Logic

Hardware versus software

In hardware such as chip design or architecture, designs are usually proven to be correct using proof tools

In software, a program is very rarely proved correct

Why?

Scope

Chip and architecture designs are all quite similar – it is a fairly small domain, so experience can be carried over, and automatic or semi-automatic tools are effective

Software covers a huge range of topics, so the background theory needed to prove a program correct might involve geometry (for 3-D programs), physics (for 'realistic' programs), graph theory (for network or web or social media programs), or any other application theory

So for software, proof has to be at least as creative as programming, and we can't usually afford the effort

Impossibility

Another reason why proof is not used more often is that it is sometimes impossible or nearly impossible

One reason is that a proof would involve solving a problem that we don't (yet) know how to solve (e.g. Collatz program, see later)

Another is that a program often has no independent specification against which to prove it correct - it is its own specification (e.g. face recognition)

Why study software logic?

To program well, you need a good mental model of what is going on in a program

Looking at logic helps you to build or refine that mental model, in other words the better your grasp of logic, the better your intuition will be

It is vital to recognize when the underlying logic is complex, and therefore intuition is likely to go astray

Earlier, I said it is important to learn to trace the execution of a program, this is just a more precise version of tracing

What logic to study?

We are going to look at Hoare logic, from a semi-inituitive point of view rather than in full mathematical detail

The aim is not to be able to contruct serious mathematical proofs, but to learn to become aware of the underlying logic when programming, to program more accurately

At a minimum, you will learn what ***precondition***, ***postcondition***, ***invariant*** and ***variant*** mean, so you can understand when people talk about them

Hoare triples

We will be looking at statements like this:

```
{ x == 41 } x = x + 1; { x == 42 }
```

The first part in curly brackets is a statement called the ***precondition***, and the last part is the ***postcondition***

In between is a fragment of a program

C notation


```
{ x == 41 } x = x + 1; { x == 42 }
```

The middle part has to be written in C if we are studying C

I'm ***choosing*** to use C notation for the conditions on either side, so as to use plain text and to make the statements readable to C programmers as boolean expressions

But they are mathematical expressions, so (a) no side-effects and (b) they can say things which C can't express

Assertions

To use logic in C itself, you can write:

```
#include <assert.h>
...
assert(x == 41);
x = x + 1;
assert(x == 42);
```

Assertions are conditions that are checked when your program runs - their purpose is ***either*** to do auto-testing ***or*** to guard against bugs

You can switch them off with `NDEBUG`

Meaning


```
{ x == 41 } x = x + 1; { x == 42 }
```

A triple is a mathematical statement which is true or false, hopefully true

It means: ***if the precondition is true just before the code is executed, then the postcondition is true just after the code has been executed***

You may or may not have to prove separately that execution reaches the end of the code

Language features

For each language feature in C, there is a way of checking that a statement about it is true

We will have a brief look at assignments, sequences, ifs, loops and functions

The assignment rule


```
{ x == 41 } x = x + 1; { x == 42 }
```

Let's say an assignment consists of a variable (`x`) being updated by a right hand side (RHS) expression (`x + 1`)

Take the postcondition, replace the variable by the right hand side expression, and then check that the result is implied by the precondition

The abstract assignment rule

In more abstract mathematical notation:

```
{ P[x\E] } x = E; { P }
```

You can read this something like "if precondition 'P with x replaced by E' is true, and code 'x becomes E' is executed, then postcondition P is true"

The meta-variables stand for arbitrary conditions, C variables, or C expressions (according to context)

Note `P[x\E]` is sometimes written `P[E/x]`

Note there is an extra rule to say pre ***implies*** post

Which version?

In my opinion, the informal version in words is not enough on its own, because it doesn't give precise details

The formal version is not enough on its own, because it doesn't say what to do

So you need both, plus examples

Example 1


```
{ x == 41 } x = x + 1; { x == 42 }
```

Take the postcondition:

```
x == 42
```

Replace the variable (`x`) by the RHS expression (`x+1`):

```
x + 1 == 42
```

Is this implied by `x == 41`?

Yes it is, so the statement is true

Example 2


```
{ x == 41 && y == 3 } y = x + 1; { x == 41 && y == 42 }
```

Take the postcondition:

```
x == 41 && y == 42
```

Replace the variable (`y`) by the RHS expression (`x+1`):

```
x == 41 && x + 1 == 42
```

Is this implied by `x == 41 && y = 3`?

