Grundbegriffe der Theoretischen Informatik

Sommersemester 2018 - Thomas Schwentick

Teil C: Berechenbarkeit und Entscheidbarkeit

12: Verschiedene Berechnungsmodelle

Version von: 5. Juni 2018 (12:14)

Inhalt

Ein einfaches algorithmisches Puzzle-Problem

- Wir betrachten zwei Varianten eines „Puzzle-Problems“
  - Eine wird sich als deutlich schwieriger als die andere herausstellen

- Einfache Variante des Puzzlespiels:
  - Gegeben: schwarze und gelbe Spielsteintypen mit Strings
  - Von jedem Steintyp stehen beliebig viele Steine zur Verfügung
  - Lässt sich das selbe Wort aus schwarzen wie aus gelben Spielsteinen legen?

Beispiel

- Schwarze Steintypen: 01, 10, 011
- Gelbe Steintypen: 101, 00, 11
- Lösungswort: 1010011
  - ... in schwarz: 10 10 011
  - ...und in gelb: 101 00 11

Definition (Pseudo-PCP)

Gegeben: eine Folge $(u_1, v_1), \ldots, (u_k, v_k)$ von Paaren nicht-leerer Strings

Frage: Gibt es Indexfolgen $i_1, \ldots, i_n$ und $j_1, \ldots, j_m$ mit $n \geqslant 1$, so dass $u_{i_1} u_{i_2} \cdots u_{i_n} = v_{j_1} v_{j_2} \cdots v_{j_m}$?

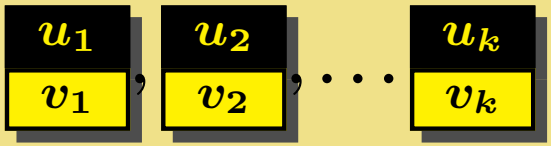
- Die Frage ist also: Gibt es einen String w, der sowohl aus u_is als auch aus v_js zusammen gesetzt werden kann

- Pseudo-PCP lässt sich mit Hilfe von Automaten in polynomieller Zeit entscheiden:
  - Konstruiere einen Automaten $\mathcal{A}$, der nichtleere Strings akzeptiert, die aus den u_i zusammengesetzt sind
  - Konstruiere einen Automaten $\mathcal{B}$, der nichtleere Strings akzeptiert, die aus den v_i zusammengesetzt sind
  - Teste ob $L(\mathcal{A}) \cap L(\mathcal{B}) \neq \varnothing$

Ein schwieriges algorithmisches Puzzle-Problem

- Jetzt betrachten wir die schwierigere Variante

- Gegeben eine Menge von Spielsteintypen

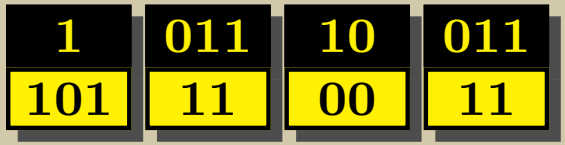


- Von jedem Typ stehen beliebig viele Steine zur Verfügung
- Lassen sich die Steine so in einer Reihe auslegen, dass das schwarze (obere) Wort gleich dem gelben (unteren) Wort ist?

Beispiel

- Steintypen:

- Mögliche Lösung:



Beispiel

- 

hat keine Lösung

Definition (PCP)

Gegeben: eine Folge $(u_1, v_1), \ldots, (u_k, v_k)$ von Paaren nicht-leerer Strings

Frage: Gibt es eine Indexfolge $i_1, \ldots, i_n$ mit $n \geqslant 1$, so dass

$$u_{i_1} u_{i_2} \cdots u_{i_n} = v_{i_1} v_{i_2} \cdots v_{i_n}?$$

- Die Frage ist also: Gibt es einen String w, der sowohl aus u_is als auch aus v_is zusammen gesetzt werden kann, und zwar **mit derselben nicht-leeren Indexfolge**?

- Wir nennen eine solche Indexfolge $\vec{i} = i_1, \ldots, i_n$ eine **Lösung** und den String $u_{i_1} u_{i_2} \cdots u_{i_n}$ einen **Lösungsstring**

✎ PCP ≡ Postsches Korrespondenzproblem

Ein komplizierteres Beispiel

Drittes Beispiel

- Steintypen: $\frac{001}{0}$, $\frac{01}{011}$, $\frac{01}{101}$, $\frac{10}{001}$
- Kleinste Lösung:

01	10	01	10	10	01	001	01	10	01
011	001	101	001	001	011	0	011	001	101

10	01	10	10	01	10	10	01	001	10	10
001	101	001	001	101	001	001	011	0	001	001

01	001	01	10	001	001	01	10	10	10	01
011	0	101	001	0	0	101	001	001	001	011

001	01	001	001	001	01	10	01	10	001	01
0	011	0	0	0	101	001	101	001	0	011

001	10	10	01	001	10	001	001	01	10	001
0	001	001	011	0	001	0	0	101	001	0

001	01	001	001	01	001	01	001	10	001	001	01
0	101	0	0	101	0	011	0	001	0	0	101

PCP ist algorithmisch nicht lösbar

- Wir werden in den nächsten Kapiteln den folgenden Satz beweisen

Satz

- PCP ist nicht entscheidbar

- Dazu benötigen wir einige Vorbereitung

- Zunächst müssen wir den Begriff „entscheidbar“ definieren
- Informell soll ein algorithmisches Problem entscheidbar sein, wenn es einen Algorithmus gibt, der bei jeder Eingabe anhält und immer die richtige Antwort gibt

- Um dies zu formalisieren benötigen wir eine Definition von „Algorithmus“

- Um zu definieren, was ein Algorithmus ist, benötigen wir ein allgemeines „Berechnungsmodell“
- Damit unsere Definition nicht zu modellspezifisch wird, ziehen wir mehrere Berechnungsmodelle in Betracht

Inhalt

Übersicht

- Wir suchen Antworten auf folgende Fragen
  - Was ist ein Algorithmus?
  - Wann ist eine Funktion berechenbar?
- Wir betrachten dazu verschiedene Berechnungsmodelle

- Zwei Modelle, die von Programmiersprachen inspiriert sind:
  - WHILE-Programme
  - GOTO-Programme

- Ein Modell, das als mächtige Erweiterung der endlichen Automaten aufgefasst werden kann, ursprünglich aber als mathematische Formalisierung des „Rechnens mit Papier und Bleistift“ gedacht war:
  - Turingmaschinen

- Später betrachten wir noch ein Modell, das durch rekursive Definitionen inspiriert ist:
  - μ-rekursive Funktionen

- Im nächsten Kapitel werden wir die Mächtigkeit dieser Berechnungsmodelle miteinander vergleichen

Partielle Funktionen

- Wir werden jetzt häufig partielle Funktionen über den natürlichen Zahlen verwenden

- Bei *partiellen Funktionen* f muss der Funktionswert nicht für alle Elemente der Grundmenge definiert sein

