
Grundlagen der theoretischen Informatik

Kurt Sieber

Fachbereich Mathematik/Theoretische Informatik
Universität Siegen

Vorlesungen vom 10.01.2005 und 11.01.2005 (Stand: 13.01.2005)

Berechenbarkeit

Wie 'schwierig' sind die Probleme aus Satz 3.19?
Sind sie wenigstens semi-entscheidbar?

Satz 3.20

1. H_0 ist semi-entscheidbar.
2. H_{\exists} ist semi-entscheidbar.
3. H_{\forall} ist **nicht** semi-entscheidbar.
4. *Equiv* ist **nicht** semi-entscheidbar.

Beweis:

1. Ein möglicher Semi-Entscheidungsalgorithmus für H_0 sieht so aus:
Bei Eingabe n führt man π_n für die Eingabe 0 aus. Falls es hält, gibt man 1 aus.

Alternative Argumentation: H_0 ist auf H reduzierbar, denn es gilt $H_0 = f^{-1}(H)$, wobei $f : \mathbb{N} \rightarrow \mathbb{N}$ definiert ist durch $f(n) = (n, 0)$.

Berechenbarkeit

Also folgt die Unentscheidbarkeit von H_0 aus der Unentscheidbarkeit von H .

2. Ein möglicher Semi-Entscheidungsalgorithmus für H_{\exists} sieht so aus: Bei Eingabe n führt man π_n für alle Eingaben $m \in \mathbb{N}$ 'parallel' aus (*dovetailing*). Sobald eine Eingabe m gefunden ist, für die es hält, gibt man 1 aus.
3. Wir zeigen durch einen Diagonalisierungsbeweis, dass H_{\forall} nicht rekursiv aufzählbar, und damit auch nicht semi-entscheidbar ist.

Angenommen, H_{\forall} ist rekursiv aufzählbar, d.h. es existiert eine totale berechenbare Funktion $f : \mathbb{N} \rightarrow \mathbb{N}$ mit $f(\mathbb{N}) = H_{\forall}$.

Dann betrachten wir den Algorithmus π , der für jede Eingabe $n \in \mathbb{N}$ wie folgt arbeitet:

- Zuerst wird $f(n)$ berechnet.
- Dann wird das Programm $\pi_{f(n)}$ für die Eingabe n ausgeführt.

Berechenbarkeit

- Dann wird das Ergebnis um 1 erhöht und ausgegeben.

Der Algorithmus π hält für *jede* Eingabe $n \in \mathbb{N}$, denn:

- Die Berechnung von $f(n)$ hält, weil f eine totale Funktion ist.
- Das Programm $\pi_{f(n)}$ hält für alle Eingabewerte, weil $f(n) \in H_{\forall}$, also hält es insbesondere für die Eingabe n .

Damit ist gezeigt, dass π eine totale Funktion berechnet, d.h. π müsste mit einem der Programme $\pi_{f(n)}$ übereinstimmen.

Das ist aber nicht möglich, da sich—per Definition des Algorithmus π —die Resultate von π und $\pi_{f(n)}$ bei Eingabe n um 1 unterscheiden.

4. Im Beweis von Satz 3.19 haben wir schon gesehen, dass sich H_{\forall} auf *Equiv* reduzieren lässt.

Also würde aus der Semi-Entscheidbarkeit von *Equiv* die Semi-Entscheidbarkeit von H_{\forall} folgen im Widerspruch zu 3. \square

Berechenbarkeit

Die bisherigen Überlegungen haben gezeigt, dass viele *semantische Eigenschaften* von Programmen unentscheidbar sind. Dabei verstehen wir unter einer semantischen Eigenschaft eines Programms π eine Eigenschaft, die sich nur auf die von π berechnete *Funktion* bezieht. Es stellt sich die Frage, ob es auch entscheidbare semantische Eigenschaften gibt.

Satz 3.21 (Satz von Rice) Sei S eine *echte, nichtleere* Teilmenge der Menge aller partiellen berechenbaren Funktionen $f : \mathbb{N}^k \hookrightarrow \mathbb{N}$. Dann ist die Menge

$$\text{Prog}_S = \{n \in \mathbb{N} \mid \text{die von } \pi_n \text{ berechnete Funktion liegt in } S\}$$

unentscheidbar.

