GZ Theoretische Informatik (WS16/17)

Lösungsvorschlag zu Aufgabenblatt 8

Aufgabe 1

Wir kodieren Registerinhalte wie folgt: Enthält Register i die Zahl j, so kodieren wir dies mit p_i^j, wobei p_i die i-te Primzahl darstellt. Sei r ein Register, in dem das Produkt der Kodierungen aller Register gespeichert sind, also $r = \prod_{i \in \mathbb{N}} p_i^{x[i]}$, wobei $x[i]$ den Inhalt des Registers i darstellt. Wir führen zunächst einige Hilfsoperationen ein:

- **Primzahltest:**
  Aus der Vorlesung ist bereits bekannt, dass wir eine Zahl p darauf testen können, ob sie eine Primzahl ist. Dies werden wir in Zukunft durch einen Aufruf der Form $isPrime(p)$ tun.

- **i-te Primzahl:**
  Für unseren Agorithmus müssen wir die i-te Primzahl berechnen können. Wenn die Zahl i im Register Nummer 1 steht, dann sieht der Algorithmus wie folgt aus:

```
Start: x_2 = 2 goto Zeile 1      // Erste Primzahl ist 2
Zeile 1: while (x_1 > 1) {
Zeile 2:        x_2 = x_2 + 1
Zeile 3:        x_3 = isPrime(x_2)
Zeile 4:        if x_3 =? 1 then x_1 = x_1 − 1
                         else  (tunix) }
Zeile 5: x_1 = x_2 goto Halt // Schreibe Ergebnis an richtige Stelle
Halt:
```

  Das Programm verwendet einigen syntaktischen Zucker aus der Übung. Im Register x_2 werden mit zwei Beginnend die natürlichen Zahlen hochgezählt und getestet, ob es sich um eine Primzahl handelt. Falls dies der Fall ist, wird die Zahl der noch zu findenden Primzahlen in Register x_1 heruntergezählt. Wenn keine weitere Primzahl mehr gefunden werden muss, wird der Inhalt des Registers x_2 ausgegeben. Wir werden dieses Programm in Zukunft als: *ithPrime* verwenden.

- **Register auslesen:**
  Um die Registermaschine mit indirekter Adressierung zu simulieren, müssen wir deren Registerinhalte aus unserem Register auslesen können. Nehmen wir an, dass sich die Zahl des Registers, welches wir auslesen wollen, im Register 1 befinde, so funktioniert das Auslesen wie folgt:

```
// Berechnen der Primzahl, die den Registerinhalt angibt:
Start: x_2 = ithPrime(x_1) goto Zeile 1
// Zu Anfang steht 0 in Registern:
Zeile 1: x_1 = 0 goto Zeile 2
// Kopiere Registerinhalt:
Zeile 2: x_4 = r goto Zeile 3
// Ist die ite Primzahl ein Primfaktor?:
Zeile 3: x_3 = x_4 mod x_2 goto Zeile 4
Zeile 4: while (x_3 =? 0) {
//Inkrementiere Ergebnis:
Zeile 5:        x_1 = x_1 + 1
// Senke Registerinhalt an Stelle i um 1 im kopierten Register:
Zeile 6:        x_4 = x_4 div x_2
// Ist die ite Primzahl immer noch ein Primfaktor?:
Zeile 7:        x_3 = x_4 mod x_2 }
Halt:
```

Um zu zeigen, dass jede Registermaschine mit indirekter Adressierung von einer gewöhnlichen Registermaschine schrittweise simuliert werden kann, müssen wir alle möglichen Operationen einer Registermaschine mit indirekter Adressierung simulieren:

- **Vergleich eines Registerinhalts mit Null (direkte Adressierung)**
  Um den Inhalt eines Registers i auf Null zu testen, müssen wir überprüfen, ob sich r durch p_i teilen lässt, also ob $p_i \mid r$. Denn falls $x[i] = 0$ gilt, dann gilt $p_i^{x[i]} = 0$ und daher gilt $p_i \nmid r$. Wir nehmen an, dass die Register 2 bis 4 ungenutzt sind. Falls $x[i] = 0$ gilt, so wird eine 1 in das erste Register geschrieben, andernfalls eine 0.

```
Start: x_2 = r goto Z1
Z1:    x_3 = ithPrime(i) goto Z2
Z2:    x_4 = x_2 mod x_3
Z3:    if (x_4 =? 0) then x_1=0 goto Halt
                     else x_1=1 goto Halt
Halt:
```

Im folgenden können wir den Vergleich eines Registerinhalts mit Null mittels direkter Adressierung mit $isZero$ aufrufen.