- Notation:
  - $f : \mathbb{N}_0 \rightharpoonup \mathbb{N}_0$
  - Um auszudrücken, dass $f(n)$ undefiniert ist, schreiben wir: $f(n) = \bot$

Beispiel

- Sei sqrt die partielle Funktion $\mathbb{N}_0 \rightharpoonup \mathbb{N}_0$, die jeder natürlichen Zahl n die natürliche Zahl m mit $m^2 = n$ zuordnet, wenn ein solches m existiert

- Es gilt also z.B.:
  - $\text{sqrt}(9) = 3$
  - $\text{sqrt}(10) = \bot$

Definition ($D(f)$, $W(f)$)

- Sei $f : \mathbb{N}_0 \rightharpoonup \mathbb{N}_0$ eine partielle Funktion
  - Der **Definitionsbereich** $D(f)$ einer partiellen Funktion f ist
    $$\{n \in \mathbb{N}_0 \mid f(n) \neq \bot\}$$
  - Der **Wertebereich** $W(f)$ einer partiellen Funktion f ist
    $$\{n \in \mathbb{N}_0 \mid \exists m \in \mathbb{N}_0 : f(m) = n\}$$

- Eine **totale Funktion** $\mathbb{N}_0 \to \mathbb{N}_0$ ist eine Funktion, die für alle natürlichen Zahlen definiert ist

✎ Der Begriff „partielle Funktion“ erzwingt nicht, dass es Zahlen gibt, für die kein Funktionswert existiert

- Jede totale Funktion ist also auch eine partielle Funktion

✎ Wir werden auch partielle Funktionen über Strings betrachten

WHILE-Programme: Beispiele

Beispiel

```
x1 := x2;
WHILE x3 ≠ 0 DO
      x1 := x1 + 1;
      x3 := x3 ∸ 1
END
```

- Variablen nehmen in WHILE-Programmen nur Werte aus $\mathbb{N}_0$ an
- Der Effekt des Programmes ist also:

$$x_1 := x_2 + x_3$$

☞ Und: $x_3 := 0$

Beispiel

```
x1 := 0;
WHILE x3 ≠ 0 DO
      x4 := x2;
      WHILE x4 ≠ 0 DO
            x1 := x1 + 1;
            x4 := x4 ∸ 1
      END;
      x3 := x3 ∸ 1
END
```

- Der (wesentliche) Effekt des Programmes ist: $x_1 := x_2 \times x_3$

Beispiel

```
x1 := 1;
WHILE x1 ≠ 0 DO
      x3 := x2 + 2
END
```

- Dieses Programm hält nie an...

WHILE-Programme: Syntax

- **WHILE-Programme** verwenden die folgenden syntaktischen Grundelemente:
  - Variablen: $x_1, x_2, x_3, \ldots$
  - Konstanten: $0, 1, 2, \ldots$
  - Trennsymbole: ; $:=$
  - Operationszeichen: $+ \dot{-}$
  - Schlüsselwörter: `WHILE`, `DO`, `END`

Definition (WHILE-Programm (Syntax))

- Die *Syntax von WHILE-Programmen* ist wie folgt definiert:

  Wertzuweisung:
  - $x_i := c$
  - $x_i := x_j$
  - $x_i := x_j + c$
  - $x_i := x_j \dot{-} c$

  sind WHILE-Programme
  (für jede Konstante c und $i, j \geqslant 1$)

  Reihung: Falls P_1 und P_2 WHILE-Programme sind, so auch $P_1; P_2$

  Bedingte Wiederholung: Ist P ein WHILE-Programm, so auch
  `WHILE` $x_i \neq 0$ `DO` P `END`

WHILE-Programme: Semantik (1/2)

- Wir definieren die Semantik von WHILE-Programmen durch ihre Wirkung auf Speicherinhalte
- Dabei modellieren wir Speicherinhalte durch Funktionen X, die die Werte der Variablen $x_1, x_2, \ldots$ repräsentieren
  - $X[i]$ repräsentiert den Wert von x_i

Definition (Speicherinhalt)

- Ein **Speicherinhalt** X ist eine Funktion $\mathbb{N} \to \mathbb{N}_0$, für die $X[i] \neq 0$ nur für endlich viele $i \in \mathbb{N}$ gilt
- Der **initiale Speicherinhalt X^b_{Init} bei Eingabe $b \in \mathbb{N}_0$** ist definiert durch:

$$X^b_{\text{Init}}[i] \stackrel{\text{def}}{=} \begin{cases} b & \text{für } i = 1 \\ 0 & \text{sonst} \end{cases}$$

✎ Wir schreiben hier $X[i]$ statt $X(i)$, um später die Unterscheidung zu anderen (runden) Klammern zu erleichtern

✎ Manchmal beschreiben wir einen Speicherinhalt als Folge von Zahlen $X[1], X[2], X[3], \ldots$
  - In dieser Sichtweise wird der initiale Speicherinhalt X^b_{Init} bei Eingabe b durch die Folge $b, 0, 0, \ldots$ repräsentiert

WHILE-Programme: Semantik (2/2)

Definition (WHILE-Programm (Semantik))

- Ist X ein Speicherinhalt und P ein WHILE-Programm, so bezeichne $P(X)$ den **Speicherinhalt nach Bearbeitung von P**
- $P(X)$ ist induktiv wie folgt definiert

- Falls P von der Form $x_i := x_j + c$ ist:

$$P(X)[k] \stackrel{\text{def}}{=} \begin{cases} X[j] + c & \text{für } k = i \\ X[k] & \text{sonst} \end{cases}$$

  - Analog für $x_i := c$ und $x_i := x_j$

- Falls P von der Form $x_i := x_j \dot{-} c$ ist:

$$P(X)[k] \stackrel{\text{def}}{=} \begin{cases} \max(X[j] - c, 0) & \text{für } k = i \\ X[k] & \text{sonst} \end{cases}$$

- Falls P von der Form $P_1; P_2$ ist:

$$P(X) \stackrel{\text{def}}{=} P_2(P_1(X))$$

Definition (Forts.)

