RWTH Aachen | **WS 2012/13**
Lehrgebiet Theoretische Informatik | Übungsblatt 2
Reidl–Ries–Rossmanith–Sanchez–Tönnis | 22.10.2012

Übung zur Vorlesung Berechenbarkeit und Komplexität

Aufgabe T3

Geben Sie die Gödelnummer $\langle M \rangle$ der unten angegebenen Turingmaschine an.

$$M = (\{q_1, q_2, q_3\}, \{0, 1\}, \{0, 1, B\}, B, q_1, q_3, \delta)$$

δ	0	1	B
q_1	$(q_1, 0, R)$	$(q_1, 1, R)$	$(q_2, 0, L)$
q_2	$(q_2, 0, L)$	$(q_2, 1, L)$	(q_3, B, R)

Nutzen Sie dabei die Definition der Gödelnummer aus der Vorlesung.

Lösungsvorschlag

Wir kodieren Zustand q_i als 0^i. Zusätzlich nummerieren wir sowohl das Alphabet durch indem wir $X_1 = 0$, $X_2 = 1$ und $X_3 = B$ setzen, als auch die möglichen Kopfbewegungen indem wir $D_1 = L$, $D_2 = N$ und $D_3 = R$ setzen. Die Gödelnummer ist dann:

$$\begin{array}{rcccccccccccl}
111 & 0^1 & 1 & 0^1 & 1 & 0^1 & 1 & 0^1 & 1 & 0^3 & \\
11 & 0^1 & 1 & 0^2 & 1 & 0^1 & 1 & 0^2 & 1 & 0^3 & \\
11 & 0^1 & 1 & 0^3 & 1 & 0^2 & 1 & 0^1 & 1 & 0^1 & \\
11 & 0^2 & 1 & 0^1 & 1 & 0^2 & 1 & 0^1 & 1 & 0^1 & \\
11 & 0^2 & 1 & 0^2 & 1 & 0^2 & 1 & 0^2 & 1 & 0^1 & \\
11 & 0^2 & 1 & 0^3 & 1 & 0^3 & 1 & 0^3 & 1 & 0^3 & 111
\end{array}$$

Aufgabe T4

Wiederholen Sie kurz das Konzept der universellen Turingmaschine.
Es sei U die universelle Turingmaschine. Wie arbeitet U auf der Eingabe $\langle U \rangle \langle U \rangle \langle U \rangle \langle M \rangle w$? Dabei sind M eine beliebige Turingmaschine und w ein beliebiges Eingabewort.

Lösungsvorschlag

Da die universelle Turingmaschine alle Turingmaschinen simulieren kann, kann sie sich natürlich auch selbst simulieren.
Daher wird U mehrfach geschachtelt simuliert, und der innerste Aufruf simuliert M auf Eingabe w. Das Ergebnis der Berechnung wird dann nach außen „weitergereicht“.

Aufgabe T5
Sind die folgenden Sprachen rekursiv? Begründen Sie Ihre Antwort.

a) $L_{100} = \{ \langle M \rangle \mid M$ hält auf der leeren Eingabe in höchstens 100 Schritten $\}$.

b) $L'_{100} = \{ \langle M \rangle \mid M$ besucht höchstens 100 Bandplätze auf der leeren Eingabe$\}$.

a) Die Sprache ist rekursiv. Nach 100 Schritten kann M höchstens 100 Bandplätze gelesen haben. Also können wir eine TM M_{100} bauen, die L_{100} entscheidet. M_{100} simuliert M auf der leeren Eingabe, und prüft ob Ml nach höchstens 100 Schritten hält.

b) Die Sprache ist rekursiv. Es können offensichtlich nur endlich viele Konfigurationen durchlaufen werden die kein Endzustand sind und nur 100 Bandplätze nutzen ($|\Gamma|^{100} \cdot (|Q| - 1) \cdot 100$). Es werden $|\Gamma|^{100} \cdot (|Q| - 1) \cdot 100$ viele Schritte von M auf der leeren Eingabe simuliert, wobei jede Zwischenkonfiguration gespeichert werden kann. Es können folgende Fälle auftreten

- Es werden mehr als 100 Plätze besucht, wir können also verwerfen.
- Eine Konfiguration wird zum zweiten mal besucht, folglich laufen wir in eine Endlosschleife. Da bis jetzt nicht mehr als 100 Plätze besucht wurden, werden auch beim nächsten mal nicht mehr besucht, daher können wir akzeptieren.
- Wurde nach $|\Gamma|^{100} \cdot (|Q| - 1) \cdot 100$ Schritten weder eine Konfiguration zweimal besucht, noch gehalten, muss M mehr als 100 Bandplätze besucht haben, wir können also verwerfen.

