10a

Primitiv und μ-rekursive Funktionen

Prof. Dr. Jasmin Blanchette

Lehr- und Forschungseinheit für
Theoretische Informatik und Theorembeweisen

Stand: 25. Juni 2024
Basierend auf Folien von PD Dr. David Sabel

Weitere Formalismen

Weitere Formalismen zur Definition der Berechenbarkeit:

- primitiv rekursive Funktionen
- μ-rekursive Funktionen.

Weitere Formalismen

Weitere Formalismen zur Definition der Berechenbarkeit:

- primitiv rekursive Funktionen
- μ-rekursive Funktionen.

Wir werden schließlich sehen:

- Primitiv rekursive Funktionen entsprechen genau den LOOP-berechenbaren Funktionen.
- μ-rekursive Funktionen entsprechen genau den turingberechenbaren (bzw. WHILE-, GOTO-berechenbaren) Funktionen.

Primitiv rekursive Funktionen

Definition

Eine Funktion $f : \mathbb{N}^k \to \mathbb{N}$ ist primitiv rekursiv, wenn sie der folgenden Definition genügt:

- Jede konstante Funktion $f(x_1, \ldots, x_k) = c \in \mathbb{N}$ ist primitiv rekursiv.
- Die Projektionsfunktionen $\pi_i^k(x_1, \ldots, x_k) = x_i$ sind primitiv rekursiv.
- Die Nachfolgerfunktion $succ(x) = x + 1$ ist primitiv rekursiv.
- Komposition/Einsetzung: Wenn $g : \mathbb{N}^m \to \mathbb{N}$ und für $i = 1, \ldots, m$: $h_i : \mathbb{N}^k \to \mathbb{N}$ primitiv rekursiv sind, dann ist auch f mit $f(x_1, \ldots, x_k) = g(h_1(x_1, \ldots, x_k), \ldots, h_m(x_1, \ldots, x_k))$ primitiv rekursiv.

(Fortsetzung folgt.)

Primitiv rekursive Funktionen

Definition

- Rekursion: Wenn $g : \mathbb{N}^{k-1} \to \mathbb{N}$ und $h : \mathbb{N}^{k+1} \to \mathbb{N}$ primitiv rekursiv sind, dann ist auch f mit

$$f(x_1, \ldots, x_k) = \begin{cases} g(x_2, \ldots, x_k) & \text{falls } x_1 = 0 \\ h(f(x_1 - 1, x_2, \ldots, x_k), x_1 - 1, x_2, \ldots, x_k) & \text{sonst} \end{cases}$$

  primitiv rekursiv.

Additionsfunktion

$add(x_1, x_2) = x_1 + x_2$ ist primitiv rekursiv:

$$add(x_1, x_2) = \begin{cases} x_2 & \text{falls } x_1 = 0 \\ succ(add(x_1 - 1, x_2)) & \text{sonst} \end{cases}$$

Grundgedanke: x_1-mal 1 zu x_2 addieren.

Additionsfunktion

$add(x_1, x_2) = x_1 + x_2$ ist primitiv rekursiv:

$$add(x_1, x_2) = \begin{cases} x_2 & \text{falls } x_1 = 0 \\ succ(add(x_1 - 1, x_2)) & \text{sonst} \end{cases}$$

Grundgedanke: x_1-mal 1 zu x_2 addieren.

Die verwendeten Funktionen g und h aus der Definition der primitiv rekursiven Funktionen sind hier

- $g = \pi_1^1$
- $h(x_1, x_2, x_3) = succ(\pi_1^3(x_1, x_2, x_3))$

Komponenten eines Tupels entfernen/vertauschen/vervielfachen

Wenn $g : \mathbb{N}^4 \to \mathbb{N}$ primitiv rekursiv ist, dann ist auch z.B. $f : \mathbb{N}^3 \to \mathbb{N}$ mit

$$f(n_1, n_2, n_3) = g(n_2, n_3, n_3, n_2)$$

denn

$$f(n_1, n_2, n_3) = g(\pi^3_2(n_1, n_2, n_3), \pi^3_3(n_1, n_2, n_3), \pi^3_3(n_1, n_2, n_3), \pi^3_2(n_1, n_2, n_3))$$

Multiplikationsfunktion

$mult(x_1, x_2) = x_1 \cdot x_2$ ist primitiv rekursiv:

$$mult(x_1, x_2) = \begin{cases} 0 & \text{falls } x_1 = 0 \\ add(mult(x_1 - 1, x_2), x_2) & \text{sonst} \end{cases}$$

Grundgedanke: x_1-mal x_2 zu 0 addieren.

