THE PARTIAL CORRECTNESS ISSUE

This last example has only one pre-condition on x; there is no constraint on y, which can be +ve or –ve. Sometimes this is not quite so easy. The next example is the partner of the last one. We are going to prove that a program to divide two numbers by repeated subtraction is partially correct. The program is:

\[
\begin{align*}
q &:= 0; \\
r &:= x; \\
\text{while } r \text{ greater } y \text{ minus } 1 \text{ do} \\
q &:= q \text{ plus } 1; \\
r &:= r \text{ minus } y \\
\text{end}
\end{align*}
\]

In this program, q is the quotient, i.e. \(x/y\), and r is the remainder. We need the final assertion to be, at least \(q = x/y\), but we can make this stronger, by adding \(r = x \mod y\) as well. The remainder could be zero, when x is an exact multiple of y, but, in any case, the remainder must be less than \(y\). So we could write \(\{q = x/y \land r = x \mod y\}\). However, it makes life simpler if we transform this into the equivalent form \(\{x = q*y + r \land 0 \leq r < y\}\). There is no loss of information here; the two are equivalent, as could be proved if we had definitions of divide, mod, and less than. The pre-condition we will leave for now; let us see what we can prove as the weakest pre-condition by applying the axioms properly. Again we need to find the invariant, and again we could use a table method to divine it. The computation part of the invariant is, in fact, the same as the post-condition: \(x = q*y + r\). The control variable part we can simply write as \(r \geq 0\), which is clearly invariant for the body of the loop. The whole invariant is this:

\[
I \equiv \{x = q*y + r \land r \geq 0\}
\]

To prove the loop, we need to push this back through both assignments:

\[
\{x = q*y + (r-y) \land (r-y) \geq 0\} r := r \text{ minus } y \{x = q*y + r \land r \geq y\}, \text{ and}
\]

\[
\{x = (q+1)*y + (r-y) \land (r-y) \geq 0\} q := q \text{ plus } 1 \{x = q*y + (r-y) \land (r-y) \geq 0\}
\]

This is clearly the same as \(\{I \land b\}\), so using the loop axiom, we can say:

\[
\{I\} \text{ while } \ldots \{I \land \sim(r \geq y)\}
\]

Writing out the post-condition gives us \(\{x = q*y + r \land r \geq 0 \land r < y\}\), or \(\{x = q*y + r \land 0 \leq r < y\}\), which is what we wanted to prove. The initialization section gives:

\[
\{x = q*y + x \land x \geq 0\} r := x \{x = q*y + r \land r \geq 0\}, \text{ and}
\]

\[
\{x = 0*y + x \land x \geq 0\} q := 0 \{x = q*y + x \land x \geq 0\}.
\]

The pre-condition reduces to \(\{x \geq 0\}\), so we have proved that as long as x is greater than or equal to zero, and the loop terminates, we will produce the correct result. This is partial correctness. However, a quick examination of the code tells us that \(\{x \geq 0\}\) is not sufficient in all cases. If y is –ve, then the loop diverges and r gets bigger, rather than reducing to less than y. The pre-condition is really too weak to cover all possibilities. We need, in this case to make it stronger, by adding \(y \geq 1\). This means it gets added all the way through the proof as an invariant “passenger”. It is actually there in the final assertion, since \(y \geq 0\) and \(y \geq 1 \Rightarrow y \geq 0\), so we end up strengthening the pre-condition and (slightly) weakening the post-condition. However, now the program has been proved correct (still partially) and the loop never diverges.

How can we be sure that the loop terminates, as we suspect? This example shows that partial correctness is only useful when the loop terminates, but we haven’t proved it. To do so, we turn to total correctness proofs.
This method, usually attributed to Floyd, is a way to prove that a loop terminates by using the properties of the natural numbers. We know that (by definition): \( 0 < 1 < 2 < 3 \ldots < n \) for any \( n \). Floyd’s strategy is in three parts:

1. prove partial correctness, then
2. find an expression involving the program variables that can be mapped onto the natural numbers, and
3. show that the value of the expression starts out \(+\)ve and reduces (not necessarily, but usually) by one every time round the loop until it reaches zero, when the loop exits. Do this for all loops in the program.

Then the termination condition can be proved to hold for the loop, i.e. the loop will terminate. The natural numbers happen to be an example of a well-founded set, which is an important tool in semantic methods. A set is well-founded if:

- its members can be compared with a relation, usually written as \( \prec \), then
- for members \( a, b, c \), \( a \prec b \land b \prec c \Rightarrow a \prec c \) (transitivity)
- \( a \prec b \Rightarrow \neg (b \prec a) \) (asymmetry)
- \( \neg (a \prec a) \) (irreflexivity)
- there exists a least element, i.e. \( \exists x \in S \forall y \in S, x \prec y \), where \( S \) is the set

The natural numbers, \( \mathbb{N} \), with the relation \( < \), is such a set, but the integers, \( \mathbb{Z} \), with any relation, is not because there is not a least element. Other examples of well-founded sets are finite strings with the substring relation, and any finite set with the proper subset (\( \subset \)) relation.

**THE FLOYD EXPRESSION**

In order to show well-foundedness in a program, we need to find an expression that can be mapped onto the natural numbers. As an example, let us take the third in a trio of simple loop programs: exponentiation by repeated multiplication. The code is:

```plaintext
e := 1;
t := y;
while t greater 0 do
   e := e mult x;
t := t minus 1
end
```

We will not show partial correctness because the proof is very similar to the multiplication example. The specification is also similar, so we can use the Hoare axioms to show:\[ \{y \geq 0\} P \{e = x \ast \ast y\} \] using the loop invariant \[ I \equiv \{e = x \ast \ast (y-t) \land 0 \leq t \leq y\} \]. It is easy in this case to find the Floyd expression. It is just the variable \( t \). However, we need to prove that it is indeed the right expression by showing:

\[ y \geq 0, I \land t > 0 \iff t \geq 0 \]

In general, this is: given the initial pre-condition and the loop entry condition (i.e. \( I \land b \)), try to prove that the Floyd expression is well-founded on 0. (\( \iff \) is the ‘it is possible to prove’ symbol). Since the invariant already contains the expression to be proved, the proof is trivial. So \( t \) is a well-founded expression.

The next step is to prove termination. Now we examine the code and look at all assignments in the loop body that can change the value of the Floyd expression. For each one we need to show that the path through these commands results in a reduction in the value of the Floyd expression. Here, there is only one assignment: \( t := t \) minus 1, so we need to prove:

\[ y \geq 0, I \land t > 0 \iff t > t \left[ t - 1 \right] \]

If there was more than one assignment to \( t \), we would have multiple substitution forms. The general form is similar: from the initial pre-condition and the loop entry condition, \( I \land b \), can we prove that the value of Floyd...
expression is greater than the value obtained after the body finishes. If so, then the value reduces each time round the loop.

The proof is equally trivial, since $t > t - 1$ for all $t$; we don’t even need the left-hand side.

From the two proofs – for well-foundedness and termination – we conclude that the loop will terminate with $t = 0$. 