Theoretische Informatik
Kapitel 8 – LOOP-, WHILE- und GOTO-Berechenbarkeit

Sommersemester 2024

Dozentin: Mareike Mutz

im Wechsel mit

Prof. Dr. M. Leuschel

Prof. Dr. J. Rothe

Syntax von LOOP-Programmen

Definition

LOOP-Programme bestehen aus:

- Variablen: $x_0, x_1, x_2, x_3, \ldots$
- Konstanten: $0, 1, 2, 3, \ldots$
- Trennsymbolen: ; und $:=$
- Operationen: $+$ und $-$
- Befehlen: `LOOP`, `DO`, `END`

Syntax von LOOP-Programmen

Definition (Fortsetzung)

Wir definieren die Syntax von LOOP-Programmen induktiv wie folgt:

1. $x_i := x_j + c$, $x_i := x_j - c$ und $x_i := c$, für Konstanten $c \in \mathbb{N}$, sind LOOP-Programme.
2. Falls P_1 und P_2 LOOP-Programme sind, so ist auch $P_1 ; P_2$ ein LOOP-Programm.
3. Falls P ein LOOP-Programm ist, so ist auch

   $$\texttt{LOOP } x_i \texttt{ DO } P \texttt{ END}$$

   ein LOOP-Programm. Dabei darf in der LOOP-Anweisung die Wiederholvariable x_i nicht im Schleifenkörper P vorkommen.

Semantik von LOOP-Programmen

- Um die Semantik von LOOP-Programmen zu beschreiben, stellen wir uns die Werte der Variablen in Registern gespeichert vor.
- Abstraktion: Es gibt unendlich viele Register unendlicher Kapazität, d.h., beliebig große Zahlen können in einem Register gespeichert werden.

Semantik von LOOP-Programmen

Definition

In einem LOOP-Programm, das eine k-stellige Funktion f berechnen soll, gehen wir davon aus, dass dieses mit den Startwerten $n_1, \ldots, n_k \in \mathbb{N}$ in den Variablen $x_1, \ldots, x_k$ (und 0 in den restlichen) gestartet wird.

1. Die Zuweisungen $x_i := x_j + c$ und $x_i := c$ werden wie üblich interpretiert. In $x_i := x_j - c$ wird x_i auf 0 gesetzt, falls $c \geq x_j$ ist.
2. Das Programm $P_1; P_2$ wird so interpretiert, dass zuerst P_1 und dann P_2 ausgeführt wird.

Semantik von LOOP-Programmen

Definition (Fortsetzung)

3. Das Programm P in

$$\texttt{LOOP}\ x_i\ \texttt{DO}\ P\ \texttt{END}$$

wird so oft ausgeführt, wie der Wert der Variablen x_i zu Beginn angibt.

Da x_i nicht in P vorkommen darf, steht die Anzahl der Iterationen vor der ersten Ausführung der Schleife fest.

LOOP-Berechenbarkeit

Definition

Eine Funktion $f : \mathbb{N}^k \to \mathbb{N}$ heißt *LOOP-berechenbar*, falls es ein LOOP-Programm P gibt, das gestartet mit

$$n_1, \ldots, n_k$$

in den Variablen $x_1, \ldots, x_k$ (und 0 in den restlichen) stoppt mit dem Wert

$$f(n_1, \ldots, n_k)$$

in der Variablen x_0.

LOOP-Berechenbarkeit

Bemerkung:

- Die Anweisung

  $$\texttt{IF } x_1 = 0 \texttt{ THEN } P \texttt{ ELSE } P' \texttt{ END}$$

  lässt sich wie folgt mit dem obigen Befehlssatz ausdrücken:

  $$x_2 := 1;\, x_3 := 1;$$
  $$\texttt{LOOP } x_1 \texttt{ DO } x_2 := 0 \texttt{ END};$$
  $$\texttt{LOOP } x_2 \texttt{ DO } P;\, x_3 := 0 \texttt{ END};$$
  $$\texttt{LOOP } x_3 \texttt{ DO } P' \texttt{ END}$$

- Die Anweisung

  $$\texttt{IF } x_1 = c \texttt{ THEN } P \texttt{ ELSE } P' \texttt{ END}$$

  lässt sich ähnlich mit LOOP-Befehlen ausdrücken (s. Übungen).

LOOP-Berechenbarkeit

- Es ist nicht möglich, mit einem LOOP-Programm unendliche Schleifen zu programmieren. Das heißt, jedes LOOP-Programm stoppt nach endlich vielen Schritten.
  Somit sind LOOP-berechenbare Funktionen stets total.