Rekursion durch das i-te Argument

Für $1 \leq i \leq k$ kann man

$$f(x_1, \ldots, x_k) = \begin{cases} g(x_1, \ldots, \boldsymbol{x_{i-1}}, \boldsymbol{x_{i+1}}, x_k), & \text{falls } x_i = 0 \\ h(f(x_1, \ldots, \boldsymbol{x_{i-1}}, \boldsymbol{x_i - 1}, \boldsymbol{x_{i+1}}, \ldots, x_k), x_1, \ldots, \boldsymbol{x_{i-1}}, \boldsymbol{x_i - 1}, \boldsymbol{x_{i+1}}, \ldots, x_k), & \text{sonst} \end{cases}$$

durch $f(x_1, \ldots, x_k) = f'(x_i, x_1 \ldots, x_{i-1}, x_{i+1}, \ldots, x_k)$ darstellen, wobei

$$\begin{aligned} f'(y_1, \ldots, y_k) &= \begin{cases} g(y_2, \ldots, y_k) & \text{falls } y_1 = 0 \\ h'(f'(y_1 - 1, y_2 \ldots, y_k), y_1 - 1, y_2, \ldots, y_k) & \text{sonst} \end{cases} \\ h'(y_0, \ldots, y_k) &= h(y_0, y_2, y_3, \ldots, y_i, y_1, y_{i+1}, y_{i+2}, \ldots, y_k) \end{aligned}$$

Angepasste Differenz

Im Allgemeinen ist $x_1 - x_2$ nicht primitiv rekursiv, weil der undefinierte Fall $x_1 < x_2$ nicht darstellbar ist.

Angepasste Differenz

Im Allgemeinen ist $x_1 - x_2$ nicht primitiv rekursiv, weil der undefinierte Fall $x_1 < x_2$ nicht darstellbar ist.

Hingegen ist die angepasste Differenz, die 0 liefert falls $x_1 < x_2$, primitiv rekursiv:

$$sub(x_1, x_2) = \begin{cases} x_1 & \text{falls } x_2 = 0 \\ pred(sub(x_1, x_2 - 1)) & \text{sonst} \end{cases}$$

wobei

$$pred(x_1) = \begin{cases} 0 & \text{falls } x_1 = 0 \\ x_1 - 1 & \text{sonst} \end{cases}$$

Nächstes Ziel

Wir wollen zeigen:
Die primitiv rekursive Funktionen sind genau die LOOP-berechenbaren Funktionen.

Nächstes Ziel

Wir wollen zeigen:
Die primitiv rekursive Funktionen sind genau die LOOP-berechenbaren Funktionen.

Dafür benötigen wir:

- eine Darstellung einer Variablenbelegung ρ als eine einzige Zahl, um sie der primitiv rekursiven Funktion als Argument zu übergeben
  (d.h. eindeutige Darstellung eines Tupels natürlicher Zahlen als eine einzige Zahl)
- Operationen zum Konvertieren in beide Richtungen.

Eine solches Verfahren nennt man Gödelisierung (nach Kurt Gödel).

Gödelisierung

Wir wollen Tupel von natürlichen Zahlen $(x_0, \ldots, x_k)$ bijektiv in die natürlichen Zahlen abbilden mit primitiv rekursiven Funktionen.

Gödelisierung

Wir wollen Tupel von natürlichen Zahlen $(x_0, \ldots, x_k)$ bijektiv in die natürlichen Zahlen abbilden mit primitiv rekursiven Funktionen.

Lösung:

$$c(x, y) = \binom{x + y + 1}{2} + x$$

die mithilfe des Binomialkoeffizients definiert ist.

Gödelisierung

Wir wollen Tupel von natürlichen Zahlen $(x_0, \ldots, x_k)$ bijektiv in die natürlichen Zahlen abbilden mit primitiv rekursiven Funktionen.

Lösung:

$$c(x, y) = \binom{x + y + 1}{2} + x$$

die mithilfe des Binomialkoeffizients definiert ist.