Yes it is, so the statement is true

The sequencing rule


```
{ true } x = 1; y = 2; { x == 1 && y == 2 }
```

(A precondition of `true` means no restriction)

Let's say a sequence consists of a first statement followed by a second statement

Find an intermediate condition that can act as a postcondition for the first statement, and a precondition for the second statement

The abstract sequencing rule

$$\frac{\{ P \}\ S1\ \{ Q \},\ \{ Q \}\ S2\ \{ R \}}{\{ P \}\ S1; S2\ \{ R \}}$$

You can read this "if you know
`{ P } S1 { Q }` and
`{ Q } S2 { R }`, then you can deduce
`{ P } S1; S2 { R }`"

The semicolon is a 'mathematical' sequencing operator which combines two statements to form a compound statement (treating C's semicolon as a separator)

Example 3


```
{ true } x = 1; y = 2; { x == 1 && y == 2 }
```

To check that the statement is true, find a condition to put in the middle, in this case obviously `x == 1` will do

```
{ true } x = 1; { x == 1 }
{ x == 1 } y = 2; { x == 1 && y == 2 }
```

Check whether these are both true using the assignment rule - they are, so the statement is true

The if rule


```
{ true } if (x<0) y = -x; else y = x; { y >= 0 }
```

Let's say an if statement has a test, a then statement, and an else statement

Add the test to the precondition and check the then statement, and also add the negation of the test to the precondition and check the else statement

The abstract if rule

$$\frac{\{\ P\ \&\&\ B\ \}\ S1\ \{\ Q\ \},\quad \{\ P\ \&\&\ !B\ \}\ S2\ \{\ Q\ \}}{\{\ P\ \}\ \texttt{if (B) S1 else S2}\ \{\ Q\ \}}$$

This captures mathematically the way the if statement works, breaking it down into two cases

Example 4


```
{ true } if (x < 0) y = -x; else y = x; { y >= 0 }
```

Add the test to the precondition and check the then statement

```
{ true && x < 0 } y = -x; { y >= 0 }
```

Add the negation of the test to the precondition and check the else statement

```
{ true && x >= 0 } y = x; { y >= 0 }
```

These are both true, so the original statement is true (though the postcondition could be stronger)

If without else

What do you do with an if statement that has no else part?

You use the fact that the default for the else part is "do nothing"

Mathematically, you need a symbol which means "do nothing" (often `skip`) and a skip rule (the precondition must directly imply the postcondition)

But we are going to gloss over this, and other details such as types and scope

The while rule


```
{ n > 0 } while (n > 1) n = n / 2; { n == 1 }
```

Here's where things get interesting - we assume the very simplest kind of while loop for now

Let's say a while loop has a test and a body (statement)

Find an ***invariant*** and check that it is implied by the precondition, preserved by the loop body, and with the negation of the test implies the postcondition

Also, find a ***variant***, an integer which starts and stays non-negative, and is reduced by the loop body

Preserving the invariant

The invariant for a while loop needs to be a strong statement, which is true every time round the loop and after the loop

Preserving the invariant means checking that if the invariant is true, and the test is true, so that the loop body is executed, then the invariant is still true afterwards

Then the invariant will be true after the loop has finished (by induction on the number of times the loop is executed), and the test will be false, and you can check the postcondition

Termination


```
{ n > 0 } while (n > 1) n = n / 2; { n == 1 }
```

How do you check termination?

Find a ***variant***, that is (e.g.) an integer which starts off `>=0`, which stays `>=0`, but which is reduced by at least one each time round the loop

There are other possible variants, but they can always be converted to a non-negative decreasing integer if you try hard

The abstract while rule


```
{ I && B } S { I }
I && ! B    =>    Q
{ I && B && V >= 0 } V0 = V; S { V >= 0 && V < V0 }
-------------------------------------------------------
{ I && V >= 0 } while (B) S { Q }
```

`I` is the invariant condition, `V` is the variant expression, and `V0` remembers the original value of `V` (with `V0` a variable not mentioned in `S`)

(In C, `p => q` can be written `!p || q`)

Example 5a


```
{ n > 0 } while (n > 1) n = n / 2; { n == 1 }
```

Let's choose `n >= 1` as the invariant, it is clearly implied by the precondition, and we can check:

```
{ n >= 1 && n > 1 } n = n / 2; { n >= 1 }
```

That is true, and now let's check that the invariant and the negation of the test imply the postcondition:

```
n >= 1 && n <= 1   =>   n == 1
```

That is also true, which just leaves termination

Example 5b


```
{ n > 0 } while (n > 1) n = n / 2; { n == 1 }
```

As our variant, we can choose n

```
{ n > 0 && n > 1 && n >= 0 } v = n; n = n / 2; { n >= 0 && n < v }
```

Here v is an added variable giving us access to the old version of the variant

This amounts to saying if `n >= 2` then `n / 2` is less than n which is true

What about continue?