Der Satz von Rice besagt, dass jede ‘nichttriviale semantische Eigenschaft’ von Programmen unentscheidbar ist. Prog_S enthält nämlich genau die (Gödelnummern der) Programme, die die semantische Eigenschaft S haben, wobei nur die ‘trivialen’ Eigenschaften $S = \emptyset$ und $S = \{f : \mathbb{N}^k \hookrightarrow \mathbb{N} \mid f \text{ berechenbar}\}$ ausgeschlossen sind.

Berechenbarkeit

Anwendungsbeispiele für den Satz von Rice sind unsere bisherigen Unentscheidbarkeits-Resultate für die Mengen

- H_0 (wähle $S = \{f : \mathbb{N} \hookrightarrow \mathbb{N} \mid f \text{ berechenbar und } 0 \in \text{Def}(f)\}$)
- H_{\exists} (wähle $S = \{f : \mathbb{N} \hookrightarrow \mathbb{N} \mid f \text{ berechenbar und } \text{Def}(f) \neq \emptyset\}$)
- H_{\forall} (wähle $S = \{f : \mathbb{N} \hookrightarrow \mathbb{N} \mid f \text{ berechenbar und } \text{Def}(f) = \mathbb{N}\}$)

Die Unentscheidbarkeit von *Equiv* ergibt sich nicht direkt aus dem Satz von Rice, weil sich *Equiv* ja auf *Paare* von Funktionen bezieht. Stattdessen kann man zeigen, dass ein *spezielles Äquivalenzproblem* unentscheidbar ist, z.B. das Problem

$$\text{Equiv}_0 = \{n \in \mathbb{N} \mid \pi_n \text{ berechnet die Funktion } \text{const}_0\}$$

(wähle $S = \{\text{const}_0\}$). Daraus ergibt sich dann wieder die Unentscheidbarkeit von *Equiv*, weil sich *Equiv*₀ auf *Equiv* reduzieren lässt.

Berechenbarkeit

Gegenbeispiele:

- Die Menge

$$\{n \in \mathbb{N} \mid \pi_n \text{ hält für Eingabe } 0 \text{ nach höchstens } 10 \text{ Schritten}\}$$

ist entscheidbar.

Die in der Menge genannte Eigenschaft bezieht sich ja nicht (nur) auf die von π berechnete Funktion, sondern auf das Programm π selbst. Deshalb ist der Satz von Rice hier *nicht* anwendbar.

Ein Entscheidungsalgorithmus für diese Menge besteht einfach darin, dass man die ersten 10 Schritte der Berechnung durchführt, und dann die entsprechende Antwort ausgibt.

- Mit der gleichen Argumentation erhält man die Entscheidbarkeit der Menge

$$H' = \{(n, m, k) \in \mathbb{N}^3 \mid \pi_n \text{ hält für Eingabe } m \\ \text{nach höchstens } k \text{ Schritten}\}$$

Berechenbarkeit

Unseren bisherigen Betrachtungen zur Berechenbarkeitstheorie sind etwas vage, weil wir den Begriff *Algorithmus* nicht präzise definiert haben.

Bisher verstehen wir unter einem Algorithmus ein 'Programm in einer hinreichend mächtigen Programmiersprache'.

Dabei haben wir (in den Beweisen) gewisse Annahmen über diese Programmiersprache gemacht, z.B.

- Zu zwei Programmen π_1, π_2 gibt es ein Programm π , das die beiden Programme (auf dem gleichen Eingabewert) 'parallel' ausführt.
- Zu jedem Programm π gibt es ein Programm π' , das π auf allen möglichen Eingabewerten 'parallel' ausführt (mittels dovetailing).
- Jedes Programm lässt sich als natürliche Zahl codieren.
- *In* der Programmiersprache kann man einen Interpreter *für* die Programmiersprache schreiben.

Berechenbarkeit

Es bleibt zu zeigen, dass eine solche Programmiersprache existiert. Dazu gibt es viele unterschiedliche Ansätze, von denen wir die folgenden drei betrachten:

1. Die Sprache der *while*-Programme.
2. Die Sprache der μ -rekursiven Funktionen.
3. Die Sprache der Turing-Programme (= 'Turing-Maschinen').
 1. ist eine kleine imperative Programmiersprache mit Zuweisungen, Komposition, bedingten Anweisungen und *while*-Schleifen, (also eine kleine Teilmenge von Sprachen wie Pascal, C, Java).
 2. ähnelt einer funktionalen Programmiersprache.
 3. ist ein Maschinenmodell, bei dem sehr kleine Berechnungsschritte auf einem (mit Zeichen beschriebenen) unendlichen Band ausgeführt werden.