- Da es nicht totale, aber intuitiv berechenbare Funktionen gibt, z.B. $f : \mathbb{N}^2 \to \mathbb{N}$ mit
  $$f(n_1, n_2) = n_1 \text{ div } n_2,$$
  kann die Menge der LOOP-berechenbaren Funktionen nicht die Menge aller intuitiv berechenbaren Funktionen umfassen.

- Es gibt sogar total definierte intuitiv berechenbare Funktionen, die nicht LOOP-berechenbar sind (z.B. die Ackermann-Funktion).

LOOP-Berechenbarkeit: Addition

Beispiel: Die Addition $f : \mathbb{N}^2 \to \mathbb{N}$ mit

$$f(n_1, n_2) = n_1 + n_2$$

ist LOOP-berechenbar.

Idee: Berechne $n_1 + n_2 = n_1 + \underbrace{1 + 1 + \ldots + 1}_{n_2}$.

$$x_0 := x_1 + 0;$$
$$\texttt{LOOP}\ x_2\ \texttt{DO}$$
$$x_0 := x_0 + 1$$
$$\texttt{END}$$

LOOP-Berechenbarkeit: Multiplikation

Beispiel: LOOP-berechenbar ist auch die Multiplikation $f : \mathbb{N}^2 \to \mathbb{N}$:

$$f(n_1, n_2) = n_1 \cdot n_2.$$

Idee: Berechne $n_1 \cdot n_2 = 0 + \underbrace{n_1 + n_1 + \ldots + n_1}_{n_2}$.

$$x_0 := 0;$$
$$\texttt{LOOP}\ x_2\ \texttt{DO}$$
$$x_0 := x_0 + x_1$$
$$\texttt{END}$$

Dabei wird $x_0 := x_0 + x_1$ durch ein Unterprogramm gemäß dem Beispiel oben für die Addition berechnet, d.h., hier entstehen zwei ineinander geschachtelte LOOP-Schleifen.

LOOP-Berechenbarkeit: Multiplikation

Beispiel: LOOP-berechenbar ist auch die Multiplikation $f : \mathbb{N}^2 \to \mathbb{N}$:

$$f(n_1, n_2) = n_1 \cdot n_2.$$

Komplette Lösung:

$x_0 := 0;$

LOOP x_2 DO

LOOP x_1 DO

$x_0 := x_0 + 1$

END END

Mit drei Schleifen ist auch die Potenzierung $n_1^{n_2}$ LOOP-berechenbar.

Syntax von WHILE-Programmen

Definition

WHILE-Programme sind wie folgt definiert:

1. Jede LOOP-Programm-Anweisung ist eine WHILE-Programm-Anweisung.
2. Falls P ein WHILE-Programm ist, so ist auch

$$\texttt{WHILE } x_i \neq 0 \texttt{ DO } P \texttt{ END}$$

   ein WHILE-Programm.
3. Falls P_1 und P_2 WHILE-Programme sind, so ist auch $P_1 ; P_2$ ein WHILE-Programm.

Semantik von WHILE-Programmen

Definition

1. Die Semantik von LOOP-Programmen wurde bereits definiert.

2. Das Programm P in einer WHILE-Anweisung

$$\texttt{WHILE } x_i \neq 0 \texttt{ DO } P \texttt{ END}$$

   wird wiederholt, solange der Wert von x_i ungleich 0 ist.

3. Das Programm $P_1 ; P_2$ wird so interpretiert, dass zuerst P_1 und dann P_2 ausgeführt wird.

WHILE-Berechenbarkeit

Definition

Eine Funktion $f : \mathbb{N}^k \to \mathbb{N}$ heißt *WHILE-berechenbar*, falls es ein WHILE-Programm P gibt, das gestartet mit

$$n_1, \dots, n_k$$

in den Variablen $x_1, \dots, x_k$ (und 0 in den restlichen) stoppt mit dem Wert

$$f(n_1, \dots, n_k)$$

in der Variablen x_0 – sofern $f(n_1, \dots, n_k)$ definiert ist. Andernfalls stoppt P nicht.

LOOP- versus WHILE-Berechenbarkeit

Folgerung: Jede LOOP-berechenbare Funktion $f : \mathbb{N}^k \to \mathbb{N}$ ist WHILE-berechenbar.

Bemerkung: In WHILE-Programmen kann man auf LOOP-Schleifen verzichten, denn offensichtlich kann man

$$\texttt{LOOP}\ x_i\ \texttt{DO}\ P\ \texttt{END}$$

simulieren durch:

$$x_j := x_i + 0;$$
$$\texttt{WHILE}\ x_j \neq 0\ \texttt{DO}\ x_j := x_j - 1;\ P\ \texttt{END}$$

WHILE-Berechenbarkeit: Potenzfunktion

Beispiel: Die Potenz $f : \mathbb{N} \rightarrow \mathbb{N}$ mit $f(n) = 2^n$ ist WHILE-berechenbar.

