

THEORETISCHE INFORMATIK UND LOGIK

4. Vorlesung: Das Halteproblem und Reduktionen

Markus Krötzsch

Lehrstuhl Wissensbasierte Systeme

TU Dresden, 19. April 2017

LOOP und WHILE, Reprise

Ankündigung

Wegen großer Nachfrage wird eine

Zusätzliche Übungsgruppe

eingerrichtet:

Zeit: Donnerstag, 1. Doppelstunde

Raum: APB E008

Tutor: Daniel Borchmann

Bisher nicht eingeschriebene Studenten können sich ab sofort dafür eintragen (über jExam)

Zusammenfassung LOOP und WHILE

LOOP-Programme

- Terminieren immer
- Können fast alle praktisch relevanten Funktionen berechnen
- Können nicht jede berechenbare Funktion berechnen, z.B. Ackermann-Funktion, Sudan-Funktion, LOOP-Busy-Beaver

WHILE-Programme

- Verallgemeinern LOOP
- Terminieren nicht immer
- Können alle berechenbaren totalen und partiellen Funktionen berechnen

Die Kraft des LOOP

Auf der vorigen Folie steht: „LOOP-Programme können fast alle praktisch relevanten Funktionen berechnen.“

Stimmt das wirklich?

Idee: Die Schleife **WHILE** $x \neq 0$ **DO** P **END** kann mit dem folgenden LOOP-Programm simuliert werden:

```
LOOP max DO
  IF  $x \neq 0$  THEN
    P
  END
END
```

wenn man den Wert von max so setzt, dass er **mindestens** so groß ist wie die maximale Anzahl von Wiederholungen der simulierten WHILE-Schleife.

Was kann LOOP?

Es gibt aber viele Fälle, in denen man (eine obere Schranke von) max_P LOOP-berechnen kann:

Satz: Die folgenden Funktionen sind LOOP-berechenbar:

- $n \cdot x$ für beliebige natürliche Zahlen n
- x^n für beliebige natürliche Zahlen n
- n^x für beliebige natürliche Zahlen n
- $n^{\dots n^x}$ für beliebig hohe Türme von Exponenten n

Korollar: Jeder Algorithmus, der in Zeit $O(n^{\dots n^x})$ – oder weniger – läuft, berechnet eine LOOP-berechenbare Funktion. Insbesondere sind alle polynomiellen, exponentiellen oder mehrfach exponentiellen Algorithmen in LOOP implementierbar.

LOOP kann WHILE simulieren

Erkenntnis: LOOP kann WHILE für beliebig viele Schritte simulieren – man muss nur wissen, wie viele Schritte benötigt werden.

Kann man das ausrechnen? Ja!

Satz: Für ein beliebiges WHILE-Programm P sei $\text{max}_P(n_1, \dots, n_k)$ die maximale Anzahl an Schleifendurchläufen, die P bei der Eingabe n_1, \dots, n_k abarbeitet, oder undefiniert, wenn P bei dieser Eingabe nicht terminiert. Diese partielle Funktion $\text{max}_P : \mathbb{N}^k \rightarrow \mathbb{N}$ ist berechenbar.

Beweisskizze: Man kann P leicht so modifizieren, dass es die Maximalzahl der Schleifendurchläufe bestimmt und ausgibt. □

Wo ist der Haken?

Die Funktion max_P ist zwar berechenbar, aber im Allgemeinen nicht LOOP-berechenbar.

Universalität

Die Universalmaschine

Eine erste wichtige Beobachtung Turings war, dass TMs stark genug sind um andere TMs zu simulieren:

Schritt 1: Kodiere Turingmaschinen \mathcal{M} als Wörter $\text{enc}(\mathcal{M})$

Schritt 2: Konstruiere eine **universelle Turingmaschine** \mathcal{U} , die $\text{enc}(\mathcal{M})$ als Eingabe erhält und dann die Berechnung von \mathcal{M} simuliert

Schritt 2: Die universelle Turingmaschine

Wir definieren die universelle TM \mathcal{U} als Mehrbandturingmaschine:

Band 1: Eingabeband von \mathcal{U} : enthält $\text{enc}(\mathcal{M})\#\#\text{enc}(w)$

Band 2: Arbeitsband von \mathcal{U}

Band 3: Speichert den Zustand der simulierten Turingmaschine

Band 4: Arbeitsband der simulierten Turingmaschine

Die Arbeitsweise von \mathcal{U} ist leicht skizziert:

