VL-09: LOOP und WHILE Programme I

(Berechenbarkeit und Komplexität, WS 2018)

Gerhard Woeginger

WS 2018, RWTH

Organisatorisches

- Nächste Vorlesung:
  Freitag, November 30, 16:30–18:00 Uhr, Audimax

- Webseite:
  http://algo.rwth-aachen.de/Lehre/WS1819/BuK.php
  ($\longrightarrow$ Arbeitsheft zur Berechenbarkeit)

Wiederholung

Wdh.: Hilberts zehntes Problem

Hilberts zehntes Problem, im Originalwortlaut (1900)

Eine *Diophantische* Gleichung mit irgend welchen Unbekannten und mit ganzen rationalen Zahlencoefficienten sei vorgelegt: man soll ein Verfahren angeben, nach welchem sich mittelst einer endlichen Anzahl von Operationen entscheiden lässt, ob die Gleichung in ganzen rationalen Zahlen lösbar ist.

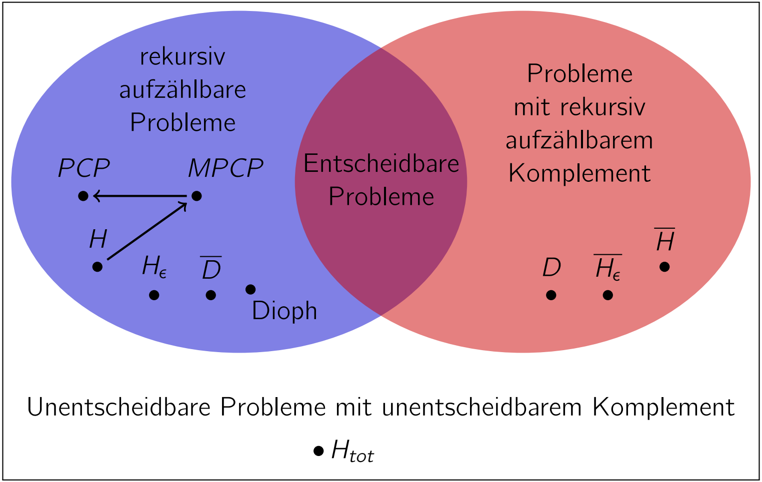
$$\text{Dioph} = \{\, \langle p \rangle \mid p \text{ ist ein Polynom mit ganzzahligen Koeffizienten und mit (mindestens) einer ganzzahligen Nullstelle}\}$$

Satz von Matiyasevich (1970)

Es ist unentscheidbar, ob ein (multivariates) ganzzahliges Polynom eine ganzzahlige Nullstelle besitzt.

Der Beweis basiert auf Vorarbeiten (1950–1970) von Martin Davis, Hilary Putnam und Julia Robinson.

Wdh.: Berechenbarkeitslandschaft



Wdh.: Turing-mächtige Rechnermodelle

Definition

Ein Rechnermodell wird als Turing-mächtig bezeichnet,
wenn jede Funktion, die durch eine TM berechnet werden kann,
auch durch dieses Rechnermodell berechnet werden kann.

- Da die Registermaschine (RAM) die Turingmaschine simulieren kann, ist sie Turing-mächtig
- Auch die Mini-RAM (eine schwächere Variante der RAM mit stark eingeschränktem Befehlssatz) ist Turing-mächtig

Wdh.: Turing-Mächtigkeit

- Reines HTML (ohne JavaScript; ohne Browser) ist nicht Turing-mächtig
- Tabellenkalkulationen (ohne Schleifen) sind nicht Turing-mächtig

- Der Lambda Calculus von Alonzo Church ist äquivalent zur TM, und daher Turing-mächtig
- Die μ-rekursiven Funktionen von Kurt Gödel sind äquivalent zur TM, und daher Turing-mächtig
- Alle gängigen höheren Programmiersprachen sind Turing-mächtig: Algol, Pascal, C, FORTRAN, COBOL, Java, Smalltalk, Ada, C++, Python, LISP, Haskell, PROLOG, etc.
- PostScript, Tex, Latex sind Turing-mächtig
- Sogar PowerPoint ist Turing-mächtig (Animated Features)

