

Berechenbarkeit und Komplexität

4. Vorlesung

Prof. Dr. Dietrich Kuske

FG Automaten und Logik, TU Ilmenau

Sommersemester 2023

While-Programme

Syntaktische Komponenten für While-Programme

wie für Loop-Programme, nur **Schlüsselwort** loop durch while ersetzt

Definition

Ein **While-Programm** ist von der Form

- $x_i := c$, $x_i := x_j + c$, $x_i := x_j \div c$ mit $c \in \{0, 1\}$ und $i, j \geq 1$ (**Wertzuweisung**) oder
- $P_1; P_2$, wobei P_1 und P_2 While-Programme sind (**sequentielle Komposition**) oder
- **while** $x_i \neq 0$ **do** P **end**, wobei P ein While-Programm ist und $i \geq 1$.

Intuition: Programm P wird so oft ausgeführt, bis der Wert von x_i gleich 0 ist.

Definition

Wie bei Loop-Programmen definieren wir zunächst für jedes While-Programm P , in dem keine Variable x_i mit $i > k$ vorkommt, induktiv eine partielle Abbildung $\llbracket P \rrbracket_k : \mathbb{N}^k \rightarrow \mathbb{N}^k$. Hierfür sei $\bar{n} = (n_1, \dots, n_k) \in \mathbb{N}^k$.

- $\llbracket x_i := c \rrbracket_k (\bar{n})$ ist definiert und gleich $(n_1, \dots, n_{i-1}, c, n_{i+1}, \dots, n_k)$.
- $\llbracket x_i := x_j + c \rrbracket_k (\bar{n})$ ist definiert und gleich $(n_1, \dots, n_{i-1}, n_j + c, n_{i+1}, \dots, n_k)$.
- $\llbracket x_i := x_j \div c \rrbracket_k (\bar{n})$ ist definiert und gleich $(n_1, \dots, n_{i-1}, n_j \div c, n_{i+1}, \dots, n_k)$.
- $\llbracket P_1; P_2 \rrbracket_k (\bar{n})$ ist genau dann definiert, wenn $\bar{m} = \llbracket P_1 \rrbracket_k (\bar{n}) \in \mathbb{N}^k$ und $\llbracket P_2 \rrbracket_k (\bar{m})$ definiert sind.

In diesem Falle gilt $\llbracket P_1; P_2 \rrbracket_k (\bar{n}) = \llbracket P_2 \rrbracket_k (\llbracket P_1 \rrbracket_k (\bar{n}))$.

Ansonsten ist $\llbracket P_1; P_2 \rrbracket_k (\bar{n})$ also undefiniert.

- Sei nun $P = \text{while } x_i \neq 0 \text{ do } A \text{ end } (i \leq k)$.

Es gibt zwei Fälle:

- ① Es gibt eine Zahl τ mit $\pi_i^k(\llbracket A \rrbracket_k^\tau(\bar{n})) = 0$.
Dann ist $\llbracket P \rrbracket_k(\bar{n})$ definiert und es gilt $\llbracket P \rrbracket_k(\bar{n}) = \llbracket A \rrbracket_k^\tau(n)$ für die kleinste Zahl τ mit $\pi_i^k(\llbracket A \rrbracket_k^\tau(\bar{n})) = 0$.
- ② Es gibt keine solche Zahl.
Dann ist $\llbracket P \rrbracket_k(\bar{n})$ undefiniert.

Definition

Eine partielle Funktion $f: \mathbb{N}^k \rightarrow \mathbb{N}$ heißt **while-berechenbar**, falls es ein $\ell \geq k$ und ein While-Programm P , in dem höchstens die Variablen x_1, \dots, x_ℓ vorkommen, gibt, so daß für alle $\bar{n} \in \mathbb{N}^k$ gilt:

- $f(\bar{n})$ definiert $\iff \llbracket P \rrbracket_\ell(\bar{n}, 0, \dots, 0)$ definiert
- Falls $f(\bar{n})$ definiert ist, gilt
 $f(\bar{n}) = \pi_1^\ell(\llbracket P \rrbracket_\ell(\bar{n}, 0, \dots, 0))$.

Beachte: Eine loop-Schleife

$$\text{loop } x \text{ do } P \text{ end}$$

ohne die Variable y kann simuliert werden durch

$$y := x; \text{ while } y \neq 0 \text{ do } y := y \div 1; P \text{ end}$$

Lemma

Jede loop-berechenbare Funktion ist auch while-berechenbar.