- \mathcal{U} prüft Eingabe, kopiert $\text{enc}(w)$ auf Band 4, verschiebt den Kopf auf Band 4 zum Anfang und initialisiert Band 3 mit $\text{enc}(0)$.
- In jedem Schritt liest \mathcal{U} ein (kodierte) Zeichen von der aktuellen Kopfposition auf Band (4), sucht für den simulierten Zustand (Band 3) einen passenden Übergang in $\text{enc}(\mathcal{M})$ auf Band 1:
 - Übergang gefunden: setze Band 3 auf den neuen Zustand; ersetzt das kodierte Zeichen auf Band 4 durch das neue Zeichen; verschiebe den Kopf auf Band 4 entsprechend
 - Übergang nicht gefunden: nimm Endzustand ein, falls der Zustand von Band 3 Endzustand in $\text{enc}(\mathcal{M})$ ist; halte

Schritt 1: Turingmaschinen kodieren

Jede vernünftige Kodierung einer TM $\mathcal{M} = \langle Q, \Sigma, \Gamma, \delta, q_0, F \rangle$ ist nutzbar, zum Beispiel die folgende (für DTMs):

- Wir verwenden das Alphabet $\{0, 1, \#\}$
- Zustände werden in beliebiger Reihenfolge nummeriert (mit Startzustand q_0) und binär kodiert:
 $Q = \{q_0, \dots, q_n\} \rightsquigarrow \text{enc}(Q) = \text{bin}(0)\#\dots\#\text{bin}(n)$
- Wir kodieren auch Γ und die Bewegungsrichtungen $\{R, L, N\}$ binär
- Ein Übergang $\delta(q_i, \sigma_n) = \langle q_j, \sigma_m, D \rangle$ wird als 5-Tupel kodiert:
 $\text{enc}(q_i, \sigma_n) = \text{bin}(i)\#\text{bin}(n)\#\text{bin}(j)\#\text{bin}(m)\#\text{bin}(D)$
- Die Übergangsfunktion wird kodiert als Liste aller dieser Tupel, getrennt mit #: $\text{enc}(\delta) = (\text{enc}(q_i, \sigma_n)\#)_{q_i \in Q, \sigma_i \in \Gamma}$
- Insgesamt setzen wir
 $\text{enc}(\mathcal{M}) = \text{enc}(Q)\#\#\text{enc}(\Sigma)\#\#\text{enc}(\Gamma)\#\#\text{enc}(\delta)\#\#\text{enc}(F)$

Passend dazu kann man auch beliebige Wörter kodieren:

- Für ein Wort $w = a_1 \dots a_\ell$ setzen wir $\text{enc}(w) = \text{bin}(a_1)\#\dots\#\text{bin}(a_\ell)$

Die Theorie der Software

Satz: Es gibt eine **universelle Turingmaschine** \mathcal{U} , die für Eingaben der Form $\text{enc}(\mathcal{M})\#\#\text{enc}(w)$ das Verhalten der DTM \mathcal{M} auf w simuliert:

- Falls \mathcal{M} auf w hält, dann hält \mathcal{U} auf $\text{enc}(\mathcal{M})\#\#\text{enc}(w)$ mit dem gleichen Ergebnis
- Falls \mathcal{M} auf w nicht hält, dann hält \mathcal{U} auf $\text{enc}(\mathcal{M})\#\#\text{enc}(w)$ ebenfalls nicht

Unsere Konstruktion ist für DTMs, die Sprachen erkennen – DTMs, die Funktionen berechnen, können ähnlich simuliert werden.

Praktische Konsequenzen:

- Universalrechner sind möglich
- Wir müssen nicht für jede neue Anwendung einen neuen Computer anschaffen
- Es gibt Software

Unentscheidbare Probleme und Reduktionen

„Beweis“ durch Intuition

Satz: Das Halteproblem P_{Halt} ist unentscheidbar.

„**Beweis:**“ Das Gegenteil wäre zu schön um wahr zu sein. Viele ungelöste Probleme könnte man damit direkt lösen.

Beispiel: Die Goldbachsche Vermutung (Christian Goldbach, 1742) besagt, dass jede gerade Zahl $n \geq 4$ die Summe zweier Primzahlen ist. Zum Beispiel ist $4 = 2 + 2$ und $100 = 47 + 53$.

Man kann leicht einen Algorithmus \mathcal{A} angeben, der die Goldbachsche Vermutung systematisch verifiziert, d.h., für alle geraden Zahlen ab 4 testet:

- Erfolg: teste die nächste gerade Zahl
- Misserfolg: terminiere mit Meldung „Goldbach hat sich geirrt!“

Die Frage „Wird \mathcal{A} halten?“ ist gleichbedeutend mit der Frage „Gilt die Goldbachsche Vermutung nicht?“

Das Halteproblem

Ein klassisches unentscheidbares Problem ist das Halteproblem:

Das Halteproblem besteht in der folgenden Frage:
Gegeben eine TM \mathcal{M} und ein Wort w ,
wird \mathcal{M} für die Eingabe w jemals anhalten?

