Proving Programs Correct

How can we be sure that a piece of code does what we want it to do? One way is to try testing the code on a large group of data. Another is to formally prove that the program is correct. This handout will cover correctness proofs – how to prove that a program does what we want it to do. We will actually be doing partial correctness proofs – proving that **if** a program does not run forever, then it will produce the correct output. This seems like a rather subtle point, but it is crucial to getting our proofs to work correctly.

Invariants

An invariant is a statement about the state of the program that are true every time a program reaches a certain point of execution. For example, consider the following Pascal statement:

```
x := 4;
```

After these this statement is executed, we know that the value of x must be 4. So, a valid invariant for right after this assignment statement is x = 4. We will write invariants inside braces { }, like so:

```
x := 4;
{ x = 4 }
```

To prove that a piece of code does what we want it to do, we need to first describe what we want the code to do. We will do this in terms of preconditions and postconditions. If we can prove that whenever the preconditions hold, the postconditions must as well, then we have proven that the code does what we want. The way we will do this is to start with the postconditions – what do we want to be true at the end of the code? We will then "back up" the postconditions through the code, using a set of rules that allow us to move invariants over statements. That is, for an invariant to be true after a statement is executed, what needs to be true before the statement is executed?

Rules for backing-up invariants

Assignment Statements :

Consider an assignment statement

x := E

where E is an arbitrary expression (we will assume that expression evaluation has no side-effects). After the assignment statement, we know that x will have the value E

x := E
{ x = E }

We also know that any invariant involving x will be true after the expression is evaluated, if that same invariant, with x replaced with E, will be true before the invariant.

{Q[E/x]} x := E {Q}

Where Q[E/x] means "Q with every occurence of x replaced with E"

This leads to our first rule

Rule to back an invariant over an assignment statement:

To back an invariant Q over an assignment statement x := E, replace all occurrences of x in Q with E

Here are some examples

```
  j := i          ──────►    { i > 0 }
{ j > 0 }                      j := i
                             { j > 0 }

  j := i - 1      ──────►    { i - 1 > 0 }
{ j > 0 }                      j := i-1
                             { j > 0  }

  j := j - 1      ──────►    { j - 1 > 0 }
{ j > 0 }                      j := j - 1
                             { j > 0 }

  z := x * y      ──────►    { x * y > -1 }
{ z > - 1 }                    z := x * y
                             { z > - 1 }
```

We can consider input and output statements to be nothing more than assignment statements. read(x) is the same as x := input. If we have multiple inputs and outputs, then we can use input1, input2, etc.

example:

```
{ (input1 > 0) and (input2 > 0) }
  read(x);
{ (x > 0) and (input2 > 0) }
  read(y);
{ (x > 0) and (y > 0) }
  write(x);
{ (output1 > 0) and (y > 0) }
  write(y);
{ (output1 > 0) and (output2 > 0) }
```

If Statements :

Consider an if statement

if E then Stm1 else Stm2

If the expression E is true before the if statement, then S1 will be executed, and if the expression is false before the if statement, then S2 will be executed. What if we know that:

1. if invariant P and expression E are both true before Stm1, then invariant Q must be true after Stm1,
2. if invariant P is true and E is false before Stm2 is executed, then Q must be true after Stm2 is executed

What can we say about the if statement? Well, if invariant P and expression E are both true before the if statement, then S1 will be executed, and Q will be true after the if statement. If P is true and Q is false before the if statement, then S2 will be executed and Q will be true after the if statement. So, if P is true before the if statement, then Q must be true after the if statement.

