Guidance on Proofs of Correctness

Based on some feedback from the examples class I have written some general guidance on how to approach questions asking you to write proofs of correctness. I have structured this guidance around questions I received in the examples class.

How do I start?

I will say some more general things before explicitly answering this question, so please bear with me.

Firstly, it is important to note that there is no general algorithm as the correctness problem is undecidable. This is a concept we will meet in the last part of the course and tells us that there is (provably) no algorithm that can always produce the right answer to this question. It is important that you are familiar with this idea that there may not be an algorithm to follow. Many real-world problem solving tasks are of this form.

However, all is not lost as many parts are algorithmic and usually a proof just requires a few small creative steps to set things up and then mechanical application of the rules.

There are only two rules that require a creative step:

- The rule for while requires a loop invariant for partial correctness and additionally a loop variant for total correctness. The rule doesn’t tell us what this should be but it gives hints of how it should fit into the proof. I discuss a method for finding suitable invariants and variants below.

- The rule of consequence doesn’t tell you when to apply it. This rule is the glue so in general you should only use it when you need to stick bits of your proof together.

Everything else can be mechanical, but what does this mean? With the exception of the rule of consequence, the inference rules tell us what the structure of the proof should look like. If we need to prove something of the form

\[
\{ \ P \ \} \ if \ b \ then \ S_1 \ else \ S_2 \ \{ \ Q \ \}
\]

then the related rule tells us that we need to prove two further partial correctness statements:

\[
\{ \ P \wedge b \ \} \ S_1 \ \{ \ Q \ \} \quad \{ \ P \wedge \neg b \ \} \ S_2 \ \{ \ Q \ \}
\]

even the while rule defines the structure of what we need to prove, it just has something else we need to fill in.

So, where do we start? Let us run through two examples where I describe the steps in detail.
Example One

Let us consider the partial specification statement

\[
\{ \ x > 0 \ \} \ \text{if} \ y > 0 \ \text{then} \ z := x + y \ \text{else} \ z := x - y \ \{ \ z > 0 \ \}
\]

how do we start producing a proof for this statement? We can immediately apply the \textit{if} \textit{p} rule to decompose this proof into two subproofs. How do we know this? Because of the structure of the program, the top-level construct is the \textbf{if then else} and this is the construct targeted by the \textit{if} \textit{p} rule. If we are constructing a linear proof we can now write

1. \{ \ x > 0 \land y > 0 \ \} \ z := x + y \ \{ \ z > 0 \ \} by ?
2. \{ \ x > 0 \land \neg(y > 0) \ \} \ z := x - y \ \{ \ z > 0 \ \} by ?
3. \{ \ x > 0 \ \} \ \text{if} \ y > 0 \ \text{then} \ z := x + y \ \text{else} \ z := x - y \ \{ \ z > 0 \ \} by \textit{if} \textit{p}(1, 2)

but now we need to produce proofs for statements 1 and 2. In a sense we can start again. Given the partial correctness statement

\[
\{ \ x > 0 \land y > 0 \ \} \ z := x + y \ \{ \ z > 0 \ \}
\]

how should we prove this? The top-level construct is assignment so we should use the assignment rule. But the assignment rule places restrictions on the precondition and postcondition. We can try applying it backwards from the postcondition \( z > 0 \) i.e. the assignment rule tells us that this partial correctness statement

\[
\{ \ x + y > 0 \ \} \ z := x + y \ \{ \ z > 0 \ \}
\]

is universally true. But the precondition we have is \( x > 0 \land y > 0 \). So where to go from here? The rule of consequence tells us that if \( (x > 0 \land y > 0) \rightarrow (x + y > 0) \) then we can glue these together. Thankfully this implication does hold. These two steps may not appear very intuitive. Instead of doing this we could introduce the following \textit{derived rule} that follows directly from the given rules:

\[
\begin{align*}
P & \rightarrow Q[x \mapsto A[a]] \quad \{ \ Q[x \mapsto A[a]] \ \} \ x := a \ \{ \ Q \ \} \\
\{ \ P \ \} x := a & \ \{ \ Q \ \}
\end{align*}
\]

If you want you can use this rule in your proofs and call it \textit{ass}_p + \textit{cons}_p. We can now update our linear proof to be

