
Grundlagen der theoretischen Informatik

Kurt Sieber

Fachbereich Mathematik/Theoretische Informatik
Universität Siegen

Vorlesung vom 31.01.2005 (Stand: 01.02.2005)

Berechenbarkeit

An der induktiven Definition sieht man im Prinzip schon, dass sich $\mu^{\leq}(g)$ durch primitive Rekursion und Fallunterscheidung auf g und andere primitiv rekursive Funktionen zurückführen lässt.

Für einen exakten Beweis führen wir weitere Umformungen durch.

Für den Induktionsanfang gilt

$$\begin{aligned} & \mu^{\leq}(g)(\bar{n}, 0) \\ &= \begin{cases} \text{const}_0^k(\bar{n}) & \text{falls } g(\text{proj}_1^k(\bar{n}), \dots, \text{proj}_k^k(\bar{n}), \text{const}_0^k(\bar{n})) = 0 \\ \text{const}_1^k(\bar{n}) & \text{sonst} \end{cases} \\ &= g'(\bar{n}) \end{aligned}$$

wobei $g' = \text{If}(Sub(g; \text{proj}_1^k, \dots, \text{proj}_k^k, \text{const}_0^k); \text{const}_1^k, \text{const}_0^k)$ primitiv rekursiv ist, weil g primitiv rekursiv ist.

Berechenbarkeit

Für den Induktionsschritt definieren wir zunächst die Hilfsfunktion

$$f : \mathbb{N}^{k+1} \rightarrow \mathbb{N}$$
$$(\bar{n}, m) \mapsto \begin{cases} m & \text{falls } g(\bar{n}, m) = 0 \\ m + 1 & \text{sonst} \end{cases}$$

Es gilt $f = If(g; succ \circ proj_{k+1}^{k+1}, proj_{k+1}^{k+1})$, also ist f primitiv rekursiv, weil g primitiv rekursiv ist.

Berechenbarkeit

Mit Hilfe von f können wir den Induktionsschritt umformen zu

$$\begin{aligned}\mu^{\leq}(g)(\bar{n}, m+1) &= \begin{cases} \mu^{\leq}(g)(\bar{n}, m) & \text{falls } \mu^{\leq}(g)(\bar{n}, m) \dot{-} m = 0 \\ f(\bar{n}, m+1) & \text{sonst} \end{cases} \\ &= h(\bar{n}, m, \mu^{\leq}(g)(\bar{n}, m))\end{aligned}$$

wobei

$$\begin{aligned}h &= \text{If}(\text{Sub}(\text{subtr}; \text{proj}_{k+2}^{k+2}, \text{proj}_{k+1}^{k+2}); \\ &\quad \text{Sub}(f; \text{proj}_1^{k+2}, \dots, \text{proj}_k^{k+2}, \text{succ} \circ \text{proj}_{k+1}^{k+2}), \\ &\quad \text{proj}_{k+2}^{k+2}).\end{aligned}$$

h ist primitiv rekursiv, weil subtr und f primitiv rekursiv sind. Also ist schließlich auch $\mu^{\leq}(g)$ primitiv rekursiv, weil $\mu^{\leq}(g) = \text{Prim}(g', h)$. \square

Berechenbarkeit

Wir beweisen jetzt die folgenden Äquivalenzen zwischen den unterschiedlichen Berechenbarkeits-Begriffen:

μ -rekursiv = *while*-berechenbar

primitiv rekursiv = *loop*-berechenbar

Die Beweise dieser Äquivalenzen sind *konstruktiv*, d.h.:

- Wir können unsere Schreibweisen für primitiv rekursive bzw. μ -rekursive Funktionen als *Programmiersprachen* auffassen.
- Dann liefern die Beweise *Übersetzungs-Algorithmen* ('Compiler') zwischen μ -rekursiven Programmen und *while*-Programmen bzw. zwischen primitiv rekursiven Programmen und *loop*-Programmen.

