CSE 331 Lecture Notes
9/26/12 – 9/28/12
Notes by Krysta Yousoufian
Updated for Au12 by Hal Perkins

Assertions

In the first lecture we started with the example of writing a max() function for an array of integers. Suppose you write this code and bring it to your boss. She says, "Prove to me that it works." OK… how do you do that? You could (and surely would) run some tests on sample input, but there's effectively an infinite number of possible inputs. Tests are useful, but they can't prove that your code works in all possible scenarios.

This is where reasoning about code comes in. Instead of running your code, you step back and read it. You ask: "What is guaranteed to be true at this point in the program, based on the statements before it?" Or, going in the in other direction: "If I want to guarantee that some fact Q is true at this point in the program, what must be true earlier to provide that guarantee?" You've surely done some of this naturally. Now you'll learn to do it in a more structured way with techniques to help.

Let's start with a simple code example:

```
x = 17;
y = 42;
z = x+y;
```

At each point before/after/in between statements, what do we know about the state of the program, specifically the values of variables? Since we're looking at this chunk of code in isolation, we don't know anything before it executes. After the first line executes, we know that x = 17. After the second line executes, we still know that x = 17, and we know that y = 42 too. After the third line executes, we also know that z = 17 + 42 = 59. We annotate the code to show this information:

```
{ true }
x = 17;
{ x = 17 }
y = 42;
{ x = 17 ∧ y = 42 }
z = x+y;
{ x = 17 ∧ y = 42 ∧ z = 59 }
```

> *An aside: notation*
> If this notation is unfamiliar to you:
> ∧ means AND
> ∨ means OR
> A way to remember which symbol is which is that the AND symbol looks like the letter A.

Each logical formula shows what must be true at that point in the program. Since we don't know anything at the beginning, only "true" itself must be true, so we simply write {true}.

Each of the lines with curly braces is an assertion. An **assertion** is a logical formula inserted at some point in a program. It is presumed to hold true at that point in the program. There are two special assertions: the precondition and the postcondition. A **precondition** is an assertion inserted prior to execution, and a **postcondition** is an assertion inserted after execution. In the example above, {true} is the precondition and { $x = 17 \land y = 42 \land z = 59$ } is the postcondition. All other assertions are called intermediate assertions. They serve as steps as you reason between precondition and postcondition, kind of like the intermediate steps in a math problem that show how you get from problem to solution.

Forward and backward reasoning

The process we just followed is called **forward reasoning**. We simulated the execution of the program, considering each statement in the order they would actually be executed. The disadvantage of forward reasoning is that the assertions may accumulate a lot of irrelevant facts as you move through the program. You don't know which parts of the assertions will come in handy to prove something later and which parts won't. As a result, you often end up listing everything you know about the program.

This happens in forward reasoning because you don't know where you're trying to go – what you're trying to prove. But when we write a block of code, we usually have a clear idea of what's supposed to be true after it executes. In other words, we know the postcondition already, and we want to prove that the expected postcondition will indeed hold true given the appropriate precondition. For this reason, **backward reasoning** is often more useful than forward reasoning, though perhaps less intuitive.

In backward reasoning, you effectively push the postcondition up through the statements to determine the precondition. You start by writing down the postcondition you want at the end of the block. Then you look at the last statement in the block and ask, "For the postcondition to be true after this statement, what must be true before it?" You write that down, move up to the next statement, and ask again: "For the assertion after this statement to be true, what must be true before it?" You keep going until you've reached the top of the statement list. Whatever must be true before the first statement is the precondition. You have guaranteed that if this precondition is satisfied before the block of code is executed, then the postcondition will be satisfied afterward.

For example, let's look at this two-line block of code:

```
x = y;
x = x + 1;
{ x > 0 }
```

The postcondition is $x > 0$. We want to know what must be true beforehand for the postcondition to be satisfied. We start with the last statement: $x = x + 1$. If $x > 0$ afterward, then the value assigned to x (namely, x+1) must be > 0 beforehand. We add this assertion:

```
x = y;
{ x + 1 > 0 }
x = x + 1;
{ x > 0 }
```

Now we look at the second-to-last statement: $x = y$. If $x + 1 > 0$ after this statement, then [the value assigned to x] + 1 > 0 beforehand. That is, $y + 1 > 0$. We add this assertion:

```
{ y + 1 > 0 }
x = y;
{ x + 1 > 0 }
x = x + 1;
{ x > 0 }
```

Since there are no more statements, we're done. We have proven that if y + 1 > 0 before this block of code executes, then x > 0 afterward.

