Script generated by TTT

Title: seidl: Theoretische_Informatik (11.06.2012)

Date: Mon Jun 11 10:19:52 CEST 2012

Duration: 83:46 min

Pages: 96

4.4 LOOP-, WHILE- und GOTO-Berechenbarkeit

LOOP, WHILE $\equiv$ strukturierte Programme mit `for`/`while`-Schleifen

4.4 LOOP-, WHILE- und GOTO-Berechenbarkeit

LOOP, WHILE $\equiv$ strukturierte Programme mit `for`/`while`-Schleifen

GOTO $\equiv$ Assembler

Syntax von LOOP-Programmen:

$$
\begin{array}{lcl}
P & \to & X \text{ := } X \text{ + } C \\
 & | & X \text{ := } X \text{ - } C \\
 & | & P;\ P \\
 & | & \texttt{IF } X = 0 \texttt{ DO } P \texttt{ ELSE} Q \texttt{ END} \\
 & | & \texttt{LOOP } X \texttt{ DO } P \texttt{ END}
\end{array}
$$

(handwritten: DO crossed out, "THEN")

4.4 LOOP-, WHILE- und GOTO-Berechenbarkeit

LOOP, WHILE $\equiv$ strukturierte Programme mit `for`/`while`-Schleifen

GOTO $\equiv$ Assembler

Syntax von LOOP-Programmen:

$$
\begin{array}{lcl}
P & \to & X \text{ := } X \text{ + } C \\
 & | & X \text{ := } X \text{ - } C \\
 & | & P;\ P \\
 & | & \texttt{IF } X = 0 \texttt{ DO } P \texttt{ ELSE} Q \texttt{ END} \\
 & | & \texttt{LOOP } X \texttt{ DO } P \texttt{ END}
\end{array}
$$

wobei X eine der Variablen $x_0, x_1, \ldots$
und C eine der Konstanten $0, 1, \ldots$ sein kann.

4.4 LOOP-, WHILE- und GOTO-Berechenbarkeit

LOOP, WHILE $\equiv$ strukturierte Programme mit `for`/`while`-Schleifen

GOTO $\equiv$ Assembler

Syntax von LOOP-Programmen:

$$
\begin{array}{lll}
P & \rightarrow & X \texttt{ := } X \texttt{ + } C \\
& | & X \texttt{ := } X \texttt{ - } C \\
& | & P\texttt{; } P \\
& | & \texttt{IF } X = 0 \texttt{ DO } P \texttt{ ELSE} Q \texttt{ END} \\
& | & \texttt{LOOP } X \texttt{ DO } P \texttt{ END}
\end{array}
$$

wobei X eine der Variablen $x_0, x_1, \ldots$
und C eine der Konstanten $0, 1, \ldots$ sein kann.

Beispiel 4.14

`LOOP` x_2 `DO` x_1 `:=` x_0`+1 END`

4.4 LOOP-, WHILE- und GOTO-Berechenbarkeit

LOOP, WHILE $\equiv$ strukturierte Programme mit `for`/`while`-Schleifen

GOTO $\equiv$ Assembler

Syntax von LOOP-Programmen:

$$
\begin{array}{lll}
P & \rightarrow & X \texttt{ := } X \texttt{ + } C \\
& | & X \texttt{ := } X \texttt{ - } C \\
& | & P\texttt{; } P \\
& | & \texttt{IF } X = 0 \texttt{ DO } P \texttt{ ELSE} Q \texttt{ END} \\
& | & \texttt{LOOP } X \texttt{ DO } P \texttt{ END}
\end{array}
$$

wobei X eine der Variablen $x_0, x_1, \ldots$
und C eine der Konstanten $0, 1, \ldots$ sein kann.

Beispiel 4.14

`LOOP` x_2 `DO` x_1 `:=` x_0`+1 END`

Die modifizierte Differenz ist $m \dot{-} n := \begin{cases} m - n & \text{falls } m \geq n \\ 0 & \text{sonst} \end{cases}$

Die modifizierte Differenz ist $m \dot{-} n := \begin{cases} m - n & \text{falls } m \geq n \\ 0 & \text{sonst} \end{cases}$

Semantik von LOOP-Programmen (informell):

x_i `:=` x_j `+` n Neuer Wert von x_i ist $x_j + n$.

Die modifizierte Differenz ist $m \dot{-} n \; := \begin{cases} m - n & \text{falls } m \geq n \\ 0 & \text{sonst} \end{cases}$

Semantik von LOOP-Programmen (informell):

`xi := xj + n` Neuer Wert von x_i ist $x_j + n$.

`xi := xj - n` Neuer Wert von x_i ist $x_j \dot{-} n$.

Die modifizierte Differenz ist $m \dot{-} n \; := \begin{cases} m - n & \text{falls } m \geq n \\ 0 & \text{sonst} \end{cases}$

Semantik von LOOP-Programmen (informell):

`xi := xj + n` Neuer Wert von x_i ist $x_j + n$.

`xi := xj - n` Neuer Wert von x_i ist $x_j \dot{-} n$.

`P1 ; P2` Führe zuerst P_1 und dann P_2 aus.

Die modifizierte Differenz ist $m \dot{-} n \; := \begin{cases} m - n & \text{falls } m \geq n \\ 0 & \text{sonst} \end{cases}$

Semantik von LOOP-Programmen (informell):

`xi := xj + n` Neuer Wert von x_i ist $x_j + n$.

`xi := xj - n` Neuer Wert von x_i ist $x_j \dot{-} n$.

`P1 ; P2` Führe zuerst P_1 und dann P_2 aus.

`LOOP xi DO P END` Führe P genau n mal aus, wobei n der *Anfangswert* von x_i ist.

Die modifizierte Differenz ist $m \dot{-} n \; := \begin{cases} m - n & \text{falls } m \geq n \\ 0 & \text{sonst} \end{cases}$

Semantik von LOOP-Programmen (informell):

`xi := xj + n` Neuer Wert von x_i ist $x_j + n$.

`xi := xj - n` Neuer Wert von x_i ist $x_j \dot{-} n$.

`P1 ; P2` Führe zuerst P_1 und dann P_2 aus.

`LOOP xi DO P END` Führe P genau n mal aus, wobei n der *Anfangswert* von x_i ist. Zuweisungen an x_i in P ändern die Anzahl n der Schleifendurchläufe nicht.

