

Fragen, Antworten und Kommentare zur aktuellen Vorlesung

Das Video zur Lösung der Aufgabe 35 finden Sie unter: <https://youtu.be/y0SKInYDtOc>

Das Video zur Lösung der Aufgabe 36 finden Sie unter: <https://youtu.be/0VdoUkjq5Cg>

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: $u=z-x[x:=x-1] \equiv u = z-x+1 \quad u=z-x+1[u:=u+1] \equiv u+1=z-x+1 \equiv u=z$ 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 $\text{zust}(x)=0, \text{zust}(z)=0$ da $\text{zust} \in \{z=x \wedge \neg(x=1)\}$ 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 $y=z-x \wedge x=1 \rightarrow y=z-1$ 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 ($true \rightarrow true$). 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.

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 ε .

Hinweis: In der Programmverifikation fokussieren wir auf den Datentypen `int`. Generell sind die gleichen Überlegungen auch bei anderen Typen wie `double` denkbar, 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-Potenz, also $10^{-\text{Integer.MAX_VALUE}}$. 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
olddb =
4.94065645841246544176568792868221372365059802614324764425585682500675507270208751
8652998363616359923797965646954457177309266567103559397963987747960107818781263007
1319031140452784581716784898210368871863605699873072305000638740915356498438731247
3397273169615140031715385398074126238565591171026658556686768187039560310624931945
2715914924553293054565444011274801297099995419319894090804165633245247571478690147
2678015935523861155013480352649347201937902681071074917033322268447533357208324319
3609238289345836806010601150616980975307834227731832924790498252473077637592724787
4656084778203734469699533647017972677717585125660551199131504891101451037862738167
2509558373897335989936648099411642057026370902792427675445652290875386825064197182
65533447265625E-324
bd =
2.470328222920623272088284396434110686182529901307162382212792841250337753635104375
9326499181808179961898982823477228588654633283551779698981993873980053909390631503
5659515570226392290858392449105184435931802849936536152500319370457678249219365623
6698636584807570015857692699037063119282795585513329278343384093519780155312465972
6357957462276646527282722005637400648549997709659947045402082816622623785739345073
6339007967761930577506740176324673600968951340535537458516661134223766678604162159
6804619144672918403005300575308490487653917113865916462395249126236538818796362393
7328042389101867234849766823508986338858792562830275599565752445550725518931369083
6254779186948667994968324049705821028513185451396213837722826145437693412532098591
327667236328125E-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.

Frage: Bedeutet der Begriff ε -Abschluss das gleiche wie ε -Hülle?

Antwort: Bei den mir bekannten Definitionen ja.