Correctness proofs

James Aspnes

January 9, 2003

A *correctness proof* is a formal mathematical argument that an algorithm meets its specification, which means that it always produces the correct output for any permitted input. Detailed correctness proofs of even moderately complex algorithms can be surprisingly long, so algorithms researchers (and writers of textbooks like [CLRS01]) often write informal arguments giving only an outline of the full proof. The relationship between the informal argument and the underlying proof is analogous to the relationship between an informal definition of an algorithm and a program that implements it: the informal argument is only justified by being backed by an implicit formal proof that is sadly too long, unenlightening, or tiresome to present. So it is important to understand what a detailed formal correctness proof looks like, because otherwise you won't know what somebody (possibly including you!) is really saying with an informal correctness argument.

1 Invariants, preconditions, and postconditions

When viewed from a high enough level of abstraction, most correctness proofs look something like this: define a predicate P that is true for "correct" states of the algorithm. Then prove:

1. P holds in the initial state.
2. P holds after step k if it holds before step k.
3. If P holds when the algorithm terminates, then the output of the algorithm is correct.

Such a predicate P is called an *invariant* of the algorithm. It is called an invariant for the simple reason that it is always true; this follows by induction from the first two statements above. It is a useful invariant because of the third statement.

Many algorithms are big, and most programs even bigger, so finding a single invariant that applies to an entire program usually makes about as much sense as trying to write the entire program in one line of code. We can break up the problem of proving a program or algorithm correct by approaching it one statement at a time. One technique for doing this is known as *Hoare logic* [Hoa69], a simplified version of which is described below.

2 Hoare logic for straight-line programs

The essential idea of Hoare logic is that we attach to each statement of a program a precondition (something that we demand is true before the statement executes) and a postcondition (something that will then be true after the statement executes). A precondition, statement, and postcondition together form a *Hoare triple*. Preconditions and postconditions are typically written inside curly braces to distinguish them from program code (Hoare logic predates the C programming language).[1] For example, here is a simple Hoare triple:

```
{ x is even }
x := x+1
{ x is odd }
```

To show that the postcondition follows from the precondition, write x for the old value of `x` and x' for the new value, and observe that $x = 0 \pmod 2$ implies $x' = x + 1 = 1 \pmod 2$.

This Hoare triple only makes sense in a larger context. There are many programs in which a variable is incremented, but most of the time we don't guarantee that the variable is even beforehand and don't care if the variable is odd afterwards. But suppose that we have some earlier statement whose postcondition guarantees that `x` is in fact even:

```
{ x is an integer }
x := 2*x
```

[1] Technical note: Hoare put the program statement in braces instead of the assertion, like this: $x = 0\{x := x + 1\}x = 1$. Some writers use this form for *partially correct* triples, where the postcondition holds only if the statement terminates, reserving the notation $\{x = 0\}\ x := x + 1\ \{x = 1\}$ for *totally correct* triples, where the statement is guaranteed to terminate (see Section 6 for more on this distinction). We will be doing partial correctness, but we are still going to put the assertions in the braces braces because it makes it easier to annotate normal-looking program code.

```
{ x is even }
```

Now we can string these statements together to produce a program for generating odd numbers:

```
{ P: x is an integer }
x := 2*x
{ Q: x is even }
x := x+1
{ R: x is odd }
```

Here we have labelled the conditions P, Q, and R. Since P implies Q and Q implies R, we have that R holds at the end of the program provided P holds at the beginning. We could thus abstract out the actual program statements and produce yet another Hoare triple:

```
{ x is an integer }
run the above program
{ x is odd }
```

People who do this for a living have a formal way of writing rules like the one we just used, which looks like this:

$$\frac{\{P\}\ S_1\ \{Q\} \wedge \{Q\}\ S_2\ \{R\}}{\{P\}\ S_1; S_2\ \{R\}}.$$

This is a fancy way of say that if we have already proven that $\{P\}\ S_1\ \{Q\}$ and $\{Q\}\ S_2\ \{R\}$ (the stuff on top) are Hoare triples, then we can assert that $\{P\}\ S_1; S_2\ \{R\}$ (the stuff on the bottom) is also a Hoare triple, where the semicolon denotes sequential execution. Rules like this, which define what new propositions can be deduced from old ones, are called *axioms*. This particular axiom is called the *composition axiom*.

