
Grundlagen der theoretischen Informatik

Kurt Sieber

Fachbereich Mathematik/Theoretische Informatik
Universität Siegen

Vorlesung vom 18.01.2005 (Stand: 24.01.2005)

Berechenbarkeit

Damit haben wir jetzt alle Bausteine, die wir für das Programmstück $Step(X_i, X_j, X_k)$ (zur Simulation von Übergangsschritten) benötigen. $Step(X_i, X_j, X_k)$ arbeitet wie folgt:

- Aus X_i erhält es eine Zahl m , aus X_j die Codierung $[st; st/]$ einer nichtleeren Anweisungsliste und aus X_k die Codierung $\langle \sigma \rangle_m$ des aktuellen Zustands.
- $[st; st/]$ wird aufgespalten in $[st]$ und $[st/]$. $[st]$ ist wieder Codierung einer Zahlenliste. Aus der ersten Zahl i dieser Liste kann man ablesen, um welche Art von Anweisung es sich handelt.
- Im Falle $0 \leq i \leq 3$ ist st eine Zuweisung. Dann liest man aus den restlichen Komponenten ab, wie die linke und rechte Seite der Zuweisung aussehen und errechnet aus $\langle \sigma \rangle_m$ die Codierung $\langle \sigma' \rangle_m$ des neuen Zustands σ' . Schließlich speichert man die Codierung $[st/]$ der noch auszuführenden Anweisungsliste in X_j ab und die Codierung $\langle \sigma' \rangle_m$ des neuen Zustands in X_k .

Berechenbarkeit

- Im Falle $i = 4$ ist st von der Form **if** X_n **then** stl_1 **else** stl_2 **fi**. Dann liest man aus den restlichen Komponenten den Index n und die Codierungen $\lceil stl_1 \rceil$ und $\lceil stl_2 \rceil$ des **then**- und **else**-Teils ab. Aus $\langle \sigma \rangle_m$ und n errechnet man $\sigma(X_n)$. Ist $\sigma(X_n) = 0$, so errechnet man die Zahl $\lceil stl_1 stl \rceil$ aus $\lceil stl_1 \rceil$ und $\lceil stl \rceil$ und speichert sie in X_j ab. Andernfalls speichert man $\lceil stl_2 stl \rceil$ in X_j ab. Den Inhalt von X_k ($= \langle \sigma \rangle_m$) lässt unverändert.
- Im Falle $i = 5$ ist st von der Form **while** X_n **do** stl_1 **od**. Auch hier berechnet man wieder $\sigma(X_n)$ und speichert je nach Ergebnis entweder $\lceil stl_1 stl \rceil$ oder $\lceil stl \rceil$ in X_j ab. Den Inhalt von X_k lässt man wieder unverändert.

Berechenbarkeit

Mit dem Programmstück $Step(X_i, X_j, X_k)$ erhält man dann leicht den eigentlichen Interpreter I^k (für k -stellige Funktionen), der wie folgt arbeitet:

Als Eingabe erhält er die Codierung (= Gödelnummer)

$$[P] = \langle [stl], m, i, i_1, \dots, i_k \rangle$$

eines Programms P und die Eingabewerte n_1, \dots, n_k für P . Aus m, i_1, \dots, i_k und n_1, \dots, n_k errechnet er die Codierung $\langle \sigma \rangle$ des Anfangszustands $\sigma = \sigma_0[n_1/X_{i_0}] \dots [n_k/X_{i_k}]$. Dann simuliert er den Ablauf der Anweisungsliste stl für diesen Anfangszustand σ mit Hilfe von $Step(X_i, X_j, X_k)$. Wenn stl abgearbeitet ist (d.h. wenn nur noch die leere Anweisungsliste übrig ist) berechnet er den Inhalt von X_i aus der Codierung des Endzustands und gibt ihn aus.

Berechenbarkeit

Wozu brauchen wir den Interpreter?

In der Berechenbarkeitstheorie haben wir an einigen Stellen die Existenz eines Interpreters vorausgesetzt, z.B.

- beim Beweis der Semi-Entscheidbarkeit des Halteproblems,
- beim Beweis, dass die Menge H_{\forall} *nicht* semi-entscheidbar ist.

Aber auch an anderen Stellen haben wir Annahmen über die Mächtigkeit unserer Programmiersprache gemacht, die sich am besten mit einem Interpreter nachweisen lassen, z.B.

- die ‘Parallelausführung’ zweier Programme auf dem gleichen Eingabewert,
 - die ‘Parallelausführung’ eines Programms auf unendlich vielen Eingabewerten mit der *dovetailing*-Technik.
-

Berechenbarkeit

Fazit:

Wir haben gezeigt, dass selbst eine so einfache Programmiersprache wie die *while*-Programme 'hinreichend mächtig' im Sinne der Berechenbarkeitstheorie ist.

