Technische Universität Chemnitz Sommersemester 2002
Professur Theoretische Informatik
Prof. Dr. A. Goerdt, Dipl.-Inf. F. Schädlich, Dipl.-Inf. A. Lanka

10. Übung „Theoretische Informatik II“

Lösungsvorschläge

Aufgabe 1

Wir speichern den Zustand der RAM, so daß wir auf dem Band jeweils die Nummer und den Inhalt der benutzten Register speichern. Die Eingabe (der Länge n) der RAM steht in den ersten n Registern. Da die RAM im logarithmischen Kostenmaß $t(n)$-zeitbeschränkt ist, können die Inhalte aller während der Rechnung benutzten Register zusammen höchstens eine Länge von $O(t(n))$ haben. Damit haben wir eine Kodierung der Register gefunden, die die Länge $O(n + t(n))$ hat.

Wir verwenden eine 2-Band-TM. Bei jedem Lesen eines Registers kann die TM in $O(n+t(n))$ den richtigen Speicherbereich finden und in $O(n + t(n))$ auf das zweite Band kopieren. Das zweite Band simuliert uns also den Akkumulator der RAM. Beim Schreiben in ein Register muß eventuell ein Teil des Bandes verschoben werden, um Platz zu schaffen oder überflüssige Teile zu überschreiben. Dies geht in Linearzeit.

Die arithmetischen Befehle (Addition, Subtraktion, Multiplikation, Division) können problemlos in quadratischer Zeit ausgeführt werden.

Jeden Befehl der RAM können wir also in (maximal) quadratischer Zeit, d. h. $O((n+t(n))^2)$ auf einer Zweiband-TM simulieren. Die Simulation auf einer Einband-TM geht daher in $O((n + t(n))^4)$.

Aufgabe 2

Für $n > 100$ stehen die Funktionswerte fest. Unser Ziel ist es, die Rekurrenz für $n \leq 100$ aufzulösen. Betrachten wir zuerst alle n mit $90 \leq n \leq 100$. Dann gilt

$$f(n) = f(f(n + 11)) = f((n + 11) - 10) = f(n + 1).$$

Für alle $90 \leq n \leq 100$ gilt also $f(n) = f(n + 1)$. Die Funktion ist in diesem Bereich also konstant. Insbesondere gilt $f(100) = f(101)$. Den Wert von $f(101)$ kennen wir bereits, er ist 91. Also wissen wir, daß $f(n)$ für $90 \leq n \leq 100$ konstant 91 ist. Mit analogen Überlegungen für kleinere n erhält man schließlich

$$f(n) = \begin{cases} 91 & n \leq 100 \\ n - 10 & n > 100 \end{cases}$$

Aufgabe 3

Wir benutzen ein LOOP-Programm mit möglichst wenigen Variablen und Zeilen:

```
x0:=x0+1;
LOOP x1 DO
   x1:=x0+0;
   x0:=x0+x2;
   x2:=x1+0;
END;
```

Wir sehen, daß dieses Programm im eigentlichen Sinne kein LOOP-Programm ist. Die störende Anweisung `x0:=x0+x2` läßt sich aber als LOOP-Programm darstellen. Aus Gründen der Übersicht benutzen wir aber diese Variante. Wir nennen den Anweisungsteil in der Schleife R, die LOOP-Anweisung Q und die initiale Zuweisung für `x0` P.

Wie sieht nun die simultan rekursive Funktion für das Programm R aus?

Die Funktion $h_i(x_0, x_1, x_2)$ soll uns den Wert von x_i liefern, nachdem R einmal ausgeführt wurde und dabei die Variableninhalte $x_0, \ldots, x_2$ zu Beginn hatte. Daraus ergibt sich:

$$\begin{array}{rcl} h_0(x_0, x_1, x_2, n) & = & add(x_0, x_2) \\ h_1(x_0, x_1, x_2, n) & = & x_0 \\ h_2(x_0, x_1, x_2, n) & = & x_0 \end{array} \tag{1}$$

Das zusätzliche n wird für die spätere Rekursion eingeführt. Die Einführung der Funktion *add* ist nötig, weil direkte Addition bei simultaner Rekursion nicht zulässig sind. Dagegen ist aber die Nachfolgerfunktion s Bestandteil der Definition. So können wir *add* definieren:

$$\begin{array}{lcl} add(0, z) & = & z \\ add(y+1, z) & = & s(add(y, z)) \end{array} \tag{2}$$

Bis jetzt haben wir drei Funktionen, die uns den Wert der drei Variablen nach Ausführung von H liefern. Dabei ist noch keine simultane Rekursion aufgetreten, diese entsteht erst, wenn LOOP-Schleifen als Funktionen umgesetzt werden.

