# Simulation einer TM durch WHILE-Programm



Konfiguration einer Turing-Maschine wird in 3 Registern gespeichert:

- Register 1: Bandinhalt

  Sei $d := |\Gamma|$. Codiere $\Gamma$ als $\{0, 1, \ldots, d-1\}$ mit $\square = 0$.
  Dann ist $x_1 = (a_1 a_2 ... a_n)_d$ wenn Bandinhalt $a_1 a_2 \ldots a_n$ ist.

- Register 2: Kopfposition

  $x_2 = i$, wenn der Kopf auf dem $i$-ten Symbol von rechts $a_{n-i}$ steht.

- Register 3: Zustand

  $x_3 = q \in Q$ wobei $Q = \{0, 1, \ldots, |Q| - 1\}$.

Der Bandinhalt wird in Register 1 als Zahl gespeichert, wobei diese in der Basis $d$ aufgefasst wird und Bandsymbole als Ziffern $0, \ldots, d-1$ gelesen werden. Die 0 repräsentiert das Leerzeichen, dadurch wird die Erweiterung und Verkürzung (Sonderfälle der Übergangsrelation $\vdash_M$) am linken Rand besonders einfach, weil implizit beliebig viele führende Nullen vorhanden sind. Zur Behandlung der Sonderfälle am rechten Rand siehe die Übung H-15.

Die Angabe der Kopfposition als Offset vom rechten Rand ist sinnvoll, da dadurch der Zugriff auf das Symbol unter dem Kopf erleichtert wird.

# Simulation einer TM durch WHILE-Programm



Programm verwendet ein zusätzliches Aktiv-Flag in $R_4$:

$x_4 := 1$ ;
`while` $x_4 > 0$ `do`
  $x_4 := 0$ ;
  $x_5 := \lfloor x_1 / d^{x_2} \rfloor \bmod d$ ;

  `if` $x_3 = q$ && $x_5 = a$ `then`
    $x_3 := p$ ;
    $x_1 := x_1 + (b - a)d^{x_2}$ ;
    $x_2 := x_2 + 1$ ;
    $x_4 := 1$
  `end`
`end`

für Übergang
$\delta(q, a) = (p, b, L)$

Das Programm hat ein Codesegment wie oben zwischen `if` $x_3 = q$ ... `end` für jeden Eintrag in der Übergangstabelle der Maschine, plus zusätzliche für die Behandlung der Sonderfälle an den Rändern.

In jedem Programmstück werden Zustand und gelesenes Symbol abgefragt, und im passenden Fall die Konfiguration aktualisiert, indem der Zustand in $x_3$ aktualisiert, in $x_1$ die $x_2$-te Ziffer $a$ von rechts durch $b$ ersetzt wird, die Kopfposition $x_2$ aktualisiert wird (ausser bei den Sonderfällen am rechten Rand – siehe H-15), und das Aktiv-flag $x_4$ gesetzt wird.

Wenn die Maschine hält, gibt es keinen passenden Übergang, das Aktiv-flag bleibt bei 0 und die Simulation bricht ab.

# GOTO-Programm auf Turing-Maschine



Simulation eines GOTO-Programms

$M_1 : A_1 \; ; \; M_2 : A_2 \; ; \; \ldots \; ; \; M_n : A_n$

Codierung der benutzen Register $R_1, \ldots, R_k$ auf Band:

$1^{x_1} \# 1^{x_2} \# \ldots \# 1^{x_k}$

Für jeden Befehl $M_i : A_i$ eine Gruppe von Zuständen

$q_{i,1} \; , \; q_{i,2} \; , \; \ldots \; , \; q_{i,m_i}$

Invariante: Vor Beginn der Ausführung des Befehls $A_i$ im Zustand $q_{i,1}$ steht der Kopf ganz links.

Jedes Programm verwendet nur endlich viele Register $R_1, \ldots R_k$, diese werden alle (auch die anfänglich leeren) auf dem Band gespeichert.

Die Übergänge für die Zustände $q_{i,1}, \ldots, q_{i,m_i}$ simulieren die Ausführung des Befehls $A_i$.

# GOTO-Programm auf Turing-Maschine



Simulation des Befehls $M_i : x_j := x_j - 1$

- Finde Register $j$:
  Bewege Kopf nach rechts, bis $j-1$ mal # gesehen.
- Dekrement:
  Kopf nach rechts bis zur letzten 1 (vor # oder □)
  Verschiebe Bandinhalt rechts davon 1 Zelle nach links.
- Bewege Kopf zurück ans linke Ende, gehe in $q_{i+1,1}$

Simulation von $M_i : x_j := x_j + 1$ geht analog.