What if a loop has a continue statement in it?

Answer: transform the code to make it disappear

```
while(...) {
    S1;
    if (b) continue;
    S2;
}
```

```
while(...) {
    S1;
    if (! b) {
        S2;
    }
}
```

What about break?

If a loop has `break` in it, transform the code:

```
while(...) {
    S1;
    if (b) break;
    S2;
}
```

```
bool ended = false;
while(! ended && ...) {
    S1;
    if (b) ended = true;
    else S2;
}
```

What about for loops?

For a for loop, transform the code into a while loop:

```
for (i=0; i<n; i++) {
    S;
}
```

```
i = 0;
while(i < n) {
    S;
    i++;
}
```

Why transform?

Some reasons for these transformations are:

- to have a 'simple' core logic that can be studied carefully
- to avoid rules which become more and more complex
- to apply the logic to algorithms rather than programs

Language features which 'jump around' are extremely difficult to capture in the logic and are, arguably, also intuitively complex, so should be used 'sparingly'

Example 6


```
{ i==0 && s==0 } while (i<n) { s=s+n; i++; }; { s == n*n }
```

Let's assume termination, and guess that the invariant is `s == i*n` and check it is implied by the precondition

```
i==0 && s==0    =>    s == i*n;
```

The invariant is preserved if:

```
{ s == i*n && i<n } s=s+n; i++; { s == i*n }
```

These two statements together are the same as an inductive proof of the invariant (show true for `i=0`, and show if true for `i` then true for `i+1`)

Proof

Actual proof is fraught with danger:

- it is normal in maths to use intuition, 'wave your hands', and let intelligent readers to fill in the gaps
- if you focus on the process, it is easy to get the details wrong
- if you focus on details, you tend to work 'mechanically' and make mistakes in the process

So I recommend testing out proof ideas by writing code with `assert` statements

Example 7: Collatz


```
long n = ...;      start with any positive integer
while (n > 1) {
    if (even(n)) n = n / 2;    if (n%2==0)
    else n = 3*n+1;
}
```

For numbers up to 2^{60}, the loop terminates, but beyond that it is still unknown (see Wikipedia)

There are other conjectures that have been shown false only for very large numbers, so we ***really*** don't know

So proof of termination is sometimes impossible or impractical

What about functions?

A function body is a single block of code, so the normal thing to do is:

First transform the function to get rid of any early returns, perhaps by introducing a result variable

Then introduce a precondition at the start of the function and a postcondition at the end

The precondition is any explicit or implicit restriction on the arguments

E.g. "`n` is the number of..." implies `n>=0`

Example 8: binary search

Even if you are not going to do a formal proof, thinking about preconditions, postconditions, invariants and variants gives you a powerful intuitive grasp which helps get code right

Let's look again at the binary search algorithm

Reminder

Here's a reminder of the code we wrote:

```
int search(char ch, int n, char a[n]) {
    int start = 0, end = n, mid;
    bool found = false;
    while (! found  && end > start) {
        mid = start + (end - start) / 2;
        if (ch == a[mid]) found = true;
        else if (ch < a[mid]) end = mid;
        else start = mid + 1;
    }
    return found ? mid : -1;
}
```

Precondition

What's the precondition?

Informally, it is "the input is a sorted array"

Slightly more formally:

```
{ i<j => a[i] <= a[j] } k = search(ch, n, a); { ... }
```

The precondition lacks "for all indexes `i` and `j` of `a`", which we would have to express in C as a double loop

Writing this down reminds us that some items in `a` may be equal, so the result is not uniquely defined, unless we say "first occurrence"

Postcondition

What's the postcondition? Something like:

```
{ ... } k = search(ch, n, a); { k == -1 || a[k] == ch; }
```

The postcondition isn't complete because it doesn't say that if the result is `-1` then `ch` isn't in the array

And it doesn't specify that the search should find the first occurrence (which our code doesn't guarantee!)

And it doesn't say that the array and `ch` are `const`, i.e. not to be altered

Invariant

What's the invariant for the loop? Something like:

```
a[start] <= ch  &&
a[end] >= ch
```

When we narrow the range inside the loop, we need to make sure that if `ch` is in the array, at least one occurrence is inside the range

If you want to guarantee the first occurrence, search for the first occurrence of `ch` instead of just for `ch`, and strengthen the invariant

Variant

What's the variant?

Given the convention we set up for `start` and `end`, we can just use `V = end - start`

We need to be sure that this always reduces, to make sure that our code can't get stuck in an infinite loop

We can be sure that `mid >= start` and `mid < end` (given that division rounds down), and the worst cases are `mid == start` in which case `start` is increased by one, and `mid = end-1` in which case `end` is decreased by one