
Grundlagen der theoretischen Informatik

Kurt Sieber

Fachbereich Mathematik/Theoretische Informatik
Universität Siegen

Vorlesung vom 17.01.2005 (Stand: 18.01.2005)

Berechenbarkeit

Die Codierung von Zahlenlisten können wir verwenden, um Speicherzustände, Anweisungen, Anweisungslisten und schließlich ganze Programme zu codieren.

Für jeden Speicherzustand σ und jede Zahl $m \in \mathbb{N}$ definieren wir die *m-stellige Codierung* von σ als die Zahl

$$\langle \sigma \rangle_m = \langle \sigma(X_0), \dots, \sigma(X_m) \rangle$$

Diese Art der Codierung genügt, weil jedes *while*-Programm nur auf endlich vielen Speicherplätzen arbeitet. Wenn nur X_0, \dots, X_m in P vorkommen, dann genügt es, während der Ausführung des Programms P die Werte $\sigma(X_0), \dots, \sigma(X_m)$ bzw. deren Codierung $\langle \sigma \rangle_m$ zu kennen.

Beispiele:

- $\langle \sigma_0 \rangle_3 = \langle 0, 0, 0, 0 \rangle = 2^1 * 3^1 * 5^1 * 7^1 = 210$
- $\langle \sigma_0[1/X_0][2/X_2] \rangle_3 = \langle 1, 0, 2, 0 \rangle = 2^2 * 3^1 * 5^3 * 7^1 = 10500$

Berechenbarkeit

Für alle Anweisungen st und Anweisungslisten stl definieren wir die Codierungen $\lceil st \rceil \in \mathbb{N}$ und $\lceil stl \rceil \in \mathbb{N}$ durch folgende Induktion über die Länge von st bzw. stl :

- $\lceil X_i := 0 \rceil = \langle 0, i \rangle$
- $\lceil X_i := X_j \rceil = \langle 1, i, j \rangle$
- $\lceil X_i := succ(X_j) \rceil = \langle 2, i, j \rangle$
- $\lceil X_i := pred(X_j) \rceil = \langle 3, i, j \rangle$
- $\lceil \text{if } X_i \text{ then } stl_1 \text{ else } stl_2 \text{ fi} \rceil = \langle 4, i, \lceil stl_1 \rceil, \lceil stl_2 \rceil \rangle$
- $\lceil \text{while } X_i \text{ do } stl \text{ od} \rceil = \langle 5, i, \lceil stl \rceil \rangle$
- $\lceil st_1; \dots; st_n \rceil = \langle \lceil st_1 \rceil, \dots, \lceil st_n \rceil \rangle$ für alle $n \in \mathbb{N}$
(d.h. im Falle $n = 0$: $\lceil \varepsilon \rceil = \langle \rangle = \prod_{i=1}^0 p_i^{n_i} = 1$)

Berechenbarkeit

Schließlich definieren wir die Codierung $\lceil P \rceil$ eines Programms P durch

$$\lceil \text{read } X_{i_1}, \dots, X_{i_k}; \text{stl write } X_i \rceil = \langle \lceil \text{stl} \rceil, m, i, i_1, \dots, i_k \rangle$$

wobei $m = \max \{j \in \mathbb{N} \mid X_j \text{ kommt in } P \text{ vor}\}$

Beispiel:

```
 $\lceil$ read  $X_0, X_1$ ;  
while  $X_0$  do  $X_0 := \text{pred}(X_0); X_1 := \text{succ}(X_1)$ ; od;  
write  $X_1$  $\rceil$ 
```

$$= \langle \lceil \text{while } X_0 \text{ do } X_0 := \text{pred}(X_0); X_1 := \text{succ}(X_1); \text{od}; \rceil, 1, 1, 0, 1 \rangle$$

$$= \langle \langle \lceil \text{while } X_0 \text{ do } X_0 := \text{pred}(X_0); X_1 := \text{succ}(X_1); \text{od} \rceil \rangle, 1, 1, 0, 1 \rangle$$

$$= \langle \langle \langle 5, 0, \lceil X_0 := \text{pred}(X_0); X_1 := \text{succ}(X_1); \rceil \rangle \rangle, 1, 1, 0, 1 \rangle$$

$$= \langle \langle \langle 5, 0, \langle \lceil X_0 := \text{pred}(X_0) \rceil, \lceil X_1 := \text{succ}(X_1) \rceil \rangle \rangle \rangle, 1, 1, 0, 1 \rangle$$

$$= \langle \langle \langle 5, 0, \langle \langle 3, 0, 0 \rangle, \langle 2, 1, 1 \rangle \rangle \rangle \rangle, 1, 1, 0, 1 \rangle$$

$$= \langle \langle \langle 5, 0, \langle 240, 1800 \rangle \rangle \rangle, 1, 1, 0, 1 \rangle = \dots$$

