Theoretische Informatik

Teil 7
Berechenbarkeit

Frühlimgssemester 2019

L. Di Caro
D. Flumini
O. Stern

Überblick Berechenbarkeit

Teil 1:

- Church-Turing-These und der Berechenbarkeitsbegriff
- Ansätze zur Formalisierung des Berechenbarkeitsbegriffes
  - Rekursive und primitiv rekursive Funktionen
  - LOOP und WHILE berechenbare Funktionen.
  - Turing-berechenbare Funktionen
- Die verschiedenen Ansätze im Vergleich
  - LOOP Berechenbarkeit und primitiv rekursive Funktionen
  - Turingvollständigkeit
  - Ackermannfunktion
  - LOOP-Interpreter

Überblick Berechenbarkeit

Teil 2:

- Entscheidbarkeit
  - Abschlusseigenschaften entscheidbarer Mengen
  - Reduktionen von entscheidbaren Mengen
- Semi-Entscheidbarkeit
  - Charakterisierung von Entscheidbarkeit durch semi-Entscheidbarkeit
  - Rekursiv aufzählbare Mengen
  - Reduktionen von semi-entscheidbaren Mengen
  - Das Halteproblem
- Satz von Rice

Church-Turing-These

Begrifflichkeiten: Die zentralen Begriffe der Church-Turing-These sind:

- Als **intuitiv berechenbare Funktionen** bezeichnen wir alle Funktionen, die algorithmisch (d.h. durch irgendein mechanisches Verfahren) berechnet werden können.
- **Turing-berechenbare Funktionen** sind Funktionen, die von einer Turing-Maschine berechnet werden können.

Anmerkung:

Da man jedes Berechnungsproblem auch mit Funktionen formulieren kann, beinhaltet die Church-Turing-These auch die Aussage, dass **jedes algorithmisch lösbare Berechnungsproblem** von einer Turing-Maschine gelöst werden kann.

Church-Turing-These

Turing 1948:

LCMs[1] can do anything that could be described as "rule of thumb" or "purely mechanical".

Church 1936:

A function of positive integers is effectively calculable only if recursive.

Church-Turing-These:

Die Klasse der Turing-berechenbaren Funktionen stimmt mit der Klasse der intuitiv berechenbaren Funktionen überein.

[1] logical computing machines: Turings Name für Turing-Maschinen.

Church-Turing-These

Gandy 1980 (Gandy's Thesis M):

> *Whatever can be calculated by a machine (working on finite data in accordance with a finite program of instructions) is Turing-machine-computable.*

Gandy's These M:
Alles, was jemals mit einer (endlichen) Maschine/physikalischen Apparatur berechnet werden kann, ist bereits von einer Turing-Maschine berechenbar.

Anmerkung: Die These M ist eine Verschärfung der ursprünglichen These.

Church-Turing-These

Informeller Charakter:

- Die Church-Turing-These nimmt Bezug auf das informelle Konzept von **intuitiv berechenbaren Funktionen**. Die Church-Turing-These ist daher **keine mathematische Aussage** und als solche **nicht (formal) beweisbar**.
- Die Church-Turing-These muss so verstanden werden, dass unsere **Anschauung** von berechenbaren Funktionen und berechenbaren Prozessen durch das Modell der Turing-Maschinen adäquat und vollständig **formalisiert** wird.

Argumente für die Church-Turing-These:

- Die **Robustheit** der Klasse der Turing-berechenbaren Funktionen.
  - Einschränkungen und Erweiterungen von Turing-Maschinen ändern nichts an der Klasse der berechenbaren Funktionen.
  - Viele, teilweise von Turing-Maschinen sehr unterschiedliche Formalismen resultieren in derselben Klasse von berechenbaren Funktionen.

    Einige Beispiele:
    - WHILE-Berechenbarkeit
    - μ-rekursive Funktionen
    - Register-Maschinen
    - λ-Kalküle
    - ...

Church-Turing-These

Argumente für die Church-Turing-These:

- Es wurde bis zum heutigen Tag **kein Gegenbeispiel** gefunden.

  Ein Gegenbeispiel wäre eine Funktion, die einerseits im intuitiven Sinn offensichtlich berechenbar ist und andererseits nicht von einer Turing-Maschine berechnet werden kann.

Computer und Turing-Maschinen

Computer und Turing-Maschinen sind “äquivalent”:

- Jeder Computer kann von einer Turing-Maschine simuliert werden.
- Jede Turing-Maschine kann von einem Computer simuliert werden[2].

Argumentation:

- Turing-Maschinen können die Funktionalität jedes $WHILE$-Programmes[3] simulieren, daher kann das Verhalten jedes Computers von einer Turing-Maschine simuliert werden.
- Offensichtlich kann man Turing-Maschinen-Simulatoren programmieren (vgl. Aufgabe).

[2] Unbegrenzter Speicher-Nachschub angenommen.

[3] Mehr dazu später.

Turing-berechenbare Funktionen

Definition

Es sei $T = (Q, \Sigma, \Gamma, \ldots)$ eine Turing-Maschine. Wir können T wie folgt als (partielle) Funktion auffassen:

$$T : \Sigma^* \to \Gamma^*$$

wobei

$$T(w) = \begin{cases} u & \text{falls } T \text{ auf } w \in \Sigma^* \text{ angesetzt, nach endlich} \\ & \text{vielen Schritten mit } u \text{ auf dem Band anhält,} \\ \uparrow & \text{falls } T \text{ bei Input } w \in \Sigma^* \text{ nicht anhält.} \end{cases}$$

Anmerkung: Wir schreiben $F(x) = \uparrow$, wenn die Funktion F für den Input x nicht definiert ist, d.h. keinen Wert annimmt.

