Technische Universität München
Prof. J. Esparza / Dr. M. Luttenberger, S. Sickert

Sommer 2016
3. Juli 2016

HA-Lösung

TA-Lösung

Einführung in die theoretische Informatik – Aufgabenblatt 10

Beachten Sie: Soweit nicht explizit angegeben, sind Ergebnisse stets zu begründen!

Hausaufgaben: Abgabe bis zum 29.06.2016 (Mittwoch) um 12:00

Hinweis: Bei allen Aufgaben zu Turingmaschinen wird erwartet, dass zunächst umgangssprachlich das Verhalten der TM beschrieben wird. Desweiteren müssen die Transitionen einer TM stets graphisch entsprechend den Vorlesungsfolien dargestellt werden.

Aufgabe 10.1 — 5P

Geben Sie eine nichtdeterministische 1-Band-TM mit $\Sigma = \{a, b\}$ an, die nichtdeterministisch die Eingabe permutiert, genauer: Ist w die Eingabe, dann terminiert die TM auf jedem Rechenpfad, und zu jeder Permutation w' von w gibt es mindestens eine Rechnung der TM, an deren Ende der Bandinhalt gerade w' ist.

Beispiel: Auf Eingabe $w = aabb$ soll die TM für jede der möglichen Permutation $w' \in \{abab, abba, baba, bbaa, baab, aabb\}$ jeweils mindestens eine Berechnung besitzen, an deren Ende genau w' auf dem Band steht. Beachten Sie, dass die TM stets terminieren muss.

Lösung:

- Merke ersten Buchstaben in Kontrolle

$$p \xrightarrow[x\in\Sigma]{x/x,N} p_x$$

- Laufe nicht deterministich zu einem beliebigen Buchstaben (immer nach rechts, nicht deterministisch Stopp samt Zustandswechsel), falls rechtes Ende erreicht wird, Terminieren samt Löschen von Markierungen

$$p_x \xrightarrow[y\in\Sigma]{y/y,R} p_x \qquad p_x \xrightarrow[y\in\Sigma]{y'/y',R} p_x \qquad p_x \xrightarrow[y\in\Sigma]{y/y,N} q_x \qquad p_x \xrightarrow{\square/\square,L} s$$

$$s \xrightarrow[y\in\Sigma]{y'/y,L} s \qquad s \xrightarrow[y\in\Sigma]{y/y,L} s \qquad s \xrightarrow{\square/\square,R} s$$

- Solange der aktuelle Buchstabe nicht markiert ist:

  Schreibe gespeicherten Buchstabe an aktuelle Position samt Markierung und merke überschriebenen Buhstaben in Kontrolle

$$q_x \xrightarrow[y\in\Sigma]{y/x',L} r_y$$

  laufe nach ganz links, suche nicht deterministich nach einem noch nicht markierten Buchstaben

$$r_x \xrightarrow[y\in\Sigma]{y'/y'|y/y,L} r_x \qquad r_x \xrightarrow{\square/\square,R} p_x$$

  und wiederhole Vorgang.

Aufgabe 10.2 **5P**

Sei $\mathsf{A} = \{a, b\}$, $\overline{\mathsf{A}} = \{\overline{a}, \overline{b}\}$ und $\Sigma = \mathsf{A} \cup \overline{\mathsf{A}}$. Ein Wort $ux\overline{x}v$ mit $u, v \in \Sigma^*$ und $x \in \mathsf{A}$ kann zu uv reduziert werden, z.B. kann $ab a\overline{a}\overline{b}a$ zu $ab\overline{b}a$ und weiter zu aa reduziert werden. $\overline{a}a$ kann jedoch nicht zu ε reduziert werden.

Geben Sie eine 1-Band-TM an, die eine Eingabe w maximal reduziert, d.h. nach Terminierung soll auf dem Band ein Wort w' stehen, so dass w' aus w durch eine endliche Anzahl von Reduktionen hervorgeht, w' selbst jedoch nicht weiter reduzierbar ist.