- Ist P von der Form

$$\texttt{WHILE } x_i \neq 0 \texttt{ DO } P_1 \texttt{ END}$$

und X ein Speicherinhalt, so sei $P(X) \stackrel{\text{def}}{=}$

$$\begin{cases} X & \text{falls } X[i] = 0 \\ P(P_1(X)) & \text{sonst} \end{cases}$$

- Die durch ein WHILE-Programm P berechnete Funktion f_P ist wie folgt definiert:
  - Für jedes $b \in \mathbb{N}_0$ ist $f_P(b) \stackrel{\text{def}}{=} P(X^b_{\text{Init}})[1]$

- Eine partielle Funktion $f : \mathbb{N}_0 \rightharpoonup \mathbb{N}_0$ heißt **WHILE-berechenbar**, falls $f = f_P$ für ein WHILE-Programm P

✎ Der durch das Programm berechnete Funktionswert entspricht also dem Inhalt von x_1 nach Ende der Berechnung

WHILE-Programme: Bemerkungen

- Intuitive Bedeutung der Semantik der bedingten Wiederholung:
  - Solange $X[i] \neq 0$ gilt, wird P_1 ausgeführt

- Zu beachten:
  - Die Semantik-Definition ist rekursiv
  - $P(X)$ ist nicht für alle P und X definiert

- Auch mehrstellige Funktionen lassen sich durch WHILE-Programme berechnen:
  - Für jedes k-Tupel $\vec{a} = (a_1, \ldots, a_k)$ sei $f_{P,k}(\vec{a})$ der Wert von $X'(1)$, wobei
    - $X' = P(X^{\vec{a}}_{\text{Init}})$
    - und $X^{\vec{a}}_{\text{Init}}$ der Folge $a_1, \ldots, a_k, 0, 0, \ldots$ entspricht

- Wir haben schon gesehen, dass WHILE-Programme verschiedene Konstrukte simulieren können:
  - Addition zweier Variablen
  - Produkt zweier Variablen

- Es ist nicht schwer zu sehen, dass mit WHILE-Programmen auch bedingte Anweisungen der Art

  $$\texttt{IF } x_1 = c \texttt{ THEN } P \texttt{ END}$$

  simuliert werden können

- Wir werden im Folgenden solche Anweisungen als „syntaktischen Zucker“ erlauben

- WHILE-Programme im Sinne der formalen Definition nennen wir im Folgenden „einfache WHILE-Programme“

- Der Begriff „WHILE-berechenbar“ bezieht sich auf einfache WHILE-Programme

Inhalt

GOTO-Programme: Syntax

Definition (GOTO-Programm (Syntax))

- Ein **GOTO-Programm** besteht aus einer Folge
  - $M_1 : A_1;$
  - $M_2 : A_2;$
  - $\vdots$
  - $M_k : A_k$

  von Anweisungen A_i mit Sprungmarken M_i

- Mögliche Anweisungen:
  - **Wertzuweisung**:
    - $x_i := c$
    - $x_i := x_j$
    - $x_i := x_j + c$
    - $x_i := x_j \dot{-} c$
  - **Bedingter Sprung**: IF $x_i = c$ THEN GOTO M_j
  - **Stopanweisung**: HALT

Beispiel

```
1: x4 := 1;
2: x1 := x2;
3: IF x3 = 0 THEN GOTO 7;
4: x1 := x1 + 1;
5: x3 := x3 ∸ 1;
6: IF x4 = 1 THEN GOTO 3;
7: HALT
```

- Intuitiv hat dieses Programm den Effekt:
  $x_1 := x_2 + x_3$
  (und Seiteneffekte für x_3 und x_4)

✎ Das Beispiel illustriert, dass sich unbedingte Sprünge durch bedingte Sprünge simulieren lassen
  - Wir erlauben im Folgenden deshalb auch unbedingte Sprünge GOTO M_j als syntaktischen Zucker

✎ Sprungmarken, die nicht als Sprungadresse dienen, lassen wir oft weg

GOTO-Programme: Semantik (1/2)

- Da die Syntax von GOTO-Programmen nicht induktiv definiert ist, lässt sich ihre Semantik auch nicht ohne weiteres induktiv definieren
- Wir definieren die Semantik deshalb mit Hilfe von *Konfigurationen*

Definition (Konfiguration eines GOTO-Programms)

- Eine **Konfiguration eines GOTO-Programmes P** ist ein Paar $(M; X)$, wobei M eine Sprungmarke von P und X ein Speicherinhalt ist
- **Start-Konfiguration** bei Eingabe b: $(M_1; X_{\text{Init}}^b)$

GOTO-Programme: Semantik (2/2)

Definition (GOTO-Programm (Semantik))

- Ist M eine Sprungmarke eines GOTO-Programms, so bezeichnet $M + 1$ die Sprungmarke der folgenden Zeile

- Die **Nachfolge-Konfiguration** $(M'; X')$ einer Konfiguration $(M_\ell; X)$ ist wie folgt definiert:
  - Ist A_ℓ eine Wertzuweisung, so ist $M' \stackrel{\text{def}}{=} M_\ell + 1$ und X' ist definiert wie bei WHILE-Programmen
  - Falls A_ℓ ein bedingter Sprung

    $$\texttt{IF } x_i = c \texttt{ THEN GOTO } M_j$$

    ist, so ist $X' \stackrel{\text{def}}{=} X$ und

    $$M' \stackrel{\text{def}}{=} \begin{cases} M_j & \text{falls } X[i] = c \\ M_\ell + 1 & \text{sonst} \end{cases}$$

- **Halte-Konfiguration**: $(M_k + 1; X)$ oder $(M_\ell; X)$ und A_ℓ ist `HALT`

Definition (Forts.)

- Die **Berechnung** von P bei Eingabe b ist die eindeutig bestimmte Folge $K_1, K_2, \ldots$ von Konfigurationen mit:
  - $K_1 = (M_1, X^b_{\text{Init}})$ und
  - jede Konfiguration K_i ist die Nachfolgekonfiguration von K_{i-1} (für $i > 1$)

- Falls die Berechnung von P bei Eingabe b endlich ist, so ist die letzte Konfiguration eine Halte-Konfiguration $(M; X)$
- Dann sei wieder: $f_P(b) \stackrel{\text{def}}{=} X[1]$
- Falls die Berechnung von P bei Eingabe b unendlich ist, so ist $f_P(b) \stackrel{\text{def}}{=} \perp$,

- Eine partielle Funktion $f : \mathbb{N}_0 \to \mathbb{N}_0$ heißt **GOTO-berechenbar**, falls $f = f_P$ für ein GOTO-Programm P

Inhalt

Warum Turingmaschinen?

- Bisher haben wir Berechnungsmodelle betrachtet, die sich an Programmiersprachen anlehnen:
  WHILE- und GOTO-Programme

- Jetzt betrachten wir ein Berechnungsmodell, das „menschliche Rechner“ zum Vorbild nimmt
- Dieses Modell wurde 1936 von Alan Turing „erfunden“

- Warum hat jemand
  - etliche Jahre vor dem Bau des ersten Computers und
  - Jahrzehnte vor der Entwicklung „richtiger Programmiersprachen“

  ein abstraktes Berechnungsmodell erfunden?

- Turing war Mathematiker und wollte beweisen, dass es kein automatisches Verfahren gibt, das zu jeder mathematischen Aussage entscheidet, ob sie wahr oder falsch ist

- Etwas anders formuliert, wollte er zeigen, dass es keinen Algorithmus für das folgende algorithmische Problem gibt

Definition (Allgemeingültigkeitsproblem)

Gegeben: Prädikatenlogische Formel φ

Frage: Gilt für alle passenden Modelle $\mathcal{M}$:
$\mathcal{M} \models \varphi$?