Ein bißchen Hintergrund

Die While-Vermutung

Eine partielle Funktion $\mathbb{N}^k \rightarrow \mathbb{N}$ ist genau dann intuitiv berechenbar, wenn sie while-berechenbar ist.

Da jede loop-berechenbare Funktion auch while-berechenbar ist, haben wir schon „viele“ while-berechenbare Funktionen. Außerdem kann man zeigen, daß die Ackermann-Funktion while-berechenbar ist.

Auf den Folien 2.2 und 2.7 haben wir gesehen, daß die Klasse der loop-berechenbaren Funktionen unter Substitution und Rekursion abgeschlossen ist, was wir jetzt auch für die Klasse der while-berechenbaren Funktionen zeigen wollen - aber was ist die Substitution von partiellen Funktionen?

Definition

Seien $f: \mathbb{N}^j \rightarrow \mathbb{N}$ und $g_i: \mathbb{N}^k \rightarrow \mathbb{N}$ für alle $1 \leq i \leq j$ (mit $j, k \geq 0$) partielle Funktionen. Sei $h: \mathbb{N}^k \rightarrow \mathbb{N}$ die partielle Funktion mit den Eigenschaften:

- $h(\bar{n})$ ist genau dann definiert ist, wenn $g_i(\bar{n})$ für alle $1 \leq i \leq j$ und $f(g_1(\bar{n}), g_2(\bar{n}), \dots, g_j(\bar{n}))$ definiert sind;
- in diesem Fall gilt $h(\bar{n}) = f(g_1(\bar{n}), g_2(\bar{n}), \dots, g_j(\bar{n}))$ gilt.

Wir schreiben $\text{subst}(f; g_1, \dots, g_j)$ für die Funktion h .

Bemerkung: Sind f und g_1, \dots, g_j total, so stimmt diese Definition mit der von Folie 2.2 überein.

Die Abbildung $(f, g_1, \dots, g_j) \mapsto \text{subst}(f; g_1, \dots, g_j)$ wird als **Substitution** bezeichnet; sind f und g_i intuitiv berechenbar, so sicher auch $\text{subst}(f; g_1, \dots, g_j)$.

Analog zum Beweis des Lemmas auf Folie 2.2 kann gezeigt werden:

Lemma

Sind f und g_1, \dots, g_j while-berechenbar, so auch $\text{subst}(f; g_1, \dots, g_j)$.

Definition

Seien $k \geq 1$ und $g: \mathbb{N}^{k-1} \rightarrow \mathbb{N}$, $h: \mathbb{N}^{k+1} \rightarrow \mathbb{N}$ und $f: \mathbb{N}^{k+1} \rightarrow \mathbb{N}$ partielle Funktionen, so daß das Folgende für alle $\bar{n} \in \mathbb{N}^k$ und $m \in \mathbb{N}$ gilt:

- $f(\bar{n}, 0)$ ist genau dann definiert, wenn $g(\bar{n})$ definiert ist; in diesem Fall gilt $f(\bar{n}, 0) = g(\bar{n})$
- $f(\bar{n}, m + 1)$ ist genau dann definiert, wenn $f(\bar{n}, m)$ und $h(\bar{n}, m, f(\bar{n}, m))$ definiert sind; in diesem Fall gilt $f(\bar{n}, m + 1) = h(\bar{n}, m, f(\bar{n}, m))$.

Dann entsteht f aus g und h mittels **Rekursion**. Hierfür schreiben wir $f = \text{rec}(g, h)$.

Bemerkung: Sind g und h total, so stimmt diese Definition mit der von Folie 2.7 überein.

Analog zum Beweis des Lemmas auf Folie 2.7 kann gezeigt werden:

Lemma

Sind g und h while-berechenbar, so auch $\text{rec}(g, h)$.

Idee einer weiteren Operation auf Klasse aller partiellen Funktionen:

Ist $f: \mathbb{N}^{k+1} \rightarrow \mathbb{N}$ eine partielle Funktion, so soll $g(\bar{n})$ für $\bar{n} \in \mathbb{N}^k$ „die kleinste Nullstelle von $f(\bar{n}, \cdot)$ “ sein:

$$g(\bar{n}) = \begin{cases} \min\{m \mid f(\bar{n}, m) \text{ definiert und } f(\bar{n}, m) = 0\} & \text{falls Menge } \neq \emptyset \\ \text{undefiniert} & \text{sonst.} \end{cases}$$

Beispiel:

- $f(0, 0) = 0, f(0, 1) = 0 \implies g(0) = 0$
- $f(1, 0) = 1, f(1, 1) = 0 \implies g(1) = 1$
- $f(2, 0)$ undefiniert, $f(2, 1) = 0$: um festzustellen, daß $g(2) = 1$ gilt, muß zunächst festgestellt werden, daß $f(2, 0) \neq 0$ gilt.