Berechenbarkeit

Es wird sich herausstellen, dass diese drei Programmiersprachen die gleiche Mächtigkeit haben, d.h. dass sie zum gleichen Begriff von *Berechenbarkeit* führen. Solche Programmiersprachen nennt man *Turing-mächtig*.

Bisher wurde keine Programmiersprache gefunden, mit der sich mehr Funktionen berechnen lassen als mit den oben genannten. Das gibt Anlass zu folgender Vermutung.

Churchsche These: Jede intuitiv berechenbare Funktion lässt sich mit einer Turing-Maschine (oder einem *while*-Programm oder einer μ -rekursiven Funktion) berechnen.

Man kann die Churchsche These nicht beweisen, da sie sich auf einen intuitiven Begriff bezieht. Es könnte höchstens sein, dass sie eines Tages widerlegt wird, indem man eine mächtigere Programmiersprache findet. Aber auch das würde die Berechenbarkeitstheorie nicht zum Einsturz bringen (weil man in den Beweisen ja immer nur annimmt, dass die Programmiersprache mächtig genug ist).

Berechenbarkeit

Syntax von *while*-Programmen

Vorgegeben sei eine abzählbar unendliche Menge $Loc = \{X_0, X_1, \dots\}$, deren Elemente wir *Speicherplätze* nennen.

Ein *while*-Programm (zur Berechnung einer k -stelligen Funktion) hat die Form

$$\text{read } X_{i_1}, \dots, X_{i_k}; \text{ stl write } X_i$$

wobei die X_{i_j} verschieden sind und *stl* eine Anweisungsliste ist. Anweisungen *st* und Anweisungslisten *stl* sind definiert durch:

$$\begin{aligned} \text{st} \quad ::= & \quad X_i := 0 \mid X_i := X_j \mid X_i := \text{succ}(X_j) \mid X_i := \text{pred}(X_j) \\ & \mid \text{if } X_i \text{ then } \text{stl}_1 \text{ else } \text{stl}_2 \text{ fi} \\ & \mid \text{while } X_i \text{ do } \text{stl}_0 \text{ od} \end{aligned}$$
$$\begin{aligned} \text{stl} \quad ::= & \quad \varepsilon \\ & \mid \text{st}; \text{stl} \end{aligned}$$

Berechenbarkeit

Semantik von *while*-Programmen

while-Programme arbeiten auf einem 'Speicher'. Jede Anweisung (oder Anweisungsliste) verändert den Inhalt einiger Speicherplätze, d.h. sie verändert den 'Speicherzustand'.

Definition 3.22

- Ein Speicherzustand (oder kurz: Zustand) ist eine totale Funktion $\sigma : Loc \rightarrow \mathbb{N}$.
- Der Speicherzustand $\sigma_0 : Loc \rightarrow \mathbb{N}$ sei definiert durch $\sigma_0(X_i) = 0$ für alle $X_i \in Loc$.
- Mit *Store* bezeichnen wir die Menge aller Speicherzustände.

Intuition:

$\sigma(X_i)$ ist der Inhalt des Speicherplatzes X_i im Zustand σ . Den speziellen Speicherzustand σ_0 benötigen wir, um den Anfangszustand eines Programms zu beschreiben.

Berechenbarkeit

Um Veränderungen des Speicherzustands auszudrücken, benötigen wir die folgende Schreibweise.

Definition 3.23 Für $\sigma \in \text{Store}$, $X_i \in \text{Loc}$ und $n \in \mathbb{N}$ sei

$$\sigma[n/X_i] : \text{Loc} \rightarrow \mathbb{N}$$

$$X_i \mapsto n$$

$$X_j \mapsto \sigma(X_j) \text{ für alle } j \neq i$$

Intuition:

$\sigma[n/X_i]$ ist der Speicherzustand, der sich aus σ ergibt, indem man den Inhalt von X_i durch n überschreibt (also der Zustand, der durch eine Zuweisung $X_i := n$ aus σ entstehen würde).