Hinweis: Gehen Sie bei der Aufgabe modular vor. Geben Sie verschiedene TMs für Teilfunktionen an und wie diese kombiniert werden.

Lösung: Wir konstruieren zwei TMs für die Unterfunktionen.

- TM M_1 zum Erkennen von Reduktionsstellen.

  Intuition: M_1 sucht von Beginn bis Ende auf dem Band nach möglichen Reduktionsstellen ($a\overline{a}$ oder $b\overline{b}$). Falls so eine Stelle gefunden wird, wird sie mit X markiert. M_2 ausgeführt. Falls keine Stellen gefunden werden terminiert M_1 und M_2 wird ausgeführt.

  q_ε ist dann der Startzustand und q_f ist der Endzustand. (d - delete, s - seek)

$$
\begin{array}{llllll}
q_\varepsilon \xrightarrow{a/a,R} q_a & q_\varepsilon \xrightarrow{b/b,R} q_b & q_\varepsilon \xrightarrow[y\in\overline{\mathsf{A}}\cup\{X\}]{y/y,R} q_\varepsilon & q_\varepsilon \xrightarrow{\square/\square,L} s & & \\
q_a \xrightarrow{a/a,R} q_a & q_a \xrightarrow{X/X,R} q_a & q_a \xrightarrow{b/b,R} q_b & q_a \xrightarrow{\overline{a}/X,L} d & q_a \xrightarrow{\overline{b}/\overline{b},R} q_\varepsilon & q_a \xrightarrow{\square/\square,L} s \\
q_b \xrightarrow{a/a,R} q_a & q_b \xrightarrow{X/X,R} q_b & q_b \xrightarrow{b/b,R} q_b & q_b \xrightarrow{\overline{a}/\overline{a},R} q_\varepsilon & q_b \xrightarrow{\overline{b}/X,L} d & q_b \xrightarrow{\square/\square,L} s \\
d \xrightarrow{X/X,L} d & d \xrightarrow[y\in\mathsf{A}]{y/X,L} q_\varepsilon & & & & \\
s \xrightarrow[y\in\mathsf{A}\cup\overline{\mathsf{A}}\cup\{X\}]{y/y,L} s & s \xrightarrow{\square/\square,R} q_f & & & &
\end{array}
$$

- TM M_2 löscht alle X vom Band und verschiebt die entstehenden Teile nach vorne.

  p ist dann der Startzustand und p_f ist der Endzustand. (c - copy, s - seek, t - remove trailing X)

$$
\begin{array}{lllll}
p \xrightarrow[x\in\mathsf{A}\cup\overline{\mathsf{A}}]{x/x,R} p & p \xrightarrow{X/X,R} c & p \xrightarrow{\square/\square,L} s & & \\
c \xrightarrow[x\in\mathsf{A}\cup\overline{\mathsf{A}}]{x/X,L} c_x & c \xrightarrow{X/X,R} c & c \xrightarrow{\square/\square,L} t & & \\
c_x \xrightarrow{X/X,L} c_x & c_x \xrightarrow[x\in\mathsf{A}\cup\overline{\mathsf{A}}\cup\{\square\}]{x/x,R} c'_x & c'_x \xrightarrow{X/x,N} p & & \\
t \xrightarrow{X/\square,L} t & t \xrightarrow[x\in\mathsf{A}\cup\overline{\mathsf{A}}\cup\{\square\}]{x/x,N} s & & & \\
s \xrightarrow[y\in\mathsf{A}\cup\overline{\mathsf{A}}]{y/y,L} s & s \xrightarrow{\square/\square,R} p_f & & &
\end{array}
$$

Aufgabe 10.3 **5P**

Sei $f\colon \{0,1\}^* \to \{0,1\}^*$ eine Funktion. Der Graph von f ist dann $G_f = \{x\#f(x)\colon x \in \{0,1\}^*\}$.