Turing-berechenbare Funktionen

Definition

Eine Funktion $F : \Sigma^* \to \Gamma^*$ heisst **Turing-berechenbar**, wenn es eine Turing-Maschine T gibt so, dass für alle $w \in \Sigma^*$

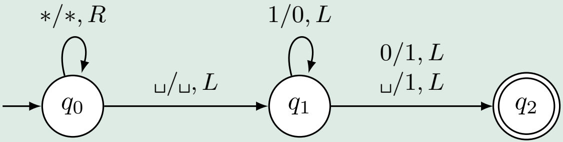
$$T(w) = F(w)$$

gilt. Eine Turing-berechenbare Funktion ist demnach eine Funktion, die von einer Turing-Maschine berechnet werden kann.

Anmerkung: Um über Turing-berechenbare Funktionen von $\mathbb{N}$ nach $\mathbb{N}$ zu sprechen, identifizieren wir jede natürliche Zahl n mit ihrer Binärdarstellung $bin(n) \in \{0, 1\}^*$. Tupel $(n_1, \ldots, n_k)$ von natürlichen Zahlen stellen wir als $bin(n_1)\# \ldots \#bin(n_k) \in \{0, 1, \#\}^*$ dar.

Beispiel (Die Nachfolgerfunktion)

Ist T durch das Übergangsdiagramm



gegeben, dann können wir T als Funktion

$$T : \mathbb{N} \to \mathbb{N} \text{ mit } T(n) = n + 1.$$

interpretieren. Die Funktion $f(n) = n + 1$ ist somit Turing-berechenbar.

Turing-berechenbare Funktionen

Definition (Die Modulo Funktion)

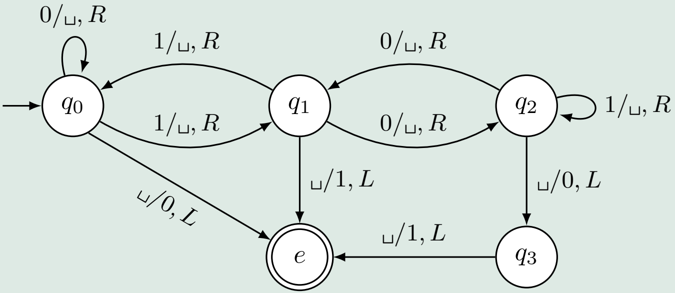
Die Funktion $\text{mod}_k : \mathbb{N} \to \mathbb{N}$ ist durch die Zuordnung

$$\text{mod}_k(n) = \begin{cases} 0 & \text{falls } n \text{ durch } k \text{ teilbar,} \\ 1 & \text{falls } n \text{ bei Division durch } k \text{ den Rest } 1 \text{ lässt,} \\ \vdots & \vdots \\ k-1 & \text{falls } n \text{ bei Division durch } k \text{ den Rest } k-1 \text{ lässt.} \end{cases}$$

gegeben.

Beispiel (mod_3 Funktion)

Ist T durch das Übergangsdiagramm



gegeben, wird die Funktion mod_3 von T berechnet.

LOOP-Programme

Zeichenvorrat: LOOP-Programme bestehen aus folgenden syntaktischen Grundelementen:

- Variablen: `x0, x1, x2, ...`
- Konstanten: `0, 1, 2, 3, 4, ...`
- Trennzeichen: ;
- Zuweisung: =
- Operationszeichen: + und -
- Schlüsselwörter: `LOOP, DO, END`

Anmerkung:

Nach Ablauf eines `LOOP`-Programms steht der Wert der Berechnung in der Variablen `x0`.

LOOP-Programme

Definition (Syntax von LOOP-Programmen)

LOOP-Programme sind wie folgt gegeben:

- **Zuweisungen:** `x = a + b` und `x = a - b`, wobei x für eine Variable und a, b für Variablen oder Konstanten stehen.
- **Sequenzen:** Sind P und Q LOOP-Programme, dann ist auch `P;Q` ein LOOP-Programm.
- **Schleifen:** Ist P ein LOOP-Programm, dann ist für jede Variable x auch `LOOP x DO P END` ein LOOP-Programm.

Anmerkung:

Zur besseren Lesbarkeit strukturieren wir LOOP-Programme mit Zeilenumbrüchen.

LOOP-Programme

Definition (Semantik von LOOP-Programmen)

Sei P ein LOOP-Programm mit mindestens den Variablen $x_0, x_1, \ldots, x_k$ für ein $k \in \mathbb{N}$. Die **von P berechnete k-stellige Funktion** $P_k : \mathbb{N}^k \to \mathbb{N}$ ist gegeben durch

$$P_k(n_1, \ldots, n_k) = \text{Wert der Variablen } x_0 \text{ nach Ablauf von } P, \text{ wenn mit den Werten } n_1, \ldots, n_k \text{ in den Variablen } x_1, \ldots, x_k \text{ gestartet wurde.}$$

Konventionen:

- x_0 und zusätzliche Variablen werden mit dem Wert 0 initialisiert.
- Variablen können als Werte natürliche Zahlen $(0, 1, 2, \ldots)$ halten.
- **Die Subtraktion eines grösseren von einem kleineren Wert ergibt immer 0.**

LOOP-Programme

Beispiel (Bedingte Ausführung)

Das LOOP-Programm:

```
xi = 1 - xi;
xi = 1 - xi;
LOOP xi DO
    P
END
```

realisiert für ein LOOP-Programm `P` die bedingte Ausführung:

```
IF xi > 0 THEN P
```

Warnung: Die bedingte Ausführung hat einen Nebeneffekt auf die Variable `xi`. Das ist insbesondere dann problematisch, wenn sie in einem Programm verwendet wird, das seinerseits auf die Variable `xi` zugreift.

