Theorie der Informatik

13. LOOP-, WHILE- und GOTO-Berechenbarkeit

Malte Helmert Gabriele Röger

Universität Basel

9. April 2014

Überblick: Vorlesung

Vorlesungsteile

I. Logik ✓
II. Automatentheorie und formale Sprachen ✓
III. Berechenbarkeitstheorie
IV. Komplexitätstheorie

Überblick: Berechenbarkeitstheorie

III. Berechenbarkeitstheorie

12. Turing-Berechenbarkeit ✓
13. LOOP-, WHILE- und GOTO-Berechenbarkeit
14. primitive Rekursion und μ-Rekursion
15. Ackermannfunktion
16. Entscheidbarkeit, Reduktionen, Halteproblem
17. Postsches Korrespondenzproblem

~~Unentscheidbare Grammatik-Probleme~~

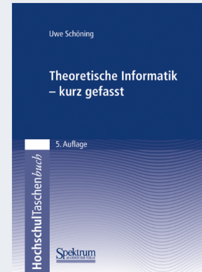
~~Gödelscher Satz und diophantische Gleichungen~~

Nachlesen

Literatur zu diesem Vorlesungskapitel

Theoretische Informatik - kurz gefasst
von Uwe Schöning (5. Auflage)

- Kapitel 2.3



Einleitung

Formale Berechnungsmodelle: LOOP/WHILE/GOTO

Formale Berechnungsmodelle

- Turingmaschinen
- LOOP-, WHILE-, GOTO-Programme
- primitiv rekursive Funktionen, μ-rekursive Funktionen

In diesem Kapitel lernen wir drei einfache Berechnungsformalismen (Programmiersprachen) kennen und vergleichen ihre Mächtigkeit mit Turingmaschinen:

- LOOP-Programme
- WHILE-Programme
- GOTO-Programme

LOOP-, WHILE- und GOTO-Programme: Grundkonzepte

- LOOP-, WHILE- und GOTO-Programme entsprechen strukturell (einfachen) „traditionellen“ Programmiersprachen
- verwenden endlich viele Variablen aus der Menge $\{x_0, x_1, x_2, \dots\}$, die Werte in $\mathbb{N}_0$ annehmen
- unterscheiden sich in Art der erlaubten „Anweisungen“

LOOP-Programme

LOOP-Programme: Syntax

Definition (LOOP-Programm)

LOOP-Programme sind definiert durch endliche Anwendung folgender Regeln:

- $x_i := x_j + c$ ist ein LOOP-Programm für jedes $i, j, c \in \mathbb{N}_0$ (Addition)
- $x_i := x_j - c$ ist ein LOOP-Programm für jedes $i, j, c \in \mathbb{N}_0$ (modifizierte Subtraktion)
- Wenn P_1 und P_2 LOOP-Programme sind, dann auch $P_1;P_2$ (Komposition)
- Wenn P ein LOOP-Programm ist, dann auch LOOP x_i DO P END für jedes $i \in \mathbb{N}_0$ (LOOP-Schleife)

LOOP-Programme: Semantik

Definition (Semantik von LOOP-Programmen)

Ein LOOP-Programm berechnet eine k-stellige Funktion $f : \mathbb{N}_0^k \to \mathbb{N}_0$. Die Berechnung von $f(n_1, \dots, n_k)$ erfolgt wie folgt:

1. Zu Beginn beinhalten die Variablen $x_1, \dots, x_k$ die Werte $n_1, \dots, n_k$, alle anderen Variablen den Wert 0.
2. Danach modifiziert das Programm die Variableninhalte wie auf den folgenden Folien beschrieben.
3. Der Inhalt der Variable x_0 nach Ausführung des Programms ist das Ergebnis der Berechnung.

LOOP-Programme: Semantik

Definition (Semantik von LOOP-Programmen)

Auswirkung von $x_i := x_j + c$:

- Der Variable x_i wird der aktuelle Wert von x_j plus c zugewiesen.
- Alle anderen Variableninhalte bleiben gleich.

LOOP-Programme: Semantik

Definition (Semantik von LOOP-Programmen)

Auswirkung von $x_i := x_j - c$:

- Der Variable x_i wird der aktuelle Wert von x_j minus c zugewiesen, sofern dieser Wert nicht-negativ ist.
- Ansonsten erhält x_i den Wert 0.
- Alle anderen Variableninhalte bleiben gleich.