Aufgabe H3 (15 Punkte)
Es ist sehr nützlich über einen Turingmaschinensimulator zu verfügen, wenn man Turingmaschinen entwirft. Ziel dieser Aufgabe ist es, einen solchen Simulator zu implementieren.
Wir wollen genau solche Turingmaschinen simulieren, wie sie in der Vorlesung definiert wurden. Der Simulator soll als Eingabe einen Dateinamen einer Datei erhalten, welche eine Beschreibung der zu simulierenden Turingmaschine $M = \{Q, \Sigma, \Gamma, B, q_0, \bar{q}, \delta\}$ enthält, und ein Eingabewort $w \in \Sigma^*$.
Der Simulator soll dann M auf der Eingabe w simulieren und alle durchlaufenen Konfiguration in jeweils einer Zeile ausgeben.
Die Beschreibung einer Turingmaschine in einer Datei ist dabei folgendermaßen aufgebaut:

1. Die erste Zeile enthält die Anzahl der Zustände n, wobei wir $Q = \{q_1, q_2, \ldots, q_n\}$ vorraussetzen.
2. Die zweite Zeile enthält alle Zeichen aus Σ.
3. Die dritte Zeile enthält alle Zeichen aus Γ.
4. Die vierte Zeile enthält die Nummer i des Anfangszustands $q_i = q_0$.
5. Die fünfte Zeile enthält die Nummer j des Endzustands $q_j = \bar{q}$.

6. Die folgenden Zeilen enthalten je einen Übergang $\delta\colon (q, a) \mapsto (q', b, D)$ der Übergangsfunktion δ, wobei q, a, q', b und D in dieser Reihenfolge durch jeweils ein Leerzeichen getrennt auftreten.

7. Wir legen fest, daß das Blanksymbol `B` ist und nicht umdefiniert wird.

Folgende Vereinbarungen vereinfachen die Implementierung und Verwendung des Simulators etwas: Sie müssen nicht überprüfen, ob die Beschreibung und das Eingabewort syntaktisch und semantisch korrekt sind. Füttert man den Simulator mit falschen Daten, dann darf etwas beliebiges passieren. Der Simulator muß daher auch nicht alle Zeilen der Eingabe berücksichtigen. Es ist beispielsweise erlaubt, die Zeilen mit Σ und Γ zu ignorieren und nur die Beschreibung von δ zu verwenden. Es ist auch erlaubt, daß δ nicht vollständig angegeben wird und einige Übergänge fehlen. Das macht zum Beispiel dann Sinn, wenn Sie wissen, daß diese Übergänge sowieso nie verwendet werden. So läßt sich viel Schreibarbeit einsparen.

Sie müssen Ihr Programm nicht hocheffizient optimieren, aber es sollte auch nicht zu langsam sein. Längere Simulationen sollten mit nur kurzer Wartezeit durchgeführt werden können. Verwenden Sie eine vernünftige Programmiersprache, die halbwegs bekannt ist und deren Programme von den Tutoren verstanden werden (Java, C, C++, usw. sind alle sehr dafür geeignet. Javascript wäre wohl keine gute Idee).

Zum Schluß läßt sich das gewünschte Verhalten am besten an einem kleinen Beispiel erläutern:

```
$ head add.tm
7
01#
01#B
1
7
1 0 1 0 R
1 1 1 1 R
1 # 1 # R
1 B 2 B L
2 1 3 0 N
$ ./tm add.tm 10#1
[1]10#1
B1[1]0#1
B10[1]#1
B10#[1]1
B10#1[1]
B10#[2]1B
B10#[3]0B
B10[3]#0B
B1[4]0#0B
B1[1]1#0B
B11[1]#0B
B11#[1]0B
B11#0[1]B
B11#[2]0B
B11[2]#1B
B11#[5]1B
B11#B[5]B
B11#[5]BB
B11[5]#BB
B1[6]1BBB
B[6]11BBB
[6]B11BBB
BB[7]11BBB
$
```

Testen Sie Ihren Simulator an verschiedenen kleinen Turingmaschinen.

Erläutern Sie kurz, wie Ihr Simulator funktioniert und geben Sie Protokolle kleiner Beispielläufe und den Quelltext mit ab.

