Aufgabenblatt 2

Ausgabe: 21. April
Abgabe: 02. Mai Punkte: 20

Thema: *Formalisierung von Aussagen, Programmverifikation mit dem Hoare-Kalkül (Teil 1)*

Auf diesem Aufgabenblatt lernen Sie, Aussagen in eine Form zu bringen, die ausreichende mathematische Präzision bietet, um damit den *Zustand* von Programmen zu beschreiben. Das Regelwerk des Hoare-Kalküls ermöglicht dann, die Korrektheit von Programmen zu beweisen.

1. Aufgabe (7 Punkte): Formalisierung von Aussagen

Definieren Sie die folgenden Prädikate.

1.1. (Tut) $teilt(n, m)$ soll ausdrücken, daß die natürliche Zahl n ein Teiler der natürlichen Zahl m ist.

Lösung:
$teilt(n,m) \stackrel{\text{def}}{=} n, m \in \mathbb{N} \wedge \big(\exists k \in \mathbb{N} : k \cdot n = m\big)$
oder
$teilt(n,m) \stackrel{\text{def}}{=} n, m \in \mathbb{N} \wedge (m \bmod n = 0)$

1.2. (1 Punkt) $istGerade(n)$ soll ausdrücken, daß die natürliche Zahl n gerade ist.

Lösung:
$istGerade(n) \stackrel{\text{def}}{=} n \in \mathbb{N} \wedge (n \bmod 2 = 0)$
oder:
$istGerade(n) \stackrel{\text{def}}{=} n \in \mathbb{N} \wedge \big(\exists k \in \mathbb{N} : n = 2k\big)$

Finden Sie für die folgenden natürlichsprachlichen Aussagen mathematische Formalisierungen, die als POST-Conditions im Rahmen des Hoare-Kalküls verwendet werden könnten. Sie dürfen dabei die oben definierten Prädikate verwenden. Nutzen Sie die Ausdruckskraft von Existenz- und Allquantor ($\exists$ und $\forall$) sowie der üblichen boolschen Verknüpfungen.

Alle Arrays in dieser Aufgabe sind N-elementige Arrays natürlicher Zahlen.

1.3. (Tut) Das Array a enthält nur ungerade Zahlen.

Lösung:
$a = (a_0, \ldots, a_{N-1}) \in \mathbb{N}^N \wedge \big(\forall k \in \{0, \ldots, N-1\} : \neg istGerade(a_k)\big)$
Mathematisch gesehen stellen Arrays Tupel dar, sind also Elemente des kartesischen Produktes von Mengen. Wir betrachten nur homogene Arrays, bei denen die Elemente alle von gleichem Typ sind. Für jedes Array tritt daher nur jeweils eine einzige Menge (hier: $\mathbb{N}$) im kartesischen Produkt auf, und zwar in einer der Arraylänge (N) entsprechenden Potenz ($\mathbb{N}^N$). Für die weitere Verwendung werden die Arrayelemente indiziert ($a_0, \ldots, a_{N-1}$).
Eigentlich müßte die Arraylänge N explizit existenzquantifiziert werden, aber unter maximaler mathematischer Präzision leidet schnell die Übersicht. Aus der Verwendung von N ist klar ersichtlich, welche Bedeutung dieser Größe zukommt.

1.4. (Tut) Das Array a enthält mindestens eine ungerade Zahl.

Lösung:
$a = (a_0, \ldots, a_{N-1}) \in \mathbb{N}^N \wedge \big(\exists k \in \{0, \ldots, N-1\} : \neg istGerade(a_k)\big)$

1.5. (1 Punkt) Das Array a ist in aufsteigender Folge sortiert.

Hinweis: Der Begriff „aufsteigend“ entspricht dem mathematischen Terminus „monoton steigend“, nicht „streng monoton steigend“.

Lösung:
$a = (a_0, \ldots, a_{N-1}) \in \mathbb{N}^N \wedge \big(\forall k \in \{1, \ldots, N-1\} : a_{k-1} \le a_k\big)$
oder:
$a = (a_0, \ldots, a_{N-1}) \in \mathbb{N}^N \wedge \big(\forall k \in \{0, \ldots, N-2\} : a_k \le a_{k+1}\big)$
oder:
$a = (a_0, \ldots, a_{N-1}) \in \mathbb{N}^N \wedge \big(a_0 \le a_1 \le \ldots \le a_{N-2} \le a_{N-1}\big)$

1.6. (Tut) m ist im Array a enthalten.

Lösung:
Mit Slice und ohne Quantor:
$a = (a_0, \ldots, a_{N-1}) \in \mathbb{N}^N \wedge m \in a_{0..N-1}$
oder ohne Slice und mit Quantor:
$a = (a_0, \ldots, a_{N-1}) \in \mathbb{N}^N \wedge \big(\exists k \in \{0, \ldots, N-1\} : m = a_k\big)$
Es ist nicht nötig, $m \in \mathbb{N}$ in die Aussage mitaufzunehmen, da es sich zwangsläufig ergibt, d.h. aus der Aussage unmittelbar folgern läßt.