Werte von $c(x, y)$ für $x, y \in \{0, \ldots, 4\}$:

		x				
		0	1	2	3	4
	0	0	2	5	9	14
	1	1	4	8	13	19
y	2	3	7	12	18	25
	3	6	11	17	24	32
	4	10	16	23	31	40

Gödelisierung

Die Funktion c ist primitiv rekursiv, da

$$\binom{0}{2} = 0 \text{ und } \binom{n+1}{2} = \binom{n}{2} + n$$

Für $(k+1)$-Tupel definieren wir

$$\langle x_0, x_1, \ldots, x_k \rangle = c(x_0, c(x_1, \ldots, c(x_k, 0)\ldots))$$

$\langle\ \rangle$ ist primitiv rekursiv, da c primitiv rekursiv und Komposition primitiv rekursiv ist.

Gödelisierung

Die Komponenten eines Tupels können zurückgewonnen werden.

Seien *left* und *right* Funktionen mit

- $left(c(x, y)) = x$ und
- $right(c(x, y)) = y$.

Im Skript wird gezeigt:

- *left* und *right* existieren.
- *left* und *right* sind primitiv rekursiv.

Gödelisierung

Zugriff auf beliebige Komponenten ist auch möglich.

Programmiere $d_i(\langle x_0, \ldots, x_k \rangle) = x_i$ durch

$$\begin{aligned} d_0(x) &= \mathit{left}(x) \\ d_1(x) &= \mathit{left}(\mathit{right}(x)) \\ d_i(x) &= \mathit{left}(\underbrace{\mathit{right}(\mathit{right}(\ldots \mathit{right}}_{i\text{-mal}}(x)\ldots))) \end{aligned}$$

Damit sind auch die d_i-Funktionen primitiv rekursiv.

Von LOOP-Programm berechnete Funktion

- Sei P ein LOOP-Programm.
- Seien $x_0, x_1, \ldots, x_n$ alle vom Programm P verwendeten Variablen.
- Die von P berechnete Funktion ist

$$g_P(\langle x_0, \ldots, x_n \rangle) = \langle x_0', \ldots, x_n' \rangle$$

LOOP-Programme berechnen primitiv rekursive Funktionen

Lemma

Für jedes LOOP-Programm P ist die zugehörige Funktion g_P primitiv rekursiv.

LOOP-Programme berechnen primitiv rekursive Funktionen

Lemma

Für jedes LOOP-Programm P ist die zugehörige Funktion g_P primitiv rekursiv.

Beweis Durch Induktion über die Größe von jedem Teilprogramm Q.

LOOP-Programme berechnen primitiv rekursive Funktionen

Lemma

Für jedes LOOP-Programm P ist die zugehörige Funktion g_P primitiv rekursiv.

Beweis Durch Induktion über die Größe von jedem Teilprogramm Q.

- Fall Q ist Zuweisung $x_i := x_j \pm c$: Für g_Q muss gelten:

$$g_Q(\langle m_0, \ldots, m_n \rangle) = \langle m_0, \ldots, m_{i-1}, m_j \pm c, m_i, \ldots, m_n \rangle$$

LOOP-Programme berechnen primitiv rekursive Funktionen

Lemma

Für jedes LOOP-Programm P ist die zugehörige Funktion g_P primitiv rekursiv.

Beweis Durch Induktion über die Größe von jedem Teilprogramm Q.

- Fall Q ist Zuweisung $x_i := x_j \pm c$: Für g_Q muss gelten:

$$g_Q(\langle m_0, \ldots, m_n \rangle) = \langle m_0, \ldots, m_{i-1}, m_j \pm c, m_i, \ldots, m_n \rangle$$

Primitiv rekursive Implementierung:

$$g_Q(x) = \langle d_0(x), \ldots, d_{i-1}(x), d_j(x) \pm c, d_{i+1}(x), \ldots d_n(x) \rangle$$

LOOP-Programme berechnen primitiv rekursive Funktionen

Beweis (Fortsetzung)

- Fall Q ist eine Sequenz $Q_1; Q_2$: Die Induktionshypothese liefert primitiv rekursive Funktionen g_{Q_1}, g_{Q_2}.

LOOP-Programme berechnen primitiv rekursive Funktionen

Beweis (Fortsetzung)

- Fall Q ist eine Sequenz $Q_1; Q_2$: Die Induktionshypothese liefert primitiv rekursive Funktionen g_{Q_1}, g_{Q_2}.

  Die Funktion $g_Q(x) = g_{Q_2}(g_{Q_1}(x))$ ist primitiv rekursiv.