LOOP-Programme

Beispiel (Verzweigung)

Das LOOP-Programm:

```
xi = 1 - xi;
LOOP xi DO
    Q
END;
xi = 1 - xi;
LOOP xi DO
    P
END
```

realisiert für LOOP-Programme `P` und `Q` die Verzweigung:

```
IF xi > 0 THEN P ELSE Q
```

LOOP-Programme

Beispiele (Addition und Multiplikation)

Die Addition und die Multiplikation von natürlichen Zahlen sind LOOP-berechenbar.

- Das LOOP-Programm `x0 = x1 + x2` berechnet die Addition $Add(x, y) = x + y$.
- Das LOOP-Programm:

```
LOOP x1 DO
    x0 = x0 + x2
END
```

  berechnet die Multiplikation $Mult(x, y) = x \cdot y$.

Anmerkung: Zur Erinnerung, x0 wird mit dem Wert 0 initialisiert.

LOOP-Programme

Beispiel (Modulo Funktion)

Das LOOP-Programm (mit `IF−THEN`-Makro):

```
x0 = x1 + 0;
LOOP x1 DO              \\ Immer x1 Iterationen!
    x1 = x1 - x2;
    IF x1 > 0 THEN x0 = x1 + 0
END;
x1 = x2 - 1;
x1 = x0 - x1;
LOOP x1 DO
    x0 = 0 + 0
END
```

berechnet die Funktion $Mod(x, y) = \text{mod}_y(x)$.

WHILE-Programme

Definition (WHILE-Programme)

Erweitert man die Sprache LOOP um den zusätzliche syntaktischen Baustein `WHILE xi > 0 DO ... END` für alle Variablen `xi`, dann erhält man die Menge aller **WHILE-Programme**.

Anmerkungen:

- Jedes LOOP-Programm ist auch ein WHILE-Programm.
- WHILE-Programme terminieren nicht immer (können unter Umständen unendlich lang laufen).
- Die Semantik von WHILE-Programmen ist analog zur Semantik von LOOP-Programmen gegeben.

WHILE-Programme

Theorem (Turing-Vollständigkeit)

Für jede natürliche Zahl $k \in \mathbb{N}$ und jede (partielle) Funktion $f : \mathbb{N}^k \to \mathbb{N}$, sind folgende Aussagen äquivalent:

- *Es gibt eine Turing-Maschine T, die f berechnet. D.h., f ist Turing-berechenbar.*
- *Es gibt ein WHILE-Programm P, das f berechnet. D.h., f ist WHILE-berechenbar.*

Anmerkungen:

- Man sagt in diesem Zusammenhang, dass das Berechnungsmodell der WHILE-berechenbaren Funktionen Turing-vollständig sei.
- Ein weiteres Turing-vollständiges Berechnungsmodell stellen die GOTO- Programme dar, sie enthalten anstelle der WHILE-Schleifen bedingte Sprungbefehle.

Primitiv rekursive Funktionen (Backup)

Induktive Definition:Die Klasse der **primitiv rekursiven Funktionen** besteht aus Funktionen $F : \mathbb{N}^k \to \mathbb{N}$ und wird nach folgendem "zweistufigen" Schema[4] definiert:

- Man beginnt mit einer Menge von **Grundfunktionen**, die ad hoc als primitiv rekursiv deklariert werden.
- Man schliesst die Menge der primitiv rekursiven Funktionen unter gewissen **Operationen** ab, d.h. man gibt Regeln von der Form: Wenn diese und jene Funktion primitiv rekursiv sind, dann auch bestimmte weitere Funktionen.

[4] Eine Definition nach diesem Muster wird üblicherweise als *induktive Definition* bezeichnet.

Primitiv rekursive Funktionen (Backup)

Definition (Grundfunktionen)

Die primitiv rekursiven **Grundfunktionen** sind:

- Für jedes $n \in \mathbb{N}$ und jede Konstante $k \in \mathbb{N}$ die n-stellige **konstante Funktion**:

$$c_k^n : \mathbb{N}^n \to \mathbb{N} \text{ mit } c_k^n(x_1, \dots, x_n) = k.$$

- Die **Nachfolgerfunktion**:

$$\eta : \mathbb{N} \to \mathbb{N} \quad \text{mit} \quad \eta(x) = x + 1.$$

- Für jedes $n \in \mathbb{N}$ und jedes $1 \leq k \leq n$ die n-stellige **Projektion** auf die k-te Komponente:

$$\pi_k^n : \mathbb{N}^n \to \mathbb{N} \quad \text{mit} \quad \pi_k^n(x_1, \dots, x_k, \dots, x_n) = x_k$$

Primitiv rekursive Funktionen (Backup)

Beispiele (Grundfunktionen)

Einige Grundfunktionen:

- $c_5^4 : \mathbb{N}^4 \to \mathbb{N}$ mit $c_5^4(x, y, z, u) = 5$.
- $\pi_1^3 : \mathbb{N}^3 \to \mathbb{N}$ mit $\pi_1^3(x, y, z) = x$.
- $\pi_1^1 : \mathbb{N} \to \mathbb{N}$ mit $\pi_1^1(x) = x$.
- $\pi_5^5 : \mathbb{N}^5 \to \mathbb{N}$ mit $\pi_5^5(x, y, z, u, v) = v$.

Anmerkung: Die Addition

$$Add : \mathbb{N}^2 \to \mathbb{N} \quad \text{mit} \quad Add(x, y) = x + y$$

ist keine Grundfunktion. Wenn wir aber die Grundfunktionen geeignet kombinieren, dann sehen wir, dass die Additionsfunktion sowie viele weitere Funktionen primitiv rekursiv sind.

Primitiv rekursive Funktionen (Backup)

Definition (Abschluss durch **Einsetzung**)

Sind $f, g_1, g_2, \ldots, g_k$ primitiv rekursive Funktionen mit $g_i : \mathbb{N}^n \to \mathbb{N}$ und $f : \mathbb{N}^k \to \mathbb{N}$, dann ist auch die Funktion

$$h : \mathbb{N}^n \to \mathbb{N} \quad \text{mit} \quad h(\vec{x}) = f(g_1(\vec{x}), \ldots, g_k(\vec{x})),$$

wobei $\vec{x}$ für $x_1, \ldots, x_n$ steht, primitiv rekursiv.