Häufig ist es erforderlich, einfache Aussagen über Arrayinhalte zu machen, die durch Quantoren zwar mathematisch präzise, aber eher unübersichtlich zu formulieren sind.
Zur Vereinfachung führen wir eine Bezeichnungskonvention für Arrayteilbereiche (Slices) ein: $a_{i..j}$ bezeichne das Array $(a_i, a_{i+1}, \ldots, a_{j-1}, a_j)$, falls $j \ge i$ gilt. Für $j < i$ enthält der Slice keine Elemente, bezeichnet also ein nullelementiges Array.
Obwohl Slices ebenso wie Arrays Tupel sind und keine (ungeordneten) Mengen[1] verwenden wir das Zeichen ‚$\in$‘, um auszudrücken, daß ein Wert im Slice vorkommt, z.B. bedeutet $m \in a_{i..j}$, daß mindestens einer der Werte $a_i, a_{i+1}, \ldots, a_j$ gleich m ist, m.a.W.
$m \in a_{i..j} \stackrel{\text{def}}{=} \exists k \in \{i, \ldots, j\} : m = a_k$.
Im Gegensatz zu den Elementen von Mengen sind die Komponenten eines Tupels angeordnet, so daß ein Ausdruck der Form $a_{0..2} = a_{3..5}$ sich auf natürliche Weise als $a_0 = a_3 \wedge a_1 = a_4 \wedge a_2 = a_5$ interpretieren läßt (gleiche Elementanzahlen auf der linken und rechten Seite!).
Wegen des praktischen Nutzens definieren wir außerdem:
$m < a_{i..j} \stackrel{\text{def}}{=} \forall k \in \{i, \ldots, j\} : m < a_k$ *(analog für $\le, >, \ge$)*

1.7. (1 Punkt) m ist das kleinste Element im Array a.

Lösung:
Mit Slices:
$a = (a_0, \ldots, a_{N-1}) \in \mathbb{N}^N \wedge m \in a_{0..N-1} \wedge m \le a_{0..N-1}$
oder mit Quantoren:
$a = (a_0, \ldots, a_{N-1}) \in \mathbb{N}^N$
$\quad \wedge \big(\exists k \in \{0, \ldots, N-1\} : m = a_k\big) \wedge \big(\forall k \in \{0, \ldots, N-1\} : m \le a_k\big)$

[1] Natürlich können Tupel mit Hilfe von Mengen beschrieben werden. Jedoch ist ein Tupel $(a_0, \ldots, a_{N-1})$ nicht das Gleiche wie eine Menge mit den Elementen $a_0, \ldots, a_{N-1}$.

1.8. **(1 Punkt)** m ist im Array a enthalten, stellt aber weder das kleinste noch das größte Element dar.

Lösung:

Mit Slice:
$a = (a_0, \ldots, a_{N-1}) \in \mathbb{N}^N \wedge m \in a_{0..N-1} \wedge \big(\exists k_1, k_2 \in \{0, \ldots, N-1\} : a_{k_1} < m < a_{k_2}\big)$
oder ohne Slice in „kompakter Form“:
$a = (a_0, \ldots, a_{N-1}) \in \mathbb{N}^N \wedge \big(\exists k_1, k_2, k_3 \in \{0, \ldots, N-1\} : a_{k_1} < m = a_{k_2} < a_{k_3}\big)$
oder ohne Slice und ausführlich:
$a = (a_0, \ldots, a_{N-1}) \in \mathbb{N}^N \wedge \big(\exists k_1, k_2, k_3 \in \{0, \ldots, N-1\} : a_{k_1} < m \wedge m = a_{k_2} \wedge m < a_{k_3}\big)$

1.9. **(1 Punkt)** Das Produkt der Elemente des Arrays a ist größer als die Summe seiner Elemente.

Lösung:

Elegant mit Produkt- und Summenzeichen:
$a = (a_0, \ldots, a_{N-1}) \in \mathbb{N}^N \wedge \big(\prod_{k=0}^{N-1} a_k\big) > \big(\sum_{k=0}^{N-1} a_k\big)$
oder ohne Produkt- und Summenzeichen:
$a = (a_0, \ldots, a_{N-1}) \in \mathbb{N}^N \wedge a_0 \cdot a_1 \cdot \cdots \cdot a_{N-2} \cdot a_{N-1} > a_0 + a_1 + \cdots + a_{N-2} + a_{N-1}$

1.10. **(Tut)** g ist der größte gemeinsame Teiler der natürlichen Zahlen n und m.

