

# Grundlagen Theoretischer Informatik 2

WiSe 2011/12 in Trier

Henning Fernau

Universität Trier

fernau@uni-trier.de

## **Grundlagen Theoretischer Informatik 2** Gesamtübersicht

- Organisatorisches; Einführung
- Ersetzungsverfahren: Grammatiken und Automaten
- **Rekursionstheorie-Einführung**
- Komplexitätstheorie-Einführung

## Der intuitive Berechenbarkeitsbegriff

Eine (partielle) Funktion  $f : \mathbb{N}^k \dashrightarrow \mathbb{N}$  heißt *berechenbar*, wenn es einen Algorithmus, z.B. ein `JAVA`-Programm, gibt, der bei der Eingabe  $(n_1, \dots, n_k)$  entweder nach endlich vielen Schritten  $f(n_1, \dots, n_k)$  als Ausgabe berechnet und dann anhält

(dann ist  $f$  bei der Eingabe  $(n_1, \dots, n_k)$  definiert)

oder nie anhält ( $f$  ist nunmehr undefiniert an der Stelle  $(n_1, \dots, n_k)$ ).

## Berechenbarkeitsbeispiele

a) Berechnung der Summe zweier natürlicher Zahlen  $m$  und  $n$  als Funktion  $f : \mathbb{N}^2 \rightarrow \mathbb{N}$  mit  $f(m, n) = n + m$ :

```
import java.util.Vector;
import java.math.BigInteger;
class Addition {
    public static void main(String args[]) {
        BigInteger m=new BigInteger(args[0]);
        BigInteger n=new BigInteger(args[1]);
        while (n.compareTo(BigInteger.ZERO) > 0) {
            n = n.subtract(BigInteger.ONE);
            m = m.add(BigInteger.ONE);
        }
        System.out.println(m.toString());
    }
}
```

## Berechenbarkeitsbeispiele

b) Berechnung einer partiellen Funktion  $f : \mathbb{N} \dashrightarrow \mathbb{N}$  durch folgenden Algorithmus:

```
import java.math.BigInteger;
class Addition {
    public static void main() {
        BigInteger n=BigInteger.ONE;
        while (n.compareTo(BigInteger.ZERO) > 0) {
            n = n.add(BigInteger.ONE);
        }
    }
}
```

Was geschieht hier?

## Berechenbarkeitsbeispiele

c) Berechnung einer totalen Funktion  $f : \mathbb{N} \rightarrow \mathbb{N}$  mit

- $f(n) = 1$ , falls die Dezimaldarstellung von  $n$  ein Anfangsabschnitt der Dezimalbruchentwicklung von  $\pi = 3, 1415\dots$  ist,
- $f(n) = 0$  sonst.

z.B.:

$$\begin{aligned}f(3) &= 1 & f(314) &= 1 & f(31415) &= 1 & f(31416) &= 0 \\f(3141592653589793238462643383279502884197169399) &= 1 \\f(3141592653589793238462643383279502884197269399) &= 0\end{aligned}$$

Dezimalbruchentwicklung von  $\pi$  mit beliebiger Stellenzahl berechenbar:

$$\pi = 4 \cdot \lim_{t \rightarrow \infty} \left( 1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \dots \pm \frac{1}{2t-1} \right)$$

## Berechenbarkeitsbeispiele

d) Berechnung einer totalen Funktion  $g : \mathbb{N} \rightarrow \mathbb{N}$  mit

- $g(n) = 1$ , falls die Dezimaldarstellung von  $n$  als Teilwort in der Dezimalbruchentwicklung von  $\pi = 3,1415\dots$  vorkommt,
- 0 sonst.

Ob diese Funktion berechenbar ist oder nicht, ist nicht bekannt!

Dazu weiß man heute zu wenig über die Zahl  $\pi$ . Möglich, dass jedes aus den Dezimalziffern gebildete Wort in der Dezimalbruchentwicklung von  $\pi$  vorkommt. . . Wenn das stimmt, dann ist  $g$  die Funktion, die überall den Wert 1 annimmt.