Wir erstellen nun Funktionen $g_i(n, x_0, x_1, x_2)$ mit der Bedeutung, daß sie uns den Wert von x_i liefern, nachdem die LOOP-Schleife n-mal ausgeführt wurde und zu Beginn die Variablen die Werte x_0, x_1, x_2 hatten. Wir verwenden $\overline{x}$ stellvertretend für x_0, x_1, x_2 um die Übersicht zu behalten. Um Q als Funktion umzusetzen, gehen wir dem Ansatz nach, daß eine n-malige Ausführung des Programmes R das gleiche ist, wie eine $n-1$-malige Ausführung und danach noch eine weitere.

$$\begin{array}{rcl} g_0(n+1, \overline{x}) & = & h_0(g_0(n, \overline{x}), g_1(n, \overline{x}), g_2(n, \overline{x}), n) \\ g_1(n+1, \overline{x}) & = & h_1(g_0(n, \overline{x}), g_1(n, \overline{x}), g_2(n, \overline{x}), n) \\ g_2(n+1, \overline{x}) & = & h_2(g_0(n, \overline{x}), g_1(n, \overline{x}), g_2(n, \overline{x}), n) \end{array} \tag{3}$$

Selbstverständlich brauchen wir noch Anfangsbedingungen für den Fall, daß die LOOP-Schleife überhaupt nicht betreten wird:

$$\begin{array}{rcl} g_0(0, \overline{x}) & = & x_0 \\ g_1(0, \overline{x}) & = & x_1 \\ g_2(0, \overline{x}) & = & x_2 \end{array} \tag{4}$$

Die Variablen werden also nicht verändert.

Die Funktionen f_i für P sind einfach aufzustellen:

$$\begin{array}{rcl} f_0(x_0,x_1,x_2) & = & s(x_0) \\ f_1(x_0,x_1,x_2) & = & x_1 \\ f_2(x_0,x_1,x_2) & = & x_2 \end{array} \tag{5}$$

Natürlich müssen wir noch darstellen, daß Q nach P ausgeführt wird. Die machen wir durch die Funktionen e_i

$$\begin{array}{rcl} e_0(x_0,x_1,x_2) & = & g_0(x_1, f_0(x_0,x_1,x_2), f_1(x_0,x_1,x_2), f_2(x_0,x_1,x_2)) \\ e_1(x_0,x_1,x_2) & = & g_1(x_1, f_0(x_0,x_1,x_2), f_1(x_0,x_1,x_2), f_2(x_0,x_1,x_2)) \\ e_2(x_0,x_1,x_2) & = & g_2(x_1, f_0(x_0,x_1,x_2), f_1(x_0,x_1,x_2), f_2(x_0,x_1,x_2)) \end{array} \tag{6}$$

Wir erhalten schließlich als Funktionen für das Programm der Fibonacci-Zahlen (zusammen mit den Definitionen (1)-(6)):

$$FIB(n) := e_0(0,n,0)$$

Mit der gleichen Vorgehensweise bilden wir nun eine simultan rekursive Funktion für $x!$. Wir verzichten zugunsten des Überblicks auch hier auf eine detaillierte Implementierung des LOOP-Programms und verwenden die Multiplikation mit dem Wissen, daß wir ein LOOP-Programm dafür finden können:

```
x0:=1;
LOOP x1 DO
   x0:=x0*x1;
   x1:=x1-1;
END;
```

Die Funktionen für den inneren Teil der Schleife können so aussehen:

$$\begin{array}{rcl} h_0(x_0,x_1,n) & = & mul(x_0,x_1) \\ h_1(x_0,x_1,n) & = & p(x1) \end{array} \tag{7}$$

Wir haben schonmal den bei der Rekursion hinzukommenden Parameter n mit aufgenommen. Die Funktion mul liefert uns eine Multiplikationsfunktion:

$$\begin{array}{rcl} mul(0,x_1) & = & 0 \\ mul(x_0+1,x_1) & = & add(mul(x_0,x_1),x_1) \end{array} \tag{8}$$

Die Funktion p liefert uns den Vorgänger des Argumentes (Der Vorgänger von 0 ist 0, da es in LOOP-Programmen nur natürliche Zahlen gibt).

$$\begin{array}{l} p(0) = 0; \\ p(n+1) = max(p(n),n) \\ max(x,n) = n; \end{array} \tag{9}$$

Auch hier ist wieder das zusätzliche n aus Definitionsgründen bei max hinzugekommen.
Für die LOOP-Schleife an sich folgt mit der gleichen Idee wie beim ersten Fall:

$$\begin{array}{lcl} g_0(0,x_0,x_1) & = & x_0 \\ g_1(0,x_0,x_1) & = & x_1 \\ g_0(n+1,x_0,x_1) & = & h_0(g_0(n,x_0,x_1),g_1(n,x_0,x_1),n) \\ g_1(n+1,x_0,x_1) & = & h_1(g_0(n,x_0,x_1),g_1(n,x_0,x_1),n) \end{array} \tag{10}$$

