

## Ergänzung zu Kapitel 1

Wir wollen drei grundlegende Techniken, die sich durch diese und die im Sommersemester anschließende Vorlesung ziehen, an Beispielen informal vorstellen, nämlich Simulation, Reduktion und Diagonalisierung.

### Simulation

Eine Simulation beschreibt ein Verfahren, wie ein in einem Formalismus beschriebenes Programm durch ein in einem anderen Formalismus beschriebenes ersetzt werden kann, so dass das Ein/Ausgabe Verhalten unverändert bleibt. Wir werden in der Vorlesung viele Simulationen kennenlernen, etwa zwischen 1-Band- und Mehrband-Turingmaschinen, zwischen deterministischen und nichtdeterministischen Turingmaschinen oder endlichen Automaten, u.v.m.

#### Ein einfaches Beispiel:

Simulation eines Programms, in dem while- und for-Schleifen erlaubt sind, durch eins, das nur while-Schleifen erlaubt. Dazu muss man zeigen, wie eine for-Schleife

```
for  $i = 1$  to  $n$  do
  begin
    "Schleifenkörper";
  end;
```

durch eine while-Schleife ersetzt werden kann. Das ist natürlich in diesem Fall einfach.

```
 $i := 1$ 
while  $i \leq n$  do
  begin
    "Schleifenkörper";
     $i := i + 1$ ;
  end;
```

#### Bemerkung:

Programme, in denen keine Unterprogramme und Sprünge, und als einziger Schleifentyp for-Schleifen erlaubt sind, (man muss natürlich formalisieren, was das genau heißt; wir wollen hier nur intuitiv argumentieren) heißen *LOOP-Programme*, solche, in denen auch while-Schleifen (und eventuell auch Unterprogramme) erlaubt sind, heißen *WHILE-Programme*.

Unsere Simulation zeigt: Alles, was ein LOOP-Programm kann, kann auch ein WHILE-Programm, in dem sogar die while-Schleife der einzige Schleifentyp ist.

**Frage:**

Können WHILE-Programme Funktionen berechnen, die durch LOOP-Programme nicht berechnet werden können?

**Antwort:** Ja!

Folgende Funktion  $ack : \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{N}$ , die *Ackermann-Funktion*, kann nicht von einem LOOP-Programm berechnet werden, wohl aber von einem WHILE-Programm. Die erste Aussage werden wir hier nicht beweisen, die zweite sollte jeder von Ihnen beweisen können. Dazu müssen Sie nichts weiter tun, als etwa ein Java-Programm zur Berechnung von  $ack$  anzugeben. Java-Programme, aber auch C-, C++-, Pascal-, und Algol-Programme sind (äquivalent zu) WHILE-Programmen. Die Funktion  $ack : \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{N}$  ist definiert durch:

$$\begin{aligned} ack(0, m) &= m + 1 \\ ack(n, 0) &= ack(n - 1, 1), \text{ falls } n \geq 1 \\ ack(n, m) &= ack(n - 1, ack(n, m - 1)), \text{ falls } n, m \geq 1 \end{aligned}$$

## Reduktion

Der Begriff der Reduktion soll formal fassen, was es bedeutet, dass ein Problem  $A$  nicht schwerer als ein Problem  $B$  zu lösen ist. Dieses gilt sicherlich insbesondere dann, wenn man  $A$  sehr einfach lösen kann, falls man einen Algorithmus für Problem  $B$  hat.

**Beispiel:** Das *Cliquen Problem* (CLIQUE) und das *Independent Set Problem* (IS).

- Beim Cliquen Problem wird bei Eingabe eines ungerichteten Graphen  $G$  und einer Zahl  $k$  gefragt, ob  $G$  eine Clique, d.h. einen vollständigen Subgraphen mit  $k$  Knoten enthält.
- Beim Independent Set Problem liegt die gleiche Eingabe vor, aber es wird nun nach der Existenz eines Independent Set der Größe  $k$ , d.h. nach einem leeren induzierten Subgraph der Größe  $k$  gefragt.