Primitiv rekursive Funktionen (Backup)

Beispiel (Einsetzung)

Da die Nachfolgerfunktion primitiv rekursiv ist, erhalten wir durch Einsetzung, dass die Funktion

$$f : \mathbb{N} \to \mathbb{N} \quad \text{mit} \quad f(x) = \eta(\eta(x)) = x + 2$$

primitiv rekursiv ist. Durch mehrmaliges Einsetzen erhalten wir für jede Konstante $c \in \mathbb{N}$, dass die Funktion $g(x) = x + c$ primitiv rekursiv ist.

Primitiv rekursive Funktionen (Backup)

Definition (Abschluss durch primitive Rekursion)

Sind $h : \mathbb{N}^n \to \mathbb{N}$ und $g : \mathbb{N}^{n+2} \to \mathbb{N}$ primitiv rekursive Funktionen, dann ist auch die (eindeutig bestimmte) Funktion $f : \mathbb{N}^{n+1} \to \mathbb{N}$ mit

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

wobei $\vec{x}$ für $x_1, \ldots, x_n$ steht, primitiv rekursiv.

Primitiv rekursive Funktionen (Backup)

Beispiel (Addition)

Die Additionsfunktion erfüllt folgende Gleichungen

$$\begin{aligned} Add(0, y) &= y \\ Add(x+1, y) &= Add(x, y) + 1 \end{aligned}$$

anders notiert heisst das

$$\begin{aligned} Add(0, y) &= \pi_1^1(y) \\ Add(x+1, y) &= \eta(\pi_1^3(Add(x, y), x, y)). \end{aligned}$$

Weil die Funktion $g(x, y, z) = \eta(\pi_1^3(x, y, z))$ durch Einsetzung primitiv rekursiv ist, ist die Addition, durch primitive Rekursion, primitiv rekursiv.

Primitiv rekursive Funktionen (Backup)

Beispiel (Multiplikation)

Die Multiplikationsfunktion erfüllt folgende Gleichungen

$$
\begin{aligned}
Mult(0, y) &= 0 \\
Mult(x + 1, y) &= Add(Mult(x, y), y)
\end{aligned}
$$

anders notiert heisst das

$$
\begin{aligned}
Mult(0, y) &= c_0^1(y) \\
Mult(x + 1, y) &= Add\big(Mult(x, y), \pi_2^2(x, y)\big).
\end{aligned}
$$

Weil die Funktion $g(x, y, z) = Add(x, \pi_2^2(y, z))$ primitiv rekursiv ist, ist auch die Multiplikation primitiv rekursiv.

Primitiv rekursive Funktionen (Backup)

Theorem

Die LOOP-berechenbaren Funktionen sind genau die primitiv rekursiven Funktionen.

Beweisidee.

- Alle prim. rek. Grundfunktionen sind LOOP-berechenbar. Die Klasse der LOOP-berechenbaren Funktionen ist unter Einsetzung und primitiver Rekursion abgeschlossen.
- Mit primitiver Rekursion kann eine LOOP-Schleife simuliert werden (vgl. Übungsaufgabe).

□

Ackermannfunktion

Definition

Die **Ackermannfunktion** $a : \mathbb{N}^2 \to \mathbb{N}$ ist durch die Gleichungen

$$\begin{aligned} a(0,m) &= m+1 \\ a(n+1,0) &= a(n,1) \\ a(n+1,m+1) &= a(n,a(n+1,m)) \end{aligned}$$

gegeben.

Anmerkung:

- Die Ackermannfunktion ist total (überall definiert).
- Die Ackermannfunktion ist (Turing-) berechenbar.

Ackermannfunktion

Beispiel

Wir berechnen den Wert $a(2,1)$:

$$
\begin{aligned}
a(2,1) &= a(1,a(2,0)) = a(1,a(1,1)) = a(1,a(0,a(1,0))) \\
&= a(1,a(0,a(0,1))) = a(1,a(0,2)) = a(1,3) = a(0,a(1,2)) \\
&= a(0,a(0,a(1,1))) = a(0,a(0,a(0,a(1,0)))) \\
&= a(0,a(0,a(0,a(0,1)))) = a(0,a(0,a(0,2))) \\
&= a(0,a(0,3)) = a(0,4) = 5
\end{aligned}
$$

Ackermannfunktion

Beispiel

Einige Funktionswerte der Ackermannfunktion.

a(x,y)	0	1	2	3
0	1	2	3	4
1	2	3	4	5
2	3	5	7	9
3	5	13	29	61
4	13	65533	$2^{65536}-3$	riesig

Anmerkung:

- Die Ackermannfunktion “wächst schneller” als jede primitiv rekursive Funktion.
- Die Ackermannfunktion ist nicht LOOP-berechenbar/primitiv rekursiv.

LOOP-Interpreter (Backup)

Wir gehen davon aus, dass sich jedes LOOP-Programm P als natürliche Zahl $\langle P \rangle$ darstellen lässt[5].

Definition (LOOP-Interpreter)

Ein **LOOP-Interpreter** I ist eine Funktion $I : \mathbb{N}^2 \to \mathbb{N}$, so dass

$$I(\langle P \rangle, x) = P_1(x)$$

für jedes LOOP-Programm P und jede natürliche Zahl x gilt.

Anmerkung: Ein LOOP-Interpreter ist eine Funktion, die ihr erstes Argument als LOOP-Programm interpretiert und die Funktionalität dieses Programmes dann auf das zweite Argument anwendet.

[5] Man stelle sich unter $\langle P \rangle$ den Bytecode von P vor.