Schließlich noch die Funktionen für die erste Anweisung:

$$\begin{array}{lcl} f_0(x_0,x_1) & = & 1 \\ f_1(x_0,x_1) & = & x_1 \end{array} \tag{11}$$

Und damit für das ganze Programm:

$$\begin{array}{lcl} e_0(x_0,x_1) & = & g_0(f(x_0,x_1),f(x_0,x_1),f(x_0,x_1)) \\ e_1(x_0,x_1) & = & g_1(f(x_0,x_1),f(x_0,x_1),f(x_0,x_1)) \end{array} \tag{12}$$

Die Funktion $x!$ ist also gleich $e_0(0,x)$.

Aufgabe 4

Das Problem bei der simultanen Rekursion ist, daß zum Beispiel die Funktion g_0 sich auf g_1 bezieht, während g_1 die Funktion g_0 benutzt. Die Funktionen sind zwar trotzdem berechenbar, aber eben unhandlich. Deswegen führt man die primitive Rekursion ein, bei der gefordert ist, daß die Funktionen, die bei der Rekursion benutzt werden bereits vollständig definiert sind. Dadurch werden solche Fälle verhindert. Man kann jedoch trotzdem aus jedem LOOP-Programm eine primitive Rekursion machen. Dafür werden alle Variablen (bijektiv) in einer Zahl kodiert. Als Resultat benötigt man nicht mehr so viele Funktionen $f_i, g_i, h_i, \ldots$, sondern nur noch jeweils eine Funktion $f, g, h, \ldots$. Dies liegt daran, daß durch die Kodierung eine Funktion mehrere Variablen zurückgeben kann, obwohl formal nur ein Wert zurückgegeben wird. Also brauchen wir nicht für jede Variable x_i eine Funktion F_i.

Wir können also aus jedem LOOP-Programm eine rekursive Funktion machen. Es genügt also für den Nachweis, daß die Funktion $\mathrm{prim}(p)$ primitiv rekursiv ist, die Angabe eines LOOP-Programmes. Dieses könnte zum Beispiel so aussehen:

```
x0:=x0+1;
x3:=x3+1;
x2:=x1-2;
LOOP x2 DO
   x3:=x3+1;
   x2:=x1 MOD x3;
   x4:=x5+1;
   LOOP x2 DO
      x4:=x5+0;
   END;
   LOOP x4 DO
      x0:=x5+0;
   END;
END;
```

Die Funktion MOD ist als LOOP-Programm darstellbar.

Bemerkung: Jede Defintion zur primitiven Rekursion enthält die Beschreibung eines Gleichungssystems, welches die Rekursion an sich enthält. Nach der Definition der Vorlesung muß z. B. f ein Gleichungssystem der folgenden Form erfüllen:

$$\begin{aligned} f(0, z_1, \ldots, z_k) &= g(z_1, \ldots, z_k) \\ f(n+1, z_1, \ldots, z_k) &= h(f(n, z_1, \ldots, z_k), z_1 \ldots, z_k, \mathbf{n}) \end{aligned}$$

wobei g, h bereits primitiv rekursive Funktionen sind. An dieser Stelle ist es wichtig, daß h den Paramter n enthalten kann.

Aufgabe 5

Wir haben die Idee, die Konfiguration der TM in drei Zahlen zu kodieren. Dazu ordnen wir jedem Zeichen des Bandalphabetes eine Zahl zu. Sei also $\Gamma = \{a_1, a_2, \ldots, a_{|\Gamma|}\}$, dann erhält a_1 die Zahl 1, a_2 die Zahl 2, usw. Außerdem wählen wir uns eine Zahl $b > |\Gamma|$.
Wenn die TM die aktuelle Konfiguration $\alpha_1 \ldots \alpha_k z_i \beta_1 \ldots \beta_l$ besitzt, speichern wir das folgendermaßen:

$$\begin{aligned} x &:= [\#\alpha_1 \ldots \#\alpha_k]_b \\ y &:= [\#\beta_l \ldots \#\beta_1]_b \\ z &:= i \end{aligned}$$

Zur Erläuterung: $\#\alpha_1$ bezeichnet die Zahl, die wir dem Bandzeichen α_1 zugeordnet haben. Wir schreiben alle diese Zahlen hintereinander und fassen das Resultat als Zahl zur Basis b auf. Kommen wir nun zur Simulation:

```
x:=x*b+m
y:=y div b
z:=l
```

und bei b):

```
y:=(y div b)*b+m
z:=l
```

Der „Kopf“ der Simulation mit den `IF ... THEN GOTO...`-Anweisungen ist der Vorlesung oder dem Buch „Schöning: Theoretische Informatik kurzgefasst“ zu entnehmen.