Wir können CLIQUE sehr einfach lösen, falls wir einen Algorithmus für IS haben: Bei Eingabe  $(G, k)$  für Clique erzeuge aus  $G$  den Graphen  $\bar{G}$ , der genau die Kanten enthält, die in  $G$  fehlen. Nun starte den Algorithmus für IS mit  $(\bar{G}, k)$ . Überlegen Sie sich, dass IS bei Eingabe  $(\bar{G}, k)$  "ja" ausgibt, d.h. dass  $\bar{G}$  einen Independent Set der Größe  $k$  enthält, genau dann, wenn  $G$  eine Clique der Größe  $k$  enthält. Wir haben somit CLIQUE auf IS reduziert. In diesem Fall (das ist längst nicht immer so), lässt sich ebenso einfach IS auf CLIQUE reduzieren. Wir werden das Konzept der Reduktion später für den Nachweis der Unentscheidbarkeit und der sog. *NP*-Vollständigkeit von Problemen einsetzen.

## Diagonalisierung

Jeder bekannte Beweis dafür, dass eine gegebene Funktion nicht berechenbar ist (d.h. von keinem Algorithmus berechnet werden kann; Formalisierung folgt später) benutzt das Beweisprinzip der *Diagonalisierung*. Der Begriff führt zurück auf das Cantor'sche Diagonalisierungsverfahren, das Sie eventuell vom Beweis für die Überabzählbarkeit der reellen Zahlen kennen.

Haben Sie sich schon mal gewundert, warum ihr Java- oder C-Compiler Ihnen nie eine Warnung der Form: "Ihr Programm  $P$  gestartet mit der vorliegenden Eingabe  $x$  wird in eine Endlosschleife gehen!" liefert? Wir wollen uns im folgenden klar machen, dass es kein Programm geben kann, das eine solche Warnung immer korrekt aussprechen kann. Dazu benutzen wir einen speziellen "Beweis durch Widerspruch", basierend auf der Idee der Diagonalisierung.

### Annahme:

Es gibt ein Programm "HALTEN( $P, x$ )", das bei Eingabe eines Programms  $P$  und einer Eingabe  $x$  entscheidet, ob  $P$  gestartet mit  $x$  hält.

Dann gibt es auch folgendes Programm "WIDERSPRUCH( $P$ )":

WIDERSPRUCH( $P$ ):

Falls HALTEN( $P, P$ ) "ja" liefert, gehe in eine Endlosschleife, sonst halte an.

### Beobachtung:

Programm WIDERSPRUCH gestartet mit Eingabe  $P$  hält also genau dann an, wenn das Programm  $P$  mit Eingabe  $P$  nicht anhält. (Beachte: Wir fassen hier  $P$  auch als Eingabe auf. Das ist kein Problem, da ja  $P$  u.a. auch eine endliche Folge von Zeichen eines endlichen Alphabets ist. Beispielsweise bekommt ein Compiler (der ja selbst ein Programm ist), immer Programme als Eingabe.)

Wir wollen nun einen Widerspruch zur Annahme herleiten. Dazu schauen wir uns an, was das Programm WIDERSPRUCH tut, wenn wir es mit sich selbst als Eingabe starten. Die obige Beobachtung sagt uns: Programm WIDERSPRUCH gestartet mit Eingabe WIDERSPRUCH hält nicht, genau dann, wenn HALTEN(WIDERSPRUCH, WIDERSPRUCH) das Resultat "ja" liefert, also wenn WIDERSPRUCH gestartet mit WIDERSPRUCH hält.

Diese Aussage ist aber offensichtlich nicht wahr, da ein Programm gestartet mit einer festen Eingabe nicht gleichzeitig halten und nicht halten kann. Somit ist die Annahme falsch, d.h. ein Programm HALTEN, das für ein Programm  $P$  und eine Eingabe  $x$  entscheidet, ob  $P$  gestartet mit  $x$  anhält, existiert nicht.

Mit der Formalisierung von Rechenmodellen und dem Begriff der Berechenbarkeit werden wir obiges Argument später formalisieren: Das *Halteproblem* ist nicht entscheidbar.