Consider trying to prove the partial correctness condition:

$$\{A_1\}\ c_1; c_2\ \{A_2\}$$

One would hope that we can use the sequencing rule:

$$\frac{\{A_1\}\ c_1\ \{A\} \qquad \{A\}\ c_2\ \{A_2\}}{\{A_1\}\ c_1; c_2\ \{A_2\}}$$

But how do we know that, when $\{A_1\}\ c_1; c_2\ \{A_2\}$ is true there exists an assertion A such that $\{A_1\}\ c_1\ \{A\}$ and $\{A\}\ c_2\ \{A_2\}$? Perhaps our assertion language is too weak to capture the set of states that correspond to this intermediate step of the computation. That is, our assertion language might not be powerful enough to describe all of the states that we need.

Fortunately it is. Unfortunately, we don't have enough time to prove that this is the case. But I'm going to give you a feel for how this can be done using something called Weakest (Liberal) Pre-Conditions.

Let us define $\mathsf{wp}[\![c, A]\!]$ (weakest liberal pre-condition of a command c and an assertion A) as follows:

$$\mathsf{wp}[\![c, A]\!] = \{s \mid \mathcal{C}[\![c]\!]s \text{ is undefined} \vee \mathcal{C}[\![c]\!]s = s' \wedge s' \vDash A\}$$

That is, $\mathsf{wp}[\![c, A]\!]$ is the set of all states s such that, when we run the command c in state s, we either diverge or else we get an output state that satisfies the assertion A.

Theorem 1. *For each command c and assertion A_2, there exists an assertion A_1 such that $\mathsf{wp}[\![c, A_2]\!] = \{s \mid s \vDash A_1\}$.*

In English, this just says that there is some assertion that can be used to precisely describe the weakest (liberal) pre-conditions of a given command and post-condition.

(Note: We say weakest "liberal" pre-condition to distinguish from true weakest pre-conditions which don't allow for non-termination and are typically used when proving total correctness instead of partial correctness.)

So, if we want to prove $\{A_1\}\ c_1; c_2\ \{A_2\}$, it suffices to calculate $\mathsf{wp}[\![c_2, A_2]\!]$ to generate an assertion A and then show that $\{A_1\}\ c\ \{A\}$ holds. For then we can use the sequencing rule, and the definition of $\mathsf{wp}[\![c_2, A_2]\!]$ to conclude $\{A_1\}\ c_1; c_2\ \{A_2\}$.

Here are the weakest liberal pre-conditions for most of the commands:

$$\begin{aligned}
&\mathsf{wp}[\![\mathsf{skip}, A]\!] = A \\
&\mathsf{wp}[\![x := e, A]\!] = A[e/x] \\
&\mathsf{wp}[\![c_1; c_2, A]\!] = \mathsf{wp}[\![c_1, \mathsf{wp}[\![c_2, A]\!]]\!] \\
&\mathsf{wp}[\![\mathsf{if}\ e\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2, A]\!] = (e \wedge \mathsf{wp}[\![c_1, A]\!]) \vee (\neg e \wedge \mathsf{wp}[\![c_2, A]\!])
\end{aligned}$$

The wp for a while loop is pretty complicated (and not practical) so I won't describe it here. (See Winskel for details if you are interested.)

It's pretty easy to convince yourself (via induction on commands) that:

$$\{\mathsf{wp}[\![c, A]\!]\}\ c\ \{A\}$$

The only interesting case above is for if-commands:

$$\{(e \wedge \mathsf{wp}[\![c_1, A]\!]) \vee (\neg e \wedge \mathsf{wp}[\![c_2, A]\!])\} \text{ if } e \text{ then } c_1 \text{ else } c_2 \ \{A\}$$

Recall the Hoare rule for if:

$$\frac{\{e \wedge A'\}\ c_1\ \{A\} \qquad \{\neg e \wedge A'\}\ c_2\ \{A\}}{\{A'\} \text{ if } e \text{ then } c_1 \text{ else } c_2\ \{A\}}$$

Substituting $(e \wedge \mathsf{wp}[\![c_1, A]\!]) \vee (\neg e \wedge \mathsf{wp}[\![c_2, A]\!])$ for A' above, we must show:

$$\{e \wedge ((e \wedge \mathsf{wp}[\![c_1, A]\!]) \vee (\neg e \wedge \mathsf{wp}[\![c_2, A]\!]))\}\ c_1\ \{A\}$$

and

$$\{\neg e \wedge ((e \wedge \mathsf{wp}[\![c_1, A]\!]) \vee (\neg e \wedge \mathsf{wp}[\![c_2, A]\!]))\}\ c_2\ \{A\}$$

For the first case, we can simplify:

$$\begin{array}{rcl} e \wedge ((e \wedge \mathsf{wp}[\![c_1, A]\!]) \vee (\neg e \wedge \mathsf{wp}[\![c_2, A]\!])) & \iff & (e \wedge e \wedge \mathsf{wp}[\![c_1, A]\!]) \vee (e \wedge \neg e \wedge \mathsf{wp}[\![c_2, A]\!]) \\ & \iff & (e \wedge \mathsf{wp}[\![c_1, A]\!]) \vee \mathsf{false} \\ & \iff & e \wedge \mathsf{wp}[\![c_1, A]\!] \end{array}$$

Similarly, the second case simplifies:

$$\neg e(\wedge(e \wedge \mathsf{wp}[\![c_1, A]\!]) \vee (\neg e \wedge \mathsf{wp}[\![c_2, A]\!])) \iff \neg e \wedge \mathsf{wp}[\![c_2, A]\!]$$

So we only have to show that:

$$\{e \wedge \mathsf{wp}[\![c_1, A]\!]\}\ c_1\ \{A\}$$

and

$$\{\neg e \wedge \mathsf{wp}[\![c_2, A]\!]\}\ c_2\ \{A\}$$

By induction, we have that $\{\mathsf{wp}[\![c_i, A]\!]\}\ c_i\ \{A\}$, and certainly $e \wedge \mathsf{wp}[\![c_1, A]\!] \implies \mathsf{wp}[\![c_1, A]\!]$ and $\neg e \wedge \mathsf{wp}[\![c_2, A]\!] \implies \mathsf{wp}[\![c_2, A]\!]$ so we are done.