1. \{ \ x + y > 0 \ \} \ z := x + y \ \{ \ z > 0 \ \} by \textit{ass}_p
2. \( (x > 0 \land y > 0) \rightarrow (x + y > 0) \) by maths
3. \{ \ x > 0 \land y > 0 \ \} \ z := x + y \ \{ \ z > 0 \ \} by \textit{cons}_p(1, 2)
4. \{ \ x > 0 \land \neg(y > 0) \ \} \ z := x - y \ \{ \ z > 0 \ \} by ?
5. \{ \ x > 0 \ \} \ \text{if} \ y > 0 \ \text{then} \ z := x + y \ \text{else} \ z := x - y \ \{ \ z > 0 \ \} by \textit{if} \textit{p}(3, 4)

I will leave filling in the remaining ? as an exercise. Note that it is completely symmetric to what we have just done but with a slightly less trivial implication. I discuss how to deal with this implications further below, in case they are troubling you.
It is important to reflect on what we just did. We iteratively did the following things

- Applied the structural rules to decompose the proof into smaller proofs
- If the partial correctness statement is an assignment, apply the derived $\text{ass}_p + \text{cons}_p$ rule

This doesn’t cover the while rule but is a method for algorithmically producing proofs of correctness for programs without loops. This is important as proofs of correctness for programs with loops will eventually turn into proofs of correctness for programs without loops as the rules are applied.

Note that this isn’t the only way we could prove this program correct. We could have applied the rule of consequence and assignment rule more creatively. For example, we could have produced this (partial) proof.

1. $\{ x > 0 \land y > 0 \land x + y = x + y \} \Rightarrow z := x + y \{ x > 0 \land y > 0 \land z = x + y \}$ by $\text{ass}_p$
2. $(z = x + y \land x > 0 \land y > 0) \Rightarrow (z > 0)$ by maths
3. $\{ x > 0 \land y > 0 \} \Rightarrow z := x + y \{ z > 0 \}$ by $\text{cons}_p(1,2)$
4. $\{ x > 0 \land \neg(y > 0) \} \Rightarrow z := x - y \{ z > 0 \}$ by $\text{?}$
5. $\{ x > 0 \} \text{ if } y > 0 \text{ then } z := x + y \text{ else } z := x - y \{ z > 0 \}$ by $\text{if}_p(3,4)$

Here we notice that the precondition $x > 0 \land y > 0$ and the equality $z = x + y$ are strong enough together to imply the required postcondition $z > 0$ so we can apply the following derived rule

$$
\begin{align*}
\frac{P \land a = a \quad x := a \quad P \land x = a \quad (P \land x = A[a]) \Rightarrow Q}{P \quad x := a \quad Q}
\end{align*}
$$

This is relying on the other direction of the rule of consequence. Previously we were weakening the precondition but now we are strengthening the postcondition. Note that this is sound as everything follows from the original set of rules but it is not in itself complete in the sense that it relies on $P$ not containing $x$ for it to be directly applied. If you are not familiar with the notions of sound and complete then look them up online.

You might wonder where these derived rules are coming from. This is just me writing down the intuition captured by multiple applications of the existing rules. You don’t need derived rules but sometimes they can help give shortcuts in proofs.

**Example Two**

Now let us consider an example containing a loop. We will use a very simple program and think about the steps we need to go through. Let us prove the following partial correctness statement:

$$
\{ x = a \land y = b \land x \geq 0 \} \text{ while } x > 0 \text{ do } (y := y + 1; x := x - 1) \{ y = a + b \}
$$

The program is destructive, in the sense that it destroys the initial values of $x$ and $y$, so we need to use auxiliary variables in the specification.

We cannot proceed as before as the while rule doesn’t fit with the pre and postconditions we have. We need a loop invariant. I discuss more tricks for how to find a loop invariant below. For now let us look at the loop body

$$
y := y + 1; x := x - 1
$$
and think about an \textit{expression} that will have the same value at the \textit{start} and \textit{end} of this loop body (\textbf{not during}). This should be straightforward; the value of $x+y$ goes up by 1 and down by 1 during the loop body so its value stays the same. But this isn’t a predicate. To make it a predicate we can consider the value of $x+y$ at the \textit{start} of the loop (before it executes). This is $a+b$ so we will try

$$x+y = a+b$$

as a loop invariant. I phrase it like this (\textit{try}) as this loop invariant might not be strong enough for what we need.

