LOOP-, WHILE-, und GOTO-Berechenbarkeit

Quelle: commons.wikimedia.org/wiki/File:Nowitna_river.jpg

LOOP-Programme I

Programmiersprache LOOP zur Berechnung von Funktionen $\mathbb{N}^k \to \mathbb{N}$:

Variablen: $x_0, x_1, \ldots$
Konstanten: $0, 1, 2, \ldots$
Trennsymbole: $;\,, :=$
Operationen: $+, -$
Schlüsselwörter: **LOOP**, **DO**, **END**

LOOP-Programme I

Programmiersprache LOOP zur Berechnung von Funktionen $\mathbb{N}^k \to \mathbb{N}$:

Variablen: $x_0, x_1, \ldots$ $\in \mathbb{N}$
Konstanten: $0, 1, 2, \ldots$
Trennsymbole: $;$, $:=$
Operationen: $+, -$
Schlüsselwörter: **LOOP**, **DO**, **END**

Eingabe: $x_1, \ldots, x_k$ (alle anderen Variablen $x_i = 0$)
Berechnungsergebnis: Wert von x_0 am Programmende

LOOP-Programme I

Programmiersprache LOOP zur Berechnung von Funktionen $\mathbb{N}^k \to \mathbb{N}$:

Variablen: $x_0, x_1, \ldots$ $\in \mathbb{N}$
Konstanten: $0, 1, 2, \ldots$
Trennsymbole: $;\,,\,:=$
Operationen: $+, -$
Schlüsselwörter: **LOOP**, **DO**, **END**

Eingabe: $x_1, \ldots, x_k$ (alle anderen Variablen $x_i = 0$)
Berechnungsergebnis: Wert von x_0 am Programmende

Syntax	**Semantik**
$\underline{x_i := x_j + c}$ (x_i, x_j Variablen, $c \in \mathbb{N}$)	Addition
$x_i := x_j - c$ (x_i, x_j Variablen, $c \in \mathbb{N}$)	Modifizierte Subtraktion $\max\{x_j - c, 0\}$
$\underline{P_1; P_2}$ (P_1, P_2 LOOP Programme)	<u>Sequenz</u> erst P_1, dann P_2
LOOP x_i **DO** $\underline{P}$ **END** (x_i Variable, P LOOP Programm)	<u>Schleife</u> #Durchläufe = Wert von $\underline{x_i}$ vor der Anweisung!

$x_2 := x_1 + 5$

$x_2 := x_1 - 5$

LOOP-Programme II

Simulation anderer Rechenoperationen durch LOOP-Programme:

$x_i := x_j$	$x_i := x_j + 0$
$x_i := c$	$x_i := x_j + c$ (für ein x_j mit $x_j = 0$)
IF $x_i = 0$ **THEN** P **END**	$x_j := 1$; **LOOP** x_i **DO** $\overbrace{x_j := 0}^{P}$ **END**; **LOOP** x_j **DO** P **END**
$x_0 := x_1 + x_2$	$x_0 := x_1$; **LOOP** x_2 **DO** $x_0 := x_0 + 1$ **END**

LOOP-Programme II

Simulation anderer Rechenoperationen durch LOOP-Programme:

$x_i := x_j$	$x_i := x_j + 0$
* $x_i := c$	$x_i := x_j + c$ (für ein x_j mit $x_j = 0$)
IF $x_i = 0$ **THEN** P **END**	$x_j := 1$; **LOOP** x_i **DO** $x_j := 0$ **END**; **LOOP** x_j **DO** P **END**
$x_0 := x_1 + x_2$	$x_0 := x_1$; **LOOP** x_2 **DO** $x_0 := x_0 + 1$ **END**

Analog: Multiplikation, ganzzahlige Division, Modulo.

$x_k := x_i \cdot x_j$

$x_k := 0$;
LOOP x_i DO $x_k := x_k + x_j$ END

LOOP-Programme II

Simulation anderer Rechenoperationen durch LOOP-Programme:

$x_i := x_j$	$x_i := x_j + 0$
$x_i := c$	$x_i := x_j + c$ (für ein x_j mit $x_j = 0$)
IF $x_i = 0$ **THEN** P **END**	$x_j := 1$; **LOOP** x_i **DO** $x_j := 0$ **END**; **LOOP** x_j **DO** P **END**
$x_0 := x_1 + x_2$	$x_0 := x_1$; **LOOP** x_2 **DO** $x_0 := x_0 + 1$ **END**

Analog: Multiplikation, ganzzahlige Division, Modulo.