Damit ist die Berechenbarkeitstheorie erst recht auf alle realistischen (imperativen) Programmiersprachen anwendbar, denn die enthalten ja *while*-Programme als Teilmenge. Insbesondere wissen wir, dass in all diesen Sprachen das Halteproblem unentscheidbar ist (und nach dem Satz von Rice sogar jedes nichttriviale semantische Problem).

Berechenbarkeit

Bevor wir alternative Ansätze zur Definition von Berechenbarkeit untersuchen, betrachten wir eine *schwächere* Programmiersprache, nämlich die sogenannten *loop-Programme*. Die *loop*-Programmiersprache erhält man aus der *while*-Programmiersprache, indem man die Produktion

$$st ::= \text{while } X_i \text{ do } stl \text{ od}$$

ersetzt durch

$$st ::= \text{loop } X_i \text{ do } stl \text{ od}$$

Der Übergangsschritt für diese *loop*-Anweisungen ist definiert durch

$$(\text{loop } X_i \text{ do } stl \text{ od}, \sigma) \vdash (\underbrace{stl \dots stl}_m, \sigma) \text{ falls } \sigma(X_i) = m$$

Man beachte, dass sich die Zahl m aus dem Anfangsinhalt von X_i ergibt, d.h. auch wenn X_i in stl vorkommt, wird stl nur m -mal ausgeführt, also kann insbesondere keine Endlosschleife entstehen.

Berechenbarkeit

Die von einem *loop*-Programm P berechnete Funktion $\llbracket P \rrbracket$ ist genau wie bei *while*-Programmen definiert. Eine Funktion f heißt *loop-berechenbar*, wenn es ein *loop*-Programm P gibt mit $\llbracket P \rrbracket = f$.

Satz 3.26 *Jede loop-berechenbare Funktion ist while-berechenbar.*

Beweis: Die Anweisung `loop X_i do stl od` lässt sich in der *while*-Programmiersprache simulieren durch

$$Y_0 := X_i; \text{ while } Y_0 \text{ do } Y_0 := \text{pred}(Y_0); \text{ stl } \text{od}$$

wobei Y_0 ein Speicherplatz ist, der sonst nirgends vorkommt. □

Fast alle Funktionen, die wir mit *while*-Programmen berechnet haben, sind auch *loop*-berechenbar, denn immer wenn sich die Schrittzahl einer *while*-Schleife von vornherein nach oben abschätzen lässt, kann man auch eine *loop*-Schleife verwenden.

Berechenbarkeit

Die Umkehrung von Satz 3.26 gilt allerdings nicht, d.h. die *loop*-Programmiersprache ist *nicht* Turing-mächtig. Es gilt nämlich

Satz 3.27 *Jede loop-berechenbare Funktion ist total.*

Beweis: Es genügt zu zeigen, dass eine Anweisungsliste *stl* immer terminiert, d.h. dass für *jeden* Zustand σ ein Zustand σ' existiert mit $(stl, \sigma) \vdash^* (\varepsilon, \sigma')$. Das folgt leicht durch Induktion über die Länge von *stl*. Insbesondere nutzt man beim Induktionsschritt für *loop* X_i *do stl od* aus, dass $\underbrace{stl \dots stl}_m$ für alle Zustände terminiert, wenn *stl* für alle Zustände terminiert. \square

Im übrigen gibt es sogar totale *while*-berechenbare Funktionen, die nicht *loop*-berechenbar sind. Ein konkretes Beispiel ist die sogenannte *Ackermann*-Funktion (vgl. Literatur).

Berechenbarkeit

Wir betrachten nun den zweiten Ansatz zur Definition von Berechenbarkeit, die sogenannten μ -rekursiven Funktionen. Bei diesem Ansatz kommt man im Prinzip *ohne* Programmiersprache aus. Andererseits wird eine präzise mathematische Schreibweise eingeführt, die man durchaus als Programmiersprache auffassen kann.

Man nennt eine Funktion *primitiv rekursiv*, wenn sie sich aus gewissen *Grundfunktionen* durch (wiederholte) Anwendung der Operatoren

1. *Substitution* und
2. *primitive Rekursion*

aufbauen lässt.

Man nennt sie *μ -rekursiv*, wenn neben 1. und 2. zusätzlich noch der *μ -Operator* zugelassen ist.

Diese neuen Begriffe werden wir jetzt nach und nach präzise definieren.