LOOP-Programme: Semantik

Definition (Semantik von LOOP-Programmen)

Auswirkung von P_1; P_2:

- Es wird zuerst P_1 und anschliessend (auf den modifizierten Variablenwerten) P_2 ausgeführt.

LOOP-Programme: Semantik

Definition (Semantik von LOOP-Programmen)

Auswirkung von LOOP x_i DO P END:

- Sei m der Wert der Variable x_i bei Beginn der Ausführung.
- Es wird m mal hintereinander das Programm P ausgeführt.

LOOP-berechenbare Funktionen

Definition (LOOP-berechenbar)

Eine Funktion $f : \mathbb{N}_0^k \to \mathbb{N}_0$ heisst LOOP-berechenbar, wenn ein LOOP-Programm existiert, das sie berechnet.

Anmerkung: Nicht-totale Funktionen sind nie LOOP-berechenbar. (Warum nicht?)

LOOP-Programme: Beispiel

Beispiel (LOOP-Programm für $f(x_1, x_2)$)

LOOP x_1 DO
  LOOP x_2 DO
    $x_0 := x_0 + 1$
  END
END

Welche 2-stellige Funktion berechnet das Programm?

Syntaktischer Zucker oder essentielles Feature?

- Wir untersuchen die Mächtigkeit vom Programmiersprachen (und anderen Formalismen).
- Reichhaltige Sprach-Features sind nützlich, um komplexe Programme schreiben zu können.
- Minimalistische Formalismen sind nützlicher, wenn wir Beweise über alle Programme führen wollen.

$\leadsto$ Interessenkonflikt!

Idee:

- Verwende minimalistischen Kern für Beweise.
- Verwende syntaktischen Zucker, um Programme zu schreiben.

Beispiel: syntaktischer Zucker

Beispiel (Syntaktischer Zucker)

Wir schlagen fünf neue Syntaxkonstrukte vor
(mit der offensichtlichen Semantik):

- $x_i := x_j$ für $i, j \in \mathbb{N}_0$
- $x_i := c$ für $i, c \in \mathbb{N}_0$
- $x_i := x_j + x_k$ für $i, j, k \in \mathbb{N}_0$
- IF $x_i \neq 0$ THEN P END für $i \in \mathbb{N}_0$
- IF $x_i = c$ THEN P END für $i, c \in \mathbb{N}_0$

Können wir diese mit den existierenden Konstrukten simulieren?

Beispiel: syntaktischer Zucker

Beispiel (Syntaktischer Zucker)

$x_i := x_j$ für $i, j \in \mathbb{N}_0$

Simulation mit existierenden Konstrukten?

Beispiel: syntaktischer Zucker

Beispiel (Syntaktischer Zucker)

$x_i := x_j$ für $i, j \in \mathbb{N}_0$

Einfache Kurzschreibweise für $x_i := x_j + 0$.

Beispiel: syntaktischer Zucker

Beispiel (Syntaktischer Zucker)

$x_i := c$ für $i, c \in \mathbb{N}_0$

Simulation mit existierenden Konstrukten?

Beispiel: syntaktischer Zucker

Beispiel (Syntaktischer Zucker)

$x_i := c$ für $i, c \in \mathbb{N}_0$

Einfache Kurzschreibweise für $x_i := x_j + c$,
wobei x_j ein frische Variable ist, d.h. eine ansonsten unbenutzte Variable, die keine Eingabevariable ist.
(Daher enthält x_j zwangsläufig den Wert 0.)

Beispiel: syntaktischer Zucker

Beispiel (Syntaktischer Zucker)

$x_i := x_j + x_k$ für $i, j, k \in \mathbb{N}_0$

Simulation mit existierenden Konstrukten?

Beispiel: syntaktischer Zucker

Beispiel (Syntaktischer Zucker)

$x_i := x_j + x_k$ für $i, j, k \in \mathbb{N}_0$

Abkürzung für:

```
x_i := x_j;
LOOP x_k DO
  x_i := x_i + 1
END
```

Analog verwenden wir im Folgenden auch:

- $x_i := x_j - x_k$
- $x_i := x_j + x_k - c - x_m + d$
- usw.

