Beispiel 1 Div2 (alte Klausuraufgabe)

Hinweis: Rechnen Sie Sie am besten erst selbst, bevor Sie sich den Lösungsvorschlag anschauen!

Betrachten Sie folgende Funktion `d2`:

```
/*
 *  requires: ?
 *  ensures: \result + \result <= x && \result + \result + 1 >= x
 */
static int d2(int x) {
    int a = x;
    int b = 0;

    while (a > b) {
        a = a - 1;
        b = b + 1;
    }

    return a;
}
```

Beweisen Sie eine möglichst schwache Vorbedingung für die Funktion `d2`, indem Sie einen Annotationsbeweis führen.

*Verwenden Sie die Maske auf der nächsten Seite und tragen Sie die von Ihnen benutze **Invariante**, sowie die ermittelte **Vorbedingung**, an den entsprechenden Stellen als Kommentare ein!*

Führen Sie Vereinfachungen und Umformungen als *eigene Schritte* durch und vermischen Sie diese *nicht* mit anderen Regelanwendungen. Sie dürfen beliebig viele solcher Schritte durchführen, achten Sie allerdings darauf, dass diese einfach nachvollziehbar sind.

```
/*
 *  requires:
 *  ensures: \result + \result <= x && \result + \result + 1 >= x
 */
static int d2(int x) {

  assert


  assert


  assert

    int a = x;

  assert

    int b = 0;

  assert

    while (a > b) { // INV:

      assert


      assert


      assert

        a = a - 1;

      assert

        b = b + 1;

      assert

    }

  assert


  assert


  assert


  assert


  assert


  assert

    return a;

  // ensures: \result + \result <= x && \result + \result + 1 >= x
}
```

Lösungshinweise Auch zum allgemeinen Vorgehen

1. Man geht in solchen Beweisen immer von unten vor. Man fängt also mit dem `ensures` Teil an und schreibt diesen als Kommentar hinter das `return` oder schon mit `\result` durch den Ausdruck im `return` substituiert vor das `return`. In diesem Fall steht der `ensures` Teil schon als Kommentar dahinter.

   Da wir ein `return` wie eine Zuweisung zu der speziellen Variable `\result` sehen können, entsteht die erste Zusicherung einfach durch eine Anwendung der Zuweisungsregel. Dabei entsteht die nächste Zusicherung *rein mechanisch* aus der Zusicherung davor. Es werden alle Vorkommen der zugewiesenen Variable (hier: `\result`) in der Zusicherung durch den Ausdruck auf der rechten Seite der Zuweisung (hier: `a`) ersetzt.

   Wenn noch weitere Zuweisungen unter der Schleife wären, würde die Zuweisungsregel entsprechend für jede einmal angewendet, auch dann rein mechanisch.

2. In diesem Fall müssen wir uns nun mit der Schleife beschäftigen und eine *Invariante* finden. Die Idee ist, dass sie gut genug ist, um – zusammen mit der *negierten Schleifenbedingung* – die Zusicherung, die wir direkt nach der Schleife ermittelt haben, zu beweisen. Andererseits dürfen wir nicht zu viel behaupten, da wir sie auch innerhalb der Schleife beweisen müssen. Sie muss also nach dem Rumpf gelten, wenn man nur sie und die Schleifenbedingung am Kopf annimmt.

   Invarianten können im Allgemeinen nicht mechanisch gefunden werden, so dass man dort kreativ sein muss. In unserem Beispiel fällt direkt auf, dass von `a` eins abgezogen und auf `b` eins drauf addiert wird. Damit bleibt deren Summe immer gleich. Schaut man über der Schleife findet man heraus, dass diese Summe wohl gleich `x` sein muss. Damit ist der erste Kandidat für die Invariante `a + b == x`. Wir müssten allerdings mit dieser und der Bedingung `a <= b` (negierte Schleifenbedinung) die Zusicherung unter der Schleife beweisen, sprich

   ```
   a + a <= x  &&  a + a + 1 >= x
   ```

   Für den linken Teil der Zusicherung würde das auch schon reichen, denn wenn `a <= b` ist und wir `b` durch `a` ersetzen in `a + b == x`, dann gilt `a + a <= x`. Was uns also noch fehlt ist eine weitere Ungleichung mit `a` und `b`, mit der sicher der rechte Teil der Zusicherung zeigen lässt. Analog zum linken Fall bräuchten wir also die Zusicherung, dass `a + 1 >= b`. Insgesamt ergibt das als Invariante also die folgende, die wir im Kommentar notieren.