Problem: Ist f intuitiv berechenbar, so muß dies nicht unbedingt für g gelten.

Daher verwenden wir nicht obige Definition, sondern die folgende:

Definition

Sei $f: \mathbb{N}^{k+1} \rightarrow \mathbb{N}$ eine partielle Funktion. Dann sei $g := \text{search}_f: \mathbb{N}^k \rightarrow \mathbb{N}$ die partielle Funktion mit

- $g(\bar{n})$ ist genau dann definiert, wenn es $\ell \in \mathbb{N}$ gibt, so daß $f(\bar{n}, 0)$, $f(\bar{n}, 1)$, \dots , $f(\bar{n}, \ell)$ definiert sind und $f(\bar{n}, \ell) = 0$;
- in diesem Fall ist $g(\bar{n})$ das kleinste $\ell \in \mathbb{N}$ mit $f(\bar{n}, \ell) = 0$.

Bemerkung

- Ist f intuitiv berechenbar, so auch search_f .
- In der Literatur wird die partielle Funktion search_f mit μf bezeichnet.

Beispiel

- Sei $f: \mathbb{N}^2 \rightarrow \mathbb{N}: (n, m) \mapsto n + m + 1$. Dann gilt $f(n, m) > 0$ für alle $n, m \in \mathbb{N}$, also ist $\text{search}_f(n)$ undefiniert für alle $n \in \mathbb{N}$. Wir bezeichnen diese nirgendwo definierte Funktion $\text{search}_f: \mathbb{N} \rightarrow \mathbb{N}$ mit Ω .
- Sei $f: \mathbb{N}^2 \rightarrow \mathbb{N}: (n, m) \mapsto n \div m \cdot m$. Für $n, m \in \mathbb{N}$ gilt dann $f(n, m) = 0 \iff n \leq m^2$, also

$$\text{search}_f(n) = \lceil \sqrt{n} \rceil.$$

Lemma

Ist $f: \mathbb{N}^{k+1} \rightarrow \mathbb{N}$ while-berechenbar, so auch $\text{search}_f: \mathbb{N}^k \rightarrow \mathbb{N}$.

Beweis: Die Funktion search_f läßt sich durch das folgende (Pseudocode-) While-Programm berechnen:

$m := 0; y := f(x_1, \dots, x_{k-1}, m);$

$\text{while } y \neq 0 \text{ do } m := m + 1; y := f(x_1, x_2, \dots, x_{k-1}, m); \text{ end;}$

$x_1 := m$



zwei mögliche Ursachen für die Nicht-Termination dieses While-Programms: Nicht-Termination der while-Schleife oder der Berechnung eines $f(x_1, \dots, x_{k-1}, m)$.

Wir haben also Argumente der Form (K^+) und (A^+) (vgl. Folie 1.24) und damit gute Gründe, der While-Vermutung zu trauen.

Ein bißchen Geschichte (vgl. Folie 1.22)

Kurt Gödel, der ebenso wie Hilbert an der Frage interessiert war, was ein „Verfahren“ sei, kannte keine While-Programme, sondern formulierte die folgende Vermutung:

Gödels Vermutung (ca. 1934)

Eine partielle Funktion $\mathbb{N}^k \rightarrow \mathbb{N}$ ist genau dann intuitiv berechenbar, wenn sie μ -rekursiv ist.

Wir werden zeigen, daß Gödels Vermutung äquivalent zur While-Vermutung ist (für deren Gültigkeit wir ja gute Gründe haben).

μ -rekursive Funktionen

Die While-Programme verallgemeinern die Loop-Programme, die ja für einfache imperative Programmiersprachen standen.

Jetzt werden wir die rek-Programme, die für einfache funktionale Programmiersprachen stehen, ebenfalls verallgemeinern und dann zeigen, daß sie äquivalent zu den While-Programmen sind.

