17 PASCALchen macht Funktionen berechenbar

Die bisher betrachteten Konzepte haben eine eingeschränkte Berechnungskapazität. Endliche Automaten erkennen reguläre Sprachen und kontextfreie Grammatiken erzeugen kontextfreie Sprachen, wobei es aber mit Hilfe des jeweiligen Pumping-Lemmas möglich ist, auch algorithmische erkenn- oder erzeugbare Sprachen zu finden, die nicht regulär und nicht kontexfrei sind. Diese Beobachtung legt einige Fragen nahe:

Mit welchen Berechnungskonzepten lassen sich auch Sprachen algorithmisch behandeln, die nicht kontextfrei sind? Was ist eigentlich überhaupt "berechenbar"? Wie sehen Probleme aus, die nicht "berechenbar" sind? Gibt es das überhaupt?

Ziel dieses Abschnitts ist, den Begriff der berechenbaren Funktion zu präzisieren. Die Betonung liegt dabei auf einem exakten Vorgehen, um einen Teil der gestellten Fragen möglichst zuverlässig zu beantworten und den Spielraum für Einwände einzuengen. Der übliche Weg, zu berechenbaren Funktionen zu kommen, ist die Einführung einer Sprache zum Schreiben von Algorithmen (Syntax) und die Bereitstellung eines Kalküls, mit dem Algorithmen ausgeführt werden können (Semantik). Es gibt eine breite Palette an Möglichkeiten, das zu konkretisieren; sie reicht von maschinenorientierten bis zu problemorientierten Sprachen. Hier wird eine für Informatikerinnen und Informatiker naheliegende Wahl getroffen: die imperative Programmiersprache PASCALchen.

PASCALchen ist eine magere Sprache, die vage an PASCAL erinnert. PASCALchen verzichtet nahezu ganz auf Benutzungskomfort. Die wesentlichen Unterschiede zu üblichen imperativen Programmiersprachen sind, dass der einzige erlaubte Datentyp die natürlichen Zahlen sind und – vorläufig – Rekursion verboten ist. Das wesentliche Konstrukt von PASCALchen ist die Iteration auf der Basis der *while*-Schleife; deshalb werden die Programme der Sprache *while*-Programme genannt.

17.1 Die Sprache PASCALchen

1. Die Programme sind aus folgenden *Zeichen* aufgebaut:
   (a) *Variablennamen* sind Wörter, die mit X beginnen, worauf eine natürliche Zahl folgt.
   (b) *Operatorensymbole*, die zum Ändern der Variablenwerte dienen, sind *succ* für die Nachfolgerfunktion, *pred* für die Vorgängerfunktion und 0 für die Nullkonstante.
   (c) Als *Relationssymbol* zum Vergleich von Variablenwerten steht $\neq$ für die Ungleichheitsabfrage zur Verfügung.
   (d) *Programmsymbole* sind := für die Zuweisung, ; für die Reihung sowie *begin*, *end*, *while*, *do*.
   (e) Als *Hilfssymbole* zum Anwenden der Operatoren gibt es runde Klammern (und).
2. Programme werden nach folgender *Syntax* gebildet:
   (a) Ein *while-Programm* ist eine Reihungsanweisung.

   (b) Eine *Reihungsanweisung* (kurz: *Reihung*) hat die Form

$$\textit{begin}\ S_1; S_2, \ldots; S_m\ \textit{end}\,,$$

   wobei $S_1, \ldots, S_m$ für $m \geq 0$ Anweisungen sind. Der Fall $m = 0$ ist so gemeint, dass die Reihungsanweisung dann einfach die Form *begin end* hat.
   (c) Eine *Wiederholungsanweisung* (kurz: *Wiederholung*) hat die Form

$$\textit{while}\ X \neq Y\ \textit{do}\ S\,,$$

   wobei S eine Anweisung ist und X, Y zwei Variablen.
   (d) Eine *Zuweisungsanweisung* (kurz: *Zuweisung*) hat die Form

$$X := 0 \quad \text{oder} \quad X := \textit{succ}(Y) \quad \text{oder} \quad X := \textit{pred}(Y)\,,$$

   wobei X, Y zwei Variablen sind.
   (e) Eine *Anweisung* kann eine Reihung, eine Wiederholung oder eine Zuweisung sein. Mit den folgenden Bemerkungen werden dann auch noch Makroanweisungen als Anweisungen zugelassen.
3. Die Variablen müssen nicht deklariert werden, da sie alle vom Typ der natürlichen Zahlen $\mathbb{N}$ sind, für die es, korrespondierend zu den syntaktischen Größen, eine Konstante 0 gibt, die Nachfolger- und Vorgängerfunktionen $\textit{succ}, \textit{pred} \colon \mathbb{N} \to \mathbb{N}$ und einen Ungleichheitstest $\neq \colon \mathbb{N}^2 \to \{\mathsf{T}, \mathsf{F}\}$. Dabei ist *pred* definiert durch $\textit{pred}(0) = 0$ und $\textit{pred}(n+1) = n$ für alle $n \in \mathbb{N}$. Die anderen Funktionen arbeiten in bekannter Weise.

Bemerkungen

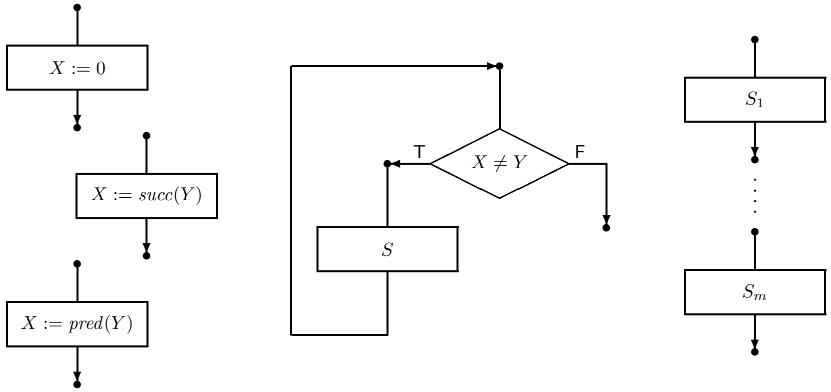
1. Dass PASCALchen so knapp zugeschnitten ist, hat einen pragmatischen und einen theoretischen – fast philosophischen – Grund.
   Es muss noch definiert werden, was *while*-Programme leisten. Das ist für wenige Konstrukte oft einfacher als für ein breites Spektrum. Andererseits ist das Ziel, das Berechenbare aufzuspüren. Von Vorgaben jedoch lässt sich nicht beweisen, dass sie Berechenbares darstellen, sondern es ist allenfalls plausibel und muß geglaubt werden. Je weniger vorausgesetzt wird, desto besser sind die Chancen, dass die Berechenbarkeit davon einsichtig ist und den Erfahrungen entspricht. Aus diesem Grunde sind insbesondere auch die natürlichen Zahlen als einziger Datentyp gewählt worden, weil Menschen seit Jahrtausenden mit ihnen rechnen. Anschaulich wird erlaubt, ein leeres Blatt Papier als 0 zu nehmen, für jedes *succ* einen Strich zu machen, für jedes *pred* einen Strich auszuradieren und von zwei Blättern festzustellen, ob sie gleich viele Striche enthalten oder nicht. Die Annahmen sind also prozesshaft und von einfacher Art. Ist eigentlich die Unterstellung der Berechenbarkeit bei Zuweisung, Wiederholung und Reihung genauso selbstverständlich?
2. So klein, wie es aussieht, ist PASCALchen gar nicht. Da jedes *while*-Programm selbst eine Anweisung ist, kann es in anderen Programmen weiterverwendet werden.