What's not so easy to see is that if $\{A'\}\ c\ \{A\}$, then $A' \implies \mathsf{wp}[\![c, A]\!]$. That is, $\mathsf{wp}[\![c, A]\!]$ really is the weakest possible assertion that we can use for the pre-condition on states to run c and get out states described by A. (Hint, hint, hint.)

The nice thing about wp is that, if you give me a (while loop-free) program c, I can automatically reduce checking whether $\{A_1\}\ c\ \{A_2\}$ holds to just checking whether $A_1 \implies \mathsf{wp}[\![c, A_2]\!]$. That's something that we could feed to a general purpose theorem prover for predicate logic (e.g., NuPRL, Boyer Moore, HOL, etc.)

Even if you add while-loops back in, as long as the while loops are annotated with an explicit loop-invariant, then calculating weakest pre-conditions is easy (though not necessarily complete – if you pick a bad invariant, then you might end up stuck in your proof. That is, it may not be that $A_1 \implies \mathsf{wp}[\![c, A_2]\!]$ even though $\{A_1\}\ c\ \{A_2\}$ is true.)

An aside on Proof-Carrying Code (PCC)

Suppose I send you an IMP program c, and I claim that it satisfies some specification $\{\mathsf{Pre}\}\ c\ \{\mathsf{Post}\}$. Suppose further that c includes loop invariants labelled on each while loop.

Now suppose that I give you a machine-checkable proof that $\mathsf{Pre} \implies \mathsf{wp}[\![c, \mathsf{Post}]\!]$. By machine-checkable, I mean that we've formally specified the assertion language and the proof rules that you can use to construct a

valid proof of an assertion in predicate logic. You can check the proof by running over the tree and validating that each node is an instance of one of the proof rules (and that the sub-proofs are valid and so on.) This is a simple walk over the tree that can be coded in a trustworthy manner (about 2-3 pages of ML code.)

Now suppose that your proof checker says "Yep, this is a valid proof that $\mathsf{Pre} \implies \mathsf{wp}[\![c, \mathsf{Post}]\!]$". What can we conclude? Well, we know that if you run c in any input state satisfying Pre, then if c terminates, it will end up in a state satisfying Post.

So, in summary, if I send you a command c annotated with loop invariants, and I send you a proof of $\mathsf{Pre} \implies \mathsf{wp}[\![c, \mathsf{Post}]\!]$, then you can automatically check that c does what it claims to do.

In a modern setting, this corresponds to downloading untrusting code from some site and being able to verify that the program does what it claims to do.

What happens if we tamper with c? For instance, suppose we change it to c'? Well, then when we run $\mathsf{wp}[\![c', \mathsf{Post}]\!]$, we get a (possibly) different assertion than if we run $\mathsf{wp}[\![c, \mathsf{Post}]\!]$. So, our proof that $\mathsf{Pre} \implies \mathsf{wp}[\![c, \mathsf{Post}]\!]$ won't allow us to conclude $\mathsf{Pre} \implies \mathsf{wp}[\![c', \mathsf{Post}]\!]$, so we'll reject the code.

Actually, in some cases, it could be that $\mathsf{wp}[\![c', \mathsf{Post}]\!] = \mathsf{wp}[\![c', \mathsf{Pre}]\!]$ (e.g., if I just insert some bogus skip statements or "$x := x$" statements.) But note that these won't change the behavior of the program!

Now suppose I tamper with the proof P that $\mathsf{Pre} \implies \mathsf{wp}[\![c, \mathsf{Post}]\!]$. Well, you're checking the proof, so if the tampering results in an invalid proof, you'll catch it. But maybe the tampering generates a valid proof, but of a different assertion (e.g., $\mathsf{Pre}' \implies \mathsf{wp}[\![c', \mathsf{Post}']\!]$). Well, then you'll check that the assertion doesn't match what you're looking for.

In short, if I send you code and a proof, then we can use wp to tie the two together in a tamper-proof way. This is the way that proof-carrying code works.

If you want to read more about PCC, then see the following:

```
http://raw.cs.berkeley.edu/pcc.html
```

PCC has been used in real systems to ensure that, for instance, a downloaded module into a kernel does not violate the security policy of the kernel. It has also been used in other settings (think smartcards, phones, etc.) where security and/or reliability are crucial.

PCC eliminates the need to trust a piece of code. However, there are two really hard problems that PCC does not address: (1) how do we formalize the specification of what a program should/ shouldn't be able to do for a specific environment (e.g., a kernel)? (2) how do we construct proofs that $\mathsf{Pre} \implies \mathsf{wp}[\![c, \mathsf{Post}]\!]$?

Solving these two problems is an active area of research these days.