Here are some other basic axioms of Hoare logic, expressed formally:

- The *pre-strengthening axiom*,[2] which says that making the precondition stronger doesn't change the truth of a Hoare triple:

$$\frac{\{Q\}\ S\ \{R\} \wedge P \Rightarrow Q}{\{P\}\ S\ \{R\}}.$$

[2]This axiom and the closely related post-weakening axiom were called *rules of consequence* in Hoare's original paper.

Real-world example: from

$$\{\text{studied} \geq 1 \text{ hour}\} \ \text{take test} \ \{\text{test-grade=A+}\},$$

we can use pre-strengthening to derive

$$\{\text{studied} \geq 1 \text{ month}\} \ \text{take test} \ \{\text{test-grade=A+}\}.$$

Pre-strengthening is mostly used to sneak in extra facts that don't appear explicitly in our original precondition, since it's a standard theorem in logic that whenever Q is true, $P \Rightarrow P \wedge Q$ is also true.

- The *post-weakening axiom*, which says that making the postcondition weaker is also allowed:

$$\frac{\{P\} \ S \ \{Q\} \wedge Q \Rightarrow R}{\{P\} \ S \ \{R\}}.$$

Real-world example: starting with

$$\{\text{studied} \geq 1 \text{ month}\} \ \text{take test} \ \{\text{test-grade=A+} \wedge \text{other-grades=F}\}.$$

we get

$$\{\text{studied} \geq 1 \text{ month}\} \ \text{take test} \ \{\text{other-grades=F}\}.$$

Post-weakening is typically used for getting rid of bits of a postcondition we don't care about.

Important caveat: The direction of the implications is important. Pre-weakening and post-strengthening do *not* produce valid proofs.

- The *assignment axiom*:

$$\{P[x/t]\} \ x := t \ \{P\}.$$

In other words, if a predicate P is true with x replaced by t before the assignment, it is true without the replacement afterwards. Here $P[x/t]$ is just a concise way of writing "P with x replaced by t." Examples:

$$\{0 = 0\} \ x := 0 \ \{x = 0\}.$$

$$\{x + 5 < 12\} \ x := x + 5 \ \{x < 12\}.$$

In this second example we might want to apply pre-strengthening to rewrite $x + 5 < 12$ as the more natural $x < 7$.

- Finally, the *baggage lemma*:

$$\frac{\{P_1\}\ S\ \{Q_1\} \wedge \{P_2\}\ S\ \{Q_2\}}{\{P \wedge P_2\}\ S\ \{Q_1 \wedge Q_2\}}.$$

  The baggage lemma is used to carry along extra baggage that you will need later. It is a lemma rather than an axiom because it follows from the other axioms.[3] For example, suppose you can prove that some very complicated statement S sets x to the square root of z (written formally, $\{\}\ S\ \{x = \sqrt{z}\}$), but you also need to know that after the statement executes, some variable y that is not assigned to in S is unchanged from its previous value (0, say). Use the assignment, if/then/else, iteration, and composition axioms as needed to prove $\{y = 0\}\ S\ \{y = 0\}$, and then use baggage to get $\{y = 0\}\ S\ \{y = 0 \wedge x = \sqrt{z}\}$.

Most of the time these axioms will not be applied explicitly, but it is the responsibility of the prover to make sure that anything they do claim can be justified using them.

3 Strategy for proving correctness

Using Hoare logic, our general strategy for proving correctness has three steps:

1. Write down the algorithm.
2. Annotate the algorithm by putting the precondition for the algorithm at the top, the postcondition at the bottom, and an assertion in between each pair of statements.
3. Prove for each statement that its postcondition follows from its precondition, using the axioms of Hoare logic and any other theorems we need to drag in from mathematics in general (via prestrengthing and post-weakening).

Since the axioms of Hoare logic are pretty simple, the last step is usually tricky only if we chose the wrong preconditions in the middle. Here the axioms of Hoare logic can help us. For example, suppose we want to show

$$\{P\}\ x := x + 1\ \{x \bmod 2 = 1\},$$

[3] The baggage lemma also did not appear in Hoare's original paper, but it is too useful to leave out.

but we don't know what to put in for P. Here we can apply the assignment axiom backwards to find the *weakest precondition* that makes this a Hoare triple:

$$\{(x+1) \bmod 2 = 1\}\ \ x := x+1\ \ \{x \bmod 2 = 1\},$$

which is implied by

$$\{x \bmod 2 = 0\}\ \ x := x+1\ \ \{x \bmod 2 = 1\},$$

using pre-strengthening and the fact that $(x+1) \bmod 2 = 1$ is equivalent to $x \bmod 2 = 0$, which we can show by adding 1 to both sides.