Definition

- CONST_0 ist ein 0-stelliges μrek -Programm
- S ist ein 1-stelliges μrek -Programm
- PROJ_i^k ist ein k -stelliges μrek -Programm (für alle $1 \leq i \leq k$)
- Sind F ein j - und G_1, \dots, G_j k -stellige μrek -Programme ($j, k \geq 0$), so ist $\text{SUBST}(F; G_1, \dots, G_j)$ ein k -stelliges μrek -Programm.
- Sind G ein $(k - 1)$ - und H ein $(k + 1)$ -stelliges μrek -Programm ($k \geq 1$), so ist $\text{REC}(G, H)$ ein k -stelliges μrek -Programm.
- Ist F ein $(k + 1)$ -stelliges μrek -Programm, so ist $\text{SEARCH}(F)$ ein k -stelliges μrek -Programm.
- Nichts ist μrek -Programm, was sich nicht mittels obiger Regeln erzeugen läßt.

Wir werden jetzt jedem k -stelligen μ rek-Programm F eine partielle Funktion $\llbracket F \rrbracket : \mathbb{N}^k \rightarrow \mathbb{N}$ zuordnen:

- Es gelte $\llbracket \text{CONST}_0 \rrbracket () = 0$. Also ist

$$\llbracket \text{CONST}_0 \rrbracket = \text{const}_0 : \mathbb{N}^0 \rightarrow \mathbb{N}$$

die überall definierte konstante 0-Funktion const_0 .

- Für $n \in \mathbb{N}$ sei $\llbracket S \rrbracket (n) = n + 1$. Also ist $\llbracket S \rrbracket : \mathbb{N} \rightarrow \mathbb{N}$ die überall definierte Nachfolgerfunktion.
- Für $\bar{n} = (n_1, \dots, n_k) \in \mathbb{N}^k$ sei $\llbracket \text{PROJ}_i^k \rrbracket (\bar{n}) = n_i$. Also ist

$$\llbracket \text{PROJ}_i^k \rrbracket = \pi_i^k : \mathbb{N}^k \rightarrow \mathbb{N}$$

die überall definierte Projektionsfunktion π_i^k der k -Tupel auf den i -ten Eintrag.

- Für $\bar{n} \in \mathbb{N}^k$
 - sei $\llbracket \text{SUBST}(F; G_1, \dots, G_j) \rrbracket (\bar{n})$ genau dann definiert, wenn $\llbracket G_i \rrbracket (\bar{n})$ f.a. i und $\llbracket F \rrbracket (\llbracket G_1 \rrbracket (\bar{n}), \dots, \llbracket G_j \rrbracket (\bar{n}))$ definiert sind und
 - in diesem Fall gelte

$$\llbracket \text{SUBST}(F; G_1, \dots, G_j) \rrbracket (\bar{n}) = \llbracket F \rrbracket (\llbracket G_1 \rrbracket (\bar{n}), \dots, \llbracket G_j \rrbracket (\bar{n})).$$

Also ist

$$\llbracket \text{SUBST}(F; G_1, \dots, G_j) \rrbracket = \text{subst}(\llbracket F \rrbracket; \llbracket G_1 \rrbracket, \dots, \llbracket G_j \rrbracket)$$

die Substitution der partiellen Funktionen $\llbracket G_1 \rrbracket, \dots, \llbracket G_j \rrbracket$ in die partielle Funktion $\llbracket F \rrbracket$ (vgl. Folie 4.7).

- Für $\bar{n} \in \mathbb{N}^k$
 - ist $\llbracket \text{SEARCH}(F) \rrbracket (\bar{n})$ genau dann definiert, wenn es $\ell \in \mathbb{N}$ gibt, so daß $\llbracket F \rrbracket (\bar{n}, i)$ für alle $i \in \{0, 1, \dots, \ell\}$ definiert ist und $\llbracket F \rrbracket (\bar{n}, \ell) = 0$ gilt;
 - in diesem Fall ist $\llbracket \text{SEARCH}(F) \rrbracket (\bar{n})$ das kleinste ℓ mit $\llbracket F \rrbracket (\bar{n}, \ell) = 0$.

Also gilt

$$\llbracket \text{SEARCH}(F) \rrbracket = \text{search}_{\llbracket F \rrbracket}$$

(vgl. Folie 4.10).