{ P and E} S1 {Q} and {P and ~E} S2 {Q} implies {P} if E then S1 else S2 {Q}

This leads to our second rule

Rule to back an invariant over an if statement:

To back an invariant Q over an if statement if E then S1 else S2:
First, back Q up over S1 to get Q1
Then, back Q up over S2 to get Q2
Finally, the backed-up invariant is:
(E and Q1) or (not E and Q2)

Let's see and example:

```
if (x >= y) then
      max := x
else
      max := y
```

```
{(max = x or max = y) and (max >= x and max >= y)}
```

First, we back the invariant over

```
max := x
```

to get

```
(x = x or x = y) and (x >= x and x >= y)
```

which simplifies nicely to

```
( x >= y )
```

Next, we back the invariant over
max := y

to get

```
(y = x or y = y) and (y >= x and y >= y)
```

which simplifies nicely to

```
( y >= x )
```

So, the final backed-up invariant is ((E and Q1) or (not E and Q2))

```
((x >= y) and ( x >= y)) or ((y > x) and (y >= x))
```

Which simplifies to

```
(x >= y) or (y > x)
```

Which simplifies even further to

```
true
```

So, we have

```
{ true }
  if (x >= y) then
      max := x
  else
      max := y

{ (max = x or max = y) and (max >= x and max >= y) }
```

What does it mean when the backed-up invariant is true? It means that there are no restrictions on what needs to be true before the code executes for the postcondtion to be true after the code executes.

Let's try another example with if statements, this time when there is no else:

```
    if (x > y) then begin
          temp := x;
          x := y;
          y := temp
    end
  { x <= y }
```

First, we back up the invariant `(x <= y)` through the body of the if statement, to get `(y <= x)` , as follows:

```
    if (x > y) then begin
        { y <= x }
          temp := x;
        { y <= temp }
          x := y;
        { x <= temp }
          y := temp
        { x <= y }
    end
  { x <= y }
```

Next, we back the statement up through the else clause (which is non-existent), to get `{ y <= x }`. So, the backed up invariant is then (E and Q1) or (not E and Q2) :

```
((x > y) and (y <= x)) or ((x <= y) and (x <= y))
```

which simplifies to

```
((x > y) or (x <= y))
```

which simplifies again to `true`, leaving

```
  { true }
    if (x > y) then begin
        { y <= x }
          temp := x;
        { y <= temp }
          x := y;
        { x <= temp }
          y := temp
        { x <= y }
    end
  { x <= y }
```

While loops :

Consider a while loop

```
while (E) do
      Stm
```

We know that after the loop finishes, E will be false.

```
while (E) do
      Stm
{ not E }
```

While true, this does not really help us prove what we need to prove. We need something a little stronger. What if there was some invariant P, that would be true at the end of the loop if it was true at the beginning. That is, what if we knew that

```
{ P and E }
  Stm
{ P }
```

If the loop is never executed, then if P is true before the loop, it will be true after the loop. Since the loop did not execute, E must be false. So, after the loop P must be true and E must be false

What if the loop is executed once? When the loop body starts, E must be true (else we would not enter the loop). If P and E are true at the start of the loop, then after Stm is executed P will still be true. The loop ends, and P is still true while E must be false (or the loop would not terminate). So P is true and E is false.

What if the loop is executed twice? When the loop body starts, E must be true (else we would not enter the loop). If P and E are true at the start of the loop, then after Stm is executed P will still be true. The loop continues, so E must be true. If P and E are true before Stm is executed the second time, then P must be true after Stm is executed the second time. The loop ends, so E must be false. After the loop, P is true and E is false.

So, if we can prove that P is a *loop invariant* (that is, `{ P and E } <loop body> { P }`), then if P is true before the loop starts, then after the loop finishes P will still be true and E will be false.