Turings Ideen zur Berechenbarkeit (1/4)

- Wie gesagt: zu Turings Zeit gab es noch keine künstlichen programmierbaren Rechner
- Wenn er von einem *Computer* sprach, meinte er einen Menschen, der nach einem festgelegten Verfahren etwas berechnet

- Seine Vorstellungen, wie ein solcher *Computer* arbeitet, hat er in der folgenden Arbeit beschrieben:
  - A. M. Turing. On computable numbers, with an application to the Entscheidungsproblem. *Proc. London Math. Soc.*, 2(42):230–265, 1936

- Die Abstraktion des Computers, die Turing in dieser Arbeit definierte, wird heute *Turingmaschine* genannt
- Schauen wir mal, welche Gedanken sich Alan Turing 1936 so gemacht hat...

Turings Ideen zur Berechenbarkeit (2/4)

- Computing is normally done by writing certain symbols on paper
- We may suppose this paper is divided into squares like a child's arithmetic book

- In elementary arithmetic the two-dimensional character of the paper is sometimes used
- But such a use is always avoidable, and I think that it will be agreed that the two-dimensional character of paper is no essential of computation
- I assume then that the computation is carried out on one-dimensional paper, i.e. on a tape divided into squares ☞ Arbeitsstring

- I shall also suppose that the number of symbols which may be printed is finite ☞ Arbeitsalphabet
  - If we were to allow an infinity of symbols, then there would be symbols differing to an arbitrarily small extent

Turings Ideen zur Berechenbarkeit (3/4)

- The **behaviour** of the computer at any moment is **determined by the symbols which he is observing and his "state of mind"** at that moment ☞ Zustand

- We may suppose that there is a **bound** B to the **number of symbols** or squares **which the computer can observe** at one moment ☞ Zeiger/Kopf
- If he wishes to observe more, he must use successive observations

- We will also suppose that the **number of states** of mind which need be taken into account **is finite**

- Let us imagine the operations performed by the computer to be split up into **"simple operations"** which are so elementary that it is not easy to imagine them further divided
- Every such operation consists of some change of the physical system consisting of the computer and his tape

- We know the state of the system if we know the sequence of symbols on the tape, which of these are observed by the computer (possibly with a special order), and the state of mind of the computer ☞ Konfiguration

- We may suppose that **in a simple operation not more than one symbol is altered**
  - Any other changes can be set up into simple changes of this kind

- The situation in regard to the squares whose symbols may be altered in this way is the same as in regard to the observed squares
- We may, therefore, without loss of generality, **assume that the squares whose symbols are changed are always "observed" squares**

Turings Ideen zur Berechenbarkeit (4/4)

- Besides these changes of symbols, the simple operations must include changes of distribution of observed squares
- The new observed squares must be immediately recognisable by the computer

☞ Kopfbewegung

- I think it is reasonable to suppose that they can only be squares whose distance from the closest of the immediately previously observed squares does not exceed a certain fixed amount
- Let us say that each of the new observed squares is within L squares of an immediately previously observed square

- The most general single operation must therefore be taken to be one of the following:

  (A) A possible change (a) of symbol together with a possible change of state of mind

  (B) A possible change (b) of observed squares, together with a possible change of state of mind

- The operation actually performed is determined, as has been suggested on p.250, by the state of mind of the computer and the observed symbols
- In particular, they determine the state of mind of the computer after the operation is carried out

☞ Transition

Vom Automaten zur Turingmaschine

- Turingmaschinen können als Erweiterung von endlichen Automaten in drei Stufen aufgefasst werden

(1) Mehr Bewegung

- Der Kopf darf sich nach rechts *und* nach links bewegen

✎ Endliche Automaten mit dieser Erweiterung können nur reguläre Sprachen entscheiden

(2) Schreiben

- Die Symbole der Eingabe können verändert werden — jeweils an der Position des Kopfes

- Das Berechnungsmodell, das endliche Automaten um (1) und (2) erweitert, wird *linear beschränkte Automaten* genannt
  - Sie entscheiden genau die kontextsensitiven Sprachen

(3) Mehr Platz

- Der Arbeitsbereich kann über die Eingabe hinaus erweitert werden (nach rechts)
- Links von der Eingabe steht ein Symbol ($\triangleright$), das den linken Rand markiert, der nicht verschoben werden kann
- Berechnungen enden nicht mehr durch Verlassen der Eingabe sondern durch Erreichen spezieller *Endzustände*

- Wir betrachten nun zunächst Beispiele von Turingmaschinen

Turingmaschinen: 1. Beispiel

Beispiel

- Turingmaschine zum Test, ob die Eingabe von der Form ww^R ist

Idee: Vergleiche jeweils das erste mit dem letzten Symbol und lösche beide (durch Überschreiben mit $\#$ bzw. $\sqcup$)

a: 0 und 1 überlesen, nach links — falls $\triangleright$ oder $\#$ nach rechts in b

b: Falls 0 nach rechts in c — falls 1 nach rechts in d (dabei $0/1$ durch $\#$ überschreiben) — falls $\sqcup$ akz.

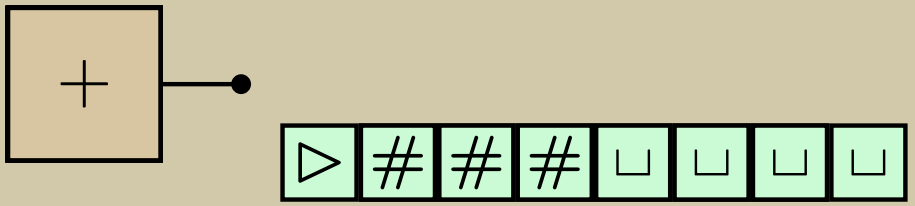
c: 0 und 1 überlesen nach rechts bis $\sqcup$, dann nach links in e

d: 0 und 1 überlesen nach rechts bis $\sqcup$, dann nach links in f

e: Falls 0 durch $\sqcup$ ersetzen nach links in a — falls 1 oder $\#$ ablehnen

f: Falls 1 durch $\sqcup$ ersetzen nach links in a — falls 0 oder $\#$ ablehnen

1. Beispielberechnung:



Turingmaschinen: 1. Beispiel, 2. Berechnung

Beispiel

Turingmaschine zum Test, ob die Eingabe von der Form ww^R ist

a: 0 und 1 überlesen, nach links — falls $\triangleright$ oder $\#$ nach rechts in b

b: Falls 0 nach rechts in c — falls 1 nach rechts in d (dabei $0/1$ durch $\#$ überschreiben) — falls $\sqcup$ akz.