Zeigen Sie: f ist Turing-berechenbar genau dann, wenn G_f entscheidbar ist.

Hinweis: Es reicht jeweils unter Verwendung der Resultate aus der Vorlesung zu beschreiben, wie man unter Verwendung einer Turing-Maschine für das eine Problem eine Turing-Maschine für das andere Problem konstruieren kann.

Lösung: Skizze:

- $f \to G_f$: Von der Eingabe $x\#y$ wird x auf ein separates Band (2-Band TM) geschrieben und die TM für f auf x ausgeführt. Überprüfe dann, ob $y = f(x)$ gilt.
- $G_f \to f$: Rate nicht-deterministisch $f(x)$ und schreibe $x\#f(x)$ auf ein separates Band. Überprüfe dies mit der TM für G_f und überschreibe x mit $f(x)$ gegebenenfalls.

Aufgabe 10.4 **5P**

Eine 1-Band-k-Kopf-TM unterscheidet sich von einer gewöhnlichen 1-Band-TM darin, dass es $k > 0$ Köpfe gibt, die unabhängig von einander bewegt werden können. Die Köpfe sind dabei von 1 bis k durchnummeriert. In jedem Schritt lesen alle Köpfe den Inhalt der jeweiligen Zelle, auf der sie gerade stehen. Das gelesene k-Tupel zusammen mit dem aktuellen Zustand der TM bestimmt dann die Transition wie gewöhnlich. Hierbei kann jeder Kopf den Inhalt der Zelle, die er gelesen hat, überschreiben. Die Köpfe schreiben dabei in der Reihenfolge, in der sie durchnummeriert sind, d.h. zuerst schreibt Kopf 1, dann Kopf 2, usw. ganz zu Schluss schreibt Kopf k. Effektiv bedeutet dies, schreiben mehrere Köpfe in dieselbe Bandzelle, so bestimmt der Kopf mit der höchsten Nummer, welches Zeichen tatsächlich geschrieben wird.

Zeigen Sie: 1-Band-k-Kopf-TM sind äquivalent zu 1-Band-TM. Skizzieren Sie hierfür – u.U. in mehreren Schritten und unter Verwendung der Resultate aus der Vorlesung – wie sich eine gegebene 1-Band-k-Kopf-TM in eine äquivalente 1-Band-TM übersetzen lässt. Äquivalent bedeutet hier, dass beide TM dieselbe Sprache akzeptieren sollen.

Lösung: Wir simulieren die 1-Band-k-Kopf-TM M mit einer $k+1$-Band-TM M'. Da k-Band-TMs von 1-Band-TMs simuliert werden können, folgt somit das gewünschte Ergebnis.

Wir gehen hierbei wie folgt vor: Auf dem ersten Band steht die Eingabe w und auf dem $i+1$-tem Band vermerken wir die Position des i-ten Kopfes mit einem $*$. Die TM M wird mit folgenden Schritten simuliert:

(a) Laufe die Bänder von links nach rechts ab und speichere die Zeichen auf dem die Köpfe stehen in einem Zustand der Form $(q, a_1, a_2, \dots a_k)$.

(b) Wenn alle Zellen, auf denen ein Kopf steht, gelesen worden sind, bestimme den Folgezustand und speichere die resultierenden Schreiboperationen und die Kopfbewegungen im Zustand.

(c) Wende diese Änderungen beginnend mit Kopf 1 an.

(d) Falls der erreichte Zustand ein Endzustand ist, terminiere. Falls nicht, gehe zu (a).

Tutoraufgaben: Besprechung in KW26

Aufgabe 10.1

Geben Sie für jede der folgenden Funktionen sowohl ein WHILE-Programm als auch ein GOTO-Programm an, das die jeweilige Funktion implementiert:

(a) $f(x_1, x_2) = x_1 \bmod x_2$
Hinweise: Verwenden Sie hier nur das Grundschema für WHILE- und LOOP-Programme. Nur $x_i := x_j + x_k$ und $x_i := x_j - x_k$ sind zusätzlich erlaubt.