Frage: Wie sehen LOOP-Programme für `div` & `mod` aus?

WHILE-Programme

WHILE-Programme $\hat{=}$ LOOP-Programme $+$ WHILE-Schleifen

WHILE-Programme

WHILE-Programme $\triangleq$ LOOP-Programme + WHILE-Schleifen

Syntax	**Semantik**
WHILE $x_i \neq 0$ **DO** P **END**	While Schleife
(x_i Variable, P WHILE Programm)	wiederhole P bis $x_i = 0$; x_i darf durch P modifiziert werden!

Berechnungsergebnis: Wert von x_0 am Programmende (falls Programm terminiert).

$x_1 := 1;$

WHILE $x_1 \neq 0$ DO $x_1 := 1$ END

WHILE-Programme

WHILE-Programme $\hat{=}$ LOOP-Programme + WHILE-Schleifen

Syntax	**Semantik**
WHILE $x_i \neq 0$ **DO** P **END**	While Schleife
(x_i Variable, P WHILE Programm)	wiederhole P bis $x_i = 0$; x_i darf durch P modifiziert werden!

Berechnungsergebnis: Wert von x_0 am Programmende (falls Programm terminiert).

Definition (LOOP-/WHILE-Berechenbarkeit)

wenn ein allwissendes Orakel P angeben kann

Eine Funktion $f : \mathbb{N}^k \to \mathbb{N}$ heißt **LOOP-berechenbar** bzw. **WHILE-berechenbar** wenn es ein LOOP- bzw. WHILE-Programm P gibt, das f berechnet, d.h. für alle $n_1, \ldots, n_k, m \in \mathbb{N}$ gilt

$$f(n_1, \ldots, n_k) = m \iff P \text{ hält bei Eingabe } n_1, \ldots, n_k \text{ mit Berechnungsergebnis } x_0 = m$$

WHILE-Programme

LOOP $\underline{x_i}$ DO P END

WHILE-Programme $\hat{=}$ LOOP-Programme + WHILE-Schleifen

Syntax	Semantik
WHILE $x_i \neq 0$ **DO** P **END**	While Schleife
(x_i Variable, P WHILE Programm)	wiederhole P bis $x_i = 0$; x_i darf durch P modifiziert werden!

Berechnungsergebnis: Wert von x_0 am Programmende (falls Programm terminiert).

Definition (LOOP-/WHILE-Berechenbarkeit)

Eine Funktion $f : \mathbb{N}^k \to \mathbb{N}$ heißt **LOOP-berechenbar** bzw. **WHILE-berechenbar** wenn es ein LOOP- bzw. WHILE-Programm P gibt, das f berechnet, d.h. für alle $n_1, \ldots, n_k, m \in \mathbb{N}$ gilt

$$f(n_1, \ldots, n_k) = m \iff P \text{ hält bei Eingabe } n_1, \ldots, n_k \text{ mit Berechnungsergebnis } x_0 = m$$

$\Leftarrow$

Beobachtung: LOOP-Programme berechnen nur totale Funktionen.

WHILE-Programme

WHILE-Programme $\hat{=}$ LOOP-Programme + WHILE-Schleifen

Syntax	**Semantik**
WHILE $x_i \neq 0$ **DO** P **END**	While Schleife
(x_i Variable, P WHILE Programm)	wiederhole P bis $x_i = 0$; x_i darf durch P modifiziert werden!

Berechnungsergebnis: Wert von x_0 am Programmende (falls Programm terminiert).

Definition (LOOP-/WHILE-Berechenbarkeit)

Eine Funktion $f : \mathbb{N}^k \to \mathbb{N}$ heißt **LOOP-berechenbar** bzw. **WHILE-berechenbar** wenn es ein LOOP- bzw. WHILE-Programm P gibt, das f berechnet, d.h. für alle $n_1, \ldots, n_k, m \in \mathbb{N}$ gilt

$$f(n_1, \ldots, n_k) = m \iff P \text{ hält bei Eingabe } n_1, \ldots, n_k \text{ mit Berechnungsergebnis } x_0 = m$$

Beobachtung: LOOP-Programme berechnen nur totale Funktionen.