Using weakest preconditions is sometimes a good way to find bugs in algorithms. If you work out that you can only have the algorithm be correct if its input is even, but it's supposed to work for all inputs, then you either need to strengthen your algorithm's precondition (in this case, change the problem so you throw up your hands in despair given an odd input), or fix your algorithm.

4 Proofs for if/then/else statements

Special axioms apply for compound statements. Suppose we want to show that the following is a Hoare triple, where P and Q are arbitrary predicates and B is some test:

```
    { P }
    if B then
1       do something
    else
2       do something else
    end if
    { Q }
```

Here we need to fill in preconditions an postconditions for Lines 1 and 2. The result will look something like this:

```
    { P }
    if B then
        { P and B }
1       do something
        { Q }
    else
```

```
        { P and not B }
2       do something else
        { Q }
    end if
    { Q }
```

And now we just need to prove that Line 1 makes Q true provided $P \wedge B$ is true and that Line 2 makes Q true provided $P \wedge \neg B$ is true. For example:

```
{ x = a }
if x < 0 then
    { x = a and x < 0 (which implies a < 0) }
    x := -x
    { x = -a = |a| }
else
    { x = a and x >= 0 (which implies a >= 0) }
    do nothing
    { x = a = |a| }
end if
{ x = |a| }
```

The formal axiom we are using (in addition to a couple of implicit applications of pre-strengthening and post-weakening), is the *if/then/else axiom*:

$$\frac{\{P \wedge B\}\ S_1\ \{Q\} \wedge \{P \wedge \neg B\}\ S_2\ \{Q\}}{\{P\}\ \textbf{if}\ B\ \textbf{then}\ S_1\ \textbf{else}\ S_2\ \textbf{end if}\ \{Q\}}.$$

5 Proofs for loops

Now consider the problem of proving correctness for a while loop:

```
{ P }
while B do
    body
end while
{ Q }
```

How do we prove that Q holds at the end of the loop when P holds at the beginning? It is tempting to insist that the body must make Q true, but maybe it takes many passes through the body before Q is true. The solution is to use an invariant R that holds at the beginning and end of each pass through the body, and which implies Q when the loop terminates.

Here is a simple example of a program that zeroes out an array of n elements, together with the precondition and postcondition we would like to use for it.

```
{ A is an array with indices 0..n-1 }
i := n
while i <> 0 do
    i := i - 1
    A[i] := 0
end while
{ A[0]..A[n-1] are all equal to zero }
```

How do we prove the postcondition given the precondition? We need an invariant R that is (1) true at the start of the loop (i.e., it follows from the initial precondition); (2) true after each pass through the body of the loop if it is true at the beginning (i.e., it is both a precondition and postcondition of the body); and (3) implies the postcondition for the program. Here is an attempt at R: we'll insist that $A[j] = 0$ for all indicies $j \geq i$. This invariant holds initially because there are no indices $j \geq i$. If it holds at the end of the loop, then the postcondition is true because i is now 0 (otherwise the loop would not terminate). To show that it is preserved by the body, we annotate the program with additional preconditions and postconditions:

```
    { A is an array with indices 0..n-1 }
    i := n
1   while i <> 0 do
        { A[j] = 0 for all j >= i }
2       i := i - 1
        { A[j] = 0 for all j >= i+1 }
3       A[i] := 0
        { A[j] = 0 for all j >= i }
    end while
    { A[0]..A[n-1] are all equal to zero }
```

And now we just have to prove for each line in the body that its precondition implies its postcondition. The proof of correctness for Lines 2 and 3 is left as an exercise to the reader (hint: use the assignment axiom and then clean up with pre-strengthening/post-weakening).

