**Fragen, Antworten, Kommentare und Hinweise**

Das Video zur Lösung der Aufgabe 35 finden Sie unter: https://youtu.be/dKDHqf3qks4

Das Video zur Lösung der Aufgabe 36 finden Sie unter: https://youtu.be/6VFiAG4YJcg

Frage: Ich habe bei Aufgabe 35a folgende Invariante berechnet, y = 1+z-x, komme aber nicht weiter.

Antwort: Sie haben eine sinnvolle Invariante gefunden, was zu Teilpunkten in der Klausur führen würde, aber mit der finalen Nachbedingung nicht passt.

„$p \wedge \neg B$“ $\equiv y=1+z-x \wedge x=1 \equiv y=1+z-1 \wedge x=1 \equiv y=z \wedge x=1$, aber es gilt nicht $y=z \wedge x=1 \rightarrow y=z-1$

Anschaulich ist Ihre Invariante um 1 zu groß. Oft können Konstanten, die nicht an Multiplikationen oder Divisionen beteiligt sind, beliebig verändert werden. Das gilt auch in diesem Fall, so ist z. B. y =42+z-x auch eine Invariante. Da die gewählte Invariante um 1 zu groß ist, passt hier y=z-x.

Frage: Ich habe die Idee bei Aufgabe 36a genutzt den Schleifenzähler einzubauen und habe die Invariante u = z-x, komme aber nicht weiter.

Antwort: $u=z-x[x:=x-1] \equiv u = z-x+1$ $u=z-x+1[u:=u+1] \equiv u+1=z-x+1 \equiv u=z$ ist eine Invariante, bei der wieder die Nachbetrachtung nicht passt.

„$p \wedge \neg B$“ $\equiv u = z-x \wedge x=y$ , daraus lässt sich folgern $u=z-x$ und $u=z-y$, aber man kommt nicht auf $u=z$.

Um zur Lösung zu kommen hilft die Frage, ob es bei der Schleifenausführung etwas gibt was konstant bleibt. Das trifft zunächst auf y zu, da es nicht verändert wird und so Teil einer Invariante sein kann. (Die Summe aus u und x bleibt ebenfalls konstant, da eine Variable hoch, die andere runtergezählt wird; daraus wurde aber bereits z=u+x zum Einstieg in die Frage hergeleitet). Weiterhin muss man mit „$p \wedge \neg B$“ durch umformen und eventuell abschwächen zu $u=z$ kommen und oben ist am Ende mit $u=z-y$ anschaulich ein y zu wenig. Da y sich nicht ändert, kann es einfach zur Invariante addiert werden und mit $u=z-x+y$ kommt man zum Ergebnis.

Frage: Bei den Aufgaben, bei denen begründet werden soll, ob ein Hoare-Tripel auch für totale Korrektheit gilt, sollen wir da eine Terminierungsfunktion angeben?

Antwort: Es ist kein formaler Beweis gefordert, nur eine präzise Begründung. Dabei gibt es zwei Fälle. Im ersten Fall terminiert die Schleife nicht, dann ist ein Fall zu benennen, der die Vorbedingung erfüllt und das Programm dazu führt, nicht zu terminieren. Bei 35c ist es $zust(x)=0$, $zust(z)=0$ da $zust \in \{z=x \wedge \neg(x=1)\}$ und die while-Schleife nicht terminiert. Im zweiten Fall terminiert die Schleife und das sollte mit Hilfe der Vorbedingung begründet werden können. Bei 36c gilt am Anfang $x>y$, dann wird in der Schleife x immer um 1 kleiner (und wir als Typen nur ganze Zahlen betrachten), deshalb muss irgendwann $x==y$ gelten und die Schleife terminieren.

Frage: Wieso ist y=z-x $\wedge$ x=1 -> y=z-1 eine Abschwächung auf der rechten Seite, ist das nicht das Gleiche?

Antwort: Vorweg, wenn es das Gleiche also äquivalent wäre, ist es trotzdem formal eine Abschwächung, da Äquivalenzumformungen halt ein Spezialfall der Abschwächung ist (true -> true). Im konkreten Fall liegt aber eine echte Abschwächung vor, da für alle Zustände z, die die Zusicherung auf der linken erfüllen, immer z(x)=1 gelten muss. Auf der rechten Seite ist x frei wählbar.

Frage: Bedeutet der Begriff $\varepsilon$-Abschluss das gleiche wie $\varepsilon$-Hülle?

Antwort: Bei den mir bekannten Definitionen ja.

