein")
    active = set([tuple(start)]) # start initialisieren
    yield tuple(start)
    while active:
        new = set()
        for p in active:
            for i in range(n):
                # für jedes element in active, erhöhe jedes Element
                # einmal um 1
                c = increase_at_index(p, i)
                if c: # wenn stop für das Element nicht erreicht worden ist
                    new.add(c)
                    yield c
        active = new


def pprint(nums):
    """
    Verbildlicht die abstrakten Zahlenfolgen.
```

```
    a b c
    aa ab ac ac bc cc ab bb bc
    aac abc acc aab abb abc abb bbb
    """
    from string import ascii_letters as letters
    n = len(nums)
    print("".join(l * x for l, x in zip(letters[:n], nums)))


def test():
    for i in triangle_iteration([0, 0, 0], [2, 2, 2]):
        pprint(i)

if __name__ == "__main__":
    test()
```

C numbering.py

```
from functools import reduce
from operator import mul
from string import ascii_lowercase

"""
`goedelisier`, findet die Gödelnummer zu einer Formel.
Die folgenden Symbole sind unterstützt:
0, S, +, *, =, (, ), ! (anstatt ¬), > (für ->), A (für Alle),
E (Existenzquantor), V (ODER), ^ (UND)
Für Variablen kann man alle Kleinbuchstaben benützen.
`inverse` ist die Umkehroperation. Sie gibt die Formel zu einer
Gödelzahl zurück.
Die Umkehroperation tauscht gelegentlich Variblennamen aus, da den Variablen
möglichst kleine Codenummern aufs neue zugeteilt werden, um die Gödelzahl
klein zu halten. Die Bedeutung bleibt aber erhalten.
"""

symbole = {"0": 1, "S" : 2, "+" : 3, "*" : 4, "=" : 5, "(" : 6, ")": 7,
        "!": 8, ">": 9, "A": 10, "E": 11}


class DefaultDict(dict):
    _counter = len(symbole)
    def __getitem__(self, index):
        try:
            return super().__getitem__(index)
        except KeyError:
            self._counter += 1
```

```
            self.__setitem__(index, self._counter)
            return self._counter

mapping = DefaultDict(symbole)
reversed_mapping = dict((v, k) for k, v in symbole.items())


def primes():
    """ Generator, der alle Primzahlen der Reihe nach generiert """
    def is_prime(x):
        """ testet ob x eine Primzahl ist """
        if x <= 1:
            return False
        for i in range(2, int(x**0.5 + 1)):
            if x % i == 0:
                return False
        return True
    n = 2
    while True:
        if is_prime(n):
            yield n
        n += 1


def goedelisier(formula):
    return reduce(mul, (prime**(mapping[char])
        for char, prime in zip(formula, primes())))


def inverse(n):
    formula = ""
    g = primes()
    p = next(g)
    try:
        while n > 1:
            i = 0
            while n % p == 0:
                i += 1
                n //= p
            p = next(g)
            if i > len(symbole):
                sym = ascii_lowercase[i-1-len(symbole)]
            else:
                sym = reversed_mapping[i]
            formula += sym
        return formula
```

```
    except:
        raise Exception("Es existiert keine Formel zu dieser Zahl")


def main():
    sso = "SSO"
    nulleins = "!(0=S0)"
    prim = "Ac(Ed:!Ea(Eb((c+Sd)=(SSa*SSb))))"
    input("Drücken Sie jeweils eine Taste, um fortzufahren")
    input(f"Die Gödelzahl zur Formel {sso}: ", )
    print(goedelisier(sso)); input()
    input(f"Die Formel hinter der Gödelzahl 180")
    print(inverse(180)); input()
    input(f"Die Gödelzahl zur Formel {nulleins}: ", )
    print(goedelisier(nulleins)); input()
    input(f"Die Formel hinter der Gödelzahl {goedelisier(nulleins)}")
    print(inverse(goedelisier(nulleins))); input()
    input(f"Die Gödelzahl zur Formel {prim} (Es gibt unendlich viele Primzahlen): ", )
    print(goedelisier(prim)); input()
    input(f"Die Formel hinter der Gödelzahl {goedelisier(prim)}: ", )
    print(inverse(goedelisier(prim)))
    print("Jetzt sind Sie dran! Benutzen Sie die Funktion goedelisier('S0+S0=SS') und inverse(720326