Something to think about: what happens if we try to prove the correctness of the following version of the program?

```
{ A is an array with indices 0..n-1 }
i := n
while i >= 0 do
    A[i] := 0
    i := i - 1
end while
{ A[0]..A[n-1] are all equal to zero }
```

The rule for while loops is that if R holds at the beginning of the loop, and R is preserved by the body when the test B is true, then $R \wedge \neg B$ holds when the loop terminates. Written formally, we have the *iteration axiom*:

$$\frac{\{R \wedge B\}\ S\ \{R\}}{\{R\}\ \textbf{while}\ B\ \textbf{do}\ S\ \textbf{end while}\ \{R \wedge \neg B\}}.$$

Often we won't bother with remembering that B is true in the body of the loop, and will just show that $\{R\}\ S\ \{R\}$ holds. If this happens, we can put B back in to get the strict form of the iteration axiom by applying pre-strengthening.

Of course, we haven't necessarily shown that the loop actually terminates. This leads us to a useful distinction, between *total correctness* proofs and *partial correctness* proofs.

6 Total vs partial correctness

In addition to proving that an algorithm produces the right output, we will also want to show that it produces this output in a reasonable amount of time, typically bounded by some function of the size of the input. This latter task is a major part of algorithm analysis, and we will be spending a lot of time on it over the rest of the semester. But since we will be showing running time bounds later, we can often save time in our correctness proof by proving only *partial correctness*, which says that the algorithm produces the correct answer whenever it terminates, but allows for the possibility that the algorithm does not terminate at all. We will call an algorithm that produces the right answer provided it terminates *partially correct*, and call it *totally correct* if it is partially correct and terminates on all inputs.

Partial correctness is most useful for algorithms involving **while** loops, because we can assume that a loop's termination condition is true when the loop finishes, even if we don't understand what is happenning inside the loop. For example, here is a partially correct algorithm that computes the value 1 given any positive integer as input:

```
procedure Collatz1(x: integer) returns integer:
    while x <> 1:
        if x is even:
            x := x/2
        else:
            x := 3*x+1
        end if
    end while
    return x
end procedure
```

Why do we know this is partially correct? Because the loop won't finish until `x` equals 1. It is widely believed that the loop body will eventually set `x` to 1 [Lag85], but at the time of this writing few mathematicians expect to see a proof of this conjecture soon.

Sometimes partial correctness is not so useful. Here is a "universal" partially correct algorithm, which we might call the Exam-Taker's Algorithm:

```
procedure ExamTaker:
    while true:
        hope for inspiration
    end while
end procedure
```

Since the algorithm never terminates, it meets the criterion for partial correctness no matter what answer it is supposed to produce. If you have ever found yourself applying this algorithm, you will understand why this is a problem.

Generally, we will only worry about proving the partial correctness of an algorithm, because we will be considering its running time as a separate step in the analysis. Provided we can put a finite bound on the running time, total correctness is immediate (because only terminating algorithms take finite time).

7 Proofs for recursive procedures

Partial correctness also comes into play in analyzing recursive procedures. Here is a procedure that implements Euclid's algorithm for computing the greatest common divisor (GCD) of two positive integers:

```
procedure Euclid(x, y: integer) returns integer
```

```
    if y = 0 then
        gcd = x
    else
        gcd = Euclid(y, x mod y)
    end if
    return gcd
end procedure
```

How do we prove that this procedure works? Let's assume as an induction hypothesis that the recursive call works, i.e. that `Euclid(y, x mod y)` really does compute the gcd of x and x mod y.

We can then fill in preconditions throughout:

```
    procedure Euclid(x, y: integer) returns integer
        { }
1       if y = 0 then
            { y = 0 }
2           gcd := x
            { gcd = gcd(x, y) }
        else
            { y <> 0 }
3           gcd := Euclid(y, x mod y)
            { gcd = gcd(x, y) }
        end if
        { gcd = gcd(x, y) }
        return gcd
    end procedure
    { return value = gcd(x, y) }
```

Now to prove partial correctness, we just walk through the lines of the program one at a time and show that each each forms a valid Hoare triple with its precondition and postcondition.

- For Line 1, apply the if/then/else axiom to the triples for lines 2 and 3.
- For Line 2, use the assignment axiom to show

$$\{y = 0 \wedge x = x\} \text{ Line 2 } \{\text{gcd} = x \wedge y = 0\},$$

  and then use post-weakening with $\text{gcd} = x \wedge y = 0 \Rightarrow \text{gcd} = \text{gcd}(x, y)$ to simplify the postcondition (since $\text{gcd}(x, y) = \text{gcd}(x, 0) = x$), and

pre-strengthening to get rid of the extraneous $x = x$ in the precondition (since $y = 0 \Rightarrow y = 0 \wedge x = x$).