Die modifizierte Differenz ist $m \dot{-} n := \begin{cases} m - n & \text{falls } m \geq n \\ 0 & \text{sonst} \end{cases}$

Semantik von LOOP-Programmen (informell):

x_i `:=` x_j `+` n Neuer Wert von x_i ist $x_j + n$.

x_i `:=` x_j `-` n Neuer Wert von x_i ist $x_j \dot{-} n$.

P_1 `;` P_2 Führe zuerst P_1 und dann P_2 aus.

`LOOP` x_i `DO` P `END` Führe P genau n mal aus, wobei n der *Anfangswert* von x_i ist. Zuweisungen an x_i in P ändern die Anzahl n der Schleifendurchläufe nicht.

Beispiel 4.15

`LOOP` x_0 `DO` x_1 `:=` x_1`+1 END` simuliert x_1 `:=` x_1 `+` x_0

Die modifizierte Differenz ist $m \dot{-} n := \begin{cases} m - n & \text{falls } m \geq n \\ 0 & \text{sonst} \end{cases}$

Semantik von LOOP-Programmen (informell):

x_i `:=` x_j `+` n Neuer Wert von x_i ist $x_j + n$.

x_i `:=` x_j `-` n Neuer Wert von x_i ist $x_j \dot{-} n$.

P_1 `;` P_2 Führe zuerst P_1 und dann P_2 aus.

`LOOP` x_i `DO` P `END` Führe P genau n mal aus, wobei n der *Anfangswert* von x_i ist. Zuweisungen an x_i in P ändern die Anzahl n der Schleifendurchläufe nicht.

Beispiel 4.15

`LOOP` x_0 `DO` x_1 `:=` x_1`+1 END` simuliert x_1 `:=` x_1 `+` x_0

Zu Beginn der Ausführung stehen die Eingaben in $x_1, \ldots, x_k$.
Alle anderen Variablen sind 0. Die Ausgabe wird in x_0 berechnet.

Definition 4.16

Eine Funktion $f : \mathbb{N}^k \to \mathbb{N}$ ist LOOP-berechenbar gdw es ein LOOP-Programm P gibt, so dass für alle $n_1, \ldots, n_k \in \mathbb{N}$:

Definition 4.16

Eine Funktion $f : \mathbb{N}^k \to \mathbb{N}$ ist LOOP-berechenbar gdw es ein LOOP-Programm P gibt, so dass für alle $n_1, \ldots, n_k \in \mathbb{N}$:

P, gestartet mit $n_1, \ldots, n_k$ in $x_1, \ldots, x_k$ (0 in den anderen Var.) terminiert mit $f(n_1, \ldots, n_k)$ in x_0.

Definition 4.16
Eine Funktion $f : \mathbb{N}^k \to \mathbb{N}$ ist LOOP-berechenbar gdw es ein LOOP-Programm P gibt, so dass für alle $n_1, \ldots, n_k \in \mathbb{N}$:

P, gestartet mit $n_1, \ldots, n_k$ in $x_1, \ldots, x_k$ (0 in den anderen Var.) terminiert mit $f(n_1, \ldots, n_k)$ in x_0.

Lemma 4.17
Alle LOOP-berechenbaren Funktionen sind total.

Definition 4.16
Eine Funktion $f : \mathbb{N}^k \to \mathbb{N}$ ist LOOP-berechenbar gdw es ein LOOP-Programm P gibt, so dass für alle $n_1, \ldots, n_k \in \mathbb{N}$:

P, gestartet mit $n_1, \ldots, n_k$ in $x_1, \ldots, x_k$ (0 in den anderen Var.) terminiert mit $f(n_1, \ldots, n_k)$ in x_0.

Lemma 4.17
Alle LOOP-berechenbaren Funktionen sind total.

Beweis mit Induktion über die Erzeugung/Struktur der LOOP-Programme.

Definition 4.16
Eine Funktion $f : \mathbb{N}^k \to \mathbb{N}$ ist LOOP-berechenbar gdw es ein LOOP-Programm P gibt, so dass für alle $n_1, \ldots, n_k \in \mathbb{N}$:

P, gestartet mit $n_1, \ldots, n_k$ in $x_1, \ldots, x_k$ (0 in den anderen Var.) terminiert mit $f(n_1, \ldots, n_k)$ in x_0.

Lemma 4.17
Alle LOOP-berechenbaren Funktionen sind total.

Beweis mit Induktion über die Erzeugung/Struktur der LOOP-Programme.

Sind alle totalen Funktionen LOOP-berechenbar?

Definition 4.16
Eine Funktion $f : \mathbb{N}^k \to \mathbb{N}$ ist LOOP-berechenbar gdw es ein LOOP-Programm P gibt, so dass für alle $n_1, \ldots, n_k \in \mathbb{N}$:

P, gestartet mit $n_1, \ldots, n_k$ in $x_1, \ldots, x_k$ (0 in den anderen Var.) terminiert mit $f(n_1, \ldots, n_k)$ in x_0.

Lemma 4.17
Alle LOOP-berechenbaren Funktionen sind total.

Beweis mit Induktion über die Erzeugung/Struktur der LOOP-Programme.

Sind alle totalen Funktionen LOOP-berechenbar?

Syntaktische Abkürzungen („Zucker“):

- x_i := x_j

Syntaktische Abkürzungen („Zucker“):

- x_i := x_j $\equiv$ x_i := x_j+0

Syntaktische Abkürzungen („Zucker“):

- x_i := x_j $\equiv$ x_i := x_j+0
- x_i := n

Syntaktische Abkürzungen („Zucker“):

- x_i := x_j $\equiv$ x_i := x_j+0
- x_i := n $\equiv$ x_i := x_j+n

Syntaktische Abkürzungen („Zucker“):

- x_i := x_j $\equiv$ x_i := x_j+0
- x_i := n $\equiv$ x_i := x_j+n
  (wobei an x_j nirgends zugewiesen wird)
- x_i := x_j+x_k

Syntaktische Abkürzungen („Zucker“):

- x_i := x_j $\equiv$ x_i := x_j+0
- x_i := n $\equiv$ x_i := x_j+n
  (wobei an x_j nirgends zugewiesen wird)
- x_i := x_j+x_k $\equiv$ x_i := x_j; LOOP x_k DO x_i := x_i+1