Gibt man *while*-Programmen Bezeichnungen und erlaubt, diese statt der Anweisungen weiterzuverwenden, lässt sich mit diesem Mechanismus von *Makroanweisungen* sogar sehr strukturiert programmieren. Viele vertraute Operationen, Programmkonstrukte und Abfragen erweisen sich als Makroanweisungen, d.h. sie lassen sich als *while*-Programme realisieren, und können deshalb wie in PASCAL und ähnlichen Sprachen verwendet werden. Dazu gehören die Zuweisungsformen $X := Y$, $X := n$ (Konstante), $X := Y + Z$, $X := Y - Z$, $X := Y \times Z$, $X := Y \operatorname{div} Z$, $X := Y \operatorname{mod} Z$, $X := 2^Y$, ...; die Wiederholungsanweisungen der Form *while* B *do* S, wobei B ein Boolescher Ausdruck ist, der aus Variablen, Konstanten, $=$, $\neq$, $<$, *and*, *or*, *not* aufgebaut ist, und die Kontrollstrukturen *if-then*, *if-then-else*, *repeat-until*.
Näheres dazu finden die Leserinnen und Leser im Abschnitt 2.2 des Buches (Kfoury, Moll, Arbib 1982). Dieses Buch ist auch sonst für das Thema Berechenbarkeit zu empfehlen.

3. In Bemerkung 1 wird begründet, dass das begrenzte Repertoire von PASCALchen die Gefahr mindert, mit den gewählten Beschreibungshilfsmitteln über das Berechenbare hinauszuschießen. In Bemerkung 2 wird erläutert, dass das einzige, was gegenüber Sprachen wie PASCAL wirklich fehlt, das Konzept der Rekursion ist. Damit ist auch die Gefahr klein, dass PASCALchen hinter den Möglichkeiten des Berechenbaren zurückbleibt. Denn eine Programmiersprache wie FORTRAN kennt auch keine Rekursion; trotzdem sind keine Klagen bekannt, dass damit bestimmte algorithmische Probleme nicht lösbar seien. Tatsächlich gilt, daß Rekursion auf Iteration zurückgeführt werden kann, wie in nahezu jedem Buch über Berechenbarkeit nachlesbar ist (z.B. Abschnitt 5.2 in (Kfoury, Moll, Arbib 1982)).
4. Um die Semantik von Programmen anzugeben, wird ausgenutzt, daß jede Anweisung durch ein Flussdiagramm wie in Abbildung 12 dargestellt werden kann. Dabei werden die drei Zuweisungen und der Test *Instruktionen* genannt. Die Definition ist rekursiv. Die mit $S, S_1, \ldots, S_m$ bezeichneten Kästen müssen solange durch eines der fünf Diagramme ersetzt werden, bis nur noch Instruktionen vorkommen. Jedes Flussdiagramm hat genau einen Eingang und einen Ausgang. Die fünf Diagramme sind genau nach den Syntaxregeln gebildet. Darüber hinaus lässt sich zeigen, dass jedes *while*-Programm genau ein Flussdiagramm besitzt.

17.2 Berechnungen von PASCALchen-Programmen

Mit Hilfe der Flussdiagramme soll nun geklärt werden, was *while*-Programme berechnen. Das Abarbeiten eines Programms mit bestimmten Anfangswerten für die Variablen entspricht einem Durchlauf durch das zugehörige Flussdiagramm. Wird dabei eine Zuweisung überquert, ändert sich der Wert der betroffenen Variablen entsprechend. Wird ein Test erreicht, bleiben alle Werte unverändert. Aber anhand der aktuellen Werte entscheidet sich, wie weitergegangen wird (in Richtung T oder F). Die jeweiligen Werte der Variablen während der Auswertung werden im Zustandsvektor festgehalten. Um dessen Definition zu vereinfachen, wird folgende Konvention getroffen.



(a) Zuweisungen (b) Wiederholung (c) Reihung

Abbildung 12: Flussdiagramme für PASCALchen-Anweisungen

Vereinbarung

Eine Anweisung (und damit auch ein *while*-Programm) wird *k-variabel* genannt, wenn höchstens die Variablen $X1, \ldots, Xk$ darin vorkommen.

Nutzt man die Reihenfolge der Variablen aus, können ihre Werte als eine Folge von k Zahlen $x_1, \ldots, x_k$ notiert werden, wobei x_i der Wert von Xi ist.

Definition (Berechnungszustand)
Ein *Berechnungszustand* eines k-variablen *while*-Programms ist ein k-dimensionaler Vektor über den natürlichen Zahlen

$$a = (x_1, \ldots, x_k) \in \mathbb{N}^k,$$

der auch *Zustand* oder *Zustandsvektor* genannt wird.

Gezielte Veränderungen der Berechnungszustände während eines Durchlaufs durch das Flussdiagramm ergeben eine Berechnung eines Programms.