Lösung:

Es muß ausgedrückt werden, daß g sowohl n als auch m teilt (gemeinsamer Teiler) und daß es keinen gemeinsamen Teiler gibt, der größer ist als g.

Mit Implikation und Allquantor:
$teilt(g,n) \wedge teilt(g,m) \wedge \big(\forall g' \in \mathbb{N} : teilt(g',n) \wedge teilt(g',m) \Rightarrow g' \leq g\big)$
oder ohne Implikation:
$teilt(g,n) \wedge teilt(g,m) \wedge \big(\forall g' \in \mathbb{N} : \neg\big(teilt(g',n) \wedge teilt(g',m)\big) \vee g' \leq g\big)$
oder mit De Morgan aufgelöst:
$teilt(g,n) \wedge teilt(g,m) \wedge \big(\forall g' \in \mathbb{N} : \neg teilt(g',n) \vee \neg teilt(g',m) \vee g' \leq g\big)$
oder mit negiertem Existenzquantor:
$teilt(g,n) \wedge teilt(g,m) \wedge \neg\big(\exists g' \in \mathbb{N} : teilt(g',n) \wedge teilt(g',m) \wedge g' < g\big)$

1.11. **(1 Punkt)** k ist das kleinste gemeinsame Vielfache der natürlichen Zahlen n und m.

Lösung:

Mit Implikation und Allquantor:
$k > 0 \wedge teilt(n,k) \wedge teilt(m,k) \wedge \big(\forall k' \in \mathbb{N} \setminus \{0\} : teilt(n,k') \wedge teilt(m,k') \Rightarrow k' \geq k\big)$
oder ohne Implikation:
$k > 0 \wedge teilt(n,k) \wedge teilt(m,k) \wedge \big(\forall k' \in \mathbb{N} \setminus \{0\} : \neg\big(teilt(n,k') \wedge teilt(m,k')\big) \vee k' \geq k\big)$
oder mit De Morgan aufgelöst:
$k > 0 \wedge teilt(n,k) \wedge teilt(m,k) \wedge \big(\forall k' \in \mathbb{N} \setminus \{0\} : \neg teilt(n,k') \vee \neg teilt(m,k') \vee k' \geq k\big)$
oder mit negiertem Existenzquantor:
$k > 0 \wedge teilt(n,k) \wedge teilt(m,k) \wedge \neg\big(\exists k' \in \mathbb{N} \setminus \{0\} : teilt(n,k') \wedge teilt(m,k') \wedge k' < k\big)$

1.12. **(1 Punkt)** p ist eine Primzahl.

Lösung:

$p \geq 2 \wedge \forall k \in \{2, \ldots, p-1\} : \neg teilt(k,p)$
oder „sparsamer“:
$p \geq 2 \wedge \forall k \in \{2, \ldots, \lfloor\sqrt{p}\rfloor\} : \neg teilt(k,p)$
oder:
$p \neq 1 \wedge \big(\forall k \in \mathbb{N} : \neg teilt(k,p) \vee k = 1 \vee k = p\big)$

2. Aufgabe (13 Punkte): Hoare-Kalkül für Einsteiger

2.1. **(Tut)** Diskutieren Sie die Grundlagen des Hoare-Kalküls.

Lösung:

In den Skripten „Korrektheit von Programmen“ und „Einführung in die Programmverifikation“ werden umfassende Erläuterungen gegeben.

2.2. **(Tut)** Das folgende Programmfragment berechnet die Fakultät einer natürlichen Zahl. Beweisen Sie mit Hilfe des Hoare-Kalküls seine totale Korrektheit.

```
1  /*
2   * SPC   f = factorial(n)
3   * PRE   n ≥ 0
4   * POST  f = ∏_{k=1}^{n} k
5   */
6  public static int  factorial ( int n )
7  {
8    int f, k;
9
10   f = 1;
11   k = 1;
12
13   while( k <= n )
14   {
15     f = f*k;
16     k = k+1;
17   }
18
19   return f;
20 }
```

Hinweis: Per Konvention gilt: $\prod_{k=k_1}^{k_2} k = 1$, falls $k_2 < k_1$.

