

Berechenbarkeit

Vorlesung 6: Rekursive Funktionen I

22. Mai 2025

Termine — Modul Berechenbarkeit

ÜBUNGEN	VORLESUNG
20.5. Übung 3 A-Woche	22.5. Rekursion I (Übungsblatt 4)
27.5. Übung 4 B-Woche	29.5. _____
3.6. Übung 4 A-Woche	5.6. Rekursion II (Übungsblatt 5)
10.6. Übung 5 B-Woche (Montag Feiertag)	12.6. Entscheidbarkeit
17.6. Übung 5 A-Woche	19.6. Unentscheidbarkeit (Übungsblatt 6)
24.6. Übung 6 B-Woche	26.6. Spez. Probleme
1.7. Übung 6 A-Woche	3.7. Klasse P
8.7. Abschlussübung beide Wochen	10.7. NP-Vollständigkeit

Prüfung

Prüfung

- schriftliche Klausur, 60 min Bearbeitungszeit
- erlaubte Hilfsmittel: 1 DIN-A4-Blatt mit Notizen
(beschrieben, bedruckt, etc.)

Mittwoch, 16. Juli 2025 von 13:30–14:30 Uhr
in AudiMax & Hs. 9

Wiederholung — Berechenbarkeit

Definition (§4.8 Turing-Berechenbarkeit; *Turing-computability*)

Partielle Funktion $f: \mathbb{N}^k \dashrightarrow \mathbb{N}$ **Turing-berechenbar**
falls deterministische TM \mathcal{M} mit $\text{bin}(f) = T(\mathcal{M})$ existiert

Definition (§4.15 & §5.18 Loop/While-Berechenbarkeit)

Partielle Funktion $f: \mathbb{N}^k \dashrightarrow \mathbb{N}$ **Loop/While-berechenbar**
falls Loop/While-Programm P mit $f = |P|_k$ existiert

Bedingte Iteration

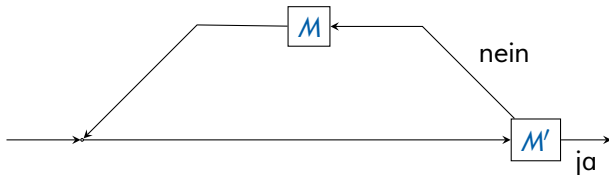
Notizen

- Turing-Berechenbarkeit benötigt deterministische TM
- Determinismus nicht erhalten unter Vereinigung & Iteration
- **Bedingte Iteration** = Abbruch Iteration bei Vorliegen Eigenschaft (bedingter Schleifenabbruch)

Bedingte Iteration

Notizen

- Turing-Berechenbarkeit benötigt deterministische TM
- Determinismus nicht erhalten unter Vereinigung & Iteration
- **Bedingte Iteration** = Abbruch Iteration bei Vorliegen Eigenschaft (bedingter Schleifenabbruch)



Bedingte Iteration

§6.1 Definition (bedingte Iteration; *conditional iteration*)

Seien $f: \Gamma^* \dashrightarrow \Gamma^*$ und $\chi: \Gamma^* \dashrightarrow \{0, 1\}$ partielle Funktionen.

Bedingte Iteration $f_\chi: \Gamma^* \dashrightarrow \Gamma^*$ ist für alle $w \in \Gamma^*$

$$f_\chi^*(w) = \begin{cases} f^t(w) & \text{falls } t \in \mathbb{N} \text{ existiert mit} \\ & \chi(f^t(w)) = 1 \text{ und } \chi(f^s(w)) = 0 \text{ für alle } s < t \\ \text{undef} & \text{sonst} \end{cases}$$

Bedingte Iteration

§6.2 Theorem

Seien $f: \Gamma^* \dashrightarrow \Gamma^*$ und $\chi: \Gamma^* \dashrightarrow \{0, 1\}$ Turing-berechenbar.
Dann bedingte Iteration $f_{\chi}^*: \Gamma^* \dashrightarrow \Gamma^*$ Turing-berechenbar

Bedingte Iteration

§6.2 Theorem

Seien $f: \Gamma^* \dashrightarrow \Gamma^*$ und $\chi: \Gamma^* \dashrightarrow \{0, 1\}$ Turing-berechenbar.
Dann bedingte Iteration $f_{\chi}^*: \Gamma^* \dashrightarrow \Gamma^*$ Turing-berechenbar

Beweisansatz

1. Normierte det. TM M und M' für f und χ ,
wobei M' Band wiederherstellt und statt
Ausgabe 1/0 in Zustand q'_+/q'_- wechselt
2. Akzeptieren in q'_+ (ja-Zweig)

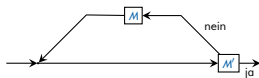
Bedingte Iteration

§6.2 Theorem

Seien $f: \Gamma^* \dashrightarrow \Gamma^*$ und $\chi: \Gamma^* \dashrightarrow \{0, 1\}$ Turing-berechenbar.
Dann bedingte Iteration $f_{\chi}^*: \Gamma^* \dashrightarrow \Gamma^*$ Turing-berechenbar

Beweisansatz

1. Normierte det. TM M und M' für f und χ ,
wobei M' Band wiederherstellt und statt
Ausgabe 1/0 in Zustand q'_+/q'_- wechselt
2. Akzeptieren in q'_+ (ja-Zweig)
3. Starten M in q'_- (nein-Zweig)
4. Starten M' im akz. Zustand von M



Verzweigung

§6.2 Theorem

Seien $f: \Gamma^* \dashrightarrow \Gamma^*$ und $\chi: \Gamma^* \dashrightarrow \{0,1\}$ Turing-berechenbar.
Dann bedingte Iteration $f_{\chi}^*: \Gamma^* \dashrightarrow \Gamma^*$ Turing-berechenbar

Verzweigung

§6.2 Theorem

Seien $f: \Gamma^* \dashrightarrow \Gamma^*$ und $\chi: \Gamma^* \dashrightarrow \{0, 1\}$ Turing-berechenbar.
Dann bedingte Iteration $f_\chi^*: \Gamma^* \dashrightarrow \Gamma^*$ Turing-berechenbar

Beweis

Seien $M = (Q, \Gamma', \Gamma, \Delta, \square, q_0, q_+, q_-)$ und
 $M' = (Q', \Gamma', \Gamma, \Delta', \square, q'_0, q'_+, q'_-)$ normierte det. TM mit $\Gamma' = \Gamma_M$,
 $T(M) = f$ und $T(M') = \chi$.

Verzweigung

§6.2 Theorem

Seien $f: \Gamma^* \dashrightarrow \Gamma^*$ und $\chi: \Gamma^* \dashrightarrow \{0, 1\}$ Turing-berechenbar.
Dann bedingte Iteration $f_\chi^*: \Gamma^* \dashrightarrow \Gamma^*$ Turing-berechenbar

Beweis

Seien $M = (Q, \Gamma', \Gamma, \Delta, \square, q_0, q_+, q_-)$ und
 $M' = (Q', \Gamma', \Gamma, \Delta', \square, q'_0, q'_+, q'_-)$ normierte det. TM mit $\Gamma' = \Gamma_M$,
 $T(M) = f$ und $T(M') = \chi$. Anpassung M' für Wiederherstellung
Eingabe und Akzeptanz statt Ausgabe 1 und Ablehnung statt
Ausgabe 0. O.B.d.A. sei $Q \cap Q' = \emptyset$. Wir konstruieren det.
TM $N = (Q \cup Q', \Gamma', \Gamma, \Delta \cup \Delta' \cup R, \square, q_0, q'_+, q_-)$

$$R = \{(q'_-, \gamma) \rightarrow (q_0, \gamma, \diamond) \mid \gamma \in \Gamma\} \cup \{(q_+, \gamma) \rightarrow (q'_0, \gamma, \diamond) \mid \gamma \in \Gamma\}$$

Dann $T(N) = T(M)_{T(M')}^* = f_\chi^*$



Verzweigung

§6.2 Theorem

Seien $f: \Gamma^* \dashrightarrow \Gamma^*$ und $\chi: \Gamma^* \dashrightarrow \{0, 1\}$ Turing-berechenbar.
Dann bedingte Iteration $f_\chi^*: \Gamma^* \dashrightarrow \Gamma^*$ Turing-berechenbar

Beweis

Seien $M = (Q, \Gamma', \Gamma, \Delta, \square, q_0, q_+, q_-)$ und
 $M' = (Q', \Gamma', \Gamma, \Delta', \square, q'_0, q'_+, q'_-)$ normierte det. TM mit $\Gamma' = \Gamma_M$,
 $T(M) = f$ und $T(M') = \chi$. Anpassung M' für Wiederherstellung
Eingabe und Akzeptanz statt Ausgabe 1 und Ablehnung statt
Ausgabe 0. O.B.d.A. sei $Q \cap Q' = \emptyset$. Wir konstruieren det.
TM $N = (Q \cup Q', \Gamma', \Gamma, \Delta \cup \Delta' \cup R, \square, q'_0, q'_+, q_-)$

$$R = \{(q'_-, \gamma) \rightarrow (q_0, \gamma, \diamond) \mid \gamma \in \Gamma\} \cup \{(q_+, \gamma) \rightarrow (q'_0, \gamma, \diamond) \mid \gamma \in \Gamma\}$$