Berechenbarkeit

Zu den *Grundfunktionen* (oder: *Basisfunktionen*) gehören

1. für jedes $k, c \in \mathbb{N}$ die *konstante Funktion*

$$\text{const}_c^k : \mathbb{N}^k \rightarrow \mathbb{N}$$

$$(n_1, \dots, n_k) \mapsto c$$

2. für jedes $i, k \in \mathbb{N}$ mit $1 \leq i \leq k$ die *Projektion*

$$\text{proj}_i^k : \mathbb{N}^k \rightarrow \mathbb{N}$$

$$(n_1, \dots, n_k) \mapsto n_i$$

3. die *Nachfolgerfunktion* $\text{succ} : \mathbb{N} \rightarrow \mathbb{N}$

$$n \mapsto n + 1$$

Man beachte, dass bei den konstanten Funktionen die Stelligkeit 0 zugelassen ist und bei den Projektionen die Stelligkeit 1: const_c^0 ist nichts anderes als die Zahl c , und proj_1^1 ist die Identität auf \mathbb{N} .

Berechenbarkeit

Substitution ist eine Verallgemeinerung der Komposition von Funktionen.

Definition 3.28 Sei $k \in \mathbb{N}$. Eine Funktion $f : \mathbb{N}^k \hookrightarrow \mathbb{N}$ entsteht durch **Substitution** (oder **Einsetzung**) aus den Funktionen $g : \mathbb{N}^l \hookrightarrow \mathbb{N}$ und $h_1, \dots, h_l : \mathbb{N}^k \hookrightarrow \mathbb{N}$, wenn für alle $n_1, \dots, n_k \in \mathbb{N}$ gilt:

$$f(n_1, \dots, n_k) = g(h_1(n_1, \dots, n_k), \dots, h_l(n_1, \dots, n_k)) \quad (*)$$

Die Funktion f bezeichnet man dann mit **Sub**($g; h_1, \dots, h_l$).

Gleichung (*) ist übrigens so zu verstehen, dass $f(n_1, \dots, n_k)$ undefiniert ist, sobald einer der Werte $h_i(n_1, \dots, n_k)$ undefiniert ist, und zwar selbst dann, wenn die Funktion g nicht alle Argumente benötigt, z.B. wenn g eine konstante Funktion oder eine Projektion ist.

Berechenbarkeit

Im Fall $l = 1$ vereinfacht sich (*) zu

$$f(n_1, \dots, n_k) = g(h(n_1, \dots, n_k))$$

es gilt also $Sub(g; h) = g \circ h$.

Auch im Fall $l > 1$ kann man $Sub(g; h_1, \dots, h_l)$ als Komposition von g mit einer geeigneten Funktion h auffassen, nämlich

$$h : \mathbb{N}^k \hookrightarrow \mathbb{N}^l$$

$$(n_1, \dots, n_k) \mapsto (h_1(n_1, \dots, n_k), \dots, h_l(n_1, \dots, n_k))$$

Diese Funktion h bezeichnet man manchmal mit $\langle h_1, \dots, h_l \rangle$. Dann gilt also $Sub(g; h_1, \dots, h_l) = g \circ \langle h_1, \dots, h_l \rangle$.

Warnung:

Die gleiche Schreibweise haben wir für die Codierung von Zahlenlisten benutzt. Um eine Verwechslung auszuschließen, sollte man also stets wissen, ob man Zahlen oder Funktionen vor sich hat.

Berechenbarkeit

Beispiele:

1. Sei $mul : \mathbb{N}^2 \rightarrow \mathbb{N}$, $(m, n) \mapsto m * n$ die Multiplikation und $square : \mathbb{N} \rightarrow \mathbb{N}$, $n \mapsto n^2$ die Quadratfunktion.

Dann gilt für alle $n \in \mathbb{N}$

$$square(n) = mul(n, n) = mul(proj_1^1(n), proj_1^1(n))$$

also ist $square = Sub(mul; proj_1^1, proj_1^1)$.

2. Sei $add : \mathbb{N}^2 \rightarrow \mathbb{N}$, $(m, n) \mapsto m + n$ die Addition und $sos : \mathbb{N}^2 \rightarrow \mathbb{N}$, $(m, n) \mapsto m^2 + n^2$ ('sum of squares')

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

$$\begin{aligned} sos(m, n) &= add(square(m), square(n)) \\ &= add(square(proj_1^2(m, n)), square(proj_2^2(m, n))) \end{aligned}$$

also ist $sos = Sub(add; square \circ proj_1^2, square \circ proj_2^2)$

Berechenbarkeit

Primitive Rekursion ist ein 'Schema' zur induktiven Definition von Funktionen.