LOOP-Interpreter (Backup)

Satz

Es gibt kein LOOP-Programm P, so dass P_2 ein LOOP-Interpreter ist.

Beweis durch Widerspruch.

Gäbe es ein LOOP-Programm P, so dass die Funktion P_2 ein LOOP-Interpreter ist, dann gäbe es auch ein LOOP-Programm Q mit

$$Q_1(x) = P_2(x, x) + 1.$$

Wenden wir nun Q auf seinen eigenen Bytecode an, dann erhalten wir den gewünschten Widerspruch:

$$\begin{aligned} Q_1(\langle Q \rangle) &= P_2(\langle Q \rangle, \langle Q \rangle) + 1 \\ &= Q_1(\langle Q \rangle) + 1 \end{aligned} \qquad \square$$

Überblick

Anmerkungen:

- Wie wir anhand der Ackermannfunktion und des LOOP-Interpreters gesehen haben, gibt es totale Turing-berechenbare Funktionen, die nicht primitiv rekursiv und somit auch nicht LOOP-berechenbar sind.
- Somit gilt: Die Berechnungsmodelle der LOOP-Programme und der primitiv rekursiven Funktionen sind **nicht** Turing-vollständig.

(Semi-) Entscheidbarkeit

Definition (Entscheidbarkeit)

Eine Sprache $A \subset \Sigma^*$ heisst **entscheidbar**, wenn eine Turingmaschine T existiert, die das Entscheidungsproblem (Σ, A) löst.

Anmerkungen:

- Ist eine Sprache $A \subset \Sigma^*$ entscheidbar, dann gibt es eine Turingmaschine T, die sich wie folgt verhält:
  - Wenn T mit Bandinhalt $x \in A$ gestartet wird, dann hält T nach endlich vielen Schritten mit Bandinhalt "1"(Ja) an.
  - Wenn T mit Bandinhalt $x \in \Sigma^* \setminus A$ gestartet wird, dann hält T nach endlich vielen Schritten mit Bandinhalt "0" (Nein) an.
- Insbesondere muss die Turingmaschine T bei jeder Eingabe $x \in \Sigma^*$ nach endlich vielen Schritten anhalten.

(Semi-) Entscheidbarkeit

Definition (Semi-Entscheidbarkeit)

Eine Sprache $A \subset \Sigma^*$ heisst **semi-entscheidbar**, wenn eine Turingmaschine T existiert, die sich wie folgt verhält:

- Wenn T mit Bandinhalt $x \in A$ gestartet wird, dann hält T nach endlich vielen Schritten mit Bandinhalt "1"(Ja) an.
- Wenn T mit Bandinhalt $x \in \Sigma^* \setminus A$ gestartet wird, dann hält T nie an.

Anmerkung:

Informell kann man sagen, dass zu einer semi-entscheidbaren Sprache A eine Turingmaschine existiert, die zum Entscheidungsproblem (Σ, A) nur die positiven ("Ja") Antworten liefert und anstelle von negativen Antworten ("Nein") gar keine Antwort zurückgibt.

(Semi-) Entscheidbarkeit

Konvention: Wie bereits erwähnt werden natürliche Zahlen mit ihrer Binärdarstellung identifiziert.

Eine Teilmenge $X \subset \mathbb{N}$ betrachten wir also genau dann als (semi-) entscheidbar, wenn die Sprache

$$\{bin(x) \mid x \in X\} \subset \{0,1\}^*$$

(semi-) entscheidbar ist.

(Semi-) Entscheidbarkeit

Folgerungen in Bezug auf Turing-vollständigkeit:

- Eine Sprache $A \subset \Sigma^*$ ist genau dann entscheidbar, wenn das Entscheidungsproblem (Σ, A) mit einem WHILE-Programm gelöst werden kann.

  Ein solches WHILE-Programm nennen wir ein **Entscheidungsverfahren** für A.

- Eine Sprache $A \subset \Sigma^*$ ist genau dann semi-entscheidbar, wenn ein WHILE-Programm existiert, das bei Eingabe von einem zu A gehörenden Wort stets terminiert und "Ja" zurückgibt und bei Eingabe von Wörtern, die nicht zu A gehören, nicht terminiert.

  Ein solches WHILE-Programm nennen wir ein **semi-Entscheidungsverfahren** für A.

(Semi-) Entscheidbarkeit

Beispiele

- Die Menge aller geraden natürlichen Zahlen ist entscheidbar, da die Funktion, $F(x) = \text{mod}_2(x+1)$, berechenbar ist.
- Die Menge aller Primzahlen ist entscheidbar, da folgender Pseudocode das entsprechende Entscheidungsproblem löst.

```
INPUT(n)
FOR i = 2 to n-1 DO
    IF Mod(n,i) = 0 THEN return 0
END
return 1
```

Anmerkung: Der Befehl “return” beendet die Ausführung der Schleife.

Beispiel (Aufgabe)

Begründen Sie: Jede entscheidbare Sprache ist auch semi-entscheidbar.

Lösung: Ist A eine entscheidbare Sprache, dann kann das Entscheidungsproblem (Σ, A) mit einem WHILE-Programm gelöst werden. Folglich können wir Abfragen von der Form $w \in A$ in einem weiteren Entscheidungsverfahren verwenden. Der folgende Pseudocode liefert das gewünschte semi-Entscheidungsverfahren:

```
INPUT(w)
WHILE true DO
   IF w in A THEN return 1
END
```

Anmerkung: “`w in A`” steht für die Abfrage $w \in A$.

(Semi-) Entscheidbarkeit

Satz

Eine Sprache $A \subset \Sigma^*$ ist genau dann entscheidbar, wenn sowohl A als auch $\overline{A}$ semi-entscheidbar ist.

Anmerkung: Der Ausdruck $\overline{A}$ steht für das Komplement von A in Σ^*:

$$\overline{A} = \Sigma^* \setminus A = \{w \in \Sigma^* \mid w \notin A\}$$

Beweis ($\Rightarrow$).