This tells us that we will be trying to produce a proof of

$$\{ x+y = a+b \} \textbf{while } x > 0 \textbf{ do } (y := y + 1; x := x - 1) \{ x+y = a+b \land -(x > 0) \}$$

but before we do that we should check that proving this allows us to prove the thing we wanted to prove in the first place. To do this we should connect this statement to our original statement via the rule of consequence, which requires us to establish

1. $(x = a \land y = b \land x \geq 0) \rightarrow (x+y = a+b)$
2. $(x+y = a+b \land -(x > 0)) \rightarrow y = a+b$

The first implication holds trivially but the second does not; we are missing something. What we really want is to show that at the end of the loop $x = 0$. We know this is true as $x \geq 0$ at the beginning but we cannot use this information here as we haven’t established that this is preserved by the loop. This tell us that we need to strengthen the loop invariant by adding $x \geq 0$. So our loop invariant is now

$$R \triangleq (x+y = a+b \land x \geq 0)$$

Our application of the rule of consequence now holds as

$$(x+y = a+b \land x \geq 0 \land (x > 0)) \rightarrow (x+y = a+b \land x = 0) \rightarrow y = a+b$$

At this point our linear proof looks like

1. $\{ R \} \textbf{while } x > 0 \textbf{ do } (y := y + 1; x := x - 1) \{ R \land -(x > 0) \}$ by ?
2. $(x = a \land y = b \land x \geq 0) \rightarrow (x+y = a+b)$ by maths
3. $(x+y = a+b \land x \geq 0 \land -(x > 0)) \rightarrow (x+y = a+b \land x = 0) \rightarrow y = a+b$ by maths
4. $\{ x = a \land y = b \land x \geq 0 \} \textbf{while } x > 0 \textbf{ do } (y := y + 1; x := x - 1) \{ y = a+b \}$ by cons$_p(1,2,3)$

where I have already named the loop invariant to make things look neater.

We can now apply the \textbf{while} rule to produce the following partial correctness statement without loops that we have to prove

$$\{ R \land x > 0 \} \textbf{ y := y + 1; x := x - 1 } \{ R \}$$

When faced with a sequence of assignments I tend to apply the assignment rule in sequence backwards. Here that would look like

$$\{x-1+y+1 = a+b \land x-1 \geq 0 \}\ y := y+1 \ {x-1+y = a+b \land x-1 \geq 0}\ x := x-1 \ {x+y = a+b \land x \geq 0}\$$

and now we can apply the rule of consequence to check that

$$(x+y = a+b \land x \geq 0 \land x > 0) \rightarrow (x-1+y+1 = a+b \land x-1 \geq 0)$$

which holds as $x > 0 \rightarrow x-1 \geq 0$ (this is the important bit, the rest is trivial).
This might have gone quite fast and it might help to break this down. Notice that this corresponds to an extension of above derived rule ass\(_p + \text{cons}\_p\) to \(n\) assignments. What I just did corresponds to the linear proof:

1. \(\{ x - 1 + y + 1 = a + b \land x - 1 \geq 0 \} \ y := y + 1 \ \{ x - 1 + y = a + b \land x - 1 \geq 0 \} \) by ass\(_p\)
2. \(\{ x - 1 + y = a + b \land x - 1 \geq 0 \} \ x := x - 1 \ \{ x + y = a + b \land x \geq 0 \} \) by ass\(_p\)
3. \(\{ x - 1 + y + 1 = a + b \land x - 1 \geq 0 \} \ y := y + 1; x := x - 1 \ \{ R \} \) by comp\(_p(1,2)\)
4. \((x + y = a + b \land x \geq 0 \land x > 0) \rightarrow (x - 1 + y + 1 = a + b \land x - 1 \geq 0)\) by maths
5. \(\{ R \land x > 0 \} \ y := y + 1; x := x - 1 \ \{ R \} \) by comp\(_p(3,4)\)
6. \(\{ R \} \ \text{while } x > 0 \text{ do } (y := y + 1; x := x - 1) \ \{ R \land \neg(x > 0) \} \) by while\(_p(5)\)