Dann $T(N) = T(M)_{T(M')}^* = f_\chi^*$



While-Berechenbarkeit

§6.3 Theorem

Jede While-berechenbare partielle Funktion ist Turing-berechenbar

While-Berechenbarkeit

§6.3 Theorem

Jede While-berechenbare partielle Funktion ist Turing-berechenbar

Beweisskizze (1/4)

Sei P While-Programm mit $\max \text{var}(P) = n$. Nutze 1 Band pro Variable und speichere x_i auf Band i . Wir konstruieren normierte det. TM

- M_{start} kopiert Startwerte auf korrekte Bänder

While-Berechenbarkeit

§6.3 Theorem

Jede While-berechenbare partielle Funktion ist Turing-berechenbar

Beweisskizze (1/4)

Sei P While-Programm mit $\max \text{var}(P) = n$. Nutze 1 Band pro Variable und speichere x_i auf Band i . Wir konstruieren normierte det. TM

- M_{start} kopiert Startwerte auf korrekte Bänder
- Induktiv per Definition While-Programm

While-Berechenbarkeit

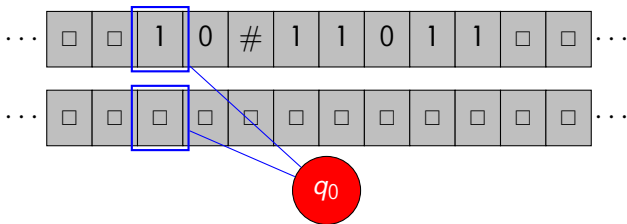
§6.3 Theorem

Jede While-berechenbare partielle Funktion ist Turing-berechenbar

Beweisskizze (1/4)

Sei P While-Programm mit $\max \text{var}(P) = n$. Nutze 1 Band pro Variable und speichere x_i auf Band i . Wir konstruieren normierte det. TM

- M_{start} kopiert Startwerte auf korrekte Bänder
- Induktiv per Definition While-Programm



While-Berechenbarkeit

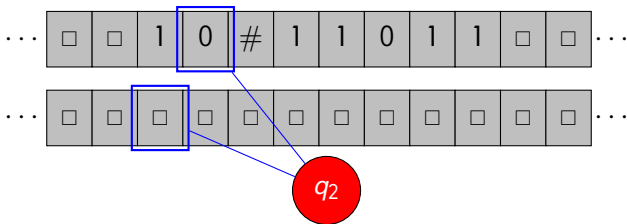
§6.3 Theorem

Jede While-berechenbare partielle Funktion ist Turing-berechenbar

Beweisskizze (1/4)

Sei P While-Programm mit $\max \text{var}(P) = n$. Nutze 1 Band pro Variable und speichere x_i auf Band i . Wir konstruieren normierte det. TM

- M_{start} kopiert Startwerte auf korrekte Bänder
- Induktiv per Definition While-Programm



While-Berechenbarkeit

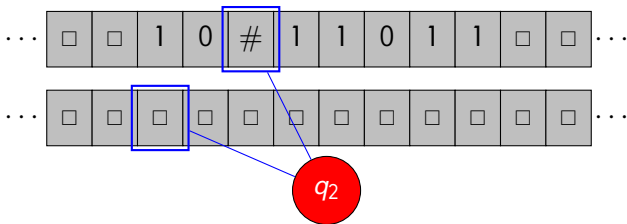
§6.3 Theorem

Jede While-berechenbare partielle Funktion ist Turing-berechenbar

Beweisskizze (1/4)

Sei P While-Programm mit $\max \text{var}(P) = n$. Nutze 1 Band pro Variable und speichere x_i auf Band i . Wir konstruieren normierte det. TM

- M_{start} kopiert Startwerte auf korrekte Bänder
- Induktiv per Definition While-Programm



While-Berechenbarkeit

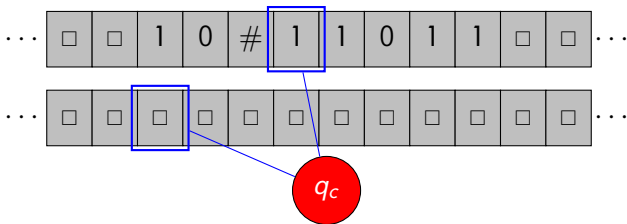
§6.3 Theorem

Jede While-berechenbare partielle Funktion ist Turing-berechenbar

Beweisskizze (1/4)

Sei P While-Programm mit $\max \text{var}(P) = n$. Nutze 1 Band pro Variable und speichere x_i auf Band i . Wir konstruieren normierte det. TM

- M_{start} kopiert Startwerte auf korrekte Bänder
- Induktiv per Definition While-Programm



While-Berechenbarkeit

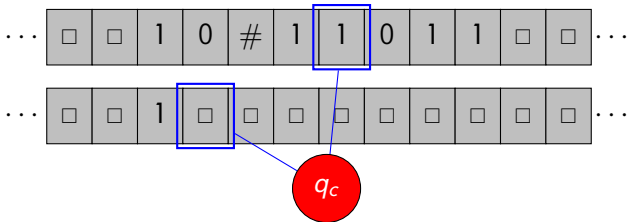
§6.3 Theorem

Jede While-berechenbare partielle Funktion ist Turing-berechenbar

Beweisskizze (1/4)

Sei P While-Programm mit $\max \text{var}(P) = n$. Nutze 1 Band pro Variable und speichere x_i auf Band i . Wir konstruieren normierte det. TM

- M_{start} kopiert Startwerte auf korrekte Bänder
- Induktiv per Definition While-Programm



While-Berechenbarkeit

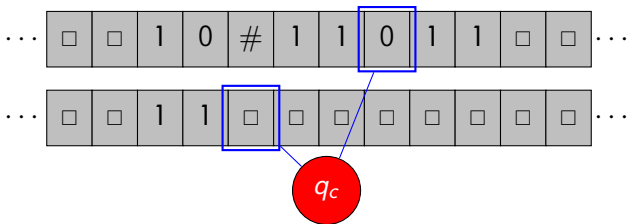
§6.3 Theorem

Jede While-berechenbare partielle Funktion ist Turing-berechenbar

Beweisskizze (1/4)

Sei P While-Programm mit $\max \text{var}(P) = n$. Nutze 1 Band pro Variable und speichere x_i auf Band i . Wir konstruieren normierte det. TM

- M_{start} kopiert Startwerte auf korrekte Bänder
- Induktiv per Definition While-Programm



While-Berechenbarkeit

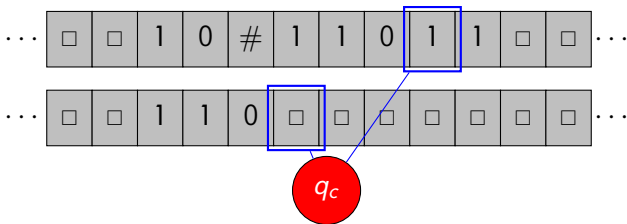
§6.3 Theorem

Jede While-berechenbare partielle Funktion ist Turing-berechenbar

Beweisskizze (1/4)

Sei P While-Programm mit $\max \text{var}(P) = n$. Nutze 1 Band pro Variable und speichere x_i auf Band i . Wir konstruieren normierte det. TM

- M_{start} kopiert Startwerte auf korrekte Bänder
- Induktiv per Definition While-Programm



While-Berechenbarkeit

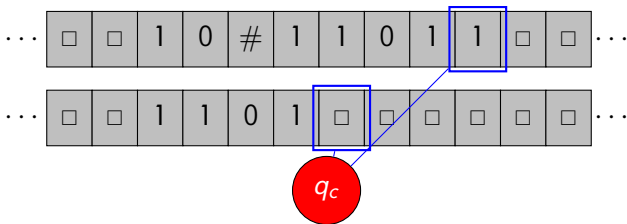
§6.3 Theorem

Jede While-berechenbare partielle Funktion ist Turing-berechenbar

Beweisskizze (1/4)

Sei P While-Programm mit $\max \text{var}(P) = n$. Nutze 1 Band pro Variable und speichere x_i auf Band i . Wir konstruieren normierte det. TM

- M_{start} kopiert Startwerte auf korrekte Bänder
- Induktiv per Definition While-Programm