Achtung: Gibt ein Test das Ergebnis

```
org.opentest4j.AssertionFailedError: nicht korrekt bearbeitet ==> expected: <true>
but was: <false>
```

aus. Ist zu beachten, dass nach dem ersten Doppelpunkt das Wort steht, das Probleme macht. Bei obiger Ausgabe ist es das Wort $\varepsilon$.

Hinweis: In der Programmverifikation fokussieren wir auf den Datentypen int. Generell sind die gleichen Überlegungen auch bei anderen Typen wie double machbar, dabei muss aber genau über die Repräsentation der Werte bei der Argumentation nachgedacht werden. Das ist aufwändig aber machbar. Das nachfolgende Programm analysiert einige Möglichkeiten bei der Nutzung von double im Bereich sehr kleiner positiver Zahlen.

```
  public static void main(String[] args) {
    double x = 1;
    double old = 0;
    int count = 0;
    while(x > 0 && x != old) {
      old = x;
      x = x / 2.0;
      count++;
    }
    System.out.println("  old = " + old);
    System.out.println("   x  = " + x);
    System.out.println("count = " + count);

    BigDecimal bd = BigDecimal.ONE;
    BigDecimal oldbd = BigDecimal.ZERO;
    while(bd.compareTo(BigDecimal.ZERO) > 0 && !bd.equals(oldbd) && count > 0) {
      oldbd = bd;
      bd = bd.divide(BigDecimal.valueOf(2));
      count--;
    }
    System.out.println("oldbd = " + oldbd);
    System.out.println(" bd   = " + bd);
```

```
    bd = new BigDecimal(BigInteger.ONE, Integer.MAX_VALUE);
    System.out.println(" bd  = " + bd);
    bd = bd.divide(BigDecimal.valueOf(2));
    System.out.println(" bd   = " + bd)
}
```

Das obige Programm berechnet eine positive Zahl, die durch 2 geteilt 0 ergibt (!?). Soll mit hoher Genauigkeit gerechnet werden, ist der Typ BigDecimal zu nutzen. Die vorherige Rechnung wird dann wiederholt. Das Ergebnis ist eine genauere kleine Zahl, die auch weiterhin durch 2 geteilt größer als 0 ist.

Abschließend wird ein sehr kleiner BigDecimal-Wert angegeben. Der zweite Wert ist der negative Exponent der 10er-Pozenz, also $10^{-Integer.MAX\_VALUE}$. Wird dieser Wert durch 2 geteilt, ist das exakte Ergebnis nicht mehr repräsentierbar, das Ergebnis ist eine Exception. Die vollständige Ausgabe sieht wie folgt aus.

```
  old = 4.9E-324
   x  = 0.0
count = 1075
oldbd =
4.940656458412465441765687928682213723650598026143247644255856825006755072702087518652998363616359923797965646954457177309266567103559397963987747960107818781263007131903114045278458171678489821036887186360569987307230500063874091535649843873124733972731696151400317153853980741262385655911710266585566867681870395603106249319452715914924553293054565444011274801297099954193198940908041656332452475714786901472678015935523861155013480352649347201937902681071074917033322268447533357208324319360923828934583680601060115061698097530783422773183292479049825247307763759272478746560847782037344696995336470179726777175851256605511991315048911014510378627381672509558373897335989936648099411642057026370902792427675445652290875386825064197182655334472656 25E-324
 bd   =
2.470328229206232720882843964341106861825299013071623822127928412503377536351043759326499181808179961898982823477228588654633283551779698981993873980053909390631503565951557022639229085839244910518443593180284993653615250031937045767824921936562366698636584807570015857692699037063119282795585513329278343384093519780155312465972635795746227664652728272200563740064854999770965994704540208281662262378573934507363390079677619305775067401763246736009689513405355374585166611342237666786041621596804619144672918403005300575308490487653917113865916462395249126236538818796362393732804238910186723484976682350898633885879256283027559956575244555072551893136908362547791869486679949683240497058210285131854513962138377228261454376934125320985913276672363281 25E-324
 bd  = 1E-2147483647
Exception in thread "main" java.lang.ArithmeticException: Non-terminating decimal
expansion; no exact representable decimal result.
      at java.base/java.math.BigDecimal.divide(BigDecimal.java:1780)
      at
kleinstePositiveZahl.KleinstePositiveZahl.main(KleinstePositiveZahl.java:44)
```

Ist ein BigDecimal-Wert nicht exakt repräsentierbar, wird eine Exception geworfen. Das passiert z. B. auch wenn sie 1 durch 3 teilen.