Leitfrage: alle ~~totalen~~ intuitiv berechenbaren Funktionen über $\mathbb{N}$ ~~LOOP~~ WHILE-berechenbar? NEIN

WHILE-Programme

WHILE-Programme $\hat{=}$ LOOP-Programme + WHILE-Schleifen

Syntax	**Semantik**
WHILE $x_i \neq 0$ **DO** P **END**	While Schleife
(x_i Variable, P WHILE Programm)	wiederhole P bis $x_i = 0$; x_i darf durch P modifiziert werden!

Berechnungsergebnis: Wert von x_0 am Programmende (falls Programm terminiert).

Definition (LOOP-/WHILE-Berechenbarkeit)

Eine Funktion $f : \mathbb{N}^k \to \mathbb{N}$ heißt **LOOP-berechenbar** bzw. **WHILE-berechenbar** wenn es ein LOOP- bzw. WHILE-Programm P gibt, das f berechnet, d.h. für alle $n_1, \ldots, n_k, m \in \mathbb{N}$ gilt

$f(n_1, \ldots, n_k) = m \iff$ P hält bei Eingabe $n_1, \ldots, n_k$ mit Berechnungsergebnis $x_0 = m$

Beobachtung: LOOP-Programme berechnen nur totale Funktionen.
Leitfrage: alle totalen intuitiv berechenbaren Funktionen über $\mathbb{N}$ LOOP-berechenbar?
Zentraler Ansatz nachfolgend: **Simulation**.

WHILE-Programme

WHILE-Programme $\hat{=}$ LOOP-Programme + WHILE-Schleifen

Syntax	**Semantik**
WHILE $x_i \neq 0$ **DO** P **END**	While Schleife
(x_i Variable, P WHILE Programm)	wiederhole P bis $x_i = 0$; x_i darf durch P modifiziert werden!

Berechnungsergebnis: Wert von x_0 am Programmende (falls Programm terminiert).

Definition (LOOP-/WHILE-Berechenbarkeit)

Eine Funktion $f : \mathbb{N}^k \to \mathbb{N}$ heißt **LOOP-berechenbar** bzw. **WHILE-berechenbar** wenn es ein LOOP- bzw. WHILE-Programm P gibt, das f berechnet, d.h. für alle $n_1, \ldots, n_k, m \in \mathbb{N}$ gilt

$$f(n_1, \ldots, n_k) = m \iff P \text{ hält bei Eingabe } n_1, \ldots, n_k \text{ mit Berechnungsergebnis } x_0 = m$$

Beobachtung: LOOP-Programme berechnen nur totale Funktionen.
Leitfrage: alle totalen intuitiv berechenbaren Funktionen über $\mathbb{N}$ LOOP-berechenbar?
Zentraler Ansatz nachfolgend: **Simulation**.

Frage: Wie kann eine LOOP Schleife durch eine WHILE Schleife simuliert werden?

WHILE-Programme & Turing-Maschinen

Theorem

Jede WHILE-berechenbare Funktionen ist Turing-berechenbar.

WHILE-Programme & Turing-Maschinen

Theorem

Jede WHILE-berechenbare Funktionen ist Turing-berechenbar.

Beweis (Skizze)

Bauen TM $M(P)$ durch Induktion über die „Termstruktur" eines WHILE-Programms P:

Annahme: M(P) hat „ausreichend viele" Bänder
(für jede Variable ein eigenes Band)

WHILE-Programme & Turing-Maschinen

Theorem

Jede WHILE-berechenbare Funktionen ist Turing-berechenbar.

Beweis (Skizze)

Bauen TM $M(P)$ durch Induktion über die „Termstruktur“ eines WHILE-Programms P:

Fall 1: $P = \underline{x_i} := \underline{x_j \pm c}$ $\leadsto$ klar Turing-berechenbar (vgl. Binärzahl-Inkrementierer)

1. x_j nach x_i kopieren
   Inhalt v. Band j auf Band i kopieren
2. c mal Inkrementierer auf Band i laufen lassen

WHILE-Programme & Turing-Maschinen

Theorem

Jede WHILE-berechenbare Funktionen ist Turing-berechenbar.