Berechenbarkeit

Der Interpreter

Kernstück des Interpreters ist ein Programmstück $Step(X_i, X_j, X_k)$, das die Übergangsschritt-Relation simuliert, d.h.: Wenn

- X_i eine Zahl $m \in \mathbb{N}$ enthält
- X_j die Codierung $\lceil stl \rceil$ einer *nichtleeren* Anweisungsliste stl
- und X_k die m -stellige Codierung $\langle \sigma \rangle_m$ eines Zustands σ

dann soll nach Ausführung von $Step(X_i, X_j, X_k)$ gelten:

- X_i enthält immer noch m
- X_j enthält $\lceil stl' \rceil$
- X_k enthält $\langle \sigma' \rangle_m$

wobei (stl', σ') die Nachfolgekonfiguration von (stl, σ) ist, d.h. die eindeutige Konfiguration mit $(stl, \sigma) \vdash (stl', \sigma')$.

Berechenbarkeit

Wir wollen skizzieren, wie man $Step(X_i, X_j, X_k)$ nach und nach aus kleineren Programmstücken zusammensetzt. Es werden benötigt

- Programmstücke für die Grundrechenarten Addition, Subtraktion, Multiplikation, Potenzieren, . . .
- Programmstücke für Abfragen auf Gleichheit, $<$ -Beziehung, Teilbarkeit, Primzahltest, . . .
- Programmstücke zur Codierung und Decodierung von Listen, z.B.
 - eines, das die i -te Primzahl liefert,
 - eines, das den Exponenten der Primzahl p in der Primfaktorzerlegung einer Zahl n liefert,
- Programmstücke, die auf den Codierungen von Anweisungslisten und Zuständen arbeiten, z.B.
 - eines, das aus $m, \langle \sigma \rangle_m, n$ und i die Zahl $\langle \sigma[n/X_i] \rangle_m$ berechnet,
 - eines, das aus $\lceil stl_1 \rceil$ und $\lceil stl_2 \rceil$ die Zahl $\lceil stl_1 stl_2 \rceil$ berechnet.

Berechenbarkeit

Problem:

while-Programme bieten keinerlei Unterstützung für den modularen Programmaufbau, insbesondere

1. keine Prozeduren (mit vernünftiger Parameterübergabe),
2. keine lokalen Variablen.

2. führt dazu, dass stets die ganze Information, die man im Programm mitführt, sichtbar ist (kein 'information hiding').

Deshalb muss man sich als Programmierer selbst überlegen, wie man Programme organisiert, d.h. man muss sich eine gewisse 'Programmierdisziplin' ausdenken und diese streng befolgen.

Berechenbarkeit

Beispiel:

Wir hatten bereits ein Programm(stück) zur Addition, nämlich

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

Dieses Programmstück ist nicht gut wiederverwendbar, da es die Inhalte der Eingabe-Speicherplätze X_0 und X_1 verändert.

Eine bessere Lösung ist das Programmstück

```
 $Y_0 := X_i$ ;  $X_k := X_j$ ;
```

```
while  $Y_0$  do  $Y_0 := pred(Y_0)$ ;  $X_k := succ(X_k)$ ; od;
```

das wir mit $Add(X_i, X_j, X_k)$ bezeichnen. Dabei seien X_i, X_j, X_k, Y_0 *verschiedene* (aber ansonsten beliebige) Speicherplätze.

$Add(X_i, X_j, X_k)$ benutzt die Speicherplätze X_i und X_j zur Eingabe (und lässt sie unverändert) und den Speicherplatz X_k zur Ausgabe. Y_0 dient als 'Hilfs-Speicherplatz'.

Berechenbarkeit

Da Y_0 im Gesamtprogramm sichtbar ist, muss man darauf achten, dass der Inhalt von Y_0 nicht für andere Programmstücke von Bedeutung ist (d.h. man kann Y_0 durchaus in anderen Teilen des Programms wiederverwenden, aber man muss eben damit rechnen, dass sein Inhalt durch $Add(X_i, X_j, X_k)$ verändert wird).