Syntaktische Abkürzungen („Zucker“):

- x_i := x_j $\equiv$ x_i := x_j+0
- x_i := n $\equiv$ x_i := x_j+n
  (wobei an x_j nirgends zugewiesen wird)
- x_i := x_j+x_k $\equiv$ x_i := x_j; LOOP x_k DO x_i := x_i+1
  (falls $i \neq k$)

Syntaktische Abkürzungen („Zucker“):

- x_i := x_j $\equiv$ x_i := x_j+0
- x_i := n $\equiv$ x_i := x_j+n
  (wobei an x_j nirgends zugewiesen wird)
- x_i := x_j+x_k $\equiv$ x_i := x_j; LOOP x_k DO x_i := x_i+1
  (falls $i \neq k$)
- x_i := x_j*x_k

Syntaktische Abkürzungen („Zucker"):

- x_i := x_j $\equiv$ x_i := x_j+0
- x_i := n $\equiv$ x_i := x_j+n
  (wobei an x_j nirgends zugewiesen wird)
- x_i := x_j+x_k $\equiv$ x_i := x_j; LOOP x_k DO x_i := x_i+1
  (falls $i \neq k$)
- x_i := x_j*x_k $\equiv$ x_i := 0; LOOP x_k DO x_i := x_i+x_j

Syntaktische Abkürzungen („Zucker"):

- x_i := x_j $\equiv$ x_i := x_j+0
- x_i := n $\equiv$ x_i := x_j+n
  (wobei an x_j nirgends zugewiesen wird)
- x_i := x_j+x_k $\equiv$ x_i := x_j; LOOP x_k DO x_i := x_i+1
  (falls $i \neq k$)
- x_i := x_j*x_k $\equiv$ x_i := 0; LOOP x_k DO x_i := x_i+x_j
  (falls $i \neq j, k$)

Syntaktische Abkürzungen („Zucker"):

- x_i := x_j $\equiv$ x_i := x_j+0
- x_i := n $\equiv$ x_i := x_j+n
  (wobei an x_j nirgends zugewiesen wird)
- x_i := x_j+x_k $\equiv$ x_i := x_j; LOOP x_k DO x_i := x_i+1
  (falls $i \neq k$)
- x_i := x_j*x_k $\equiv$ x_i := 0; LOOP x_k DO x_i := x_i+x_j
  (falls $i \neq j, k$)
- DIV, MOD, ...

Syntaktische Abkürzungen („Zucker"):

- x_i := x_j $\equiv$ x_i := x_j+0
- x_i := n $\equiv$ x_i := x_j+n
  (wobei an x_j nirgends zugewiesen wird)
- x_i := x_j+x_k $\equiv$ x_i := x_j; LOOP x_k DO x_i := x_i+1
  (falls $i \neq k$)
- x_i := x_j*x_k $\equiv$ x_i := 0; LOOP x_k DO x_i := x_i+x_j
  (falls $i \neq j, k$)
- DIV, MOD, ...
- x := komplexer Ausdruck

Syntaktische Abkürzungen („Zucker“):

- x_i `:=` x_j $\equiv$ x_i `:=` x_j`+`0
- x_i `:=` n $\equiv$ x_i `:=` x_j`+`n
  (wobei an x_j nirgends zugewiesen wird)
- x_i `:=` x_j`+`x_k $\equiv$ x_i `:=` x_j`;` `LOOP` x_k `DO` x_i `:=` x_i`+`1
  (falls $i \neq k$)
- x_i `:=` x_j`*`x_k $\equiv$ x_i `:=` 0`;` `LOOP` x_k `DO` x_i `:=` x_i`+`x_j
  (falls $i \neq j, k$)
- `DIV`, `MOD`, ...
- x `:=` komplexer Ausdruck
- `IF` $x = 0$ `THEN` P `END`

Syntaktische Abkürzungen („Zucker“):

- x_i `:=` x_j $\equiv$ x_i `:=` x_j`+`0
- x_i `:=` n $\equiv$ x_i `:=` x_j`+`n
  (wobei an x_j nirgends zugewiesen wird)
- x_i `:=` x_j`+`x_k $\equiv$ x_i `:=` x_j`;` `LOOP` x_k `DO` x_i `:=` x_i`+`1
  (falls $i \neq k$)
- x_i `:=` x_j`*`x_k $\equiv$ x_i `:=` 0`;` `LOOP` x_k `DO` x_i `:=` x_i`+`x_j
  (falls $i \neq j, k$)
- `DIV`, `MOD`, ...
- x `:=` komplexer Ausdruck
- `IF` $x = 0$ `THEN` P `END`

WHILE-Programme sind LOOP-Programme erweitert um WHILE-Schleifen:

$$P \;\rightarrow\; \texttt{WHILE } X \neq 0 \texttt{ DO } P \texttt{ END}$$

WHILE-Programme sind LOOP-Programme erweitert um WHILE-Schleifen:

$$P \;\rightarrow\; \texttt{WHILE } X \neq 0 \texttt{ DO } P \texttt{ END}$$

Die Semantik ist wie üblich.

Fakt 4.18
WHILE-Schleifen können LOOP-Schleifen simulieren.

WHILE-Programme sind LOOP-Programme erweitert um WHILE-Schleifen:

$$P \rightarrow \texttt{WHILE } X \neq 0 \texttt{ DO } P \texttt{ END}$$

Die Semantik ist wie üblich.

Fakt 4.18
WHILE-Schleifen können LOOP-Schleifen simulieren.

Fakt 4.19
LOOP-Schleifen können WHILE-Schleifen nicht simulieren.

Definition 4.20
Eine Funktion $f : \mathbb{N}^k \rightarrow \mathbb{N}$ ist WHILE-berechenbar gdw es ein WHILE-Programm P gibt, so dass für alle $n_1, \ldots, n_k \in \mathbb{N}$:

Definition 4.20
Eine Funktion $f : \mathbb{N}^k \rightarrow \mathbb{N}$ ist WHILE-berechenbar gdw es ein WHILE-Programm P gibt, so dass für alle $n_1, \ldots, n_k \in \mathbb{N}$:

P, gestartet mit $n_1, \ldots, n_k$ in $x_1, \ldots, x_k$ (0 in den anderen Var.)