Lösung:

```
1  public static int  factorial ( int n )
2  {
3    // PRE: n ≥ 0
4
5    int f, k;
6
7    // PRE ∧ 1 = 1
8    f = 1;                                              [Zuweisungsaxiom]
9    // PRE ∧ f = 1
10   // PRE ∧ f = 1 ∧ 1 = 1
11   k = 1;                                              [Zuweisungsaxiom]
12   // (PRE : n ≥ 0) ∧ f = 1 ∧ k = 1
13   // 1 ≤ n + 1 ∧ f = 1 ∧ k = 1 ∧ 1 = ∏_{i=1}^{0} i
14   // I : k ≤ n + 1 ∧ f = ∏_{i=1}^{k-1} i              [Invariante für Iterationsregel]
15   while( k <= n )                                     [Iterationsregel]
16   {
17     // (B : k ≤ n) ∧ (I : k ≤ n + 1 ∧ f = ∏_{i=1}^{k-1} i)   ∧ τ(k) ≡ n − k + 1 = c
18     // B ∧ f = ∏_{i=1}^{k-1} i                                ∧ n − k + 1 = c
```

```
19    // B ∧ f · k = (∏_{i=1}^{k-1} i) · k = ∏_{i=1}^{k} i                    ∧ n − k + 1 = c
20    f = f*k;                                                              [Zuweisungsaxiom]
21    // (B : k ≤ n) ∧ f = ∏_{i=1}^{k} i                                      ∧ n − k + 1 = c
22    // (k + 1) − 1 ≤ n ∧ f = ∏_{i=1}^{(k+1)−1} i                            ∧ n − (k + 1) + 2 = c
23    k = k+1;                                                              [Zuweisungsaxiom]
24    // k − 1 ≤ n ∧ f = ∏_{i=1}^{k−1} i                                      ∧ n − k + 2 = c
25    // (I : k ≤ n + 1 ∧ f = ∏_{i=1}^{k−1} i)                                ∧ n − k + 2 = c
26  }
27  // (¬B : ¬(k ≤ n)) ∧ (I : k ≤ n + 1 ∧ f = ∏_{i=1}^{k−1} i)                [Iterationsregel]
28  // n < k ≤ n + 1 ∧ f = ∏_{i=1}^{k−1} i
29  // k = n + 1 ∧ f = ∏_{i=1}^{k−1} i
30  // f = ∏_{i=1}^{(n+1)−1} i
31  return f;
32  // POST: f = ∏_{i=1}^{n} i = n!
33 }
```

Ohne die fett gedruckten Teile der Zusicherungen wird lediglich ein Beweis im Rahmen des Kalküls partieller Korrektheit geführt. Um die totale Korrektheit zu beweisen, muß außerdem die Terminierung der Schleife mit der Terminierungsregel bewiesen werden:

1. *Am Beginn des Schleifenrumpfes gilt $\tau(k) \equiv n - k + 1 = c$, d.h. $\tau(k) = c$. Dabei ist c eine ad hoc eingeführte Größe, die gerade dazu dient, den Wert von $\tau(k)$ am Beginn des Schleifenrumpfes zu „protokollieren". Für einen Durchlauf des Schleifenrumpfes ist c also eine Konstante. Beim nächsten Durchlauf hat c dann einen neuen Wert.*
2. *Am Ende des Schleifenrumpfes gilt $c = n - k + 2$ und $\tau(k) \equiv n - k + 1 < n - k + 2 = c$, d.h. $\tau(k) < c$, also wird $\tau(k)$ im Schleifenrumpf echt kleiner. Man beachte, daß sich der Wert von c gegenüber Punkt 1 nicht verändert hat, wohl aber der von k.*
3. *$\tau(k) \leq 0 \Leftrightarrow n - k + 1 \leq 0 \Leftrightarrow k \geq n + 1 \Leftrightarrow k > n \Leftrightarrow \neg(k \leq n) \Leftrightarrow \neg B$, also $\tau(k) \leq 0 \Rightarrow \neg B$, d.h. wenn $\tau(k)$ den Wert Null erreicht (oder unterschreitet), wird die Schleifeneintrittsbedingung verletzt und die Schleife somit verlassen.*
4. *Der Schleifenrumpf terminiert, da es keine Schleifen innerhalb der Schleife gibt und keine evtl. nicht-terminierenden Methodenaufrufe.*

Anmerkungen:

- *In den Zeilen 7 und 10 sind die „$1 = 1$"-Ausdrücke nicht redundant, sondern ergeben sich zwangsläufig aus dem Zuweisungsaxiom.*
- *Die Konvention, daß leere Produkte (Obergrenze kleiner als Untergrenze) den Wert 1 zugeordnet bekommen (neutrales Element der Multiplikation) sichert die Gültigkeit der Invariante vor Eintritt in die Schleife (Zeile 13/14).*
- *Bei Beweisführung von oben nach unten mag der Ausdruck $k \leq n + 1$ in der Invariante zunächst überflüssig erscheinen, da er aus $B : k \leq n$ unmittelbar folgt und daher schon beim Übergang von Zeile 17 zu 18 direkt nach Schleifeneintritt „weggefolgert" wird. Seine Bedeutung wird erst nach der Schleife erkennbar, wo ohne diesen Ausdruck lediglich auf $k > n$ geschlossen werden könnte, aber nicht auf $k = n + 1$.*
  Theoretischer Hintergrund ist die Iterationsregel, die u.a. besagt, daß die POST-Condition der Schleife gerade $\neg B \wedge I$ lautet. Es ist grundsätzlich nicht zulässig, Zusicherungen aus der Schleife „hinaus zu retten", es sei denn, sie sind Teil der Invariante oder der negierten Schleifeneintrittsbedingung. Das ist kein Mangel der Iterationsregel, sondern eine Konsequenz der Tatsache, daß u.U. gar kein Schleifeneintritt erfolgt und es daher logisch grob falsch wäre, sich auf Zusicherungen zu stützen, die auf nicht-ausgeführtem Code basieren.
  Falls die Beweisführung zielgerichtet von unten nach oben durchgeführt wird, ergibt sich die Notwendigkeit des zusätzlichen Terms recht zwanglos.