While-Berechenbarkeit

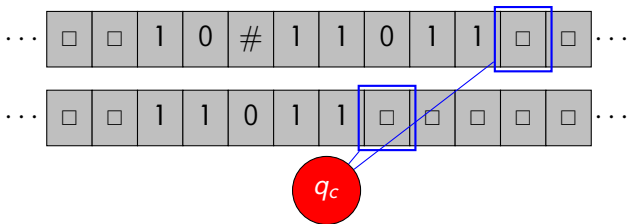
§6.3 Theorem

Jede While-berechenbare partielle Funktion ist Turing-berechenbar

Beweisskizze (1/4)

Sei P While-Programm mit $\max \text{var}(P) = n$. Nutze 1 Band pro Variable und speichere x_i auf Band i . Wir konstruieren normierte det. TM

- M_{start} kopiert Startwerte auf korrekte Bänder
- Induktiv per Definition While-Programm



While-Berechenbarkeit

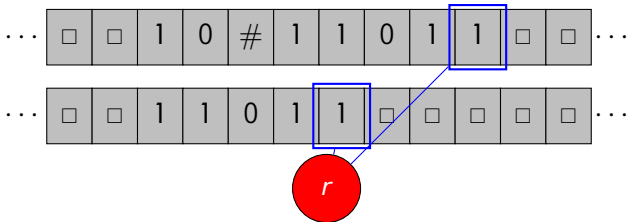
§6.3 Theorem

Jede While-berechenbare partielle Funktion ist Turing-berechenbar

Beweisskizze (1/4)

Sei P While-Programm mit $\max \text{var}(P) = n$. Nutze 1 Band pro Variable und speichere x_i auf Band i . Wir konstruieren normierte det. TM

- M_{start} kopiert Startwerte auf korrekte Bänder
- Induktiv per Definition While-Programm



While-Berechenbarkeit

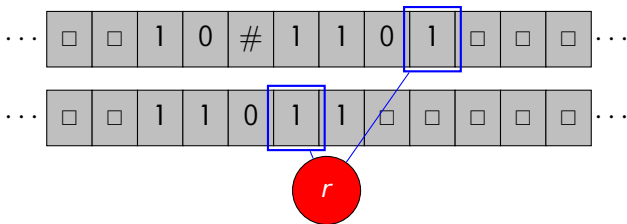
§6.3 Theorem

Jede While-berechenbare partielle Funktion ist Turing-berechenbar

Beweisskizze (1/4)

Sei P While-Programm mit $\max \text{var}(P) = n$. Nutze 1 Band pro Variable und speichere x_i auf Band i . Wir konstruieren normierte det. TM

- M_{start} kopiert Startwerte auf korrekte Bänder
- Induktiv per Definition While-Programm



While-Berechenbarkeit

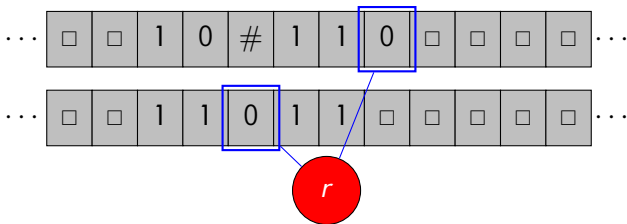
§6.3 Theorem

Jede While-berechenbare partielle Funktion ist Turing-berechenbar

Beweisskizze (1/4)

Sei P While-Programm mit $\max \text{var}(P) = n$. Nutze 1 Band pro Variable und speichere x_i auf Band i . Wir konstruieren normierte det. TM

- M_{start} kopiert Startwerte auf korrekte Bänder
- Induktiv per Definition While-Programm



While-Berechenbarkeit

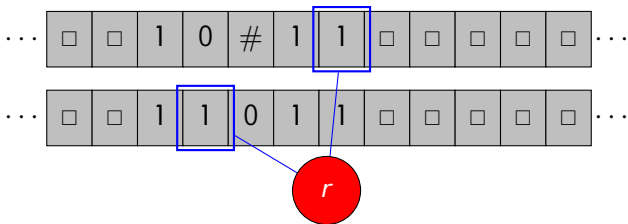
§6.3 Theorem

Jede While-berechenbare partielle Funktion ist Turing-berechenbar

Beweisskizze (1/4)

Sei P While-Programm mit $\max \text{var}(P) = n$. Nutze 1 Band pro Variable und speichere x_i auf Band i . Wir konstruieren normierte det. TM

- M_{start} kopiert Startwerte auf korrekte Bänder
- Induktiv per Definition While-Programm



While-Berechenbarkeit

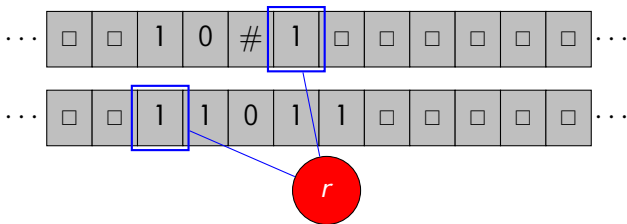
§6.3 Theorem

Jede While-berechenbare partielle Funktion ist Turing-berechenbar

Beweisskizze (1/4)

Sei P While-Programm mit $\max \text{var}(P) = n$. Nutze 1 Band pro Variable und speichere x_i auf Band i . Wir konstruieren normierte det. TM

- M_{start} kopiert Startwerte auf korrekte Bänder
- Induktiv per Definition While-Programm



While-Berechenbarkeit

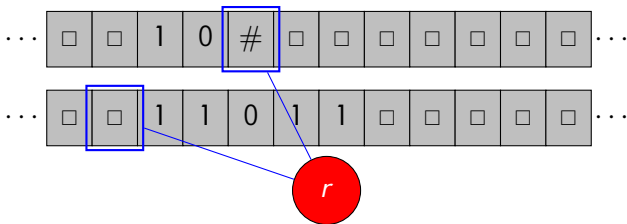
§6.3 Theorem

Jede While-berechenbare partielle Funktion ist Turing-berechenbar

Beweisskizze (1/4)

Sei P While-Programm mit $\max \text{var}(P) = n$. Nutze 1 Band pro Variable und speichere x_i auf Band i . Wir konstruieren normierte det. TM

- M_{start} kopiert Startwerte auf korrekte Bänder
- Induktiv per Definition While-Programm



While-Berechenbarkeit

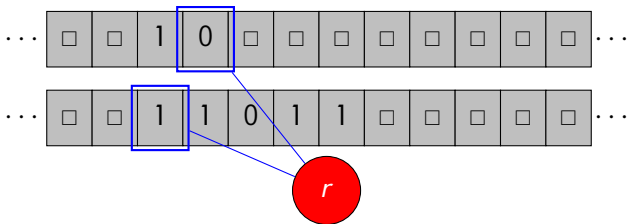
§6.3 Theorem

Jede While-berechenbare partielle Funktion ist Turing-berechenbar

Beweisskizze (1/4)

Sei P While-Programm mit $\max \text{var}(P) = n$. Nutze 1 Band pro Variable und speichere x_i auf Band i . Wir konstruieren normierte det. TM

- M_{start} kopiert Startwerte auf korrekte Bänder
- Induktiv per Definition While-Programm



While-Berechenbarkeit

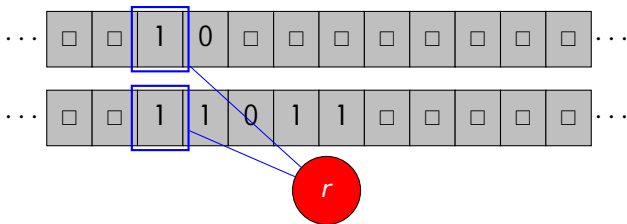
§6.3 Theorem

Jede While-berechenbare partielle Funktion ist Turing-berechenbar

Beweisskizze (1/4)

Sei P While-Programm mit $\max \text{var}(P) = n$. Nutze 1 Band pro Variable und speichere x_i auf Band i . Wir konstruieren normierte det. TM

- M_{start} kopiert Startwerte auf korrekte Bänder
- Induktiv per Definition While-Programm



While-Berechenbarkeit

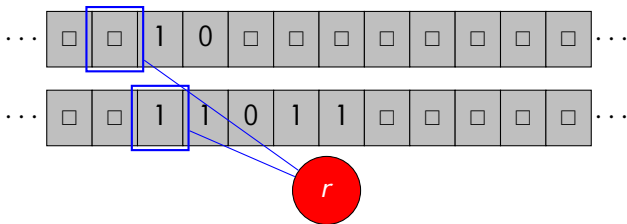
§6.3 Theorem

Jede While-berechenbare partielle Funktion ist Turing-berechenbar

Beweisskizze (1/4)

Sei P While-Programm mit $\max \text{var}(P) = n$. Nutze 1 Band pro Variable und speichere x_i auf Band i . Wir konstruieren normierte det. TM

- M_{start} kopiert Startwerte auf korrekte Bänder
- Induktiv per Definition While-Programm