## Berechenbarkeitsbeispiele

e) Berechnung einer totalen Funktion  $h : \mathbb{N} \rightarrow \mathbb{N}$  mit

- $h(n) = 1$ , falls die 7 mindestens  $n$  mal hintereinander in der Dezimalbruchentwicklung von  $\pi$  vorkommt,
- 0 sonst.

Klar: Entweder ist  $h$  konstant Eins oder es gibt ein  $n_0$ , sodass die Sieben zwar  $n_0$ -mal hintereinander vorkommt, aber nicht  $n_0 + 1$ -mal.

Für jede dieser Möglichkeiten ist  $h$  berechenbar.

Welche Möglichkeit die Richtige ist, ist unbekannt.

Anmerkung: analog zu Teil c) setze für  $r \in \mathbb{R}$

- $f_r(n) = 1$ , falls  $n$  Anfangsabschnitt der Dezimalbruchentwicklung von  $r$  ist,
- $f_r(n) = 0$ , sonst.

Aber:

- Es gibt überabzählbar viele reelle Zahlen...
- Es gibt nur abzählbar viele (JAVA-)Algorithmen...

⇒ Die Dezimalbruchentwicklungen fast aller Zahlen kann man nicht berechnen!

$f_r$  ist im Allgemeinen nicht berechenbar!

## Churchsche These

Die durch Turingmaschinen berechenbaren Funktionen

*(gleichbedeutend: WHILE-/GOTO-berechenbar oder  $\mu$ -rekursiv)*

sind genau die im intuitiven Sinn berechenbaren (partiellen) Funktionen auf  $\mathbb{N}$ .

Beweisbar schwächere Berechenbarkeitsbegriffe liefern sog. LOOP-Programme (äquivalenter Begriff: primitiv-rekursiv).

## Nochmals: Turingmaschinen

Bisher: Turingmaschinen als Akzeptoren für Typ-0-Sprachen

Jetzt: Turingmaschinen zur Berechnung von Funktionen

$$f : E^* \dashrightarrow E^* \text{ bzw. } f : \mathbb{N}^k \dashrightarrow \mathbb{N}$$

Notwendig: Notation der Zahleneingaben...

- Binärnotation  $\text{bin}(n)$ , z.B.  $\text{bin}(13) = 1101$  und  $\text{bin}(0) = 0$  oder
- Unärnotation:  $n$  durch  $n$  Zeichen, z.B. 13 durch 1111111111111
- Eingabe von Vektoren: mit Trennsymbol (z.B. Blank oder #) zwischen den Zahlen
- zu Beginn: Lese-Kopf steht über dem ersten Eingabe-Zeichen (von links)
- am Ende: Ausgabe steht auf dem Band, Kopf steht links über dem ersten Zeichen

## Der formalisierte Berechenbarkeitsbegriff

Eine Funktion  $f : E^* \dashrightarrow E^*$  heißt *Turing-berechenbar*, falls es eine deterministische Turingmaschine  $TM$  gibt derart, dass für  $x, y \in E^*$  genau dann  $f(x) = y$  gilt, wenn es einen Endzustand  $s' \in F$  und Worte  $u, v$  gibt mit

$$s_0 x \vdash_{TM}^* u s' v$$

und  $y$  das längste Präfix von  $v$  über  $E$  ist.

Eine Funktion  $f : \mathbb{N}^k \dashrightarrow \mathbb{N}$  heißt Turing-berechenbar, falls es eine deterministische Turingmaschine  $TM$  gibt derart, dass für  $n_1, \dots, n_k, m$  aus  $\mathbb{N}$  genau dann  $f(n_1, \dots, n_k) = m$  gilt, wenn es einen Endzustand  $s' \in F$  und Worte  $u, v$  gibt mit

$$s_0 \text{bin}(n_1) \square \text{bin}(n_2) \square \dots \square \text{bin}(n_k) \vdash_{TM}^* u s' v$$

und  $\text{bin}(m)$  das längste Präfix von  $v$  über  $\{0, 1\}$  ist.  
Dabei sei  $\text{bin}(n)$  die Binärdarstellung von  $n$ .