Vorlesung VL-09
LOOP und WHILE Programme I

- Die Programmiersprache LOOP
- Die Programmiersprache WHILE
- WHILE versus LOOP
- WHILE ist Turing-mächtig

Die Programmiersprache LOOP

Programmiersprache LOOP

Wir betrachten eine einfache Programmiersprache namens LOOP, deren Programme aus den folgenden syntaktischen Komponenten aufgebaut sind:

- Variablen: x_1 x_2 x_3 ...
- Konstanten: 0 und 1
- Symbole: $:=$ $+$;
- Schlüsselwörter: LOOP DO ENDLOOP

LOOP / Syntax (1)

Die Syntax von LOOP ist induktiv definiert.

Induktive Definition / Induktionsanfang:

Zuweisungen

Für alle Variablen x_i und x_j und für jede Konstante $c \in \{0, 1\}$ ist die Zuweisung

$$x_i \ := \ x_j + c$$

ein LOOP Programm.

LOOP / Syntax (2)

Induktive Definition / Induktionsschritte:

Hintereinanderausführung

Falls P_1 und P_2 LOOP Programme sind, so ist auch

$$P_1;\ P_2$$

ein LOOP Programm.

LOOP-Konstrukt

Falls P ein LOOP Programm ist, so ist auch

$$\text{LOOP } x_i \text{ DO } P \text{ ENDLOOP}$$

ein LOOP Programm.

LOOP / Semantik (1)

- Die Eingabe ist in den Variablen $x_1, \ldots, x_m$ enthalten.
  Alle anderen Variablen werden mit 0 initialisiert.
- Das Resultat eines LOOP Programms ist die Zahl, die sich am Ende der Abbarbeitung in der Variablen x_0 ergibt.

LOOP Programme der Form $x_i := x_j + c$
sind Zuweisungen des Wertes $x_j + c$ an die Variable x_i.

In einem LOOP Programm $P_1; P_2$
wird zunächst P_1 und danach P_2 ausgeführt.

Das Programm LOOP x_i DO P ENDLOOP hat folgende Bedeutung:
P wird x_i-mal hintereinander ausgeführt.

(Nur der Wert von x_i zu Beginn der Schleife ist relevant. Ändert sich der Wert von x_i im Inneren von P, so hat dies keinen Einfluss auf die Anzahl der Wiederholungen.)

LOOP / Semantik (2)

Ein LOOP Programm P mit k Variablen
berechnet eine k-stellige Funktion der Form $[P]\colon \mathbb{N}^k \to \mathbb{N}^k$.

Ist P die Zuweisung $x_i := x_j + c$,
so ist $[P](r_1, \ldots, r_k) = (r_1, \ldots, r_{i-1}, r_j + c, r_{i+1}, \ldots, r_k)$.

Ist $P = P_1; P_2$ eine Hintereinanderausführung,
so ist $[P](r_1, \ldots, r_k) = [P_2]([P_1](r_1, \ldots, r_k))$.

Ist $P = \text{LOOP } x_i \text{ DO } Q \text{ ENDLOOP}$ ein LOOP-Konstrukt,
so gilt $[P](r_1, \ldots, r_k) = [Q]^{r_i}(r_1, \ldots, r_k)$.

LOOP: Beispiele und Macros

LOOP Programme / Beispiele (1)

Das folgende Programm simuliert die Zuweisung $x_j := x_i$.

Beispiel A

$$x_j := x_i + 0$$

Es sei x_{zero} eine Dummy-Variable, die mit 0 initialisiert wird und deren Wert nie verändert wird. Das folgende $(c+1)$-zeilige Programm simuliert die Zuweisung $x_j := c$ eines konstanten Wertes $c \geq 0$ an eine Variable.