Deshalb merken wir uns die folgende 'Spezifikation':

- $Add(X_i, X_j, X_k)$ wirkt wie eine Zuweisung $X_k := X_i + X_j$ (die es in der *while*-Programmiersprache so nicht gibt).
- Neben X_i, X_j, X_k benutzt es nur den Speicherplatz Y_0 .

Unter den Hilfs-Speicherplätzen Y_i kann man sich Speicherplätze in einem bestimmten 'Speicherbereich' vorstellen, den man sonst nicht benötigt, z.B. $Y_i = X_{i+100}$.

Berechenbarkeit

Natürlich lässt sich die Semantik des Programmstücks $Add(X_i, X_j, X_k)$ auch formal beschreiben:

Für alle $\sigma \in Store$ gilt

$$(Add(X_i, X_j, X_k), \sigma) \vdash^* (\varepsilon, \sigma[\sigma(X_i) + \sigma(X_j)/X_k][0/Y_0]) \quad (ADD)$$

Um (ADD) zu beweisen, formuliert man zunächst eine geeignete Aussage über die *while*-Schleife, nämlich:

Für alle $\sigma \in Store$ gilt

$$\begin{aligned} &(\text{while } Y_0 \text{ do } Y_0 := \text{pred}(Y_0); X_k := \text{succ}(X_k); \text{od};, \sigma) \\ &\vdash^* (\varepsilon, \sigma[0/Y_0][\sigma(Y_0) + \sigma(X_k)/X_k]) \end{aligned} \quad (*)$$

(*) ergibt sich—analog zum alten Additionsprogramm—durch Induktion über $\sigma(Y_0)$. Aus (*) erhält man dann (ADD), indem man noch die Semantik der Zuweisungen $Y_0 := X_i; X_k := X_j$ berücksichtigt.

Berechenbarkeit

Alternative:

Da der Endinhalt des Hilfs-Speicherplatzes Y_0 nicht interessiert, kann man (ADD) durch eine schwächere Aussage ersetzen, etwa:

Für jedes $\sigma \in Store$ existiert ein $\sigma' \in Store$ mit

$$(Add(X_i, X_j, X_k), \sigma) \vdash^* (\varepsilon, \sigma'[\sigma(X_i) + \sigma(X_j)/X_k]) \quad (ADD')$$

und σ' unterscheidet sich von σ nur in Y_0 .

Entsprechend schwächt man die Induktionsbehauptung (*) ab. Der Induktionsbeweis geht dann immer noch durch, und die schwächere Aussage (ADD') genügt, um Aussagen über Programme zu beweisen, die $Add(X_i, X_j, X_k)$ verwenden.

Berechenbarkeit

Weitere nützliche Programmstücke:

Für die *Subtraktion* benutzen wir das Programmstück

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

das wir mit $Sub(X_i, X_j, X_k)$ bezeichnen. Es wirkt wie eine Zuweisung $X_k := X_i \dot{-} X_j$ und verändert ansonsten nur den Inhalt von Y_0 .

Formale Semantik:

Für alle $\sigma \in Store$ gilt

$$(Sub(X_i, X_j, X_k), \sigma) \vdash^* (\varepsilon, \sigma[\sigma(X_i) \dot{-} \sigma(X_j)/X_k][0/Y_0]) \quad (\text{SUB})$$

Schwächere Spezifikation:

Für jedes $\sigma \in Store$ existiert ein $\sigma' \in Store$ mit

$$(Sub(X_i, X_j, X_k), \sigma) \vdash^* (\varepsilon, \sigma'[\sigma(X_i) \dot{-} \sigma(X_j)/X_k]) \quad (\text{SUB}')$$

und σ' unterscheidet sich von σ nur in Y_0 .

Berechenbarkeit

Ein Programmstück $Mul(X_i, X_j, X_k)$ zur Multiplikation erhält man durch Wiederverwendung von $Add(X_i, X_j, X_k)$. Es sei definiert durch

$$Y_1 := X_j; X_k := 0;$$
$$\text{while } Y_1 \text{ do } Y_1 := \text{pred}(Y_1); Y_2 := X_k; Add(Y_2, X_i, X_k) \text{ od};$$

Hier werden *neue* Hilfs-Speicherplätze Y_1, Y_2 verwendet. Y_0 zu verwenden wäre falsch, da sein Inhalt durch $Add(\dots)$ verändert wird.

