Berechenbarkeit

Nicht-berechenbare Funktionen

Nach der Church-Turing-These kann alles, was berechenbar ist, mit einer Turing-Maschine oder einer While-Maschine programmiert werden. Das Programm muss natürlich nach endlich vielen Schritten anhalten.
Eine solche Turing-Maschine können wir auch mit einer beliebigen Programmiersprache simulieren. Wir können jeden Algorithmus z.B. in Java programmieren. Für jeden Algorithmus gibt es also einen Quelltext in Java. Einen Quelltext betrachten wir jetzt als ein einzelnes Wort bestehend aus den erlaubten und endlich vielen Zeichen für die Java-Syntax.
Alle diese Quelltexte ordnen wir nun nach der Größe beginnend mit den kleinsten Programmen. Z.B. gibt es dann viele Quelltexte mit der Länge 100. Es können aber nur endlich viele sein, da nur endlich viele verschiedene Buchstaben und Zeichen zur Verfügung stehen. Diese ordnen wir nun in alphabetischer Ordnung wie in einem Lexikon.

> **Wir können alle möglichen Algorithmen als Java-Quelltexte schreiben und ordnen, d.h. nummerieren.**
>
> **Die Menge der berechenbaren Funktionen ist abzählbar.**

Zur Vereinfachung betrachten wir nun nur noch Funktionen f, die aus einer natürlichen Zahl $n \in \mathbb{N}$ einen Funktionswert $f(n)$ berechnen. Sie werden alle in einer Tabelle dargestellt:

	1	2	3	4	5	...
f_1	$f_1(1)$	$f_1(2)$	$f_1(3)$	$f_1(4)$	$f_1(5)$	...
f_2	$f_2(1)$	$f_2(2)$	$f_2(3)$	$f_2(4)$	$f_2(5)$	...
f_3	$f_3(1)$	$f_3(2)$	$f_3(3)$	$f_3(4)$	$f_3(5)$	...
f_4	$f_4(1)$	$f_4(2)$	$f_4(3)$	$f_4(4)$	$f_4(5)$	...
...	...	...	...	...	...	

Diese Tabelle enthält alle Funktionen, und zwar in der ersten Zeile alle Funktionswerte der Funktion f_1, in der zweiten Zeile alle Funktionswerte der Funktion f_2 usw.
Wir konstruieren nun eine Funktion g auf die folgende Art und Weise:
$g(1) = f_1(1) + 1$. Damit unterscheidet sich g von der Funktion f_1.
$g(2) = f_2(2) + 1$. Damit unterscheidet sich g von der Funktion f_2.
$g(3) = f_3(3) + 1$. Damit unterscheidet sich g von der Funktion f_3.
Dies machen wir für alle Funktionen f_i. Damit unterscheidet sich g von allen Funktionen f_i. Sie ist aber offensichtlich berechenbar, also müsste sie in der Folge $f_1, f_2, f_3, \ldots$ schon vorkommen. Dies ist ein Widerspruch. Also muss die Annahme falsch sein, dass die Liste schon **alle** berechenbaren Funktionen enthält [1].
Was haben wir damit bewiesen?

[1] Das dargestellte Verfahren ist bekannt als *Cantorsches Diagonalisierungsverfahren.* Damit kann man z.B. zeigen, dass die Menge der reellen Zahlen zwischen 0 und 1 überabzählbar ist. Wie? Übung!

Es gibt arithmetische Funktionen, die nicht berechenbar sind.

Es gibt überabzählbar viele arithmetische Funktionen, von denen aber nur abzählbar viele berechenbar sind.

Das, was ein Computer kann, ist im Vergleich zu dem, was er nicht kann, vernachlässigbar.

Beispiele

Einige bekannte mathematische Probleme haben sich hartnäckig einer algorithmischen Lösung widersetzt. Kandidaten für Probleme, die nicht berechenbar sind, könnten die dargestellten klassischen mathematischen Probleme sein.

