Berechenbarkeit und Komplexität
4. Vorlesung

TECHNISCHE UNIVERSITÄT ILMENAU

Prof. Dr. Dietrich Kuske

FG Automaten und Logik, TU Ilmenau

Sommersemester 2024

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

- $x_i := c$, $x_i := x_j + c$, $x_i := x_j \dot{-} c$ 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 $[\![P]\!]_k : \mathbb{N}^k \dashrightarrow \mathbb{N}^k$. Hierfür sei $\overline{n} = (n_1, \ldots, n_k) \in \mathbb{N}^k$.

- $[\![x_i := c]\!]_k(\overline{n})$ ist definiert und gleich $(n_1, \ldots, n_{i-1}, c, n_{i+1}, \ldots, n_k)$.
- $[\![x_i := x_j + c]\!]_k(\overline{n})$ ist definiert und gleich $(n_1, \ldots, n_{i-1}, n_j + c, n_{i+1}, \ldots, n_k)$.
- $[\![x_i := x_j \dot{-} c]\!]_k(\overline{n})$ ist definiert und gleich $(n_1, \ldots, n_{i-1}, n_j \dot{-} c, n_{i+1}, \ldots, n_k)$.
- $[\![P_1; P_2]\!]_k(\overline{n})$ ist genau dann definiert, wenn $\overline{m} = [\![P_1]\!]_k(\overline{n}) \in \mathbb{N}^k$ und $[\![P_2]\!]_k(\overline{m})$ definiert sind.

  In diesem Falle gilt $[\![P_1; P_2]\!]_k(\overline{n}) = [\![P_2]\!]_k\big([\![P_1]\!]_k(\overline{n})\big)$.

  Ansonsten ist $[\![P_1; P_2]\!]_k(\overline{n})$ also undefiniert.

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

  Es gibt zwei Fälle:

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

Definition

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

- $f(\overline{n})$ definiert $\iff$ $[\![P]\!]_\ell(\overline{n}, 0, \ldots, 0)$ definiert
- Falls $f(\overline{n})$ definiert ist, gilt $f(\overline{n}) = \pi_1^\ell\big([\![P]\!]_\ell(\overline{n}, 0, \ldots, 0)\big)$.

Beachte: Eine `loop`-Schleife

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

ohne die Variable y kann simuliert werden durch

$$y := x;\ \texttt{while}\ y \neq 0\ \texttt{do}\ y := y \dot{-} 1;\ P\ \texttt{end}$$

Lemma

Jede loop-berechenbare Funktion ist auch while-berechenbar.

Ein bißchen Hintergrund

Die While-Vermutung

Eine partielle Funktion $\mathbb{N}^k \dashrightarrow \mathbb{N}$ 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 <u>partiellen</u> Funktionen?

Definition

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

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

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

Bemerkung: Sind f und $g_1, \ldots, g_j$ total, so stimmt diese Definition mit der von Folie 2.2 überein.
Die Abbildung $(f, g_1, \ldots, g_j) \mapsto \mathrm{subst}(f; g_1, \ldots, g_j)$ wird als Substitution bezeichnet; sind f und g_i intuitiv berechenbar, so sicher auch $\mathrm{subst}(f; g_1, \ldots, g_j)$.
Analog zum Beweis des Lemmas auf Folie 2.2 kann gezeigt werden:

Lemma

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

Definition

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

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

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

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 $\mathrm{rec}(g, h)$.

Idee einer weiteren Operation auf Klasse aller partiellen Funktionen:
Ist $f\colon \mathbb{N}^{k+1} \dashrightarrow \mathbb{N}$ eine partielle Funktion, so soll $g(\overline{n})$ für $\overline{n} \in \mathbb{N}^k$ „die kleinste Nullstelle von $f(\overline{n}, .)$“ sein:

$$g(\overline{n}) = \begin{cases} \min\left\{ m \;\middle|\; f(\overline{n}, m) \text{ definiert und } f(\overline{n}, m) = 0 \right\} & \text{falls Menge } \neq \varnothing \\ \text{undefiniert} & \text{sonst.} \end{cases}$$

Beispiel:

- $f(0,0) = 0,\ f(0,1) = 0 \Longrightarrow g(0) = 0$
- $f(1,0) = 1,\ f(1,1) = 0 \Longrightarrow g(1) = 1$
- $f(2,0)$ 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 $f\colon \mathbb{N}^{k+1} \dashrightarrow \mathbb{N}$ eine partielle Funktion. Dann sei $g := \mathrm{search}_f\colon \mathbb{N}^k \dashrightarrow \mathbb{N}$ die partielle Funktion mit

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

Bemerkung

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

Beispiel

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

$$\mathrm{search}_f(n) = \left\lceil \sqrt{n} \right\rceil .$$

Lemma

Ist $f\colon \mathbb{N}^{k+1} \dashrightarrow \mathbb{N}$ while-berechenbar, so auch $\mathrm{search}_f\colon \mathbb{N}^k \dashrightarrow \mathbb{N}$.

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

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

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

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 $\mathbb{N}^k \dashrightarrow \mathbb{N}$ 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

- $\mathtt{CONST}_0$ ist ein 0-stelliges μrek-Programm
- S ist ein 1-stelliges μrek-Programm
- $\mathtt{PROJ}_i^k$ ist ein k-stelliges μrek-Programm (für alle $1 \leq i \leq k$)
- Sind F ein j- und $G_1, \ldots, G_j$ k-stellige μrek-Programme ($j, k \geq 0$), so ist $\mathtt{SUBST}(F; G_1, \ldots, G_j)$ ein k-stelliges μrek-Programm.
- Sind G ein $(k-1)$- und H ein $(k+1)$-stelliges μrek-Programm ($k \geq 1$), so ist $\mathtt{REC}(G, H)$ ein k-stelliges μrek-Programm.
- Ist F ein $(k+1)$-stelliges μrek-Programm, so ist $\mathtt{SEARCH}(F)$ ein k-stelliges μrek-Programm.
- Nichts ist μrek-Programm, was sich nicht mittels obiger Regeln erzeugen läßt.

Wir werden jetzt jedem k-stelligen μrek-Programm F eine partielle Funktion $[\![F]\!] : \mathbb{N}^k \dashrightarrow \mathbb{N}$ zuordnen:

- Es gelte $[\![\mathtt{CONST}_0]\!]\,() = 0$. Also ist

$$[\![\mathtt{CONST}_0]\!] = \mathrm{const}_0 : \mathbb{N}^0 \dashrightarrow \mathbb{N}$$

  die überall definierte konstante 0-Funktion const_0.
- Für $n \in \mathbb{N}$ sei $[\![S]\!]\,(n) = n + 1$. Also ist $[\![S]\!] : \mathbb{N} \dashrightarrow \mathbb{N}$ die überall definierte Nachfolgerfunktion.
- Für $\overline{n} = (n_1, \ldots, n_k) \in \mathbb{N}^k$ sei $[\![\mathtt{PROJ}_i^k]\!]\,(\overline{n}) = n_i$. Also ist

$$[\![\mathtt{PROJ}_i^k]\!] = \pi_i^k : \mathbb{N}^k \dashrightarrow \mathbb{N}$$

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

- Für $\overline{n} \in \mathbb{N}^k$
  - sei $[\![\mathtt{SUBST}(F; G_1, \ldots, G_j)]\!](\overline{n})$ genau dann definiert, wenn $[\![G_i]\!](\overline{n})$ f.a. i und $[\![F]\!]\big([\![G_1]\!](\overline{n}), \ldots, [\![G_j]\!](\overline{n})\big)$ definiert sind und
  - in diesem Fall gelte $[\![\mathtt{SUBST}(F; G_1, \ldots, G_j)]\!](\overline{n}) = [\![F]\!]\big([\![G_1]\!](\overline{n}), \ldots, [\![G_j]\!](\overline{n})\big)$.

  Also ist

  $$\big[\!\big[\mathtt{SUBST}(F; G_1, \ldots, G_j)\big]\!\big] = \mathrm{subst}\big([\![F]\!]; [\![G_1]\!], \ldots, [\![G_j]\!]\big)$$

  die Substitution der partiellen Funktionen $[\![G_1]\!], \ldots, [\![G_j]\!]$ in die partielle Funktion $[\![F]\!]$ (vgl. Folie 4.7).
- Für $\overline{n} \in \mathbb{N}^k$
  - ist $[\![\mathtt{SEARCH}(F)]\!](\overline{n})$ genau dann definiert, wenn es $\ell \in \mathbb{N}$ gibt, so daß $[\![F]\!](\overline{n}, i)$ für alle $i \in \{0, 1, \ldots, \ell\}$ definiert ist und $[\![F]\!](\overline{n}, \ell) = 0$ gilt;
  - in diesem Fall ist $[\![\mathtt{SEARCH}(F)]\!](\overline{n})$ das kleinste ℓ mit $[\![F]\!](\overline{n}, \ell) = 0$.