Spezifikation:

Für jedes $\sigma \in Store$ existiert ein $\sigma' \in Store$ mit:

$$(Mul(X_i, X_j, X_k), \sigma) \vdash^* (\varepsilon, \sigma'[\sigma(X_i) * \sigma(X_j) / X_k]) \quad (\text{MUL})$$

und σ' unterscheidet sich von σ nur in Y_0, Y_1, Y_2 .

Um (MUL) zu beweisen, formuliert man wieder eine geeignete Induktionsbehauptung für die *while*-Schleife und verwendet dann die Spezifikation (ADD') für das Programmstück $Add(Y_2, X_i, X_k)$.

Berechenbarkeit

Mit der Subtraktion lassen sich *Vergleichsoperatoren* implementieren:

$Less(X_i, X_j, X_k)$ sei das Programmstück

$$Sub(X_j, X_i, X_k)$$

Es testet, ob $\sigma(X_i) < \sigma(X_j)$ ist, und zwar in folgendem Sinne: Wenn ja, so liefert es eine Zahl $n > 0$ (= 'true') im Speicherplatz X_k ab, andernfalls liefert es die Zahl 0 (= 'false') ab.

$Leq(X_i, X_j, X_k)$ sei das Programmstück

$$Y_1 := succ X_j; Less(X_i, Y_1, X_k)$$

Es testet, ob $\sigma(X_i) \leq \sigma(X_j)$ ist.

Schließlich sei $Eq(X_i, X_j, X_k)$ das Programmstück

$$Leq(X_i, X_j, X_k); \text{ if } X_k \text{ then } Leq(X_j, X_i, X_k) \text{ else fi};$$

Es testet, ob $\sigma(X_i) = \sigma(X_j)$ ist.

Berechenbarkeit

Mit Hilfe der Vergleichsoperatoren können wir jetzt *Teilbarkeit* testen. Um zu überprüfen, ob m ein Teiler von n ist, berechnet man die Vielfachen $0 * m, 1 * m, 2 * m, \dots$, die nicht größer als n sind, und überprüft, ob eine dieser Zahlen mit n übereinstimmt.

Auf dieser Idee basiert das Programmstück $Divides(X_i, X_j, X_k)$:

```
Y1 := 0; Y2 := succ(Y1);  
while Y2  
do Eq(Y1, Xj, Xk)  
  if Xk then Y2 := 0; else Leq(Y1, Xj, Y2) fi  
  if Y2 then Y3 := Y1; Add(Y3, Xi, Y1) else fi  
od
```

Es testet, ob $\sigma(X_i)$ ein Teiler von $\sigma(X_j)$ ist (unter der Voraussetzung $\sigma(X_i) \neq 0$).

Berechenbarkeit

Erläuterung der Arbeitsweise von $Divides(X_i, X_j, X_k)$:

Sei $\sigma(X_i) = m \neq 0$ und $\sigma(X_j) = n$. Der Speicherplatz Y_1 enthält stets das aktuelle Vielfache von m , das mit n verglichen wird. Er wird mit 0 vorbelegt und in jedem Schleifendurchlauf um m erhöht.

Im Speicherplatz Y_2 merkt man sich, ob die Suche nach dem passenden Vielfachen von m noch fortgesetzt werden muss. Deshalb wird er mit 1 (= 'true') vorbelegt. Sein Inhalt wird zu 0 verändert

- entweder durch $Y_2 := 0$, wenn mit $Eq(Y_1, X_j, X_k)$ festgestellt wurde, dass das aktuelle Vielfache von m mit n übereinstimmt,
- oder durch $Leq(Y_1, X_j, Y_2)$, wenn das aktuelle Vielfache von m bereits größer als n ist.

Das Programmstück terminiert, weil einer dieser beiden Fälle irgendwann eintritt. In beiden Fällen wird X_k durch den letzten Test $Eq(Y_1, X_j, X_k)$ mit der korrekten Antwort besetzt.

Berechenbarkeit

Nach diesen Beispielen dürfte es klar sein, wie man größere *while*-Programme nach und nach aus kleinen Bausteinen zusammensetzen kann. Deshalb geben wir für alle weiteren Bausteine unseres Interpreters nur noch die Idee an:

Mit Hilfe von $Divides(X_i, X_j, X_k)$ programmiert man einen *Primzahltest*: Eine Zahl $n \geq 2$ ist genau dann eine Primzahl, wenn man unter den Zahlen $2, \dots, n - 1$ keinen Teiler von n findet.

