Musterlösung zu Nichtsequentielle Programmierung: Übung 1

April 26, 2015

1. Begriffsklärungen

0.5 Punkte für Ja, Nein; 0.5 für Konstruktion von Beispiel/Begründung

a) Jeder deterministische Algorithmus ist determiniert.

Bei einem deterministischen Algorithmus ist sein Ablauf – als Folge von definierten Zuständen – reproduzierbar und identische Eingaben werden identische Ausgaben abgebildet. Bei Vergleich der Definitionen fällt auf, dass Determinismus Determiniertheit enthält. [1]. Die richtige Antwort ist also *Ja*.

[1] Determinismus = Determiniertheit ∧ eindeutige Zustandsfolge

b) Jeder determinierte Algorithmus ist deterministisch.

Nein, das ist nicht korrekt wie folgendes Gegenbeispiel in Pseudocode(|| ist hierbei der Operator für parallele Komposition atomarer Instruktionen daher *gleichzeitige* Ausführung) zeigt:

```
1    a := 0
2    {a = a+1} || {a = a+1} || {a = a−1} || {a = a+1}
```

Offensichtlich ist der Algorithmus determiniert, denn a nimmt nach jeder möglichen Ausführungfolge den Wert 2 an. Allerdings ist die Reihenfolge der Additionen und der Subtraktion nicht festgelegt, die eindeutige Zustandsfolge gibt es nicht [2]. Mit der vorherigen Aufgabe folgt für Determinismus und Determiniertheit bei Algorithmen eine echte Teilmengenrelation: Determinismus $\subset$ Determiniertheit.

[2] Vergleiche beispielsweise die Ausführungsreihenfolgen ++-+ und -+++, wobei + für die Addition und - für die Subtraktion steht.

c) Jeder deterministische Algorithmus ist sequentiell.

Die Definition von Determinismus sieht eine eindeutige Zustandsfolge vor, in welcher der nachfolgende Zustand eindeutig über den bisherigen Zustand und die Eingaben definiert wird. Alternativ kann man die Kontraposition der Aussage betrachten: Nichtsequentielle Algorithmen sind nichtdeterministisch. Die Antwort ist also *Ja*.

d) Jeder nichtsequentielle Algorithmus ist nicht determiniert.

Man betrachte den Algorithmus in b). Offensichtlich ist er nichtsequentiell, allerdings ist sein Ergebnis determiniert. Die Antwort ist also *Nein*.

Für eine schöne Diskussion werden – je nach deren Güte – entsprechende Teile des Bonuspunktes zu vergeben.

e) Ist jeder sequentielle Algorithmus determiniert?

Näheren wir uns der Aufgabe mit dem Versuch ein Gegenbeispiel zu konstruieren: einen sequentiellen Algorithmus der nicht-determiniert ist. Nimmt man an, dass man Seiteneffekte nutzen kann, ist die Formulierung eines passenden Algorithmus nicht schwer: Nehme als Eingabe eine Zahl i, generiere 2 echte – daher nicht aus dem Pseudozustandsgenerator – Zufallszahlen j und k, liefere $i + j + k$ als Ausgabe wieder. Die Reihenfolge der Schritte ist wohldefiniert, wobei das Ergebnis nicht eindeutig ist, da manche der Zustände Seiteneffekte haben.

2. Kombinatorische Explosion

Aufgabe a: 1 Punkt für 2 richtige Rechnungen Aufgabe b: 1 Punkt für die RICHTIGE Anwendung der Formel
1 Punkt für den Vergleich mit dem Radius des Universums

Wenn es p Folgen von $n_1 \dots n_p$ atomaren Anweisungen gibt, dann ist die Anzahl der möglichen Ablaufreihenfolgen

$$\frac{\left(\sum_{i=1}^{p} n_i\right)!}{\prod_{i=1}^{p} n_i!}$$

.

a) 1) 2 Folgen mit je 3 Anweisungen: $\frac{(3+3)!}{3!\cdot 3!} = \frac{6!}{6\cdot 6} = \frac{5!}{6} = 20 \cdot (10^6)^0$
   2) 4 Folgen mit je 4 Anweisungen: $\frac{(4+4+4+4)!}{4!\cdot 4!\cdot 4!\cdot 4!} = \frac{16!}{24^4} = 63,063 \cdot 10^6$
   3) 6 Folgen mit je 5 Anweisungen: $\frac{(6\cdot 5)!}{(5!)^6} = \frac{30!}{120^6} \approx 88,8 \cdot (10^6)^3$
   4) 8 Folgen mit je 6 Anweisungen: $\frac{(8\cdot 6)!}{(6!)^8} = \frac{48!}{720^8} \approx 172 \cdot (10^6)^6$

b) 10 Folgen mit je 10 Anweisungen:

$$\frac{(10 \cdot 10)!}{10^{10}} \approx 2,36 \cdot 10^{92}$$

Wir nehmen an, dass eine mögliche Ausführung der 100 Zeilen auf ein doppelseitiges DIN A4-Blatt passt. Ein DIN A4-Blatt besitzt ein Volumen von

$$\frac{1}{16} m^2 \cdot 0.0001m = 6,25 \cdot 10^{-6} m^3.$$

Damit erhalten wir das Gesamtvolumen

$$2,36 \cdot 10^{92} \cdot 6,25 \cdot 10^{-6} m^3 = 1,475 \cdot 10^{87} m^3.$$

Der Radius ist dann $r_P = \sqrt[3]{\frac{3}{4\pi} \cdot 10^{87} m^3} \approx 7,1 \cdot 10^{28}$km. Das Universum hat jedoch nur einen Radius von $r_U = 45 \cdot 10^9 \cdot 9,46 \cdot 10^{12}km\approx 4,3 \cdot 10^{23}$km. Im Endeffekt haben wir also $r_P \approx 10000 \cdot 16,5 r_U$.

3. Ausführungsreihenfolgen

a) Welche Ausführungsreihenfolgen liefern das korrekte Ergebnis?
   Notwendige Abhängigkeiten zwischen Instruktionen:

```
a: 1;3;5;6
b: 2;4;6
```

Gültige Reihenfolgen:
Alle, in denen obige Abhängigkeiten erfüllt werden.
(2 Pkt für Abhängigkeit (1 Pkt.) und Anzahl (1 Pkt.); alternativ für die Angabe aller Folgen ebenfalls 2 Pkt)

   Insgesamt gibt es 35 Möglichkeiten nach $\frac{(3+4)!}{3!*4!}$
   Z.B.: 1,3,5,2,4,6
   2,1,3,4,5,6
   ...

b) Zu welchen Fehlern kann es bei beliebiger Ausführungsreihenfolge kommen?

Jeder Fehler (Anstrich) 0.5 Pkt

- Arbeiten mit nicht initialisierten Variablen

- Ergebnis wird 0 unabhängig von tatsächlichen Summen
- Reihenfolge 1,2,4,6,5,3 (Bsp.) erzeugt falsches Endergebnis (B statt A wird gewählt)
- Reihenfolge 1,2,3,5,6,4 (Bsp.) erzeugt falsches Endergebnis (A statt B wird gewählt)

4. Nichtsequentieller Vergleich

2 Punkte für die richtige Anwendung der Nebenläufigkeit, d.h. das Erstellen von zwei Prozessen zur Traversierung und die Informationen sinnvoll mit Kanälen weiterreichen. Es ist unkritisch, wenn kein dritter Prozess zum Vergleichen erzeugt wird, sondern das im main-Prozess gemacht wird.
2 Punkte für korrekten Quelltext. Da das die erste 'richtige' go-Abgabe ist, wird darauf geachtet, dass der Code vernünftig aussieht und sich an die go-Prinzipien hält.

```
1  package main
2
3  type tree struct {
4    value int
5    leftChild,
6    rightChild *tree
7  }
8
9  func (t *tree) traverseTree (ch chan int) {
10   t.traverseTreeWalk(ch)
11   close(ch)
12 }
13
14 func (t *tree) traverseTreeWalk (ch chan int) {
15   if t != nil {
16     ch <- t.value
17     t.leftChild.traverseTreeWalk(ch)
18     t.rightChild.traverseTreeWalk(ch)
19   }
20 }
21
22 func (t *tree) compareTrees (t1 *tree) bool {
23   ch, ch1:= make (chan int), make (chan int)
24   go t.traverseTree(ch)
25   go t1.traverseTree(ch1)
26   for {
27     value, more:= <-ch
28     value1, more1:= <-ch1
29     if !more && !more1 {
30       return true
31     }
32     if more != more1 || value != value1 {
33       return false
34     }
35   }
36   return false
37 }
38
39 func leaf (n int) *tree {
40   return &tree{n, nil, nil}
41 }
42
43 func main() {
44 /*        1                1                1
45         /   \            /   \              \
46        2      6         2      3             2
47      /   \                   /   \            \
48    3       4               4       6           3
49          /                /                  /   \
50        5                5                  6       5
51                                                    \
52                                                      4     */
53   t1:= &tree{1, &tree{2, leaf(3), &tree{4, leaf(5), nil}}, leaf(6)}
54   t2:= &tree{1, leaf(2), &tree{3, &tree{4, leaf(5), nil}, leaf(6)}}
55   t3:= &tree{1, nil, &tree{2, nil, &tree{3, leaf(6), &tree{5, nil, leaf(4)}}}}
56   println (t1.compareTrees(t1))
```

```
57    println (t1.compareTrees(t2))
58    println (t1.compareTrees(t3))
59  }
```