While-Berechenbarkeit

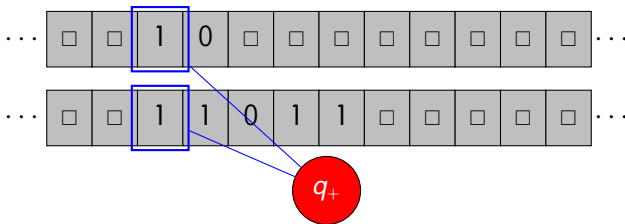
§6.3 Theorem

Jede While-berechenbare partielle Funktion ist Turing-berechenbar

Beweisskizze (1/4)

Sei P While-Programm mit $\max \text{var}(P) = n$. Nutze 1 Band pro Variable und speichere x_i auf Band i . Wir konstruieren normierte det. TM

- M_{start} kopiert Startwerte auf korrekte Bänder
- Induktiv per Definition While-Programm



While-Berechenbarkeit

Beweisskizze (2/4)

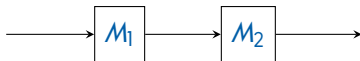
- Sei P Zuweisung $x_i = x_\ell + z$
- Simuliert durch
 1. Kopier-TM $M_{\ell \rightarrow i}$ kopiert Band ℓ auf Band i
 2. z mal Inkrement- oder Dekrement-TM auf Band i



While-Berechenbarkeit

Beweisskizze (3/4)

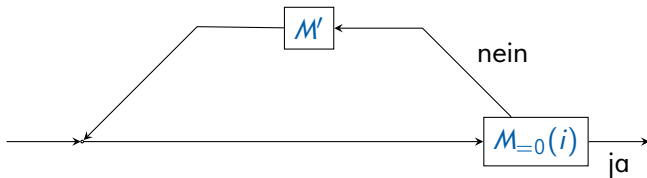
- Sei $P = P_1 ; P_2$
- Simuliert durch Verkettung zugeh. det. TM M_1 und M_2



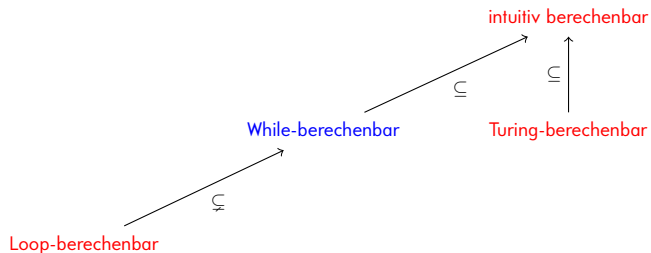
While-Berechenbarkeit

Beweisskizze (4/4)

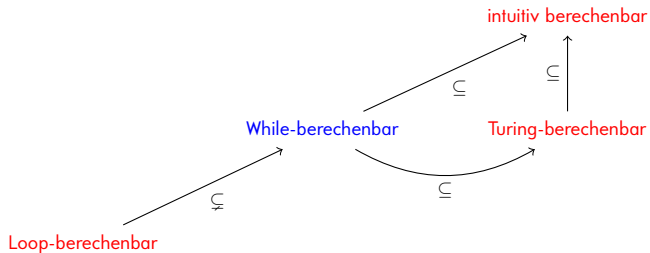
- Sei $P = \text{WHILE}(x_i \neq 0) \{P'\}$
- Simuliert durch
 1. Sei M' det. TM für P'
 2. Sei $M_{=0}$ det. TM für Gleichheit mit 0
 3. Bedingte Iteration von M' mit Bedingung $M_{=0}(i)$



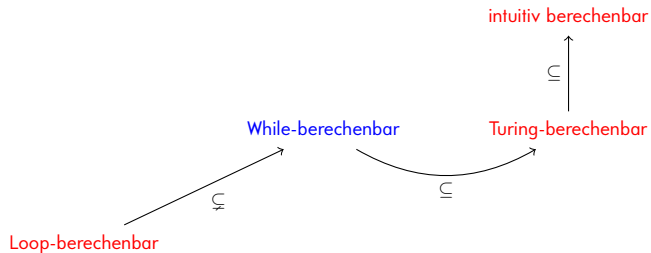
While-Berechenbarkeit



While-Berechenbarkeit



While-Berechenbarkeit



Simulation Turingmaschine

Ansatz

- Simulation det. TM $(Q, \Sigma, \Gamma, \Delta, \square, q_0, q_+, q_-)$ durch While-Programm
- Kodierung Zustand & Band benötigt
- Globalsituation $u q w$ in 3 Variablen: x_1 für w ; x_2 für q ; x_3 für u^R
- Nummerierung Zustände per Bijektion $h_Q: Q \rightarrow \{0, \dots, |Q| - 1\}$ mit $h_Q(q_+) = 0$ und $h_Q(q_-) = 1$

Simulation Turingmaschine

Ansatz

- Simulation det. TM $(Q, \Sigma, \Gamma, \Delta, \square, q_0, q_+, q_-)$ durch While-Programm
- Kodierung Zustand & Band benötigt
- Globalsituation $u q w$ in 3 Variablen: x_1 für w ; x_2 für q ; x_3 für u^R
- Nummerierung Zustände per Bijektion $h_Q: Q \rightarrow \{0, \dots, |Q| - 1\}$ mit $h_Q(q_+) = 0$ und $h_Q(q_-) = 1$

Beispiel

- $Q = \{q_0, q, q_+, q_-\}$ kodiert via

$$q_0 \mapsto 3 \quad q \mapsto 2 \quad q_+ \mapsto 0 \quad q_- \mapsto 1$$

Simulation Turingmaschine

Stellenwertsystem zur Basis $n = |\Gamma|$

- Nummerierung Symbole aus Γ per Bijektion $h_\Gamma: \Gamma \rightarrow \{0, \dots, n-1\}$ mit $h_\Gamma(\square) = 0$
- Kodiere Wort $w \in \Gamma^*$ im inversen Stellenwertsystem zur Basis n

$$\text{code}_{h_\Gamma}(\gamma_1 \cdots \gamma_\ell) = \sum_{i=1}^{\ell} h_\Gamma(\gamma_i) \cdot n^{i-1}$$

Simulation Turingmaschine

Stellenwertsystem zur Basis $n = |\Gamma|$

- Nummerierung Symbole aus Γ per Bijektion $h_\Gamma: \Gamma \rightarrow \{0, \dots, n-1\}$ mit $h_\Gamma(\square) = 0$
- Kodiere Wort $w \in \Gamma^*$ im inversen Stellenwertsystem zur Basis n

$$\text{code}_{h_\Gamma}(\gamma_1 \cdots \gamma_\ell) = \sum_{i=1}^{\ell} h_\Gamma(\gamma_i) \cdot n^{i-1}$$

Beispiel

- $\Gamma = \{\square, a, b\}$ mit $h_\Gamma(\square) = 0$, $h_\Gamma(a) = 1$ und $h_\Gamma(b) = 2$
- $w = abab$

$$\begin{aligned}\text{code}_{h_\Gamma}(w) &= h_\Gamma(a) \cdot 3^0 + h_\Gamma(b) \cdot 3^1 + h_\Gamma(a) \cdot 3^2 + h_\Gamma(b) \cdot 3^3 \\ &= 1 + 2 \cdot 3 + 1 \cdot 9 + 2 \cdot 27 = 1 + 6 + 9 + 54 = 70\end{aligned}$$

Simulation Turingmaschine

Rechnen im Stellenwertsystem zur Basis n

- 1. Zeichen Kodierung k ist $h_{\Gamma}^{-1}(k \bmod n)$

$$h_{\Gamma}^{-1}(\text{code}_{h_{\Gamma}}(\gamma w) \bmod n) = \gamma$$

Schreibweise: $\text{TOP}(x_i) = x_i \bmod n$

Simulation Turingmaschine

Rechnen im Stellenwertsystem zur Basis n

- 1. Zeichen Kodierung k ist $h_r^{-1}(k \bmod n)$

$$h_r^{-1}(\text{code}_{h_r}(\gamma w) \bmod n) = \gamma$$

Schreibweise: **TOP**(x_i) = $x_i \bmod n$

- Entferne 1. Zeichen aus Kodierung k ist $k \text{ DIV } n$

$$\text{code}_{h_r}(\gamma w) \text{ DIV } n = \text{code}_{h_r}(w)$$

Schreibweise: **POP**(x_i) = $x_i \text{ DIV } n$

Simulation Turingmaschine

Rechnen im Stellenwertsystem zur Basis n

- 1. Zeichen Kodierung k ist $h_{\Gamma}^{-1}(k \bmod n)$

$$h_{\Gamma}^{-1}(\text{code}_{h_{\Gamma}}(\gamma w) \bmod n) = \gamma$$

Schreibweise: **TOP**(x_i) = $x_i \bmod n$

- Entferne 1. Zeichen aus Kodierung k ist $k \text{ DIV } n$

$$\text{code}_{h_{\Gamma}}(\gamma w) \text{ DIV } n = \text{code}_{h_{\Gamma}}(w)$$

