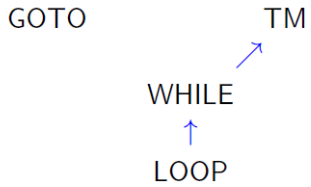
Script generated by TTT

Title: Seidl: Theoretische_Informatik (20.06.2013)

Date: Thu Jun 20 16:06:44 CEST 2013

Duration: 101:47 min

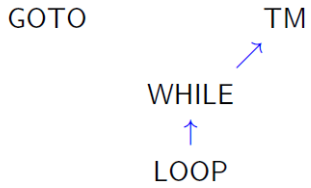
Pages: 88



Übersetzungen

Ein GOTO-Programm ist eine Sequenzen von markierten Anweisungen

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



Übersetzungen

Ein GOTO-Programm ist eine Sequenzen von markierten Anweisungen

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

Ein GOTO-Programm ist eine Sequenzen von markierten Anweisungen

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

(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

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

(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 → GOTO)

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

Satz 4.23 (GOTO → WHILE)

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

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

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
```


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 $\quad\mapsto\quad$ 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 $\quad\mapsto\quad$ x_i := x_j +/- n; pc := pc+1

GOTO M_i $\quad\mapsto\quad$ 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	$\mapsto$	x_i := x_j +/- n; pc := pc+1
GOTO M_i	$\mapsto$	pc := i
IF $x_j = 0$ GOTO M_i	$\mapsto$	IF $x_j = 0$ THEN pc:=i ELSE pc:=pc+1 END
HALT	$\mapsto$	pc := 0

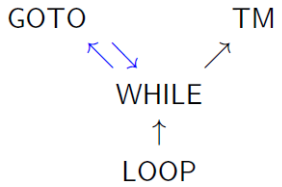
□

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.

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

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

Satz 4.26 (TM → GOTO)

Jede TM kann durch ein GOTO-Programm simuliert werden.

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

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

Eine Konfiguration

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

Satz 4.26 (TM → GOTO)

Jede TM kann durch ein GOTO-Programm simuliert werden.

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

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

Eine Konfiguration

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

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

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

Satz 4.26 (TM → GOTO)

Jede TM kann durch ein GOTO-Programm simuliert werden.

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

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

Eine Konfiguration

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

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

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

wobei $(i_p \ldots i_1)_b$ die Zahl $i_p \ldots i_1$ 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;

Satz 4.26 (TM → GOTO)

Jede TM kann durch ein GOTO-Programm simuliert werden.

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

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

Eine Konfiguration

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

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

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

wobei $(i_p \ldots i_1)_b$ die Zahl $i_p \ldots i_1$ 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};
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 $d = L$:

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 $z \in F$ 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 $d = L$:

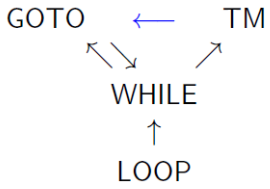
z := r;	Zustand aktualisieren
y := y DIV b;	Löschen von a_j
y := b*y + s;	Schreiben von a_s
y := b*y + (x MOD b);	Bewegung L (I): Einfügen von a_{i_1} in y

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

M: IF $z \in F$ 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
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



Übersetzungen

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

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 4.27 (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

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

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.

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

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

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.

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.

Beweis:

Induktion über den Aufbau der primitiv-rekursiven Funktionen. □

Beispiel 4.32

Alle Konstanten $n \in \mathbb{N}$ sind PR: $1 = s(0)$, $2 = s(1)$, $\ldots$

Beispiel 4.32

Alle Konstanten $n \in \mathbb{N}$ sind PR: $1 = s(0)$, $2 = s(1)$, $\ldots$

Addition ist PR:

$$h(r, x, y) = s(\pi_1^3(r, x, y))$$

Beispiel 4.32

Alle Konstanten $n \in \mathbb{N}$ sind PR: $1 = s(0)$, $2 = s(1)$, $\ldots$

Addition ist PR:

$$\begin{aligned} h(r, x, y) &= s(\pi_1^3(r, x, y)) \\ add(0, y) &= \pi_1^1(y) \\ add(x + 1, y) &= h(add(x, y), x, y) \end{aligned}$$

Beispiel 4.32

Alle Konstanten $n \in \mathbb{N}$ sind PR: $1 = s(0)$, $2 = s(1)$, $\ldots$

Addition ist PR:

$$\begin{aligned} h(r, x, y) &= s(\pi_1^3(r, x, y)) \\ add(0, y) &= \pi_1^1(y) \\ add(x + 1, y) &= h(add(x, y), x, y) \end{aligned}$$

Lesbarer, aber nicht dem syntaktischen PR-Format entsprechend:

$$\begin{aligned} add(0, y) &= y \\ add(x + 1, y) &= s(add(x, y)) \end{aligned}$$

Beispiel 4.32

Alle Konstanten $n \in \mathbb{N}$ sind PR: $1 = s(0)$, $2 = s(1)$, ...

Addition ist PR:

$$\begin{aligned} h(r,x,y) &= s(\pi^3_1(r,x,y)) \\ add(0,y) &= \pi^1_1(y) \\ add(x+1,y) &= h(add(x,y),x,y) \end{aligned}$$

Lesbarer, aber nicht dem syntaktischen PR-Format entsprechend:

$$\begin{aligned} add(0,y) &= y \\ add(x+1,y) &= s(add(x,y)) \end{aligned}$$

Streng genommen also kein Nachweis, dass Addition PR ist.

Beispiel 4.33

Multiplikation ist PR:

$$\begin{aligned} k(y) &= 0 \\ h(r,x,y) &= add(\pi^3_1(r,x,y),\pi^3_3(r,x,y)) \\ mult(0,y) &= k(y) \\ mult(x+1,y) &= h(mult(x,y),x,y) \end{aligned}$$

Lesbarer, aber nicht dem syntaktischen PR-Format entsprechend:

$$\begin{aligned} mult(0,y) &= 0 \\ mult(x+1,y) &= add(mult(x,y),y) \end{aligned}$$

Beispiel 4.33

Multiplikation ist PR:

$$\begin{aligned} k(y) &= 0 \\ h(r,x,y) &= add(\pi^3_1(r,x,y),\pi^3_3(r,x,y)) \\ mult(0,y) &= k(y) \\ mult(x+1,y) &= h(mult(x,y),x,y) \end{aligned}$$

Lesbarer, aber nicht dem syntaktischen PR-Format entsprechend:

$$\begin{aligned} mult(0,y) &= 0 \\ mult(x+1,y) &= add(mult(x,y),y) \end{aligned}$$

In Zukunft: $+$ und $*$ statt add und $mult$.

Definition 4.34

Wir bezeichen f als eine erweiterte Komposition der Funktionen $g_1, \ldots, g_k$ falls

$$f(x_1, \ldots, x_n) = t$$

so dass t ein Ausdruck ist, der nur aus den Funktionen $g_1, \ldots, g_k$ und den Variablen $x_1, \ldots, x_n$ besteht.

Definition 4.34
Wir bezeichen f als eine erweiterte Komposition der Funktionen $g_1, \ldots, g_k$ falls