Berechenbarkeit

Satz 3.41 *Zu jedem primitiv rekursiven Programm lässt sich ein äquivalentes loop-Programm konstruieren, also ist jede primitiv rekursive Funktion loop-berechenbar.*

Beweis:

Es genügt zu zeigen

1. wie sich die Grundfunktionen als *loop*-Programme implementieren lassen,
2. wie sich aus *loop*-Programmen für g, h_1, \dots, h_l ein *loop*-Programm für $Sub(g; h_1, \dots, h_l)$ konstruieren lässt,
3. wie sich aus *loop*-Programmen für g und h ein *loop*-Programm für $Prim(g, h)$ konstruieren lässt.

1. s. Übungsblatt 14, Aufgabe 2.
-

Berechenbarkeit

2. Seien P, P_1, \dots, P_l *loop*-Programme (oder *while*-Programme), die die Funktionen $g : \mathbb{N}^l \hookrightarrow \mathbb{N}$ und $h_1, \dots, h_l : \mathbb{N}^k \hookrightarrow \mathbb{N}$ berechnen.

Wir dürfen annehmen, dass die Programme P und P_i ($i = 1, \dots, l$) ‘geeignete’ Form haben, etwa:

$$P_i = \text{read } X_1, \dots, X_k; \text{ stl}_i \text{ write } X_{k+i}$$

(d.h. alle P_i arbeiten mit den gleichen Eingabe-Speicherplätzen und jedes hat seinen eigenen Ausgabe-Speicherplatz)

$$P = \text{read } X_{k+1}, \dots, X_{k+l}; \text{ stl write } X_0$$

(d.h. P benutzt die Ausgabe-Speicherplätze der P_i zur Eingabe) und dass sie alle ‘sauber’ programmiert sind, d.h. dass die Inhalte von X_1, \dots, X_k nicht verändert werden und dass alle anderen Speicherplätze explizit vorbelegt werden.

Berechenbarkeit

Dann lässt sich die Funktion $Sub(g; h_1, \dots, h_l) : \mathbb{N}^k \hookrightarrow \mathbb{N}$ berechnen durch

`read X_1, \dots, X_k ; $stl_1 \dots stl_l$ stl write X_0`

3. Seien P_0, P_1 Programme, die die Funktionen $g : \mathbb{N}^k \hookrightarrow \mathbb{N}$ und $h : \mathbb{N}^{k+2} \hookrightarrow \mathbb{N}$ berechnen. Wir dürfen annehmen, dass

$P_0 = \text{read } X_1, \dots, X_k; stl_0 \text{ write } X_{k+2}$

$P_1 = \text{read } X_1, \dots, X_{k+2}; stl_1 \text{ write } X_{k+2}$

und dass beide 'sauber' programmiert sind. Dann lässt sich die Funktion $Prim(g, h) : \mathbb{N}^{k+1} \hookrightarrow \mathbb{N}$ berechnen durch

`read X_1, \dots, X_k, X_0 ;`

`$X_{k+1} := 0$; stl_0`

`loop X_0 do stl_1 $X_{k+1} := succ(X_{k+1})$ od`

`write X_{k+2}`

Berechenbarkeit

Erläuterung:

Das angegebene Programm berechnet nacheinander die Werte

$$Prim(g, h)(\bar{n}, 0), \dots, Prim(g, h)(\bar{n}, m)$$

wobei X_{k+2} als Speicherplatz für das Zwischenergebnis benutzt wird. X_{k+1} dient als Zähler, der die Werte $0, \dots, m$ durchläuft (wobei der letzte Wert m nicht mehr benötigt wird).

Durch stl_0 wird X_{k+2} zunächst mit $g(\bar{n}) = Prim(g, h)(\bar{n}, 0)$ vorbelegt. Dann wird der Inhalt von X_{k+2} in jedem Schleifendurchlauf durch stl_1 verändert: Wenn zu Beginn eines Schleifendurchlaufs der Wert $Prim(g, h)(\bar{n}, i)$ in X_{k+2} steht, dann bewirkt stl_1 , dass am Ende des Schleifendurchlaufs der Wert $h(\bar{n}, m, Prim(g, h)(\bar{n}, i)) = Prim(g, h)(\bar{n}, i + 1)$ dort steht.