Schreibweise: **POP**(x_i) = $x_i \text{ DIV } n$

- Einfügen γ als 1. Zeichen in Kodierung k ist $h_{\Gamma}(\gamma) + k \cdot n$

$$h_{\Gamma}(\gamma) + \text{code}_{h_{\Gamma}}(w) \cdot n = \text{code}_{h_{\Gamma}}(\gamma w)$$

Schreibweise: **PUSH**(x_i, z) = $z + x_i \cdot n$

Simulation Turingmaschine

Sei $\Gamma = \{\square, a, b\}$ mit $h_\Gamma(\square) = 0$, $h_\Gamma(a) = 1$ und $h_\Gamma(b) = 2$

Beispiel

- Sei $x_1 = 70$ (Kodierung von "abab")
- $\text{TOP}(x_1) = 70 \bmod 3 = 1$ (entspricht 'a')
- $\text{POP}(x_1) = 70 \text{ DIV } 3 = 23$ (entspricht "bab" [$2 + 1 \cdot 3 + 2 \cdot 3^2$])
- $\text{PUSH}(x_1, 2) = 70 \cdot 3 + 2 = 212$
(entspricht "babab" [$2 + 1 \cdot 3 + 2 \cdot 3^2 + 1 \cdot 3^3 + 2 \cdot 3^4$])

Simulation Turingmaschine

Sei $\Gamma = \{\square, a, b\}$ mit $h_\Gamma(\square) = 0$, $h_\Gamma(a) = 1$ und $h_\Gamma(b) = 2$

Beispiel

- Sei $x_1 = 70$ (Kodierung von "abab")
- $\text{TOP}(x_1) = 70 \bmod 3 = 1$ (entspricht 'a')
- $\text{POP}(x_1) = 70 \text{ DIV } 3 = 23$ (entspricht "bab" [$2 + 1 \cdot 3 + 2 \cdot 3^2$])
- $\text{PUSH}(x_1, 2) = 70 \cdot 3 + 2 = 212$
(entspricht "babab" [$2 + 1 \cdot 3 + 2 \cdot 3^2 + 1 \cdot 3^3 + 2 \cdot 3^4$])

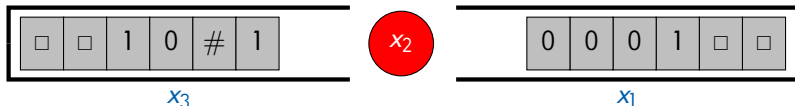
Notiz

- Kellerspeicher mit n Symbolen

Simulation Turingmaschine

Überblick

- Alle Komponenten beisammen
- Kodierung Zustand via h_Q (in x_2)
- Kodierung Band u links des Kopfes als Keller für u^R via h_Γ (in x_3)
- Kodierung sonstiges Band w als Keller via h_Γ (in x_1)



Simulation Turingmaschine

Hauptprogramm

...Kodierung Eingabe in x_1 ...

$x_2 = h_Q(q_0) ; x_3 = 0$

(Startzustand & leeres Band)

WHILE($x_2 > 1$) {

(kein Endzustand)

CASE(x_2 , **TOP**(x_1)) **OF**

(Fallunterscheidung linke Seite Übergang)

 ...

 (q, γ): ... führe (q, γ)-Übergang aus ...

 ...

ELSE { $x_2 = h_Q(q_-)$ }

}

...Dekodierung Band x_1 & Finalprüfung ...

Simulation Turingmaschine

Regelanwendung

Für jeden Übergang $(q, \gamma) \rightarrow (q', \gamma', d) \in \Delta$

$$x_2 = h_Q(q')$$

(Zustandswechsel)

$$x_1 = \text{POP}(x_1)$$

(Entferne Zeichen unter Kopf)

$$x_1 = \text{PUSH}(x_1, h_\Gamma(\gamma'))$$

(Füge neues Zeichen ein)

Simulation Turingmaschine

Regelanwendung

Für jeden Übergang $(q, \gamma) \rightarrow (q', \gamma', d) \in \Delta$

$$x_2 = h_Q(q')$$

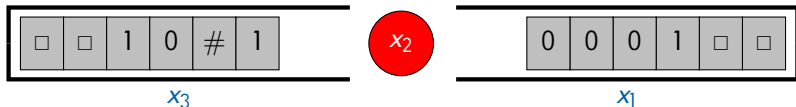
$$x_1 = \text{POP}(x_1)$$

$$x_1 = \text{PUSH}(x_1, h_\Gamma(\gamma'))$$

(Zustandswechsel)

(Entferne Zeichen unter Kopf)

(Füge neues Zeichen ein)



Simulation Turingmaschine

Regelanwendung

Für jeden Übergang $(q, \gamma) \rightarrow (q', \gamma', d) \in \Delta$

$$x_2 = h_Q(q')$$

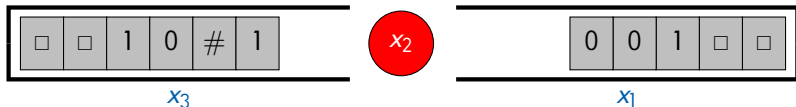
$$x_1 = \text{POP}(x_1)$$

$$x_1 = \text{PUSH}(x_1, h_\Gamma(\gamma'))$$

(Zustandswechsel)

(Entferne Zeichen unter Kopf)

(Füge neues Zeichen ein)



Simulation Turingmaschine

Regelanwendung

Für jeden Übergang $(q, \gamma) \rightarrow (q', \gamma', d) \in \Delta$

$$x_2 = h_Q(q')$$

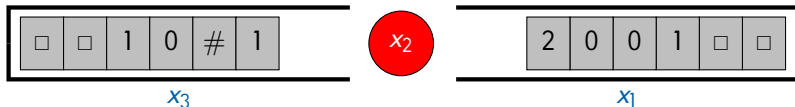
$$x_1 = \text{POP}(x_1)$$

$$x_1 = \text{PUSH}(x_1, h_\Gamma(\gamma'))$$

(Zustandswechsel)

(Entferne Zeichen unter Kopf)

(Füge neues Zeichen ein)



Simulation einer Turingmaschine

Regelanwendung

Falls $d = \triangleleft$ dann zusätzlich

$x_1 = \text{PUSH}(x_1, \text{TOP}(x_3))$

(Füge 1. Zeichen von links ein)

$x_3 = \text{POP}(x_3)$

(Entferne 1. Zeichen von links)

Simulation einer Turingmaschine

Regelanwendung

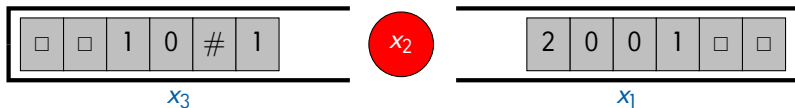
Falls $d = \triangleleft$ dann zusätzlich

$x_1 = \text{PUSH}(x_1, \text{TOP}(x_3))$

$x_3 = \text{POP}(x_3)$

(Füge 1. Zeichen von links ein)

(Entferne 1. Zeichen von links)



Simulation einer Turingmaschine

Regelanwendung

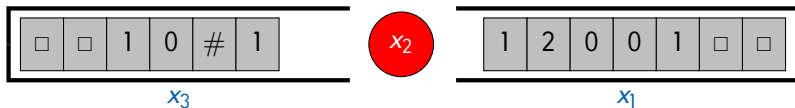
Falls $d = \triangleleft$ dann zusätzlich

$x_1 = \text{PUSH}(x_1, \text{TOP}(x_3))$

$x_3 = \text{POP}(x_3)$

(Füge 1. Zeichen von links ein)

(Entferne 1. Zeichen von links)



Simulation einer Turingmaschine

Regelanwendung

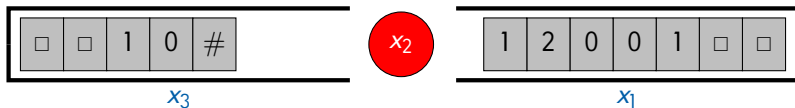
Falls $d = \triangleleft$ dann zusätzlich

$x_1 = \text{PUSH}(x_1, \text{TOP}(x_3))$

$x_3 = \text{POP}(x_3)$

(Füge 1. Zeichen von links ein)

(Entferne 1. Zeichen von links)



Simulation einer Turingmaschine

Regelanwendung

Falls $d = \triangleright$ dann zusätzlich

$x_3 = \text{PUSH}(x_3, \text{TOP}(x_1))$

(Füge 1. Zeichen von rechts ein)

$x_1 = \text{POP}(x_1)$

(Entferne 1. Zeichen von rechts)

Simulation einer Turingmaschine

Regelanwendung

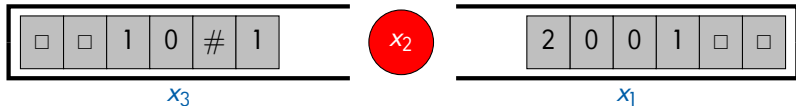
Falls $d = \triangleright$ dann zusätzlich