(b) $g(x_1) = \begin{cases} \max\{k \in \mathbb{N} \colon \frac{x_1}{2^k} \in \mathbb{N}\} & \text{falls } x_1 > 0 \\ \bot & \text{falls } x_1 > 0 \end{cases}$

(c) $h(x_1, x_2) = (2x_1 + 1) \cdot 2^{x_2}$

Halten Sie sich an die Konventionen aus der Vorlesung: Soll ein WHILE-Programm eine Funktion $F \colon \mathbb{N}^k \to \mathbb{N}$ berechnen, so werden die Variablen $x_1, \ldots, x_k$ entsprechend mit den Eingabewerten $n_1, \ldots, n_k \in \mathbb{N}$ initialisiert, während alle anderen Variablen zu Beginn auf 0 initialisiert werden. Nach Terminierung speichert die Variable x_0 den Funktionswert $F(n_1, \ldots, n_k)$.

Erinnerung: $0 \in \mathbb{N}$

Lösung:

(a) Zur besseren Lesbarkeit:

Eingaben in x für x_1 und ggf. y für x_2,

Rückgabe in z für x_0

Kleinbuchstaben für sonstige Hilfsvariaben (nach Konvention zu Beginn mit 0 initialisiert), beliebig dann den Bezeichnern x_i zuordnen z.B. durch Durchnummerieren.

```
z := x - y;
WHILE z ≠ 0 DO
   x := z;
   z := z - y;
END;
z := x + 1;
z := z - y;
IF z = 0 DO
   z := x
ELSE
   z := 0
END
```

(b) Ab jetzt MOD als zusätzlicher Befehl. Strikt nach Vorlesungsfolien müsste man z.B. die in obigem Programm verwendeten Variablenbezeichner disjunkt von den im folgenden Progamm verwendeten Bezeichnern wählen (z.B. durch x' statt x usw.) und noch zusätzlich Code einbauen (z.B. $x' := x$ und $u := z'$), der die Werteübergabe übernimmt.

```
z := 1;
y := 2;
b := 1;
WHILE b ≠ 0 DO
   u := x MOD y;
   IF u = 0 DO //y = 2^z mit x/2^z ∈ ℕ, teste, ob auch x/2^(z+1) ∈ ℕ
      z := z + 1;
      u := y; // Konvention aus Folien: x_i := x_j + x_k nur fuer i ≠ k gestattet.
      y := u + u;
   ELSE //x/2^z ∉ ℕ, aber x/2^(z-1) ∈ ℕ
      b := 0; //beende While-Schleife
      z := z - 1;
   END
END
```

(c) h :

```
z := y + y;
z := z + 1;
y := 1;
WHILE x ≠ 0 DO
   u := y;
   y := u + u;
   x := x - 1;
END;
u := z;
z := y * u;
```

Aufgabe 10.2

Wir erweitern GOTO-Programme um die Möglichkeit, Variablenwerte auf einem globalen Stack zwischenzuspeichern. Mittels der Anweisung **PUSH** x_i wird der aktuelle Wert der Variable x_i auf den Stack gelegt, mittels $x_i :=$ **POP** wird der aktuell oberste Wert vom Stack genommen und in der Variable x_i gespeichert – sollte der Stack aktuell keine Werte enthalten, wird x_i auf 0 gesetzt. Ob der Stack aktuell einen Wert enthält, kann mittels $x_i :=$ **EMPTY** erfragt werden, wobei x_i auf 1 gesetzt wird, falls der Stack leer ist, ansonsten wird x_i auf 0 gesetzt.