Da in X_0 das $(k + 1)$ -te Argument m eingelesen wird, finden m Schleifendurchläufe statt, also enthält X_{k+2} am Ende des Programms das gewünschte Ergebnis $Prim(g, h)(\bar{n}, m)$. \square

Berechenbarkeit

Satz 3.42 *Zu jedem μ -rekursiven Programm lässt sich ein äquivalentes while-Programm konstruieren, also ist jede μ -rekursive Funktion while-berechenbar.*

Beweis: Es bleibt nur noch zu zeigen, wie sich Anwendungen des μ -Operators in die *while*-Programmiersprache übersetzen lassen. Sei also

$$P = \text{read } X_1, \dots, X_k, X_{k+1}; \text{ stl write } X_0$$

ein 'sauberes' *while*-Programm, das die Funktion $g : \mathbb{N}^{k+1} \hookrightarrow \mathbb{N}$ berechnet. Dann lässt sich $\mu(g) : \mathbb{N}^k \hookrightarrow \mathbb{N}$ berechnen durch

`read` X_1, \dots, X_k ;

$X_{k+1} := 0$; $X_0 := 1$;

`while` X_0 `do stl` $X_{k+1} := \text{succ}(X_{k+1})$ `od`;

$X_{k+1} := \text{pred}(X_{k+1})$;

`write` X_{k+1}

Berechenbarkeit

Erläuterung:

Das angegebene Programm berechnet nacheinander die Werte

$$g(\bar{n}, 0), g(\bar{n}, 1), \dots$$

bis es die erste Zahl m mit $g(\bar{n}, m) = 0$ findet.

X_{k+1} dient als Zähler für die Anzahl der Schleifendurchläufe und wird deshalb mit 0 vorbelegt. Im i -ten Schleifendurchlauf wird durch Ausführung von stl der Wert $g(\bar{n}, i)$ berechnet und in X_0 gespeichert. Anschließend wird X_k auf $i + 1$ gesetzt.

Also terminiert die Schleife, sobald die erste Zahl m mit $g(\bar{n}, m) = 0$ gefunden ist, und dann enthält X_{k+1} den Wert $m + 1$. Deshalb muss X_{k+1} noch um 1 heruntersgesetzt werden.

Man beachte noch, dass das Programm *nicht* terminiert, wenn *kein* Wert m mit $g(\bar{n}, m) = 0$ und $(\bar{n}, i) \in \text{Def}(f)$ für alle $i < m$ existiert. Auch das entspricht der Definition des μ -Operators. □

Berechenbarkeit

Satz 3.43 *Zu jedem loop-Programm lässt sich ein äquivalentes primitiv rekursives Programm konstruieren, also ist jede loop-berechenbare Funktion primitiv rekursiv.*

Beweis: Sei $m \in \mathbb{N}$. Für jede Anweisungsliste stl , die höchstens die Speicherplätze X_0, \dots, X_m enthält, definieren wir die *Komponentenfunktionen*

$$\begin{aligned} \llbracket stl \rrbracket_i &: \mathbb{N}^{m+1} \hookrightarrow \mathbb{N} \\ (n_0, \dots, n_m) &\mapsto \begin{cases} \sigma(X_i) & \text{falls } (stl, \sigma_0[n_0/X_0] \dots [n_m/X_m]) \vdash^* (\varepsilon, \sigma) \\ \text{undefiniert} & \text{sonst} \end{cases} \end{aligned}$$

für $i = 0, \dots, m$, und die *Gesamtfunktion*

$$\begin{aligned} \llbracket stl \rrbracket &: \mathbb{N}^{m+1} \hookrightarrow \mathbb{N}^{m+1} \\ (n_0, \dots, n_m) &\mapsto (\llbracket stl \rrbracket_0(n_0, \dots, n_m), \dots, \llbracket stl \rrbracket_m(n_0, \dots, n_m)) \end{aligned}$$

Berechenbarkeit

Die i -te Komponentenfunktion gibt also an, wie sich der Endinhalt des Speicherplatzes X_i aus den Anfangsinhalten der Speicherplätze X_0, \dots, X_m ergibt. Die Gesamtfunktion fasst all diese Informationen zusammen.