Definition 4.20
Eine Funktion $f : \mathbb{N}^k \rightarrow \mathbb{N}$ ist WHILE-berechenbar gdw es ein WHILE-Programm P gibt, so dass für alle $n_1, \ldots, n_k \in \mathbb{N}$:

P, gestartet mit $n_1, \ldots, n_k$ in $x_1, \ldots, x_k$ (0 in den anderen Var.)

- terminiert mit $f(n_1, \ldots, n_k)$ in x_0, falls $f(n_1, \ldots, n_k)$ definiert ist,

Definition 4.20

Eine Funktion $f : \mathbb{N}^k \rightarrow \mathbb{N}$ ist WHILE-berechenbar gdw es ein WHILE-Programm P gibt, so dass für alle $n_1, \ldots, n_k \in \mathbb{N}$:

P, gestartet mit $n_1, \ldots, n_k$ in $x_1, \ldots, x_k$ (0 in den anderen Var.)

- terminiert mit $f(n_1, \ldots, n_k)$ in x_0, falls $f(n_1, \ldots, n_k)$ definiert ist,
- terminiert nicht, falls $f(n_1, \ldots, n_k)$ undefiniert ist.

Definition 4.20

Eine Funktion $f : \mathbb{N}^k \rightarrow \mathbb{N}$ ist WHILE-berechenbar gdw es ein WHILE-Programm P gibt, so dass für alle $n_1, \ldots, n_k \in \mathbb{N}$:

P, gestartet mit $n_1, \ldots, n_k$ in $x_1, \ldots, x_k$ (0 in den anderen Var.)

- terminiert mit $f(n_1, \ldots, n_k)$ in x_0, falls $f(n_1, \ldots, n_k)$ definiert ist,
- terminiert nicht, falls $f(n_1, \ldots, n_k)$ undefiniert ist.

Turingmaschinen können WHILE-Programme simulieren:

Satz 4.21 (WHILE → TM)

Jede WHILE-berechenbare Funktion ist auch Turing-berechenbar.

Definition 4.20

Eine Funktion $f : \mathbb{N}^k \rightarrow \mathbb{N}$ ist WHILE-berechenbar gdw es ein WHILE-Programm P gibt, so dass für alle $n_1, \ldots, n_k \in \mathbb{N}$:

P, gestartet mit $n_1, \ldots, n_k$ in $x_1, \ldots, x_k$ (0 in den anderen Var.)

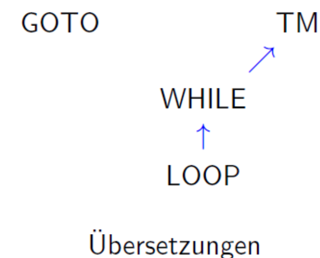
- terminiert mit $f(n_1, \ldots, n_k)$ in x_0, falls $f(n_1, \ldots, n_k)$ definiert ist,
- terminiert nicht, falls $f(n_1, \ldots, n_k)$ undefiniert ist.

Turingmaschinen können WHILE-Programme simulieren:

Satz 4.21 (WHILE → TM)

Jede WHILE-berechenbare Funktion ist auch Turing-berechenbar.

Beweis:
Jede Programmvariable wird auf einem eigenen Band gespeichert. Wir haben bereits gezeigt: Alle Konstrukte der WHILE-Sprache können von einer Mehrband-TM simuliert werden, und eine Mehrband-TM kann von einer 1-Band TM simuliert werden. □



Ein GOTO-Programm ist eine Sequenzen von markierten Anweisungen

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

Ein GOTO-Programm ist eine Sequenzen von markierten Anweisungen

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

(wobei alle Marken verschieden und optional sind)

Mögliche Anweisungen A_i sind:

- x_i `:=` x_j `+` n
- x_i `:=` x_j `-` n
- `GOTO` M_i
- `IF` $x_i = n$ `GOTO` M_j
- `HALT`

Ein GOTO-Programm ist eine Sequenzen von markierten Anweisungen

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

(wobei alle Marken verschieden und optional sind)

Mögliche Anweisungen A_i sind:

- x_i `:=` x_j `+` n
- x_i `:=` x_j `-` n
- `GOTO` M_i
- `IF` $x_i = n$ `GOTO` M_j
- `HALT`

Die Semantik ist wie erwartet.

Ein GOTO-Programm ist eine Sequenzen von markierten Anweisungen

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

(wobei alle Marken verschieden und optional sind)

Mögliche Anweisungen A_i sind:

- x_i `:=` x_j `+` n
- x_i `:=` x_j `-` n
- `GOTO` M_i
- `IF` $x_i = n$ `GOTO` M_j
- `HALT`

Die Semantik ist wie erwartet.

Fakt 4.22 (WHILE $\rightarrow$ GOTO)

Jedes WHILE-Programm kann durch ein GOTO-Programm simuliert werden.

Satz 4.23 (GOTO $\to$ WHILE)

Jedes GOTO-Programm kann durch ein WHILE-Programm simuliert werden.

Satz 4.23 (GOTO $\to$ WHILE)

Jedes GOTO-Programm kann durch ein WHILE-Programm simuliert werden.

Beweis: Simuliere $M_1 : A_1;\ M_2 : A_2;\ \ldots;\ M_k : A_k$ durch

Satz 4.23 (GOTO $\to$ WHILE)

Jedes GOTO-Programm kann durch ein WHILE-Programm simuliert werden.

Beweis: Simuliere $M_1 : A_1;\ M_2 : A_2;\ \ldots;\ M_k : A_k$ durch

```
pc := 1;
WHILE pc ≠ 0 DO
  IF pc = 1 THEN P1 ELSE
  ⋮
  IF pc = k THEN Pk ELSE pc := 0
END
```

Satz 4.23 (GOTO $\to$ WHILE)

Jedes GOTO-Programm kann durch ein WHILE-Programm simuliert werden.

Beweis: Simuliere $M_1 : A_1;\ M_2 : A_2;\ \ldots;\ M_k : A_k$ durch

```
pc := 1;
WHILE pc ≠ 0 DO
  IF pc = 1 THEN P1 ELSE
  ⋮
  IF pc = k THEN Pk ELSE pc := 0
END
```

wobei $A_i \mapsto P_i$ wie folgt definiert ist:

Satz 4.23 (GOTO → WHILE)

Jedes GOTO-Programm kann durch ein WHILE-Programm simuliert werden.

