Principles of Software Practice Problems

Part 1, Set 1

February 2020

1. Hoare Logic. Are the following Hoare triples valid or invalid (not necessarily valid)? Circle the correct answer for every triple. If a Hoare triple is valid, prove it. If it is invalid (not necessarily valid), present an example that shows it. Then suggest how a precondition or a postcondition can be changed to make this Hoare triple valid without changing the code.

1. `{false} x = 3; { x == 8 }`
   (a) Valid (b) Invalid

2. `{true} x = y + 1; {false}`
   (a) Valid (b) Invalid

3. `{x == 5} x = x * 2; { x == 10 || x == 5 }`
   (a) Valid (b) Invalid

4. `{y == y + 1} x = y + 1; {y == x}`
   (a) Valid (b) Invalid

5. `{x - y > 3} x = x - y; {x > 2}`
   (a) Valid (b) Invalid

6. `{y > 2} x = y - 1; z = x + 1; {z > 3}`
(a) Valid (b) Invalid

7. `{x == y} if (x < y) x = y - 1; {x >= y}`
(a) Valid (b) Invalid

8. `{x % y != 0} if (x < y) x = y - 1; {x >= y}`
(a) Valid (b) Invalid

9. `{x == y} if (x == 0) x = y + 1; else z = y + 1; {(x == y + 1) || (z == x + 1))}`
(a) Valid (b) Invalid

10. `{x == 0} while (x == 0) x = x + 1; {x == 1}`
(a) Valid (b) Invalid

11. `{x == a && y == b} x = x + y; y = x - y; x = x - y; {x == b && y == a}`
(a) Valid (b) Invalid

2. Forward Reasoning.

Compute the strongest postcondition using forward reasoning. Fill in all intermediate conditions at the designated places. Simplify your final postcondition. Assume all variables are integers.

2.1.

```
{ x > 0 }
x = 10;
{ x = 10 }
y = 2 * x;
{________________________________}
z = y + 4;
{________________________________}
x = z / 2;
{________________________________}
y = 0;
{________________________________}
```

2.2.

```
{ x > 0 }
y = x;
{________________________________}
y = y + 2;
{________________________________}
```

2.3.

```
{ |x| > 10 }
x = -x;
{________________________________}
x = x / 2;
{________________________________}
x = x + 1;
{________________________________}
```

2.4.

```
{ y > 2 * x }
y = y * 2;
{________________________________}
x = x + 1;
{________________________________}
```

2.5.

```
{ x < -3 && y == x }
x = x - 4;
{________________________________}
y = x + abs(x);
{________________________________}
z = (y + 5) * (x + 2);
{________________________________}
```

2.6.

```
{ |x| > 5 }
if (x > 0)
    {________________________________}
    x = 3 - x;
    {________________________________}
}
else {
    {________________________________}
    x = x - 1;
    {________________________________}
}
{________________________________}
```

2.7.

```
{ x > 0 && z > 4 }
if (x < 5 && z < x)
    {________________________________}
    y = z + x;
    {________________________________}
}
{________________________________}
```

3. Backward Reasoning. Compute the weakest precondition using backward reasoning. Fill in all intermediate conditions at the designated places. Simplify your weakest precondition. Assume all variables are integers.

3.1.

```
{____________________________}
x = x + 5;
{____________________________}
y = 2 * x;
{ y > 10 }
```

3.2.

```
{____________________________}
y = x + 6;
{____________________________}
z = x + y;
{ z <= 0 }
```

3.3.

```
{____________________________}
y = w - 10;
{____________________________}
x = 2 * x;
{ x > y }
```

3.4.

```
{____________________________}
t = 2 * s;
{____________________________}
r = w + 4;
{____________________________}
s = 2 * s + w;
{ r > s && s > t }
```

3.5.

```
{___________________________}
w = 2 * w;
{___________________________}
z = -w;
{___________________________}
y = v + 1;
{___________________________}
x = min(y, z);
{ x < 0 }
```

3.6.

```
{___________________________}
if (x == y) {
    {___________________________}
    x= -1;
    {___________________________}
} else {
    {___________________________}
    x = y - 1;
    {___________________________}
}
{ x < y }
```

3.7.

```
{___________________________}
if (x > 0) {
    {___________________________}
    x = x + 6;
    {___________________________}
} else {
    {___________________________}
    x = x / 2;
    {___________________________}
}
{ |x| < 7 }
```

4. Loops.

4.1. Consider the loop in `mul()`. `mul()` requires `a >= 0 && b >= 0` and returns $a \times b$. For example, `mul(5, 3) == 15`, `mul(5, 0) == 0`, and `mul(0, 9) == 0`.

```
//precondition: a >= 0 && b >= 0
int mul(int a, int b) {
   int x = 0;
   int p = 0;
   while (p < b) {
         x = x + a;
         p = p + 1;
   }
   return x;
}
// postcondition: x ==  a * b
```

Verify correctness of `mul()`. Optionally, convert your code to Dafny syntax to check if it would verify with the loop invariant and the decrementing function that you found.

Use steps below to carry out the proof:

(a) Find the loop invariant and show that it is true for the base case before the loop executes.

(b) Show that it holds for the general case. That is, assume it holds for iteration m and show that it holds for iteration $m + 1$.

(c) Show that at exit, the loop invariant and the exit condition imply the postcondition.

(d) Complete the total correctness proof by showing that loop terminates. Make sure you clearly show your decrementing function D and carry out all steps of the proof.