2.3. (1 Punkt) Erläutern Sie kurz den Zusammenhang zwischen der textuellen Aufgabenbeschreibung, der formalen Spezifikation und dem Korrektheitsbeweis eines Programmes (siehe auch Skript „Korrektheit von Programmen"). Welche Bedeutung hat der Begriff „Korrektheit" in diesem Kontext?

Lösung:

*Die Spezifikation stellt eine Formalisierung der textuellen Aufgabenbeschreibung dar. Der Korrektheitsbeweis (z.B. mit dem Hoare-Kalkül) weist mit mathematischer Strenge nach, daß das Programm diese Spezifikation (SPC, PRE, POST) erfüllt (*Verifikation*). Dadurch allein ist jedoch noch nicht gesichert, daß das Programm die textuell beschriebene Aufgabe tatsächlich löst. Besonders bei komplexen Aufgabenstellungen ist es erforderlich, nachzuweisen, daß die Spezifikation die Aufgabe adäquat beschreibt (*Validierung *der Spezifikation), wozu sich der Hoare-Kalkül jedoch nicht eignet. In diesem Sinn bezieht sich der Begriff „Korrektheit" ausschließlich auf die korrekte Erfüllung der Spezifikation durch das Programm.*

2.4. Quadratzahlen (5 Punkte) Implementieren Sie (schriftlich) eine Methode, die die Summe der ersten n Quadratzahlen berechnet und beweisen Sie mit Hilfe des Hoare-Kalküls ihre totale Korrektheit.

```
1  /*
2   * SPC   s = sqsum(n)
3   * PRE   n ≥ 0
4   * POST  s = ∑_{k=0}^{n−1} k²
5  */
6  public static int sqsum( int n )
7  {
8    // ???
9  }
```

Hinweis: Per Konvention gilt: $\sum_{k=k_1}^{k_2} k = 0$, falls $k_2 < k_1$.

Lösung:

Die Lösung der Tutoriumsaufgabe läßt sich weitestgehend übertragen:

```
1  /*
2   * SPC   s = sqsum(n)
3   * PRE   n ≥ 0
4   * POST  s = ∑_{k=0}^{n−1} k²
5  */
6  public static int sqsum( int n )
7  {
8    // PRE: n ≥ 0
9    int s, k;
10
11   // PRE ∧ 0 = 0
12   s = 0;                                                    [Zuweisungsaxiom]
13   // PRE ∧ s = 0
14   // PRE ∧ s = 0 ∧ 0 = 0
15   k = 0;                                                    [Zuweisungsaxiom]
16   // PRE ∧ s = 0 ∧ k = 0
17   // (PRE : n ≥ 0) ∧ s = 0 ∧ k = 0 ∧ 0 = ∑_{i=0}^{−1} i²
18   // k ≤ n ∧ s = ∑_{i=0}^{k−1} i²
```

```
19
20    // I : k ≤ n ∧ s = Σ_{i=0}^{k-1} i²                          [Invariante für Iterationsregel]
21    while( k < n )                                              [Iterationsregel]
22    {
23      // (B : k < n) ∧ (I : k ≤ n ∧ s = Σ_{i=0}^{k-1} i²)        ∧ τ(k) ≡ n − k = c
24      // B ∧ s = Σ_{i=0}^{k-1} i²                                ∧ n − k = c
25      // B ∧ s + k · k = (Σ_{i=0}^{k-1} i²) + k² = Σ_{i=0}^{k} i²   ∧ n − k = c
26      s = s + k*k;                                              [Zuweisungsaxiom]
27      // (B : k < n) ∧ s = Σ_{i=0}^{k} i²                        ∧ n − k = c
28      // (k + 1) − 1 < n ∧ s = Σ_{i=0}^{(k+1)-1} i²              ∧ n − (k + 1) + 1 = c
29      k = k + 1;                                                [Zuweisungsaxiom]
30      // k − 1 < n ∧ s = Σ_{i=0}^{k-1} i²                        ∧ n − k + 1 = c
31      // (I : k ≤ n ∧ s = Σ_{i=0}^{k-1} i²)                      ∧ n − k + 1 = c
32    }
33    // (¬B : ¬(k < n)) ∧ (I : k ≤ n ∧ s = Σ_{i=0}^{k-1} i²)      [Iterationsregel]
34    // k = n ∧ s = Σ_{i=0}^{k-1} i²
35    return s;
36    // POST: s = Σ_{i=0}^{n-1} i²
37  }
```