Wir können das Halteproblem formal als Entscheidungsproblem ausdrücken, wenn wir \mathcal{M} und w kodieren:

Das Halteproblem ist das Wortproblem für die Sprache
 $P_{\text{Halt}} = \{\text{enc}(\mathcal{M})\#\text{enc}(w) \mid \mathcal{M} \text{ hält bei Eingabe } w\}$,
wobei $\text{enc}(\mathcal{M})$ und $\text{enc}(w)$ geeignete Kodierungen von \mathcal{M} und w sind, so dass $\#$ als Trennwort verwendet werden kann.

Anmerkung: Falsch kodierte Eingaben werden hier auch abgelehnt.

Abschweifung

Ist die Goldbachsche Vermutung entscheidbar?

Beweis durch „Diagonalisierung“

Satz: Das Halteproblem P_{Halt} ist unentscheidbar.

Beweis: Per Widerspruch: Wir nehmen an, dass es einen Entscheider \mathcal{H} für das Halteproblem gibt.

Dann kann man eine TM \mathcal{D} konstruieren, die folgendes tut:

- (1) Prüfe, ob die Eingabe eine TM-Kodierung $\text{enc}(\mathcal{M})$ ist
- (2) Simuliere \mathcal{H} auf der Eingabe $\text{enc}(\mathcal{M})\#\#\text{enc}(\text{enc}(\mathcal{M}))$, d.h. prüfe, ob \mathcal{M} auf $\text{enc}(\mathcal{M})$ hält
- (3) Falls ja, dann gehe in eine Endlosschleife; falls nein, dann halte und akzeptiere

Akzeptiert \mathcal{D} die Eingabe $\text{enc}(\mathcal{D})$?

\mathcal{D} hält und akzeptiert genau dann wenn \mathcal{D} nicht hält

Widerspruch. □

Turing-Reduktionen

Unser Beweis konstruiert den Algorithmus für ein Problem (Busy Beaver) durch Aufruf von Subroutinen für ein anderes (Halteproblem)

Diese Idee lässt sich verallgemeinern:

Ein Problem P ist **Turing-reduzierbar** auf ein Problem Q (in Symbolen: $P \leq_T Q$), wenn man P mit einem Programm lösen kann, welches ein Programm für Q als Unterprogramm aufrufen darf.

Anmerkung: Das ist etwas informell. Eine ganz formelle Definition verwendet den Begriff des **Orakels** für Turingmaschinen.

Beispiel: Unser Beweis basiert auf einer Turing-Reduktion der Berechnung der Busy-Beaver-Funktion auf das Halteproblem.

Beweis durch Reduktion

Satz: Das Halteproblem P_{Halt} ist unentscheidbar.

Beweis: Nehmen wir an, das Halteproblem wäre entscheidbar.

Ein Algorithmus:

- Eingabe: (binärkodierte) natürliche Zahl k
- Iteriere über alle Turingmaschinen \mathcal{M} mit k Zuständen über dem Arbeitsalphabet $\{x, \sqcup\}$:
 - Entscheide ob \mathcal{M} bei leerer Eingabe ϵ hält (möglich, wenn das Halteproblem entscheidbar ist)
 - Falls ja, dann simuliere \mathcal{M} auf der leeren Eingabe und zähle nach der Terminierung von \mathcal{M} die x auf dem Band (möglich, da es universelle Turingmaschinen gibt)
- Ausgabe: die maximale Zahl der geschriebenen x .

Dieser Algorithmus würde die Busy-Beaver-Funktion $\Sigma : \mathbb{N} \rightarrow \mathbb{N}$ berechnen.

Wir wissen, dass das unmöglich ist – Widerspruch. □

Turing-Reduktionen: Beispiel

Beispiel: Das Nicht-Halteproblem \bar{P}_{Halt} ist definiert als

$$\bar{P}_{\text{Halt}} = \{\text{enc}(\mathcal{M})\#\#\text{enc}(w) \mid \mathcal{M} \text{ hält bei nicht Eingabe } w\}$$

\bar{P}_{Halt} ist Turing-reduzierbar auf P_{Halt} : (1) Prüfe Eingabeformat, (2) entscheide Halteproblem, (3) invertiere Ergebnis.

Analog kann auch P_{Halt} auf \bar{P}_{Halt} Turing-reduziert werden.

Daraus ergibt sich:

Satz: Das Nicht-Halteproblem \bar{P}_{Halt} ist unentscheidbar.

ε-Halten

Sonderfälle des Halteproblems sind in der Regel nicht einfacher:

Das ε-Halteproblem besteht in der folgenden Frage:
Gegeben eine TM \mathcal{M} ,
wird \mathcal{M} für die leere Eingabe ϵ jemals anhalten?