4.2. Now, consider the loop in a different version of `mul()`. Note that the precondition of this `mul()` is different from the previous one as it only requires `a >= 0`. The returned value is still $a \times b$. For example, `mul(5, 3) == 15`, `mul(5, -9) == -45`, and `mul(0, -9) == 0`.

```
//precondition: a >= 0
int mul(int a, int b) {
  int x = a, y = b, total = 0;
  while (x > 0) {
    if(x % 2 == 1) {
      x = (x - 1) / 2;
      total = total + y;
    } else {
       x = x / 2;
    }
    y = y * 2;
  }
  return total;
}
// postcondition: total ==  a * b
```

Verify correctness of `mul()`. Optionally, convert your code to Dafny syntax to check if it would verify with the loop invariant and the decrementing function that you found.

Use steps below to carry out the proof:

(a) Find the loop invariant and show that it is true for the base case before the loop executes.

(b) Show that it holds for the general case. That is, assume it holds for iteration m and show that it holds for iteration $m + 1$.

(c) Show that at exit, the loop invariant and the exit condition imply the postcondition.

(d) Complete the total correctness proof by showing that loop terminates. Make sure you clearly show your decrementing function D and carry out all steps of the proof.

4.3. Finally, consider the version of `mul()` which uses only increment and decrement operations to implement multiplication but contains nested loops. This `mul()` requires `a >= 0 && b >= 0` and returns $a \times b$. For example, `mul(5, 3) == 15`, `mul(5, 0) == 0`, and `mul(0, 9) == 0`.

```
//precondition: a >= 0 && b >= 0
int mul(int a, int b) {
  int x = a, total = 0;
  while (x > 0) {
    int y = b;
    while (y > 0) {
      total = total + 1;
      y = y - 1;
    }
    x = x - 1;
  }
  return total;
}
// postcondition: total ==  a * b
```

Verify correctness of `mul()` by proving the correctness of the inner loop and the outer loop. Optionally, convert your code to Dafny syntax to check if it would verify with the loop invariants and decrementing functions that you found.

Use the steps below to carry out the proof separately for the inner loop and the outer loop:

(a) Find the loop invariant and show that it is true for the base case before the loop executes.

(b) Show that it holds for the general case. That is, assume it holds for iteration m and show that it holds for iteration $m + 1$.

(c) Show that at exit, the loop invariant and the exit condition imply the postcondition.

(d) Complete the total correctness proof by showing that loop terminates. Make sure you clearly show your decrementing function D and carry out all steps of the proof.

4.4. Consider the loop in `computeFib()`. `computeFib()` requires `n >= 0` and returns F_n (the Fibonacci number of n). For example, $computeFib(0) == 0$, $computeFib(1) == 1$, and $computeFib(8) == 21$.

```
//precondition: n >= 0
int computeFib(int n) {
  int i = 0, k = 1, f = 0, t;
  while (i < n) {
    t = f;
    f = k;
    k = t + k;
    i = i + 1;
  }
  return f;
}
// postcondition: f == fib(n)
```

Verify correctness of `computeFib()`. Optionally, convert your code to Dafny syntax to check if it would verify with the loop invariant and the decrementing function that you found.

Use steps below to carry out the proof:

(a) Find the loop invariant and show that it is true for the base case before the loop executes.

(b) Show that it holds for the general case. That is, assume it holds for iteration m and show that it holds for iteration $m + 1$.

(c) Show that at exit, the loop invariant and the exit condition imply the postcondition.

(d) Complete the total correctness proof by showing that loop terminates. Make sure you clearly show your decrementing function D and carry out all steps of the proof.

4.5. Consider the loop in `sumn()`. `sumn()` requires `n >= 0` and returns $\binom{n+1}{2}$ (the sum of the n natural numbers from 1 to n). For example, $sumn(0) == 0$, $sumn(1) == 1$, $sumn(2) == 3$, and $sumn(6) == 21$.

```
//precondition: n >= 0
int sumn(int n) {
  int i = 0, t = 0;
  while (i < n) {
    i = i + 1;
    t = t + i;
  }
  return t;
}
// postcondition: t == n * (n + 1) / 2
```

Verify correctness of `sumn()`. Optionally, convert your code to Dafny syntax to check if it would verify with the loop invariant and the decrementing function that you found.

Use steps below to carry out the proof:

(a) Find the loop invariant and show that it is true for the base case before the loop executes.

(b) Show that it holds for the general case. That is, assume it holds for iteration m and show that it holds for iteration $m + 1$.

(c) Show that at exit, the loop invariant and the exit condition imply the postcondition.

(d) Complete the total correctness proof by showing that loop terminates. Make sure you clearly show your decrementing function D and carry out all steps of the proof.

4.6. Consider the loop in `add()` which adds two polynomials represented by integer arrays of coefficients. `add()` requires `p and q are non-empty arrays, r is an array of length max(p.length, q.length)` and returns $r[i] = (p + q)[i]$ for each $0 \leq i < result.length$.

```
//precondition: p and q are non-empty integer arrays, r is an array of length max(p.length, q.length)
int[] add(int[] p, int[] q) {
  int d = 0, s;
  s = Math.max(p.length, q.length);
  int[] r = new int[s];
  while (d != s) {
    if (d < p.length && d < q.length) {
      r[d] = p[d] + q[d];
    }
    else if (d < p.length) {
      r[d] = p[d];
    }
    else {
      r[d] = q[d];
    }
    d = d + 1;
  }
  return r;
}
// postcondition: r[i] = (p + q)[i] for each 0 <= i < result.length
```

Verify correctness of `add()`. Use steps below to carry out the proof:

(a) Find the loop invariant and show that it is true for the base case before the loop executes.

(b) Show that it holds for the general case. That is, assume it holds for iteration m and show that it holds for iteration $m + 1$.

(c) Show that at exit, the loop invariant and the exit condition imply the postcondition.

(d) Complete the total correctness proof by showing that loop terminates. Make sure you clearly show your decrementing function D and carry out all steps of the proof.