Wenn wir ein Entscheidungsverfahren für die Menge A haben, dann erhalten wir durch Verneinung der Ausgabe ein Entscheidungsverfahren für $\overline{A}$, somit ist mit jeder entscheidbaren Sprache A auch das Komplement $\overline{A}$ entscheidbar und somit auch semi-entscheidbar.

(Semi-) Entscheidbarkeit

Fortsetzung Beweis ($\Leftarrow$).

Wir müssen zeigen, dass für jede semi-entscheidbare Sprache mit semi-entscheidbarem Komplement ein Entscheidungsverfahren existiert. Dies erreichen wir durch folgenden Algorithmus (Pseudocode):

```
INPUT(w)
FOR n = 1 to infinity DO
    IF A(w,n) THEN return 1
    IF B(w,n) THEN return 0
END
```

wobei die Abfragen `A(w,n)` und `B(w,n)` wie folgt zu interpretieren sind:

- `A(w,n)` $=$ Das semi-Entscheidungsverfahren von A terminiert nach n Schritten.
- `B(w,n)` $=$ Das semi-Entscheidungsverfahren von $\overline{A}$ terminiert nach n Schritten.

(Semi-) Entscheidbarkeit

Satz (Abschlusseigenschaften)

- *Ist* $A \subset \Sigma^*$ *eine entscheidbare Sprache, dann ist auch* $\overline{A}$ *eine entscheidbare Sprache.*
- *Sind* A, B *(semi-) entscheidbare Sprachen, dann sind auch* $A \cup B$ *und* $A \cap B$ *(semi-) entscheidbare Sprachen.*

Beweis.

Die erste Tatsache folgt sofort aus dem soeben bewiesenen Satz. Die zweite Behauptung ist als Übung zu beweisen. □

(Semi-) Entscheidbarkeit

Satz (Charakterisierungen)

Folgende Aussagen für $A \subset \Sigma^*$ sind äquivalent:

- A *ist rekursiv aufzählbar.*
- A *ist semi-entscheidbar*[6]*.*
- A *ist der Wertebereich einer totalen berechenbaren Funktion.*
- A *ist der Definitionsbereich einer berechenbaren Funktion.*

[6] Hält genau für alle $w \in A$

Reduktion

Wir betrachten zwei Entscheidungsprobleme:

Problem P_1

Gegeben: Eine natürliche Zahl x.
Gefragt: Ist x eine Primzahl?

Problem P_2

Gegeben: Ein Paar (x, y) von natürlichen Zahlen.
Gefragt: Ist x der kleinste Primfaktor von y?

Reduktion

Fragestellung: Können wir ein Lösungsverfahren vom Problem P_2 auch dazu verwenden das Problem P_1 zu lösen?

Ansatz: Für jede natürliche Zahl x gilt:

$$x \text{ erfüllt } P_1 \Leftrightarrow (x, x) \text{ erfüllt } P_2.$$

Die Frage ob x zu P_1 gehört, lässt sich also auf die Frage **reduzieren** ob das Paar (x, x) zu P_2 gehört.

Anmerkung:

Offenbar können wir jede Instanz des Problems P_1 zu einer (gleichwertigen) Instanz des Problems P_2 umformulieren. Solch eine Umformulierung nennt man eine **Reduktion** von P_1 auf P_2.

Reduktion

Definition

Eine Sprache $A \subset \Sigma^*$ heisst auf eine Sprache $B \subset \Gamma^*$ **reduzierbar**, wenn es eine totale, Turing-berechenbare Funktion $F : \Sigma^* \to \Gamma^*$ gibt, so dass für alle $w \in \Sigma^*$

$$w \in A \Longleftrightarrow F(w) \in B$$

gilt. Ist die Sprache A auf die Sprache B reduzierbar, dann schreiben wir $A \preceq B$.

Reduktion

Lemma (Transitivität)

Für beliebige Sprachen A, B und C gilt

$$A \preceq B \text{ und } B \preceq C \Rightarrow A \preceq C.$$

Beweis.

Dies folgt aus der Tatsache, dass die Komposition (Einsetzung) von totalen, Turing-berechenbaren Funktionen total, Turing-berechenbar ist. □

Reduktion

Lemma

Für beliebige Sprachen $A \subset \Sigma^*$ und $B \subset \Gamma^*$ gilt:

- *Ist B entscheidbar und $A \preceq B$, dann ist auch A entscheidbar.*
- *Ist B semi-entscheidbar und $A \preceq B$, dann ist auch A semi-entscheidbar.*

Beweis.

Wir gehen von einem Entscheidungsverfahren `P` für B und einer totalen berechenbaren Funktion $F : \Sigma^* \to \Gamma^*$ mit

$$w \in A \iff F(w) \in B$$

aus. Wir müssen auf dieser Grundlage ein Entscheidungsverfahren für A angeben.

Reduktion

Fortsetzung Beweis.

Da die Funktion F berechenbar ist, können wir sie in unserem Entscheidungsverfahren aufrufen (Pseudocode).

```
input(w)
  u = F(w)
return P(u)
```

Aus der Totalität von F folgt, dass dieses Programm das Entscheidungsproblem (Σ, A) löst. Somit ist A entscheidbar.

Der Beweis von der zweiten Behauptung geht analog. $\square$

Das Halteproblem

Codes[7] von Turingmaschinen:

- Wir ordnen jeder Turingmaschine einen Code aus $w \in \{0,1\}^*$ zu.
- Für jeden Code $w \in \{0,1\}^*$ sei T_w die Turing-Maschine mit Code w.
- Es sei M eine beliebige[8] aber feste Turing-Maschine. Für alle Wörter $w \in \{0,1\}^*$, die nicht Code einer Turing-Maschine sind, setzen wir $T_w = M$. Somit ist jedes Binärwort der Code einer Turing-Maschine.