$$f(x_1, \ldots, x_n) \;=\; t$$

so dass t ein Ausdruck ist, der nur aus den Funktionen $g_1, \ldots, g_k$ und den Variablen $x_1, \ldots, x_n$ besteht.

Beispiel: $f(x, y) = g_1(x, g_2(y, g_3(x)))$

Definition 4.34
Wir bezeichen f als eine erweiterte Komposition der Funktionen $g_1, \ldots, g_k$ falls

$$f(x_1, \ldots, x_n) \;=\; t$$

so dass t ein Ausdruck ist, der nur aus den Funktionen $g_1, \ldots, g_k$ und den Variablen $x_1, \ldots, x_n$ besteht.

Beispiel: $f(x, y) = g_1(x, g_2(y, g_3(x)))$

Lemma 4.35
Eine erweiterte Komposition von PR Funktionen ist wieder PR.

Definition 4.34
Wir bezeichen f als eine erweiterte Komposition der Funktionen $g_1, \ldots, g_k$ falls

$$f(x_1, \ldots, x_n) \;=\; t$$

so dass t ein Ausdruck ist, der nur aus den Funktionen $g_1, \ldots, g_k$ und den Variablen $x_1, \ldots, x_n$ besteht.

Beispiel: $f(x, y) = g_1(x, g_2(y, g_3(x)))$

Lemma 4.35
Eine erweiterte Komposition von PR Funktionen ist wieder PR.

Beweis:
Mit Induktion über Aufbau/Größe von t. □

Definition 4.34
Wir bezeichen f als eine erweiterte Komposition der Funktionen $g_1, \ldots, g_k$ falls

$$f(x_1, \ldots, x_n) \;=\; t$$

so dass t ein Ausdruck ist, der nur aus den Funktionen $g_1, \ldots, g_k$ und den Variablen $x_1, \ldots, x_n$ besteht.

Beispiel: $f(x, y) = g_1(x, g_2(y, g_3(x)))$

Lemma 4.35
Eine erweiterte Komposition von PR Funktionen ist wieder PR.

Beweis:
Mit Induktion über Aufbau/Größe von t. □

Beispiel:

$$\begin{aligned} h_1(x, y) &= g_3(\pi_1^2(x, y)) \\ h_2(x, y) &= g_2(\pi_2^2(x, y), h_1(x, y)) \\ f(x, y) &= g_1(\pi_1^2(x, y), h_2(x, y)) \end{aligned}$$

Das erweiterte Schema der primitiven Rekursion erlaubt

$$\begin{aligned} f(0,\overline{x}) &= t_0 \\ f(m+1,\overline{x}) &= t \end{aligned}$$

wobei

- t_0 enthält nur PR Funktionen und die x_i,

Das erweiterte Schema der primitiven Rekursion erlaubt

$$\begin{aligned} f(0,\overline{x}) &= t_0 \\ f(m+1,\overline{x}) &= t \end{aligned}$$

wobei

- t_0 enthält nur PR Funktionen und die x_i,
- t enthält nur $f(m,\overline{x})$, PR Funktionen, m und die x_i.

Das erweiterte Schema der primitiven Rekursion erlaubt

$$\begin{aligned} f(0,\overline{x}) &= t_0 \\ f(m+1,\overline{x}) &= t \end{aligned}$$

wobei

- t_0 enthält nur PR Funktionen und die x_i,
- t enthält nur $f(m,\overline{x})$, PR Funktionen, m und die x_i.

Lemma 4.36

Das erweiterte Schema der primitiven Rekursion führt nicht aus PR hinaus.

Beweis:

Mit Hilfe der erweiterten Komposition. □

Das erweiterte Schema der primitiven Rekursion erlaubt

$$\begin{aligned} f(0,\overline{x}) &= t_0 \\ f(m+1,\overline{x}) &= t \end{aligned}$$

wobei

- t_0 enthält nur PR Funktionen und die x_i,
- t enthält nur $f(m,\overline{x})$, PR Funktionen, m und die x_i.

Lemma 4.36

Das erweiterte Schema der primitiven Rekursion führt nicht aus PR hinaus.

Beweis:

Mit Hilfe der erweiterten Komposition. □

Moral:

Primitive Rekursion erlaubt

$f(m+1,\overline{x}) = \ldots f(m,\overline{x}) \ldots$

Beispiel 4.37

Die Vorgängerfunktion ist PR:

$$\begin{aligned} pred(0) &= 0 \\ pred(x+1) &= x \end{aligned}$$

Beispiel 4.37

Die Vorgängerfunktion ist PR:

$$\begin{aligned} pred(0) &= 0 \\ pred(x+1) &= x \end{aligned}$$

Die modifizierte Differenz ist PR:

$$\begin{aligned} x \dot{-} 0 &= x \\ x \dot{-} (y+1) &= pred(x \dot{-} y) \end{aligned}$$

Definition 4.38

Sei $P(x)$ ein Prädikat, d.h. ein logischer Ausdruck, der in Abhängigkeit von $x \in \mathbb{N}_0$ den Wert **true** oder **false** liefert.