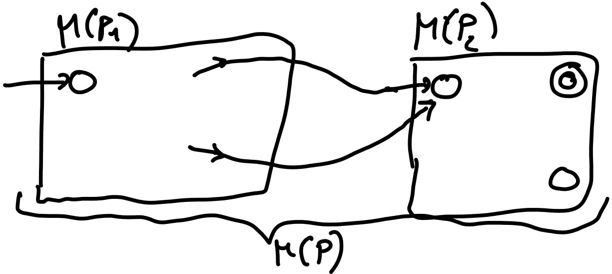
Beweis (Skizze)

Bauen TM $M(P)$ durch Induktion über die „Termstruktur“ eines WHILE-Programms P:

Fall 1: $P = x_i := x_j \pm c$ $\rightsquigarrow$ klar Turing-berechenbar (vgl. Binärzahl-Inkrementierer)

Fall 2: $P = P_1; P_2$ $\rightsquigarrow$ Hintereinanderschaltung von $M(P_1)$ und $M(P_2)$

Identifiziere Endzustände von $M(P_1)$ mit Startzustand von $M(P_2)$.



WHILE-Programme & Turing-Maschinen

Theorem

Jede WHILE-berechenbare Funktionen ist Turing-berechenbar.

Beweis (Skizze)

Bauen TM $M(P)$ durch Induktion über die „Termstruktur“ eines WHILE-Programms P:

Fall 1: $P = x_i := x_j \pm c$ $\rightsquigarrow$ klar Turing-berechenbar (vgl. Binärzahl-Inkrementierer)

Fall 2: $P = P_1; P_2$ $\rightsquigarrow$ Hintereinanderschaltung von $M(P_1)$ und $M(P_2)$

Identifiziere Endzustände von $M(P_1)$ mit Startzustand von $M(P_2)$.

Fall 3: $P =$ **WHILE** $x_i \neq 0$ **DO** P_1 **END** $\rightsquigarrow$ Erweiterung von $M(P_1)$

WHILE-Programme & Turing-Maschinen

Theorem

Jede WHILE-berechenbare Funktionen ist Turing-berechenbar.

Beweis (Skizze)

Bauen TM $M(P)$ durch Induktion über die „Termstruktur" eines WHILE-Programms P:

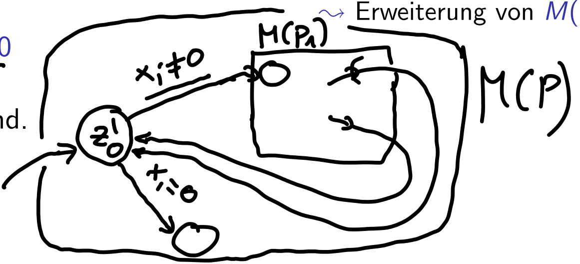
Fall 1: $P = x_i := x_j \pm c$ $\rightsquigarrow$ klar Turing-berechenbar (vgl. Binärzahl-Inkrementierer)

Fall 2: $P = P_1; P_2$ $\rightsquigarrow$ Hintereinanderschaltung von $M(P_1)$ und $M(P_2)$

Identifiziere Endzustände von $M(P_1)$ mit Startzustand von $M(P_2)$.

Fall 3: $P =$ **WHILE** $x_i \neq 0$ **DO** P_1 **END** $\rightsquigarrow$ Erweiterung von $M(P_1)$

1. neuer Startzustand z_0', der prüft ob $x_i \neq 0$
   - ja $\rightsquigarrow$ Wechsel in Startzustand von M_1.
   - nein $\rightsquigarrow$ Stoppe in einem neuen Endzustand.



WHILE-Programme & Turing-Maschinen

Theorem

Jede WHILE-berechenbare Funktionen ist Turing-berechenbar.

Beweis (Skizze)

Bauen TM $M(P)$ durch Induktion über die „Termstruktur“ eines WHILE-Programms P:

Fall 1: $P = x_i := x_j \pm c$ $\rightsquigarrow$ klar Turing-berechenbar (vgl. Binärzahl-Inkrementierer)

Fall 2: $P = P_1; P_2$ $\rightsquigarrow$ Hintereinanderschaltung von $M(P_1)$ und $M(P_2)$

Identifiziere Endzustände von $M(P_1)$ mit Startzustand von $M(P_2)$.