Lösungsvorschlag

Hier ist ein nicht besonders effizientes Programm in der Sprache D, welche das Gewünschte leistet:

```
import std.stdio;
import std.file;
import std.format;
import std.stream;

class TM {
  private int n; // number of states
  private string sigma, gamma; // input and tape alphabets
  private int q0, qbar; // initial and final state
  struct ta {int state; char c; char dir;}
  private ta delta[int][char];

  private int q; // state we are in
  private string left, right; // left and right tape content

  // Have we reached the final state yet?
  bool blocked() {return q == qbar;}

  // Start simulation on word w
  void start(string w) {
    q = q0;
    left = "";
    right = w;
  }

  // Simulate one step
  void step() {
    if(blocked()) return;
    if(right.length == 0) right = "B";
    if(left.length == 0) left = "B";
    int nextstate = delta[q][right[0]].state;
    char symbol_written = delta[q][right[0]].c;
    char dir = delta[q][right[0]].dir;
    q = nextstate;
    right = symbol_written~right[1..$];
    if(dir == 'L') {right = left[$ - 1]~right; left =
left[0..$ - 1];}
    else if(dir == 'R') {left~ = right[0]; right =
right[1..$];}
    else assert(dir == 'N');
  }

  // String representation of current configuration
  string configuration() {
    return left~'['~std.conv.to!string(q)~']'~right;
  }

  // Initialize from a description in a file
  this(string filename) {
    assert(filename.isFile());
    Stream file = new BufferedFile(filename);
    char line[] = file.readLine(); // read n
    formattedRead(line, "%d", &n);
    line = file.readLine(); // read Sigma
    line = file.readLine(); // read Gamma
    line = file.readLine(); // read q0
    formattedRead(line, "%d", &q0);
    line = file.readLine(); // read qbar
    formattedRead(line, "%d", &qbar);
    while(!file.eof) {
      line = file.readLine();
      int p, q;
      char char_read, char_write, dir;
      formattedRead(line, "%d %c %d %c %c",
                &p, &char_read, &q, &char_write, &dir);
      delta[p][char_read] = ta(q, char_write, dir);
      assert(dir == 'L' || dir == 'R' || dir == 'N');
    }
  }
}

void main(string args[]) {
  assert(args.length == 3, "Usage: tm M w");
  TM M = new TM(args[1]);
  M.start(args[2]);
  while(!M.blocked()) {
    writeln(M.configuration());
    M.step();
  }
  writeln(M.configuration());
}
```

Aufgabe H4 (8 Punkte)

Entwerfen Sie eine Turingmaschine, welche zwei durch # getrennte, binärkodierte Zahlen entgegennimmt und als Ausgabe ihre binärkodierte Summe präsentiert.

Führen Sie mehrere Simulationen dieser Turingmaschine mithilfe des Simulators aus Aufgabe H3 an aussagekräftigen Beispieleingaben durch.

Lösungsvorschlag

Eine Möglichkeit, die Addition durchzuführen, ist folgende: Es werden abwechselnd die rechte Zahl um eins verringert und die linke um eins erhöht. Das ganze endet, wenn die rechte Zahl 0 war. Dann wird alles bis auf die linke Zahl gelöscht und der Kopf auf das erste Zeichen der linken Zahl gesetzt:

```
7
01#
01#B
1
7
1 0 1 0 R  scan right to #
1 1 1 1 R
1 # 1 # R
1 B 2 B L  goto rightmost digit of second word
2 1 3 0 N  subtract one and goto state 3
2 0 2 1 L
2 # 5 # R  if this happens the number was already 0
3 0 3 0 L  scan to rightmost digit of first word
3 1 3 1 L
3 # 4 # L
4 0 1 1 N  add one to first word and goto 5
4 1 4 0 L
4 B 1 1 N
5 1 5 B R  delete second word
5 B 5 B L
5 # 6 B L
6 0 6 0 L  return to first word and stop
6 1 6 1 L
6 B 7 B R
```

Zwei typische Simulation mit dieser Turingmaschine erzeugen folgende Ausgabe:

```
$ ./tm add.tm 0#0
[1]0#0
B0[1]#0
B0#[1]0
B0#0[1]
B0#[2]0B
B0[2]#1B
B0#[5]1B
B0#B[5]B
B0#[5]BB
B0[5]#BB
B[6]0BBB
[6]B0BBB
BB[7]0BBB
$ ./tm add.tm 101#11
[1]101#11
B1[1]01#11
B10[1]1#11
B101[1]#11
B101#[1]11
B101#1[1]1
B101#11[1]
B101#1[2]1B
B101#1[3]0B
B101#[3]10B
B101[3]#10B
B10[4]1#10B
B1[4]00#10B
B1[1]10#10B
B11[1]0#10B
B110[1]#10B
B110#[1]10B
B110#1[1]0B
B110#10[1]B
B110#1[2]0B
B110#[2]11B
B110#[3]01B
B110[3]#01B
B11[4]0#01B
B11[1]1#01B
B111[1]#01B
B111#[1]01B
B111#0[1]1B
B111#01[1]B
B111#0[2]1B
B111#0[3]0B
B111#[3]00B
B111[3]#00B
B11[4]1#00B
B1[4]10#00B
B[4]100#00B
[4]B000#00B
B[1]1000#00B
B1[1]000#00B
B10[1]00#00B
B100[1]0#00B
B1000[1]#00B
B1000#[1]00B
B1000#0[1]0B
B1000#00[1]B
B1000#0[2]0B
B1000#[2]01B
B1000[2]#11B
B1000#[5]11B
B1000#B[5]1B
B1000#BB[5]B
B1000#B[5]BB
B1000#[5]BBB
B1000[5]#BBB
B100[6]0BBBB
B10[6]00BBBB
B1[6]000BBBB
B[6]1000BBBB
[6]B1000BBBB
BB[7]1000BBBB
$
```

Natürlich addiert eine solche Turingmaschine relativ langsam, wenn es sich um sehr große Zahlen handelt:

```
$ time ./tm add.tm 1101101010110110#101001110110010 | tail -1
BB[7]111011111010001000BBBBBBBBBBBBBBBB
```

```
real 0m0.451s
user 0m0.444s
sys 0m0.028s
$
```