Konvention:

Ist T eine Turing-Maschine mit Code w, dann schreiben wir für die von T berechnete Funktion auch F_w.

---

[7] Für eine konkrete Codierung, die alle oben erwähnten Eigenschaften besitzt, sei auf die separaten Folien auf OLAT zur universellen TM verwiesen.

[8] Z.B. $M = (\{q\}, \{0\}, \{0, \sqcup\}, \varnothing, q, \sqcup, \varnothing)$

Das Halteproblem

Das spezielle Halteproblem: Das spezielle Halteproblem, auch Selbstanwendungsproblem genannt, kann man wie folgt als Entscheidungsproblem formulieren:

Spezielles Halteproblem H_S

Gegeben: Der Code $w \in \{0,1\}^*$ einer Turing-Maschine T_w.

Gefragt: Hält die Turing-Maschine T_w an, wenn man sie auf ihren eigenen Code w (als Input) ansetzt?

Das Halteproblem

Das spezielle Halteproblem als Sprache formuliert.

Definition (Das spezielle Halteproblem)

Das **spezielle Halteproblem** ist die Sprache

$$H_S := \{w \in \{0,1\}^* \mid T_w \text{ angesetzt auf } w \text{ hält}\}.$$

Anmerkung:

Das spezielle Halteproblem H_S wird auch als **Selbstanwendungsproblem** bezeichnet.

Theorem (Unentscheidbarkeit von H_S)

Das spezielle Halteproblem ist nicht entscheidbar.

Widerspruchsbeweis.

Wir nehmen an, dass es eine Turing-Maschine T das Halteproblem H_S entscheidet. Wir konstruieren, ausgehend von T, eine neue Turing-Maschine P.

Die TM P:

$$\begin{tikzcd}
\boxed{\text{TM } T} \arrow[r] & \boxed{\text{Bandinhalt?}} \arrow[r, "0"] \arrow[loop below, "1"] & \boxed{\text{Band } := 1} \arrow[r] & \boxed{\text{Halt}}
\end{tikzcd}$$

Das Halteproblem

Fortsetzung Beweis.

Nun sei w der Code der Turing-Maschine P, i.e. $T_w = P$. Aus der Konstruktion von P erhalten wir

$$P \text{ angesetzt auf } w \text{ hält } \Leftrightarrow T(w) = 0.$$

Weil T das spezielle Halteproblem entscheidet, erhalten wir auch

$$\begin{aligned} T(w) = 0 &\Leftrightarrow T_w \text{ angesetzt auf } w \text{ hält nicht} \\ &\Leftrightarrow P \text{ angesetzt auf } w \text{ hält nicht} \end{aligned}$$

und damit den gesuchten Widerspruch. $\square$

Das Halteproblem

Anmerkungen:

- Lässt sich ein unentscheidbares Problem A auf ein Problem B reduzieren, i.e. gilt $A \preceq B$, dann ist auch das Problem B unentscheidbar.
- Ein gängiges Vorgehen zum Nachweis der Unentscheidbarkeit eines Problems, ist die, dass man das (spezielle) Halteproblem auf dieses Problem reduziert.

Das Halteproblem

Das allgemeine Halteproblem: Das allgemeine Halteproblem kann man wie folgt als Entscheidungsproblem formulieren:

Allgemeines Halteproblem H

Gegeben: Der Code $w \in \{0,1\}^*$ einer Turing-Maschine T_w und ein Input x.

Gefragt: Hält die Turing-Maschine T_w an, wenn man sie auf x ansetzt?

Das Halteproblem

Das allgemeine Halteproblem als Sprache formuliert.

Definition (Das allgemeine Halteproblem)

Das **allgemeine Halteproblem** ist die Sprache

$$H := \{w\#x \in \{0,1,\#\}^* \mid T_w \text{ angesetzt auf } x \text{ hält}\}.$$

Anmerkung:

Die Funktion des Zeichens $\#$ ist das Trennen des Inputstrings in zwei Inputs.

Das Halteproblem

Theorem (Unentscheidbarkeit von H)

Das allgemeine Halteproblem ist nicht entscheidbar.

Beweis.

Offensichtlich ist die Funktion $F : \{0,1\}^* \to \{0,1,\#\}^*$ die durch die Zuordnung

$$F(x) = x\#x$$

gegeben ist, eine Reduktion von H_S auf H. Die Unentscheidbarkeit von H folgt damit aus der Unentscheidbarkeit von H_S. $\square$

Unmittelbare Konsequenz: Man kann allgemein nicht algorithisch überprüfen (d. h. per Programm), ob ein gegebenes Programm für eine konkrete Eingabe terminiert.

Das Halteproblem

Das leere Halteproblem: Das Halteproblem auf leerem Band kann man wie folgt als Entscheidungsproblem formulieren:

Halteproblem auf leerem Band H_0

Gegeben: Der Code $w \in \{0,1\}^*$ einer Turing-Maschine T_w.

Gefragt: Hält die Turing-Maschine T_w an, wenn man sie auf das leere Band ansetzt?

Das Halteproblem

Das Halteproblem auf leerem Band als Sprache formuliert.

Definition (Das leere Halteproblem)

Das **leere Halteproblem** ist die Sprache

$$H_0 := \{w \in \{0,1\}^* \mid T_w \text{ angesetzt auf das leere Band hält}\}.$$

Das Halteproblem

Theorem (Unentscheidbarkeit von H_0)

Das Halteproblem auf leerem Band ist nicht entscheidbar.

Beweisidee.

Das allgemeine Halteproblem H wird auf das Halteproblem auf leerem Band reduziert. Anschaulich funktioniert die Reduktion wie folgt.

Die Entscheidung, ob eine Turing-Maschine T auf dem Input x anhält, ist äquivalent zur Entscheidung, ob die Turing-Maschine T' auf dem leeren Band anhält.