   ```
   a + b == x  &&  a + 1 >= b
   ```

3. Nun müssen wir die Schleifenregel mit dieser Invariante anwenden. Das können wir mit jeder Formal tun, es spielt keine Rolle, ob die Invariante korrekt oder ausreichend ist. Diese Anwendung ist also wieder *rein mechanisch*. Wir schreiben die Invariante vor die Schleife und ans Ende des Schleifenrumpfs. An den Anfang der Schleifenrumpfs schreiben wir die Invariante plus Schleifenbedingung. Hinter der Schleife steht die Invariante plus negierte Schleifenbedingung. In unserem Fall ist die Schleifenbedingung (`a > b`) stärker als der zweite Teil der Invariante (`a + 1 >= b`, was entspricht: `a + 2 > b`), so dass man diesen Teil direkt weg lassen kann. Die mechanische Regelanwendung liefert diesen Teil aber, so das ein Abschwächungsschritt notwendig ist.

   Die vier annotierten Zusicherungen verstehen sich nun so, dass die vor der Schleife und die am Ende des Rumpfs noch zu beweisen sind, wohingegen die anderen beiden von nun an angenommen werden dürfen und wir zu beweisende Zusicherungen nun auf diese zurückführen wollen.

4. Um die Verbindung hinter der Schleife zu ziehen können wir nur noch Abschwächungsschritte machen, d.h. die Formeln entweder umformen bei gleicher Aussage oder die Formeln nach unten schwächer machen. Das letztere heißt wir dürfen in Auswertungsrichtung des Programms Teilformen wegfallen lassen oder umgekehrt bei Rückwärtsschritten darf noch beliebiges dazu kommen (wir müssen es ja schließlich dann weiter oben beweisen).

   Hier gehen wir mal von oben vor und bedienen uns der Tatsache, dass `a == x - b` ist, was eine einfache Umformung von `a + b == x` ist. Wir machen dies als eigenen Schritt sichtbar und ändern auch noch die Reihenfolge.

   Nun können wir `b` überall einsetzen und die Formel für `b` fallen lassen (Abschwächung). Damit ist der Zusammenhang zu unserer ursprünglichen Zusicherung sichtbar gemacht und die Aussage bewiesen.

5. Bevor wir über der Schleife weiter machen, sollten wir überprüfen, ob wir die Invariante überhaupt beweisen können. Wir haben schließlich nur den ersten Teil aus der Schleife selbst, den zweiten brauchten wir einfach um hinter

der Schleife stark genug zu sein. Ist die ganze Implementierung falsch könnte das zum Beispiel an diesem Punkt klar werden.

Wir fangen dazu mit der Invariante am Ende des Schleifenrumpfs an und verwenden einfache, mechanische Zuweisungsregeln, um an den Kopf der Schleife zu gelangen. Auch hier gilt wieder, einfach die Variable auf die zugewiesen wird überall in der Zusicherung durch den Ausdruck der Zuweisung ersetzen.

6. Vereinfachungen nehmen wir erst später, in einem eigenen Schritt vor, sonst sind die Regelanwendungen nicht mehr nachvollziehbar. In diesem Fall verrechnen wir einfach erstmal nur die Konstanten. Oben steht ja auch schon eine Zusicherung die wir benutzen dürfen und der erste Teil `a + b == x` ist damit schon bewiesen. Um auch den zweiten nachvollziehbar zu machen, nutzen wir aus, dass auf ganzen Zahlen eine Ungleichung immer in einer echte Ungleichung umgeformt werden kann, indem man auf der passenden Seite eins addiert oder weg lässt (da sollte man genau aufpassen).

   Wie wir sehen entspricht der zweite Teil nun genau dem Teil der Zusicherung, der bei der Schleifenregel aus der Schleifenbedingung folte, und ist damit bewiesen. Das wir den dritten Teil der Zusicherung gar nicht verwendet haben spielt keine Rolle, da wir diesen mit einer Abschwächung einfach wegfallen lassen können. Oben haben wir ja auch schon erwähnt, dass dieser komplett in der Schleifenbedingung enthalten ist.

7. Damit ist die Invariante bewiesen (korrekt) und hat uns erlaubt die Zusicherung nach der Schleife zu beweisen (ausreichend). Jetzt können wir über der Schleife mit der Invariante weiter machen und diese bis zum Anfang der Prozedur ziehen. Dafür braucht es wieder nur Zuweisungsregeln.

8. Dabei werden oft Aussagen trivial und können leicht optimiert werden. Insbesondere fällt **`true`** natürlich weg, solange noch andere Aussagen übrig bleiben. Steht am Ende nur noch **`true`** in der Vorbedingung ist dies ein Glücksfall, da die Funktion dann immer korrekt arbeitet, unabhängig von den Parametern.