({ P and E} Stm {P}) implies {P} while E do Stm { P and not E}

This leads to our third rule

Rule to back an invariant over a while loop:

To back an invariant Q over a while loop while E do Stm:
- First, pick a loop invariant P
- Then, show that P is a loop invariant
  - back up P through the loop body to get P'
  - show that (P and E) implies P'
- Finally, show that (P and (not E)) implies Q
- The backed-up invariant is P

Here are some examples:

```
  while (x > 0) do
        x := x-1;
  end;
{x = 0}
```

First, we need to choose a loop invariant. This is the hardest part of proving code correct – choosing a loop invariant. We'll choose for `P: (x >= 0 )`. Now, we back P up over the loop body to get

```
{ x >= 1 }                    (Since   x-1 >=0   is equivalent to   x >= 1 )
  x := x-1;
{ x >= 0 }
```

Next, we need to show that P and not E imply P'. That is, that

`((x > 0) and (x >= 0))` implies `(x >= 1).`

Simplifying, we get

`(x > 0)` implies `(x >= 1).`

which is true for integers. Finally, we need to show that P and (not E) implies Q

`((x >= 0) and (x <= 0))` implies `(x = 0).`

Simplifying, we get

`(x = 0)` implies `(x = 0).`

which is true.

So, the proof in its entirety:

```
  while (x > 0) do              Loop invariant: { x >= 0 }
      { x >= 1 }                      (x > 0)  and (x >= 0) implies (x >= 1)
        x := x-1;                     (x >= 0) and (x <= 0) implies (x = 0)
      { x >= 0 }
  end;
{ x = 0 }
```

What happens if x is negative before the loop starts? Well, the program will run forever. Is that a problem? It means that the proof is only valid if computation terminates. So these proofs are partial correctness proofs, not complete correctness proofs. They only say that if the program terminates, it gives the correct answer. They say nothing about whether or not the program will run forever. In Automata Theory you will learn that in general it is not possible to prove that a program does not run forever.

Let's look at another proof.

```
  while x =! 0 do begin
        x := x - 2;
        y := y - 2
  end;
{ x > y and x = 0 }
```

Here we go:

```
{ x > y }
  while x =! 0 do begin       Loop Invariant: (x > y)
      { x - 2 > y - 2 }             (x > y) and (x != 0) implies (x > y)
        x := x - 2;                 (x > y) and (x = 0) implies (x > y) and (x = 0)
      { x > y - 2 }
        y := y - 2
      { x > y }
  end;
{x > y and x = 0}
```

Let's now look at an extended example

We want to prove the following:

```
{ input1 > 0 and input2 > 0 }
  begin
        read(x);
        read(y);
        div := 0;
        while (x >= y) do begin
              div := div + 1;
              x := x - y
        end;
        write(div);
        write(x);
  end;
{ (input1 = output1*input2+output2)  and (0 =< output2 < input2) }
```

Notice how the program computes integer division and mod, and the postcondition invariant describes how output = input1 div input2 and output2 = input1 mod input2

First, we back the invariant over a few assignment statements:

```
{ input1 > 0 and input2 > 0 }
  begin
        read(x);
        read(y);
        div := 0;
        while (x >= y) do begin
              div := div + 1;
              x := x - y
        end;
      { (input1 = div * input2 + x) and (0 =< x < input2) }
        write(div);
      { (input1 = output1 * input2 + x) and (0 =< x < input2) }
        write(x);
  end;
{ (input1 = output1 * input2 + output2)  and (0 =< output2 < input2) }
```

Now we come to the loop. We need to pick a loop invariant P, prove that P is a loop invariant, and then show that `P and (not (x >= y))` implies `{(input1 = div * input2 + x) and (0 =< x < input2)}`

Now we need a loop invariant. One choice is

`L1: { input1 = div * y + x }`

For this to be acceptable, it must pass two tests:

L1 must actually be a loop invariant, that is (L1 and `x >= y`) implies `L1`

`L1 and (not (x >=y))` must imply
`{ (input1 = div * input2 + x) and (0 =< x < input2) }`

First, let's show that L1 is a loop invariant, by backing it up through the body of the while loop:

```
{ input1 = (div + 1) * y + (x - y) }
  div := div + 1;
{ input1 = div * y + (x - y) }
  x := x - y
{ input1 = div * y + x }
```

Since `({ input1 = div * y + x } and { (x >= y) } implies`
`{ input1 = (div + 1) * y + (x - y) }`

L1 must be a loop invariant. Next test:

Does L1 and not E imply `{ (input1 = div * input2 + x) and (0 =< x < input2) } ?`

```
{ input1 = div * y + x } and { (x < y) } does not imply
      { (input1 = div * input2 + x) and (0 =< x < input2) }
```

So L1, while a loop invariant, it doesn't help us. Let's try something else :

L2: `(input1 = div * input2 + x) and (0 =< x < input2)`

First, we'll try to show that L2 is in fact a loop invariant, by backing it up through the body of the while loop

```
{ (input1 = (div + 1) * input2 + x - y) and (0 =< x - y < input2) }
  div := div + 1;
{ (input1 = div * input2 + x - y) and (0 =< x - y < input2) }
  x := x - y
{ (input1 = div * input2 + x) and (0 =< x < input2) }
```

Unfortunately, we now see that L2 is **not** in fact a loop invariant, since

```
(input1 = div * input2 + x) and (0 =< x < input2) and (x >= y)
      does not imply
(input1 = (div + 1) * input2 + x - y) and (0 =< x - y < input2)
```

So, L1 was an invariant, but was not sufficient to show Q, and L2 was sufficient to show Q but was not a loop invariant. So we'll try one more time with

```
L3: input1 = div * y + x and  x >= 0 and y = input2
```

```
{ input1 = (div + 1) * y + x - y and  x - y >= 0 and y = input2 }
  div := div + 1;
{ input1 = div * y + x - y and  x - y >= 0 and y = input2 }
  x := x - y
{ input1 = div * y + x and  x >= 0 and y = input2 }
```

We now test to see if (L3 and E) implies L3'

```
(input1 = div * y + x and  x >= 0 and y = input2 and x >= y) implies
     (input1 = (div + 1) * y + x - y and  x - y >= 0 and y = input2)
```

Simplifying a bit:

```
(input1 = div * y + x and  x >= 0 and y = input2 and x >= y) implies
      (input1 = (div  * y + x  and  x  >= y and y = input2
```

which is clearly correct. Finally, we need to test if L3 and (not E) implies Q:

```
input1 = div * y + x and  x >= 0 and y = input2 and x < y implies
    (input1 = div * input2 + x) and (0 =< x < input2)
```

It does, so we are done. To complete the proof, we need to back the loop invariant up past the rest of the code:

```
      { input1 = input1 and  input1 >= 0 and input2 = input2 }
        read(x);
      { input1 = x and  x >= 0 and input2 = input2 }
        read(y);
      { input1 = x and  x >= 0 and y = input2 }
        div := 0;
      { input1 = div * y + x and  x >= 0 and y = input2 }
```

And voila, we are done. Let's take a look at the complete proof, in all its glory:

```
      { input1 >= 0 }
        read(x);
      { input1 = x and  x >= 0 }
        read(y);
      { input1 = x and  x >= 0 and y = input2 }
        div := 0;
      { input1 = div * y + x and  x >= 0 and y = input2 }

      { Loop Invariant:  input1 = div * y + x and  x >= 0 and y = input2
            (input1 = div * y + x and  x >= 0 and y = input2) and (x >= y) implies
                   (input1 = div * y + x  and  x >= y and y = input2)

            (input1 = div * y + x and  x >= 0 and y = input2) and (x < y) implies
                  (input1 = div * input2 + x) and (0 =< x < input2)                }

        while (x >= y) do begin
            { input1 = (div + 1) * y + x - y and  x - y >= 0 and y = input2 }
              div := div + 1;
            { input1 = div * y + x - y and  x - y >= 0 and y = input2 }
              x := x - y
            { input1 = div * y + x and  x >= 0 and y = input2 }
        end;
      {  (input1 = div * input2 + x) and (0 =< x < input2) }
        write(div);
      {  (input1 = output1 * input2 + x) and (0 =< x < input2) }
        write(x);
  end;
{  (input1 = output1 * input2 + output2)  and (0 =< output2 < input2) }
```