Teil II

Berechenbarkeit

Übersicht

Berechenbare Funktionen

Eine (partielle) Funktion $f : \mathbb{N}^k \to \mathbb{N}$ ist berechenbar, wenn es einen Algorithmus gibt, der

- bei Eingabe von $(n_1, \ldots, n_k)$ genau dann terminiert, wenn $f(n_1, \ldots n_k)$ definiert ist,
- und dann den Wert $f(n_1, \ldots n_k)$ ausgibt.

Thema dieses zweiten Teils ist die Frage nach den prinzipiellehn Grenzen dessen, mit Computern möglich ist.
Positive Aussagen können dabei informell – durch Angabe eines Algorithmus – belegt werden. Um aber zu zeigen, dass ein gewisses Problem grundsätzlich nicht lösbar ist, braucht es ein formales Modell, über das Aussagen mit mathematischen Methoden bewiesen werden können. Wir werden zwei verschiedene Maschinenmodelle kennenlernen, *Registermaschinen* und *Turing-Maschinen*, und dann zeigen dass diese äquivalent sind.
Die Beschränkung auf natürliche Zahlen als Bereich ist eigentlich keine, da alle im Computer verarbeitetet Objekte als Bitstrings, und somit als natürliche Zahlen, codiert werden können.
Beispiele berechenbarer Funktionen sind die nirgends definierte Funktion, sowie die Funktion f mit $f(n) = \begin{cases} 0 & \text{falls die Goldbach-Vermutung wahr ist,} \\ 1 & \text{sonst.} \end{cases}$
Dieses Beispiel zeigt, dass nur die Existenz eines Algorithmus gefordert ist. Für die obige Funktioniert existiert ein Algorithmus, sie ist entweder die Konstante 0 oder die Konstante 1, auch wenn niemand weiß, welcher von beiden Algorithmen es ist. ch Dieses Beispiel it künstlich, aber es gibt im Bereich der Graphentheorie natürliche Probleme, für die bekannt ist, dass es einen Algorithmus geben muss, obwohl keiner bekannt ist. Dies folgt aus dem sog. Minorensatz für Graphen.

Die Registermaschine

Eine Registermaschine hat unendlich viele Register R_1, R_2, ..., die jeweils eine Zahl $n \in \mathbb{N}$ enthalten können.

Inhalt des Registers R_i wird über Variable x_i referenziert.

Die elementaren Anweisungen sind:

- Initialisierung: $x_i := 0$
- Kopieren: $x_i := x_j$
- Inkrement: $x_i := x_i + 1$
- Dekrement: $x_i := x_i - 1$

Die Abstraktion von einem realen Computer ist einerseits, dass in einer Speicherzelle eine nbeiebig groe Zahl gespeichert werden kann, andererseits dass es unendlich viele Speicherzellen gibt. Allerdings verwendet jedes konkrete Programm natürlich nur endlich viele Register.

Wir werden zwei Arten von Programmen, mit denen die Registermaschine programmiert werden kann, definieren, die WHILE-Programme und die GOTO-Programme.

WHILE-Programme

- Jede elementare Anweisung ist ein WHILE-Programm
- Sind P und Q WHILE-Programme, dann auch $P; Q$
- Ist P ein WHILE-Programm, dann auch

  `while` $x_i > 0$ `do` P `end`

$f : \mathbb{N}^k \to \mathbb{N}$ ist WHILE-berechenbar, wenn f durch ein WHILE-Programm berechnet wird, mit:

- am Beginn sind Eingaben $n_1, \ldots, n_k$ in $R_1, \ldots, R_k$, alle anderen Register haben Wert 0.
- wenn das Programm hält, wird die Ausgabe aus R_1 gelesen.

Die Semantik von WHILE-Programmen ist offensichtlich und wird hier nicht formal definiert werden. Sie assen sich z.B. als ein Fragment einer höheren Programmiersprache wie C oder Java verstehen.

Einige Beispielprogramme

Addition:

```
while x2 > 0 do
  x1 := x1 + 1 ;
  x2 := x2 − 1
end
```

Multiplikation:

```
while x2 > 0 do
  x3 := x3 + x1 ;
  x2 := x2 − 1
end;
x1 := x3 ;
```

Parität:

```
x2 := x1 − 1;
while x2 > 0 do
  x1 := x1 − 2 ;
  x2 := x2 − 2
end
```

Division durch 2^n:

```
while x2 > 0 do
  x1 := x1 − 1 ;
  x3 := 0 ;
  while x1 > 0 do
    x1 := x1 − 2 ;
    x3 := x3 + 1
  end;
  x1 := x3 ;
  x2 := x2 − 1
end
```

Mittels der Division durch Zweierpotenzen und der Paritätsfunktion hat man Zugriff auf jedes Bit in der Binärdarstellung einer Zahl und kann diese somit as Bitstring verstehen. Dies macht plausibel, dass auch gänzlich andere als arithmetische Operationen mit WHILE-Programmen programmiert werden könnten.