Definition (Berechnung)
Eine *Berechnung* durch ein k-variables *while*-Programm ist eine (möglicherweise unendliche) Sequenz der Form

$$a_0 A_1 a_1 A_2 a_2 \cdots a_{i-1} A_i a_i A_{i+1} a_{i+1} \cdots,$$

wobei die a_i Berechnungszustände und die A_i Instruktionen mit folgenden Eigenschaften sind:

(a) Es gibt einen Weg durch das zugehörige Flussdiagramm, der beim Eingang beginnt und genau die Instruktionen $A_1, A_2, \ldots, A_i, \ldots$ in dieser Reihenfolge durchläuft.
(b) a_0 ist frei wählbar.
(c) Für alle $i \geq 1$ ergibt sich a_i aus a_{i-1} und A_i nach folgenden Regeln:
  – Ist A_i ein Test, dann ist $a_i = a_{i-1}$.
  – Ist A_i eine Zuweisung der Form $Xu := g(Xv)$, wobei $g(Xv)$ für $succ(Xv)$ oder $pred(Xv)$ oder 0 steht und $u, v \in \{1, \ldots k\}$; ist ferner $a_{i-1} = (x_1, \ldots, x_k)$, dann gilt:
$$a_i = (x_1, \ldots, x_{u-1}, g(x_v), x_{u+1}, \ldots, x_k).$$
  Ist A_i die letzte Instruktion der Berechnung, so erreicht man hinter A_i den Ausgang. Ansonsten ist A_{i+1} die nächste Instruktion hinter A_i.
(d) Ist A_i ein Test der Form $Xu \neq Xv$, $a_{i-1} = (x_1, \ldots, x_k)$ sowie $x_u \neq x_v$, dann ist A_{i+1} die erste Instruktion in der Anweisung der zugehörigen Wiederholung. Ist jedoch $x_u = x_v$, muss man dem F-Zweig folgen. Dann erreicht man den Ausgang, falls A_i die letzte Instruktion der Berechnung ist. Ansonsten ist A_{i+1} die nächste Instruktion nach dieser Wiederholung.
(e) Ist die Sequenz endlich, endet sie mit einem Berechnungszustand und hat deshalb die Form:
$$a_0 A_1 a_1 \cdots a_{n-1} A_n a_n \qquad (n \geq 0).$$
  Dabei ist A_n die letzte Instruktion vor dem Ausgang des Flußdiagramms. Ist A_n ein Test $Xu \neq Xv$, muss ferner $x_u = x_v$ gelten für $a_{n-1} = a_n = (x_1, \ldots, x_k)$.
(f) Im Falle von (e) wird n die *Länge* der Berechnung genannt.

Unmittelbar aus der Konstruktion ergibt sich das folgende Ergebnis.

Beobachtung 13
Zu jedem k-variablen *while*-Programm und jedem Berechnungszustand a_0 gibt es genau eine Berechnung mit a_0 als Anfang.

Bemerkungen

1. Für den Umgang mit Berechnungen sind einige Sprechweisen hilfreich:
   (a) Fängt eine Berechnung mit a_0 an, wird a_0 *Eingabe* genannt.
   (b) Für eine endliche Berechnung $a_0 A_1 a_1 \cdots A_n a_n$ sagt man, dass sie (in n Schritten) *terminiert*; a_n wird dann *Ausgabe* genannt.
   (c) Entsprechend *terminiert* eine unendliche Berechnung *nicht*; ihre Ausgabe ist *undefiniert*.
2. Die hier beschriebene Auswertung von *while*-Programmen ist ein typisches Beispiel für eine operationelle Semantik, wobei anhand des Programmtextes eine Eingabe schrittweise zur Ausgabe umgeformt wird. Da die Semantik dadurch entsteht, dass Berechnungszustände umgeformt werden, wird sie auch als Zustandstransformationssemantik bezeichnet.

17.3 Semantik von PASCALchen-Programmen

Die obige Beobachtung legt nahe, die Semantik eines Programms als partielle Funktion auf den natürlichen Zahlen zu definieren, denn Berechnungen sind pro Eingabe eindeutig, liefern jedoch nur bei Termination Ergebnisse. Aus technischen Gründen wird dabei von den verfügbaren k Variablenwerten in modifizierter Form Gebrauch gemacht. Während als Funktionswert nur der Wert der Variablen $X1$ betrachtet wird, wird die Funktion auf j Argumente angewendet. Ist $j < k$, bekommen die ersten j Variablen die Argumente als Eingangswerte, während die restlichen durch Nullen aufgefüllt werden. Ist $j \geq k$, fallen die eventuell überschüssigen Argumente bei der Berechnung weg. Durch diesen "Trick" wird die Zahl der Argumente unabhängig von der Zahl der Variablen.

Definition (Semantikfunktion)
Sei P ein k-variables *while*-Programm und $j \in \mathbb{N}$. Dann ist die j-stellige *Semantikfunktion* von P
$$SEM_P \colon \mathbb{N}^j \to \mathbb{N}$$
für die Argumente $(x_1, \ldots, x_j) \in \mathbb{N}^j$ nach folgenden Regeln definiert:

(a) Aus den Argumenten wird eine Eingabe $a \in \mathbb{N}^k$ hergestellt, wobei $a = (x_1, \ldots, x_k)$ ist für $j \geq k$ und $a = (x_1, \ldots, x_j, 0, \ldots, 0)$ mit $k - j$ Nullen für $j < k$.
(b) P wird mit Eingabe a berechnet.
(c) Terminiert die Berechnung mit der Ausgabe $(y_1, \ldots, y_k)$, so ist $SEM_P(x_1, \ldots, x_j) = y_1$.
(d) Terminiert sie nicht, ist $SEM_P(x_1, \ldots, x_j)$ undefiniert.

Bemerkungen

1. Beachte, dass die Semantikfunktion wegen (d) im allgemeinen nicht immer definiert ist. Sie ist genau dann eine totale Funktion, wenn jede Berechnung terminiert.
2. Jedes Programm kann wegen der Wahlfreiheit von j unendlich viele Funktionen berechnen, die sich allerdings nur wenig voneinander unterscheiden.
3. Soll das j betont werden, wird $SEM_P^{(j)}$ statt SEM_P geschrieben.
4. Ist $SEM_P(x_1, \ldots, x_j)$ undefiniert, wird im folgenden manchmal die Bezeichnung $SEM_P(x_1, \ldots, x_j) = \perp$ dafür verwendet.

17.4 Berechenbarkeit

Nach diesen Vorbereitungen ist es nun möglich, den zentralen Begriff der Berechenbarkeitstheorie einzuführen. Berechenbare Funktionen präzisieren durch Computer lösbare Aufgaben.