Beweis: Simuliere $M_1 : A_1;\ M_2 : A_2;\ \ldots;\ M_k : A_k$ durch

```
pc := 1;
WHILE pc ≠ 0 DO
  IF pc = 1 THEN P_1 ELSE
  ⋮
  IF pc = k THEN P_k ELSE pc := 0
END
```

wobei $A_i \mapsto P_i$ wie folgt definiert ist:

x_i := x_j +/- n ↦ x_i := x_j +/- n; pc := pc+1

Satz 4.23 (GOTO → WHILE)

Jedes GOTO-Programm kann durch ein WHILE-Programm simuliert werden.

Beweis: Simuliere $M_1 : A_1;\ M_2 : A_2;\ \ldots;\ M_k : A_k$ durch

```
pc := 1;
WHILE pc ≠ 0 DO
  IF pc = 1 THEN P_1 ELSE
  ⋮
  IF pc = k THEN P_k ELSE pc := 0
END
```

wobei $A_i \mapsto P_i$ wie folgt definiert ist:

x_i := x_j +/- n ↦ x_i := x_j +/- n; pc := pc+1

GOTO M_i ↦ pc := i

Satz 4.23 (GOTO → WHILE)

Jedes GOTO-Programm kann durch ein WHILE-Programm simuliert werden.

Beweis: Simuliere $M_1 : A_1;\ M_2 : A_2;\ \ldots;\ M_k : A_k$ durch

```
pc := 1;
WHILE pc ≠ 0 DO
  IF pc = 1 THEN P_1 ELSE
  ⋮
  IF pc = k THEN P_k ELSE pc := 0
END
```

wobei $A_i \mapsto P_i$ wie folgt definiert ist:

x_i := x_j +/- n ↦ x_i := x_j +/- n; pc := pc+1

GOTO M_i ↦ pc := i

IF $x_j = 0$ GOTO M_i ↦ IF $x_j = 0$ THEN pc:=i ELSE pc:=pc+1 END

Satz 4.23 (GOTO → WHILE)

Jedes GOTO-Programm kann durch ein WHILE-Programm simuliert werden.

Beweis: Simuliere $M_1 : A_1;\ M_2 : A_2;\ \ldots;\ M_k : A_k$ durch

```
pc := 1;
WHILE pc ≠ 0 DO
  IF pc = 1 THEN P_1 ELSE
  ⋮
  IF pc = k THEN P_k ELSE pc := 0
END
```

wobei $A_i \mapsto P_i$ wie folgt definiert ist:

x_i := x_j +/- n ↦ x_i := x_j +/- n; pc := pc+1

GOTO M_i ↦ pc := i

IF $x_j = 0$ GOTO M_i ↦ IF $x_j = 0$ THEN pc:=i ELSE pc:=pc+1 END

HALT ↦ pc := 0 □

Korollar 4.24

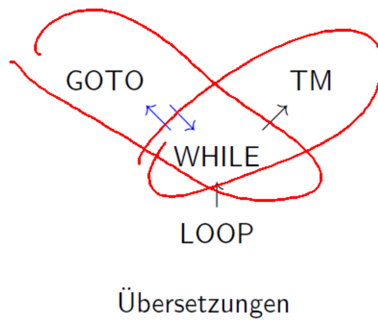
WHILE- und GOTO-Berechenbarkeit sind äquivalent.

Korollar 4.24

WHILE- und GOTO-Berechenbarkeit sind äquivalent.

Korollar 4.25 (Kleenesche Normalform)

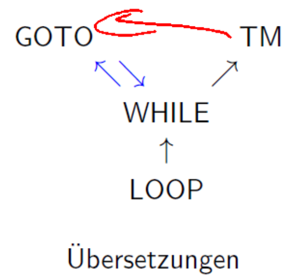
Jedes WHILE-Programm ist zu einem WHILE-Programm mit genau einer WHILE-Schleife äquivalent.



Übersetzungen

Satz 4.26 (TM → GOTO)

Jede TM kann durch ein GOTO-Programm simuliert werden.



Satz 4.26 (TM → GOTO)

Jede TM kann durch ein GOTO-Programm simuliert werden.

Satz 4.26 (TM → GOTO)

Jede TM kann durch ein GOTO-Programm simuliert werden.

Übersetzung einer TM $(Q, \Sigma, \Gamma, \delta, q_0, \Box, F)$ in ein GOTO-Programm. Zeichen und Zeichenreihen werden als Zahlen kodiert.

Satz 4.26 (TM → GOTO)

Jede TM kann durch ein GOTO-Programm simuliert werden.

Übersetzung einer TM $(Q, \Sigma, \Gamma, \delta, q_0, \Box, F)$ in ein GOTO-Programm. Zeichen und Zeichenreihen werden als Zahlen kodiert.

$$Q = \{q_0, \ldots, q_k\} \qquad \Gamma = \{a_0 (= \Box), \ldots, a_n\}$$

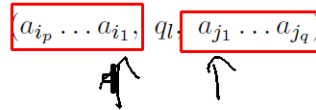
Satz 4.26 (TM → GOTO)

Jede TM kann durch ein GOTO-Programm simuliert werden.

Übersetzung einer TM $(Q, \Sigma, \Gamma, \delta, q_0, \Box, F)$ in ein GOTO-Programm. Zeichen und Zeichenreihen werden als Zahlen kodiert.

$$Q = \{q_0, \dots, q_k\} \qquad \Gamma = \{a_0 (= \Box), \dots, a_n\}$$

Eine Konfiguration



Satz 4.26 (TM → GOTO)

Jede TM kann durch ein GOTO-Programm simuliert werden.

Übersetzung einer TM $(Q, \Sigma, \Gamma, \delta, q_0, \Box, F)$ in ein GOTO-Programm. Zeichen und Zeichenreihen werden als Zahlen kodiert.

$$Q = \{q_0, \dots, q_k\} \qquad \Gamma = \{a_0 (= \Box), \dots, a_n\}$$

Eine Konfiguration

$$(a_{i_p} \dots a_{i_1},\ q_l,\ a_{j_1} \dots a_{j_q})$$

wird durch die Programmvariablen x, y, z wie folgt repräsentiert:

$$x = (i_p \dots i_1)_b, \quad y = (j_q \dots j_1)_b, \quad z = l$$

Satz 4.26 (TM → GOTO)

Jede TM kann durch ein GOTO-Programm simuliert werden.