$x_3 = \text{PUSH}(x_3, \text{TOP}(x_1))$

$x_1 = \text{POP}(x_1)$

(Füge 1. Zeichen von rechts ein)

(Entferne 1. Zeichen von rechts)



Simulation einer Turingmaschine

Regelanwendung

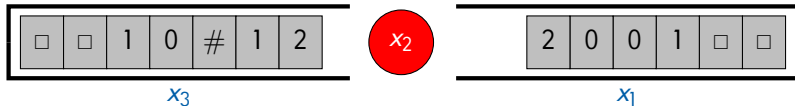
Falls $d = \triangleright$ dann zusätzlich

$x_3 = \text{PUSH}(x_3, \text{TOP}(x_1))$

$x_1 = \text{POP}(x_1)$

(Füge 1. Zeichen von rechts ein)

(Entferne 1. Zeichen von rechts)



Simulation einer Turingmaschine

Regelanwendung

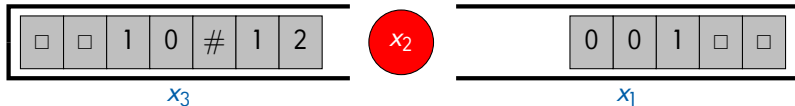
Falls $d = \triangleright$ dann zusätzlich

$x_3 = \text{PUSH}(x_3, \text{TOP}(x_1))$

$x_1 = \text{POP}(x_1)$

(Füge 1. Zeichen von rechts ein)

(Entferne 1. Zeichen von rechts)



Simulation einer Turingmaschine

Banddekodierung für Binärzahl & Finalprüfung

```
IF( $x_3 = 0$  und  $x_2 = 0$ ) {           (teste linken Bandinhalt & Finalzustand)
   $x_4 = 0$  ;  $x_5 = 0$                  (Initialisierung Ausgabewert & Stellenwert)
  WHILE( $x_1 \neq 0$ ) {
    IF( $1 \leq \text{TOP}(x_1) \leq 2$ ) {      (gültiges Bit [0 = □])
       $x_4 = x_4 + (\text{TOP}(x_1) - 1) \cdot 2^{x_5}$   (dekodiere Binärdarstellung)
       $x_5 = x_5 + 1$                       (nächste Stelle)
       $x_1 = \text{POP}(x_1)$                   (entferne erstes Bit)
    } ELSE ... Endlosschleife ...
  }
   $x_1 = x_4$                            (kopiere Ausgabewert)
} ELSE ... Endlosschleife ...
```

While-Berechenbarkeit

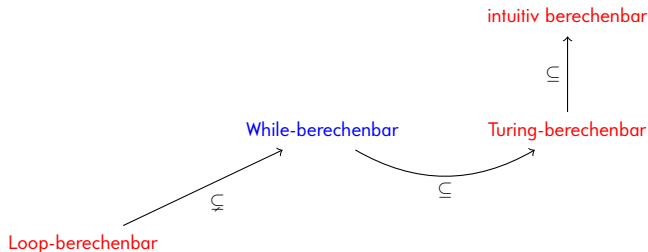
§6.4 Theorem

Jede Turing-berechenbare partielle Funktion ist While-berechenbar

While-Berechenbarkeit

§6.4 Theorem

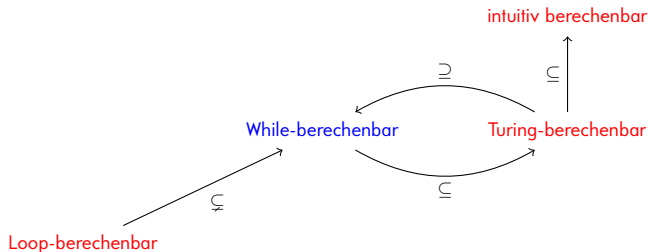
Jede Turing-berechenbare partielle Funktion ist While-berechenbar



While-Berechenbarkeit

§6.4 Theorem

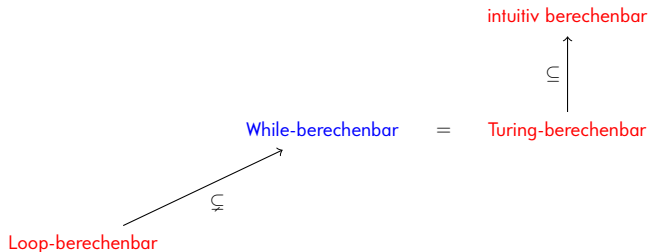
Jede Turing-berechenbare partielle Funktion ist While-berechenbar



While-Berechenbarkeit

§6.4 Theorem

Jede Turing-berechenbare partielle Funktion ist While-berechenbar



Ackermann-Funktion

Definition (§5.1 Ackermann-Funktion; engl. *Ackermann function*)

Für alle $x, y \in \mathbb{N}$ sei

$$a(x, y) = \begin{cases} y + 1 & \text{falls } x = 0 \\ a(x - 1, 1) & \text{falls } x \neq 0 \text{ und } y = 0 \\ a(x - 1, a(x, y - 1)) & \text{sonst} \end{cases}$$

Ackermann-Funktion

Definition (§5.1 Ackermann-Funktion; engl. *Ackermann function*)

Für alle $x, y \in \mathbb{N}$ sei

$$a(x, y) = \begin{cases} y + 1 & \text{falls } x = 0 \\ a(x - 1, 1) & \text{falls } x \neq 0 \text{ und } y = 0 \\ a(x - 1, a(x, y - 1)) & \text{sonst} \end{cases}$$

Ackermann-Funktion While-berechenbar?

Ackermann-Funktion

Kellerspeicher

- Implementiert für n Kellersymbole
- Speicherung Elemente von \mathbb{N} per Binärokodierung über $\{0, 1, \#\}$

Ackermann-Funktion

Kellerspeicher

- Implementiert für n Kellersymbole
- Speicherung Elemente von \mathbb{N} per Binärokodierung über $\{0, 1, \#\}$
- Kellerende markiert durch 4. Symbol

Ackermann-Funktion

Einfügen natürliche Zahl

PUSH($x_i, 2$)

(Trennsymbol einfügen)

$x_\ell = x_k$

(kopiere x_k)

WHILE($x_\ell \neq 0$) {

PUSH($x_i, x_\ell \text{ MOD } 2$)

(letztes Bit speichern)

$x_\ell = x_\ell \text{ DIV } 2$

}

Schreibweise: $x_i = \text{PUSH}_{\mathbb{N}}(x_i, x_k)$

($i \neq k$; x_ℓ unbenutzt)

Ackermann-Funktion

Einfügen natürliche Zahl

PUSH($x_i, 2$)

(Trennsymbol einfügen)

$x_\ell = x_k$

(kopiere x_k)

WHILE($x_\ell \neq 0$) {

PUSH($x_i, x_\ell \text{ MOD } 2$)

(letztes Bit speichern)

$x_\ell = x_\ell \text{ DIV } 2$

}

Schreibweise: $x_i = \text{PUSH}_{\mathbb{N}}(x_i, x_k)$

($i \neq k$; x_ℓ unbenutzt)

Entfernen oberste natürliche Zahl

WHILE(**TOP**(x_i) < 2) { $x_i = \text{POP}$ (x_i) }

(entferne 0/1-Bits)

IF(**TOP**(x_i) = 2) { $x_i = \text{POP}$ (x_i) }

(teste auf & entferne Trennsymbol)

ELSE ... *Endlosschleife* ...

Schreibweise: $x_i = \text{POP}_{\mathbb{N}}(x_i)$

Ackermann-Funktion

Auslesen oberste natürliche Zahl

$x_\ell = 0$

(initialisiere x_ℓ)

WHILE(**TOP**(x_i) < 2) {

(bis Trenn- oder Endesymbol)

$x_\ell = x_\ell \cdot 2 + \mathbf{TOP}(x_i)$

(dekodiere Binärzahl)

$x_i = \mathbf{POP}(x_i)$

(erstes Bit entfernen)

}

IF(**TOP**(x_i) = 2) { $x_i = \mathbf{POP}(x_i)$ }

(teste auf & entferne Trennsymbol)

ELSE ...*Endlosschleife* ...