- Die Vermutung von Fermat (1601-1665): Es gibt keine natürlichen Zahlen $n \geq 3$, für die positive ganze Zahlen x, y und z mit $x^n + y^n = z^n$ existieren.

  Diese Vermutung konnte erst 1994 von dem genialen Mathematiker Andrew Wiles bestätigt werden. Der Beweis geht über ca. 100 Seiten[2].

- Die Vermutung von Goldbach (1690-1764): Jede gerade Zahl $n \geq 4$ lässt sich als Summe von zwei Primzahlen darstellen.

  Die Vermutung ist bis heute weder bewiesen noch widerlegt worden. Man konnte bis jetzt nur beweisen, dass jede gerade Zahl die Summe von nicht mehr als 800 000 Primzahlen ist.

- Vollkommene Zahlen: Eine natürliche Zahl heißt *vollkommen*, wenn ihre Teiler addiert die Zahl selbst ergeben.

  6 ist teilbar durch 1, 2 und 3 und $6 = 1 + 2 + 3$.

  28 ist teilbar durch 1, 2, 4, 7, 14 und $28 = 1 + 2 + 4 + 7 + 14$.

  In den letzten Jahrtausenden hat man nur 30 vollkommene Zahlen entdeckt. Die bisher größte vollkommene Zahl hat über 840 000 Stellen und ergibt sich aus dem Term

  $$2^{1398268} \cdot (2^{1398269} - 1).$$

  1. Behauptung: Alle vollkommenen Zahlen sind gerade.

  2. Behauptung: Es gibt unendlich viele vollkommene Zahlen.

  Beide Behauptungen konnten bis heute weder bewiesen noch widerlegt werden.

- Primzahlzwillinge sind zwei Primzahlen, die sich nur durch 2 unterscheiden. Bis heute ist nicht entschieden, ob es unendlich viele davon gibt.

Hätte Hilberts Entscheidungsproblem eine Lösung, so könnte man alle dargestellten Probleme einem Algorithmus als Eingabe anbieten und auf die Antwort „richtig“ oder „falsch“ endliche Zeit warten.

[2]Ein sehr schönes Buch dazu ist: Simon Singh, Fermats letzter Satz - die abenteuerliche Geschichte eines mathematischen Rätsels, dtv, ISBN 3-446-19313-8

Das Entscheidungsproblem

Wir betrachten wieder die abzählbare Menge aller Algorithmen in Java, die aus einer natürlichen Zahl irgendeine Zahl nach endlicher Zeit berechnen können. Die Programme werden wie oben nummeriert: $Programm_1$, $Programm_2$, $Programm_3$, usw. Z.B. gelte

$$\begin{aligned} Programm_{27}(n) &= n! \\ Programm_{28}(n) &= 2 \cdot n \\ Programm_{29}(n) &= n\text{-te Primzahl} \end{aligned}$$

Jedes Programm hat also eine Nummer, die man auch *Gödel-Nummer* nennt.
Nun konstruieren wir ein neues Programm $NeuProgramm$, das den Funktionswert $NeuProgramm(n)$ wie folgt berechnet: Suche das Programm mit der Nummer n in der Liste, „füttere“ dieses mit n, also seiner eigenen Nummer, nimm seinen Funktionswert $Programm_n(n)$ und addiere 1 dazu.
Also: $NeuProgramm(n) = Programm_n(n) + 1$
Beispiel: $NeuProgramm(28) = Programm_{28}(28) + 1 = 2 \cdot 28 + 1 = 57$
$NeuProgramm$ müsste eigentlich schon in der Liste sein, weil es eine berechnete Zahl liefert. Weil wir seine Nummer nicht kennen, nehmen wir x als seine eigene Nummer an. Dann gilt also:
$NeuProgramm = Programm_x$
Nun „füttern“ wir $NeuProgramm$ mit seiner eigenen Nummer und erhalten:
$NeuProgramm(x) = Programm_x(x) + 1$
$Programm_x(x)$ ist aber wieder $NeuProgramm(x)$
Also folgt zusammengefasst:
$NeuProgramm(x) = NeuProgramm(x) + 1$,
was offensichtlich ein Widerspruch ist. NeuProgramm liefert keine berechenbare Zahl. Also gehört es doch nicht zur Liste. Die Liste war demnach schon vollständig.
Dennoch existiert dieses Programm und liefert als außenstehendes, nicht zur Liste gehöriges Programm, vernünftige Ergebnisse. Also ist die Liste doch nicht vollständig.
Wir erhalten eine paradoxe Situation: **Es gibt keine Möglichkeit, zu entscheiden, ob das neue Programm zur Liste gehört oder nicht.**
Andere Formulierung:

> **Es gibt keinen Algorithmus, der zu jedem Text T, der die Berechnung einer Funktion $f : \mathrm{N} \to \mathrm{N}$ beschreibt, nach endlich vielen Schritten das Ergebnis liefert, ob die Funktion f für alle Werte ein Ergebnis liefert oder nicht.**

Verallgemeinerung:

> **Es gibt keinen Algorithmus, mit dem sich die Wahrheit einer mathematischen Aussage überprüfen lässt.**

Wir haben nicht bewiesen[3], dass wir einen solchen Algorithmus nicht kennen, sondern dass es niemals einen solchen geben wird!

[3] In der theoretischen Informatik wird dieser Beweis natürlich viel exakter, und zwar mit Hilfe der Turing-Maschine, geführt.

Das Halteproblem

Gibt es Probleme, die für Informatiker von größerem Interesse und auch nicht-berechenbar sind?
Ein allgemeines Problem bei Computerinstallationen auf der ganzen Welt ist, dass Programmierer beim Programmschreiben Fehler machen. Dadurch können Programme in eine Endlosschleife geraten und halten somit nicht an. Es wäre doch segensreich, wenn man durch ein Programm ein anderes Programm auf das Vorhandensein von Endlosschleifen untersuchen könnte, bevor das Programm ausgeführt wird. Dieses Problem ist als das *Halteproblem* bekannt geworden.
Man könnte annehmen, dass Softwarefirmen weltweit beträchtliche Anstrengungen unternähmen, um das Halteproblem zu lösen. Glücklicherweise hat sich ein Ergebnis der theoretischen Informatik mittlerweile herum gesprochen:

> **Das Halteproblem ist nicht berechenbar.**

Das Halteproblem für Java-Programme

> **Gibt es ein Java-Programm, mit dessen Hilfe man für jedes beliebige Java-Programm entscheiden kann, ob es mit jeder beliebigen Eingabe nach endlich vielen Schritten abbricht oder nicht?**

Jedes Java-Programm (Quelltext) kann als eine einzige Zeichenkette (ein String) aufgefasst werden und kann als solches dann auch als Eingabe eines anderen Java-Programmes verwendet werden (Erinnerung: Wir haben schon einmal ein Java-Programm geschrieben, das andere Programme und auch sich selbst im Quelltext auf dem Bildschirm ausgeben konnte).
Statt des oben angegebenen Halteproblems kann man auch folgendes Problem betrachten. Wir nennen es **Selbstanwendbarkeitsproblem.**

> **Gibt es ein Java-Programm, das von jedem beliebigen Java-Programm, das sich selbst als Eingabe hat, entscheidet, ob es nach endlich vielen Schritten anhält oder nicht?**