Definition (Berechenbare Funktion)
Eine partielle Funktion $f \colon \mathbb{N}^j \to \mathbb{N}$ heißt *berechenbar*, wenn ein *while*-Programm existiert

mit

$$f = SEM_P^{(j)}.$$

Mit Hilfe dieser exakten Version von Berechenbarkeit lässt sich nun auch diesogenannte Churchsche These für die Zwecke dieser Lehrveranstaltung formulieren. Der folgende Abschnitt zeigt, dass sie sehr bequem ist, auch wenn man ihr nicht unbedingt glaubt. Sie besagt, dass man mit PASCALchen alles berechnen kann, eas überhaupt möglich ist. Es muss allerdings darauf hingewiesen werden, dass die Berechenbarkeitstheorie und mathematische Logik bisher nur Belege für ihre Richtigkeit gefunden haben und dass sie auch den Erfahrungen der meisten Informatikerinnen und Informatiker entspricht.

Churchsche These
Jede partielle Funktion $f\colon \mathbb{N}^j \to \mathbb{N}$, die durch irgendeinen Mechanismus oder auf Grund irgendeiner Überlegung algorithmisch berechnet werden kann, ist bereits berechenbar (durch ein *while*-Programm).

18 Programme sind aufzählbar – doch ihr (Ver-)Halten macht Kummer

Zu den technologischen Durchbrüchen der Computertechnik in den 40er Jahren zählt das von Neumann entdeckte Prinzip der Programmspeicherung, durch das Programme jederzeit abrufbar und wie Daten bearbeitbar werden. Heutzutage ist allen Informatikerinnen und Informatikern angesichts von Editoren, Compilern u.ä. vertraut, dass Programme Programme verarbeiten können. Theoretisch wurde die Tatsache, dass Algorithmen Daten von Algorithmen sein können, noch einige Jahre früher von Gödel erkannt und hat in Logik und Berechenbarkeitstheorie zu fundamentalen Erkenntnissen geführt. Um einiges davon vorstellen zu können, ist noch eine Schwierigkeit zu überwinden: Der Algorithmusbegriff des vorigen Kapitels arbeitet auf natürlichen Zahlen; damit *while*-Programme Daten werden und umgekehrt, müssen natürliche Zahlen als Programme und Programme als natürliche Zahlen aufgefasst werden können. Zu diesem Zweck erhält jedes *while*-Programm einen Index. Außerdem wird ein algorithmisches Verfahren angegeben, wie aus einer natürlichen Zahl ein Programm entsteht, dessen Index sie ist. Auf diese Weise werden *while*-Programme "effektiv" aufgezählt.

18.1 Bestimmung des Indexes eines Programms

1. Nach Punkt 6.1.1 in Verbindung mit der Konvention in Abschnitt 6.2 besteht der Zeichensatz A von PASCALchen aus 22 Zeichen.
2. Für jedes Zeichen $a \in A$ wird nun eine eindeutige 6-Bit-Darstellung $code(a) = b_1 \cdots b_6$ mit $b_1 = 1$ und $b_2, \ldots, b_6 \in \{0,1\}$ ausgewählt. Es wird also eine injektive Abbildung $code\colon A \to \{0,1\}^*$ fixiert. Eine solche Wahl ist möglich, weil es 32 verschiedene 6-Bit-Muster mit 1 am Anfang gibt.

3. Die Abbildung *code* lässt sich auf A^* fortsetzen, indem jedes Zeichen eines Wortes von links nach rechts durch seine 6-Bit-Darstellung ersetzt wird. Das ergibt eine Abbildung $code^*\colon A^* \to \{0,1\}^*$, die definiert ist durch
   (i) $code^*(\lambda) = \lambda$ und
   (ii) $code^*(av) = code(a)code^*(v)$ für $a \in A$, $v \in A^*$.
4. Jedes *while*-Programm ist ein Wort über A und besitzt deshalb ein Bitmuster. Jedes Bitmuster lässt sich als Binärdarstellung einer natürlichen Zahl auffassen. Das erlaubt folgende Definition.
5. Der *Index* eines *while*-Programms P ist die natürliche Zahl, deren Binärdarstellung $code^*(P)$ ist.
6. Diese Verwandlung syntaktischer Objekte in Zahlen wird *Arithmetisierung* oder *Gödelnumerierung* genannt. Sie macht es prinzipiell möglich, mit Programmen zu rechnen.

Es ist äußerst wichtig, dass verschiedene Programme verschiedene Indizes erhalten. Das jedoch folgt unmittelbar aus folgendem Lemma.

Lemma 14 (Injektivität der Gödelnumerierung)
Die Gödelnumerierung $code^*$ ist injektiv.

Beweis.
Betrachte die Abbildung $decode\colon \{0,1\}^* \to A^*$, die definiert ist für alle $B \in \{0,1\}^*$ und $b_1, \ldots, b_6 \in \{0,1\}$ durch

(i) $decode(B) = \lambda$ für alle B kürzer als 6,

(ii) $decode(b_1 \cdots b_6 B) = \begin{cases} a\ decode(B) & \text{wenn } code(a) = b_1 \cdots b_6 \\ \lambda & \text{sonst.} \end{cases}$

Dann gilt $decode(code^*(w)) = w$ für alle $w \in A^*$, wie sich durch vollständige Induktion beweisen lässt.

IA: $decode(code^*(\lambda)) = decode(\lambda) = \lambda$.

IV: Die Aussage gelte für $v \in A^*$.

IS: $decode(code^*(av)) = decode(code(a)code^*(v)) = a\ decode(code^*(v)) = av$.

Daraus ergibt sich die behauptete Injektivität:
$code^*(u) = code^*(v)$ impliziert $decode(code^*(u)) = decode(code^*(v))$, also $u = v$. □

Umgekehrt kann jeder Zahl algorithmisch ein Programm zugeordnet werden, dessen Index sie ist.

18.2 Bestimmung des Programms eines Indexes

Betrachte zu jeder natürlichen Zahl n ihre Binärdarstellung $B(n)$, und wende darauf die Abbildung *decode* aus dem vorangegangenen Beweis an. Ist $decode(B(n))$ ein *while*-Programm, so wird es als von n aufgezählt betrachtet:

$$AUFZÄHLUNG(n) = decode(B(n)).$$

Andernfalls wird n ein Programm zugeordnet, das niemals hält:

$$
\begin{aligned}
AUFZ\ddot{A}HLUNG(n) = \mathit{begin} \\
& X1 := 0;\ X2 := 1; \\
& \mathit{while}\ X1 \neq X2\ \mathit{do}\ X1 := X1 \\
& \mathit{end}.
\end{aligned}
$$

Damit ist insgesamt eine Abbildung $AUFZ\ddot{A}HLUNG$ definiert, die jeder Zahl ein Programm zuordnet.