LOOP-Programme berechnen primitiv rekursive Funktionen

Beweis (Fortsetzung)

- Fall Q ist eine Sequenz $Q_1; Q_2$: Die Induktionshypothese liefert primitiv rekursive Funktionen g_{Q_1}, g_{Q_2}.

  Die Funktion $g_Q(x) = g_{Q_2}(g_{Q_1}(x))$ ist primitiv rekursiv.
- Fall Q ist **LOOP** x_i **DO** P **END**: Die Induktionshypothese liefert die primitiv rekursive Funktion g_P.

LOOP-Programme berechnen primitiv rekursive Funktionen

Beweis (Fortsetzung)

- Fall Q ist eine Sequenz $Q_1; Q_2$: Die Induktionshypothese liefert primitiv rekursive Funktionen g_{Q_1}, g_{Q_2}.

  Die Funktion $g_Q(x) = g_{Q_2}(g_{Q_1}(x))$ ist primitiv rekursiv.
- Fall Q ist **LOOP** x_i **DO** P **END**: Die Induktionshypothese liefert die primitiv rekursive Funktion g_P.

  Die Konstruktion von g_Q wendet g_P x_i-mal an.

$$
\begin{aligned}
g_Q(x) &= run(d_i(x), x) \\
run(n, x) &= \begin{cases} x, & \text{falls } n = 0 \\ g_P(run(n-1, x)), & \text{sonst} \end{cases}
\end{aligned}
$$

run ist primitiv rekursiv. Damit ist auch g_Q primitiv rekursiv. □

LOOP-berechenbare Funktionen sind primitiv rekursiv

Satz

Jede LOOP-berechenbare Funktion ist primitiv rekursiv.

LOOP-berechenbare Funktionen sind primitiv rekursiv

Satz

Jede LOOP-berechenbare Funktion ist primitiv rekursiv.

Beweis Sei $f : \mathbb{N}^k \to \mathbb{N}$ eine LOOP-berechenbar Funktion.

LOOP-berechenbare Funktionen sind primitiv rekursiv

Satz

Jede LOOP-berechenbare Funktion ist primitiv rekursiv.

Beweis Sei $f : \mathbb{N}^k \to \mathbb{N}$ eine LOOP-berechenbar Funktion.

Daher gibt es ein LOOP-Programm P mit $\rho = \{x_1 \mapsto n_1, \ldots, x_k \mapsto n_k\}$, $(\rho, P) \xrightarrow[\text{LOOP}]{}^* (\rho', \varepsilon)$ und $\rho'(x_0) = f(n_1, \ldots, n_k)$.

LOOP-berechenbare Funktionen sind primitiv rekursiv

Satz

Jede LOOP-berechenbare Funktion ist primitiv rekursiv.

Beweis Sei $f : \mathbb{N}^k \to \mathbb{N}$ eine LOOP-berechenbar Funktion.

Daher gibt es ein LOOP-Programm P mit $\rho = \{x_1 \mapsto n_1, \ldots, x_k \mapsto n_k\}$, $(\rho, P) \xrightarrow[\text{LOOP}]{}^* (\rho', \varepsilon)$ und $\rho'(x_0) = f(n_1, \ldots, n_k)$.

Es gilt $f(n_1, \ldots, n_k) = d_0(g_P(\langle 0, n_1, \ldots, n_k, 0, \ldots, 0\rangle))$.

LOOP-berechenbare Funktionen sind primitiv rekursiv

Satz

Jede LOOP-berechenbare Funktion ist primitiv rekursiv.

Beweis Sei $f : \mathbb{N}^k \to \mathbb{N}$ eine LOOP-berechenbar Funktion.

Daher gibt es ein LOOP-Programm P mit $\rho = \{x_1 \mapsto n_1, \ldots, x_k \mapsto n_k\}$, $(\rho, P) \xrightarrow[\text{LOOP}]{}^* (\rho', \varepsilon)$ und $\rho'(x_0) = f(n_1, \ldots, n_k)$.

Es gilt $f(n_1, \ldots, n_k) = d_0(g_P(\langle 0, n_1, \ldots, n_k, 0, \ldots, 0\rangle))$.

Da $\langle\ \rangle$, d_0 und g_P primitiv rekursiv sind, ist auch f primitiv rekursiv. □

Primitiv rekursive Funktionen sind LOOP-berechenbar

Satz

Jede primitiv rekursive Funktion ist LOOP-berechenbar.

Primitiv rekursive Funktionen sind LOOP-berechenbar

Satz

Jede primitiv rekursive Funktion ist LOOP-berechenbar.