Beispiel: syntaktischer Zucker

Beispiel (Syntaktischer Zucker)

IF $x_i \neq 0$ THEN P END für $i \in \mathbb{N}_0$

Simulation mit existierenden Konstrukten?

Beispiel: syntaktischer Zucker

Beispiel (Syntaktischer Zucker)

IF $x_i \neq 0$ THEN P END für $i \in \mathbb{N}_0$

Abkürzung für:

```
x_j := 0;
LOOP x_i DO
   x_j := 1
END;
LOOP x_j DO
   P
END
```

wobei x_j eine frische Variable ist.

Beispiel: syntaktischer Zucker

Beispiel (Syntaktischer Zucker)

IF $x_i = c$ THEN P END für $i, c \in \mathbb{N}_0$

Simulation mit existierenden Konstrukten?

Beispiel: syntaktischer Zucker

Beispiel (Syntaktischer Zucker)

IF $x_i = c$ THEN P END für $i, c \in \mathbb{N}_0$

Abkürzung für:

$x_j := 1;$
$x_k := x_i - c;$
IF $x_k \neq 0$ THEN $x_j := 0$ END;
$x_k := c - x_i;$
IF $x_k \neq 0$ THEN $x_j := 0$ END;
IF $x_j \neq 0$ THEN
  P
END

wobei x_j und x_k frische Variablen sind.

Geht es noch minimalistischer?

- Wir sehen, dass wir z. B. IF-Konstrukte nicht benötigen, da sie syntaktischer Zucker sind.
- Können wir LOOP-Programme noch minimalistischer gestalten als in unserer Definition?

Geht es noch minimalistischer?

- Wir sehen, dass wir z. B. IF-Konstrukte nicht benötigen, da sie syntaktischer Zucker sind.
- Können wir LOOP-Programme noch minimalistischer gestalten als in unserer Definition?

Vereinfachung 1

Statt $x_i := x_j + c$ und $x_i := x_j - c$ reicht es, nur die Konstrukte

- $x_i := x_j$,
- $x_i := x_i + 1$ und
- $x_i := x_i - 1$

zu erlauben.
Warum?

Geht es noch minimalistischer?

- Wir sehen, dass wir z. B. IF-Konstrukte nicht benötigen, da sie syntaktischer Zucker sind.
- Können wir LOOP-Programme noch minimalistischer gestalten als in unserer Definition?

Vereinfachung 2:

Das Konstrukt $x_i := x_j$ kann weggelassen werden,
da man es durch die anderen Konstrukte simulieren kann:

LOOP x_i DO
  $x_i := x_i - 1$
END;
LOOP x_j DO
  $x_i := x_i + 1$
END

WHILE-Programme

WHILE-Programme: Syntax

Definition (WHILE-Programm)

WHILE-Programme sind definiert durch endliche Anwendung folgender Regeln:

- $x_i := x_j + c$ ist ein WHILE-Programm für jedes $i, j, c \in \mathbb{N}_0$ (Addition)
- $x_i := x_j - c$ ist ein WHILE-Programm für jedes $i, j, c \in \mathbb{N}_0$ (modifizierte Subtraktion)
- Wenn P_1 und P_2 WHILE-Programme sind, dann auch $P_1;P_2$ (Komposition)
- Wenn P ein WHILE-Programm ist, dann auch WHILE $x_i \neq 0$ DO P END für jedes $i \in \mathbb{N}_0$ (WHILE-Schleife)

WHILE-Programme: Semantik

Definition (Semantik von WHILE-Programmen)

Die Semantik von WHILE-Programmen ist genauso definiert wie bei LOOP-Programmen.

Auswirkung von WHILE $x_i \neq 0$ DO P END:

- Falls x_i den Wert 0 beinhaltet, ist die Ausführung des Programms beendet.
- Ansonsten führe P aus.
- Wiederhole die vorigen beiden Schritte beliebig oft.

WHILE-berechenbare Funktionen

Definition (WHILE-berechenbar)

Eine Funktion $f : \mathbb{N}_0^k \to \mathbb{N}_0$ heisst WHILE-berechenbar, wenn ein WHILE-Programm existiert, das sie berechnet.