  Also gilt

  $$[\![\mathtt{SEARCH}(F)]\!] = \mathrm{search}_{[\![F]\!]}$$

  (vgl. Folie 4.10).

- Für $\overline{n} \in \mathbb{N}^{k-1}$ und $m \in \mathbb{N}$
  -
    - ist $[\![\mathtt{REC}(G,H)]\!](\overline{n},0)$ genau dann definiert, wenn $[\![G]\!](\overline{n})$ definiert ist und
    - in diesem Fall gilt $[\![\mathtt{REC}(G,H)]\!](\overline{n},0) = [\![G]\!](\overline{n})$
  -
    - ist $[\![\mathtt{REC}(G,H)]\!](\overline{n},m+1)$ genau dann definiert, wenn $[\![\mathtt{REC}(G,H)]\!](\overline{n},m)$ und $[\![H]\!]\big(\overline{n},m,[\![\mathtt{REC}(G,H)]\!](\overline{n},m)\big)$ definiert sind und
    - in diesem Fall gilt $[\![\mathtt{REC}(G,H)]\!](\overline{n},m+1) = [\![H]\!]\big(\overline{n},m,[\![\mathtt{REC}(G,H)]\!](\overline{n},m)\big)$.

  Also ensteht

$$[\![\mathtt{REC}(G,H)]\!] = \mathrm{rec}\big([\![G]\!],[\![H]\!]\big)$$

  aus den partiellen Funktionen $[\![G]\!]$ und $[\![H]\!]$ durch Rekursion (vgl. Folie 4.8).

Definition

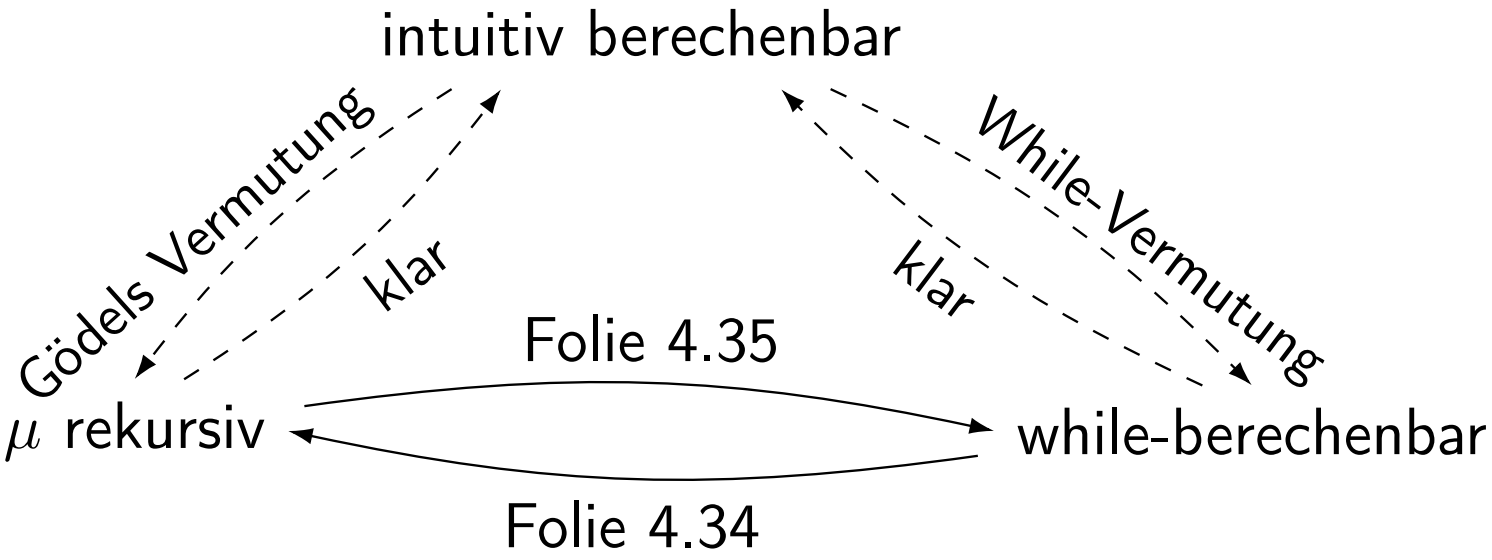
Eine partielle Funktion $f\colon \mathbb{N}^k \dashrightarrow \mathbb{N}$ ist μrek-berechenbar oder μ-rekursiv, wenn es ein k-stelliges μrek-Programm F gibt mit $[\![F]\!] = f$.