## Beispiele

a) Wir haben bereits früher gesehen, dass die folgende *Nachfolger-Funktion*  $S$  (bezüglich der Binärdarstellung) berechenbar ist:

$$f(n) = S(n) = n + 1$$

b) Die nirgends definierte Funktion  $f : \mathbb{N}^k \dashrightarrow \mathbb{N}$  ist ebenfalls berechenbar.

Passende Turingmaschine:

- Startzustand  $s_0$ ,
- Endzustand  $s_1 \neq s_0$ ,
- Übergänge  $\delta(s_0, a) = (s_0, a, N)$  für alle  $a \in A$

## Nachtrag Formale Sprachen 1

- Typ-0-Sprachen = Sprachen, die von n.det. TM erkannt werden.
- n.det. TM durch det. TM simulierbar
- d.h. L ist vom Typ 0, wenn es eine det. TM gibt mit  $L = L(TM)$

Dabei:  $x \in L(TM) \Leftrightarrow$  von  $s_0x$  ist  $us_fv$  mit  $s_f \in F$  erreichbar.

Verhalten der TM nach Erreichen von  $s_f$  für Akzeptieren unwichtig!

Daher: Turingmaschinen so definiert, dass sie nach Erreichen eines Endzustandes nicht weiterrechnen (können).

Jede TM ist so modifizierbar, dass sie nur mit  $\dots \square_{s_f} \square \dots$  akzeptiert...

$\leadsto$  **Satz:** Eine Sprache L ist genau dann vom Typ 0,

wenn die folgende Funktion  $g : E^* \dashrightarrow E^*$  berechenbar ist:

$$g(x) = \begin{cases} \lambda & \text{für } x \in L \\ \text{undefiniert} & \text{für } x \notin L \end{cases}$$

## Nachtrag Formale Sprachen 2

Bei  $x \notin L(TM)$  wird *nie* ein Endzustand  $s_f$  erreicht, mögliche Gründe:

1. Die Maschine rechnet endlos oder
2. die Maschine 'bleibt stecken'  
(d.h. zu  $s$  und  $a$  kein Übergang  $\delta(s, a)$  definiert)

Fall (2) vermeidbar mit Modifikation  $\delta'$  von  $\delta$ :

- neuer Zustand  $s_l$  mit  $\delta'(s_l, a) = (s_l, a, N)$  für alle  $a \in A$
- $\delta'(s, a) = (s_l, a, N)$ , falls  $\delta(s, a)$  nicht definiert (für  $s \in S \setminus F$ )

$\leadsto$  **Satz:** Eine Sprache  $L$  ist genau dann vom Typ 0, wenn es eine deterministische Turingmaschine gibt, die genau bei den Wörtern aus  $L$  (als Eingabe) *anhält*.

## Mehrband-Turingmaschine $MTM$

- mit  $k \geq 1$  Bändern und Schreibköpfen
- alle Schreibköpfe können unabhängig voneinander operieren
- Übergangsfunktion  $\delta : S \times A^k \rightarrow S \times A^k \times \{L, R, N\}^k$
- Definition von Konfiguration und Übergangsrelation passend dazu...

**Satz:** Zu jeder Mehrbandmaschine  $MTM$  gibt es eine Einbandmaschine  $TM$ , die dieselbe Sprache akzeptiert, bzw. dieselbe Funktion berechnet.