WHILE-Berechenbarkeit vs. LOOP-Berechenbarkeit

Satz (WHILE-Berechenbarkeit vs. LOOP-Berechenbarkeit)

Jede LOOP-berechenbare Funktion ist WHILE-berechenbar.
Die Umkehrung gilt nicht.

WHILE-Programme sind also echt mächtiger
als LOOP-Programme.

WHILE-Berechenbarkeit vs. LOOP-Berechenbarkeit

Beweis.

Teil 1: Jede LOOP-berechenbare Funktion ist WHILE-berechenbar.

Wir konstruieren zu jedem LOOP-Programm
ein äquivalentes WHILE-Programm.
Ersetze hierfür jedes Auftreten von LOOP x_i DO P END durch

```
x_j := x_i;
WHILE x_j ≠ 0 DO
  x_j := x_j − 1;
  P
END
```

wobei x_j jeweils eine frische Variable ist. ...

WHILE-Berechenbarkeit vs. LOOP-Berechenbarkeit

Beweis (Fortsetzung).

Teil 2: Nicht jede WHILE-berechenbare Funktion ist LOOP-berechenbar.

Das WHILE-Programm

$x_1 := 1$
WHILE $x_1 \neq 0$ DO
  $x_1 := 1$
END

zeigt, dass die überall undefinierte Funktion WHILE-berechenbar ist. LOOP-berechenbare Funktionen sind aber immer total. □

Anmerkung: Wir werden später zeigen, dass es auch totale Funktionen gibt, die WHILE-berechenbar, aber nicht LOOP-berechenbar sind.

WHILE-Berechenbarkeit vs. Turing-Berechenbarkeit

Satz (WHILE-Berechenbarkeit vs. Turing-Berechenbarkeit)

Jede WHILE-berechenbare Funktion ist Turing-berechenbar.

(Die umgekehrte Aussage diskutieren wir später.)

WHILE-Berechenbarkeit vs. Turing-Berechenbarkeit

Beweisskizze.

Wir konstruieren zu jedem WHILE-Programm
eine äquivalente deterministische Turingmaschine.

Seien $x_1, \ldots, x_k$ die Eingabevariablen des WHILE-Programms,
und seien $x_0, \ldots, x_m$ alle verwendeten Variablen.

Grundideen:

- Die DTM simuliert die einzelnen Ausführungsschritte des WHILE-Programms.
- Vor und nach jedem WHILE-Programm-Schritt enthält das Band den Inhalt $bin(n_0)\#bin(n_1)\#\ldots\#bin(n_m)$, wobei n_i der Wert der WHILE-Programm-Variablen x_i ist.
- Es reicht, „minimalistische“ WHILE-Programme nachzubilden ($x_i := x_i + 1$, $x_i := x_i - 1$, Komposition, WHILE-Schleife).

...

WHILE-Berechenbarkeit vs. Turing-Berechenbarkeit

Beweisskizze (Fortsetzung).

Die DTM besteht aus drei hintereinander geschalteten Teilen:

- Initialisierung:
  - Schreibe 0# vor den benutzten Teil des Bandes.
  - Schreibe $(m - k)$ mal #0 hinter den benutzten Teil des Bandes.
- Programmausführung:
  Simuliere das WHILE-Programm (siehe nächste Folie).
- Aufräumen:
  - Ersetze alle Zeichen ab dem ersten # durch □
    und gehe abschliessend zum ersten Zeichen, das nicht □ ist.

...

WHILE-Berechenbarkeit vs. Turing-Berechenbarkeit

Beweisskizze (Fortsetzung).

Simulation von $x_i := x_i + 1$:

1. Gehe nach links bis Blank erreicht,
   dann einen Schritt nach rechts.
2. $(i + 1)$ mal: Gehe nach rechts, bis # oder $\square$ erreicht.
3. Gehe einen Schritt nach links.

$\rightsquigarrow$ Wir stehen jetzt auf der letzten Ziffer der Kodierung von x_i.

4. Führe DTM für Inkrement um 1 aus.
   (Schwierigster Teil: „Platz schaffen“,
   wenn sich die Zahl der Binärziffern erhöht.)

...

WHILE-Berechenbarkeit vs. Turing-Berechenbarkeit

Beweisskizze (Fortsetzung).