Weakest precondition

In the example above, y + 1 > 0 is not the only valid precondition. How about the precondition y = 117? Or y > 100? These, too, guarantee the postcondition. Technically they're correct, but intuitively they're not as useful. They are more restrictive about the values of y for which the program is guaranteed to be correct. We usually want to use the precondition that guarantees correctness for the broadest set of inputs. Stated differently, we want the **weakest precondition:** the most general precondition needed to establish the postcondition. The terms "weak" and "strong" refer to how general or specific an assertion is. The weaker an assertion is, the more general it is; the stronger it is, the more specific it is. We write P = wp(S,Q) to indicate that P is the weakest precondition for statement S and postcondition Q.

In our example, y + 1 > 0 is the weakest precondition. The precondition y > 100 is not as weak because it allows only a subset of the values accepted by y + 1 > 0. The precondition y = 117 is the strongest of these three assertions because it allows only a single value that was accepted by either of the other two assertions.

Hoare triples

To formalize all this talk about assertions, we introduce something called a Hoare triple, named for Tony Hoare. (Hoare also invented quicksort and many other cool things.) A **Hoare triple**, written {P} S {Q}, consists of a precondition P, a statement S, and a postcondition Q. In a valid Hoare triple, if S is executed in a state where P is true, then Q is guaranteed to be true afterwards. For example:

{ x != 0 }	*P*
y = x*x;	*S*
{ y > 0 }	*Q*

If S is executed in a state where P is false, Q might be true or it might be false; a valid Hoare triple doesn't have to promise anything either way. On the other hand, a Hoare triple is invalid if there can be a scenario where P is *true*, S is executed, and Q is false afterwards. For example, consider the initial state x = 1, y = -1 for this *invalid* Hoare triple:

> *An aside: {P} S {Q} versus P {S} Q*
>
> In other places (including notes from past quarters of 331) you may see curly braces used for statements instead of assertions. Hoare used both conventions in his original paper to mean slightly different things. The difference is subtle, and for the purposes of this course it's just important to pick one convention and stick with it. We chose to put the curly braces around assertions, and you should do the same in this class.

```
{ x > 0 }        1 > 0; P is satisfied (note that P says nothing about y)
x = y;           x = -1
{ x > 0 }        -1 < 0; Q is not satisfied. Invalid Hoare triple!
```

To give a subtler example of an invalid Hoare triple:

```
{ x >= 0 }       Invalid Hoare triple
y = 2*x;
{ y > x }
```

Suppose x = 0 in the initial state. P is satisfied initially, but afterward y = 0 = x and Q is not satisfied. If we change Q from y > x to y >= x, then the Hoare triple becomes valid.

IF/ELSE statements

So far, we have only looked at sequences of assignment statements executed one after another. We will now consider Hoare triples involving if/else statements:

```
{P} if (B) S1 else S2 {Q}
```

When reasoning about if/else statements, once again it helps to add an intermediate assertion before/after each line of code. We give the complete structure below, followed by an explanation of each new line:

```
{P}
if (B)
        { P Λ B }
        S1;
        {Q1}
else
        { P Λ !B }
        S2;
        {Q2}

{Q1} V {Q2} => {Q}
{Q}
```

What do we know immediately after entering the IF case containing S1? We know that B is true, or we wouldn't have reached this point. We also know that P is true, because we haven't executed any code that could break it. So, we have the assertion P Λ B. What do we know immediately after entering the ELSE case containing S2? Again, P must still be true, and B must be false to have entered the ELSE case. So, we have the assertion P Λ !B.

What about Q1 and Q2, and that funny line with the arrow (=>)? Q1 and Q2 indicate what's known after S1 or S2 is executed, respectively. Because we always enter one case or the other, we can be sure that Q1 or Q2 will be true after executing the entire IF/ELSE statement. So to conclude that Q always holds true, we just need to show that Q is true as long as either Q1 or Q2 is true. Written formally, {Q1} V {Q2}

=> Q. If you've never seen this notation before, it is read as "Q1 or Q2 implies Q." It means that if (Q1 V Q2) is true, then Q is also true. In other words, if Q1 is true then Q is true, and if Q2 is true then Q is true. (If neither Q1 nor Q2 is true, we don't know anything about Q – it could be true or false.)

Notice that {Q1} V {Q2} => {Q} is the second-to-last line in our annotated IF/ELSE block. This indicates that to prove that Q always holds, we need to demonstrate that (Q1 V Q2) => Q.

As an example, let's consider writing code to compute the max of two variables x and y and store it in a variable m. We want to prove that the code works correctly. It should work for all inputs, so we have the trivial precondition {true}. The postcondition is { m = max(x,y) }, or stated more explicitly, { (m=x Λ x >= y) V (m=y Λ y >=x) }. Try writing this code and annotating it with the pattern above to prove that Q always holds.