   Diese Umformungen machen wir allerdings auch wieder alle erst nach den Zuweisungsregeln in einem eigenen Schritt.

9. In diesem Fall erhalten wir als Vorbedingung und Lösung der Aufgabe also `x > -2`, was wir oben in den dafür vorgesehenen Kommentar bei `\requires` eintragen.

   Man ist hier leicht versucht aus der Ungleichung `x + 1 >= 0` einfach `x > 0` zu machen “weil es hübscher ist”, korrekt ist aber in der Tat die eins auf der anderen Seite zu addieren (bzw. abzuziehen).

Lösungsvorschlag Die ausgefüllte Maske

```
/*
 *  requires: x > -2                                              // (9)
 *  ensures: \result + \result <= x && \result + \result + 1 >= x
 */
static int d2(int x) {

  assert x > -2;                                                  // (8)
  assert x + 0 == x  &&  x + 1 >= 0;                              // (7)

    int a = x;

  assert a + 0 == x  &&  a + 1 >= 0;                              // (7)

    int b = 0;

  assert a + b == x  &&  a + 1 >= b;                              // (3)

    while (a > b) { // INV:  a + b == x  &&  a + 1 >= b           // (2)

      assert a > b  &&  a + b == x  &&  a + 1 >= b;               // (3)
      assert a + b == x  &&  a > b;                               // (6)
      assert a + b == x  &&  a >= b + 1;                          // (6)
      assert a - 1 + b + 1 == x  &&  a - 1 + 1 >= b + 1;          // (5)

        a = a - 1;

      assert a + b + 1 == x  &&  a + 1 >= b + 1;                  // (5)

        b = b + 1;

      assert a + b == x  &&  a + 1 >= b;                          // (3)
    }

  assert a <= b  &&  a + b == x  &&  a + 1 >= b;                  // (3)
  assert a <= b  &&  a + 1 >= b  &&  b == x - a;                  // (4)
  assert a <= x - a  &&  a + 1 >= x - a;                          // (4)
  assert a + a <= x  &&  a + a + 1 >= x;                          // (1)

    return a;

  // ensures: \result + \result <= x && \result + \result + 1 >= x
}
```

Beispiel 2 Quad (alte Übungsaufgabe)

Gegeben ist folgendes Programmfragment:

```
public void f(int N) {
    i = 1;
    s = 0;
    while (i < N) {
        s = s + i * i;
        i = i + 1;
    }
    j = s + 10;
    return j;
}
```

Sie können davon ausgehen, dass `N` aus den natürlichen Zahlen ist, also `N >= 0`.

a) Geben Sie eine möglichst vollständige Invariante an, die beim Betreten der Schleife vor jedem Durchlauf gilt.

b) Geben Sie möglichst viele Eigenschaften an, die direkt nach dem Abarbeiten der Schleife gelten.

c) Geben Sie eine Spezifikation des Programms an und verifizieren Sie es.

Lösungsvorschlag

a) Wir müssen eine Invariante bestimmen, also schauen wir uns die beteiligten Variablen `s` und `i` an, ihre Werte vor der Schleife und wie sie im Rumpf verändert werden. `i` wird immer inkrementiert, während auf `s` das aktuelle Quadrat von `i` addiert wird. Damit ist `s` wohl die Summe der ersten Quadratzahlen, was sich auch mit den Startwerten vereinen lässt. Genau müssen es dann die ersten `i - 1` Quadratzahlen sein, da auch `i` erst nach der Addition auf `s` inkrementiert wird.

   Die Frage ist nun wie wir das spezifizieren, da wir uns in der Vorlesung/Übung auf Spezifikationen mit Java Code beschränken. Denkbar sind auch mathematische Spezifikationen oder Spezifikationen in einer Logik, zum Beispiel der Higher-Order Logik. Zur Zeit dieser Übungsaufgabe wurde auch mathematisch spezifiziert in SE-1.

   Wir bedienen uns aber lieber der geschlossenen Form für die Summe, damit wäre unsere Invariante die folgende:

   ```
   s == (i - 1) * i * (2 * i - 1) / 6
   ```

   Ob die wirklich stimmt, sehen wir ja auch nachher im Beweis, muss man sich also hier nicht drauf verlassen.