$x_i = \mathbf{PUSH}_{\mathbb{N}}(x_i, x_\ell)$

(Wert zurückschreiben)

Schreibweise: $x_\ell = \mathbf{TOP}_{\mathbb{N}}(x_i)$

($i \neq \ell$)

Ackermann-Funktion

Test Leerheit

TOP(x_i) – 2

- Liefert 1 falls leer
- Liefert 0 sonst

Schreibweise: **EMPTY**(x_i)

Ackermann-Funktion

Test Leerheit

TOP(x_i) – 2

- Liefert 1 falls leer
- Liefert 0 sonst

Schreibweise: **EMPTY**(x_i)

Kellerspeicher für natürliche Zahlen

- Speicherung beliebiger natürliche Zahlen
- Unterstützung Standardoperationen
(Test Leerheit, Einfügen, Entfernen, Auslesen)

Ackermann-Funktion

Definition (§5.1 Ackermann-Funktion; engl. *Ackermann function*)

Für alle $x, y \in \mathbb{N}$ sei

$$a(x, y) = \begin{cases} y + 1 & \text{falls } x = 0 \\ a(x - 1, 1) & \text{falls } x \neq 0 \text{ und } y = 0 \\ a(x - 1, a(x, y - 1)) & \text{sonst} \end{cases}$$

Ackermann-Funktion

Implementation

$x_3 = 3$	(leerer Keller)
PUSH _N (x_3, x_1) ; PUSH _N (x_3, x_2)	(füge x_1 & x_2 ein)
WHILE (SIZE _N (x_3) > 1) {	(mind. 2 Elemente im Keller)
$x_2 = \mathbf{TOP}_N(x_3)$; $x_3 = \mathbf{POP}_N(x_3)$	(2. Parameter vom Keller)
$x_1 = \mathbf{TOP}_N(x_3)$; $x_3 = \mathbf{POP}_N(x_3)$	(1. Parameter vom Keller)
IF ($x_1 = 0$) { $x_3 = \mathbf{PUSH}_N(x_3, x_2 + 1)$ }	(liefere 2. Parameter + 1)
ELSE {	(2. oder 3. Fall)
$x_3 = \mathbf{PUSH}_N(x_3, x_1 - 1)$	(Rekursion über 1. Parameter + 1)
IF ($x_2 = 0$) { $x_3 = \mathbf{PUSH}_N(x_3, 1)$ }	(2. Fall mit Konstante 1)
ELSE { $x_3 = \mathbf{PUSH}_N(x_3, x_1)$; $x_3 = \mathbf{PUSH}_N(x_3, x_2 - 1)$ }	
}	
}	
$x_1 = \mathbf{TOP}_N(x_3)$	(Ergebnis im Keller)

Ackermann-Funktion

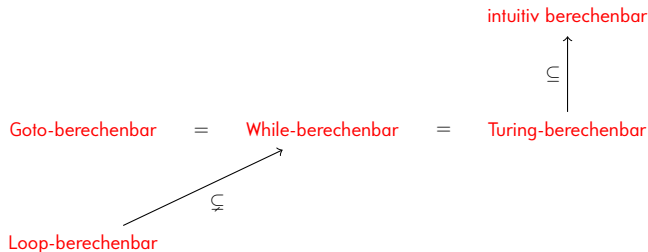
§6.5 Theorem

Ackermann-Funktion ist While-berechenbar

Ackermann-Funktion

§6.5 Theorem

Ackermann-Funktion ist While-berechenbar



Rekursive Funktionen

Konventionen

- Partielle Funktionen des Typs $f: \mathbb{N}^k \dashrightarrow \mathbb{N}$
- Addition (und Subtraktion) weiterhin auf \mathbb{N} begrenzt

Definition (§4.13 Projektion; *projection*)

Für $n \in \mathbb{N}$ und $1 \leq i \leq n$ ist $\pi_i^{(n)}: \mathbb{N}^n \rightarrow \mathbb{N}$ **n -stellige Projektion auf i -te Stelle**

$$\pi_i^{(n)}(a_1, \dots, a_n) = a_i \qquad a_1, \dots, a_n \in \mathbb{N}$$

Rekursive Funktionen

§6.6 Definition (rekursive Basisfunktionen; *recursive primitives*)

Folgende Funktionen sind **rekursive Basisfunktionen**

- n -stellige **a -konstante** Funktion $a^{(n)}: \mathbb{N}^n \rightarrow \mathbb{N}$
mit $a^{(n)}(a_1, \dots, a_n) = a$ für alle $n, a, a_1, \dots, a_n \in \mathbb{N}$ (Konstanten)

Rekursive Funktionen

§6.6 Definition (rekursive Basisfunktionen; *recursive primitives*)

Folgende Funktionen sind **rekursive Basisfunktionen**

- n -stellige **a -konstante** Funktion $a^{(n)}: \mathbb{N}^n \rightarrow \mathbb{N}$
mit $a^{(n)}(a_1, \dots, a_n) = a$ für alle $n, a, a_1, \dots, a_n \in \mathbb{N}$ (Konstanten)
- **Projektion** $\pi_i^{(n)}: \mathbb{N}^n \rightarrow \mathbb{N}$ für alle $n \in \mathbb{N}$ und $1 \leq i \leq n$
(Projektionen)

Rekursive Funktionen

§6.6 Definition (rekursive Basisfunktionen; *recursive primitives*)

Folgende Funktionen sind **rekursive Basisfunktionen**

- n -stellige **a -konstante** Funktion $a^{(n)}: \mathbb{N}^n \rightarrow \mathbb{N}$
mit $a^{(n)}(a_1, \dots, a_n) = a$ für alle $n, a, a_1, \dots, a_n \in \mathbb{N}$ (Konstanten)
- **Projektion** $\pi_i^{(n)}: \mathbb{N}^n \rightarrow \mathbb{N}$ für alle $n \in \mathbb{N}$ und $1 \leq i \leq n$
(Projektionen)
- **Inkrementfunktion** $nf: \mathbb{N} \rightarrow \mathbb{N}$
mit $nf(a) = a + 1$ für alle $a \in \mathbb{N}$ (Nachfolgerfunktion)

Rekursive Funktionen

§6.6 Definition (rekursive Basisfunktionen; *recursive primitives*)

Folgende Funktionen sind **rekursive Basisfunktionen**

- n -stellige **a -konstante** Funktion $a^{(n)}: \mathbb{N}^n \rightarrow \mathbb{N}$
mit $a^{(n)}(a_1, \dots, a_n) = a$ für alle $n, a, a_1, \dots, a_n \in \mathbb{N}$ (Konstanten)
- **Projektion** $\pi_i^{(n)}: \mathbb{N}^n \rightarrow \mathbb{N}$ für alle $n \in \mathbb{N}$ und $1 \leq i \leq n$
(Projektionen)
- **Inkrementfunktion** $nf: \mathbb{N} \rightarrow \mathbb{N}$
mit $nf(a) = a + 1$ für alle $a \in \mathbb{N}$ (Nachfolgerfunktion)

Keine weiteren rekursiven Basisfunktionen

Rekursive Funktionen

§6.6 Definition (rekursive Basisfunktionen; *recursive primitives*)

Folgende Funktionen sind **rekursive Basisfunktionen**

- n -stellige **a -konstante** Funktion $a^{(n)}: \mathbb{N}^n \rightarrow \mathbb{N}$
mit $a^{(n)}(a_1, \dots, a_n) = a$ für alle $n, a, a_1, \dots, a_n \in \mathbb{N}$ (Konstanten)
- **Projektion** $\pi_i^{(n)}: \mathbb{N}^n \rightarrow \mathbb{N}$ für alle $n \in \mathbb{N}$ und $1 \leq i \leq n$
(Projektionen)
- **Inkrementfunktion** $nf: \mathbb{N} \rightarrow \mathbb{N}$
mit $nf(a) = a + 1$ für alle $a \in \mathbb{N}$ (Nachfolgerfunktion)

Keine weiteren rekursiven Basisfunktionen

Notizen

- Nutzung mathematischen Syntax & Funktionssemantik

Rekursive Funktionen

§6.6 Definition (rekursive Basisfunktionen; *recursive primitives*)

Folgende Funktionen sind **rekursive Basisfunktionen**

- n -stellige **a -konstante** Funktion $a^{(n)}: \mathbb{N}^n \rightarrow \mathbb{N}$
mit $a^{(n)}(a_1, \dots, a_n) = a$ für alle $n, a, a_1, \dots, a_n \in \mathbb{N}$ (Konstanten)
- **Projektion** $\pi_i^{(n)}: \mathbb{N}^n \rightarrow \mathbb{N}$ für alle $n \in \mathbb{N}$ und $1 \leq i \leq n$
(Projektionen)
- **Inkrementfunktion** $nf: \mathbb{N} \rightarrow \mathbb{N}$
mit $nf(a) = a + 1$ für alle $a \in \mathbb{N}$ (Nachfolgerfunktion)

Keine weiteren rekursiven Basisfunktionen

Notizen

- Nutzung mathematischen Syntax & Funktionssemantik
- Basisfunktionen total (Vereinfachungen folgen)

Primitiv rekursive Funktionen

§6.7 Definition (primitiv rek. Fkt. [1/2]; *primitive rec. function*)

Genau folgende partielle Funktionen sind **primitiv rekursiv**

- Jede **rekursive Basisfunktion**

Primitiv rekursive Funktionen

§6.7 Definition (primitiv rek. Fkt. [1/2]; *primitive rec. function*)