Für bedingten Sprung $M_i$ : `if` $x_j = 0$ `goto` $M_k$

- Finde Register $j$
- Falls # oder □ gelesen ($x_j = 0$):
  bewege Kopf nach links zurück, gehe in $q_{k,1}$
  sonst bewege Kopf nach links zurück, gehe in $q_{i+1,1}$

Wir haben keine Simuation der elementaren Anweisungen $x_i := 0$ (Initialisierung) und $x_i := x_j$ (Kopieren) sowie von unbedingten Sprünganweisungen angegeben.

Diese sind aber eigentlich redundant in GOTO-Programmen: ein Register kann auf 0 gesetzt werden, indem es dekrementiert wird bis 0 erreicht ist. Genauso kann durch iteriertes Inkrementieren ein Register kopiert werden. Ein unbedingter Sprung kann durch einenbedingten Sprung ersetzt werden, der ein Register abfragt, dass stets den Wert 0 hat.

# Churchsche These



## Theorem

*Die Turing-berechenbaren Funktionen sind genau die WHILE- bzw. GOTO-berechenbaren Funktionen.*

Auch: Church-Turing-These

Die Turing- (oder WHILE-, GOTO-) berechenbaren Funktionen erfassen genau den intuitiven Begriff von Berechenbarkeit.

Die Äquivalenz der verschiedenen Maschinenmodelle - sowie zahlreicher weiterer Modelle, die z.T. von sehr verschiedener Natur sind, z.B. Lambda-Kalkül, rekursive Funktionen - führt zu der Annahme, dass diese Modelle den intuitiven Begriff von algorithmischer Berechenbarkeit tatsächlich genau erfassen, der sog. Churchschen These.

Wegen der Churchschen These werden wir im Folgenden von jedem irgendwie spezifizierten Algorithmus annehmen, dass er als Turing-Maschine implementiert werden kann. In jedem konkreten Fall wäre die Konstruktion einer entsprechenden Maschine aufwändig, aber nicht wirklich schwierig.

Das Bild zeigt Alonzo Church, den Doktorvater von Alan Turing (und von dem im Teil I erwähnten Stephen Kleene). Church hat u.a. den Lambda-Kalkül eingeführt, ein weiteres zur Turing- und Registermaschine äquivalentes Berechnungsmodell. Dieser bildet die mathematische Grundlage funktionaler Programmiersprachen wie LISP und ML, und ist Ihnen vermutlich aus der Vorlesung *Programmierung und MOdelierung* bekannt.

# Übersicht





# Codierung von Turing-Maschinen

Konventionen für alle DTM:

- $\Sigma = \{0, 1\}$
- $Q = \{q_1, q_2, \ldots, q_k\}$ für ein $k$
- Anfangszustand ist $q_1$, Endzustände $F = \{q_2\}$
- $\Gamma = \{a_1, a_2, \ldots, a_m\}$ für ein $m$
- $a_1 = 0$, $a_2 = 1$ und $a_3 = \square$
- $D_1 := L$ und $D_2 := R$

# Codierung von Turing-Maschinen



Jeder Übergang $\delta(q_i, a_j) = (q_k, a_\ell, D_m)$ wird codiert durch den String

$$0^i\,1\,0^j\,1\,0^k\,1\,0^\ell\,1\,0^m.$$

Sind $C_1$, $C_2$, ..., $C_n$ die Codes der Übergänge von $M$, dann ist der Code für $M$ der String

$$C_1\,11\,C_2 11\,\ldots\,11\,C_n$$

Ist $e \in \Sigma^*$, so bezeichnet $M_e$ die DTM, deren Code $e$ ist.

Die Konvention, dass jede DTM nur einen Endzustand hat, ist keine echte Einschränkung: jede DTM kann leicht so modifiziert werden, dass sie diese Bedingung erfüllt, ohne dass sich an ihrem Verhalten sonst etwas ändert.

Ist ein Wort $e$ nicht von der Form, die durch Codierung einer DTM entsteht, z.B. wenn es drei oder mehr 1 hintereinander enthält, definieren wir $M_e$ als eine beliebige, aber fest gewählte Turing-Maschine, z.B. die triviale Maschine, die nur einen Zustand und eine nirgends definierte Übergangsfunktion hat.

# Universelle Turing-Maschine



Es gibt eine universelle Turing-Maschine $U$ mit:

- $U$ hält bei Eingabe $(e, w)$ genau dann, wenn $M_e$ bei Eingabe $w$ hält.
- $U$ erreicht bei $(e, w)$ den Endzustand, genau dann, wenn $M_e$ bei $w$ den Endzustand erreicht.
- $U$ berechnet bei $(e, w)$ die selbe Ausgabe wie $M_e$ bei $w$.