and this fills in the ? in the above proof. Notice that I could have organised this slightly differently as I could have applied the rule of consequence in different places. I find this the neater way of organising things but the proof system allows you to be creative. However, it is very important that you make non-trivial applications of the rule of consequence explicit. An example of a trivial application is one that applies very basic arithmetic reasoning to an expression \(a = b\) so that it is in the form \(a = a\) and then removes the equivalence. Another example is of the form \(x > 0 \land x < 0\) being resolved to \(false\).

**A summary of how to start**

Let me finish this section with a rough guide to how to approach constructing a proof of correctness for the partial correctness specification

\[
\{ P \} S \{ Q \}
\]

For total correctness one just needs to add the loop variant in the appropriate place. Let us assume the program is of the form

\[
S_1; (\text{while } b \text{ do } S_2); S_3
\]

Now follow these steps:

1. If there is a loop find a loop invariant \(R\); if it is a total correctness specification find a loop variant also. See below for hints on how to find these

2. Deal with the bits of the program before and after the loop i.e. produce proofs for \(\{ P \} S_1 \{ R \} \) and \(\{ R \land \neg b \} S_3 \{ Q \} \). If \(S_1\) and \(S_3\) are empty then you just need to show that \(P \rightarrow R\) and \((R \land \neg b) \rightarrow Q\). If you cannot produce this proofs (or the implications do not hold) then you might need to strengthen the loop invariant.

3. Now produce a proof for the loop body i.e. \(\{ R \land b \} S_2 \{ R \} \) (or with a loop invariant if it is total correctness). To do this follow the above steps of

   (a) applying the structural rules until you get an assignment (or sequence of assignments)

   (b) applying the assignments backwards from the postcondition to produce some new precondition

   (c) applying the rule of consequence to connect the new precondition to the existing one
What are we proving?

This question is asking what does it mean if we have produce a proof of a correctness statement i.e. if we have a proof such as

\[
\begin{align*}
\{ x > 0 \land x > 0 \} & \vdash y := x \{ y > 0 \} & \{ x > 0 \land \neg(x > 0) \} \triangleright \{ y > 0 \} \\
\{ x > 0 \} & \text{if } x > 0 \text{ then } y := x \text{ else skip } \{ y > 0 \}
\end{align*}
\]

what have we achieved?

There are two approaches to this question:

- If the inference system is sound then we have shown that the statement at the bottom is true. What does sound mean? It means that whenever we have a rule of the form \(\text{premise} \rightarrow \text{conclusion}\) if the premise is true then the conclusion is necessarily true. Soundness means that we cannot prove things that are false. For rules without a premise this means that they are axioms i.e. always true. Note that our rules have metavariables in them, like \(P\), \(Q\) and \(S\), this means that they should hold for any instantiation of those metavariables.

- If the statement at the bottom is true what does that mean? This returns to the meaning of the triple \(\{ P \} S \{ Q \}\) which is that given any state \(s\) if \(P(s)\) is true then when executing \(S\) on \(s\) (recall the operational semantics) to produce \(s'\) (recall that it is deterministic and terminating) we have \(Q(s)\) being true. More mathematically we might say

\[
\forall s, s' \in \text{State} : (P(s) \land \langle S, s \rangle \Rightarrow s') \rightarrow Q(s')
\]

Perhaps the meaning of \(P(s)\) and \(Q(s')\) is unclear. When we write

\[
\{ x > 0 \} \text{if } x > 0 \text{ then } y := x \text{ else skip } \{ y > 0 \}
\]

we are implicitly defining the precondition predicate

\[
P(s) = s(x) > 0
\]

and the postcondition predicate

\[
Q(s) = s(y) > 0
\]

so our notation is a bit lazy in this respect. A special case of this is when we write \(\{ \text{true} \} S \{ Q \}\) or \(\{ \} S \{ Q \}\) where true stands for the predicate

\[
\text{true}(s) = \text{true}
\]

i.e. the predicate that always returns true and ignores the state, and we take the empty predicate to be true as it places no restrictions on the states.