   Gefragt war aber auch sehr spezifisch nach dem was am Anfang der Schleife gilt, demnach würde man hier auch die Schleifenbedingung mit angeben, also zusammen

   ```
   i < N  &&  s == (i - 1) * i * (2 * i - 1) / 6
   ```

b) Nach der Schleife gilt unsere Invariante und die negierte Schleifenbedingung, also

   ```
   i >= N  &&  s == (i - 1) * i * (2 * i - 1) / 6
   ```

   In der damaligen Lösung steht, dass `i == N` gilt, was mit dem Inkrement im Rumpf und $\texttt{N} \in \mathbb{N}$ auch fast sinnvoll ist, aber man sollte sich wirklich einfach an die Schleifenregel halten. In diesem Fall würde diese Zusicherung mit `N == 0` schon verletzt werden, da `i == 1` gilt und die Schleife nie durchlaufen wird.

c) Die Frage ist natürlich sehr offen formuliert, jedoch werden Lösungen meist nur akzeptiert, wenn die Spezifikation auch *sinnvoll* ist. In diesem Fall sollten wir nach den Vorüberlegungen als Nachbedingung die folgende nehmen, da auf `s` noch `10` addiert wird.

   ```
   \result == (N - 1) * N * (2 * N - 1) / 6 + 10
   ```

   Als Schleifeninvariante sollten wir aber nach der Anmerkung oben nun auch noch `i <= N` nehmen, damit wir nach der Schleife auch wirklich auf `i == N` rauskommen. Vollständig annotiert sieht das dann so wie auf der nächsten Seite aus.

```
/*
 *  requires: N > 0
 *  ensures: \result  ==  (N - 1) * N * (2 * N - 1) / 6 + 10
 */
public void f(int N) {

  assert N > 0;
  assert 1 <= N;
  assert 0 == 0 * 1 * 1 / 6  &&  1 <= N;
  assert 0 == (1 - 1) * 1 * (2 * 1 - 1) / 6  &&  i <= N;

    i = 1;

  assert 0 == (i - 1) * i * (2 * i - 1) / 6  &&  i <= N;

    s = 0;

  assert s == (i - 1) * i * (2 * i - 1) / 6  &&  i <= N;

    while (i < N) { // INV:  s == (i - 1) * i * (2 * i - 1) / 6  &&  i <= N
      assert i < N  &&  s == (i - 1) * i * (2 * i - 1) / 6  &&  i <= N;
      assert i < N  &&  s == (i - 1) * i * (2 * i - 1) / 6;
      assert i < N  &&  s == i * (2 * i * i - i - 2 * i + 1) / 6;

      assert s == i * (2 * i * i - 3 * i + 1) / 6  &&  i < N;
      assert s == (2 * i * i * i + 2 * i * i + i * i + i - 6 * i * i) / 6  &&  i < N;
      assert s == (i * i + i) * (2 * i + 1) / 6 - i * i  &&  i < N;
      assert (s + i * i) == i * (i + 1) * (2 * i + 1) / 6  &&  i < N;
      assert (s + i * i) ==
          ((i + 1) - 1) * (i + 1) * (2 * (i + 1) - 1) / 6  &&  (i + 1) <= N;

        s = s + i * i;

      assert s == ((i + 1) - 1) * (i + 1) * (2 * (i + 1) - 1) / 6  &&  (i + 1) <= N;

        i = i + 1;

      assert s == (i - 1) * i * (2 * i - 1) / 6  &&  i <= N;
    }

  assert i <= N  &&  s == (i - 1) * i * (2 * i - 1) / 6  &&  i <= N;
  assert i == N  &&  s == (i - 1) * i * (2 * i - 1) / 6;
  assert s == (N - 1) * N * (2 * N - 1) / 6;
  assert s + 10 == (N - 1) * N * (2 * N - 1) / 6 + 10;

    j = s + 10;

  assert j == (N - 1) * N * (2 * N - 1) / 6 + 10;

    return j;

  // ensures: \result  ==  (N - 1) * N * (2 * N - 1) / 6 + 10
}
```

- Die Anzahl der Zwischenschritte ist nicht entscheidend, aber sie sollten nachvollziehbar sein. Am Kopf der Schleife sieht man, dass es sich manchmal lohnt, von beiden Seiten Umformungen zu machen. Die Leerzeile markiert die Stelle an der die beiden zusammen laufen und der Zusammenhang erkennbar wird.
- Erst durch die erweiterte Invariante konnten wir unter der Schleife die Gleichheit annehmen. Dadurch entsteht dann über der Schleife die letztendliche Bedingung, dass `N` echt größer als Null sein muss. Das deckt sich mit dem Sonderfall, der sonst ein Problem wäre. Die Rahmenbedingung aus der Aufgabe reicht also nicht und ist irrelevant.