Fall 3: $P = \textbf{WHILE } x_i \neq 0 \textbf{ DO } P_1 \textbf{ END}$ $\rightsquigarrow$ Erweiterung von $M(P_1)$

1. neuer Startzustand z_0', der prüft ob $x_i \neq 0$
   ja $\rightsquigarrow$ Wechsel in Startzustand von M_1.
   nein $\rightsquigarrow$ Stoppe in einem neuen Endzustand.
2. Identifiziere Endzustände von M_1 mit z_0'.

GOTO-Programme

Quelle: pixabay.com/en/mountain-goats-jumping-leaping-1156056/

GOTO-Programme I

GOTO-Programme $\hat{=}$ Marken und Anweisungen:

$$\begin{aligned} P = \quad & M_1 : A_1; \\ & M_2 : A_2; \\ & \quad\vdots \\ & M_k : A_k \end{aligned}$$

Konvention: Nicht benutzte Marken weglassen!

GOTO-Programme I

GOTO-Programme $\hat{=}$ Marken und Anweisungen:

$$P = \begin{array}{l} M_1 : A_1; \\ M_2 : A_2; \\ \vdots \\ M_k : A_k \end{array}$$

Konvention: Nicht benutzte Marken weglassen!

Syntax: Mögliche Anweisungen A_i:

- $x_i := x_j \pm c$
- **GOTO** M_i
- **IF** $x_i = c$ **THEN GOTO** M_i

Semantik klar!

GOTO-Programme I

GOTO-Programme $\triangleq$ Marken und Anweisungen:

$$\begin{aligned} P = \quad & M_1 : A_1; \\ & M_2 : A_2; \\ & \vdots \\ & M_k : A_k \end{aligned}$$

Konvention: Nicht benutzte Marken weglassen!

Syntax: Mögliche Anweisungen A_i:

- $x_i := x_j \pm c$
- **GOTO** M_i
- **IF** $x_i = c$ **THEN GOTO** M_i

Semantik klar!

Definition

GOTO-Berechenbarkeit analog zu WHILE-Berechenbarkeit.

GOTO-Programme I

GOTO-Programme $\hat{=}$ Marken und Anweisungen:

$$P = \begin{aligned} M_1 &: A_1; \\ M_2 &: A_2; \\ &\vdots \\ M_k &: A_k \end{aligned}$$

Konvention: Nicht benutzte Marken weglassen!

Syntax: Mögliche Anweisungen A_i:

- $x_i := x_j \pm c$
- **GOTO** M_i
- **IF** $x_i = c$ **THEN GOTO** M_i

Semantik klar!

Definition

GOTO-Berechenbarkeit analog zu WHILE-Berechenbarkeit.

Theorem

Jede WHILE-berechenbare Funktion ist GOTO-berechenbar.

WHILE ...
WHILE ...

END
END

GOTO-Programme I

GOTO-Programme $\hat{=}$ Marken und Anweisungen:

$$
\begin{aligned}
P = \quad & M_1 : A_1; \\
& M_2 : A_2; \\
& \vdots \\
& M_k : A_k
\end{aligned}
$$

Konvention: Nicht benutzte Marken weglassen!

Syntax: Mögliche Anweisungen A_i:

- $x_i := x_j \pm c$
- **GOTO** M_i
- **IF** $x_i = c$ **THEN GOTO** M_i

Semantik klar!

Definition

GOTO-Berechenbarkeit analog zu WHILE-Berechenbarkeit.

Theorem

Jede WHILE-berechenbare Funktion ist GOTO-berechenbar.

Beweis (simuliere WHILE $x_i \neq 0$ DO P END)

M_1 : **IF** $x_i = 0$ **THEN GOTO** M_2;

M_3: P;

M_4 : **GOTO** M_1;

M_2 : $x_i := 0$

GOTO-Programme II

Theorem

Jede GOTO-berechenbare Funktion ist WHILE-berechenbar.