What direction does it go in?

This is a very similar question to how do I start as the question is really asking whether we start with the goal or axioms. Sometimes in mathematical proof there seems to be a moral obligation to start from axioms and derive the thing you want to derive. Perhaps more pragmatically some inference systems work like that, for example the modus ponens rule

\[
\begin{align*}
P \rightarrow Q & \quad P \\
\hline & \quad Q
\end{align*}
\]

starts from knowing \(P \rightarrow Q\) and \(P\) and deriving \(Q\). It is designed to go from top to bottom as from the bottom it is not clear what should go on the top.
However, many proof systems, this one included, are *goal-directed* which means that we start from the thing we want to prove and decompose this statement into other things we want to prove. So to answer the original question, we go ‘up’.

Although this is not the full story. As suggested earlier, I tend to

- Go ‘up’ if the program is an if statement or a sequence
- Go ‘up and backwards’ if the program is an assignment of sequence of assignments i.e. use the derived rule introduced earlier to go up on the right hand side, backwards through the assignment rule and then join up with the rule of consequence
- Start form the ‘middle’ if the program is a while i.e. find the loop invariant and produce an upwards proof from that and a downwards proof to join it up with the initial pre and post conditions

**Does a loop invariant or a loop variant always exist?**

No, for the simple reason that the program might not be correct and might not be terminating. As a more complicated answer, the proof system is *not complete*. Which means that there are correctness statements we can write down but cannot prove in the system. A simple example is

\[
\{ \text{true} \} \ S \ \{ \text{false} \}
\]

in general as this collapses to solving the Halting Problem. As another argument consider

\[
\{ \text{true} \} \ x := x \ \{ P \}
\]

for some predicate \(P\), this collapses to determining whether \(P\) is valid. We haven’t discussed the specification language we use for predicates but generally we have used this is first-order logic with arithmetic. We know that first-order logic is partially decidable and arithmetic is undecidable. So for any reasonably useful specification language for predicates the proof system is clearly incomplete.

**How do I find loop invariants?**

The first thing to point out is that there is no general algorithm that given a triple

\[
\{ P \} \text{ while } b \text{ do } S \ \{ Q \}
\]

always find a loop invariant \(R\) such that

- \(P \rightarrow R\)
- \((R \land \lnot b) \rightarrow Q\)
- \(\{ R \land b \} \ S \ \{ R \}\)

which is what we need for \(R\) to be a usable loop invariant. A lot of research has gone into finding good ways of solving this problem but due to the general undecidability of arithmetic it is not even in general possible to check if the above implications hold for a given \(R\).
So now I have told you there is no algorithm what should you do? This is one of the creative parts of this course but there are some hints you can follow:

- The loop invariant should contain all of the variables that the loop body updates and are referred to either after the loop body or in the postcondition. If it doesn’t then it is almost certain that the loop invariant will not be strong enough.

- It is common to have a loop condition of the form \(x > 0\) and to assume at the end of the loop that \(x = 0\). For this to hold you need the loop invariant to include \(x \geq 0\) (or similar). Just because this holds before the loop doesn’t mean you can assume it holds afterwards, anything you want to be true after the loop has to go in the loop invariant.

- Sometimes you might have definitions that are invariant across the whole program. These still need to go in the loop invariant but you can ignore them during the proof.

- Look at the postcondition, what do you need to be true at the end. If the postcondition is an inequality then you probably want an inequality as the loop invariant. Possibly start with the postcondition as the loop invariant and work out what you need to change. Remember that you will need to show \((R \land \neg b) \rightarrow Q\).

- The loop invariant should be invariant to the effect of the loop body. So a good starting point is working out what this effect and then use this to produce an expression invariant to that effect. As some examples:

  - The loop body \(x := x - 1; y := y + 1\) has the effect of making \(x\) smaller by 1 and \(y\) bigger by 1, it is then simple to see that those 1s cancel out in \(x + y\)

  - The loop body \(x := x - 2; y := y + 2\) has the effect of making \(x\) smaller by 1 and \(y\) bigger by 2, we can cancel out the effect of the 2 by multiplying 1 by 2, giving us an invariant expression of \(2x + y\).