Idee: Berechne $2^n = 1 \cdot \underbrace{2 \cdot 2 \cdot \ldots \cdot 2}_{n}$.

$x_0 := 1;$

$\texttt{WHILE}\ x_1 \neq 0\ \texttt{DO}$

$x_0 := x_0 + x_0;$

$x_1 := x_1 - 1$

$\texttt{END}$

Wobei die Addition $x_0 + x_0$ hier wieder über das entsprechende LOOP-Programm definiert ist.

LOOP- versus WHILE-Berechenbarkeit

Beispiel: Das WHILE-Programm

$$x_3 := x_1 - 4;$$
$$\texttt{WHILE}\ x_3 \neq 0\ \texttt{DO}\ x_1 := x_1 + 1\ \texttt{END};$$
$$\texttt{LOOP}\ x_1\ \texttt{DO}\ x_0 := x_0 + 1\ \texttt{END};$$
$$\texttt{LOOP}\ x_2\ \texttt{DO}\ x_0 := x_0 + 1\ \texttt{END}$$

berechnet die Funktion $f : \mathbb{N}^2 \to \mathbb{N}$ mit

$$f(n_1, n_2) = \begin{cases} n_1 + n_2 & \text{falls } n_1 \leq 4 \\ \textit{undefiniert} & \text{sonst.} \end{cases}$$

f ist aber nicht LOOP-berechenbar, da f nicht total ist. Es gibt also WHILE-berechenbare Funktionen, die nicht LOOP-berechenbar sind.

WHILE- versus Turing-Berechenbarkeit

Theorem

Jedes WHILE-Programm kann durch eine Turingmaschine simuliert werden, d.h., jede WHILE-berechenbare Funktion ist auch Turing-berechenbar. **ohne Beweis**

Beweisidee:
k-Band-TM, für jede Variable des WHILE-Programms ein Band. Die einzelnen Operationen des WHILE-Programms in δ codieren.

Syntax von GOTO-Programmen

Definition

GOTO-Programme bestehen aus Folgen von markierten Anweisungen:

$$M_1 : A_1;\ M_2 : A_2;\ \ldots;\ M_m : A_m;$$

Anweisungen A_i dürfen dabei sein:

- Zuweisung: $x_i := x_j + c$, $x_i := x_j - c$ und $x_i := c$, für Konstanten $c \in \mathbb{N}$
- unbedingter Sprung: $\texttt{GOTO}\ M_i$
- bedingter Sprung: $\texttt{IF}\ x_i = c\ \texttt{THEN}\ \texttt{GOTO}\ M_j$
- Abbruchanweisung: $\texttt{HALT}$

Semantik von GOTO-Programmen

Definition

- Mit der `GOTO` Anweisung springt man zu der Anweisung mit der angegebenen Marke.
- Die `HALT` Anweisung beendet ein GOTO Programm, d.h., die letzte Anweisung sollte entweder `GOTO` oder `HALT` sein.

Bemerkung: GOTO-Programme können auch unendliche Schleifen enthalten, z.B.:

$$M_1 : \texttt{GOTO}\ M_1;$$

GOTO-Berechenbarkeit

Definition (GOTO-Berechenbarkeit)

Eine Funktion $f : \mathbb{N}^k \to \mathbb{N}$ heißt *GOTO-berechenbar*, falls es ein GOTO-Programm P gibt, das gestartet mit

$$n_1, \ldots, n_k$$

in den Variablen $x_1, \ldots, x_k$ (und 0 in den restlichen) stoppt mit dem Wert

$$f(n_1, \ldots, n_k)$$

in der Variablen x_0 – sofern $f(n_1, \ldots, n_k)$ definiert ist, andernfalls stoppt P nicht.

GOTO-Berechenbarkeit: Addition

Beispiel: Die Addition $f : \mathbb{N}^2 \to \mathbb{N}$, $f(n_1, n_2) = n_1 + n_2$ ist GOTO-berechenbar.

Idee: Berechne $n_1 + n_2 = n_1 + \underbrace{1 + 1 + \ldots + 1}_{n_2}$.

$$
\begin{array}{lcl}
M_1 & : & x_0 := x_1 + 0; \\
M_2 & : & \texttt{IF } x_2 = 0 \texttt{ THEN GOTO } M_6; \\
M_3 & : & x_0 := x_0 + 1; \\
M_4 & : & x_2 := x_2 - 1; \\
M_5 & : & \texttt{GOTO } M_2; \\
M_6 & : & \texttt{HALT};
\end{array}
$$

GOTO-Berechenbarkeit: Multiplikation

Beispiel: Die Multiplikation $f : \mathbb{N}^2 \to \mathbb{N}$, definiert durch

$$f(n_1, n_2) = n_1 \cdot n_2,$$

ist GOTO-berechenbar (s. Übungen).