Simulation von $x_i := x_i - 1$:

1. Gehe zur letzten Ziffer von x_i (siehe vorige Folie).
2. Teste, ob die Ziffer eine 0 und das Symbol links davon # oder $\square$ ist. Dann fertig.
3. Ansonsten: führe DTM für Dekrement um 1 aus.
   Wenn dabei eine führende 0 gefolgt von einer Ziffer entsteht, die führende 0 entfernen. (Band „zusammen schieben“.)

...

WHILE-Berechenbarkeit vs. Turing-Berechenbarkeit

Beweisskizze (Fortsetzung).

Simulation von P_1; P_2:

1. Baue rekursiv DTMs M_1 für P_1 und M_2 für P_2.
2. Baue daraus eine DTM für P_1; P_2,
   indem alle Übergänge auf Endzustände von M_1
   statt dessen auf den Startzustand von M_2 übergehen.

...

WHILE-Berechenbarkeit vs. Turing-Berechenbarkeit

Beweisskizze (Fortsetzung).

Simulation von WHILE $x_i \neq 0$ DO P END:

1. Baue rekursiv DTM M für P.
2. Baue eine DTM M' für WHILE $x_i \neq 0$ DO P END, die wie folgt arbeitet:
   1. Gehe zur letzten Ziffer von x_i.
   2. Teste, ob dort das Symbol 0 und davor # oder $\square$ steht. Falls ja: fertig.
   3. Ansonsten führe M aus, wobei in M alle Übergänge auf den Endzustand von M durch Übergänge auf den Anfangszustand von M' ersetzt werden.

Zwischenstand

Wir wissen jetzt:

- WHILE-Programme sind echt mächtiger als LOOP-Programme.
- Deterministische Turingmaschinen sind mindestens so mächtig wie WHILE-Programme.

Sind DTMs echt mächtiger als WHILE-Programme oder gleich mächtig?

Um diese Frage zu beantworten, machen wir einen Umweg über einen weiteren Programmiersprachen-Formalismus.

GOTO-Programme

GOTO-Programme: Syntax

Definition (GOTO-Programm)

Ein GOTO-Programm ist gegeben durch eine endliche Folge
$L_1 : A_1, L_2 : A_2, \ldots, L_n : A_n$
von Marken (Labels) und Anweisungen.

Anweisungen sind von folgender Form:

- $x_i := x_j + c$ für jedes $i, j, c \in \mathbb{N}_0$ (Addition)
- $x_i := x_j - c$ für jedes $i, j, c \in \mathbb{N}_0$ (modifizierte Subtraktion)
- HALT (Programmende)
- GOTO L_j für $1 \leq j \leq n$ (Sprung)
- IF $x_i = c$ THEN GOTO L_j für $i, c \in \mathbb{N}_0$, $1 \leq j \leq n$ (bedingter Sprung)

GOTO-Programme: Semantik

Definition (Semantik von GOTO-Programmen)

- Ein- und Ausgaben und Variableninhalte funktionieren genauso wie bei LOOP- und WHILE-Programmen.
- Addition und modifizierte Subtraktion funktionieren genauso wie bei LOOP- und WHILE-Programmen.
- Die Ausführung beginnt mit der Anweisung A_1.
- Nach der Anweisung A_i wird die Anweisung A_{i+1} ausgeführt. (Falls $i = n$, ist die Ausführung beendet.)
- Ausnahmen zur vorigen Regel:
  - HALT beendet die Programmausführung.
  - Nach GOTO L_j geht die Ausführung mit Anweisung A_j weiter.
  - Nach IF $x_i = c$ THEN GOTO L_j geht die Ausführung mit A_j weiter, falls die Variable x_i aktuell den Wert c beinhaltet.

GOTO-berechenbare Funktionen

Definition (GOTO-berechenbar)

Eine Funktion $f : \mathbb{N}_0^k \to \mathbb{N}_0$ heisst GOTO-berechenbar, wenn ein GOTO-Programm existiert, das sie berechnet.

GOTO-Berechenbarkeit vs. WHILE-Berechenbarkeit

Satz (GOTO-Berechenbarkeit vs. WHILE-Berechenbarkeit)

Jede GOTO-berechenbare Funktion ist WHILE-berechenbar.