- Für $\bar{n} \in \mathbb{N}^{k-1}$ und $m \in \mathbb{N}$
 - ist $\llbracket \text{REC}(G, H) \rrbracket(\bar{n}, 0)$ genau dann definiert, wenn $\llbracket G \rrbracket(\bar{n})$ definiert ist und
 - in diesem Fall gilt $\llbracket \text{REC}(G, H) \rrbracket(\bar{n}, 0) = \llbracket G \rrbracket(\bar{n})$
 - ist $\llbracket \text{REC}(G, H) \rrbracket(\bar{n}, m+1)$ genau dann definiert, wenn $\llbracket \text{REC}(G, H) \rrbracket(\bar{n}, m)$ und $\llbracket H \rrbracket(\bar{n}, m, \llbracket \text{REC}(G, H) \rrbracket(\bar{n}, m))$ definiert sind und
 - in diesem Fall gilt

$$\llbracket \text{REC}(G, H) \rrbracket(\bar{n}, m+1) = \llbracket H \rrbracket(\bar{n}, m, \llbracket \text{REC}(G, H) \rrbracket(\bar{n}, m)).$$

Also entsteht

$$\llbracket \text{REC}(G, H) \rrbracket = \text{rec}(\llbracket G \rrbracket, \llbracket H \rrbracket)$$

aus den partiellen Funktionen $\llbracket G \rrbracket$ und $\llbracket H \rrbracket$ durch Rekursion (vgl. Folie 4.8).

Definition

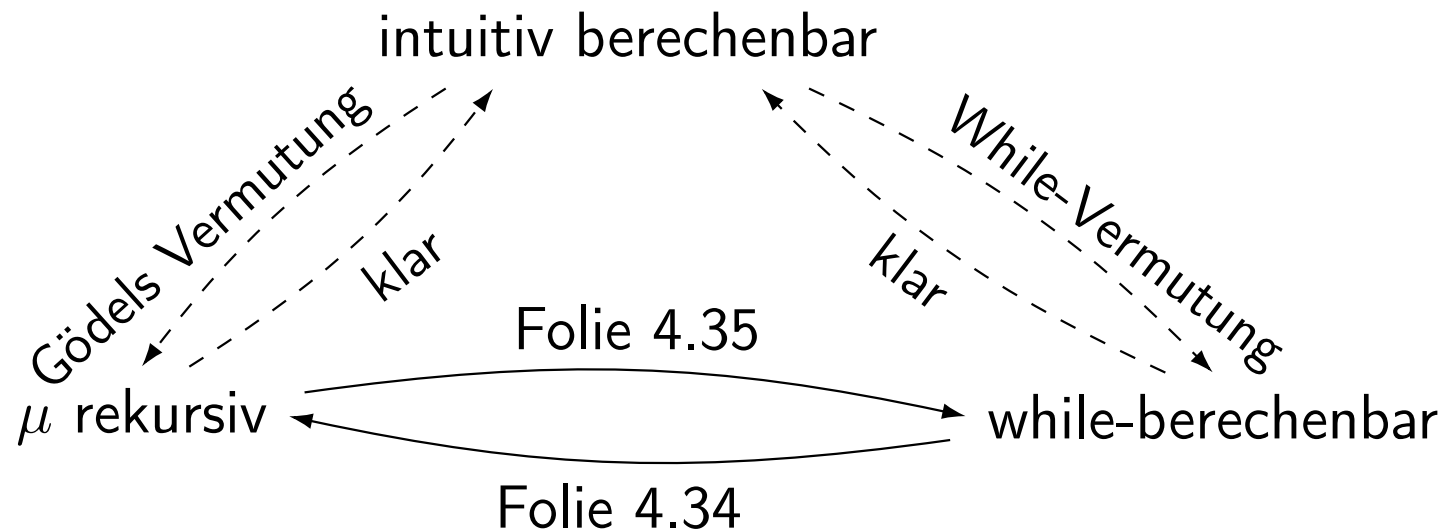
Eine partielle Funktion $f: \mathbb{N}^k \rightarrow \mathbb{N}$ ist **μ rek-berechenbar** oder **μ -rekursiv**, wenn es ein k -stelliges μ rek-Programm F gibt mit $\llbracket F \rrbracket = f$.

Satz

Eine partielle Funktion $f: \mathbb{N}^k \rightarrow \mathbb{N}$ ist genau dann while-berechenbar, wenn sie μ -rekursiv ist.