Bemerkung
Durch $AUFZ\ddot{A}HLUNG$ wird insbesondere der Index n eines Programms P auf P abgebildet.

Beweis.
Nach Definition des Indexes n eines *while*-Programms P gilt: $B(n) = \mathit{code}^*(P)$. Daraus folgt mit der Injektivität der Gödelnumerierung:

$$\mathit{decode}(B(n)) = \mathit{decode}(\mathit{code}^*(P)) = P.$$

Nach obiger Definition von $AUFZ\ddot{A}HLUNG$ ergibt sich wie behauptet:
$AUFZ\ddot{A}HLUNG(n) = \mathit{decode}(B(n)) = P$. □

Bemerkungen

1. Im folgenden wird eine natürliche Zahl n *Index* des *while*-Programms P genannt, wenn $AUFZ\ddot{A}HLUNG(n) = P$. Wegen der obigen Beobachtung umfasst diese Definition die bisherigen Indizes. Statt $AUFZ\ddot{A}HLUNG(n)$ wird oft kurz P_n geschrieben. P_n bezeichnet also das Programm mit dem Index n.
   Jede natürliche Zahl ist nun Index, so dass *while*-Programme mühelos Indizes manipulieren können. Von $AUFZ\ddot{A}HLUNG$ erfährt man dann, wie sich das auf die indizierten Programme auswirkt. Beachte, dass dadurch alle Programme erfasst werden, weil jedes Programm einen Index hat.
2. Da jedes Programm P für jede Argumentzahl j eine berechenbare Funktion $SEM_P^{(j)}$ bestimmt, kann jede Zahl auch als Index einer berechenbaren Funktion aufgefasst werden:
   $$SEM_n^{(j)} := SEM_{P_n}^{(j)} \text{ für alle } n \in \mathbb{N}.$$
   Statt $SEM_n^{(1)}$ wird kürzer SEM_n geschrieben, denn dieser Fall kommt am häufigsten vor.
   Beim "Aufzählen" der natürlichen Zahlen werden also gleichzeitig alle *while*-Programme und damit insbesondere alle berechenbaren Funktionen mit einem Argument "mitaufgezählt".

$$
\begin{array}{ccccccc}
0 & 1 & 2 & \ldots & n & \ldots & \mathbb{N} \\
\downarrow & \downarrow & \downarrow & & \downarrow & & \downarrow AUFZ\ddot{A}HLUNG \\
P_0 & P_1 & P_2 & \ldots & P_n & \ldots & \textit{while}\text{-Programme} \\
\downarrow & \downarrow & \downarrow & & \downarrow & & \downarrow SEM \\
SEM_0 & SEM_1 & SEM_2 & \ldots & SEM_n & \ldots & \text{berechenbare Funktionen}
\end{array}
$$

   Für jede andere Argumentzahl kann die Aufzählung entsprechend durchgeführt werden; das wird aber im folgenden nicht gebraucht.
3. Beachte, dass die Indizes nicht für das praktische Rechnen mit Programmen geeignet sind, weil sie sehr groß ausfallen. Ein Programm aus nur 10 Zeichen beispielsweise hat bereits einen kleinsten Index zwischen 2^{59} und 2^{60}.

18.3 Effektive Aufzählbarkeit und nicht berechenbare Funktionen

1. Man nennt eine Menge M von Objekten *abzählbar* oder *aufzählbar*, wenn es eine surjektive Abbildung $\mathit{num}\colon \mathbb{N} \to M$ gibt. M heißt *effektiv aufzählbar*, wenn diese Numerierung durch einen Algorithmus vorgenommen wird.
2. Da die Binärdarstellung einer Zahl berechnet werden kann, da die Werte von *decode* algorithmisch bestimmt sind, da es schließlich einen Algorithmus gibt, der entscheidet, ob ein Text über A ein *while*-Programm ist oder nicht, erweist sich $AUFZ\ddot{A}HLUNG$ als effektiv. Graphisch sei das ausgedrückt durch die Darstellung in Abbildung 13.

$$n \longrightarrow \boxed{AUFZ\ddot{A}HLUNG} \xrightarrow{P_n}$$

Abbildung 13: Effektivität von $AUFZ\ddot{A}HLUNG$

3. Dass die effektive Aufzählung etwas Besonderes ist, zeigt folgende Überlegung: Abzählbar ist insbesondere jede Teilmenge der natürlichen Zahlen, von denen es überabzählbar viele gibt. Effektiv aufzählbar sind jedoch nur abzählbar viele Teilmengen, weil es nach Abschnitt 3 nur abzählbar viele Algorithmen gibt. Die dabei verwendete Überabzählbarkeit der Teilmengen von $\mathbb{N}$ ergibt sich analog zur folgenden Überlegung.
   Die Konstruktionen in Abschnitt 3 zeigen, dass es nur abzählbar viele berechenbare Funktionen gibt. Mit einem bekannten Argument (dem Diagonalisierungsverfahren von Cantor, mit dem dieser gezeigt hat, dass es überabzählbar viele reelle Zahlen gibt) stellt sich heraus, dass die (partiellen) Funktionen auf $\mathbb{N}$ nicht abzählbar sind.

Es muss also Funktionen geben, die nicht berechenbar sind.

Theorem 15 (Existenz nicht-berechenbarer Funktionen)
Es gibt eine Funktion $f\colon \mathbb{N} \to \mathbb{N}$, die nicht berechenbar ist.

Beweis (durch Widerspruch)**.**
Angenommen, jede Funktion $f\colon \mathbb{N} \to \mathbb{N}$ wäre berechenbar. Dann gäbe es zu f einen Index i mit $f = \mathit{SEM}_i$. Die (totalen) Funktionen lassen sich dann folgendermaßen aufzählen: f_0 ist die erste Funktion in der Aufzählung $\mathit{SEM}_0, \mathit{SEM}_1, \mathit{SEM}_2, \ldots$. Sind $f_0, \ldots, f_k$ bereits bestimmt, dann ist f_{k+1} die nächste Funktion hinter f_k in $\mathit{SEM}_0, \mathit{SEM}_1, \mathit{SEM}_2, \ldots$. Betrachte dann die Funktion $\mathit{plus}\colon \mathbb{N} \to \mathbb{N}$, die definiert ist durch

$$\mathit{plus}(n) = f_n(n) + 1.$$

Fasst man die Argumente als Spaltennummern und die Indizes der Funktionen als Zeilennummern auf und trägt in die Felder (m, n) immer gerade $f_m(n)$ ein, dann entsteht *plus* dadurch, dass jede Zahl in der Diagonalen dieser unendlichen Matrix um 1 hochgezählt wird.