GOTO-Programme II

Theorem

Jede GOTO-berechenbare Funktion ist WHILE-berechenbar.

nächste Sprungmarke

Beweis

$P = M_1 : A_1; \ldots; M_k : A_k$ ein GOTO-Programm; ungenutzte Variable x_N

GOTO-Programme II

Theorem

Jede GOTO-berechenbare Funktion ist WHILE-berechenbar.

$i \rightsquigarrow A_i$

Beweis

$P = M_1 : A_1; \ldots; M_k : A_k$ ein GOTO-Programm; ungenutzte Variable x_N

$\rightsquigarrow$ WHILE-Programm P_W unter Benutzung des **IF-THEN**-Konstrukts:

LOOP-Berechenbar

IF $x_i = 0$ THEN

$x_N := 1;$
WHILE $x_N \neq 0$ **DO**
IF $x_N = 1$ **THEN** A'_1 **END**;
IF $x_N = 2$ **THEN** A'_2 **END**;
$\ldots$
IF $x_N = k$ **THEN** A'_k **END**;
IF $x_N = k + 1$ **THEN** $x_N := 0$ **END**
END

frage überlegen Sie sich, dass IF $x_i = c$ THEN LOOP ber.

GOTO-Programme II

Theorem

Jede GOTO-berechenbare Funktion ist WHILE-berechenbar.

Beweis

$P = M_1 : A_1; \ldots; M_k : A_k$ ein GOTO-Programm; ungenutzte Variable x_N
$\rightsquigarrow$ WHILE-Programm P_W unter Benutzung des **IF-THEN**-Konstrukts:

$x_N := 1$;
WHILE $x_N \neq 0$ **DO**
IF $x_N = 1$ **THEN** A'_1 **END**;
IF $x_N = 2$ **THEN** A'_2 **END**;
$\ldots$
IF $x_N = k$ **THEN** A'_k **END**;
IF $x_N = k + 1$ **THEN** $x_N := 0$ **END**
END

GOTO-Anweisung A_i	$\rightsquigarrow$	WHILE-Anweisung A'_i
$x_j := x_i \pm c$	$\rightsquigarrow$	$x_j := x_i \pm c$; $x_N := x_N + 1$

GOTO-Programme II

Theorem

Jede GOTO-berechenbare Funktion ist WHILE-berechenbar.

Beweis

$P = M_1 : A_1; \ldots; M_k : A_k$ ein GOTO-Programm; ungenutzte Variable x_N

$\rightsquigarrow$ WHILE-Programm P_W unter Benutzung des **IF-THEN**-Konstrukts:

$x_N := 1$;
WHILE $x_N \neq 0$ **DO**
IF $x_N = 1$ **THEN** A'_1 **END**;
IF $x_N = 2$ **THEN** A'_2 **END**;
$\ldots$
IF $x_N = k$ **THEN** A'_k **END**;
IF $x_N = k+1$ **THEN** $x_N := 0$ **END**
END

GOTO-Anweisung A_i	$\rightsquigarrow$	WHILE-Anweisung A'_i
$x_j := x_i \pm c$	$\rightsquigarrow$	$x_j := x_i \pm c$; $x_N := x_N + 1$
GOTO M_j	$\rightsquigarrow$	$\underline{x_N := j}$

GOTO-Programme II

Theorem

Jede GOTO-berechenbare Funktion ist WHILE-berechenbar.

Beweis

$P = M_1 : A_1; \ldots ; M_k : A_k$ ein GOTO-Programm; ungenutzte Variable x_N

$\leadsto$ WHILE-Programm P_W unter Benutzung des **IF-THEN**-Konstrukts:

$x_N := 1;$
WHILE $x_N \neq 0$ **DO**
IF $x_N = 1$ **THEN** A'_1 **END**;
IF $x_N = 2$ **THEN** A'_2 **END**;
$\ldots$
IF $x_N = k$ **THEN** A'_k **END**;
IF $x_N = k + 1$ **THEN** $x_N := 0$ **END**
END

GOTO-Anweisung A_i	$\leadsto$	WHILE-Anweisung A'_i
$x_j := x_i \pm c$	$\leadsto$	$x_j := x_i \pm c$; $x_N := x_N + 1$
GOTO M_j	$\leadsto$	$x_N := j$
IF $x_j = c$ **THEN GOTO** M_n	$\leadsto$	$x_N := x_N + 1$; **IF** $x_j = c$ **THEN** $x_N := n$

GOTO-Programme II

Theorem

Jede GOTO-berechenbare Funktion ist WHILE-berechenbar.