  - The loop body \(x := x + y\) has the effect of making \(x\) bigger by \(y\) but this doesn’t immediately give us an invariant expression. If we also know that \(y > 0\), perhaps from the loop condition, then the expression \(x > 0\) is invariant.

Once we have an invariant expression we can easily find an invariant by finding the expressions value at the start of the loop.

- If the loop body contains conditional statements then the loop invariant has to hold for each path through the loop. One approach would be to consider each path separately and then guard them. For example, given a loop body of the form \(\text{if } b \text{ then } S_1 \text{ else } S_2\) one could attempt to build a loop invariant of the form \((b \rightarrow R_1) \land (\neg b \rightarrow R_2)\).

The best practice is to look at examples and try it out. As a last bit of advice, if you write python programs for the while programs you can add assert statements to check if an expression really is invariant.

**How do I find loop variants?**

Due to the Halting Problem we know that it is not possible to produce an algorithm that finds suitable loop variants. But for the kinds of programs we will be looking at loop variants will be easier to find. In most cases the while loops are really for loops with a loop counter decreasing on each step and
the loop terminating when that counter becomes zero. Here I will point out that for such cases we can use a derived rule for total correctness of the form:

\[
\frac{P \land B \left[ b \right] \land E = n}{C \left[ P \land E = n - 1 \right]} \quad \text{if } P \land B \left[ b \right] \rightarrow E \geq 0
\]

where we explicitly show that the expression \( E \) gets smaller by 1 on each iteration.

Here I discuss a few special alternative cases:

- Nothing gets smaller and the loop condition bounds something from above. For example, in \( \text{while } x < 100 \text{ do } x := x + 1 \) the value of \( x \) just gets bigger but the loop terminates when \( x \) is 'big enough'. But there are immediately a few things that decrease. Simply the value of \( -x \), as long as \( x > 0 \) to begin with, and we can show that \((x > 0 \land -x = n) \rightarrow (1 - x < n)\), which is what we would need.

- Either \( E_1 \) or \( E_2 \) will get smaller. In this case you can just use \( E_1 + E_2 \) as a loop variant.

- Either \( E_1 \) gets smaller or \( E_2 \) gets bigger and is always positive. In this case you can use \( E_1 - E_2 \) as a loop invariant as this decreases when \( E_1 \) gets smaller and \( E_2 \) gets bigger (depending on \( E_2 \) being strictly positive).

- The loop condition does not use an inequality so there is no inherent notion of bounding from above or below. In this case you will need to infer the bound. For example, if the loop condition is \( x \neq 0 \) and the loop invariant contains \( x \geq 0 \) then we can show that \( x \) is bounded from below. However, note that \( x \neq 0 \) is not enough as the loop body might be \( x := x - 2 \).

In general you should look at the loop and ask \textit{why should this terminate}? If you can work that out the you should be able to phrase the reason as an expression which strictly decreases.

\textbf{How do I know what implications I need for the rule of consequence?}

I would argue that these appear in well-defined places when constructing a proof, even though you’re allowed to apply the rule of consequence wherever you want. In fact, what I haven’t told you is that there is a way of automating all of this, assuming loop invariant annotations, via the generation of \textit{verification conditions} that can be passed to a theorem prover\(^1\). These verification conditions are the implications that appear whilst constructing a proof. The places they occur when I produce proofs are in my derived assignment rule, i.e. attaching the initial precondition to the new one, and when relating the loop invariant to the initial pre and post conditions.

\textbf{How do I solve the implications?}

This is mostly straightforward maths and logic. The handout has some hints (see the online version as one of the hints was wrong in the printed version). Remember that a proof is an argument that something is true and you just need to write down enough for the argument to be convincing. You don’t need to break down \((a > 0 \land b > 0) \rightarrow (a + b > 0)\) any further, it is clearly true. But something very complex involving a number of reasoning steps should be broken down to show what those steps are.

\(^1\)Chapter 3 of these notes describes how this is done: \url{http://www.cl.cam.ac.uk/~mjcg/Teaching/2015/Hoare/Notes/Notes.pdf}. These notes also discuss the soundness and completeness of the proof system. Other good references exist and if you are interested in this topic do ask me about them.