Beispiel B

$$\begin{aligned}
x_j &:= x_{zero};\\
x_j &:= x_j + 1;\\
x_j &:= x_j + 1;\\
\vdots &\quad \vdots\\
x_j &:= x_j + 1;
\end{aligned}$$

LOOP Programme / Beispiele (2)

Beispiel C

$x_0 := x_1$;
LOOP x_2 DO $x_0 := x_0 + 1$ ENDLOOP

Dieses Programm berechnet die Addition $x_0 := x_1 + x_2$

Beispiel D

$x_0 := 0$;
LOOP x_2 DO $x_0 := x_0 + x_1$ ENDLOOP

Dieses Programm berechnet die Multiplikation $x_0 := x_1 \cdot x_2$

LOOP Programme / Beispiele (3)

Übung

Skizzieren Sie LOOP Programme, die die folgenden Operationen berechnen:

- Die (modifizierte) Subtraktion $x_0 := x_1 \dot{-} x_2$.
  Für $x_1 < x_2$ erhält x_0 den Wert 0; andernfalls den Wert $x_1 - x_2$
- Die Division ohne Rest $x_0 := x_1 \text{ DIV } x_2$
- Die Modulo-Operation $x_0 := x_1 \text{ MOD } x_2$

LOOP Programme / Beispiele (4)

Es seien P_1 und P_2 LOOP Programme, in denen die drei Variablen x_1, x_2 und x_3 nicht vorkommen.

Beispiel E

$x_2 := 1;\ \ x_3 := 0;$
LOOP x_1 DO $x_2 := 0$; $x_3 := 1$ ENDLOOP;
LOOP x_2 DO P_1 ENDLOOP;
LOOP x_3 DO P_2 ENDLOOP

Dieses Programm entspricht dem Konstrukt:
IF $x_1 = 0$ THEN P_1 ELSE P_2 ENDIF

Übung

Skizzieren Sie ein LOOP Programm,
das "IF $x_1 = c$ THEN P_1 ELSE P_2 ENDIF" simuliert.

Die Programmiersprache WHILE

Programmiersprache WHILE

Die Programme der Programmiersprache WHILE sind aus den folgenden syntaktischen Komponenten aufgebaut:

- Variablen: x_1 x_2 x_3 ...
- Konstanten: 0 und 1
- Symbole: $:=$ $+$ $;$ $\neq$
- Schlüsselwörter: WHILE DO ENDWHILE

WHILE / Syntax

- Die Syntax von WHILE ist induktiv definiert, und stimmt weitgehend mit der Syntax von LOOP überein.
- Zuweisungen $x_i \ := \ x_j + c$ und die Hintereinanderausführung $P_1; P_2$ sind genau wie in LOOP definiert.
- Der Hauptunterschied zu LOOP besteht im Schleifen-Konstrukt.

WHILE-Konstrukt

Falls P ein WHILE Programm ist und x_i eine Variable, so ist auch

$$\text{WHILE } x_i \neq 0 \text{ DO } P \text{ ENDWHILE}$$

ein WHILE Programm.

WHILE / Semantik

- Die Eingabe ist in den Variablen $x_1, \ldots, x_m$ enthalten.
  Alle anderen Variablen werden mit 0 initialisiert.
- Das Resultat eines WHILE Programms ist die Zahl, die sich am Ende der Abbarbeitung in der Variablen x_0 ergibt.

Das Programm WHILE $x_i \neq 0$ DO P ENDWHILE hat folgende Bedeutung:
P wird solange ausgeführt, bis x_i den Wert 0 erreicht.

Ein WHILE Programm P mit k Variablen
berechnet eine k-stellige Funktion der Form $[P]\colon \mathbb{N}^k \to \mathbb{N}^k$.