Übersetzung einer TM $(Q, \Sigma, \Gamma, \delta, q_0, \Box, F)$ in ein GOTO-Programm. Zeichen und Zeichenreihen werden als Zahlen kodiert.

$$Q = \{q_0, \dots, q_k\} \qquad \Gamma = \{a_0 (= \Box), \dots, a_n\}$$

Eine Konfiguration

$$(a_{i_p} \dots a_{i_1},\ q_l,\ a_{j_1} \dots a_{j_q})$$

wird durch die Programmvariablen x, y, z wie folgt repräsentiert:

$$x = (i_p \dots i_1)_b, \quad y = (j_q \dots j_1)_b, \quad z = l$$

wobei $(i_p \dots i_1)_b$ die Zahl $i_p \dots i_1$ zur Basis $b := n + 1$ ist:

Satz 4.26 (TM → GOTO)

Jede TM kann durch ein GOTO-Programm simuliert werden.

Übersetzung einer TM $(Q, \Sigma, \Gamma, \delta, q_0, \Box, F)$ in ein GOTO-Programm. Zeichen und Zeichenreihen werden als Zahlen kodiert.

$$Q = \{q_0, \dots, q_k\} \qquad \Gamma = \{a_0 (= \Box), \dots, a_n\}$$

Eine Konfiguration

$$(a_{i_p} \dots a_{i_1},\ q_l,\ a_{j_1} \dots a_{j_q})$$

wird durch die Programmvariablen x, y, z wie folgt repräsentiert:

$$x = (i_p \dots i_1)_b, \quad y = (j_q \dots j_1)_b, \quad z = l$$

wobei $(i_p \dots i_1)_b$ die Zahl $i_p \dots i_1$ zur Basis $b := n + 1$ ist:

$$x = \sum_{r=1}^{p} i_r b^{r-1}$$

Der Kern des GOTO-Programms ist die iterierte Simulation von δ:

M: IF $z \in F$ GOTO M_{end};

Der Kern des GOTO-Programms ist die iterierte Simulation von δ:

M: IF $z \in F$ GOTO M_{end};
a := y MOD b;

Der Kern des GOTO-Programms ist die iterierte Simulation von δ:

M: IF $z \in F$ GOTO M_{end};
a := y MOD b;
IF $z = 0$ AND $a = 0$ GOTO M_{00};
IF $z = 0$ AND $a = 1$ GOTO M_{01};
...
IF $z = k$ AND $a = n$ GOTO M_{kn};

Der Kern des GOTO-Programms ist die iterierte Simulation von δ:

M: IF $z \in F$ GOTO M_{end};
a := y MOD b;
IF $z = 0$ AND $a = 0$ GOTO M_{00};
IF $z = 0$ AND $a = 1$ GOTO M_{01};
...
IF $z = k$ AND $a = n$ GOTO M_{kn};
M_{00}: P_{00}; GOTO M;
M_{01}: P_{01}; GOTO M;
...
M_{kn}: P_{kn}; GOTO M

Der Kern des GOTO-Programms ist die iterierte Simulation von δ:

M: IF $z \in F$ GOTO M_{end};
a := y MOD b;
IF $z = 0$ AND $a = 0$ GOTO M_{00};
IF $z = 0$ AND $a = 1$ GOTO M_{01};
...
IF $z = k$ AND $a = n$ GOTO M_{kn};
M_{00}: P_{00}; GOTO M;
M_{01}: P_{01}; GOTO M;
...
M_{kn}: P_{kn}; GOTO M

wobe P_{ij} die Simulation von $\delta(q_i, a_j) = (q_r, a_s, d)$ ist.

Der Kern des GOTO-Programms ist die iterierte Simulation von δ:

M: IF $z \in F$ GOTO M_{end};
a := y MOD b;
IF $z = 0$ AND $a = 0$ GOTO M_{00};
IF $z = 0$ AND $a = 1$ GOTO M_{01};
...
IF $z = k$ AND $a = n$ GOTO M_{kn};
M_{00}: P_{00}; GOTO M;
M_{01}: P_{01}; GOTO M;
...
M_{kn}: P_{kn}; GOTO M

wobe P_{ij} die Simulation von $\delta(q_i, a_j) = (q_r, a_s, d)$ ist. Für $d = L$:

z := r; Zustand aktualisieren

Der Kern des GOTO-Programms ist die iterierte Simulation von δ:

M: IF $z \in F$ GOTO M_{end};
a := y MOD b;
IF $z = 0$ AND $a = 0$ GOTO M_{00};
IF $z = 0$ AND $a = 1$ GOTO M_{01};
...
IF $z = k$ AND $a = n$ GOTO M_{kn};
M_{00}: P_{00}; GOTO M;
M_{01}: P_{01}; GOTO M;
...
M_{kn}: P_{kn}; GOTO M

wobe P_{ij} die Simulation von $\delta(q_i, a_j) = (q_r, a_s, d)$ ist. Für $d = L$:

z := r; Zustand aktualisieren
y := y DIV b; Löschen von a_j

Der Kern des GOTO-Programms ist die iterierte Simulation von δ:

M: IF $z \in F$ GOTO M_{end};
a := y MOD b;
IF $z = 0$ AND $a = 0$ GOTO M_{00};
IF $z = 0$ AND $a = 1$ GOTO M_{01};
...
IF $z = k$ AND $a = n$ GOTO M_{kn};
M_{00}: P_{00}; GOTO M;
M_{01}: P_{01}; GOTO M;
...
M_{kn}: P_{kn}; GOTO M

wobe P_{ij} die Simulation von $\delta(q_i, a_j) = (q_r, a_s, d)$ ist. Für $d = L$:

z := r; Zustand aktualisieren
y := y DIV b; Löschen von a_j
y := b*y + s; Schreiben von a_s

Der Kern des GOTO-Programms ist die iterierte Simulation von δ:

```
M:    IF z ∈ F GOTO M_end;
      a := y MOD b;
      IF z = 0 AND a = 0 GOTO M_00;
      IF z = 0 AND a = 1 GOTO M_01;
      ...
      IF z = k AND a = n GOTO M_kn;
M_00: P_00; GOTO M;
M_01: P_01; GOTO M;
      ...
M_kn: P_kn; GOTO M
```

wobe P_{ij} die Simulation von $\delta(q_i, a_j) = (q_r, a_s, d)$ ist. Für $d = L$:

`z := r;`	Zustand aktualisieren
`y := y DIV b;`	Löschen von a_j
`y := b*y + s;`	Schreiben von a_s
`y := b*y + (x MOD b);`	Bewegung L (I): Einfügen von a_{i_1} in y

Der Kern des GOTO-Programms ist die iterierte Simulation von δ:

```
M:    IF z ∈ F GOTO M_end;
      a := y MOD b;
      IF z = 0 AND a = 0 GOTO M_00;
      IF z = 0 AND a = 1 GOTO M_01;
      ...
      IF z = k AND a = n GOTO M_kn;
M_00: P_00; GOTO M;
M_01: P_01; GOTO M;
      ...
M_kn: P_kn; GOTO M
```

wobe P_{ij} die Simulation von $\delta(q_i, a_j) = (q_r, a_s, d)$ ist. Für $d = L$:

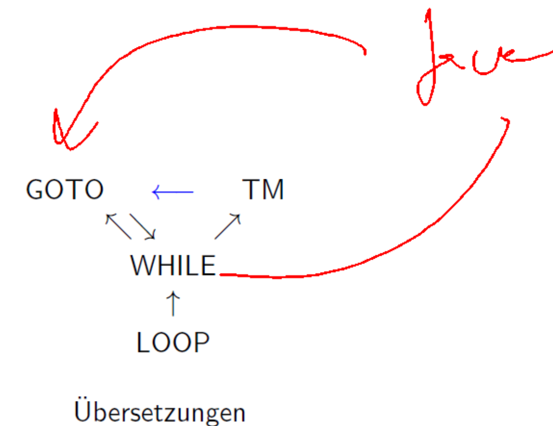
`z := r;`	Zustand aktualisieren
`y := y DIV b;`	Löschen von a_j
`y := b*y + s;`	Schreiben von a_s
`y := b*y + (x MOD b);`	Bewegung L (I): Einfügen von a_{i_1} in y
`x := x DIV b`	Bewegung L (II): Löschen von a_{i_1} aus x

Der Kern des GOTO-Programms ist die iterierte Simulation von δ:

```
M:    IF z ∈ F GOTO M_end;
      a := y MOD b;
      IF z = 0 AND a = 0 GOTO M_00;
      IF z = 0 AND a = 1 GOTO M_01;
      ...
      IF z = k AND a = n GOTO M_kn;
M_00: P_00; GOTO M;
M_01: P_01; GOTO M;
      ...
M_kn: P_kn; GOTO M
```

wobe P_{ij} die Simulation von $\delta(q_i, a_j) = (q_r, a_s, d)$ ist. Für $d = L$:

`z := r;`	Zustand aktualisieren
`y := y DIV b;`	Löschen von a_j
`y := b*y + s;`	Schreiben von a_s
`y := b*y + (x MOD b);`	Bewegung L (I): Einfügen von a_{i_1} in y
`x := x DIV b`	Bewegung L (II): Löschen von a_{i_1} aus x



4.5 Primitiv rekursive Funktionen

4.5 Primitiv rekursive Funktionen

Klassen von Programmiersprachen

Imperativ	Funktional
C, Java,... TM, WHILE, GOTO	OCaml, Lisp, ... μ-rekursiv
LOOP	Primitiv rekursiv

4.5 Primitiv rekursive Funktionen

Klassen von Programmiersprachen

Imperativ	Funktional
C, Java,... TM, WHILE, GOTO	OCaml, Lisp, ... μ-rekursiv
LOOP	Primitiv rekursiv

Wir betrachten Funktionen $\mathbb{N}^k \to \mathbb{N}$, $k \geq 0$.

4.5 Primitiv rekursive Funktionen

Klassen von Programmiersprachen

Imperativ	Funktional
C, Java,... TM, WHILE, GOTO	OCaml, Lisp, ... μ-rekursiv
LOOP	Primitiv rekursiv

Wir betrachten Funktionen $\mathbb{N}^k \to \mathbb{N}$, $k \geq 0$.
Wir identifizieren $\mathbb{N}^0 \to \mathbb{N}$ mit $\mathbb{N}$ und $c()$ mit c.

4.5 Primitiv rekursive Funktionen

Klassen von Programmiersprachen

Imperativ	Funktional
C, Java,... TM, WHILE, GOTO	OCaml, Lisp, ... μ-rekursiv
LOOP	Primitiv rekursiv

Wir betrachten Funktionen $\mathbb{N}^k \to \mathbb{N}$, $k \geq 0$.
Wir identifizieren $\mathbb{N}^0 \to \mathbb{N}$ mit $\mathbb{N}$ und $c()$ mit c.

Definition der primitiv rekursiven Funktionen:

Fixe Basisfunktionen zB $s(x) = x + 1$

4.5 Primitiv rekursive Funktionen

Klassen von Programmiersprachen

Imperativ	Funktional
C, Java,... TM, WHILE, GOTO	OCaml, Lisp, ... μ-rekursiv
LOOP	Primitiv rekursiv

Wir betrachten Funktionen $\mathbb{N}^k \to \mathbb{N}$, $k \geq 0$.
Wir identifizieren $\mathbb{N}^0 \to \mathbb{N}$ mit $\mathbb{N}$ und $c()$ mit c.

Definition der primitiv rekursiven Funktionen:

Fixe Basisfunktionen zB $s(x) = x + 1$

Funktionskomposition zB $f(x, y) = g(x, h(x, y))$

4.5 Primitiv rekursive Funktionen

Klassen von Programmiersprachen

Imperativ	Funktional
C, Java,... TM, WHILE, GOTO	OCaml, Lisp, ... μ-rekursiv
LOOP	Primitiv rekursiv

Wir betrachten Funktionen $\mathbb{N}^k \to \mathbb{N}$, $k \geq 0$.
Wir identifizieren $\mathbb{N}^0 \to \mathbb{N}$ mit $\mathbb{N}$ und $c()$ mit c.