Nach Annahme existiert ein Index l mit $\mathit{plus} = f_l$. Das aber führt zu einem Widerspruch, denn *plus* erweist sich als verschieden zu f_l:

$$\mathit{plus}(l) = f_l(l) + 1 \neq f_l(l).$$

Da die Annahme falsch ist, muss die Behauptung richtig sein. □

18.4 Unlösbarkeit des Halteproblems

Die allgemeine Überlegung zur Nicht-Berechenbarkeit hat den Nachteil, dass sich eine ziemlich uninteressante Funktion als nicht-berechenbar erweist. Es ist nicht übermäßig wahrscheinlich, dass irgendjemand auf die Idee gekommen wäre, gerade diese Funktion zu berechnen. Anders verhält es sich mit dem folgenden Problem, bei dem versucht wird, eine äußerst wichtige Eigenschaft von Programmen zu ermitteln, nämlich ob bestimmte Berechnungen terminieren oder nicht. In dem hier betrachteten Fall des "Halteproblems" wird jedes Programm auf den eigenen Index angewendet, weshalb es in der Literatur auch häufig "Selbstanwendungsproblem" genannt wird. Es stellt sich heraus, dass es unmöglich ist, zu berechnen, ob ein Programm bei Eingabe seines Indexes hält oder nicht.

Theorem 16 (Unlösbarkeit des Halteproblems)
Die (totale) Funktion $\mathit{HALTEPROBLEM}\colon \mathbb{N} \to \mathbb{N}$, die definiert ist für alle $i \in \mathbb{N}$ durch

$$\mathit{HALTEPROBLEM}(i) = \begin{cases} 1 & \text{wenn } \mathit{SEM}_i(i) \text{ definiert ist} \\ 0 & \text{sonst} \end{cases}$$

ist nicht berechenbar.

Beweis (durch Widerspruch)**.**
Angenommen, die Funktion $\mathit{HALTEPROBLEM}$ wäre durch das *while*-Programm HALT berechenbar. Dann ist auch die partielle Funktion $\mathit{konfus}\colon \mathbb{N} \to \mathbb{N}$ mit

$$\mathit{konfus}(i) = \begin{cases} 1 & \text{für } \mathit{HALTEPROBLEM}(i) = 0 \\ \perp & \text{sonst} \end{cases}$$

berechenbar durch das Programm:

```
begin
  HALT;
  while X1 ≠ 0 do X1 := X1;
  X1 := 1
end
```

Es existiert also ein Index j mit $\mathit{konfus} = \mathit{SEM}_j$. Betrachte nun die Anwendung von SEM_j auf j.

1. Fall: $\mathit{SEM}_j(j)$ ist definiert.
Dann ist $\mathit{HALTEPROBLEM}(j) = 1$ und damit $\mathit{konfus}(j) = \mathit{SEM}_j(j)$ undefiniert.

2. Fall: $\mathit{SEM}_j(j)$ ist undefiniert.
Dann ist $\mathit{HALTEPROBLEM}(j) = 0$ und damit $\mathit{konfus}(j) = \mathit{SEM}_j(j) = 1$.

Mehr Fälle gibt es nicht, und $\mathit{SEM}_j(j)$ kann nicht gleichzeitig definiert und undefiniert sein. Also muss schon die Annahme falsch und damit die Behauptung des Theorems richtig sein. □

Wem es immer noch "exotisch" vorkommt, dass man Programme auf sich selbst bzw. ihre Indizes anwendet, den mag die Unlösbarkeit des allgemeinen Halteproblems mehr überzeugen, bei dem es darum geht, ob ein Programm für irgendeine Eingabe hält oder nicht. Das ist auch ein erstes Beispiel für das hilfreiche Reduktionsprinzip, bei dem die Nicht-Berechenbarkeit eines Problems dadurch gezeigt wird, dass das Problem auf eines zurückgeführt wird, dessen Nicht-Berechenbarkeit bereits bekannt ist.

Korollar 17 (Unlösbarkeit des allgemeinen Halteproblems)
Die Funktion $\mathit{HALTEPROBLEM2}\colon \mathbb{N} \times \mathbb{N} \to \mathbb{N}$ mit

$$\mathit{HALTEPROBLEM2}(i, j) = \begin{cases} 1 & \text{wenn } \mathit{SEM}_i(j) \text{ definiert ist} \\ 0 & \text{sonst} \end{cases}$$

ist nicht berechenbar.

Beweis (durch Widerspruch)**.**
Angenommen, $\mathit{HALTEPROBLEM2}$ wäre durch das *while*-Programm $\mathit{HALT2}$ berechenbar. Dann wäre auch $\mathit{HALTEPROBLEM}$ berechenbar durch das Programm

begin
  $X2 := X1;$
  $HALT2$
end

im Widerspruch zum obigen Theorem. □

Bemerkung
Für die Nicht-Berechenbarkeit von Problemen, die auch als Unlösbarkeit angesprochen wird, kann man häufig auch die Bezeichnung *Nichtentscheidbarkeit* finden, wenn das Problem eine ja/nein-Frage ist, d.h. wenn die zugehörige Funktion für jede Eingabe einen von zwei Werten (in der Regel 0 und 1) liefert.

19 Ein Programm rechnet für alle

Die Nicht-Berechenbarkeit des Halteproblems wird verursacht durch den Wunsch, auch erfahren zu wollen, wenn Programme nicht halten. Verzichtet man darauf und begnügt sich damit, die Werte von Programmberechnungen zu bekommen, soweit sie existieren, so erweist sich diese abgeschwächte Aufgabe als berechenbar.

19.1 PASCALchen-Interpreter als universelle Funktion

Um diese Behauptung einzusehen, muss ein Interpreter konstruiert werden, der bei Eingabe eines Programms P und der Argumente $(x_1, \ldots, x_j)$ der von P berechneten Funktion SEM_P den Wert $SEM_P(x_1, \ldots, x_j)$ liefert, vorausgesetzt P terminiert für diese Eingabe. Graphisch ist dies in Abbildung 14 dargestellt.