Definition 3.29 Sei $k \in \mathbb{N}$. Eine Funktion $f : \mathbb{N}^{k+1} \hookrightarrow \mathbb{N}$ entsteht durch **primitive Rekursion** aus den Funktionen $g : \mathbb{N}^k \hookrightarrow \mathbb{N}$ und $h : \mathbb{N}^{k+2} \hookrightarrow \mathbb{N}$, wenn für alle $n_1, \dots, n_k, m \in \mathbb{N}$ gilt:

$$\begin{aligned}f(n_1, \dots, n_k, 0) &= g(n_1, \dots, n_k) \\f(n_1, \dots, n_k, m + 1) &= h(n_1, \dots, n_k, m, f(n_1, \dots, n_k, m))\end{aligned}$$

Die Funktion f bezeichnet man dann mit **Prim**(g, h).

Man beachte, dass die Induktion nur über den letzten Parameter m von f laufen darf. Man bezeichnet m als **Rekursionsparameter** und n_1, \dots, n_k als **Nebenparameter**. Die Funktion g liefert den Wert für den Induktionsanfang (in Abhängigkeit von den Nebenparametern), und die Funktion h beschreibt den Induktionsschritt.

Berechenbarkeit

Im Spezialfall $k = 0$ ist g eine 0-stellige Funktion, d.h. $g \in \mathbb{N}$. Dann vereinfacht sich das Schema der primitiven Rekursion zu:

$$\begin{aligned}f(0) &= g \\f(m+1) &= h(m, f(m))\end{aligned}$$

Beispiele:

1. Sei $pred : \mathbb{N} \rightarrow \mathbb{N}$, $m \mapsto m - 1$ die Vorgängerfunktion.

Dann ist

$$pred(0) = 0 = const_0^0$$

und für alle $m \in \mathbb{N}$ gilt

$$pred(m+1) = m = proj_1^2(m, pred(m))$$

Also ist $pred = Prim(const_0^0; proj_1^2)$.

Berechenbarkeit

2. Für die Addition $add : \mathbb{N}^2 \rightarrow \mathbb{N}$ gilt

$$add(n, 0) = n = proj_1^1(n)$$

und

$$\begin{aligned} add(n, m + 1) &= n + (m + 1) \\ &= (n + m) + 1 \\ &= succ(add(n, m)) \\ &= succ(proj_3^3(n, m, add(n, m))) \\ &= (succ \circ proj_3^3)(n, m, add(n, m)) \end{aligned}$$

Also ist

$$\begin{aligned} add &= Prim(proj_1^1; succ \circ proj_3^3) \\ &= Prim(proj_1^1; Sub(succ; proj_3^3)) \end{aligned}$$

Berechenbarkeit

Definition 3.30 Die Menge \mathcal{PR} der **primitiv rekursiven Funktionen** ist die kleinste Menge von Funktionen $f : \mathbb{N}^k \hookrightarrow \mathbb{N}$ (für alle $k \in \mathbb{N}$), die

- alle Grundfunktionen enthält und
- abgeschlossen ist unter Substitution und primitiver Rekursion.

Mit anderen Worten:

Eine Funktion $f : \mathbb{N}^k \hookrightarrow \mathbb{N}$ ist genau dann primitiv rekursiv, wenn sie durch (wiederholte) Anwendung der Operatoren *Sub* und *Prim* aus den Grundfunktionen $const_c^k$, $proj_i^k$ und *succ* entsteht.

Beispiele:

1. *pred* ist primitiv rekursiv wegen $pred = Prim(const_0^0; proj_1^2)$
2. *add* ist primitiv rekursiv wegen $add = Prim(proj_1^1; succ \circ proj_3^3)$

Berechenbarkeit

3. Für die Subtraktion $subtr : \mathbb{N}^2 \rightarrow \mathbb{N}$, $(n, m) \mapsto n \dot{-} m$ gilt

$$n \dot{-} 0 = n = proj_1^1(n)$$

und

$$n \dot{-} (m + 1) = (n \dot{-} m) \dot{-} 1 = pred(proj_3^3(n, m, n \dot{-} m))$$

also ist $subtr = Prim(proj_1^1; pred \circ proj_3^3) \in \mathcal{PR}$ wegen $pred \in \mathcal{PR}$.

4. Für die Multiplikation gilt

$$n * 0 = 0 = const_0^1(n)$$

und

$$\begin{aligned} n * (m + 1) &= (n * m) + n \\ &= add(proj_3^3(n, m, n * m), proj_1^1(n, m, n * m)) \end{aligned}$$

also ist $mul = Prim(const_0^1; Sub(add; proj_3^3, proj_1^1)) \in \mathcal{PR}$ wegen $add \in \mathcal{PR}$.

5. Wegen $add, mul \in \mathcal{PR}$ folgt $square, sos \in \mathcal{PR}$.