Definition 4.38

Sei $P(x)$ ein Prädikat, d.h. ein logischer Ausdruck, der in Abhängigkeit von $x \in \mathbb{N}_0$ den Wert **true** oder **false** liefert.

Dann können wir P in natürlicher Weise eine Funktion

$$\hat{P} : \mathbb{N} \to \{0, 1\}$$

zuordnen: $\hat{P}(x) = 1$ gdw $P(x) =$ **true**.

Definition 4.38

Sei $P(x)$ ein Prädikat, d.h. ein logischer Ausdruck, der in Abhängigkeit von $x \in \mathbb{N}_0$ den Wert **true** oder **false** liefert. Dann können wir P in natürlicher Weise eine Funktion

$$\hat{P} : \mathbb{N} \to \{0, 1\}$$

zuordnen: $\hat{P}(x) = 1$ gdw $P(x) =$ **true**.

Wir nennen P primitiv rekursiv genau dann, wenn $\hat{P}$ primitiv rekursiv ist.

Ist P primitiv rekursiv, dann auch der beschränkte max-Operator

$$\max\{x \leq n \mid P(x)\}$$

wobei $\max \emptyset := 0$.

Ist P primitiv rekursiv, dann auch der beschränkte max-Operator

$$\max\{x \leq n \mid P(x)\} \;=:\; q(n)$$

wobei $\max \emptyset := 0$.

$$q(0) \;=\; 0$$

Ist P primitiv rekursiv, dann auch der beschränkte max-Operator

$$\max\{x \leq n \mid P(x)\} \;=:\; q(n)$$

wobei $\max \emptyset := 0$.

$$\begin{aligned} q(0) &= 0 \\ q(n+1) &= q(n) + \hat{P}(n+1) * ((n+1) \dot{-} q(n)) \end{aligned}$$

Ist P primitiv rekursiv, dann auch der beschränkte max-Operator

$$\max\{x \leq n \mid P(x)\} \;=:\; q(n)$$

wobei $\max \emptyset := 0$.

$$\begin{aligned} q(0) &= 0 \\ q(n+1) &= q(n) + \hat{P}(n+1) * ((n+1) \dot{-} q(n)) \\ &= \begin{cases} n+1 & \text{falls } P(n+1) \\ q(n) & \text{sonst} \end{cases} \end{aligned}$$

Ist P primitiv rekursiv, dann auch der beschränkte Existenzquantor

$$\exists x \leq n.\ P(x) \quad =:\; Q(x)$$

denn:

$$\begin{aligned} \hat{Q}(0) &= \hat{P}(0) \\ \hat{Q}(n+1) &= \hat{Q}(n) + \hat{P}(n+1) * (1 \dot{-} \hat{Q}(n)) \end{aligned}$$

4.6 PR = LOOP

Hauptproblem bei LOOP $\rightarrow$ PR:
Kodierung *aller* Variablen eines LOOP Programms in *einer* Zahl.

4.6 PR = LOOP

Hauptproblem bei LOOP $\rightarrow$ PR:
Kodierung *aller* Variablen eines LOOP Programms in *einer* Zahl.

Satz 4.39
Die Cantorsche Paarungsfunktion

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

ist eine Bijektion zwischen $\mathbb{N}^2$ *und* $\mathbb{N}$.

4.6 PR = LOOP

Hauptproblem bei LOOP $\to$ PR:
Kodierung *aller* Variablen eines LOOP Programms in *einer* Zahl.

Satz 4.39
Die Cantorsche Paarungsfunktion

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

ist eine Bijektion zwischen $\mathbb{N}^2$ *und* $\mathbb{N}$.

	0	1	2	3	...
0	0	2	5	9	...
1	1	4	8	13	...
2	3	7	12	18	...
3	6	11	17	24	...
⋮	⋮	⋮	⋮	⋮	⋱

Die Funktion $x \mapsto \binom{x}{2}$ ist PR:

$$\begin{aligned} \binom{0}{2} &= 0 \\ \binom{n+1}{2} &= \binom{n}{2} + n \end{aligned}$$

Die Funktion $x \mapsto \binom{x}{2}$ ist PR:

$$\begin{aligned} \binom{0}{2} &= 0 \\ \binom{n+1}{2} &= \binom{n}{2} + n \end{aligned}$$

Mit Komposition ist auch c PR:

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

Mit c kodiert man $k+1$ Tupel:

$$\langle n_0, n_1, \ldots, n_k \rangle := c(n_0, c(n_1, \ldots, c(n_k, 0)\ldots))$$

4.6 PR = LOOP

Hauptproblem bei LOOP $\rightarrow$ PR:
Kodierung *aller* Variablen eines LOOP Programms in *einer* Zahl.

Satz 4.39
Die Cantorsche Paarungsfunktion

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

ist eine Bijektion zwischen $\mathbb{N}^2$ *und* $\mathbb{N}$.

	0	1	2	3	...
0	0	2	5	9	...
1	1	4	8	13	...
2	3	7	12	18	...
3	6	11	17	24	...
⋮	⋮	⋮	⋮	⋮	⋱