## Beweisidee für die Simulation von Mehrbandmaschinen

**Gegeben:** Mehrbandmaschine  $MTM$  mit  $k$  Bändern (1 bis  $k$ ) und Arbeitsalphabet  $A$ .

**Aufgabe:** Konstruiere Turingmaschine  $TM$ , die  $MTM$  simuliert:

- Schreib-Leseband der  $TM$  in  $2k$  *Spuren* (1 bis  $2k$ ) unterteilt
- Spur  $2i-1$  enthält Inhalt von Band  $i$  der  $MTM$
- Spur  $2i$  hat an genau einer Stelle das Zeichen  $*$ , ansonsten nur Blanks.
- Das Zeichen  $*$  zeigt an, dass der Lese-Schreibkopf des  $i$ -ten Bandes der  $MTM$  gerade an dieser Stelle steht.
- Kopf der  $TM$  testet alle Kopfpositionen der  $MTM$  nacheinander
- Neues Arbeitsalphabet  $A' = A \cup (A \cup \{*\})^{2k}$ .

## Arbeitsweise der Einband-TM

- Gestartet wird mit der Eingabe  $a_1 \dots a_n \in E^*$ .
- Zunächst erzeugt TM die Startkonfiguration von MTM:
  - Eingabewort  $a_1 \dots a_n$  auf Spur 1
  - alle anderen ungeraden Spuren sind leer
  - alle geraden Spuren haben \* unter dem Lese-Schreibkopf
- In Spuren unterteilter Bereich des Bandes ist in Blanks eingeschlossen  
⇒ TM kann stets durch Lauf über alle beschriebenen Zellen Information über den nächsten Schritt der MTM sammeln und ihn simulieren
- jeder Einzelschritt der MTM wird durch mehrere (=viele) Schritte der TM simuliert.

## Vorteile von Mehrbandmaschinen

- Jedes Band stellt ein unendlich langes Register dar.
- Besitzt eine Maschine nur eines dieser Bänder, ist sie meist damit beschäftigt, auf diesem Band hin- und herzufahren...
- Mehrbandmaschine: Daten auf Bänder verteilt, mit leichterem Zugriff

Z.B.: Simulation einer Einbandmaschine mit k-Band-TM auf Band  $i$  ohne Änderung der anderen Bänder:

Übergangsfunktion dazu z.B. statt  $\delta(s, c) = (s', d, R)$  jetzt

$$\delta'(s, a_1, \dots, c, \dots, a_k) = (s', a_1, \dots, d, \dots, a_k, N, \dots, R, \dots, N)$$

## Programmieren mit Turingmaschinen

Erinnerung: Inkrementieren von Zahlen mit Einbandmaschine

- Analog MTM, die Inhalt von Band  $i$  inkrementiert  
Schreibweise:  $\text{Band } i := \text{Band } i + 1$
- Analog: MTM, die Zahlen dekrementiert,  
d.h. aus  $n > 0$  wird  $n - 1$ , Dekrement von  $0$  sei wieder  $0$   
Schreibweise:  $\text{Band } i := \text{Band } i - 1$
- Weitere naheliegende Abkürzungen:  $\text{Band } i := 0$  und  $\text{Band } i := \text{Band } j$

Hier Verzicht auf Angabe der Turingtabellen!

## Programmieren mit Turingmaschinen

Hintereinanderschaltung von Turingmaschinen:

Gegeben

$$M_1 = (S_1, E, A_1, \delta_1, s_{01}, \square, F_1)$$

$$M_2 = (S_2, E, A_2, \delta_2, s_{02}, \square, F_2)$$

O.B.d.A.  $S_1 \cap S_2 = \emptyset$

Schreibweise:  $M_1; M_2$  für

$$M = (S_1 \cup S_2, E, A_1 \cup A_2, \delta, s_{01}, \square, F_2)$$

mit

$$\delta = \delta_1 \cup \delta_2 \cup \{((s_f, a), (s_{02}, a, N)) \mid s_f \in F_1, a \in A_1\}$$

(Hier: Übergangsfunktionen als Relationen über kartesischen Produkten)

Beispiel: Die durch

Band  $i :=$  Band  $i + 1$ ;

Band  $i :=$  Band  $i + 1$ ;

Band  $i :=$  Band  $i + 1$

beschriebene Maschine realisiert die Operation ‘Band  $i :=$  Band  $i + 3$ ’.

