CSE 331 Software Design and Implementation

Section 2: *Reasoning About Code, Part II*

Contents:

Forward Reasoning with Assignments

Forward reasoning with assignments can be deceptive, since performing multiple assignments incorrectly can result in a correct postcondition.

For example, take the following Hoare triple:

```
{ n = 2^x }
n = n * 2;
x = x + 1;
{           }
```

A common mistake is to perform forward assignment in the same manner as backward assignment. This results in the following incorrect proof:

```
// THIS PROOF IS NOT CORRECT!

{ n = 2^x }
n = n * 2;                    // Replace "n" with "2n"
{ 2n = 2^x }=> { n = 2^x-1 }
x = x + 1;                    // Replace "x" with "x+1"
{ n = 2^x - 1 + 1 } ? { n = 2^x }
```

The overall postcondition we obtained is correct. However, the intermediate assertion is incorrect. This can be shown by executing the code with values for x and n.

Assignment using algebraic manipulation

A correct approach is to manipulate the precondition to obtain the second half of the assignment statement (2n, for example), and then replace all instances of that expression with the variable being assigned (n).

```
{n = 2^x } ? { 2n = 2^x+1 }
n = n * 2;
{n = 2^x+1 }    // Replace "2n" with "n"
x = x + 1;
{ n = 2^x }     // Replace "x+1" with "x"
```

More generally, whenever we have an assignment statement "a = b" , any instances of b in the precondition are replaced by a.

Assignment using previous values

Another approach is to define variables *in your assertions* (for example, x_{pre} or n_{post})

which store the value a variable *in the program* had before being reassigned. Here is a proof of our example triple using that method:

```
{ n = 2^x } => { n_pre = 2^x_pre & n = n_pre & x = x_pre }
n = n * 2; // equivalant to n_post = n_pre * 2

{ n_pre = 2^x_pre & n_post = 2*n_pre & x = x_pre }
=> { n_post / 2 = 2^x_pre & x = x_pre}
=> { n_post = 2^x_pre + 1 & x = x_pre }
x = x + 1; // equivalant to x_post = x_pre + 1

{ n_post = 2^x_pre + 1 & x_post = x_pre + 1 }
=> { n_post = 2^x_post }
```

Exercises

Find the strongest postcondition of the following Hoare tripes.

```
{ a = 2b }
b = b + 10;
{         }

{ x/y = 2 & x>0 & y>0}
x = x * x;
y = y * 2;
{           }

{ p^2 + q^2 = r }
r = r / p;
q = q * q / p;
{               }
```

Solutions:

```
{ a = 2b } => { a = 2(b+10) - 20 }
b = b + 10;
{ a = 2b - 20 }

{ x/y = 2 & x>0 & y>0 } => { x^2/y^2 = 4 & x>0 & y>0 }
x = x * x    ;
{ x/y = 4 & x>0 & y>0 } => { 4x/(2y)^2 = 4 & x>0 & y>0 }
y = y * 2;
{4x/y^2 = 4 & x>0 & y>1} => {x/y^2 = 1 & x>0 & y>1 }

{ p^2 + q^2 = r } ? { p + q^2/p = r/p }
r = r / p;
{ p^2 + q^2 = r * p }
q = q * q / p;
{ p + q = r }
```

Loop Invariants

Consider the following Java method:

```
int twos( int x ) {
        n = 1;
        while(x != 0) {
                n = n * 2;
                x = x - 1;
        }
        return n;
}
```

Before we proceed, we should specify the behavior of our method.

- Requires: x >= 0
- Modifies:
- Throws:
- Effects:
- Returns: 2^x

Now that we have our specification, let's prove the correctness of this implementation using Hoare logic. First, we need to insert the overall pre and post conditions, as dictated by our specification:

```
int twos( int x ) {
        { x?0 }
        n = 1;
        while(x != 0) {
                n = n * 2;
                x = x - 1;
        }
        { n = 2^x }
        return n;
}
```

What's wrong with these assertions? Since we change the value of x during our while loop, the assertion { n = 2^x } does not assert what we intend to be our postcondition. How can we express this postcondition correctly?

Again, we need to preserve the initial value of x in an assertion, so that we can refer to it later in our proof. Here are the resulting pre and postconditions:

```
int twos( int x ) {
        { x?0 ? x=x_pre}
        n = 1;
        while(x != 0) {
                n = n * 2;
                x = x - 1;
        }
        { n = 2^x_pre }
        return n;
}
```

We can easily insert another assertion:

```
int twos( int x ) {
        { x?0 ? x=x_pre}
        n = 1;
        { x?0 ? x=x_pre ? n=1}
        while(x != 0) {
                n = n * 2;
                x = x - 1;
        }
        { n = 2^x_pre }
        return n;
}
```

Now, we must prove the correctness of the while loop. In this example we will focus on proving that, assuming the loop terminates, it produces the correct behavior. In order to do so, we must develop a loop invariant. Recall the three conditions needed for a valid invariant:

{P} while b, S {Q}
Let LI be a loop invariant.

a. **P => LI** (The invariant must be true before the loop executes)

b. **{LI & b } S {LI}** (The invariant must be re-established after every execution of the loop)

c. **{LI & ¬b} => Q** (Termination of the loop must establish the desired postcondition)

A suitable invariant for this loop is **$n = 2^{x_{pre} - x}$**.

To show that this invariant is valid, we need to show that it meets the three conditions enumerated previously. This can be done in-line with our code as follows:

```
int twos( int x ) {
       { x>=0 & x=x_pre}
        n = 1;
        { x>=0 & x=x_pre & n=1 } => { x>=0 & n = 2^(x_pre-x) } //Invariant initially true
        while(x != 0) {
                { n = 2^(x_pre-x) & x >= 0 } => { 2n = 2^(x_pre-x + 1) & x & 0 }
                n = n * 2;
                { n = 2^(x_pre-x + 1) & x >= 0 } => { n = 2^(x_pre - (x-1)) & x >= 0 }
                x = x - 1;
                {n = 2^(x_pre-x)}
        }
        {n = 2^(x_pre-x) ? x=0 } ?
        { n = 2^x_pre }
        return n;
}
```

Decrementing Functions

Proving that the loop terminates can be accomplished using a decrementing function. This function must be reduced after ever execution of the loop, and prove that the loop eventually terminates.

For example, take the Euclidian Greatest Common Divisor algorithm:

```
// Returns the greatest common divisor of a and b
int gcd(int a, int b) {
        int x = a;
        int y = b;
        while( y != 0) {
                int r = x % y;
                x = y;
                y = r;
        }

        return x;
}
```

What is an appropriate decrementing function for the while loop contained in this method?

The function **D(y) = y** is a possible decrementing function, since the value of y after the loop executes will always be smaller than the value of y at the beginning of each execution. This function will also guarantee that the loop terminates when D(y) reaches some minimal value (in this case, D(y) = 0).

Other Notes

When using a pictorial invariant, such as the Dutch National Flag example given in lecture 3, make sure to explicitly state how the program variables relate to your picture. For example, if the variable k represents the border between red and white elements, make sure you define whether k represents the last red element, or the first white element.