Satz

Eine partielle Funktion $f\colon \mathbb{N}^k \dashrightarrow \mathbb{N}$ 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 $\texttt{SEARCH}(F)$ bzw. um die while-Schleife erweitert werden (siehe Zusatzmaterial dieser Vorlesung). $\square$

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 $P = A_1; A_2; \ldots; A_m$ von Anweisungen A_i der folgenden Form:

- $x_i := c$, $x_i := x_j + c$, $x_i := x_j \dot{-} c$ 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 $P = A_1; A_2; \ldots; A_m$ ein Goto-Programm, in dem keine Variable x_i mit $i > k$ vorkommt.
Eine Konfiguration von P ist ein $(k + 1)$-Tupel $(n_1, n_2, \ldots, n_k, p) \in \mathbb{N}^k \times \{1, \ldots, m + 1\}$, wobei n_i die Belegung der Variablen x_i und p den Wert des Programmzählers beschreibt.

Definition

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

- $A_p = (x_i := c)$, $\overline{n}' = (n_1, \ldots, n_{i-1}, c, n_{i+1}, \ldots, n_k)$ und $p' = p + 1$
- $A_p = (x_i := x_j + c)$, $\overline{n}' = (n_1, \ldots, n_{i-1}, n_j + c, n_{i+1}, \ldots, n_k)$ und $p' = p + 1$
- $A_p = (x_i := x_j \dot{-} c)$, $\overline{n}' = (n_1, \ldots, n_{i-1}, n_j \dot{-} c, n_{i+1}, \ldots, n_k)$ und $p' = p + 1$
- $A_p = (\texttt{goto}\ \ell)$, $\overline{n}' = \overline{n}$ und $p' = \ell$
- $A_p = (\texttt{if}\ x_i = 0\ \texttt{then}\ \ell)$, $n_i = 0$, $\overline{n}' = \overline{n}$, $p' = \ell$
- $A_p = (\texttt{if}\ x_i = 0\ \texttt{then}\ \ell)$, $n_i \neq 0$, $\overline{n}' = \overline{n}$, $p' = p + 1$

Definition

Für jedes Goto-Programm $P = A_1; A_2; \ldots; A_m$, in dem keine Variable x_i mit $i > k$ vorkommt, definieren wir zunächst eine partielle Funktion $[\![P]\!]_k : \mathbb{N}^k \dashrightarrow \mathbb{N}^k$: Sei $\overline{n} \in \mathbb{N}^k$.

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

Definition

Eine partielle Funktion $f : \mathbb{N}^k \dashrightarrow \mathbb{N}$ 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 $\overline{n} \in \mathbb{N}^k$ gilt:

- $f(\overline{n})$ definiert $\iff$ $[\![P]\!]_\ell(\overline{n}, 0, \ldots, 0)$ definiert
- Falls $f(\overline{n})$ definiert ist, gilt $f(\overline{n}) = \pi_1^\ell\big([\![P]\!]_\ell(\overline{n}, 0, \ldots, 0)\big)$.

Lemma

Jede while-berechenbare Funktion ist goto-berechenbar.

Beweisidee:
Eine `while`-Schleife

$$\texttt{while } x \neq 0 \texttt{ do } P \texttt{ end}$$

kann simuliert werden durch das Goto-Programm-Stück

$$\begin{array}{rcl} A_p & = & (\texttt{if } x = 0 \texttt{ then } q+1) \\ \vdots & & \text{Übersetzung des While-Programms } P \\ A_q & = & \texttt{goto } p \\ \vdots & & \text{Übersetzung des restlichen While-Programms} \end{array}$$

□

Lemma

Jede goto-berechenbare Funktion ist while-berechenbar.