Beweis Durch Induktion über die Größe der primitiv rekursiven Funktion.

Primitiv rekursive Funktionen sind LOOP-berechenbar

Satz

Jede primitiv rekursive Funktion ist LOOP-berechenbar.

Beweis Durch Induktion über die Größe der primitiv rekursiven Funktion.

- Fall $f(x) = c$, $f = succ$ oder $f = \pi_n^k$: Dann gibt es auch ein LOOP-Programm dazu.

Primitiv rekursive Funktionen sind LOOP-berechenbar

Beweis (Fortsetzung)

- Fall $f(x_1, \ldots, x_k) = h(g_1(x_1, \ldots, x_k), \ldots, g_n(x_1, \ldots, x_k))$:
  Die Induktionshypothese liefert LOOP-Programme P_h und $P_{g_1}, \ldots, P_{g_n}$, die $h, g_1, \ldots, g_n$ berechnen.

Primitiv rekursive Funktionen sind LOOP-berechenbar

Beweis (Fortsetzung)

- Fall $f(x_1, \ldots, x_k) = h(g_1(x_1, \ldots, x_k), \ldots, g_n(x_1, \ldots, x_k))$:
  Die Induktionshypothese liefert LOOP-Programme P_h und $P_{g_1}, \ldots, P_{g_n}$, die $h, g_1, \ldots, g_n$ berechnen.

  Konstruiere Programm für f nach dem Schema

  $$\begin{aligned} y_1 &:= g_1(x_1, \ldots, x_k); \\ &\vdots \\ y_n &:= g_n(x_1, \ldots, x_k); \\ x_0 &:= h(y_1, \ldots, y_n) \end{aligned}$$

Primitiv rekursive Funktionen sind LOOP-berechenbar

Beweis (Fortsetzung)

- Fall $f(x_1, \ldots, x_k) = h(g_1(x_1, \ldots, x_k), \ldots, g_n(x_1, \ldots, x_k))$:
  Die Induktionshypothese liefert LOOP-Programme P_h und $P_{g_1}, \ldots, P_{g_n}$, die $h, g_1, \ldots, g_n$ berechnen.

  Konstruiere Programm für f nach dem Schema

  $$\begin{aligned} y_1 &:= g_1(x_1, \ldots, x_k); \\ &\vdots \\ y_n &:= g_n(x_1, \ldots, x_k); \\ x_0 &:= h(y_1, \ldots, y_n) \end{aligned}$$

  Genauer: $P_{g_1}, \ldots, P_{g_n}, P_h$ abändern, sodass sie auf disjunkten Variablenmengen arbeiten, entsprechende Variableninhalte für $x_1, \ldots, x_k$ verdoppeln.

Primitiv rekursive Funktionen sind LOOP-berechenbar

Beweis (Fortsetzung)

- Fall $f(x_1, \ldots, x_k) = \begin{cases} g(x_2, \ldots, x_k) & \text{falls } x_1 = 0 \\ h(f(x_1 - 1, x_2, \ldots, x_k), x_1 - 1, x_2, \ldots, x_k) & \text{sonst} \end{cases}$

  wobei $g : \mathbb{N}^{k-1} \to \mathbb{N}$ und $h : \mathbb{N}^{k+1} \to \mathbb{N}$ primitiv rekursiv sind:

Primitiv rekursive Funktionen sind LOOP-berechenbar

Beweis (Fortsetzung)

- Fall $f(x_1, \ldots, x_k) = \begin{cases} g(x_2, \ldots, x_k) & \text{falls } x_1 = 0 \\ h(f(x_1 - 1, x_2, \ldots, x_k), x_1 - 1, x_2, \ldots, x_k) & \text{sonst} \end{cases}$

  wobei $g : \mathbb{N}^{k-1} \to \mathbb{N}$ und $h : \mathbb{N}^{k+1} \to \mathbb{N}$ primitiv rekursiv sind:

  Die Induktionshypothese liefert LOOP-Programme, die g, h berechnen.
  Konstruiere LOOP-Programm für f nach dem Schema

  $y := 0;$
  $x_0 := g(x_2, \ldots, x_k);$
  LOOP x_1 **DO**
  $\quad x_0 := h(x_0, y, x_2, \ldots, x_k);$
  $\quad y := y + 1$
  END

□

Äquivalenz von LOOP-Berechenbarkeit und von primitive Rekursion

Theorem

Die primitiv rekursiven Funktionen sind genau die LOOP-berechenbaren Funktionen.