Definition der primitiv rekursiven Funktionen:

Fixe Basisfunktionen zB $s(x) = x + 1$

Funktionskomposition zB $f(x, y) = g(x, h(x, y))$

Fixe Art der Rekursion zB

$$\begin{aligned} f(0) &= 1 \\ f(n+1) &= n * f(n) \end{aligned}$$

Definition 4.27 (Basisfunktionen)

- Die konstante Funktion 0
- Die Nachfolgerfunktion $s(n) = n + 1$

Definition 4.27 (Basisfunktionen)

- Die konstante Funktion 0
- Die Nachfolgerfunktion $s(n) = n + 1$
- Die Projektionsfunktionen $\pi_i^k : \mathbb{N}^k \to \mathbb{N}$, $1 \leq i \leq k$:

$$\pi_i^k(x_1, \ldots, x_k) = x_i$$

4.5 Primitiv rekursive Funktionen

Klassen von Programmiersprachen

Imperativ	Funktional
C, Java,... TM, WHILE, GOTO	OCaml, Lisp, ... μ-rekursiv
LOOP	Primitiv rekursiv

Wir betrachten Funktionen $\mathbb{N}^k \to \mathbb{N}$, $k \geq 0$.
Wir identifizieren $\mathbb{N}^0 \to \mathbb{N}$ mit $\mathbb{N}$ und $c()$ mit c.

Definition der primitiv rekursiven Funktionen:

- Fixe Basisfunktionen: zB $s(x) = x + 1$
- Funktionskomposition: zB $f(x, y) = g(x, h(x, y))$
- Fixe Art der Rekursion: zB

$$\begin{aligned} f(0) &= 1 \\ f(n+1) &= n * f(n) \end{aligned}$$

Definition 4.27 (Basisfunktionen)

- Die konstante Funktion 0
- Die Nachfolgerfunktion $s(n) = n + 1$
- Die Projektionsfunktionen $\pi_i^k : \mathbb{N}^k \to \mathbb{N}$, $1 \leq i \leq k$:

$$\pi_i^k(x_1, \ldots, x_k) = x_i$$

Definition 4.28
Die Komposition von g und $h_1, \ldots, h_k$ erzeugt die Funktion f

$$f(\overline{x}) = g(h_1(\overline{x}), \ldots, h_k(\overline{x}))$$

Definition 4.27 (Basisfunktionen)

- Die konstante Funktion 0
- Die Nachfolgerfunktion $s(n) = n + 1$
- Die Projektionsfunktionen $\pi_i^k : \mathbb{N}^k \to \mathbb{N}$, $1 \leq i \leq k$:

$$\pi_i^k(x_1, \ldots, x_k) = x_i$$

Definition 4.28
Die Komposition von g und $h_1, \ldots, h_k$ erzeugt die Funktion f

$$f(\overline{x}) = g(h_1(\overline{x}), \ldots, h_k(\overline{x}))$$

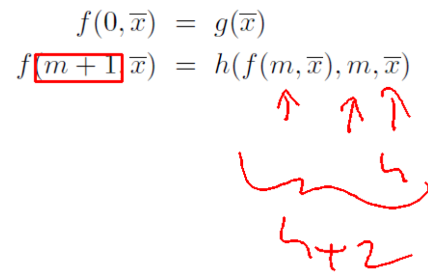
wobei $\overline{x} = (x_1, \ldots, x_n)$ und

$$\begin{aligned} &f : \mathbb{N}^n \to \mathbb{N} \\ &g : \mathbb{N}^k \to \mathbb{N} \\ &h_i : \mathbb{N}^n \to \mathbb{N} \qquad (i = 1, \ldots, k) \end{aligned}$$

Definition 4.29

Das Schema der primitiven Rekursion erzeugt aus g und h die Funktion f

$$\begin{aligned} f(0, \overline{x}) &= g(\overline{x}) \\ f(m+1, \overline{x}) &= h(f(m, \overline{x}), m, \overline{x}) \end{aligned}$$



Definition 4.29

Das Schema der primitiven Rekursion erzeugt aus g und h die Funktion f

$$\begin{aligned} f(0, \overline{x}) &= g(\overline{x}) \\ f(m+1, \overline{x}) &= h(f(m, \overline{x}), m, \overline{x}) \end{aligned}$$

wobei $\overline{x} = (x_1, \ldots, x_n)$ und

$$\begin{aligned} f &: \mathbb{N}^{n+1} \to \mathbb{N} \\ g &: \mathbb{N}^{n} \to \mathbb{N} \\ h &: \mathbb{N}^{n+2} \to \mathbb{N} \end{aligned}$$

Definition 4.30 (PR)

Die Menge PR der primitiv rekursiven Funktionen ist die folgende induktiv definierte Teilmenge aller Funktionen $\mathbb{N}^k \to \mathbb{N}$, $k \geq 0$:

Definition 4.30 (PR)

Die Menge PR der primitiv rekursiven Funktionen ist die folgende induktiv definierte Teilmenge aller Funktionen $\mathbb{N}^k \to \mathbb{N}$, $k \geq 0$:

- Die Basisfunktionen 0, s, und π_i^k sind primitiv rekursiv.
- Sind g und h_i primitiv rekursiv, dann auch ihre Komposition

$$f(\overline{x}) = g(h_1(\overline{x}), \ldots, h_k(\overline{x}))$$

- Sind g und h primitiv rekursiv, dann auch die mit primitiver Rekursion definierte Funktion

$$\begin{aligned} f(0, \overline{x}) &= g(\overline{x}) \\ f(m+1, \overline{x}) &= h(f(m, \overline{x}), m, \overline{x}) \end{aligned}$$

Definition 4.30 (PR)

Die Menge PR der primitiv rekursiven Funktionen ist die folgende induktiv definierte Teilmenge aller Funktionen $\mathbb{N}^k \to \mathbb{N}$, $k \geq 0$:

- Die Basisfunktionen 0, s, und π_i^k sind primitiv rekursiv.
- Sind g und h_i primitiv rekursiv, dann auch ihre Komposition

$$f(\overline{x}) = g(h_1(\overline{x}), \ldots, h_k(\overline{x}))$$

- Sind g und h primitiv rekursiv, dann auch die mit primitiver Rekursion definierte Funktion

$$\begin{aligned} f(0, \overline{x}) &= g(\overline{x}) \\ f(m+1, \overline{x}) &= h(f(m, \overline{x}), m, \overline{x}) \end{aligned}$$

Lemma 4.31

Jede primitiv-rekursive Funktion ist total.