if __name__ == "__main__":
    main()
```

D genereate_chains.py

```
from itertools import combinations, product

def generate_chains(alphabet):
    """
    alphabet: Liste mit Symbolen
    returns: Generator mit allen möglichen Kombinationen
    Es wird alles generiert, was mit dem Alphabet möglich ist, das beinhaltet
    alle Sätze, Wohlgeformte Formeln, aber auch alle nicht Sätze und alle
    Schlechtgeformte Formeln.
    """
    i = 0
    while True:
        for chain in product(alphabet, repeat=i):
            yield "".join(chain)
        i += 1

if __name__ == "__main__":
```

```
    # Druckt die ersten 100 Ketten, die der Länge nach geordnet in ALI gäbe
    g = generate_chains(["A", "L", "I"])
    for i in range(100):
        print(next(g), end=" ")
```

E goodstein.py

```
from functools import lru_cache # memoization für bessere Performance


def goodstein(n):
    """
    Theorem: jede Goodstein-Folge terminiert mit 0
    Unbeweisbar in der Arithmetik, aber wahr in Mengenlehre
    """
    print(f"Goodstein-Folge für {n}")
    i = 2
    s = hereditary_base(n, i)
    print(f"Base {i}: {s} = {n}")
    i += 1
    while n:
        # ersetze k mit k + 1 und subtrahiere 1
        s = s.replace(str(i-1), str(i))
        n = int(eval(s)) - 1
        print(f"Base {i}: {s} - 1 = {n}")
        s = hereditary_base(n, i)
        i += 1


@lru_cache
def hereditary_base(n, k):
    """
    stellt n in Basis k dar sowohl die Exponenten rekursiv in Basis k
    hereditary_base(42, 2) = 2**5 + 2**3 + 2**1 = 2**(2**2 + 1) + 2**(2 + 1) + 2
    """
    q = -1
    while n >= k**q:
        q += 1
    string = ""
    while n >= 1:
        div, n = divmod(n, k**q)
        if div != 0: # Wenn der Koeffizient 0 ist, kann man den Schritt überspringen
            exponent = q
            if exponent > k: # Wenn der Exponent grösser ist als die Basis, wird er rekursiv zur Bas
                exponent = hereditary_base(exponent, k)
            if div != 1: # 1 als Koeffizient kann weggelassen werden
```

```
                string += str(div) + " * "
            string += f"{k}**({exponent})"
            if n >= 1: string += " + "
        q -= 1
    string = string.replace(f"{str(k)}**(0)", "1") # leserlicherer Output
    string = string.replace(f"{str(k)}**(1)", str(k)) # leserlicherer Output
    return string