One possible solution:

```
{true}
if (x > y)
        { true Λ x > y } => { x > y }
        m = x;
        { m = x Λ x > y }
else
        { true Λ x <= y } => { x <= y }
        m = y;
        { m = y Λ x <= y }
{ (m = x Λ x > y) V (m = y Λ x <= y) }
{ m = max(x,y) }
```

In this example, we wrote out Q1 V Q2 and found that it matched up with our earlier definition of Q. Other times, a bit more manipulation might be required to show that (Q1 V Q2) => Q.

Rules for finding the weakest precondition

When we reason about code, we usually want to find the weakest precondition. Even if we're trying to show that our code works in all initial states with no precondition, this can be approached as finding a weakest precondition of {true}. For each type of statement, we need a rule for how to find the weakest precondition.

Assignment statements

We want to find P = wp(x=e, Q). Here, e represents an expression rather than a variable, so it could be replaced with a constant, a variable, a sum of variables … anything that can go on the right-hand side of an assignment statement. As in earlier examples, anything that is true of x after the assignment statement must be true of the value assigned to x beforehand. The weakest precondition P is simply Q with all free occurrences of x replaced by e.

wp(x=e, Q) = Q with all free occurrences of x replaced by e

For example, to find wp(x=y+1, x > 0) we replace x with y+1 in the postcondition x > 0, obtaining the weakest precondition y+1 > 0.

We did a few additional examples in class, starting with the postcondition and statements and filling in the intermediate assertions and weakest precondition:

```
x = x - 2;
z = x + 1;
{ z != 0 }
```

```
x = 2 * y;
z = x + y;
{ z > 0 }
```

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

The solutions are:

```
{ x !- 1 }
x = x - 2;
{ x != -1 }
z = x + 1;
{ z != 0 }
```

```
{ y > 0 }
x = 2 * y;
{ x + y > 0 }
z = x + y;
{ z > 0 }
```

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

Statement lists

We want to find the weakest precondition for two consecutive statements, P = wp(S1; S2, Q). It helps to break down the problem by adding an intermediate assertion between S1 and S2, giving:

{P} S1 {X} S2 {Q}

Then we work backwards. We start by finding the weakest precondition for S2 and use this for X, i.e. X = wp(S2, Q). Next, we use X as the postcondition for S1 and find wp(S1, X). The result will be the weakest precondition for the series of statements S1;S2. Replacing X in wp(S1, X), we get:

wp(S1;S2, Q) = wp(S1, wp(S2,Q))

If/else statements

We want to find the weakest precondition for an if/else statement wp(IF, Q). As before, we write the statement as

if (B) S1 else S2

Suppose B is true. Because S1 is executed and Q must be true afterward, the weakest precondition for the entire IF statement will be the weakest precondition for S1 and Q, i.e. wp(S1, Q). Analogously, if B is false the weakest precondition will be wp(S2, Q). Putting these two cases together, the weakest precondition for the entire if/else statement is wp(S1,Q) when B is true and wp(S2, Q) when B is false. Written formally:

$$\begin{aligned} wp(IF, Q) &= (B \Rightarrow wp(S1,Q) \;\wedge\; !B \Rightarrow wp(S2,Q)) \\ &= (B \wedge wp(S1, Q)) \;\vee\; (!B \;\wedge\; wp(S2, Q)) \end{aligned}$$

An example from lecture was:

```
if (x < 5)
        x = x*x;
else
        x = x+1;
{ x >= 9 }
```

Using the formula above:

$$\begin{aligned} wp(IF, x >= 9) &= (x < 5 \wedge wp(x = x*x, x>=9)) \;\vee\; (x >=5 \;\wedge\; wp(x = x+1, x >=9)) \\ &= (x < 5 \wedge x*x >= 9) \;\vee\; (x >= 5 \wedge x+1 >= 9) \\ &= (x <= -3) \;\vee\; (x >= 3 \wedge x < 5) \;\vee\; (x >= 8) \end{aligned}$$

We also started another example in section. Try finding the weakest precondition of this statement for practice:

```
if (x != 0)
        z = x;
else
        z = x+1;
{ z > 0 }
```

The solution:

$$\begin{aligned} wp(IF, z > 0) &= (x != 0 \;\wedge\; wp(z = x, z > 0)) \;\vee\; (x == 0 \;\wedge\; wp(z = x+1, z > 0)) \\ &= (x != 0 \;\wedge\; x > 0) \;\vee\; (x == 0 \;\wedge\; x+1 > 0) \\ &= (x > 0) \vee (x == 0) \\ &= (x >= 0) \end{aligned}$$