Genau folgende partielle Funktionen sind **primitiv rekursiv**

- Jede **rekursive Basisfunktion**
- Für alle $m, n \in \mathbb{N}$ und primitiv rekursiven partiellen Funktionen $f: \mathbb{N}^m \dashrightarrow \mathbb{N}$ und $g_1, \dots, g_m: \mathbb{N}^n \dashrightarrow \mathbb{N}$ ist **Komposition**
 $f\langle g_1, \dots, g_m \rangle: \mathbb{N}^n \dashrightarrow \mathbb{N}$ mit

$$f\langle g_1, \dots, g_m \rangle(a_1, \dots, a_n) = f(g_1(a_1, \dots, a_n), \dots, g_m(a_1, \dots, a_n))$$

für alle $a_1, \dots, a_n \in \mathbb{N}$ primitiv rekursiv (Komposition)

Primitiv rekursive Funktionen

§6.7 Definition (primitiv rek. Fkt. [2/2]; *primitive rec. function*)

Genau folgende partielle Funktionen sind **primitiv rekursiv**

- Für alle $n \in \mathbb{N}$ und primitiv rekursiven partiellen Funktionen $f: \mathbb{N}^n \dashrightarrow \mathbb{N}$ und $g: \mathbb{N}^{n+2} \dashrightarrow \mathbb{N}$ ist durch **Schema primitive Rekursion** definierte partielle Funktion $\mathbf{pr}[f, g]: \mathbb{N}^{n+1} \dashrightarrow \mathbb{N}$

$$\mathbf{pr}[f, g](0, a_1, \dots, a_n) = f(a_1, \dots, a_n)$$

$$\mathbf{pr}[f, g](a+1, a_1, \dots, a_n) = g(\mathbf{pr}[f, g](a, a_1, \dots, a_n), a, a_1, \dots, a_n)$$

für alle $a, a_1, \dots, a_n \in \mathbb{N}$ (primitive Rekursion)

Keine weitere primitiv rekursiven partiellen Funktionen

Primitiv rekursive Funktionen

n -stelliges Inkrement Komponente i

$$\text{nf}_i^{(n)} = \text{nf}\langle \pi_i^{(n)} \rangle$$

$$\text{nf}_i^{(n)}(a_1, \dots, a_n) = \text{nf}(\pi_i^{(n)}(a_1, \dots, a_n)) = \text{nf}(a_i) = a_i + 1$$

Primitiv rekursive Funktionen

n -stelliges Inkrement Komponente i

$$\text{nf}_i^{(n)} = \text{nf} \langle \pi_i^{(n)} \rangle$$

$$\text{nf}_i^{(n)}(a_1, \dots, a_n) = \text{nf}(\pi_i^{(n)}(a_1, \dots, a_n)) = \text{nf}(a_i) = a_i + 1$$

Addition

$$\text{add} = \mathbf{pr}[\pi_1^{(1)}, \text{nf}_1^{(3)}]$$

$$\text{add}(0, b) = \pi_1^{(1)}(b) = b$$

$$\text{add}(a + 1, b) = \text{nf}_1^{(3)}(\text{add}(a, b), a, b) = \text{add}(a, b) + 1$$

Primitiv rekursive Funktionen

n -stelliges Inkrement Komponente i

$$\text{nf}_i^{(n)} = \text{nf} \langle \pi_i^{(n)} \rangle$$

$$\text{nf}_i^{(n)}(a_1, \dots, a_n) = \text{nf}(\pi_i^{(n)}(a_1, \dots, a_n)) = \text{nf}(a_i) = a_i + 1$$

Addition

$$\text{add} = \mathbf{pr}[\pi_1^{(1)}, \text{nf}_1^{(3)}]$$

$$\text{add}(0, b) = \pi_1^{(1)}(b) = b$$

$$\text{add}(a + 1, b) = \text{nf}_1^{(3)}(\text{add}(a, b), a, b) = \text{add}(a, b) + 1$$

Multiplikation

$$\text{mult} = \mathbf{pr}[0^{(1)}, \text{add} \langle \pi_1^{(3)}, \pi_3^{(3)} \rangle]$$

$$\text{mult}(0, b) = 0^{(1)}(b) = 0$$

$$\begin{aligned} \text{mult}(a + 1, b) &= \text{add} \left(\pi_1^{(3)}(\text{mult}(a, b), a, b), \pi_3^{(3)}(\text{mult}(a, b), a, b) \right) \\ &= \text{add}(\text{mult}(a, b), b) = \text{mult}(a, b) + b \end{aligned}$$

Primitiv rekursive Funktionen

Vereinfachungen

- Direkte Verwendung Projektion (ohne explizite Angabe)
- Freie Verwendung Parameter

Primitiv rekursive Funktionen

Vereinfachungen

- Direkte Verwendung Projektion (ohne explizite Angabe)
- Freie Verwendung Parameter
- Verwendung Makros & übliche Schreibweisen
(für bereits als primitiv rekursiv bekannte Funktionen)
- Schreibweise “+1” statt `nf`

Primitiv rekursive Funktionen

Addition

wesentliche Rekursion $(a + 1) + b = (a + b) + 1$

$$\text{add}(0, b) = b$$

$$\text{add}(a + 1, b) = \text{add}(a, b) + 1$$

Primitiv rekursive Funktionen

Addition

wesentliche Rekursion $(a + 1) + b = (a + b) + 1$

$$\text{add}(0, b) = b$$

$$\text{add}(a + 1, b) = \text{add}(a, b) + 1$$

Multiplikation

wesentliche Rekursion $(a + 1) \cdot b = (a \cdot b) + b$

$$\text{mult}(0, b) = 0$$

$$\text{mult}(a + 1, b) = \text{mult}(a, b) + b$$

Primitiv rekursive Funktionen

Vorgänger

$$\text{vg} = \mathbf{pr}[0^{(0)}, \pi_2^{(2)}]$$

$$\text{vg}(0) = 0$$

$$\text{vg}(a + 1) = a$$

Primitiv rekursive Funktionen

Vorgänger

$$\text{vg} = \mathbf{pr}[0^{(0)}, \pi_2^{(2)}]$$

$$\text{vg}(0) = 0$$

$$\text{vg}(a+1) = a$$

Subtraktion

$$\text{sub}' = \mathbf{pr}[\pi_1^{(1)}, \text{vg}\langle \pi_1^{(3)} \rangle]$$

wesentliche Rekursion $a - (b+1) = (a-b) - 1$

$$\text{sub}'(0, a) = a$$

$$\text{sub}'(b+1, a) = \text{vg}(\text{sub}'(b, a))$$

$$\text{sub}(a, b) = \text{sub}'(b, a)$$

$$\text{sub} = \text{sub}'\langle \pi_2^{(2)}, \pi_1^{(2)} \rangle$$

Primitiv rekursive Funktionen

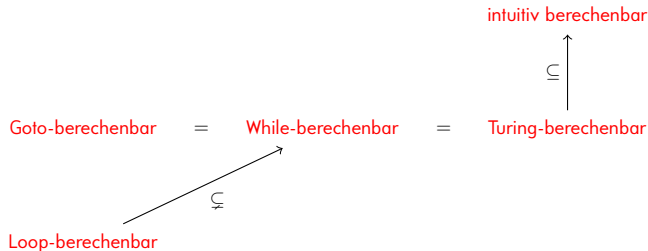
Notizen

- Primitiv rekursive Funktionen total
- Beschränkte Rekursion über 1 Argument
- Ähnlichkeit zu Loop-Programmen

Primitiv rekursive Funktionen

Notizen

- Primitiv rekursive Funktionen total
- Beschränkte Rekursion über 1 Argument
- Ähnlichkeit zu Loop-Programmen

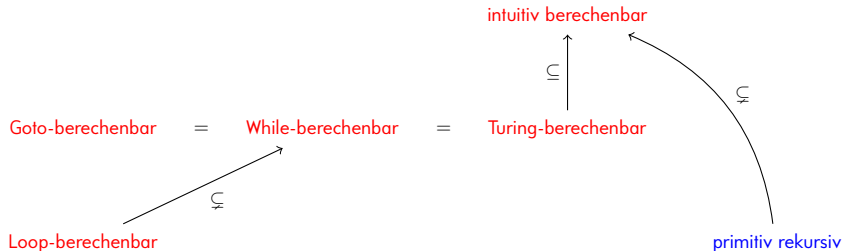


primitiv rekursiv

Primitiv rekursive Funktionen

Notizen

- Primitiv rekursive Funktionen total
- Beschränkte Rekursion über 1 Argument
- Ähnlichkeit zu Loop-Programmen



Zusammenfassung

- While-berechenbar = Turing-berechenbar
- Ackermann-Funktion While-berechenbar
- Primitiv rekursive Funktionen

Vierte Übungsserie bereits im Moodle