Beweis:

Sei $P = (A_1; \ldots A_m)$ 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:

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

$\square$

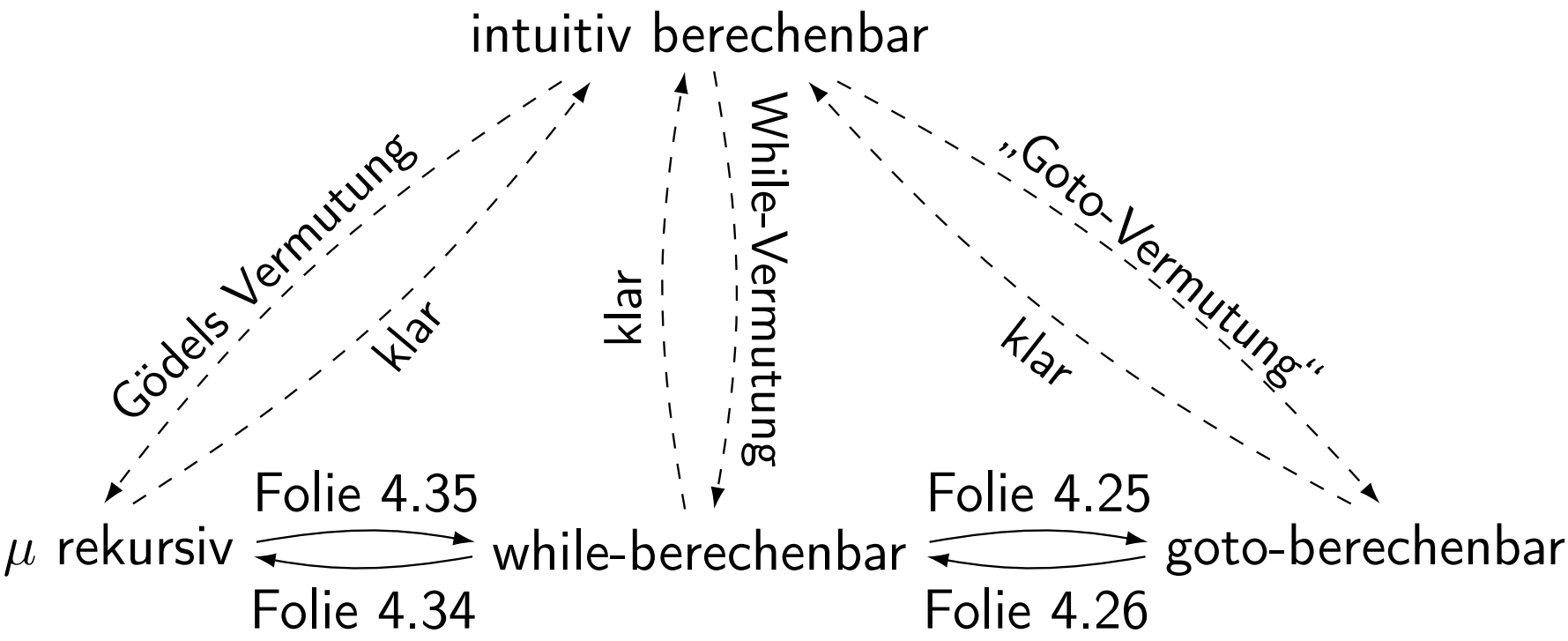
Ein kleiner Ausflug - Kleenesche Normalform

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

Wir müssen zeigen, daß $\langle\!\langle P \rangle\!\rangle_k \colon \mathbb{N} \dashrightarrow \mathbb{N} \colon n \mapsto \left\langle [\![P]\!]_k \left(d_1(n), \ldots, d_k(n) \right) \right\rangle$ μ-rekursiv ist (vgl. Folie 2.26).

Nach Induktion ist dies für $\langle\!\langle Q \rangle\!\rangle_k$ bereits der Fall.

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

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

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

Lemma

Jede μ-rekursive Funktion ist while-berechenbar.

Beweis: Hierzu müssen wir den Beweis des Lemmas auf Folie 2.19 um den Fall $\texttt{SEARCH}(F)$ erweitern.
Das haben wir aber bereits auf Folie 4.12 getan. $\square$