Beispiel:

```
M1: IF x1 = 0 GOTO M2;
    IF x1 = 1 GOTO M3;
    x1 := x1 − 1;
    PUSH x1;
    x1 := x1 − 1;
    GOTO M1;
M2: x0 := x0 + 0;
    GOTO M4;
M3: x0 := x0 + 1;
    GOTO M4;
M4: x1 := EMPTY;
    IF x1 = 1 GOTO M5;
    x1 := POP;
    GOTO M1;
M5: HALT
```

(a) Bestimmen Sie die Funktion $\mathbb{N} \to \mathbb{N}$, die von Programm aus dem Beispiel berechnet wird.

(b) Skizzieren Sie, wie sich jedes GOTO-Programm mit Stack in ein GOTO-Programm ohne Stack übersetzen lässt.

Lösung:

(a) Fibonacci-Zahlen mit $F_0 = 0, F_1 = 1, F_{n+2} = F_{n+1} + F_n$

(Beweisskizze)

Induktionsanfang:

Rechenpfad zu $x_1 = 0$: $M_1 \to M_2 \to M_4 \to M_5$ mit $x_0 = 0$ (nach Konvention alle Variablen außer Eingabevariablen auf 0 initialisiert).

Rechenpfad zu $x_1 = 1$: $M_1 \to M_3 \to M_4 \to M_5$ mit $x_0 = 1$

Induktionsschritt: $n \geq 2$ beliebig fixiert.

Annahme: Programm berechnet F_k für alle $k < n$.

Erster Durchlauf $M_1 \to M_1$: $n - 1$ wird auf Stack gepushed, es gilt $x_1 = n - 2$ bei zweitem Besuch von M_1.

Betrachte Berechnung β_{n-2} von Programm auf Eingabe $n - 2$. Nach Annahme terminiert Programm, d.h. erreicht M_5 mit $x_0 = F_{n-2}$. Sei β'_{n-2} der Präfix von β_{n-2} bis einschließlich dem letzten Test, ob der Stack leer ist.

In der Berechnung zu n wird nach einmaliger Schleife $M_1 \to M_1$ die Berechnung β'_{n-2} ausgeführt. Im Gegensatz zu β_{n-2} ist der Stack nicht mehr leer, weswegen statt dem Sprung zu M_5 der Stack gepushed und damit x_1 auf $n - 1$ gesetzt wird, bevor wieder zu M_1 gesprungen wird. Danach ist man in der Situation einer Berechnung zur Eingabe $n - 1$ bei anfänglich leerem Stack mit dem einzigen Unterschied, dass x_0 bereits den Wert F_{n-2} hat.

Da das Programm stets nur 0 oder 1 zu dem Wert von x_0 hinzuaddiert, folgt, dass der Durchlauf zu $n-1$ dann den Wert von x_0 noch zusätzlich um F_{n-1} erhöht, womit dann bei Terminierung x_0 den Wert $F_n = F_{n-1} + F_{n-2}$ enthält.

(b) Simulieren von Stack in einer Variablen z.B. mittels Iterieren von Cantorscher Paarungsfunktion oder mit Funktionen aus TA10.1:

Leerer Stack: $s = 0$

Test auf leerer Stack: **IF** s = 0 **DO** z := 1; **ELSE** z := 0; **END**.

Push von x auf Stack s: $s := h(x, s)$

Top von Stack s: $t := \frac{s}{2^{g(s)}}$

Pop von Stack: $s := g(s)$

Beispiel:

- $s = 0$ zu Beginn
- Push 3: $s = 7 \cdot 2^0 = 7$
- Push 2: $s = 5 \cdot 2^7 = 640$

- Top: $\frac{640}{2^7} = 5$
- Pop: $s = 7$
- Pop: $s = 0$

Aufgabe 10.3

Wir betrachten vereinfachte JAVA/C/C++-Programme:

Der einzige Datentyp ist $\mathbb{N}$ ("BigInteger"), der allerdings per *call-by-value* analog zu **int** bei Funktionsaufrufen übergeben wird. Innerhalb von Funktionen sind nur die üblichen arithmetischen Operationen x+=1;x+=y;x=y+z;x=2*z;..., if-Anweisungen der Form **if**(x == n){ ... } **else** { ... } (nur Test auf eine Konstante $n \in \mathbb{N}$) und return-Anweisungen der Form **return** x; (nur eine Variable, keine Terme) erlaubt, sonst sind keine weiteren Anweisungen erlaubt, insbesondere kein new, **for**(;;) {}, **while**() {} etc.

Mit *main* sei die Hauptfunktion bezeichnet, welche die von dem Programm berechnete Funktion bestimmt.

Beispiel:

```
ℕ main(ℕ x) {
  ℕ y;
  y = f(x,0);
  return y;
}

ℕ f(ℕ x, ℕ y) {
  if( x == 0 )
    return y;
  if( x == 1 ) {
    y += 1;
    return y;
  }
  x -= 1;
  y = f(x,y);
  x -= 1;
  y = f(x,y);
  return y;
}
```

Skizzieren Sie, wie sich jedes solche Programm in ein GOTO-Programm mit Stack (siehe TA10.2) übersetzen lässt.

Lösung:

Jeden Funktionsstart versieht man mit einer Sprungmarke E_i plus ggf. Code, der die lokalen Variablen initialisiert, falls nötig.

Vor dem Aufruf einer Funktion sichert man auf dem Stack alle Variablenwerte, welche für die aufrufende Funktion lokal sind, kopiert die Variablenwerte ggf. noch in die entsprechende Eingabevariablen der aufgerufenen Funktion, um dann zu der entsprechenden Marke zu springen.

Nach jedem Funktionsaufruf baut man entsprechend zusätzlichen Code ein, der die Werte der lokalen Variablen mittels dem Stack wiederherstellt zzg. Rückgabwert. Diesen Code versieht man mit einem Label R_i, dessen Index man nach Sicherung der lokalen Variablen noch zusätzlich auf den Stack pushed.

Jede return-Anweisung wird nun durch ein Code ersetzt, welcher die Rücksprungadresse vom Stack holt, den Rückgabewert pushed und dann mittels GOTO zurück an die entsprechende Stelle in der aufrufenden Funktion springt.

Im Fall von Tail-Rekursion lässt sich natürlich viel vom Code für das Wegsichern und Wiederherstellen vermeiden, siehe Code in TA10.2, wodurch Rekursion in diesem Fall zu demselben Code wie eine While-Schleife führt.

Beispiel:

```
E0:
  PUSH x; PUSH y; PUSH 0; GOTO E1;
R0:
  r := POP; y := POP; x := POP; y := r
  HALT;

E1:
  IF x = 0 DO
    i := POP;
    PUSH y;
    IF i = 0 DO GOTO R0; END;
    IF i = 1 DO GOTO R1; END;
    IF i = 2 DO GOTO R2; END;
    GOTO ERR;
  END;
  IF x = 1 DO
    y := y + 1;
    i := POP;
```

```
      PUSH y;
      IF i = 0 DO GOTO R_0; END;
      IF i = 1 DO GOTO R_1; END;
      IF i = 2 DO GOTO R_2; END;
      GOTO ERR;
    END;
    x := x − 1;
    PUSH x; PUSH y; PUSH 1; GOTO E_1;
R_1:
    r := POP; y := POP; x := POP; y := r;
    x := x − 1;
    PUSH x; PUSH y; PUSH 2; GOTO E_1;
R_2:
    r := POP; y := POP; x := POP; y := r;
    PUSH y;
    IF i = 0 DO GOTO R_0; END;
    IF i = 1 DO GOTO R_1; END;
    IF i = 2 DO GOTO R_2; END;
    GOTO ERR;
ERR:
    GOTO ERR;
```