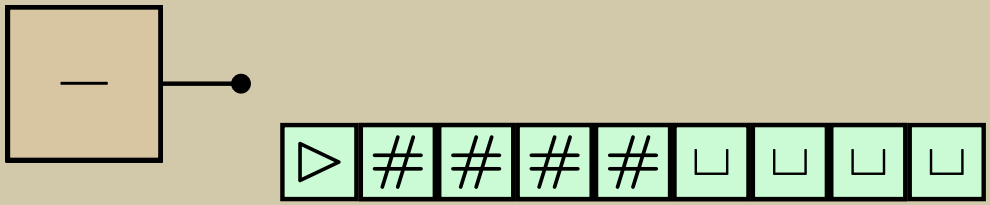
c: 0 und 1 überlesen nach rechts bis $\sqcup$, dann nach links in e

d: 0 und 1 überlesen nach rechts bis $\sqcup$, dann nach links in f

e: Falls 0 durch $\sqcup$ ersetzen nach links in a — falls 1 oder $\#$ ablehnen

f: Falls 1 durch $\sqcup$ ersetzen nach links in a — falls 0 oder $\#$ ablehnen

2. Beispielberechnung:



Turingmaschinen: Definition

Definition (Turingmaschine (Syntax))

- Eine **Turingmaschine** $M = (Q, \Gamma, \delta, s)$ besteht aus
  - einer endliche Menge Q, (**Zustandsmenge**)
  - einem Alphabet Γ, mit $\sqcup, \triangleright \in \Gamma$, (**Arbeitsalphabet**)
  - einer Funktion
    $\delta : Q \times \Gamma \to (Q \cup \{\text{ja}, \text{nein}, h\}) \times \Gamma \times \{\leftarrow, \downarrow, \rightarrow\}$
    (**Transitionsfunktion**),
  - einem ausgezeichneten Zustand $s \in Q$ (**Startzustand**)
- Dabei seien $Q, \Gamma, \{h, \text{ja}, \text{nein}\}$ und $\{\leftarrow, \downarrow, \rightarrow\}$ paarweise disjunkt

- Turingmaschinen müssen außerdem die folgenden Bedingungen erfüllen:
  - Das Symbol $\triangleright$ für den linken Rand darf nicht überschrieben werden
  - von $\triangleright$ darf sich der Kopf nicht nach links bewegen

- Das lässt sich dadurch erreichen, dass $\delta(q, \triangleright)$ immer von der Form $(q', \triangleright, d)$ mit $d \in \{\downarrow, \rightarrow\}$ ist

Turingmaschinen: Diagramm-Darstellung

Zustände der Beispiel-TM

a: $\mathbf{0}$ und $\mathbf{1}$ überlesen, nach links — falls $\triangleright$ oder $\#$ nach rechts in b

b: Falls $\mathbf{0}$ nach rechts in c — falls $\mathbf{1}$ nach rechts in d (dabei $\mathbf{0}/\mathbf{1}$ durch $\#$ überschreiben) — falls $\sqcup$ akz.

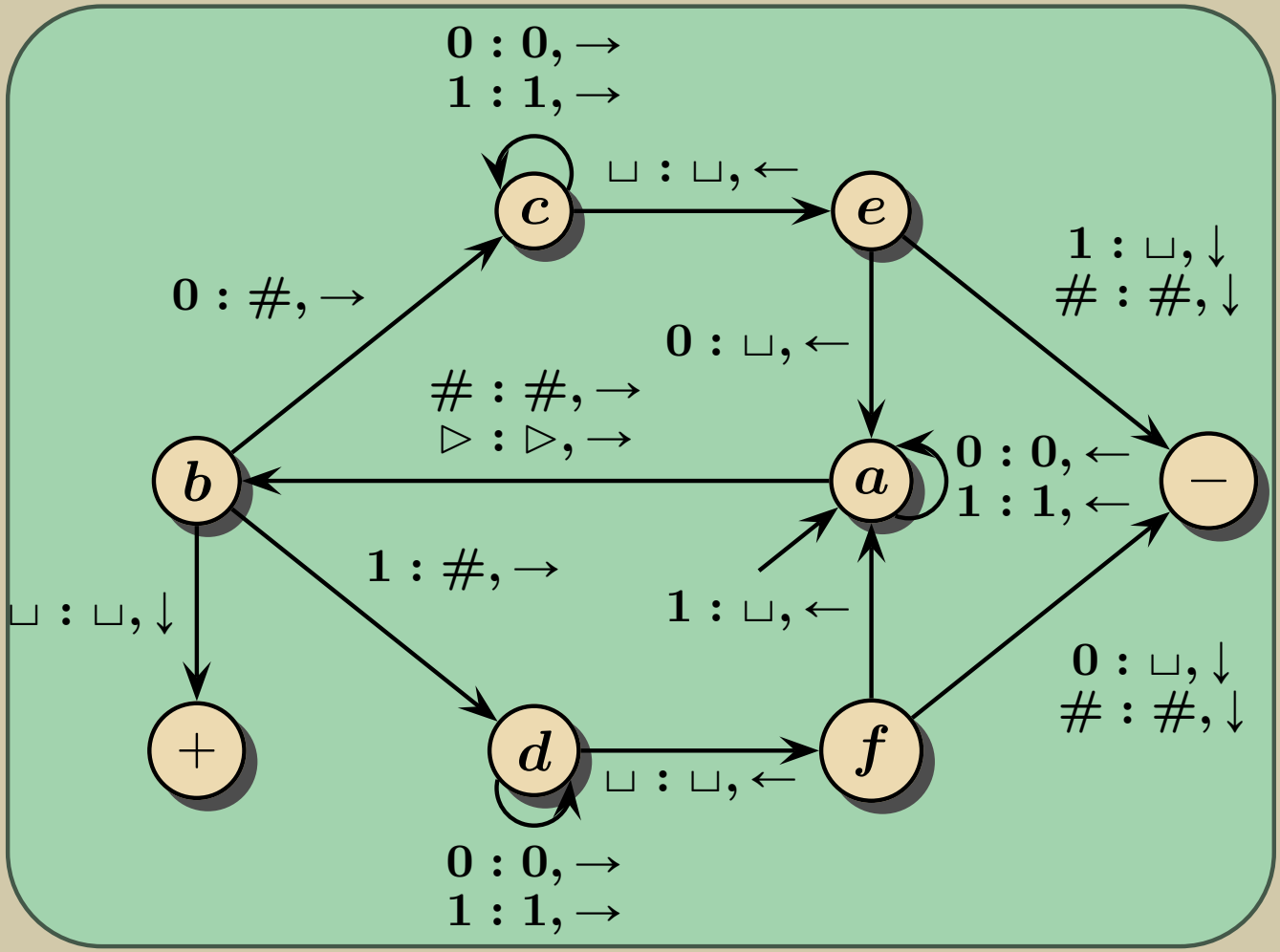
c: $\mathbf{0}$ und $\mathbf{1}$ überlesen nach rechts bis $\sqcup$, dann nach links in e

d: $\mathbf{0}$ und $\mathbf{1}$ überlesen nach rechts bis $\sqcup$, dann nach links in f

e: Falls $\mathbf{0}$ durch $\sqcup$ ersetzen nach links in a — falls $\mathbf{1}$ oder $\#$ ablehnen

f: Falls $\mathbf{1}$ durch $\sqcup$ ersetzen nach links in a — falls $\mathbf{0}$ oder $\#$ ablehnen