Die Terminierung der Schleife ist gesichert:

1. *Am Beginn des Schleifenrumpfes gilt $\tau(k) \equiv n - k = c$, d.h. $\tau(k) = c$.*
2. *Am Ende des Schleifenrumpfes gilt $c = n - k + 1$ und $\tau(k) \equiv n - k < n - k + 1 = c$, d.h. $\tau(k) < c$, also wird $\tau(k)$ im Schleifenrumpf echt kleiner.*
3. *$\tau(k) \leq 0 \Leftrightarrow n - k \leq 0 \Leftrightarrow k \geq n \Leftrightarrow \neg(k < n) \Leftrightarrow \neg B$, also $\tau(k) \leq 0 \Rightarrow \neg B$, d.h. wenn $\tau(k)$ den Wert Null erreicht (oder unterschreitet), wird die Schleifeneintrittsbedingung verletzt und die Schleife somit verlassen.*
4. *Der Schleifenrumpf terminiert, da es keine Schleifen innerhalb der Schleife gibt und keine evtl. nicht-terminierenden Methodenaufrufe.*

2.5. Arrayreversion (7 Punkte) Beschreiben Sie kurz umgangssprachlich den zugrundeliegenden Algorithmus des nachfolgenden Programmfragments.

Beweisen Sie dann mit Hilfe des Hoare-Kalküls, daß die Spezifikation erfüllt wird (partielle Korrektheit). Zur Schreibersparnis dürfen Sie dabei auf die Niederschrift der Beweisschritte innerhalb des Schleifenrumpfes verzichten, jedoch nicht auf die Invariante.

Um die Notation zu vereinfachen, definieren wir für die Arrays $a = (a_0, \ldots, a_{N-1})$ und $A = (A_0, \ldots, A_{N-1})$ sowie ihre gemeinsame Länge N für alle $i, j \in \{0, \ldots, N-1\}$ zwei Prädikate:

$Fwd(i,j) \stackrel{\text{def}}{=} \forall k \in \{i, \ldots, j\} : \; a_k = A_k$
$Bwd(i,j) \stackrel{\text{def}}{=} \forall k \in \{i, \ldots, j\} : \; a_k = A_{N-k-1}$

Dabei gilt $\{i, \ldots, j\} = \emptyset$, falls $i > j$.

a und A können als implizite Argumente angesehen werden, d.h. bei der Verwendung von Fwd und Bwd müssen sie nicht explizit angegeben werden.

```
1  /*
2   * SPC   reverse(a)
3   * PRE   a = (a_0, ..., a_{N-1}) = (A_0, ..., A_{N-1}),  d.h. Fwd(0, N − 1)
4   * POST  a = (a_0, ..., a_{N-1}) = (A_{N-1}, ..., A_0),  d.h. Bwd(0, N − 1)
5   */
6  void reverse ( int [] a )
```



```
7  {
8    final int N = a.length;  // N steht im Beweis als Konstante zur Verfügung
9    int t;
10
11   // PRE: Fwd(0, N − 1)
12
13   i = 0;
14
15   while( i < N/2 )
16   {
17     t          = a[i];
18     a[i]       = a[N−i−1];
19     a[N−i−1] = t;
20     i = i + 1;
21   }
22   // POST: Bwd(0, N − 1)
23 }
```

Hinweis: Das Wesentliche an der Aufgabe ist das Finden der korrekten Invariante sowie einer mathematisch schlüssigen Argumentation, daß daraus (und der negierten Schleifeneintrittsbedingung) unter allen Umständen die POST-Condition folgt.

Beachten Sie, daß bei der Ganzzahldivision in Richtung Null gerundet wird. Beispielsweise wird in Programmzeile 11 der Ausdruck $\lfloor N/2 \rfloor$ gebildet ($N \geq 0$). Es könnte nützlich sein, zu wissen, daß für alle ganzen Zahlen n gilt: $\lfloor n/2 \rfloor + \lceil n/2 \rceil = n$.

Lösung:

Es findet eine Wanderung von den Außenrändern des Arrays nach innen zur Mitte hin statt. Dabei wird sukzessive das erste mit dem letzten Element vertauscht, dann das zweite mit dem vorletzten usw. bis der Arrayinhalt rückwärts angeordnet ist.