Berechenbarkeit

Den Ablauf eines *while*-Programms beschreiben wir durch Übergangsschritte zwischen Konfigurationen. In einer Konfiguration merken wir uns

- den aktuellen Speicherzustand,
- die Anweisungsliste, die noch abzuarbeiten ist.

Definition 3.24 Eine **Konfiguration** ist ein Paar (stl, σ) , wobei *stl* eine Anweisungsliste ist und $\sigma \in \text{Store}$.

Auf der Menge aller Konfigurationen definieren wir eine Relation \vdash (die **Übergangsschrittrelation**). Wir schreiben wieder \vdash^n , \vdash^+ und \vdash^* für die n -te Potenz, den transitiven Abschluss und den reflexiven transitiven Abschluss von \vdash .

Berechenbarkeit

\vdash ist definiert als die kleinste Relation mit folgenden Eigenschaften:

- $(X_i := 0; stl, \sigma) \vdash (stl, \sigma[0/X_i])$
- $(X_i := X_j; stl, \sigma) \vdash (stl, \sigma[\sigma(X_j)/X_i])$
- $(X_i := succ(X_j); stl, \sigma) \vdash (stl, \sigma[\sigma(X_j) + 1/X_i])$
- $(X_i := pred(X_j); stl, \sigma) \vdash (stl, \sigma[\sigma(X_j) \dot{-} 1/X_i])$
wobei $m \dot{-} n = m - n$ falls $m \geq n$ und $m \dot{-} n = 0$ sonst
- $(\text{if } X_i \text{ then } stl_1 \text{ else } stl_2 \text{ fi}; stl, \sigma) \vdash (stl_1; stl, \sigma)$ falls $\sigma(X_i) \neq 0$
- $(\text{if } X_i \text{ then } stl_1 \text{ else } stl_2 \text{ fi}; stl, \sigma) \vdash (stl_2; stl, \sigma)$ falls $\sigma(X_i) = 0$
- $(\text{while } X_i \text{ do } stl_0 \text{ od}; stl, \sigma) \vdash (stl_0; \text{while } X_i \text{ do } stl_0 \text{ od}; stl, \sigma)$
falls $\sigma(X_i) \neq 0$
- $(\text{while } X_i \text{ do } stl_0 \text{ od}; stl, \sigma) \vdash (stl, \sigma)$ falls $\sigma(X_i) = 0$

Berechenbarkeit

Die von einem *while*-Programm

$$P = \text{read } X_{i_1}, \dots, X_{i_k}; \text{ stl write } X_i$$

berechnete Funktion $f : \mathbb{N}^k \hookrightarrow \mathbb{N}$ ist definiert durch:

$$f(n_1, \dots, n_k) = \begin{cases} \sigma(X_i) & \text{falls } (\text{stl}, \sigma_0[n_1/X_{i_1}] \dots [n_k/X_{i_k}]) \vdash^* (\varepsilon, \sigma) \\ \text{undefiniert} & \text{falls kein solches } \sigma \text{ existiert} \end{cases}$$

In Worten:

- Durch *read* X_{i_1}, \dots, X_{i_k} werden die X_{i_j} mit den Eingabewerten n_j besetzt und alle anderen Speicherplätze mit 0. Das liefert den Anfangszustand $\sigma_0[n_1/X_{i_1}] \dots [n_k/X_{i_k}]$.
- In diesem Anfangszustand wird die Anweisungsliste *stl* ausgeführt.
- Wenn deren Ausführung terminiert, d.h. wenn sich irgendwann eine Konfiguration der Form (ε, σ) ergibt, in der *stl* vollständig abgearbeitet ist, dann wird durch *write* X_i der Wert $\sigma(X_i)$ als Ergebnis der Funktion f abgeliefert. Andernfalls ist f undefiniert.

Berechenbarkeit

Beispiel:

Sei P das Programm

`read X_0, X_1 ;`

`while X_0 do $X_0 := pred(X_0)$; $X_1 := succ(X_1)$ od;`

`write X_1`

Welches Ergebnis liefert dieses Programm für die Eingabe (2,5)?

Dazu betrachten wir die Abarbeitung der Anweisungsliste im Anfangszustand $\sigma_0[2/X_0][5/X_1]$.