Beispiel-Turingmaschine als Diagramm

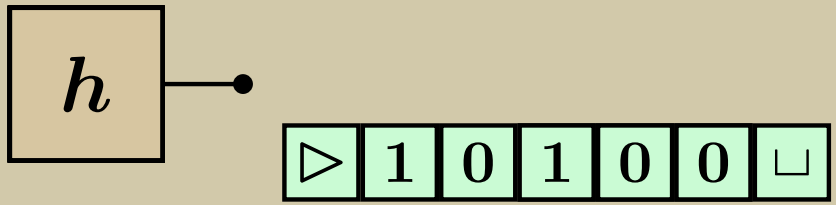


- Konvention: ist für ein Paar $(q, \sigma) \in Q \times \Gamma$ kein Übergang eingezeichnet, so sei $\delta(q, \sigma) \stackrel{\text{def}}{=} (\text{nein}, \sigma, \downarrow)$

Turingmaschinen: 2. Beispiel

2. Beispiel-TM: Inkrementieren einer Binärzahl

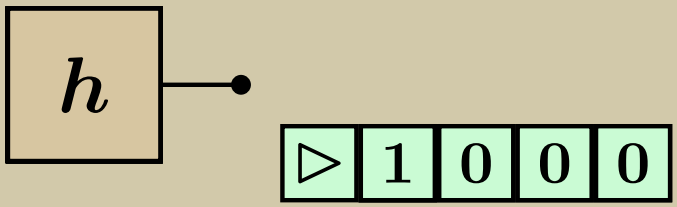
- Beschreibung der Zustände:
  - a: Die TM läuft, ohne etwas zu verändern, nach rechts bis zum ersten Leerzeichen, und dann einen Schritt nach links in den Zustand b.
  - b: Das maximale Suffix der Form 1^i wird durch 0^i ersetzt.
    - Ist davor eine 0, so wird sie durch 1 ersetzt und M geht in den Zustand e
      - ✎ Das Suffix der Eingabe ist von der Form 01^i und wird durch 10^i ersetzt
    - Ist davor ein $\triangleright$, so geht M in Zustand c
      - ✎ Die Eingabe war von der Form 1^i, wurde in 0^i geändert, und muss noch in 10^i umgewandelt werden
  - c: Ersetzt die erste 0 durch eine 1, und geht in den Zustand d
  - d: Fügt eine 0 hinten an und geht in den Zustand e
  - e: läuft zum linken Rand und geht in den Endzustand h



Turingmaschinen: 2. Beispiel, 2. Berechnung

2. Beispiel-TM: Inkrementieren einer Binärzahl (Forts.)

- Turingmaschine zum Inkrementieren einer Binärzahl:
- Beschreibung der Zustände:
  - a: Die TM läuft, ohne etwas zu verändern, nach rechts bis zum ersten Leerzeichen, und dann einen Schritt nach links in den Zustand b.
  - b: Das maximale Suffix der Form 1^i wird durch 0^i ersetzt.
    - Ist davor eine 0, so wird sie durch 1 ersetzt und M geht in den Zustand e
    - Ist davor ein $\triangleright$, so geht M in Zustand c
  - c: Ersetzt die erste 0 durch eine 1, und geht in den Zustand d
  - d: Fügt eine 0 hinten an und geht in den Zustand e
  - e: läuft zum linken Rand und geht in den Endzustand h



Turingmaschinen: 2. Beispiel als Diagramm

Beispiel: Zustände der 2. TM

a: Laufe nach rechts bis zum ersten $\sqcup$, dann einen Schritt nach links in Zustand b.

b: Gehe nach links bis zur nächsten 0 — ersetze dabei jede 1 durch 0

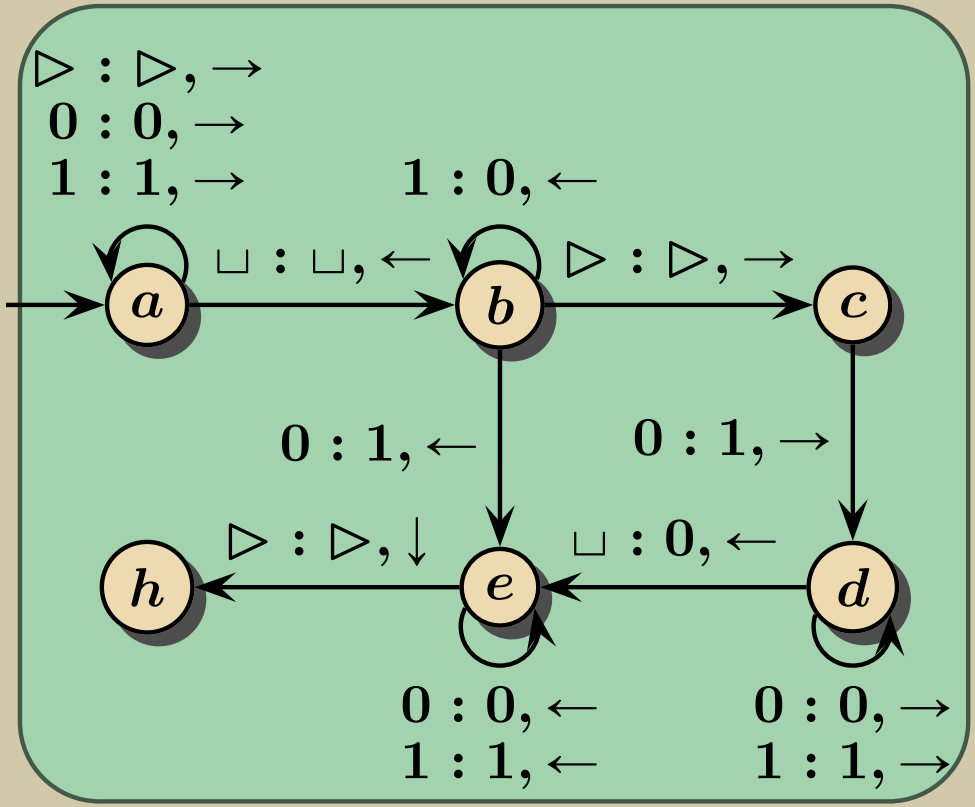
- Ist davor 0, ersetze durch 1 und gehe in Zustand e
- Ist davor $\triangleright$, gehe in Zustand c

c: Ersetze die erste 0 durch eine 1, und gehe in den Zustand d

d: Fügt eine 0 hinten an und gehe in den Zustand e

e: Laufe zum linken Rand und gehe in den Endzustand h

Diagramm zur 2. TM



Turingmaschinen: Konfigurationen (1/2)

- Um die aktuelle Situation einer TM zu beschreiben verwenden wir Konfigurationen, bestehend aus einem Zustand und einer String-Zeigerbeschreibung

Definition (String-Zeigerbeschreibung)