Beweis: Hierfür müssen die Beweise der Sätze auf den Folien 2.19 bzw. 2.27 um Programme der Form $\text{SEARCH}(F)$ bzw. um die while-Schleife erweitert werden (siehe Zusatzmaterial dieser Vorlesung). □

Zwischenbilanz



(U⁺) zwei äquivalente Vorschläge, den Begriff „intuitiv berechenbar“ zu formalisieren: while-berechenbar und μ -rekursiv

(K⁺) viele Funktionen sind while-berechenbar bzw. μ -rekursiv

(A⁺) die beiden Klassen erfüllen viele Abschlußeigenschaften

Damit haben wir gute Gründe, der While-Vermutung (Folie 4.6) bzw. Gödels Vermutung (Folie 4.13) zuzustimmen, oder?

Goto-Programme

- Die While-Programme abstrahieren strukturierte („höhere“) Programmiersprachen, d.h. while-berechenbare Funktionen sind durch Programme einer „höheren“ Programmiersprache berechenbar.
- μ -rekursive Funktionen sind durch Programme einer funktionalen Programmiersprache berechenbar.

Frage

Können Assembler- oder Maschinensprachen vielleicht mehr Funktionen berechnen?

Definition

Ein **Goto-Programm** ist eine endliche nichtleere Folge $P = A_1; A_2; \dots; A_m$ von Anweisungen A_i der folgenden Form:

- $x_i := c$, $x_i := x_j + c$, $x_i := x_j \div c$ mit $c \in \{0, 1\}$ und $i, j \geq 1$
(**Wertzuweisung**)
- goto ℓ mit $1 \leq \ell \leq m + 1$ (**unbedingter Sprung**)
- if $x_i = 0$ then ℓ mit $i \geq 1$ und $1 \leq \ell \leq m + 1$ (**bedingter Sprung**)

Definition

Sei $P = A_1; A_2; \dots; A_m$ ein Goto-Programm, in dem keine Variable x_i mit $i > k$ vorkommt.

Eine **Konfiguration von P** ist ein $(k + 1)$ -Tupel

$(n_1, n_2, \dots, n_k, p) \in \mathbb{N}^k \times \{1, \dots, m + 1\}$, wobei n_i die Belegung der Variablen x_i und p den Wert des Programmzählers beschreibt.

Definition

Seien $P = A_1; A_2; \dots; A_m$ ein Goto-Programm und $(\bar{n}, p), (\bar{n}', p')$ zwei Konfigurationen. Wir setzen $(\bar{n}, p) \vdash_P (\bar{n}', p')$, falls $p < m + 1$ und eine der folgenden Bedingungen gilt:

- $A_p = (x_i := c)$, $\bar{n}' = (n_1, \dots, n_{i-1}, c, n_{i+1}, \dots, n_k)$ und $p' = p + 1$
- $A_p = (x_i := x_j + c)$, $\bar{n}' = (n_1, \dots, n_{i-1}, n_i + c, n_{i+1}, \dots, n_k)$ und $p' = p + 1$
- $A_p = (x_i := x_j \div c)$, $\bar{n}' = (n_1, \dots, n_{i-1}, n_i \div c, n_{i+1}, \dots, n_k)$ und $p' = p + 1$
- $A_p = (\text{goto } \ell)$, $\bar{n}' = \bar{n}$ und $p' = \ell$
- $A_p = (\text{if } x_i = 0 \text{ then } \ell)$, $n_i = 0$, $\bar{n}' = \bar{n}$, $p' = \ell$
- $A_p = (\text{if } x_i = 0 \text{ then } \ell)$, $n_i \neq 0$, $\bar{n}' = \bar{n}$, $p' = p + 1$

Definition

Für jedes Goto-Programm $P = A_1; A_2; \dots; A_m$, in dem keine Variable x_i mit $i > k$ vorkommt, definieren wir zunächst eine partielle Funktion

$\llbracket P \rrbracket_k : \mathbb{N}^k \rightarrow \mathbb{N}^k$: Sei $\bar{n} \in \mathbb{N}^k$.

$\llbracket P \rrbracket_k(\bar{n})$ ist definiert, falls es $\bar{n}' \in \mathbb{N}^k$ gibt mit $(\bar{n}, 1) \vdash_P^* (\bar{n}', m+1)$.
In diesem Fall gilt $\llbracket P \rrbracket_k(\bar{n}) = \bar{n}'$.