(`while X_0 do $X_0 := pred(X_0)$; $X_1 := succ(X_1)$; od;`,

$\sigma_0[2/X_0][5/X_1]$)

\vdash (`$X_0 := pred(X_0)$; $X_1 := succ(X_1)$; while X_0 do ... od;`,

$\sigma_0[2/X_0][5/X_1]$)

Berechenbarkeit

- ⊢ $(X_1 := succ(X_1); \text{while } X_0 \text{ do } \dots \text{ od}; ,$
 $\sigma_0[1/X_0][5/X_1])$
- ⊢ $(\text{while } X_0 \text{ do } X_0 := pred(X_0); X_1 := succ(X_1); \text{od}; ,$
 $\sigma_0[1/X_0][6/X_1])$
- ⊢ $(X_0 := pred(X_0); X_1 := succ(X_1); \text{while } X_0 \text{ do } \dots \text{ od}; ,$
 $\sigma_0[1/X_0][6/X_1])$
- ⊢ $(X_1 := succ(X_1); \text{while } X_0 \text{ do } \dots \text{ od}; ,$
 $\sigma_0[0/X_0][6/X_1])$
- ⊢ $(\text{while } X_0 \text{ do } X_0 := pred(X_0); X_1 := succ(X_1); \text{od}; ,$
 $\sigma_0[0/X_0][7/X_1])$
- ⊢ $(\varepsilon, \sigma_0[0/X_0][7/X_1])$

Berechenbarkeit

Für den Endzustand $\sigma = \sigma_0[0/X_0][7/X_1]$ gilt $\sigma(X_1) = 7$.

Also gilt für die vom Programm P berechnete Funktion f :

$$f(2, 5) = 7$$

Definition 3.25

- Die vom *while*-Programm P berechnete Funktion bezeichnen wir mit $\llbracket P \rrbracket$.
- Eine Funktion $f : \mathbb{N}^k \hookrightarrow \mathbb{N}$ heißt **while-berechenbar**, wenn es ein *while*-Programm P (für k -stellige Funktionen) gibt mit $\llbracket P \rrbracket = f$.

Vermutung: Unser Beispiel-Programm berechnet die Funktion

$$\text{plus} : \mathbb{N}^2 \rightarrow \mathbb{N}$$

$$(m, n) \mapsto m + n$$

Also ist *plus* while-berechenbar.

Berechenbarkeit

Mit Hilfe unserer formalen Semantik können wir eine solche Aussage über ein *while*-Programm beweisen. Dazu formulieren wir eine geeignete Behauptung über die im Programm vorkommende *while*-Schleife: Für alle $\sigma \in Store$ gilt

$$\begin{aligned} & (\text{while } X_0 \text{ do } X_0 := \text{pred}(X_0); X_1 := \text{succ}(X_1); \text{od};, \sigma) \\ & \vdash^* (\varepsilon, \sigma[0/X_0][\sigma(X_0) + \sigma(X_1)/X_1]) \end{aligned} \quad (*)$$

Diese Behauptung beweisen wir durch Induktion über $\sigma(X_0)$.

$\sigma(X_0) = 0$:

In diesem Falle gilt

$$(\text{while } X_0 \text{ do } X_0 := \text{pred}(X_0); X_1 := \text{succ}(X_1); \text{od};, \sigma) \vdash (\varepsilon, \sigma)$$

und wegen $\sigma(X_0) = 0$ ist

$$\sigma = \sigma[0/X_0][\sigma(X_0) + \sigma(X_1)/X_1]$$

d.h. σ ist der in (*) verlangte Endzustand.

Berechenbarkeit

$\sigma(X_0) > 0$:

In diesem Falle gilt

(**while** X_0 **do** $X_0 := \text{pred}(X_0); X_1 := \text{succ}(X_1);$ **od**;;, σ)

$\vdash (X_0 := \text{pred}(X_0); X_1 := \text{succ}(X_1);$ **while** X_0 **do** ... **od**;;, σ)

$\vdash^2 (\text{while } X_0 \text{ do } \dots \text{ od};, \underbrace{\sigma[\sigma(X_0) - 1/X_0][\sigma(X_1) + 1/X_1]}_{\sigma'})$

Nun ist $\sigma'(X_0) = \sigma(X_0) - 1 < \sigma(X_0)$, also gilt die Behauptung (*) nach Induktionsannahme für den Zustand σ' , d.h.