Ist $P =$ WHILE $x_i \neq 0$ DO Q ENDWHILE ein WHILE-Konstrukt,
so ist $[P](r_1, \ldots, r_k) = [Q]^\ell(r_1, \ldots, r_k)$ für die kleinste Zahl ℓ,
für die die i-te Komponente von $[Q]^\ell(r_1, \ldots, r_k)$ gleich 0 ist.
Falls solch ein ℓ nicht existiert, so ist $[P](r_1, \ldots, r_k)$ undefiniert.

WHILE versus LOOP

WHILE versus LOOP (1)

Beobachtung

Die LOOP-Schleife

$$\text{LOOP } x_i \text{ DO } P \text{ ENDLOOP}$$

kann durch die folgende WHILE-Schleife simuliert werden:

$$y := x_i$$
$$\text{WHILE } y \neq 0 \text{ DO } y := y - 1;\ P \ \text{ ENDWHILE}$$

Ergo: Jede LOOP-berechenbare Funktion $f : \mathbb{N}^k \to \mathbb{N}$ ist auch WHILE-berechenbar.

WHILE versus LOOP (2a)

Es gibt WHILE Programme, die nicht terminieren:

Beispiel

$x_1 := 1;$
WHILE $x_1 \neq 0$ DO $x_1 := x_1 + 1$ ENDWHILE

WHILE versus LOOP (2b)

LOOP Programme terminieren immer:

Satz

Jedes LOOP Programm hält auf jeder möglichen Eingabe nach endlich vielen Schritten an.

Beweis: Durch Induktion über den Aufbau des Programms.

- Zuweisungen
- Hintereinanderausführung $P = P_1; P_2$
- LOOP-Konstrukt $P = \text{LOOP } x_i \text{ DO } Q \text{ ENDLOOP}$

WHILE versus LOOP (3)

Wir werden zeigen:

Satz (wird heute bewiesen)

Die Programmiersprache WHILE ist Turing-mächtig.

(In anderen Worten: Jede berechenbare Funktion kann von einem WHILE Programm berechnet werden.)

Satz (wird in der nächsten Vorlesung bewiesen)

Die Programmiersprache LOOP ist nicht Turing-mächtig.

(In anderen Worten: Es existiert eine berechenbare totale Funktion, die von keinem LOOP Programm berechnet werden kann.)

Mächtigkeit von WHILE

Mächtigkeit von WHILE

Satz

Die Programmiersprache WHILE ist Turing-mächtig.

Beweis:

Wir zeigen, dass jede Funktion, die durch eine TM berechnet werden kann, auch durch ein WHILE Programm berechnet werden kann.

Simulation von TM durch WHILE (1)

Wir betrachten eine TM M.

- Zustandsmenge $Q = \{q_0, \ldots, q_t\}$
- Der Anfangszustand ist q_1, und der Endzustand ist q_0
- TM im Zustand $q_i \iff$ WHILE Variable $\text{Zustand} = i$

- Bandalphabet $\Gamma = \{1, 2, B\}$
- WHILE kodiert Buchstaben 1 durch Dezimalziffer 1, Buchstaben 2 durch Dezimalziffer 2, und Buchstaben B durch Dezimalziffer 0.

- Alle WHILE Variablen enthalten im Folgenden Dezimalzahlen

Simulierte Turingmaschine M:

$\cdots$	B	1	1	2	2	$\boxed{2}$	1	2	1	1	1	B	B	B	$\cdots$

δ	1	2	B
q_1			
q_2			
q_3		$(q_2, 1, R)$	

q_3

Entsprechende Konfiguration: $1122\, q_3\, 212111$

Vier entsprechende Variablen im WHILE Programm:

Band-vor-Kopf	1122	Variable BandLinks
Kopf; Zustand	2; 3	Variable UntermKopf; Zustand
pfoK-ba-danB	11121	Variable BandRechts

Simulation von TM durch WHILE (2)

Jeder Rechenschritt von M wird durch einige WHILE-Befehle simuliert.