Definition

Eine partielle Funktion $f: \mathbb{N}^k \rightarrow \mathbb{N}$ heißt **goto-berechenbar**, falls es ein $\ell \geq k$ und ein Goto-Programm P , in dem keine Variable x_i mit $i > \ell$ vorkommt, gibt, so daß für alle $\bar{n} \in \mathbb{N}^k$ gilt:

- $f(\bar{n})$ definiert $\iff \llbracket P \rrbracket_\ell(\bar{n}, 0, \dots, 0)$ definiert
- Falls $f(\bar{n})$ definiert ist, gilt $f(\bar{n}) = \pi_1^\ell(\llbracket P \rrbracket_\ell(\bar{n}, 0, \dots, 0))$.

Lemma

Jede while-berechenbare Funktion ist goto-berechenbar.

Beweisidee:

Eine while-Schleife

`while $x \neq 0$ do P end`

kann simuliert werden durch das Goto-Programm-Stück

$A_p = (\text{if } x = 0 \text{ then } q + 1)$

\vdots Übersetzung des While-Programms P

$A_q = \text{goto } p$

\vdots Übersetzung des restlichen While-Programms



Lemma

Jede goto-berechenbare Funktion ist while-berechenbar.

Beweis:

Sei $P = (A_1; \dots; A_m)$ ein Goto-Programm.

Wir simulieren P durch das folgende While-Programm Q mit nur einer while-Schleife:

```
count := 1;  
while count  $\neq$   $m + 1$  do  
    if count = 1 then  $A'_1$  end;  
    if count = 2 then  $A'_2$  end;  
     $\vdots$   
    if count =  $m$  then  $A'_m$  end;  
end
```

Hierbei ist das While-Programm A'_i in Abhängigkeit von A_i wie folgt definiert:

- $A'_i = (x_j := c; \text{count} := \text{count} + 1)$ falls $A_i = (x_j := c)$
- $A'_i = (x_j := x_\ell + c; \text{count} := \text{count} + 1)$ falls $A_i = (x_j := x_\ell + c)$
- $A'_i = (x_j := x_\ell \div c; \text{count} := \text{count} + 1)$ falls $A_i = (x_j := x_\ell \div c)$
- $A'_i = (\text{count} := k)$ falls $A_i = (\text{goto } k)$
- $A'_i =$ (if $x_j = 0$ then $\text{count} := \ell$ end;
if $x_j \neq 0$ then $\text{count} := \text{count} + 1$ end)
falls $A_i = (\text{if } x_j = 0 \text{ then } \ell)$



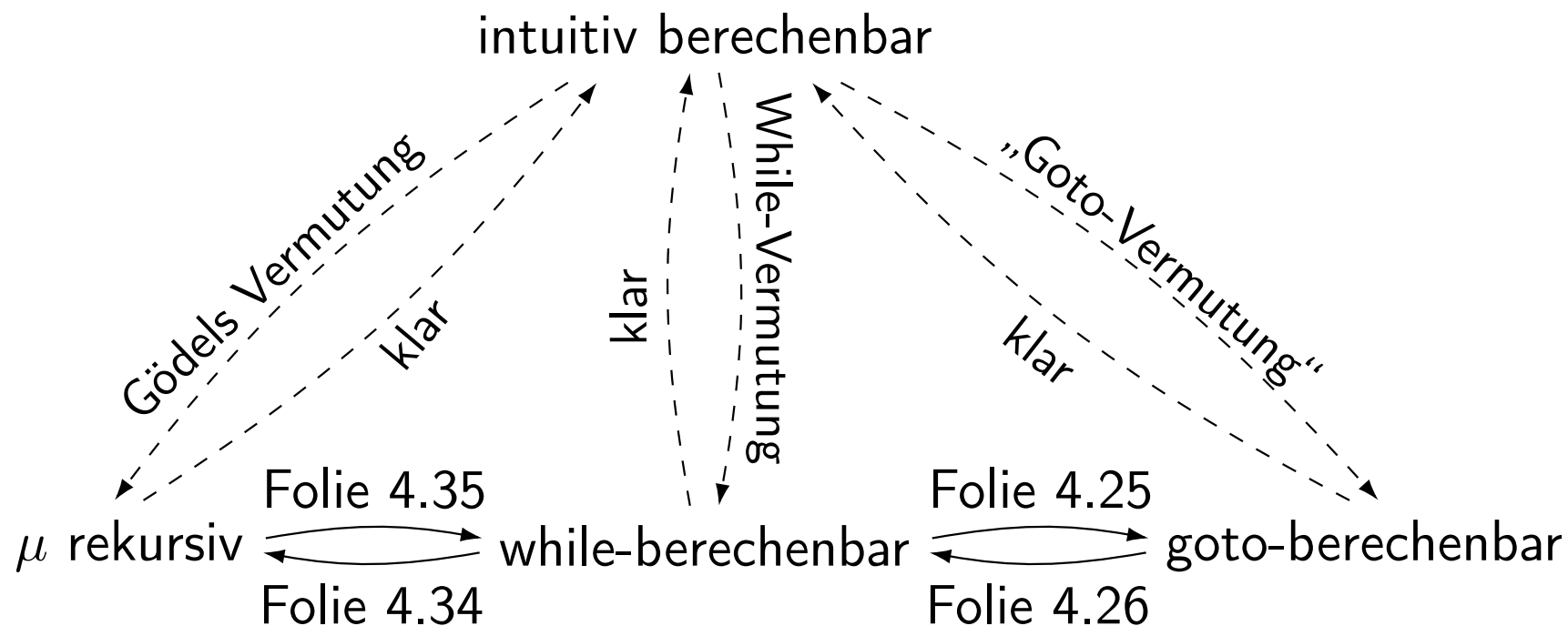
Ein kleiner Ausflug - Kleenesche Normalform