(**while** X_0 **do** ... **od**;;, σ') \vdash^* (ε , $\sigma'[0/X_0][\sigma'(X_0) + \sigma'(X_1)/X_1]$)

Durch Zusammensetzen der beiden Berechnungsfolgen erhalten wir

(**while** X_0 **do** ... **od**;;, σ) \vdash^* (ε , $\underbrace{\sigma'[0/X_0][\sigma'(X_0) + \sigma'(X_1)/X_1]}_{\sigma''}$)

also bleibt zu zeigen, dass σ'' der in (*) verlangte Endzustand ist.

Berechenbarkeit

Dazu betrachten wir die Werte $\sigma''(X_i)$ für alle $X_i \in Loc$:

- $\sigma''(X_0) = 0$
- $\sigma''(X_1) = \sigma'(X_0) + \sigma'(X_1) = \sigma(X_0) - 1 + \sigma(X_1) + 1 = \sigma(X_0) + \sigma(X_1)$
- $\sigma''(X_i) = \sigma'(X_i) = \sigma(X_i)$ für alle $i \geq 2$

Also ist $\sigma'' = \sigma[0/X_0][\sigma(X_0) + \sigma(X_1)/X_1]$, d.h. (*) ist bewiesen.

Insbesondere gilt dann für alle $(m, n) \in \mathbb{N}^2$:

$(\text{while } X_0 \text{ do } \dots \text{ od};, \sigma_0[m/X_0][n/X_1]) \vdash^* (\varepsilon, \sigma_0[0/X_0][m + n/X_1])$

also berechnet das Programm

$P = \text{read } X_0, X_1; \text{ while } X_0 \text{ do } \dots \text{ od}; \text{ write } X_1$

tatsächlich die Funktion $plus : \mathbb{N}^2 \rightarrow \mathbb{N}$ mit $plus(m, n) = m + n$. □

Berechenbarkeit

Um uns von der Mächtigkeit der *while*-Programmiersprache zu überzeugen, überlegen wir uns, wie wir einen Interpreter *für* *while*-Programme *in* der *while*-Programmiersprache schreiben können.

Ein Interpreter für *while*-Programme der Stelligkeit k ist ein *while*-Programm I^k , das für jede Eingabe $(m, n_1, \dots, n_k) \in \mathbb{N}^{k+1}$ die Ausführung des *while*-Programms mit Gödelnummer m auf der Eingabe (n_1, \dots, n_k) simuliert.

Eine solche Simulation können wir durchführen, indem wir uns an unserer formalen Semantik für *while*-Programme orientieren.

Deshalb codieren wir nicht nur *while*-Programme als natürliche Zahlen, sondern auch Anweisungen, Anweisungslisten und Speicherzustände.

Als ersten Schritt in diese Richtung überlegen wir uns, wie man eine endliche Liste n_1, \dots, n_k von natürlichen Zahlen als eine einzige Zahl $\langle n_1, \dots, n_k \rangle \in \mathbb{N}$ codiert.

Berechenbarkeit

Wir nutzen dazu die Tatsache aus, dass jede natürliche Zahl eine eindeutige Primfaktorzerlegung besitzt, also eine eindeutige Darstellung als Produkt von Primzahlpotenzen.

Sei p_1, p_2, \dots die Folge *aller* Primzahlen in aufsteigender Reihenfolge, also $p_1 = 2, p_2 = 3, p_3 = 5, \dots$

Dann codieren wir jede Liste n_1, \dots, n_k natürlicher Zahlen als die Zahl

$$\langle n_1, \dots, n_k \rangle = \prod_{i=1}^k p_i^{n_i+1}$$

z.B. gilt $\langle 3, 0, 2, 0 \rangle = 2^4 * 3^1 * 5^3 * 7^1 = 42000$.

Man beachte, dass man aus der *Zahl* $\langle n_1, \dots, n_k \rangle$ stets die *Liste* der Zahlen n_1, \dots, n_k rekonstruieren kann. Wegen der Eindeutigkeit der Primfaktorzerlegung gilt nämlich

- $k = \max \{i \in \mathbb{N} \mid p_i \text{ ist Teiler von } \langle n_1, \dots, n_k \rangle\}$
 - $n_i = \max \{n \in \mathbb{N} \mid p_i^{n+1} \text{ ist Teiler von } \langle n_1, \dots, n_k \rangle\}$
-