Abbildung 14: Ein Interpreter für *while*-Programme

Ein solcher Algorithmus $INTERPRETER_0$ ergibt sich gerade aus den im 6. Kapitel eingeführten Konzepten und Konstruktionen:

(1) Wandle das k-variable Programm P in sein Flussdiagramm um.
(2) Wandle $(x_1, \ldots, x_j)$ in eine Eingabe a um.
(3) Berechne P für Eingabe a.

(4) Gib bei Termination den Wert von $X1$ aus.

Zwei Klippen sind noch zu umschiffen, bevor dieser Vorgang sich als berechenbar herausstellen kann. Der Interpreter verarbeitet Programme, *while*-Programme und berechenbare Funktionen nur natürliche Zahlen. Diese Diskrepanz lässt sich beseitigen, wenn der Interpreter wie in Abbildung 15 mit der Aufzählung des 7. Kapitels gekoppelt wird, da die Aufzählung ja gerade dazu dient, Zahlen stellvertretend für Programme als Eingabe zu akzeptieren.

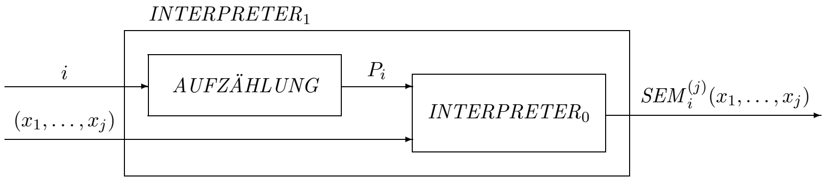


Abbildung 15: Eine Interpreter-Funktion

Diese neue Interpreterfunktion hat jetzt nur noch natürlichzahlige Argumente, und zwar genau ein Argument mehr als die Funktionen, die berechnet werden. Es gibt nach den bisherigen Überlegungen einen Algorithmus, der die Werte dieser Funktion ermittelt; aber gibt es auch ein *while*-Programm, das das leistet? Die bejahende Antwort ergibt sich aus der CHURCHSCHEN THESE. Das ist ein interessantes Beispiel für ihren vorteilhaften Gebrauch: Um Berechenbarkeit sicherzustellen, muss nicht immer ein *while*-Programm geschrieben werden; irgendein Algorithmus tut es auch. Beachte, dass es gerade auch für die Konstruktionen des 6. und 7. Kapitels reichlich mühsam wäre, simulierende *while*-Programme zu entwickeln.

Zusammengefasst hat sich folgende Erkenntnis ergeben:

Theorem 18 (universelle Funktion)
Für alle $j \in \mathbb{N}$ ist die partielle Funktion $interpret\colon \mathbb{N}^{j+1} \to \mathbb{N}$, die definiert ist durch

$$interpret(i, x_1, \ldots, x_j) = SEM_i^{(j)}(x_1, \ldots, x_j),$$

berechenbar.

Bemerkung
Da *interpret eine* Funktion ist, die *alle* berechenbaren Funktionen mit j Argumenten ausrechnet, wird sie *universelle Funktion* genannt. So verblüffend dieser Sachverhalt klingen mag, eigentlich passiert etwas Simples. Die Indizes der berechenbaren Funktionen werden zu einem zusätzlichen Argument. Und die Berechnung eines *while*-Programms ist ein algorithmischer Vorgang.

Zum Abschluss dieses Kapitels wird an fünf Beispielen die Grenzlinie zwischen Berechenbarem und Unberechenbarem kommentiert, die universelle Funktionen vom Halteproblem trennt. Vier der Beispiele nutzen dabei die Existenz universeller Funktionen in spezieller Form:

Es gibt ein k-variables *while*-Programm *INTERPRETER* mit

$$SEM_{INTERPRETER}(i,x) = SEM_i(x) \text{ für alle } i,x \in \mathbb{N}.$$

19.2 Beispiele

1. Abhängig von den Berechnungen des Interpreters lassen sich auch andere Aufgaben erledigen. Ein einfaches Beispiel dafür ist die partielle Funktion $halt_1 \colon \mathbb{N}^2 \to \mathbb{N}$ mit

$$halt_1(x,y) = \begin{cases} y & \text{wenn } SEM_x(x) \text{ definiert ist} \\ \perp & \text{sonst.} \end{cases}$$

Die Funktion $halt_1$ erweist sich als berechenbar durch das in Abbildung 16 dargestellte $(k+1)$-variable *while*-Programm $HALT_1$. Rechts sind die wichtigsten Ände-

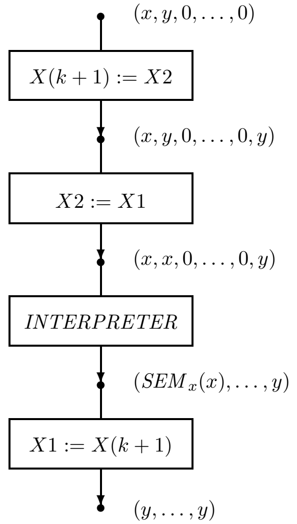


Abbildung 16: Das Programm $HALT_1$

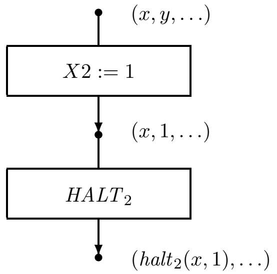


Abbildung 17: Reduktion des Halteproblems auf $halt_2$

rungen der Zustandsvektoren verzeichnet, wobei der vorletzte Zustand nur entsteht, wenn der Interpreter hält. Die Berechnungen zeigen, dass die partielle Funktion $halt_1$ berechnet wird.

2. Die Grenze zur Unberechenbarkeit wird wieder gerade überschritten, wenn über $halt_1$ hinaus auch Werte produziert werden sollen, wo der Interpreter nicht hält. Deshalb ist die Funktion $halt_2 \colon \mathbb{N}^2 \to \mathbb{N}$ nicht berechenbar, falls

$$halt_2(x,y) = \begin{cases} y & \text{wenn } SEM_x(x) \text{ definiert ist} \\ 0 & \text{sonst.} \end{cases}$$

Angenommen, $halt_2$ wird durch das *while*-Programm $HALT_2$ berechnet. Betrachte dann das in Abbildung 17 dargestellte Programm. Die rechts aufgezeichneten Berechnungen zeigen (für $y = 0$), dass dieses Programm P die einstellige Funktion $SEM_P \colon \mathbb{N} \to \mathbb{N}$ berechnet mit

$$SEM_P(x) = halt_2(x,1) = \begin{cases} 1 & \text{wenn } SEM_x(x) \text{ definiert ist} \\ 0 & \text{sonst.} \end{cases}$$

Das ist das Halteproblem, das gar nicht lösbar ist; also muss die Annahme falsch sein.