Dabei reicht die Verwendung einer einzigen WHILE-Schleife aus.

(Die umgekehrte Aussage diskutieren wir später.)

GOTO-Berechenbarkeit vs. WHILE-Berechenbarkeit

Beweisskizze.

Wir konstruieren zu jedem GOTO-Programm ein äquivalentes WHILE-Programm mit nur einer WHILE-Schleife.

Ideen:

- Verwende eine frische Variable, um die nächste auszuführende Anweisung zu speichern.
  - $\rightsquigarrow$ Die Variable hat natürlich die Form x_i, aber zur Lesbarkeit schreiben wir sie pc für „program counter“.
- GOTO simuliert durch Zuweisung an pc.
- Hat pc den Wert 0, wird das Programm beendet.

...

GOTO-Berechenbarkeit vs. WHILE-Berechenbarkeit

Beweisskizze (Fortsetzung).

Sei $\mathrm{L}_1 : A_1, \mathrm{L}_2 : A_2, \ldots, \mathrm{L}_n : A_n$ das gegebene GOTO-Programm.

Grundstruktur des WHILE-Programms:

$pc := 1;$
WHILE $pc \neq 0$ DO
  IF $pc = 1$ THEN (Übersetzung für A_1) END;
  ...
  IF $pc = n$ THEN (Übersetzung für A_n) END;
  IF $pc = n + 1$ THEN $pc := 0$ END
END

...

GOTO-Berechenbarkeit vs. WHILE-Berechenbarkeit

Beweisskizze (Fortsetzung).

Übersetzung der einzelnen Anweisungen:

- $x_i := x_j + c$
  $\rightsquigarrow x_i := x_j + c;\ pc := pc + 1$
- $x_i := x_j - c$
  $\rightsquigarrow x_i := x_j - c;\ pc := pc + 1$
- HALT
  $\rightsquigarrow pc := 0$
- GOTO L_j
  $\rightsquigarrow pc := j$
- IF $x_i = c$ THEN GOTO L_j
  $\rightsquigarrow pc := pc + 1$; IF $x_i = c$ THEN $pc := j$ END

Nächster Zwischenstand

Wir wissen jetzt:

- WHILE-Programme sind echt mächtiger als LOOP-Programme.
- Deterministische Turingmaschinen sind mindestens so mächtig wie WHILE-Programme.
- WHILE-Programme sind mindestens so mächtig wie GOTO-Programme.

Wir zeigen nun, dass GOTO-Programme mindestens so mächtig sind wie DTMs und schliessen damit einen Kreis.

Turing-Berechenbarkeit vs. GOTO-Berechenbarkeit

Satz (Turing-Berechenbarkeit vs. GOTO-Berechenbarkeit)

Jede Turing-berechenbare numerische Funktion ist GOTO-berechenbar.

Beweis.

$\rightsquigarrow$ Tafel.

Endstand

Folgerung

Sei $f : \mathbb{N}_0^k \to \mathbb{N}_0$ eine partielle Funktion.

Die folgenden Aussagen sind äquivalent:

- *f ist Turing-berechenbar.*
- *f ist WHILE-berechenbar.*
- *f ist GOTO-berechenbar.*

Ferner gilt:

- *Jede LOOP-berechenbare Funktion ist Turing-/WHILE-/GOTO-berechenbar.*
- *Die Umkehrung gilt im Allgemeinen nicht.*

Zusammenfassung

Zusammenfassung

- drei neue Berechnungsformalismen für numerische Funktionen:
  - LOOP-, WHILE-, GOTO-Programme
  - programmiersprachenähnlicher als Turingmaschinen
- grundlegender Ansatz zum Vergleich von Formalismen:
  - Wie kann man mit bestimmten Features
    andere Features simulieren
    (vgl. syntaktischer Zucker, minimalistische Formalismen)?
  - Wie kann man mit einem Formalismus
    den anderen Formalismus simulieren?

Zusammenfassung

Ergebnisse der Untersuchung:

- Turingmaschinen, WHILE- und GOTO-Programme sind gleich mächtig.
- LOOP-Programme sind echt weniger mächtig.
- Für letzteres Ergebnis bisher nur ein Trivialbeispiel (LOOP-berechenbare Funktionen sind immer total; dies gilt nicht für die anderen Formalismen).