( Das Paar $(e, w)$ wird codiert durch $e\,111\,w$ )

Die universelle Maschine ist für den Informatiker etwas ganz natürliches, nämlich ein *Interpreter* für Turing-Maschinen, geschrieben als Turing-Maschine, so wie es Interpreter für Programmersprachen gibt, die in der jeweiligen Sprache selbst geschrieben sind.

# Unentscheidbare Probleme



Aus Kardinalitätsgründen gibt es unentscheidbare Sprachen:

- es gibt nur abzählbar viele Turing-Maschinen,
- aber es gibt überabzählbar viele Sprachen $L \subseteq \{0,1\}^*$.

Im Folgenden: eine konkrete Sprache, die unentscheidbar ist.

Zur Erinnerung: eine unendliche Menge $M$ ist abzählbar, wenn sie gleichmächtig mit $\mathbb{N}$ ist, d.h. wenn es eine bijektive Abbildung von $\mathbb{N}$ auf $M$ gibt. In diesem Fall lassen sich die Elemente von $M$ aufzählen als $M = \{m_0, m_1, m_2, \ldots\}$

Da $\{0,1\}^*$ gleichmächtig mit $\mathbb{N}$ ist (eine Bijektion ist z.B. die *dyadische Codierung*), und sich DTM in Wörter über $\{0,1\}$ codieren lassen, gibt es also nur abzählbar viele DTM.

Sprachen dagegen sind wegen der Bijektion in 1-1-Korrespondenz mit Mengen von natürlichen Zahlen, von denen es überabzählbar viele gibt. Hier zur Erinnerung und wegen der Analogie zum Folgenden der Beweis mittels der Diagonalisierungsmethode von Cantor:

Angenommen, es gäbe nur abzählbar viele Teilmengen von $\mathbb{N}$, aufgezählt als $N_0, N_1, N_2, \ldots$.
Dann definiere die Menge $D := \{\, i \in \mathbb{N};\ i \notin N_i\}$. Die Menge $D$ müsste in der Aufzählung vorkommen, also $D = N_d$ für ein $d \in \mathbb{N}$. Dann ist aber

$$d \in D \quad \text{gdw.} \quad d \notin N_d \quad \text{gdw.} \quad d \notin D$$

ein Widerspruch! Also ist die Annahme falsch, und es gibt mehr als abzählbar viele Teilmengen von $\mathbb{N}$.

# Das Halteproblem



Das Halteproblem ist die Sprache

$$H := \{ (e, w) \; ; \; M_e \text{ hält bei Eingabe } w \}$$

**Theorem**

*Das Halteproblem $H$ ist unentscheidbar.*

Die Unentscheidbarkeit des Halteproblems hat eigentlich nichts mit Turing-Maschinen zu tun, sie lässt sich nur so am einfachsten formulieren und beweisen. Die selbe Aussage gilt aber - mit im wesentlicheb dem selben Beweis - auch für jeden anderen Berechnungsformalismus, also auch für jede Programmiersprache.

# Beweis der Unentscheidbarkeit von $H$



Angenommen, $M_H$ sei eine DTM, die $H$ entscheidet.

Konstruiere eine neue DTM $D$ wie folgt:

> Bei Eingabe $w$ schreibe $(w, w)$ auf Band; lasse $M_H$ laufen.
>
> Endet $M_H$ im Endzustand, starte Endlosschleife.
>
> Andernfalls halte im Endzustand.

Sei $d$ Code dieser Maschine, also $D = M_d$. Dann gilt:

> $D$ hält bei Eingabe $d$
> $\leftrightarrow$ $M_d$ hält bei Eingabe $d$
> $\leftrightarrow$ $(d, d) \in H$
> $\leftrightarrow$ $M_H$ erreicht Endzustand bei Eingabe $(d, d)$
> $\leftrightarrow$ $D$ hält nicht bei Eingabe $d$

Widerspruch! Also kann es $M_H$ nicht geben.

---

Beachten sie die strukturelle Ähnlichkeit mit dem Diagonalisierungsbeweis für die Überabzählbarkeit der Potenzmenge von $\mathbb{N}$.

Die Maschine $D$ betrachtet die Eingabe $w$ as Code einer Turing-Maschine, und fragt (mittels der angenommenen Maschine für $H$), ob diese bei ihrem eigene Code als Eingabe hält – und verhält sich dann genau umgekehrt: wenn $M_w$ bei Einhabe $w$ hält, dann betritt sie eine Endlosschleife, hält also gerade nicht, andernfalls hält sie.

Der Widerspruch entsteht dann, wenn wir fragen, ob diese Maschine $D$ ihren eigenen Code akzeptiert.