Die Simulation von Goto-Programmen durch While-Programme verwendet nur eine while-Schleife (falls man `if ... then` als elementares Konstrukt erlaubt).

Das bedeutet: Ein While-Programm kann durch Umwandlung in ein Goto-Programm und Zurückumwandlung in ein While-Programm in ein **äquivalentes While-Programm mit nur einer While-Schleife** umgewandelt werden (Kleenesche Normalform für While- Programme).

Die analoge Aussage für Loop-Programme gilt nicht (siehe Übung).

Zwischenbilanz



- (U⁺) drei äquivalente Vorschläge, den Begriff „intuitiv berechenbar“ zu formalisieren: while-berechenbar, μ -rekursiv und goto-berechenbar
- (K⁺) viele Funktionen sind while-berechenbar, μ -rekursiv bzw. goto-berechenbar
- (A⁺) die drei Klassen erfüllen viele Abschlußeigenschaften

Damit haben wir gute Gründe, der While-Vermutung (Folie 4.6), Gödels Vermutung (Folie 4.13) bzw. der „Goto-Vermutung“ zuzustimmen, oder?

In der kommenden Vorlesung werden wir noch ein völlig neues Argument für diese Vermutungen behandeln ...

Zusammenfassung 4. Vorlesung

in dieser Vorlesung neu

- while- und goto-berechenbare Funktionen, μ -rekursive Funktionen
- Äquivalenz von While-, Goto- und Gödels Vermutung

kommende Vorlesung

- Modellierung der Tätigkeit des Rechnens durch Turing-Maschinen

Zusatzmaterial

Lemma

Jede while-berechenbare partielle Funktion ist μ -rekursiv.

Beweis:

Es genügt, den Beweis des Satzes auf Folie 2.27 um die while-Schleife zu erweitern.

Sei also $P = (\text{while } x_i \neq 0 \text{ do } Q \text{ end})$ ein While-Programm, in dem die Variablen x_j für $j > k$ nicht vorkommen.

Wir müssen zeigen, daß $\llbracket P \rrbracket_k: \mathbb{N} \rightarrow \mathbb{N}: n \mapsto \left\langle \llbracket P \rrbracket_k (d_1(n), \dots, d_k(n)) \right\rangle$ μ -rekursiv ist (vgl. Folie 2.26).

Nach Induktion ist dies für $\llbracket Q \rrbracket_k$ bereits der Fall.

Die Funktion $h: \mathbb{N}^2 \rightarrow \mathbb{N}: (n, m) \mapsto \langle\langle Q \rangle\rangle_k^m(n)$ ist μ -rekursiv (vgl. Beweis des Lemmas auf Folie 2.27 (Fall 4))

Dann ist auch $k: \mathbb{N}^2 \rightarrow \mathbb{N}: (n, m) \mapsto d_i(h(n, m))$ μ -rekursiv

und es gilt $\langle\langle P \rangle\rangle_k(n) = h(n, \mu k(n))$, d.h. auch $\langle\langle P \rangle\rangle_k$ ist μ -rekursiv. □

Lemma

Jede μ -rekursive Funktion ist while-berechenbar.

Beweis: Hierzu müssen wir den Beweis des Lemmas auf Folie 2.19 um den $\text{SEARCH}(F)$ erweitern.

Das haben wir aber bereits auf Folie 4.12 getan. □