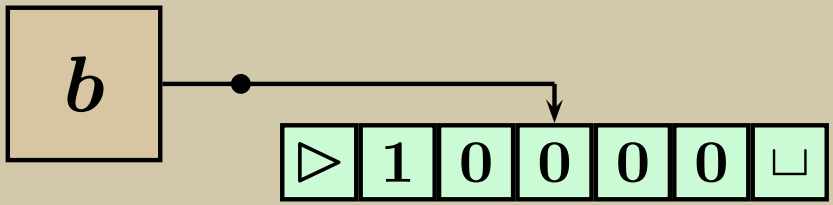
- Eine **String-Zeigerbeschreibung** (w, z) besteht aus
  - einem String $w \in \Gamma^*$
  - und einer Zeigerposition $z \in \mathbb{N}$, $z \leqslant |w|$

- Manchmal verwenden wir eine andere Notation und schreiben (u, σ, v) statt (w, z), falls
  - $w = u\sigma v$,
  - $|u\sigma| = z$
- σ ist dann das Zeichen, auf das der Zeiger zeigt, v ist der String rechts vom Zeiger, u ist der String links vom Zeiger

✎ Die Definition ist gegenüber den Vorjahren verändert: dort wurde $\triangleright$ nicht in Konfigurationen repräsentiert

Beispiel

Die String-Zeigerbeschreibung zu



ist

- $(\triangleright 10000\sqcup, 4)$ oder
- $(\triangleright 10, 0, 00\sqcup)$

Definition (Konfiguration)

- Eine **Konfiguration** von M ist ein Tupel $(q, (w, z))$ mit
  - $q \in Q \cup \{\text{ja}, \text{nein}, h\}$
    ☞ aktueller Zustand
  - w der aktuelle String
  - z die Position des Zeigers der TM ☞ linker Rand ist Position 1

Turingmaschinen: Konfigurationen (2/2)

Definition (Nachfolgekonfiguration, $\vdash_M$)

- Sei $K = (q, (w, z))$ eine Konfiguration mit $w[z] = \sigma$ und sei $\delta(q, \sigma) = (q', \tau, d)$ mit $\tau \in \Gamma$, $d \in \{\leftarrow, \downarrow, \rightarrow\}$

- Dann ist die **Nachfolgekonfiguration** $K' = (q', (w', z'))$ von K wie folgt definiert:

  ☞ Schreibweise: $K \vdash_M K'$

  - $z' = z + 1$, falls $d = \rightarrow$,
  - $z' = z$, falls $d = \downarrow$,
  - $z' = z - 1$, falls $d = \leftarrow$,

    ✎ Da vom linken Rand kein Linksschritt möglich ist, wird z niemals kleiner als 1

  - $w' = w[z/\tau]\sqcup$, falls $z = |w|$ und $d = \rightarrow$,
  - $w' = w[z/\tau]$, andernfalls

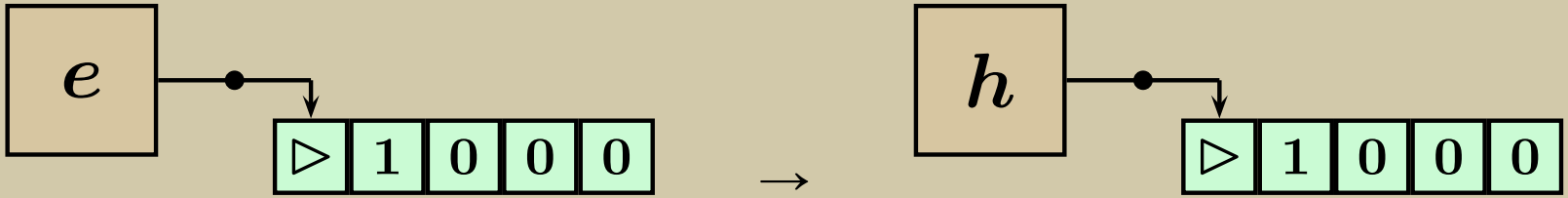
    ✎ Dabei bezeichnet $w[z/\tau]$ den String, der aus w entsteht, indem das Zeichen an Position z durch τ ersetzt wird

- Wir schreiben $K \vdash_M^* K'$, falls es Konfigurationen $K_1, \ldots, K_m$ gibt mit $K \vdash_M K_1 \vdash_M \cdots \vdash_M K_m \vdash_M K'$

Turingmaschinen: Illustration der Nachfolgekonfiguration (1/2)

Beispiel: Schritt ohne Kopf-Bewegung

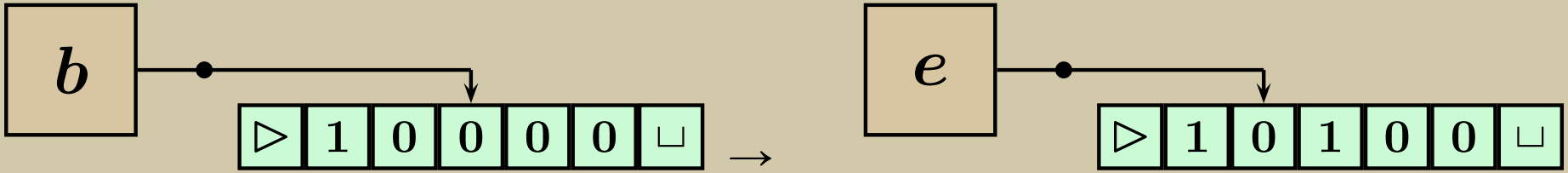
- $\delta(e, \triangleright) = (h, \triangleright, \downarrow)$



$(e, (\triangleright 1000, 1)) \rightarrow (h, (\triangleright 1000, 1))$

Beispiel: Linksschritt

- $\delta(b, 0) = (e, 1, \leftarrow)$

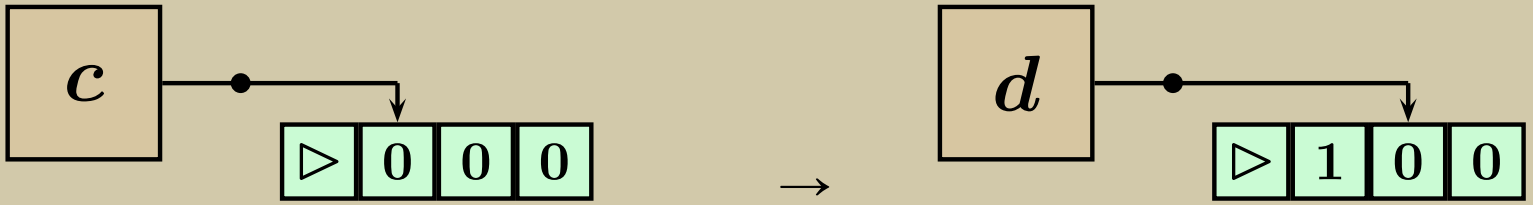


$(b, (\triangleright 10000\sqcup, 4)) \rightarrow (e, (\triangleright 10100\sqcup, 3))$

Turingmaschinen: Illustration der Nachfolgekonfiguration (2/2)