Fallunterscheidung

Die Fallunterscheidung:

$\texttt{if } x_i = 0 \texttt{ then } P \texttt{ end}$

kann programmiert werden als:

$z := x_i \texttt{ ; } y := 1 \texttt{ ;}$
$\texttt{while } z > 0 \texttt{ do } y := 0 \texttt{ ; } z := 0 \texttt{ end;}$
$\texttt{while } y > 0 \texttt{ do } P \texttt{ ; } y := 0 \texttt{ end}$

Ähnlich programmiert man Verzweigungen mit:

- Alternativen: $\texttt{if } x_i = 0 \texttt{ then } P \texttt{ else } Q \texttt{ end}$
- komplexeren Bedingungen: $\texttt{if } x_i = n \texttt{ then } P \texttt{ end}$

Hierbei sollen y und z irgendwelche Variablen x_k bezeichnen, die in dem Programm sonst nicht verwendet werden.

GOTO-Programme

Ein GOTO-Programm ist eine Folge von Paaren

$$M_1 : A_1 \; ; \; M_2 : A_2 \; ; \; \ldots \; ; \; M_n : A_n$$

aus Marken M_i und Anweisungen A_i.

Jede Anweisung A_i ist eine der folgenden:

- eine elementare Anweisung
- ein unbedinger Sprung: `goto` M_k
- ein bedingter Sprung: `if` $x_j = 0$ `goto` M_k
- Stopp-Anweisung: `halt`

GOTO-berechenbare Funktionen sind analog zu WHILE-berechenbaren definiert.

Auch bei den GOTO-Programmen sollte die Semantik klar sein:
Es wird mit dem ersten Befehl A_1 begonnen.

Ist der Befehl A_i keine Sprunganweisung, oder ein bedingter Sprung, bei dem die Bedingung falsch ist, dann wird mit dem nächsten Befehl A_{i+1} weitergemacht. Andernfalls wird bei A_k weitergemacht, wenn der Sprung zur Marke M_k geht.

Wir gehen davon aus, dass das Programm immer mit einer `halt`-Anweisung endet.

Simulation von WHILE- durch GOTO-Programme

Eine Schleife: `while` $x_i > 0$ `do` P `end`; ...
wird simuliert durch:

M_1: `if` $x_i = 0$ `goto` M_2
P
`goto` M_1
M_2: ...

Theorem

Jede WHILE-berechenbare Funktion ist auch GOTO-berechenbar.

Dies ist nichts anderes als die übliche Implementierung einer WHILE-Schleife durch bedingte Sprünge, wie sie auch bei der Übersetzung von höheren Programmiersprachen in Maschinencode verwendet wird.

Simulation von GOTO- durch WHILE-Programme

Simulation von $M_1 : A_1 \; ; \; M_2 : A_2 \; ; \; \ldots \; ; \; M_n : A_n$

$c := 1$
`while` $c > 0$ `do`
  `if` $c = 1$ `then` A'_1 `end;`
  `if` $c = 2$ `then` A'_2 `end;`
  $\ldots$
  `if` $c = n$ `then` A'_n `end`
`end`

Dabei ist A'_i definiert durch:

A_i	A'_i
elementar	$A_i \; ; \; c := c + 1$
`goto` M_k	$c := k$
`if` $x_j = 0$ `goto` M_k	`if` $x_j = 0$ `then` $c := k$ `else` $c := c + 1$
`halt`	$c := 0$

Variable c, der *Programmzähler*, ist wiederum eine Variable, die im Programm ansonsten nicht vorkommt.
In dem Programmsegment A'_i wird ggf. der A_i entsprechende Befehl ausgeführt und der Programmzähler c aktualisiert, wie in der Tabelle spezifiziert.

Kleenesche Normalform

Theorem

Jede WHILE-berechenbare Funktion ist auch GOTO-berechenbar.

Eine `while`-Schleife ist unwesentlich, wenn Sie in der Definition einer Verzweigung `if-then-else` vorkommt.

Korollar (Kleenesche Normalform)

Jede WHILE-berechenbare Funktion kann durch ein WHILE-Programm berechnet werden, das nur eine wesentliche `while`-Schleife enthält.

Der Begriff einer unwesentlichen WHILE-schleife könnte auch folgendermaßen präzisiert und syntaktischer gemacht werden: die Variable in der Schleifenbedingung wird am Ende des Scheifenrumpfes auf 0 gesetzt.

Dies ist bei unserer Implementierung des `if` der Fall. Es gilt dann, dass der Scheifenrumpf höchstens einmal durchaufen wird.

Eine weitere Folgerung aus der Simulation von GOTO- durch WHILE-Programme ist programmiertechnisch: sie zeigt, dass `goto`-Anweisungen redundant sind und alles, was überhaupt programmiert werden kann, auch strukturiert programmiert werden kann.