Offensichtlich ist dies ein Spezialfall des zuerst angegebenen Problems. Wenn es sich aber für den Spezialfall herausstellen sollte, dass es ein solches Programm nicht gibt, dann gibt es ein solches Programm auch nicht für den allgemeineren Fall.
Wir nennen ein Java-Programm *selbststoppend*, wenn es bei Anwendung auf sich selbst nach endlich vielen Schritten stoppt.
Wir nehmen nun an, es gäbe ein Java-Programm, das entscheidet, ob ein anderes Java-Programm selbststoppend ist oder nicht. Wir nennen es `Stopptester`. Es könnte etwa so aussehen:

```
class Stopptester
{
  static boolean esStoppt;

  public static void main(String args[])
  { // hier wird das eingegebene Programm
    // untersucht und je nach Ergebnis
    // die Variable esStoppt gesetzt.
```

```
    if (esStoppt)
         Out.println("Das Programm ist selbststoppend.");
    else Out.println("Das Programm ist nicht selbststoppend.");
  }
}
```

Wenn ein solches Programm `Stopptester` existiert, dann existiert auch das folgende Programm, dass man `Seltsam` nennen könnte:

```
class Seltsam
{
  static boolean esStoppt;

  public static void main(String args[])
  { // hier wird das eingegebene Programm
    // untersucht und je nach Ergebnis
    // die Variable esStoppt gesetzt.
    // Alles so wie in Stopptester.

    if (esStoppt)
         { Out.println("Das Programm ist selbststoppend.");
           while (1==1) { // Endlosschleife }
         }
    else Out.println("Das Programm ist nicht selbststoppend.");
  }
}
```

Der Unterschied zwischen den beiden Programmen besteht lediglich in der eingebauten Endlosschleife.
Ist das Programm `Seltsam` nun selbststoppend?

1. Wir nehmen zuerst an, `Seltsam` sei selbststoppend.

   Zuerst wird der Testteil ausgeführt und die boolesche Variable erhält den Wert `true`, weil `Seltsam` ja laut Annahme selbststoppend ist. Dann wird der Text „Das Programm ist selbststoppend.“ ausgegeben und das Programm läuft in eine Endlosschleife, was im Widerspruch zur Annahme ist.

2. Nun nehmen wir an, `Seltsam` sei nicht selbststoppend.

   Wieder wird der Testteil ausgeführt und die boolesche Variable erhält den Wert `false`, weil `Seltsam` ja laut Annahme nicht selbststoppend ist. Das führt aber dazu, dass der Text „Das Programm ist nicht selbststoppend.“ ausgegeben und das Programm anschließend stoppt, was wiederum im Widerspruch zur Annahme ist.

Das Programm `Seltsam` trägt seinen Namen zu recht, denn es kann nicht entschieden werden, ob es selbsstoppend ist oder nicht. Nach den Gesetzen der Logik kann es ein solches Programm nicht geben. Dann kann es aber auch das Programm `Stopptester` nicht geben, da `Seltsam` ja aus `Stopptester` konstruiert wurde.
Damit ist dann vollständig bewiesen, dass es kein Java-Programm gibt, mit dessen Hilfe man für jedes andere Java-Programm entscheiden kann, ob es mit jeder beliebigen Eingabe anhält oder nicht.
Der obige Beweis kann wie folgt zusammen gefasst werden:

1. Nehme an, es kann ein Programm `Stopptester` geschrieben werden.
2. Benutze es, um damit ein Programm `Seltsam` zu schreiben.
3. Zeige, dass das Programm `Seltsam` irgendeine undenkbare Eigenschaft (hier: es kann weder selbststoppend noch nicht-selbststoppend sein) hat.
4. Folgere, dass die Annahme in Schritt 1 falsch ist.

Es ist alles noch viel schlimmer!

Der Satz von Rice (1953) liefert ein noch viel vernichtenderes Ergebnis. Er besagt, dass wir nicht nur Programme nicht verifizieren können oder ihr Anhalteverhalten bestimmen, sondern dass wir im Grunde *gar nichts* über sie heraus finden können. Keine nichttriviale Eigenschaft der Berechenbarkeit kann algorithmisch entschieden werden. Um genauer zu sein: Angenommen, wir interessieren uns für eine Eigenschaft von Programmen, die

1. von einigen Programmen erfüllt wird und von anderen nicht, und die
2. syntaxunabhängig ist, d.h. eine Eigenschaft des zugrunde liegenden Algorithmus ist und nicht der speziellen Form, die er in einer Programmiersprache annimmt.