Wenn wir zeigen können, dass die Funktionen $\llbracket stl \rrbracket_i$ stets primitiv rekursiv sind (was gleichbedeutend ist zur primitiven Rekursivität von $\llbracket stl \rrbracket$), so ist auch die von einem Programm

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

berechnete Funktion $\llbracket P \rrbracket : \mathbb{N}^k \hookrightarrow \mathbb{N}$ primitiv rekursiv, denn es gilt

$$\llbracket P \rrbracket = \text{Sub}(\llbracket stl \rrbracket_i; h_0, \dots, h_m) \quad \text{mit} \quad h_l = \begin{cases} \text{proj}_j^k & \text{falls } l = i_j \\ \text{const}_0^k & \text{falls } l \notin \{i_1, \dots, i_k\} \end{cases}$$

Die primitive Rekursivität der Funktionen $\llbracket stl \rrbracket_i$ (bzw. $\llbracket stl \rrbracket$) beweist man durch Induktion über die Größe von stl .

Berechenbarkeit

- Den Induktionsanfang bilden die Zuweisungen, z.B. gilt

$$\llbracket X_j := \text{pred}(X_k); \rrbracket_i = \left\{ \begin{array}{ll} \text{pred} \circ \text{proj}_k^{m+1} & \text{falls } i = j \\ \text{proj}_i^{m+1} & \text{sonst} \end{array} \right\} \in \mathcal{PR}$$

weil $\text{pred} \in \mathcal{PR}$. Analog für die übrigen Zuweisungen.

- Im Induktionsschritt sind die übrigen Anweisungslisten zu betrachten. Es gilt

$$\begin{aligned} \llbracket stl \ stl' \rrbracket &= \llbracket stl' \rrbracket \circ \llbracket stl \rrbracket \\ \llbracket \text{if } X_j \text{ then } stl \text{ else } stl' \text{ fi} \rrbracket &= \text{If}(\text{proj}_j^{m+1}; \llbracket stl \rrbracket, \llbracket stl' \rrbracket) \\ \llbracket \text{loop } X_j \text{ do } stl \text{ od} \rrbracket &= \text{Iter}(\text{proj}_j^{m+1}; \llbracket stl \rrbracket) \end{aligned}$$

Diese Funktionen sind primitiv rekursiv, weil nach Induktionsannahme die Funktionen $\llbracket stl \rrbracket$ und $\llbracket stl' \rrbracket$ primitiv rekursiv sind. \square

Berechenbarkeit

Satz 3.44 *Zu jedem while-Programm lässt sich ein äquivalentes μ -rekursives Programm konstruieren, also ist jede while-berechenbare Funktion μ -rekursiv.*

Beweis: Es ist nur noch zu zeigen, dass sich jede *while*-Anweisung

while X_i *do stl od*

in die Schreibweise für μ -rekursive Funktionen übersetzen lässt.

Die Idee besteht darin, mit Hilfe des μ -Operators die kleinste Anzahl k von Schleifendurchläufen zu bestimmen, bei der ein Zustand σ mit $\sigma(X_i) = 0$ entsteht, und dann die Schleife k -mal auszuführen.

Genauer: Für jedes Tupel $(n_0, \dots, n_m) \in \mathbb{N}^{m+1}$ betrachten wir die Zustände σ_j mit

$$(\underbrace{stl \dots stl}_j, \sigma_0[n_0/X_0] \dots [n_m/X_m]) \vdash^* (\varepsilon, \sigma_j)$$

und suchen die Zahl $k = \min \{j \in \mathbb{N} \mid \sigma_j(X_i) = 0\}$.

Berechenbarkeit

Diese Zahl k erhalten wir als Ergebnis einer μ -rekursiven Funktion:

$$\begin{aligned}k &= \min \{j \in \mathbb{N} \mid \text{proj}_i^{m+1}(\llbracket stl \rrbracket^j(n_0, \dots, n_m)) = 0\} \\ &= \min \{j \in \mathbb{N} \mid \text{proj}_i^{m+1}(\text{Iter } \llbracket stl \rrbracket (n_0, \dots, n_m, j)) = 0\} \\ &= \underbrace{\mu(\text{proj}_i^{m+1} \circ \text{Iter } \llbracket stl \rrbracket)}_f(n_0, \dots, n_m)\end{aligned}$$