Beispiel: Rechtsschritt

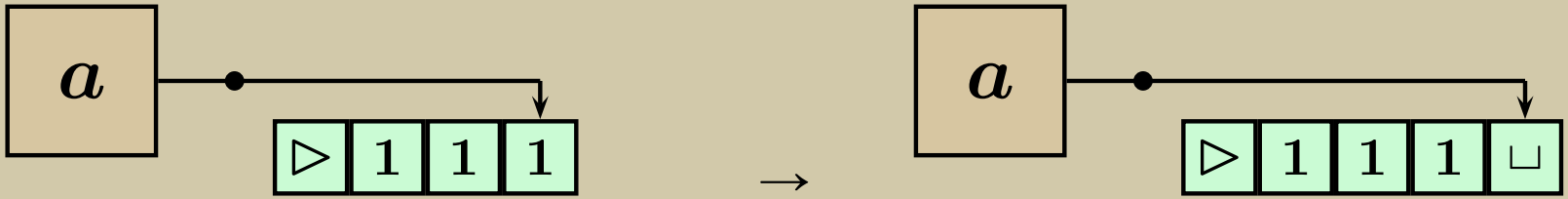
- $\delta(c, 0) = (d, 1, \rightarrow)$



$(c, (\triangleright 000, 2)) \rightarrow (d, (\triangleright 100, 3))$

Beispiel: Rechtsschritt mit neuem Blank

- $\delta(a, 1) = (a, 1, \rightarrow)$



$(a, (\triangleright 111, 4)) \rightarrow (a, (\triangleright 111\sqcup, 5))$

Turingmaschinen: Semantik (1/2)

Definition (Turingmaschine (Semantik))

- Sei $\Sigma \subseteq \Gamma - \{\triangleright, \sqcup\}$ (**Ein-/Ausgabe-Alphabet**)

- Die **Startkonfiguration** von M bei Eingabe $u \in \Sigma^*$ ist
$$K_0(u) \stackrel{\text{def}}{=} (s, (\triangleright u, 1))$$
- $(q, (w, z))$ heißt **Haltekonfiguration**, falls
$$q \in \{h, \text{ja}, \text{nein}\}$$

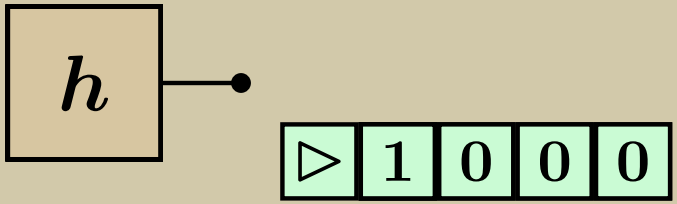
- $K_0, K_1, \ldots, K_t$ heißt **endliche Berechnung von M bei Eingabe u**, falls
  - $K_0 = K_0(u)$,
  - $K_i \vdash_M K_{i+1}$ für alle $i < t$, und
  - K_t eine Haltekonfiguration ist

- **M akzeptiert u**, falls $K_0(u) \vdash_M^* (\text{ja}, (w, z))$
- **M lehnt u ab**, falls $K_0(u) \vdash_M^* (\text{nein}, (w, z))$
  - (für gewisse $w \in \Gamma^*$, $z \in \mathbb{N}$)

- $M(u) \stackrel{\text{def}}{=}$ die (endliche oder unendliche) Berechnung von M bei Eingabe u ☞ $M(u)$ bezeichnet nicht die Ausgabe!

Turingmaschinen: Beispiel einer Konfigurationsfolge

Beispiel



$$
\begin{aligned}
(a, (\epsilon, \triangleright, 111)) &\vdash_M (a, (\triangleright, 1, 11)) \\
&\vdash_M (a, (\triangleright 1, 1, 1)) \\
&\vdash_M (a, (\triangleright 11, 1, \epsilon)) \\
&\vdash_M (a, (\triangleright 111, \sqcup, \epsilon)) \\
&\vdash_M (b, (\triangleright 11, 1, \sqcup)) \\
&\vdash_M (b, (\triangleright 1, 1, 0\sqcup)) \\
&\vdash_M (b, (\triangleright, 1, 00\sqcup)) \\
&\vdash_M (b, (\epsilon, \triangleright, 000\sqcup)) \\
&\vdash_M (c, (\triangleright, 0, 00\sqcup)) \\
&\vdash_M (d, (\triangleright 1, 0, 0\sqcup)) \\
&\vdash_M (d, (\triangleright 10, 0, \sqcup)) \\
&\vdash_M (d, (\triangleright 100, \sqcup, \epsilon)) \\
&\vdash_M (e, (\triangleright 10, 0, 0)) \\
&\vdash_M (e, (\triangleright 1, 0, 00)) \\
&\vdash_M (e, (\triangleright, 1, 000)) \\
&\vdash_M (e, (\epsilon, \triangleright, 1000)) \\
&\vdash_M (h, (\epsilon, \triangleright, 1000))
\end{aligned}
$$

Turingmaschinen: Semantik (2/2)

Definition ($L(M)$)

- Eine TM M **entscheidet** eine Sprache L, falls für jedes $u \in \Sigma^*$ gilt:
  - $u \in L \Rightarrow M$ akzeptiert u
  - $u \notin L \Rightarrow M$ lehnt u ab
- $L(M) \stackrel{\text{def}}{=}$ Menge aller von M akzeptierten Wörter

Zu Beachten

- $L(M)$ ist immer definiert
- Aber M entscheidet $L(M)$ nicht immer!
  - Es könnte sein, dass M für gewisse Eingabewörter, die nicht in $L(M)$ sind, nicht anhält
- M entscheidet $L(M)$ genau dann, wenn M für jedes Eingabewort anhält

- Turingmaschinen können auch Funktionen berechnen:

Definition

- $f_M(u) \stackrel{\text{def}}{=} v \in \Sigma^*$, falls
  - $K_0(u) \vdash_M^* (h, (\triangleright v, 1))$ oder
  - $K_0(u) \vdash_M^* (h, (\triangleright v \tau w, 1))$ für ein $\tau \in \Gamma - \Sigma$, $w \in \Gamma^*$

✎ $f_M(u)$ ist nur dann definiert, wenn der Zeiger von M im Haltezustand ganz links steht
  - $f_M(u)$ ist dann die maximale Folge von Zeichen aus Σ, die direkt rechts vom Zeiger stehen
- Im Allgemeinen ist f_M also eine partielle Funktion $\Sigma^* \rightharpoonup \Sigma^*$

Definition (Turing-berechenbar)

- Eine partielle Funktion $f : \Sigma^* \rightharpoonup \Sigma^*$ heißt **Turing-berechenbar**, falls $f = f_M$, für eine Turingmaschine M

Literaturangaben

- Die Darstellung in diesem Kapitel richtet sich weitgehend nach
  - Uwe Schöning. *Theoretische Informatik - kurzgefaßt (3. Aufl.)*. Hochschultaschenbuch. Spektrum Akademischer Verlag, 1997