μ-Operator

Definition

Sei $h : \mathbb{N}^{k+1} \to \mathbb{N}$ eine (partielle oder totale) Funktion.
Dann ist $\mu h : \mathbb{N}^k \to \mathbb{N}$ definiert als

$$(\mu h)(x_1, \ldots, x_k) = \begin{cases} n & \text{falls } h(n, x_1, \ldots, x_k) = 0 \text{ und für} \\ & \text{alle } m < n\text{: } h(m, x_1, \ldots, x_k) \text{ ist definiert} \\ & \text{und } h(m, x_1, \ldots, x_k) > 0 \\ \text{undefiniert} & \text{sonst} \end{cases}$$

μ-Operator

Definition

Sei $h : \mathbb{N}^{k+1} \to \mathbb{N}$ eine (partielle oder totale) Funktion.
Dann ist $\mu h : \mathbb{N}^k \to \mathbb{N}$ definiert als

$$(\mu h)(x_1, \ldots, x_k) = \begin{cases} n & \text{falls } h(n, x_1, \ldots, x_k) = 0 \text{ und für} \\ & \text{alle } m < n\text{: } h(m, x_1, \ldots, x_k) \text{ ist definiert} \\ & \text{und } h(m, x_1, \ldots, x_k) > 0 \\ \text{undefiniert} & \text{sonst} \end{cases}$$

Der μ-Operator „sucht“ nach der ersten Nullstelle von h.

Wenn diese nicht existiert (entweder da h keine Nullstelle hat, oder da h undefiniert ist für Werte, die kleiner als die Nullstelle sind), dann ist auch μh undefiniert.

Beispiel für den μ-Operator

Sei $bus(x_1, x_2) = sub(x_2, x_1)$, wobei sub die angepasste Differenz berechnet.
$(\mu bus)(5) = ?$

Beispiel für den μ-Operator

Sei $bus(x_1, x_2) = sub(x_2, x_1)$, wobei sub die angepasste Differenz berechnet.

$(\mu bus)(5) = 5$ weil

$bus(0, 5) = 5$

$bus(1, 5) = 4$

$bus(2, 5) = 3$

$bus(3, 5) = 2$

$bus(4, 5) = 1$

$bus(5, 5) = 0.$

μ-rekursive Funktionen

Definition

Eine Funktion $f : \mathbb{N}^k \to \mathbb{N}$ ist μ-rekursiv, wenn sie der folgenden Definition genügt:

- Jede konstante Funktion $f(x_1, \ldots, x_k) = c \in \mathbb{N}$ ist μ-rekursiv.
- Die Projektionsfunktionen $\pi_i^k(x_1, \ldots, x_k) = x_i$ sind μ-rekursiv.
- Die Nachfolgerfunktion $succ(x) = x + 1$ ist μ-rekursiv.
- Komposition/Einsetzung: Wenn $g : \mathbb{N}^m \to \mathbb{N}$ und für $i = 1, \ldots, m$: $h_i : \mathbb{N}^k \to \mathbb{N}$ μ-rekursiv sind, dann ist auch f mit $f(x_1, \ldots, x_k) = g(h_1(x_1, \ldots, x_k), \ldots, h_m(x_1, \ldots, x_k))$ μ-rekursiv.

(Fortsetzung folgt.)

μ-rekursive Funktionen

Definition

- Rekursion: Wenn $g : \mathbb{N}^{k-1} \to \mathbb{N}$ und $h : \mathbb{N}^{k+1} \to \mathbb{N}$ μ-rekursiv sind, dann ist

$$f(x_1, \ldots, x_k) = \begin{cases} g(x_2, \ldots, x_k) & \text{falls } x_1 = 0 \\ h(f(x_1 - 1, x_2, \ldots, x_k), x_1 - 1, x_2, \ldots, x_k) & \text{sonst} \end{cases}$$

  auch μ-rekursiv.
- μ-Operator: Wenn $h : \mathbb{N}^{k+1} \to \mathbb{N}$ μ-rekursiv ist, dann ist auch $f = \mu h$ μ-rekursiv.

WHILE-berechenbare Funktionen sind μ-rekursiv

Satz

Jede WHILE-berechenbare Funktion ist μ-rekursiv.

WHILE-berechenbare Funktionen sind μ-rekursiv

Satz

Jede WHILE-berechenbare Funktion ist μ-rekursiv.