Alternative Darstellung durch gerichteten Graphen, der Befehle wie “Band  $i :=$  Band  $i + 1$ ” als Knotenbeschriftungen trägt.

Hiermit lassen sich auch *bedingte Verzweigungen* leicht notieren, wenn man Endzustände als Kantenbeschriftungen zulässt.

## Bedingte Verzweigungen

Beispiel: Test, ob 0 (als Zahl) auf dem Band steht, und bedingte Verzweigung

Band = 0?

Zustandsmenge  $S := \{s_0, s_1, ja, nein\}$

Endzustände ja und nein

Arbeitsalphabet  $A = \{0, 1, \square\}$ .

Übergangsfunktion:

$\delta$	1	0	$\square$
$s_0$	(nein, 1, N)	( $s_1$ , 0, R)	(ja, $\square$ , N)
$s_1$	(nein, 1, L)	( $s_1$ , 0, R)	(ja, $\square$ , L)

Analog: Band  $i = 0$ ?

## Schleifen

Simulation einer WHILE-Schleife:

WHILE Bed  $i \neq 0$  DO  $M$

kann man durch wie soeben beschriebenes Testen realisieren,  
gefolgt von möglichen Ausführungen von  $M$   
und nochmaligem Testen der Bedingung.

Somit haben wir die Grundbestandteile des *imperativen Programmierparadigmas* beisammen:

Zuweisungen, Hintereinanderausführung, bedingtes Verzweigen und Schleifen  
~> stärkt Gefühl, dass TM Berechenbarkeit “richtig” modellieren

## LOOP-Berechenbarkeit

### Syntaktische Grundbausteine von LOOP-Programmen:

- Variablen:  $x_0 x_1 x_2 \dots$
- Konstanten:  $0 1 2 \dots$   
(also für *jede* natürliche Zahl eine Konstante)
- Trennsymbole und Operationszeichen:  $;$   $:=$   $+$   $-$
- Schlüsselwörter: `LOOP DO END`

*LOOP-Programme* sind dann:

- Jede Wertzuweisung  $x_i := x_j + c$  und  $x_i := x_j - c$  ist ein LOOP-Programm.  
(Achtung: kleinste Zahl ist die Null, also modifizierte Subtraktion)
- Sind  $P_1$  und  $P_2$  LOOP-Programme,  
dann auch  $P_1; P_2$
- Ist  $x_j$  Variable und  $P$  LOOP-Programm,  
dann auch `LOOP  $x_i$  DO  $P$  END`
- Weitere Konstruktionen sind nicht zugelassen.

## Semantik von LOOP-Programmen

- Die Werte der Variablen  $x_i$  sind (beliebig) aus  $\mathbb{N}$ .
- Bei  $x_i := x_j + c$  erhält  $x_i$  den Wert  $x_j + c$   
analog zu üblichen Programmiersprachen...
- Bei  $x_i := x_j - c$  erhält  $x_i$  den Wert  $x_j - c$ , falls  $x_j \geq c$ ,  
ansonsten den Wert 0. (Damit bleiben alle Werte in  $\mathbb{N}$ .)
- Bei  $P_1; P_2$  wird erst  $P_1$  ausgeführt, dann  $P_2$ .
- Bei `LOOP  $x_i$  DO P END` wird das Programm  $P$  so oft ausgeführt, wie der Wert von  $x_i$  *zu Beginn*, also beim vor dem ersten Schleifeneintritt, angibt.

## LOOP-Berechenbarkeit

Eine Funktion  $f : \mathbb{N}^k \rightarrow \mathbb{N}$  (oder vorsichtiger  $f : \mathbb{N}^k \dashrightarrow \mathbb{N}$ ) heißt LOOP-berechenbar, falls

- es ein LOOP-Programm  $P$  gibt, so dass  $P$ ,
- gestartet mit  $n_1, \dots, n_k$  aus  $\mathbb{N}$  in den Variablen  $x_1, \dots, x_k$
- und 0 in allen anderen Variablen
- mit dem Wert  $f(n_1, \dots, n_k)$  in der Variablen  $x_0$  stoppt.