- For Line 3, we have to do a little number theory. We want to show that when $y \neq 0$, $\gcd(x, y) = \gcd(y, x \bmod y)$. Observe that if k divides both x and y, then $y = km$ for some m, $x = kn$ for some n, and there is an integer a such that $x \bmod y = x - ay = kn - kam = k(n - am)$, implying k divides both y and $(x \bmod y)$. On the other hand, if k divides both y and $(x \bmod y)$, then $x = ay + (x \bmod y) = kam + kb = k(am + b)$ and k divides x. So the common divisors of x and y are equal to the common divisors of y and $(x \bmod y)$, and in particular the greatest common divisors of the two pairs are the same. So $\text{Euclid}(y, x \bmod y) = \gcd(y, x \bmod y) = \gcd(x, y)$ is true given our induction hypothesis.

  Now let's do some Hoare logic. Start with this result of the assignment axiom plus post-weakening (to get rid of the extra $y = 0$ term):

  $$\{y \neq 0 \wedge \text{Euclid}(y, x \bmod y) = \gcd(x, y)\} \text{ Line 3 } \{\text{gcd} = \gcd(x, y)\}.$$

  Our exercise in number theory showed that $P \Rightarrow Q$, where P is $y = 0$ and Q is $\text{Euclid}(y, x \bmod y) = \gcd(x, y)$. A standard tautology says that in this case $P \Rightarrow P \wedge Q$, and so we can drop Q from the precondition using pre-strengthening:

  $$\{y \neq 0\} \text{ Line 3 } \{\text{gcd} = \gcd(x, y)\},$$

  which is what we set out to prove.

We have now completed the proof of partial correctness for the Euclid algorithm.

Note that we still haven't shown that the algorithm terminates. Indeed, if we reverse the order of y and $x \bmod y$ in the arguments to the recursive call, the algorithm will still be partially correct (by the same argument), but it will no longer be totally correct. Proving termination will be left for later, when we look at this algorithm's runtime.

8 A practical example

Let's put together what we've learned. Suppose we want to prove the correctness of the following binary search procedure:

```
    { A[i] < A[j] when i < j; A[i] = t for some 0 <= i < n }
1   l := 0
2   h := n-1
3   while l < h do
4       m = (l + h) / 2        # C-style integer division
5       if A[m] < t then
6           l = m + 1
        else
7           h = m
        end if
    end while
    { A[h] = t }
```

Working backwards, we see that we need an invariant R for the while loop that implies $A[h] = t$ when $l = h$. How do we get this invariant? Well, our intuition for binary search is that we are keeping trace of a lower bound l and an upper bound h on the position of t. So we'd like to say something like $A[l] \leq t \leq A[h]$. Let's try plugging this in:

```
    { A[i] < A[j] when i < j; A[i] = t for some 0 <= i < n }
1   l := 0
2   h := n-1
    { A[l] <= t <= A[h] }
3   while l < h do
        { A[l] <= t <= A[h] }
4       m = (l + h) / 2        # C-style integer division
5       if A[m] < t then
6           l = m + 1
        else
7           h = m
        end if
        { A[l] <= t <= A[h] }
    end while
    { A[h] = t }
```

Now we'll use the if/then/else axiom and the assignment axiom to fill in some preconditions.

```
    { A[i] < A[j] when i < j; A[i] = t for some 0 <= i < n }
1   l := 0
```

```
2   h := n-1
    { A[l] <= t <= A[h] }
3   while l < h do
        { A[l] <= t <= A[h] }
4       m = (l + h) / 2        # C-style integer division
        { ??? }
5       if A[m] < t then
            { A[m] < t; A[m+1] <= t <= A[h] }
6           l = m + 1
            { A[l] <= t <= A[h] }
        else
            { A[m] < t; A[l] <= t <= A[m] }
7           h = m
            { A[l] <= t <= A[h] }
        end if
        { A[l] <= t <= A[h] }
    end while
    { A[h] = t }
```