Die Schleifeninvariante kann gefunden werden durch die Beobachtung, daß stets drei Bereiche im Array existieren, nämlich ein unveränderter Kernbereich und zwei Randbereiche (links und rechts). Die aktuellen Elemente eines Randbereiches sind dabei immer umgekehrt angeordnet wie die ursprünglichen Elemente des anderen Randbereichs.

Anfänglich sind die Randbereiche leer und der Kernbereich umfaßt das gesamte Array. In gleichem Maße wie der Kernbereich schrumpft, nehmen dann die Randbereiche zu bis sie sich (fast) berühren und der Kernbereich nur noch ein Element umfaßt (ungerade Arraygröße) oder sogar auf Nulllänge abnimmt (gerade Arraygröße).

Die Eigenschaft des Kernbereichs, unverändert zu sein, läßt sich durch das Prädikat Fwd ausdrücken, zur Beschreibung der Randbereichszustände ist Bwd hilfreich. Dadurch kann der etwas mühselige Einsatz von Quantoren vermieden werden und die Zusicherungen werden kürzer (sind allerdings immer noch recht lang).

```
1  void reverse ( int [] a )
2  {
3    final int N = a.length;
4    int t;
5
6    // PRE: Fwd(0, N − 1)
7
8    // PRE ∧ 0 = 0
9    i = 0;
10   // PRE ∧ i = 0
```

```
11  // i = 0 ∧ Bwd(0,−1) ∧ (PRE : Fwd(0, N − 1)) ∧ Bwd(N, N − 1)
12  //    (zur Erinnerung: Bwd(0,−1) und Bwd(N, N − 1) sind wahr,
13  //     weil sie Aussagen über den Inhalt leerer Bereiche machen)
14
15  // I : i ≤ ⌊N/2⌋ ∧ Bwd(0, i − 1) ∧ Fwd(i, N − i − 1) ∧ Bwd(N − i, N − 1)
16  while( i < N/2 )
17  {
18    // (B : i < ⌊N/2⌋)∧(I : i ≤ ⌊N/2⌋∧Bwd(0,i−1)∧Fwd(i,N−i−1)∧Bwd(N−i,N−1))
19    // B ∧ Bwd(0, i − 1) ∧ Fwd(i, N − i − 1) ∧ Bwd(N − i, N − 1)
20    // B ∧ Bwd(0, i − 1) ∧ a_i = A_i ∧ Fwd(i + 1, N − i − 1) ∧ Bwd(N − i, N − 1)
21    t = a[i];
22    // B ∧ Bwd(0, i − 1) ∧ t = A_i ∧ Fwd(i + 1, N − i − 1) ∧ Bwd(N − i, N − 1)
23    // B∧Bwd(0,i−1)∧t = A_i∧Fwd(i+1, N−i−2)∧a_{N−i−1} = A_{N−i−1}∧Bwd(N−i,N−1)
24    a[i] = a[N−i−1];
25    // B ∧ Bwd(0, i − 1) ∧ t = A_i ∧ Fwd(i + 1, N − i − 2) ∧ a_i = A_{N−i−1} ∧ Bwd(N − i, N − 1)
26    // B ∧ Bwd(0, i) ∧ t = A_i ∧ Fwd(i + 1, N − i − 2) ∧ Bwd(N − i, N − 1)
27    a[N−i−1] = t;
28    // B ∧ Bwd(0, i) ∧ a_{N−i−1} = A_i ∧ Fwd(i + 1, N − i − 2) ∧ Bwd(N − i, N − 1)
29    // (B : i < ⌊N/2⌋) ∧ Bwd(0, i) ∧ Fwd(i + 1, N − i − 2) ∧ Bwd(N − i − 1, N − 1)
30    // (i + 1) ≤ ⌊N/2⌋ ∧ Bwd(0, (i + 1) − 1) ∧ Fwd((i + 1), N − (i + 1) − 1)
31    //     ∧ Bwd(N − (i + 1), N − 1)
32    i = i + 1;
33    // I : i ≤ ⌊N/2⌋ ∧ Bwd(0, i − 1) ∧ Fwd(i, N − i − 1) ∧ Bwd(N − i, N − 1)
34  }
35  // (¬B : ¬(i < ⌊N/2⌋)) ∧ (I : i ≤ ⌊N/2⌋ ∧ Bwd(0, i − 1) ∧ Fwd(i, N − i − 1)
36  //     ∧ Bwd(N − i, N − 1))
37  // i ≥ ⌊N/2⌋ ∧ i ≤ ⌊N/2⌋ ∧ Bwd(0, i − 1) ∧ Fwd(i, N − i − 1) ∧ Bwd(N − i, N − 1)
38  // i = ⌊N/2⌋ ∧ Bwd(0, i − 1) ∧ Fwd(i, N − i − 1) ∧ Bwd(N − i, N − 1)
39  // Bwd(0, ⌊N/2⌋ − 1) ∧ Fwd(⌊N/2⌋, N − ⌊N/2⌋ − 1) ∧ Bwd(N − ⌊N/2⌋, N − 1)
40  // Bwd(0, ⌊N/2⌋ − 1) ∧ Fwd(⌊N/2⌋, ⌈N/2⌉ − 1) ∧ Bwd(⌈N/2⌉, N − 1)
41  // POST: Bwd(0, N − 1)    (siehe nachfolgende Erläuterung)
42 }
```