Die TM T': $\boxed{\text{Band} := \text{x}} \longrightarrow \boxed{\text{T}}$

(Die TM T' schreibt zunächst die Eingabe x auf das leere Band und verhält sich dann wie die TM T.) $\square$

Das Halteproblem

Satz

Die Probleme H_0, H_S und H sind semi-entscheidbar.

Beweisidee.

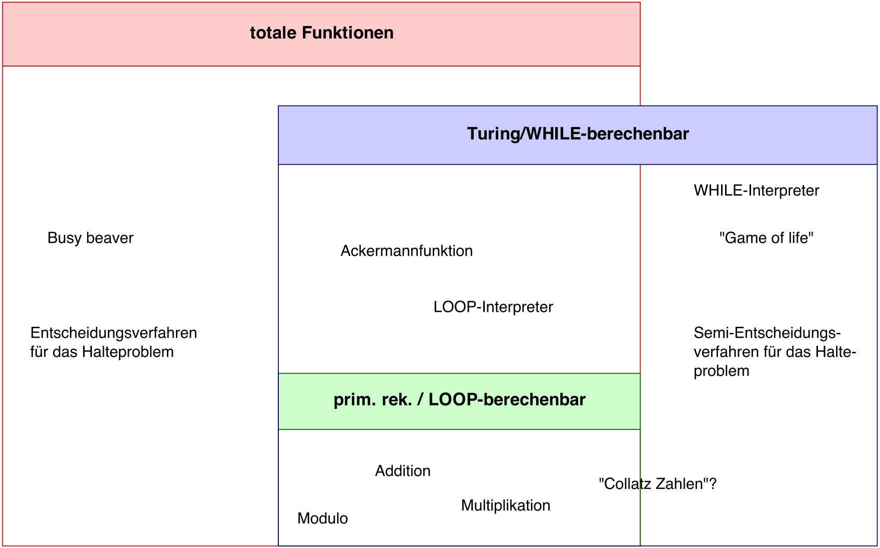
Wegen $H_s \preceq H \preceq H_0$ genügt es nachzuweisen, dass H_0 semi-entscheidbar ist.

Ein semi-Entscheidungsverfahren für H_0 kann gemäss dem folgenden Schema mit Hilfe einer universellen Turing-Maschine U angegeben werden:

$$\boxed{\text{U}} \longrightarrow \boxed{\text{Band} := 1}$$

□

Überblick



totale Funktionen
Turing/WHILE-berechenbar
prim. rek. / LOOP-berechenbar
Busy beaver
Entscheidungsverfahren für das Halteproblem
Ackermannfunktion
LOOP-Interpreter
WHILE-Interpreter
"Game of life"
Semi-Entscheidungs-verfahren für das Halte-problem
Addition
Multiplikation
Modulo
"Collatz Zahlen"?

Satz von Rice

Theorem

Ist R die Menge aller berechenbaren Funktionen und $S \subset R$ (mit $S \neq \varnothing$ und $S \neq R$), dann ist die Sprache

$$C(S) = \{w \in \{0,1\}^* \mid F_w \in S\}$$

unentscheidbar.

Beweis.

Für einen Beweis sei auf das Buch «Theoretische Informatik – kurz gefasst» (Seiten 122 und 123) von Uwe Schöning verwiesen $\square$

Satz von Rice

Konsequenzen:

- Es ist (im Allgemeinen) unmöglich mechanisch zu überprüfen, ob ein gegebenes Programm eine bestimmte Spezifikation erfüllt.
- Es ist (im Allgemeinen) unmöglich mechanisch zu überprüfen, ob ein gegebenes Programm frei von "bugs" ist.
- Es ist (im Allgemeinen) unmöglich mechanisch zu überprüfen, ob ein gegebenes Programm bei jeder Eingabe terminiert.
- Es ist (im Allgemeinen) unmöglich mechanisch zu überprüfen, ob zwei gegebene Programme dieselbe Funktionalität haben.

Backup: (Semi-) Entscheidbarkeit

Beispiel (Collatz-Zahlen)

Gegeben: Eine natürliche Zahl $n > 0$

Bildungsvorschrift: Ist n gerade, setze $n = n/2$
Ist n ungerade: setze $n = 3n + 1$

Gefragt: Mündet die Folge mit Startwert n in den Zyklus $4, 2, 1$?

Für $n = 8 : \quad 8 \to 4 \to 2 \to 1$

Für $n = 11 : \quad 9 \to 28 \to 14 \to 7 \to 22 \to 11 \to 34 \to 17 \to 52 \to 26$
$\to 13 \to 40 \to 20 \to 10 \to 5 \to 16 \to 8 \to 4 \to 2 \to 1$

Backup: (Semi-) Entscheidbarkeit

Beispiel (Collatz-Zahlen)

Frage: Sind 27, $6‘171$ und $837‘799$ Collatz-Zahlen?

$$27 \to 82, 41, 124, 62, 31, 94, 47, 142, 71, 214, 107, 322, 161, 484, 242\\
121, 364, 182, 91, 274, 137, 412, 206, 103, 310, 155, 466, 233, 700,\\
350, 175, 526, 263, 790, 395, 1186, 593, 1780, 890, 445, 1336, 668,\\
334, 167, 502, 251, 754, 377, 1132, 566, 283, 850, 425, 1276, 638,\\
319, 958, 479, 1438, 719, 2158, 1079, 3238, 1619, 4858, 2429,\\
7288, 3644, 1822, 911, 2734, 1367, 4102, 2051, 6154, 3077, \mathbf{9232},\\
4616, 2308, 1154, 577, 1732, 866, 433, 1300, 650, 325, 976, 488,\\
244, 122, 61, 184, 92, 46, 23, 70, 35, 106, 53, 160, 80, 40, 20, 10, 5,\\
16, 8, 4, 2, 1$$