Mit Hilfe des Primzahltests schreibt man ein Programm, das zu jeder Zahl $i > 0$ die *i -te Primzahl* p_i findet: Man führt den Primzahltest für $2, 3, 4, \dots$ aus, bis er zum i -ten Mal erfolgreich ist.

Damit erhält man leicht ein Programm, das zu zwei Zahlen i und n die *Primzahl-Potenz* p_i^n berechnet.

Indem man dieses Programm mit dem Teilbarkeitstest kombiniert, kann man zu zwei Zahlen $i, n \neq 0$ die größte Zahl k berechnen, für die p_i^k Teiler von n ist. Diese Zahl k ist der *Exponent* von p_i in der Primfaktorzerlegung von n .

Berechenbarkeit

Damit können wir die *Länge* und die einzelnen *Komponenten* einer codierten Zahlenliste bestimmen: Wenn $n = \langle n_1, \dots, n_m \rangle$, so ist $m = i - 1$, wobei p_i die erste Primzahl ist, die n nicht teilt, und $n_i = k - 1$, wobei k der Exponent von p_i in der Primfaktorzerlegung von n ist.

Insbesondere erhalten wir den *Inhalt eines Speicherplatzes* aus einem codierten Zustand: Wenn $n = \langle \sigma \rangle_m = \langle \sigma(X_0), \dots, \sigma(X_m) \rangle$, so ist $\sigma(X_i)$ die $(i + 1)$ -te Komponente der durch n codierten Liste.

Zur Simulation des Übergangsschrittes müssen wir uns jetzt noch überlegen

1. wie man einen Zustand an einer Stelle verändert, d.h. wie man aus der Codierung von σ die Codierung von $\sigma[n/X_k]$ erhält,
2. wie man aus einer Anweisungsliste die erste Anweisung abspaltet, d.h. wie man $\lceil st \rceil$ und $\lceil st/ \rceil$ aus $\lceil st; st/ \rceil$ erhält,
3. wie man Anweisungslisten aneinanderhängt, d.h. wie man $\lceil st/_1 st/_2 \rceil$ aus $\lceil st/_1 \rceil$ und $\lceil st/_2 \rceil$ erhält.

Berechenbarkeit

Die *Veränderung eines Zustands* führt man so durch:

Gegeben seien $m, n, k \in \mathbb{N}$ und die Codierung $\langle \sigma \rangle_m$ eines Zustands σ . Zu berechnen ist die Codierung $\langle \sigma' \rangle_m$ des veränderten Zustands $\sigma' = \sigma[n/X_k]$. Per Definition der Codierung gilt

$$\langle \sigma' \rangle_m = \langle \sigma'(X_0), \dots, \sigma'(X_m) \rangle = \prod_{i=0}^m p_{i+1}^{\sigma'(X_i)+1}$$

Man berechnet dieses Produkt, indem man für $i = 0, \dots, m$

- p_{i+1} bestimmt,
- $\sigma(X_i)$ aus $\langle \sigma \rangle_m$ errechnet,
- im Falle $i = k$ die Potenz p_i^{n+1} und im Falle $i \neq k$ die Potenz $p_i^{\sigma(X_i)+1}$ berechnet,
- und das Zwischenergebnis mit dieser Potenz multipliziert.

Dies kann man in einer *while*-Anweisung durchführen.

Berechenbarkeit

Aneinanderhängen von Anweisungslisten:

Anweisungslisten sind ja als Zahlenlisten codiert, also muss man aus

$$\langle n_1, \dots, n_k \rangle = \prod_{i=1}^k p_i^{n_i+1} \quad \text{und} \quad \langle m_1, \dots, m_l \rangle = \prod_{i=1}^l p_i^{m_i+1}$$

die Zahl

$$\langle n_1, \dots, n_k, m_1, \dots, m_l \rangle = \prod_{i=1}^k p_i^{n_i+1} * \prod_{i=1}^l p_{k+i}^{m_i+1}$$

errechnen. Das lässt sich ähnlich durchführen wie die Veränderung eines Zustands.

Aufspalten einer Anweisungsliste:

Hier muss man aus $\langle n_1, \dots, n_k \rangle$ die erste Zahl n_1 und die Codierung

$$\langle n_2, \dots, n_k \rangle = \prod_{i=2}^k p_{i-1}^{n_i+1}$$

errechnen. Auch das lässt sich wieder ähnlich durchführen.