Der Übergang von Zeile 39 zu 40 läßt sich durch eine Fallunterscheidung erklären:

1. *N ist gerade*

   Dann gilt $\lfloor N/2 \rfloor = \lceil N/2 \rceil = N/2$ *und es ergibt sich:*

   $$\begin{aligned} & Bwd(0, \lfloor N/2 \rfloor - 1) \wedge Fwd(\lfloor N/2 \rfloor, \lceil N/2 \rceil - 1) \wedge Bwd(\lceil N/2 \rceil, N - 1) \\ \Leftrightarrow\ & Bwd(0, N/2 - 1) \wedge Fwd(N/2, N/2 - 1) \wedge Bwd(N/2, N - 1) \\ \Leftrightarrow\ & Bwd(0, N/2 - 1) \wedge Bwd(N/2, N - 1), \text{ da } Fwd \text{ über einen leeren Bereich spricht} \\ \Leftrightarrow\ & Bwd(0, N - 1) \end{aligned}$$

2. *N ist ungerade*

   Dann gilt $\lfloor N/2 \rfloor = \lceil N/2 \rceil - 1 = (N - 1)/2$ *und es ergibt sich:*

   $$\begin{aligned} & Bwd(0, \lfloor N/2 \rfloor - 1) \wedge Fwd(\lfloor N/2 \rfloor, \lceil N/2 \rceil - 1) \wedge Bwd(\lceil N/2 \rceil, N - 1) \\ \Leftrightarrow\ & Bwd(0, (N-1)/2 - 1) \wedge Fwd((N-1)/2, (N-1)/2) \wedge Bwd((N-1)/2 + 1, N - 1) \\ \Leftrightarrow\ & Bwd(0, (N-1)/2 - 1) \wedge Bwd((N-1)/2, (N-1)/2) \wedge Bwd((N-1)/2 + 1, N - 1) \\ \Leftrightarrow\ & Bwd(0, N - 1) \end{aligned}$$

Dabei wurde ausgenutzt, daß wegen $N - (N-1)/2 - 1 = (N-1)/2$ *gilt:*

$$\begin{aligned} & Fwd((N-1)/2, (N-1)/2) \\ \Leftrightarrow\ & a_{(N-1)/2} = A_{(N-1)/2} \\ \Leftrightarrow\ & a_{(N-1)/2} = A_{N-(N-1)/2-1} \\ \Leftrightarrow\ & Bwd((N-1)/2, (N-1)/2) \end{aligned}$$

2.6. Zusatzaufgabe (0 Punkte) Ergänzen Sie Ihre Lösung von Aufgabe 2.5 um die Beweisschritte für den Schleifenrumpf.

Lösung:

Die Lösung ist in der Lösung von Aufgabe 2.5 enthalten.

2.7. Zusatzaufgabe (0 Punkte) Der Gaststudent Joe B. Clever ersetzt im Programmcode zu Aufgabe 2.5 in Zeile 11 die Schleifeneintrittsbedingung durch `i <= N/2`, testet fleissig und stellt fest, daß das Programm wie erwünscht arbeitet. Was läßt sich über die Auswahl seiner Testwerte aussagen?

Lösung:

Der gute Joe hat wohl eine Vorliebe für ungerade Arraylängen. Die zusätzliche Vertauschung des mittleren Elementes macht sich dann nicht bemerkbar. Vielleicht hat er auch mal eine gerade Arraylänge getestet, aber dabei ungeschickterweise die beiden Elemente links und rechts der (gedachten) Mitte gleich gewählt.

2.8. Zusatzaufgabe (0 Punkte) Joe schickt das Programm an seinen Bruder Jim, der beschließt, den Schleifeneintritt durch `i < (N − 1)/2` zu regeln. Das klappt scheinbar ganz gut. Ist er klüger als Joe?

Lösung:

Auch er sollte sich mal für gerade Arraylängen interessieren. Während Joe die mittleren Elemente zweimal vertauscht, vertauscht Jim sie gar nicht. Das Ergebnis ist das Gleiche. Beiden sei die Teilnahme an der Informatik-2-Veranstaltung empfohlen, wo sie lernen können, Beweise mit dem Hoare-Kalkül zu führen, auf daß ihre Programme immer korrekt sein mögen ☺