## LOOP-Berechenbarkeit: Beispiele 1

- Zuweisung ' $x_i := x_k$ ' durch ' $x_i := x_k + 0$ ' mit der Konstanten  $c = 0$
- Zuweisung ' $x_i := c$ ' durch ' $x_i := x_k + c$ ' für eine Variable  $x_k$ , die immer Wert 0 hat.
- Bedingte Ausführung ' $\text{IF } x_k = 0 \text{ THEN } Q \text{ END}$ ' durch

```
y := 1;  
LOOP x_k DO y := 0 END;  
LOOP y DO Q END
```

mit einer ansonsten nicht benutzten Variablen  $y$ .

- Bedingte Ausführung ' $\text{IF } x_k = 0 \text{ THEN } Q \text{ ELSE } R \text{ END}$ ' als Übungsaufgabe...

## LOOP-Berechenbarkeit: Beispiele 2

- Additionen ' $x_i := x_j + x_k$ ' mit  $i \neq j$ :

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

Spezialfall ' $x_0 := x_1 + x_2$ ':

berechnet Addition als Funktion  $+$  :  $\mathbb{N}^2 \rightarrow \mathbb{N}$ .

- Der Fall  $i = j$ , d.h. ' $x_i := x_i + x_k$ ', ist sogar noch einfacher:

```
    LOOP  $x_k$  DO  $x_i := x_i + 1$  END
```

Subtraktion ' $x_i := x_j - x_k$ ': ähnlich...

(wieder mit Wert 0 bei  $x_j < x_k$ )

## LOOP-Berechenbarkeit: Beispiele 3

- Multiplikationen ' $x_i := x_j \cdot x_k$ '

```
 $x_i := 0;$   
LOOP  $x_j$  DO  $x_i := x_i + x_k$  END
```

was wiederum nur eine Abkürzung ist für

```
 $x_i := 0;$   
LOOP  $x_j$  DO  
    LOOP  $x_k$  DO  $x_i := x_i + 1$  END  
END
```

## LOOP-Berechenbarkeit: Beispiele 4

- Die *Signum-Funktion*  $sg : \mathbb{N} \rightarrow \mathbb{N}$  mit

$$sg(x) := \begin{cases} 1 & \text{falls } x > 0 \\ 0 & \text{falls } x = 0 \end{cases}$$

ist realisierbar über:

```
x0 := 1;  
IF x1 = 0 THEN x0 := 0 END
```

- Inverse Signum-Funktion  $\overline{sg}(x) := 1 - sg(x)$  ähnlich ...
- Die *Gleichheits-Funktion*  $se(x, y)$  mit

$$se(x, y) := \begin{cases} 1 & \text{falls } x = y \\ 0 & \text{falls } x \neq y \end{cases}$$

kann über die Funktionen  $sg$  und  $\overline{sg}$  realisiert werden.

## LOOP-Berechenbarkeit: Beispiele 5

- Ganzzahldivision  $x_i := x_j \text{ DIV } x_k$  mit  $x_k > 0$ :

Dazu zunächst 'IF  $x_k \leq x_j$  THEN Q END' über

```
x0 := xk - xj;  
x1 :=  $\overline{sg}(x_0)$ ;  
LOOP x1 do Q END
```

Damit erhalten wir ' $x_i := x_j \text{ DIV } x_k$ ' durch:

```
xi := 0;  
LOOP xj DO  
  IF xk ≤ xj THEN xi := xi + 1 END;  
  xj := xj - xk;  
END
```

- Zur Übung realisieren Sie z.B. die Modulo-Funktion MOD als LOOP-Programm (wird später noch benötigt...).