Jeder Rechenschritt der TM besteht (gemäss Überführungsfunktion) aus

(A) Update von Zustand

(B) Update von Symbol unterm Kopf

(C) Bewegung des Kopfes L,R,N

Beginn der Rechenschritt Simulation

- Aktueller Zustand steht in der Variablen Zustand
- Aktuelles Symbol unterm Kopf steht in der Variablen UntermKopf

Simulation von TM durch WHILE (3A)

Jeder Rechenschritt der TM besteht (gemäss Überführungsfunktion) aus

(A) Update von Zustand

(B) Update von Symbol unterm Kopf

(C) Bewegung des Kopfes L,R,N

Der Zustand wird auf den neuen Zustand q_i gesetzt,
indem man das folgende Programmstück ausführt:

```
Zustand := i;
```

Simulation von TM durch WHILE (3B)

Jeder Rechenschritt der TM besteht (gemäss Überführungsfunktion) aus

(A) Update von Zustand

(B) Update von Symbol unterm Kopf

(C) Bewegung des Kopfes L,R,N

Das Symbol unterm Kopf wird auf neues Symbol $\sigma \in \{0, 1, 2\}$ gesetzt, indem man das folgende Programmstück ausführt:

UntermKopf $:=$ σ;

Simulation von TM durch WHILE (3C-links)

Jeder Rechenschritt der TM besteht (gemäss Überführungsfunktion) aus

(A) Update von Zustand

(B) Update von Symbol unterm Kopf

(C) Bewegung des Kopfes L,R,N

Der Kopf wird einen Schritt nach links (L) bewegt,
indem man das folgende Programmstück ausführt:

$$\begin{aligned}
&\text{BandRechts} := 10 \cdot \text{BandRechts} + \text{UntermKopf};\\
&\text{UntermKopf} := \text{BandLinks}\ \ \text{MOD}\ \ 10;\\
&\text{BandLinks} := \text{BandLinks DIV } 10;
\end{aligned}$$

Simulation von TM durch WHILE (3C-rechts)

Jeder Rechenschritt der TM besteht (gemäss Überführungsfunktion) aus

(A) Update von Zustand

(B) Update von Symbol unterm Kopf

(C) Bewegung des Kopfes L,R,N

Der Kopf wird einen Schritt nach rechts (R) bewegt,
indem man das folgende Programmstück ausführt:

```
BandLinks := 10·BandLinks + UntermKopf;
UntermKopf := BandRechts MOD 10;
BandRechts := BandRechts DIV 10;
```

Simulation von TM durch WHILE (3C-nichts)

Jeder Rechenschritt der TM besteht (gemäss Überführungsfunktion) aus

(A) Update von Zustand

(B) Update von Symbol unterm Kopf

(C) Bewegung des Kopfes L,R,N

Der Kopf wird nicht bewegt (N),
indem man gar nichts macht und alle Variablen unverändert lässt:

Simulation von TM durch WHILE (4)

Schlussendlich die Grobstruktur der Simulation:

Initialisierung

Zustand := 1;
BandLinks := 0;
UntermKopf := *Erstes Symbol im Eingabewort (als Dezimalziffer)*;
BandRechts := *Restliches gespiegeltes Eingabewort (dezimal)*;

Simulation von TM durch WHILE (5)

Die äussere Schleife

```
WHILE  Zustand ≠ 0  DO
    IF Zustand=1 AND UntermKopf=0 THEN Schritt ENDIF;
    IF Zustand=1 AND UntermKopf=1 THEN Schritt ENDIF;
    IF Zustand=1 AND UntermKopf=2 THEN Schritt ENDIF;

    IF Zustand=2 AND UntermKopf=0 THEN Schritt ENDIF;
    IF Zustand=2 AND UntermKopf=1 THEN Schritt ENDIF;
    IF Zustand=2 AND UntermKopf=2 THEN Schritt ENDIF;

    IF Zustand=3 AND UntermKopf=0 THEN Schritt ENDIF;
       ⋮          ⋮          ⋮          ⋮
       ⋮          ⋮          ⋮          ⋮
    IF Zustand=t AND UntermKopf=0 THEN Schritt ENDIF;
    IF Zustand=t AND UntermKopf=1 THEN Schritt ENDIF;
    IF Zustand=t AND UntermKopf=2 THEN Schritt ENDIF;
ENDWHILE
```