Beweis Analog zum Beweis, dass die LOOP-berechenbaren Funktionen primitiv rekursiv sind.

WHILE-berechenbare Funktionen sind μ-rekursiv

Satz

Jede WHILE-berechenbare Funktion ist μ-rekursiv.

Beweis Analog zum Beweis, dass die LOOP-berechenbaren Funktionen primitiv rekursiv sind.

Neuer Fall: P ist **WHILE** $x_i \neq 0$ **DO** Q **END**.

WHILE-berechenbare Funktionen sind μ-rekursiv

Satz

Jede WHILE-berechenbare Funktion ist μ-rekursiv.

Beweis Analog zum Beweis, dass die LOOP-berechenbaren Funktionen primitiv rekursiv sind.

Neuer Fall: P ist **WHILE** $x_i \neq 0$ **DO** Q **END**.

Die Induktionshypothese liefert μ-rekursive Funktion g_Q für Q. Konstruiere:

$$
\begin{aligned}
g_P(x) &= run(\mu(run_i)(x), x) \\
run_i(n, x) &= d_i(run(n, x)) \\
run(n, x) &= \begin{cases} x & \text{falls } n = 0 \\ g_Q(run(n-1, x)) & \text{sonst} \end{cases}
\end{aligned}
$$

WHILE-berechenbare Funktionen sind μ-rekursiv

Beweis (Fortsetzung)

$$
\begin{aligned}
g_P(x) &= run((\mu run_i)(x), x) \\
run_i(n, x) &= d_i(run(n, x)) \\
run(n, x) &= \begin{cases} x & \text{falls } n = 0 \\ g_Q(run(n-1, x)) & \text{sonst} \end{cases}
\end{aligned}
$$

- $run(n, x)$ führt g_Q n-mal aus.
- $(\mu run_i)(x)$ berechnet, wie oft die Schleife minimal durchlaufen werden muss, bis x_i den Wert 0 hat.
- Divergiert die Schleife, so ist μrun_i undefiniert und g_P undefiniert. □

μ-rekursive Funktionen sind WHILE-berechenbar

Satz

Jede μ-rekursive Funktion ist WHILE-berechenbar.

μ-rekursive Funktionen sind WHILE-berechenbar

Satz

Jede μ-rekursive Funktion ist WHILE-berechenbar.

Beweis Analog zum Beweis, dass die primitiv rekursiven Funktionen LOOP-berechenbar sind.

μ-rekursive Funktionen sind WHILE-berechenbar

Satz

Jede μ-rekursive Funktion ist WHILE-berechenbar.

Beweis Analog zum Beweis, dass die primitiv rekursiven Funktionen LOOP-berechenbar sind.

Neuer Fall: $f = \mu h$ für eine μ-rekursive Funktion h.

μ-rekursive Funktionen sind WHILE-berechenbar

Satz

Jede μ-rekursive Funktion ist WHILE-berechenbar.

Beweis Analog zum Beweis, dass die primitiv rekursiven Funktionen LOOP-berechenbar sind.

Neuer Fall: $f = \mu h$ für eine μ-rekursive Funktion h.

Die Induktionshypothese liefert ein WHILE-Programm P_h, das h berechnet. Konstruiere P_f nach dem Schema

$x_0 := 0;$
$y := h(0, x_1, \ldots, x_n);$
WHILE $y \neq 0$ **DO**
  $x_0 := x_0 + 1;$
  $y := h(x_0, \ldots, x_n)$
END

μ-rekursive Funktionen sind WHILE-berechenbar

Beweis (Fortsetzung)

$x_0 := 0;$
$y := h(0, x_1, \ldots, x_n);$
WHILE $y \neq 0$ **DO**
  $x_0 := x_0 + 1;$
  $y := h(x_0, \ldots, x_n)$
END

- Die **WHILE**-Schleife berechnet den minimalen x_0, sodass $h(x_0, \ldots, x_n) = 0$.
- Wenn dieser Wert nicht existiert, terminiert die Schleife nicht.
- Dies entspricht der Berechnung von μh. □

Äquivalenz von WHILE-Berechenbarkeit und von μ-Rekursion

Theorem

Die μ-rekursiven Funktionen entsprechen genau den WHILE-berechenbaren (und damit auch den GOTO- und turingberechenbaren) Funktionen.

Überblick über die Berechenbarkeitsformalismen