Am Unterschied von $halt_1$ und $halt_2$ wird noch einmal deutlich, wodurch Unberechenbarkeit verursacht wird. Solange der Interpreter läuft, kann nicht gesagt werden, ob die Berechnung nach weiteren Schritten abbricht oder nie zum Stehen kommt. Denn die Schrittzahl von Berechnungen kann jede endliche Schranke überschreiten. Ob der Wert 0 sein muss, lässt sich beim Berechnen nicht erfahren.

3. Die Verhältnisse können aber auch wesentlich undurchsichtiger sein. Die (partielle) Funktion $halt_3 \colon \mathbb{N}^2 \to \mathbb{N}$ mit

$$halt_3(x,y) = \begin{cases} \mu(y) & \text{wenn } SEM_x(x) \text{ definiert ist} \\ \nu(y) & \text{sonst,} \end{cases}$$

wobei $\mu, \nu \colon \mathbb{N} \to \mathbb{N}$ beide berechenbare Funktionen sind, hat große Ähnlichkeit mit $halt_2$. Und doch ist sie unter bestimmten Umständen berechenbar, wenn nämlich $\nu \leq \mu$ ist, d.h. wann immer $\nu(y)$ definiert ist, so ist auch $\mu(y)$ definiert, und in diesem Fall gilt: $\mu(y) = \nu(y)$.

Ein Beispiel für solche Funktionen wäre etwa gegeben durch

$$\mu(y) = \begin{cases} y^2 & \text{für } y < 200 \\ \perp & \text{sonst} \end{cases} \qquad \text{und} \qquad \nu(y) = \begin{cases} y^2 & \text{für } y < 100 \\ \perp & \text{sonst.} \end{cases}$$

Die Berechenbarkeit von $halt_3$ lässt sich folgendermaßen einsehen, wobei $\nu \leq \mu$ vorausgesetzt wird.

(a) Berechne gleichzeitig $\nu(y)$ und $SEM_x(x)$.
(b) Terminiert zuerst $SEM_x(x)$, tritt der erste Fall in Kraft. Beende deshalb die Berechnung von $\nu(y)$ und beginne die von $\mu(y)$.
(c) Terminiert zuerst die Berechnung von $\nu(y)$, dann ist nach Voraussetzung auch $\mu(y)$ definiert, und es gilt: $\mu(y) = \nu(y)$. Der Wert von $halt_3(x,y)$ ist damit gefunden, unabhängig davon, ob $SEM_x(x)$ terminiert.

(d) Terminieren beide nicht, so ist auch $halt_3(x, y)$ undefiniert.
Deshalb beschreiben die Punkte (a)-(c) einen Algorithmus, der $halt_3$ ausrechnet. Nach der Churchschen These ist $halt_3$ also berechenbar.

4. Eine sehr praktische Möglichkeit, die Schwierigkeiten mit dem Halteproblem zu umgehen, besteht darin, nach bestimmter "Zeit" die Berechnung gezielt abzubrechen. Im Beispiel wird das erreicht, indem die Rechenschritte gezählt werden und bei Überschreiten einer vorgegebenen Schranke aufgehört wird.
Die Funktion $halt_4 \colon \mathbb{N}^3 \to \mathbb{N}$ mit
$$halt_4(x, y, z) = \begin{cases} 1 & \text{wenn } P_x \text{ mit Eingabe } y \text{ nach spätestens } z \text{ Schritten hält} \\ 0 & \text{sonst} \end{cases}$$
ist berechenbar.
Denn ein Algorithmus für $halt_4$ kann wie folgt arbeiten:
(a) Berechne P_x für Eingabe y.
(b) Zähle die Schritte (Länge des Berechnungsweges im Flussdiagramm).
(c) Gib 1 aus, wenn der Zähler bei $count \leq z$ stehen bleibt.
(d) Sonst gib 0 aus.
Die Berechenbarkeit folgt wieder aus der Churchschen These.
Diese Vorgehensweise wird bei vielen Problemen der Informatik verwendet. Ihre "Philosophie" ist, eine Frage als unbeantwortet zu betrachten, wenn die Antwort zu lange ausbleibt.

5. Das letzte Beispiel ist nicht auf den Interpreter bezogen, sondern soll zeigen, dass auch im Bereich des Berechenbaren sehr undurchsichtige Situationen entstehen können. Es rankt sich um die Dezimalentwicklung von $\pi = 3,14159265\ldots$. Gesucht werden Sequenzen aufeinanderfolgender 5en:
$$3,14\ldots\underbrace{55\ldots5}_{x-\text{mal}}\ldots$$
Dabei spielen die vorausgehenden und nachfolgenden Ziffern keine Rolle (insbesondere dürfen auch sie 5en sein).
Aus diesen Elementen lässt sich eine Funktion $foggy \colon \mathbb{N} \to \mathbb{N}$ bilden mit
$$foggy(x) = \begin{cases} 1 & \text{wenn } \pi \text{ eine 5er-Sequenz der Länge } x \text{ enthält} \\ 0 & \text{sonst.} \end{cases}$$
Verglichen mit $halt_2$ könnte *foggy* für unberechenbar gehalten werden. Denn die Dezimalentwicklung von π lässt sich beliebig weit algorithmisch herstellen, wird jedoch nie fertig. So betrachtet, kann man nie wissen, ob der Wert 0 sein muss oder noch eine geeignete Sequenz beim weiteren Entwickeln der Dezimalstellen kommt.
Mit einer anderen Überlegung erweist sich *foggy* als berechenbar. Entweder gibt es in π 5er-Sequenzen jeder Länge. Dann ist $foggy(x) = 1$ für alle $x \in \mathbb{N}$ und als konstante Funktion berechenbar. Oder es existiert eine 5er-Sequenz mit maximaler Länge max. Dann gilt
$$foggy(x) = \begin{cases} 1 & \text{für } x \leq max \\ 0 & \text{sonst.} \end{cases}$$

Für jede natürliche Zahl $max \in \mathbb{N}$ lässt sich ein *while*-Programm schreiben, das *foggy* berechnet (*if* $x \leq max$ *then* 1 *else* 0).
Es gibt für *foggy* damit unendlich viele Möglichkeiten; in jedem Fall aber ist die Berechenbarkeit sichergestellt. Der einzige Haken: Niemand weiß, welcher Fall zutrifft.