Zum Beispiel könnten wir wissen wollen, ob ein Programm länger als eine gewisse Zeit braucht, jemals die Antwort „ja“ liefert, ob es immer Zahlen ausgibt, ob es einem anderen Programm gleichwertig ist und so weiter.
Der Satz von Rice sagt uns, dass *keine* solche Eigenschaft von Programmen entschieden werden kann. Wir können es vergessen, über Programme etwas auf maschinelle Weise aussagen zu wollen. Praktisch *nichts*, was Berechenbarkeit betrifft, ist berechenbar!

Konsequenzen

Nun könnte man meinen, dass es vielleicht doch „gute“ Stopptester gibt, die viele Programme korrekt dahingehend untersuchen können, ob sie halten oder nicht. Für einfache Endlosschleifen mit offensichtlichen Fehlern ist dies sicherlich der Fall. Betrachtet man sich den Beweis, könnte man die Vermutung äußern, dass das Versagen erst durch die Anwendung des Programmes auf sich selbst auftritt.
Dies ist aber nicht der Fall. Wir könnten z.B. ein Java-Programm schreiben, dass die natürlichen Zahlen nach dem Auftreten von Primzahlzwillingen untersucht, die gefundenen ausgibt und stoppt, wenn es keine weiteren mehr findet. Natürlich müssten wir Algorithmen entwickeln, um beliebig große Primzahlen untersuchen zu können. Dazu bräuchten wir auch einen unendlich großen Speicher (wie das unendlich lange Band bei der Turing-Maschine) oder zumindest einen Administrator, der bei Anforderung des Programmes neue Speicherchips in den Computer oder in ein externes Speichermodul einsteckt. Ein Betriebssystem müsste die Speichererweiterung während des Programmlaufes unterstützen. Das ist aber alles prinzipiell kein Problem.
Wenn es ein Programm `Stopptester` gäbe, könnte dieses Programm das seit Jahrhunderten ungelöste Problem lösen, ob es unendlich viele Primzahlzwillinge gibt.
Viele mathematische Probleme wären auf einen Schlag durch den Einsatz von `Stopptester` gelöst und viele Mathematiker arbeitslos. In diesem Sinne zeigt sich die enge Verwandschaft

des Halteproblems mit dem Entscheidungsproblem. Vielen würde die Mathematik auch keinen Spass mehr machen, wenn alles durch eine Maschine gelöst werden könnte.
Tatsächlich kann man für viele andere Probleme zeigen, dass sie unlösbar sind, indem man sie auf das Halteproblem zurückführt.
Die Informatiker (und allen voran Alan Turing) haben gezeigt, dass es Probleme gibt, die von Menschen durch Nachdenken gelöst, die aber von einem Computer niemals gelöst werden können. Computer können den aktuellen Schachweltmeister besiegen, aber niemals auch nur andeutungsweise das lösen, was mit einem menschlichen Gehirn lösbar ist. Irgendwie muss ein Gehirn ganz anders als ein Computer arbeiten. Das „Irgendwie“ im letzten Satz ist die große Herausforderung der Zukunft, die uns Fragen zu der Art des menschlichen Denkens beantworten sollte.
Ein pessimistischer Ausblick wäre der, dass man an die Probleme der Computerprogramme beim Anwenden auf sich selbst denkt, und daraus den Schluss zieht, dass das Gehirn beim Nachdenken über sich selbst und seine Arbeitsweise ebenso in paradoxe Situationen gerät und wir deshalb nie verstehen werden, warum wir verstehen und wie wir verstehen. Fragen zum Computer und zu dem, was ein Computer kann und was ihn vom Menschen unterscheidet, sind damit auch tiefgehende philosophische Fragestellungen.
Mehr dazu in einem der nächsten Abschnitte: „Können Computer denken?“.