if __name__ == "__main__":
    for i in range(1, 5):
        print("#" * 40)
        goodstein(i)
        if i == 4:
            input("Achtung, Goodstein(4) läuft sehr lange! Drücken Sie Ctrl+C, um es anzuhalten.")
        else:
            input("Beliebige Taste drücken, um fortzufahren")
```

Literatur

[1] Wikipedia contributors. *Cantor's diagonal argument*. Online; abgefragt 25-Dezember-2019. URL: https://en.wikipedia.org/wiki/Cantor%27s_diagonal_argument.

[2] Wikipedia contributors. *Continuum hypothesis*. Online; abgefragt 25-Dezember-2019. URL: https://en.wikipedia.org/wiki/Continuum_hypothesis.

[3] Wikipedia contributors. *Formales System*. Online; abgefragt 15-November-2019. URL: https://de.wikipedia.org/wiki/Formales_System.

[4] Wikipedia contributors. *Gödelscher Unvollständigkeitssatz*. Online; abgefragt 12-Dezember-2019. URL: https://de.wikipedia.org/wiki/G%C3%B6delscher_Unvollst%C3%A4ndigkeitssatz#Zweiter_Unvollst%C3%A4ndigkeitssatz.

[5] Wikipedia contributors. *Goodstein-Folge Probleme*. Online; abgefragt 25-Dezember-2019. URL: https://de.wikipedia.org/wiki/Goodstein-Folge.

[6] Wikipedia contributors. *Hilbertsche Probleme*. Online; abgefragt 17-Dezember-2019. URL: https://de.wikipedia.org/wiki/Hilbertsche_Probleme#Hilberts_zweites_Problem.

[7] Wikipedia contributors. *Primfaktorzerlegung*. Online; abgefragt 20-November-2019. URL: https://de.wikipedia.org/wiki/Primfaktorzerlegung#Fundamentalsatz_der_Arithmetik.

[8] Wikipedia contributors. *Principle of explosion*. Online; abgefragt 15-Dezember-2019. URL: https://en.wikipedia.org/wiki/Principle_of_explosion.

[9] Wikipedia contributors. *Rekursiv aufzählbare Menge*. Online; abgefragt 15-Dezember-2019. URL: https://de.wikipedia.org/wiki/Rekursiv_aufz%C3%A4hlbare_Menge.

[10] Wikipedia contributors. *Robinson Arithmetic*. Online; abgefragt 16-November-2019. URL: https://en.wikipedia.org/wiki/Robinson_arithmetic.

[11] Wikipedia contributors. *μ-Rekursion*. Online; abgefragt 15-Dezember-2019. URL: https://de.wikipedia.org/wiki/%CE%9C-Rekursion.

[12] Douglas R. Hofstadter. *Gödel, Escher, Bach*. 1979.

[13] Andrew David Irvine und Harry Deutsch. „Russell's Paradox". In: *The Stanford Encyclopedia of Philosophy*. Hrsg. von Edward N. Zalta. Winter 2016. Metaphysics Research Lab, Stanford University, 2016.

[14] Russel O'Connor. *Essential Incompleteness of Arithmetic Verified by Coq*. abgefragt 26-Dezember-2019. URL: http://r6.ca/Goedel/goedel1.html.

[15] „Omega-consistency". In: *Encyclopedia of Mathematics*. URL: URL : %20http://www.encyclopediaofmath.org/index.php?title=Omega-consistency&oldid=43594.

[16] Panu Raatikainen. „Gödel Numbering". In: *The Stanford Encyclopedia of Philosophy*. Hrsg. von Edward N. Zalta. Fall 2018. Metaphysics Research Lab, Stanford University, 2018.

[17] Panu Raatikainen. „Gödel's Incompleteness Theorems". In: *The Stanford Encyclopedia of Philosophy*. Hrsg. von Edward N. Zalta. Fall 2018. Metaphysics Research Lab, Stanford University, 2018.

[18] Panu Raatikainen. „The Diagonalization Lemma". In: *The Stanford Encyclopedia of Philosophy*. Hrsg. von Edward N. Zalta. Fall 2018. Metaphysics Research Lab, Stanford University, 2018.

[19] Marcus du Sautoy. Hrsg. von Brady Haran. 2017. URL: https://www.youtube.com/watch?v=O4ndIDcDSGc.

[20] Karl Heinz Wagner. *Grundbegriffe der Aussagenlogik*. 2019. URL: http://www.fb10.uni-bremen.de/khwagner/grundkurs2/kapitel3.aspx.

[21] Eric W. Weisstein. *Presburger Arithmetic*. From MathWorld–A Wolfram Web Resource; abgefragt 15-Dezember-2019. URL: http://mathworld.wolfram.com/PresburgerArithmetic.html.

[22] Richard Zach. „Hilbert's Program". In: *The Stanford Encyclopedia of Philosophy*. Hrsg. von Edward N. Zalta. Fall 2019. Metaphysics Research Lab, Stanford University, 2019.