Now we have a big blank space where we need to find a precondition for Line 5 that implies both preconditions for Lines 6 and 7. We know from the precondition on Line 4 that t lies between $A[l]$ and $A[h]$; so perhaps it is enough to show that m lies between l and h. So let's make the precondition on Line 5 read:

```
        { A[l] <= t <= h; l <= m < h }
```

and then use the baggage lemma to carry the precondition at the top throughout:

```
    { P: A[i] < A[j] when i < j; A[i] = t for some 0 <= i < n }
1   l := 0
    { P; A[l] <= t }
2   h := n-1
    { P; A[l] <= t <= A[h] }
3   while l < h do
        { P; A[l] <= t <= A[h]; l < h }
4       m := (l + h) / 2        # C-style integer division
        { P; A[l] <= t <= A[h]; l <= m < h }
5       if A[m] < t then
            { P; A[l] <= t <= A[h]; l <= m < h; A[m] < t }
```

```
6           l := m + 1
            { P; A[l] <= t <= A[h] }
        else
            { P; A[l] <= t <= A[h]; l <= m < h; A[m] >= t }
7           h := m
            { P; A[l] <= t <= A[h] }
        end if
        { P; A[l] <= t <= A[h] }
    end while
    { A[h] = t }
```

Here I've replaced the preconditions we used to have on Lines 6 and 7 with ones that follow from the precondition on Line 5. We'll have to show that these imply the preconditions we really wanted (which means we can use post-strengthening to show that Lines 6 and 7 give Hoare triples). But we've now finished step 2 of our general proof technique: annotating the program with true propositions in between each statement. What remains is step 3—proving that each statement is the middle of a Hoare triple.

Proof for Line 1: Let's start by applying the assignment axiom:

$$\{P \wedge A[0] \leq t\}\ \ l := 0\ \ \{P \wedge A[l] \leq t\}\,. \tag{1}$$

From the precondition we have $A[i] = t$ for some i with $0 \leq i$. Since $0 \leq i \Rightarrow A[0] \leq A[i] = t$, we have $P \Rightarrow P \wedge A[0] \leq A[i]$. So now pre-strengthening says we can replace the ugly precondition from (1) with the stronger condition $\{P\}$.

Proof for Line 2: Essentially the same as for Line 1. Use the assignment axiom to get

$$\{P \wedge A[l] \leq A[t] \leq A[n-1]\}\ \ h := n-1\ \ \{P \wedge A[l] \leq A[t] \leq A[h]\}\,,$$

and then apply pre-strengthening as before to get rid of the extra inequality in the precondition.

Proof for Line 3: The iteration axiom gives

$$\{P \wedge A[l] \leq t \leq A[h]\}\ \text{Line 3}\ \{P \wedge A[l] \leq t \leq A[h] \wedge l \geq h\}\,.$$

But now $l \geq h$ implies $A[l] \geq A[h]$ and we have $A[l] \leq t \leq A[h] \leq A[l]$. This only works if in fact $A[l] = t = A[h]$. Now use post-weakening to get rid of everything except $A[h] = t$.

Proof for Line 4: The assignment axiom gives us:

$$\begin{gathered}\left\{P \wedge A[l] \leq t \leq A[h] \wedge l < h \wedge l \leq \left\lfloor \tfrac{l+h}{2} \right\rfloor < h\right\} \\ m := (l+h)/2 \\ \left\{P \wedge A[l] \leq t \leq A[h] \wedge l < h \wedge l \leq \left\lfloor \tfrac{l+h}{2} \right\rfloor < h\right\}\end{gathered} .$$

Observe that $l < h$ implies $l \leq \frac{l+l}{2} \leq \left\lfloor \frac{l+h}{2} \right\rfloor < \frac{h+h}{2} = h$. Now use pre-strengthening to remove this term from the precondition and post-weakening to remove $l < h$ from the postcondition.

Proof for Line 5: Immediate application of the if/then/else axiom.

Proof for Line 6: Use the assignment axiom to get:

$$\{P \wedge A[m+1] \leq t \leq A[h]\}\ \ l := m+1\ \ \{P \wedge A[l] \leq t \leq A[h]\} \qquad (2)$$

Now apply pre-strengthening after observing that the precondition to Line 6 implies the precondition in (2): since $A[m] < t$, there exists $i > m$ for which $A[i] = t$. But then $i \geq m+1$ and so $A[m+1] \leq A[i] = t$.

Proof for Line 7: This is simpler than the proof of Line 6 because we don't need P. Start with the assignment axiom:

$$\{P \wedge A[l] \leq t \leq A[m]\}\ \ h := m\ \ \{P \wedge A[l] \leq t \leq A[h]\}\,,$$

and apply pre-strengthening to add in $l \leq m < h$ and $t \leq A[h]$ and get the full precondition.