Beweis

$P = M_1 : A_1; \ldots; M_k : A_k$ ein GOTO-Programm; ungenutzte Variable x_N
$\rightsquigarrow$ WHILE-Programm P_W unter Benutzung des **IF-THEN**-Konstrukts:

$x_N := 1;$
WHILE $x_N \neq 0$ **DO**
IF $x_N = 1$ **THEN** A'_1 **END**;
IF $x_N = 2$ **THEN** A'_2 **END**;
LOOP
...
IF $x_N = k$ **THEN** A'_k **END**;
IF $x_N = k+1$ **THEN** $x_N := 0$ **END**
END

GOTO-Anweisung A_i	$\rightsquigarrow$	WHILE-Anweisung A'_i
$x_j := x_i \pm c$	$\rightsquigarrow$	$x_j := x_i \pm c;$ $x_N := x_N + 1$
GOTO M_j	$\rightsquigarrow$	$x_N := j$
IF $x_j = c$ **THEN GOTO** M_n	$\rightsquigarrow$	$x_N := x_N + 1;$ **IF** $x_j = c$ **THEN** $x_N := n$

Bemerkung: Nur eine einzige WHILE-Schleife im Programm!

GOTO-Programme III

WHILE berechenbar

Theorem

Jede Turing-berechenbare Funktion ist GOTO-berechenbar.

GOTO-Programme III

Theorem

Jede Turing-berechenbare Funktion ist GOTO-berechenbar.

Beweis (Skizze)

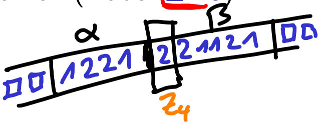
Sei $M = (Z, \Sigma, \Gamma, \delta, z_1, \Box, E)$ mit $\Gamma = \{\Box, \overbrace{1, 2 \ldots, m-1}^{\subseteq \mathbb{N}}\}$, $Z = \{z_1, \ldots, z_k\}$ eine DTM.

Idee: Darstellen der Konfiguration $\alpha z_\ell \beta$ als drei Zahlen (wobei $\boxed{\Box \hat{=} 0}$):

$x_1 = x = [\alpha]_m \rightsquigarrow$ linker Bandinhalt

$x_2 = y = [rev(\beta)]_m \rightsquigarrow$ rechter Bandinhalt

$x_3 = z = \ell \rightsquigarrow$ Zustandsnummer



$\Gamma = \{\Box, 1, 2\}$

$x = [1221]_3 = 52$

$y = [121122]_3 = 449$

$z = 4$

GOTO-Programme III

Theorem

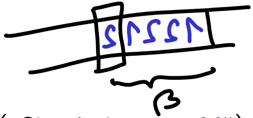
Jede Turing-berechenbare Funktion ist GOTO-berechenbar.

Beweis (Skizze)

Sei $M = (Z, \Sigma, \Gamma, \delta, z_1, \Box, E)$ mit $\Gamma = \{\Box, 1, 2 \ldots, m-1\}$, $Z = \{z_1, \ldots, z_k\}$ eine DTM.
Idee: Darstellen der Konfiguration $\alpha z_\ell \beta$ als drei Zahlen (wobei $\Box \hat{=} 0$):
$x_1 = x = [\alpha]_m \rightsquigarrow$ linker Bandinhalt
$x_2 = y = [rev(\beta)]_m \rightsquigarrow$ rechter Bandinhalt
$x_3 = z = \ell \rightsquigarrow$ Zustandsnummer

Das GOTO-Programm hat nun folgende Gestalt:
Phase 1: „Erzeugung von x, y, z aus der Startkonfiguration“ (LOOP-berechenbar)
Phase 2: „Simulation von M“ *
Phase 3: „Rückübersetzung von $\underline{y}$ in Ausgabe x_0“ (LOOP-berechenbar)

GOTO-Programme IV



$y = [12212]_3$

Beweis (Fortsetzung)

Zu zeigen: GOTO-Programm für Phase 2 („Simulation von M“).