Die Ackermann Funktion

Ackermann Funktion: Definition

Definition

Die Ackermann Funktion $A : \mathbb{N}^2 \to \mathbb{N}$ ist folgendermassen definiert:

$$
\begin{array}{lcll}
A(0, n) & = & n + 1 & \text{für } n \geq 0 \\
A(m + 1, 0) & = & A(m, 1) & \text{für } m \geq 0 \\
A(m + 1, n + 1) & = & A(m, A(m + 1, n)) & \text{für } m, n \geq 0
\end{array}
$$

Ackermann Funktion: Beispiele (1)

$$\begin{array}{lcll} A(0,n) & = & n+1 & \text{für } n \geq 0 \\ A(m+1,0) & = & A(m,1) & \text{für } m \geq 0 \\ A(m+1,n+1) & = & A(m, A(m+1,n)) & \text{für } m,n \geq 0 \end{array}$$

Ein paar Beispiele für $m = 1$:

$A(1,0) = A(0,1) = 2$

$A(1,1) = A(0, A(1,0)) = A(1,0) + 1 = 3$

$A(1,2) = A(0, A(1,1)) = A(1,1) + 1 = 4$

$A(1,3) = A(0, A(1,2)) = A(1,2) + 1 = 5$

Beobachtung

$$A(1,n) = n+2$$

Ackermann Funktion: Beispiele (2)

$$
\begin{array}{lcll}
A(0,n) & = & n+1 & \text{für } n \geq 0 \\
A(m+1,0) & = & A(m,1) & \text{für } m \geq 0 \\
A(m+1,n+1) & = & A(m,A(m+1,n)) & \text{für } m,n \geq 0
\end{array}
$$

Ein paar Beispiele für $m = 2$:

$A(2,0) = A(1,1) = 3$

$A(2,1) = A(1,A(2,0)) = A(2,0) + 2 = 5$

$A(2,2) = A(1,A(2,1)) = A(2,1) + 2 = 7$

Beobachtung

$$A(2,n) = 2n + 3$$

Ackermann Funktion: Beispiele (3)

$$
\begin{array}{lcll}
A(0,n) & = & n+1 & \text{für } n \geq 0 \\
A(m+1,0) & = & A(m,1) & \text{für } m \geq 0 \\
A(m+1,n+1) & = & A(m,A(m+1,n)) & \text{für } m,n \geq 0
\end{array}
$$

Und ein paar Beispiele für $m = 3$:

$$A(3,0) = A(2,1) = 5$$

$$A(3,1) = A(2,A(3,0)) = 2 \cdot A(3,0) + 3 = 13$$

$$A(3,2) = A(2,A(3,1)) = 2 \cdot A(3,1) + 3 = 29$$

$$A(3,3) = A(2,A(3,2)) = 2 \cdot A(3,2) + 3 = 61$$

Beobachtung

$$A(3,n) = 2^{n+3} - 3$$

Ackermann Funktion: Beispiele (4)

Zusammenfassung der Beispiele

Wenn man den ersten Parameter fixiert ...

- $A(1, n) = n + 2$
- $A(2, n) = 2n + 3$
- $A(3, n) = 2^{n+3} - 3$
- $A(4, n) = \underbrace{2^{2^{\cdot^{\cdot^{2}}}}}_{\substack{n+3 \text{ viele} \\ \text{Zweien}}} - 3$

Bereits der Wert $A(4, 2) = 2^{65536} - 3$
ist grösser als die Anzahl aller Atome im Weltraum.