WHILE- versus GOTO-Berechenbarkeit

Theorem

Jedes WHILE-Programm kann durch ein GOTO-Programm simuliert werden, d.h., jede WHILE-berechenbare Funktion ist auch GOTO-berechenbar.

Beweis: Wir simulieren die Schleife

$$\mathtt{WHILE}\ x_i \neq 0\ \mathtt{DO}\ P\ \mathtt{END}$$

durch

$$\begin{array}{lcl} M_1 & : & \mathtt{IF}\ x_i = 0\ \mathtt{THEN}\ \ \mathtt{GOTO}\ M_2; \\ \ldots & & P; \\ \ldots & & \mathtt{GOTO}\ M_1; \\ M_2 & : & \ldots \end{array}$$

❑

GOTO- versus WHILE-Berechenbarkeit

Theorem

Jedes GOTO-Programm kann durch ein WHILE-Programm mit einer WHILE-Schleife simuliert werden, d.h., jede GOTO-berechenbare Funktion ist auch WHILE-berechenbar.

GOTO- versus WHILE-Berechenbarkeit

Beweis: Wir betrachten das GOTO-Programm P:

$$M_1 : A_1; \; M_2 : A_2; \; \ldots; \; M_k : A_k;$$

Wir simulieren P durch folgendes WHILE-Programm mit einer zusätzlichen Variablen x_{Sprung} zur Simulation der Sprungmarken:

$x_{\text{Sprung}} := 1;$

$\texttt{WHILE}\ x_{\text{Sprung}} \neq 0\ \texttt{DO}$

$\texttt{IF}\ x_{\text{Sprung}} = 1\ \texttt{THEN}\ B_1\ \texttt{END};$

$\ldots$

$\ldots$

$\texttt{IF}\ x_{\text{Sprung}} = k\ \texttt{THEN}\ B_k\ \texttt{END}$

$\texttt{END}$

GOTO- versus WHILE-Berechenbarkeit

Dabei ist B_i, $1 \leq i \leq k$, wie folgt definiert:

$$B_i = \begin{cases} A_i;\, x_{\text{Sprung}} := x_{\text{Sprung}} + 1 & \text{falls } A_i \text{ eine Zuweisung ist} \\ x_{\text{Sprung}} := n & \text{falls } A_i = \texttt{GOTO } M_n \text{ ist} \\ x_{\text{Sprung}} := x_{\text{Sprung}} + 1; & \\ \texttt{IF } x_j = c \texttt{ THEN } x_{\text{Sprung}} := n \texttt{ END} & \text{falls } A_i = \texttt{IF } x_j = c \texttt{ THEN GOTO } M_n \text{ ist} \\ x_{\text{Sprung}} := 0 & \text{falls } A_i = \texttt{HALT} \text{ ist} \end{cases}$$

Kleenesche Normalform für WHILE-Programme

Folgerung: Jede WHILE-berechenbare Funktion $f : \mathbb{N}^k \rightarrow \mathbb{N}$ kann durch ein WHILE-Programm mit nur einer WHILE-Schleife (und mehreren IF-THEN-ELSE-Anweisungen) berechnet werden.

Kleenesche Normalform für WHILE-Programme

Beweis:

- Es sei P ein beliebiges WHILE-Programm, das eine Funktion f berechnet.
- Nach dem vorletzten Satz gibt es ein GOTO-Programm P', das f berechnet.
- Nach dem letzten Satz gibt es ein WHILE-Programm P'' mit nur einer WHILE-Schleife und mehreren IF-THEN-ELSE-Anweisungen, das f berechnet. ❑

Turing- versus GOTO-Berechenbarkeit

Theorem

Jede Turingmaschine kann durch ein GOTO-Programm simuliert werden, d.h., jede Turing-berechenbare Funktion ist auch GOTO-berechenbar. **ohne Beweis**

Idee: eine Konfiguration $a_{i_1} \dots a_{i_p} z_l a_{j_1} \dots a_{j_q}$ mit $\Gamma = \{a_1, \dots, a_m\}$ wird dargestellt durch drei Variablen im GOTO Programm:

- $x = (i_1 \dots i_p)_b$ (Zahl in Basis $b > m$)
- $y = (j_q \dots j_1)_b$
- $z = l$

Der Bandinhalt auf dem Lesekopf ist y *mod* b.

LOOP- vs. Turing-, WHILE-, GOTO-Berechenbarkeit

Bemerkung:

- Aus den bisherigen Sätzen ergibt sich:

  GOTO-berechenbar =

  WHILE-berechenbar =

  Turing-berechenbar

  LOOP-berechenbar

- Es gibt WHILE-berechenbare Funktionen, die nicht LOOP-berechenbar sind (z.B. die Ackermann-Funktion).