This completes the proof of partial correctness for the binary search algorithm. As before, we'll defer proving termination to the the running time analysis.

8.1 A note on the invariant

We were fortunate that we picked a loop invariant that was (a) true, and (b) strong enough to prove itself. The usual way that proofs like this break down is when the invariant turns out to be false (e.g., consider $\{A[l] \leq t < A[h]\}$, or is too weak (we came close to this by not putting $l \leq h$ into the invariant, but in this case P saves us). False invariants sometimes manifest themselves as unproveable postconditions (for example, we can't prove $t < A[h]$ after Line 2 without having $t < A[n-1]$ before, which we don't know to be true),

but often both false and true invariants break down inside the loop where the precondition on the loop body has to be stronger than the invariant plus the loop test. Whittling an invariant into shape often requires a fair bit of creativity and understanding of the algorithm. If you can't find a good invariant, you may also want to look closely at the algorithm to see if it really works.

8.2 Informal proof of correctness

Much of the subconscious education of mathematicians involves learning when one can get away with writing a step of a proof informally, with the understanding that the reader will be able to fill in the details if necessary. The proof we gave above for the binary search algorithm is much more detailed than you would usually see. A more typical proof might look something like this (footnotes are added to show what Hoare logic axioms to appeal to when the reader complains we are cheating):

Partial correctness proof: Observe that the precondition P is not affected by any line of the algorithm; so we will assume that it holds throughout.[4] Let the loop invariant consist of P plus the proposition $A[l] \leq t \leq A[h]$. This invariant clearly holds at the start of the loop since $l = 0$, $h = n - 1$, and $A[0] \leq t \leq A[n-1]$ from P.[5] To see that it holds at the end of the loop, note that either $A[m] < t$, in which case t is at position $m + 1$ or higher and $A[l'] = A[m + 1] \leq t$, or $t \leq A[m]$, in which case $t \leq A[m] = A[h']$.[6] Finally, since $h \leq l$ when the loop exits, we have $A[h] \leq A[l] \leq t \leq A[h]$, where the first inequality follows from P and the remaining inequalities are the invariant, implying $A[h] = t$ as claimed.[7]

Here's an even shorter proof:

[4] Assignment axiom—none of the assignments modify any of the variables in P.

[5] Here we omitted the (relatively trivial) proofs for Lines 1 and 2 completely, figuring that the reader remembers how assignment works well enough to believe us. The word "clearly" in this statement fulfills the same function as the crooked tailors' assertion that only intelligent people could see the Emperor's New Clothes—it warns the rabble that they will be embarrassed if they complain about leaving the details out and the details turn out to be as trivial as we claim. But beware of using clearly for things that aren't clear, or, even worse, that may be false–lest you act out the part of the naked emperor yourself.

[6] This is compressing down the proofs of lines 5-7 by appealing to our intuition about where t lives. Note the implicit appeal to the if/then/else axiom where we split into two cases.

[7] While axiom.

Sketch of partial correctness proof: Use $A[l] \leq t \leq A[h]$ as a loop invariant.

What style of proof should you write? You should be able to write a detailed Hoare logic proof like the one we did first, where you carefully annotate the program and separately prove that the postcondition on each line follows from its precondition—but you generally don't have to unless you are specifically asked to. Most of the correctness proofs you write will look like the medium-compact proof (only without the footnotes). But a confused reader has the right to expect that you could expand your compact proof into a detailed proof if necessary.

You will see sketches like the last "proof" in research papers where the authors were limited for space and had a lot of confidence in their reader's ability to regenerate the entire proof from a single key hint.[8] Some lecturers are also fond of putting up sketchy proof fragments like this and sneaking away. You should probably not use this trick yourself until you've been publishing successfully for a while. The TA who grades your assignments will most likely *not* assume that you are just leaving out the boring parts of a proof you understand.

References

[CLRS01] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. McGraw-Hill, 2001.

[Hoa69] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12:576–580, 1969.

[Lag85] Jeffrey C. Lagarias. The $3x + 1$ problem and its generalizations. *American Mathematical Monthly*, 92(1):3–23, January 1985.

[8] Such confidence is not always misplaced— the average research paper is only ever read by two people, and both of them are likely to be specialists (one of them is the author).