Also ergeben sich auch die Inhalte der Speicherplätze im Zustand σ_k durch eine μ -rekursive Funktion:

$$\begin{aligned}(\sigma_k(X_0), \dots, \sigma_k(X_m)) &= \llbracket stl \rrbracket^k(n_0, \dots, n_m) \\ &= \text{Iter } \llbracket stl \rrbracket (n_0, \dots, n_m, k) \\ &= \text{Iter } \llbracket stl \rrbracket (n_0, \dots, n_m, f(n_0, \dots, n_m)) \\ &= \text{Sub}(\text{Iter } \llbracket stl \rrbracket; \text{proj}_1^{m+1}, \dots, \text{proj}_{m+1}^{m+1}, f)(n_0, \dots, n_m)\end{aligned}$$

Berechenbarkeit

Andererseits ist σ_k natürlich der Endzustand der *while*-Anweisung:

$$(\text{while } X_i \text{ do } stl \text{ od}, \sigma_0[n_0/X_0] \dots [n_m/X_m]) \vdash^* (\varepsilon, \sigma_k)$$

d.h. die obige μ -rekursive Funktion ist die gesuchte “Übersetzung”:

$$\llbracket \text{while } X_i \text{ do } stl \text{ od} \rrbracket = \text{Sub}(\text{Iter} \llbracket stl \rrbracket; \text{proj}_1^{m+1}, \dots, \text{proj}_{m+1}^{m+1}, f)$$

Man beachte noch, dass diese Überlegungen auch dann korrekt sind, wenn die *while*-Anweisung *nicht* terminiert. Dann existiert nämlich der oben genannte Zustand σ_k nicht, also liefert auch die Anwendung der Funktion $f = \mu(\text{proj}_i^{m+1} \circ \text{Iter} \llbracket stl \rrbracket)$ auf die Anfangswerte (n_0, \dots, n_m) kein Ergebnis. \square

Damit sind alle gewünschten Äquivalenzen zwischen unseren Programmiersprachen bewiesen. Man kann sich nun noch fragen, wie mächtig die Sprache der primitiv rekursiven Funktionen (bzw. die dazu äquivalente Sprache der *loop*-Programme) ist. Eine gewisse Antwort gibt der folgende Satz.

Berechenbarkeit

Satz 3.45 *In der Programmiersprache der primitiv rekursiven Programme lässt sich kein Interpreter für die Sprache selbst schreiben.*

Beweis: (durch Diagonalisierung)

Wie bei jeder Programmiersprache können wir die Programme “durchnummerieren”, d.h. wir haben eine unendliche Folge π_1, π_2, \dots , die *alle* primitiv rekursiven Programme enthält. Die vom Programm π_i berechnete (totale) Funktion bezeichnen wir mit f_i .

Von einem Interpreter π erwarten wir, dass er bei Eingabe (i, n) das Programm π_i für den Eingabewert n simuliert, also das Ergebnis $f_i(n)$ liefert. Angenommen, π berechnet selbst eine primitiv rekursive Funktion f . Dann ist auch $g : \mathbb{N} \rightarrow \mathbb{N}, i \mapsto f(i, i) + 1$ primitiv rekursiv, also müsste g mit einer der Funktionen f_i übereinstimmen. Es gilt aber

$$g(i) = f(i, i) + 1 = f_i(i) + 1 \neq f_i(i)$$

für jedes $i \in \mathbb{N}$. Das ist ein Widerspruch. □

Berechenbarkeit

Als unmittelbare Folgerung von Satz 3.45 erhalten wir

Korollar 3.46 *Es gibt eine totale berechenbare Funktion, die nicht primitiv rekursiv ist.*

Beweis:

Ein Beispiel ist die Funktion f im Beweis von Satz 3.45.

Sie ist total, da sie für jedes Paar (i, n) das Ergebnis $f_i(n)$ liefert. Sie ist berechenbar, weil sich ein Interpreter für primitiv rekursive Programme z.B. in der *while*-Programmiersprache schreiben lässt. \square

Ein anderes Beispiel ist die bereits erwähnte Ackermann-Funktion (vgl. Literatur).