$M_2 : x_4 := y \textbf{ MOD } m;$

GOTO-Programme IV

$0 \hat{=} \square$

Beweis (Fortsetzung)

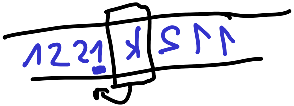
Zu zeigen: GOTO-Programm für Phase 2 („Simulation von M“).

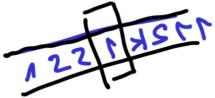
$M_2 : x_4 := y$ **MOD** m;

IF $z = 1$ **AND** $x_4 = 0$ **THEN GOTO** $M_{(1,0)}$;
IF $z = 1$ **AND** $x_4 = 1$ **THEN GOTO** $M_{(1,1)}$;
...
IF $z = k$ **AND** $x_4 = m - 1$ **THEN GOTO** $M_{(k,m-1)}$;

$M_{(1,0)}$: Simulation von $\delta(z_1, \square)$; **GOTO** M_2;
$M_{(1,1)}$: Simulation von $\delta(z_1, 1)$; **GOTO** M_2;
...
$M_{(k,m-1)}$: Simulation von $\delta(z_k, m - 1)$; **GOTO** M_2;

GOTO-Programme IV





Beweis (Fortsetzung)

Zu zeigen: GOTO-Programm für Phase 2 („Simulation von M“).

M_2 : $x_4 := y$ **MOD** m;

IF $z = 1$ **AND** $x_4 = 0$ **THEN GOTO** $M_{(1,0)}$;

IF $z = 1$ **AND** $x_4 = 1$ **THEN GOTO** $M_{(1,1)}$;

...

IF $z = k$ **AND** $x_4 = m - 1$ **THEN GOTO** $M_{(k,m-1)}$;

$y = 112k1$

$x = 122$

$M_{(1,0)}$: Simulation von $\delta(z_1, \square)$; **GOTO** M_2;

$M_{(1,1)}$: Simulation von $\delta(z_1, 1)$; **GOTO** M_2;

...

$M_{(k,m-1)}$: Simulation von $\delta(z_k, m-1)$; **GOTO** M_2;

Simulation von δ

$\delta(z_\ell, i) = (z_j, k, L)$

1 2 3

1 $z := j$;

2 $y := y$ **DIV** m;

$y := y \cdot m + k$;

3 $y := y \cdot m + (x$ **MOD** $m)$;

$x := x$ **DIV** m

$z_j \in E \rightsquigarrow$ **GOTO** Phase 3

$\delta(z_\ell, i) = \bot \rightsquigarrow$ ~~$\rightsquigarrow$ Endlosschleife~~

GOTO-Programme IV

Beweis (Fortsetzung)

Zu zeigen: GOTO-Programm für Phase 2 („Simulation von M“).

$M_2 : x_4 := y$ **MOD** m;
IF $z = 1$ **AND** $x_4 = 0$ **THEN GOTO** $M_{(1,0)}$;
IF $z = 1$ **AND** $x_4 = 1$ **THEN GOTO** $M_{(1,1)}$;
...
IF $z = k$ **AND** $x_4 = m - 1$ **THEN GOTO** $M_{(k,m-1)}$;

$M_{(1,0)}$: Simulation von $\delta(z_1, \square)$; **GOTO** M_2;
$M_{(1,1)}$: Simulation von $\delta(z_1, 1)$; **GOTO** M_2;
...
$M_{(k,m-1)}$: Simulation von $\delta(z_k, m - 1)$; **GOTO** M_2;

Simulation von δ

$\delta(z_\ell, i) = (z_j, k, L)$

$z := j$;
$y := y$ **DIV** m;
$y := y \cdot m + k$;
$y := y \cdot m + (x$ **MOD** $m)$;
$x := x$ **DIV** m

$z_j \in E \rightsquigarrow$ **GOTO** Phase 3

$\delta(z_\ell, i) = \perp$
$\rightsquigarrow$ Endlosschleife

Fazit dieses Kapitels:

WHILE-Berechenbarkeit $\equiv$ GOTO-Berechenbarkeit $\equiv$ Turing-Berechenbarkeit.