Satz: Das ε-Halteproblem ist unentscheidbar.

Beweis: Angenommen das Problem wäre entscheidbar.

Ein Algorithmus:

- Eingabe: Eine Turingmaschine \mathcal{M} und ein Wort w .
- Konstruiere eine TM \mathcal{M}_w , die zwei Schritte ausführt:
 - (1) Lösche das Eingabeband und fülle es mit dem Wort w
 - (2) Verarbeite diese Eingabe wie \mathcal{M}
- Entscheide das ε-Halteproblem für \mathcal{M}_w .
- Ausgabe: Ergebnis des ε-Halteproblems

Dies würde das Halteproblem entscheiden – Widerspruch. □

Many-One-Reduktionen

Idee: Beim ε-Halteproblem verwenden wir das Halteproblem nicht als Subroutine eines komplexen Programms, sondern wir formen unser Problem in ein Halteproblem um

Eine berechenbare totale Funktion $f : \Sigma^* \rightarrow \Sigma^*$ ist eine **Many-One-Reduktion** von einer Sprache \mathbf{P} auf eine Sprache \mathbf{Q} (in Symbolen: $\mathbf{P} \leq_m \mathbf{Q}$), wenn für alle Wörter $w \in \Sigma^*$ gilt:

$$w \in \mathbf{P} \quad \text{genau dann wenn} \quad f(w) \in \mathbf{Q}$$

Beispiel: Die folgende Funktion definiert eine Many-One-Reduktion vom Halteproblem auf das ε-Halteproblem:

$$f(v) = \begin{cases} \text{enc}(\mathcal{M}_w) & \text{falls } v = \text{enc}(\mathcal{M})\#\#\text{enc}(w) \text{ für eine TM } \mathcal{M} \\ \# & \text{falls die Eingabe nicht korrekt kodiert ist} \end{cases}$$

Dabei ist \mathcal{M}_w die TM aus dem Beweis.

Beweistechniken im Vergleich

Wir haben zwei ähnliche Unentscheidbarkeitsbeweise gesehen:

Halteproblem

- Reduktion der Busy-Beaver-Funktion
- Algorithmus ruft Subroutine für Halteproblem exponentiell oft auf
- Ausgabe wird durch weitere TM-Simulationen berechnet

~> Turing-Reduktion!

ε-Halteproblem

- Reduktion des Halteproblems
- Algorithmus ruft Subroutine für ε-Halteproblem immer genau einmal auf
- Ausgabe ist das Ergebnis der ε-Halteproblem-Routine

~> Turing-Reduktion?

Entscheidbarkeit durch Reduktion

Das folgende Resultat drückt die wesentliche Idee hinter Reduktionen aus:

Satz: Wenn $\mathbf{P} \leq_m \mathbf{Q}$ und \mathbf{Q} entscheidbar ist, dann ist auch \mathbf{P} entscheidbar.

Beweis: Die Reduktion liefert einen Entscheidungsalgorithmus. □

Eigentlich benutzen wir bisher vor allem die Umkehrung:

Satz: Wenn $\mathbf{P} \leq_m \mathbf{Q}$ und \mathbf{P} unentscheidbar ist, dann ist auch \mathbf{Q} unentscheidbar.

Many-One vs. Turing

Many-One-Reduktionen sind schwächer als Turing-Reduktionen:

Satz: Jede Many-One-Reduktion kann als Turing-Reduktion ausgedrückt werden.

Beweis: Die Turing-Reduktion ergibt sich, wenn man die (berechenbare) Many-One-Reduktionsfunktion als Teil einer TM implementiert. \square

Satz: Es gibt Probleme P und Q , für die $P \leq_T Q$ gilt, aber nicht $P \leq_m Q$.

Beweis: Wir haben bereits gesehen, dass $P_{\text{Halt}} \leq_T \bar{P}_{\text{Halt}}$. Aber es gilt nicht $P_{\text{Halt}} \leq_m \bar{P}_{\text{Halt}}$ – wir werden in der nächsten Vorlesung sehen, warum nicht. \square

Zusammenfassung und Ausblick

LOOP-Programme können wirklich fast alle praktisch relevanten Probleme lösen

Durch Reduktionen können wir aus der (Un)Lösbarkeit eines Problems die (Un)Lösbarkeit eines anderen ableiten

Turing-Reduktionen $P \leq_T Q$ verwenden die Lösung von Q als Subroutine in einem Algorithmus für P

Many-One-Reduktionen $P \leq_m Q$ Formen eine Problemstellung für P in eine Problemstellung für Q um

Was erwartet uns als nächstes?

- Mehr zu Semi-Entscheidbarkeit
- Ein unentscheidbares Problem von Emil Post ...
- ... und unendlich viele von Gordon Rice