- **Vergleich eines Registerinhalts mit Null (indirekte Adressierung)**
  Um den Inhalt eines Registers $x[i]$ auf Null zu testen, müssen wir zunächst den Registerinhalt von Register i ermitteln. Sei also $j := x[i]$. Dann müssen wir mit Hilfe des obigen Vergleichs von Registerinhalten mit Null mittels direkter Adressierung prüfen, ob $x[j] = 0$ gilt. Falls $x[x[i]] = 0$ gilt, so wird eine 1 in das erste Register geschrieben, andernfalls eine 0.

```
Start: x_1 = getRegister(i) goto Z1
Z1:    x_1 = isZero(x_1) goto Halt
Halt:
```

- **Inkrementieren eines Registers (direkte Adressierung)**
  Um ein Register i zu inkrementieren, müssen wir r mit p_i multiplizieren, da $p_i^j \cdot p_i = p_i^{j+1}$.

```
Start: x_1 = ithPrime(i) goto Z1
Z1:    r = r · x_1 goto Halt
Halt:
```

  Im folgenden können wir das Inkrementieren eines Registers mittels direkter Adressierung mit inc aufrufen.

- **Inkrementieren eines Registers (indirekte Adressierung)** Um ein Register $x[i]$ zu inkrementieren, müssen wir zunächst den Registerinhalt von Register i ermitteln. Sei also $j := x[i]$. Anschließend müssen wir mit Hilfe des obigen Inkrements von Registern mittels direkter Adressierung das Register j inkrementieren.

```
Start: x_1 = getRegister(i) goto Z1
Z1:    r = inc(x_1) goto Halt
Halt:
```

- **Dekrementieren eines Registers (direkte Adressierung)** Zum dekrementieren des Registers i lesen wir den Inhalt des Registers i aus und, falls es nicht 0 ist, dekrementieren wir es, indem wir den Registerinhalt von r durch die i-te Primzahl teilen

```
Start: x_1 = getRegister(i) goto Zeile 1
Zeile 1: x_2 = ithPrime(i) goto Zeile 2
Zeile 2: if (x_1 =?0) then (tunix) goto Halt
                      else r = r div x_2 goto Halt
Halt:
```

- **Dekrementieren eines Registers (indirekte Adressierung)** Zuerst lesen wir mit Hilfe von $x_1 = getRegister(i)$, den Registerinhalt des Registers i aus. Dann wird der Algorithmus zur direkten Dekrementierung mit $i = x_1$ benutzt.

Da wir in allen Teilprogrammen nur endliche viele Register benötigen, benötigen wir auch insgesamt nur endlich viele Register.

Aufbage 2

Nimm hierzu an $U = (\Sigma, Q, q_0, F, \Delta)$ sei ein solcher universeller deterministischer endlicher Automat mit $n \in \mathbb{N}$ vielen Zuständen. Definiere

$$L_n := \{w \in \{a,b\}^* | \text{der } n\text{-letzte Buchstabe von } w \text{ ist ein } a\}.$$

Nach Aufgabe 3.4(b) benötigt ein DEA der L_n akzeptiert mindestens 2^n Zustände. Sei w die Kodierung eines solchen Automaten und q der Zustand, der U nach Abarbeitung von $w\$$ erreicht. Dann akzeptiert $M = (\Sigma, Q, q, F, \Delta)$ genau die Sprache L_n. Allerdings hat M nur $n < 2^n$ viele Zustände. Also kann es keinen universellen endlichen Automaten geben.