

Iteration Program Correctness

Correctness of Programs without Iteration:

Let pre and post be predicates over states, and let p be a program, then

$$[(p, s) \dashrightarrow Q \wedge \text{pre}(s) \Rightarrow \text{post}(Q)] \text{ implies } p \text{ is correct}$$

for all $s \in \text{state}$. We often write

$$\{\text{pre}\}p\{\text{post}\} \text{ implies } p \text{ is correct}$$

where the state s is implicit.

Program Correctness

Unfortunately this notion of correctness does not extend to the general case of programs with iteration.¹ However, we can still use this proof rule to prove *specific* programs correct. This program computes the factorial of 3:

$$p \equiv \text{while}(le(1, i), \text{assign}(z, \text{mult}(z, i)) \text{ seq } \text{assign}(i, \text{sub}(i, 1)))$$

with the precondition

$$\text{pre}(S) \equiv \text{lookup}(i, S, 3) \wedge \text{lookup}(z, S, 1)$$

and the postcondition

$$\text{post}(Q) \equiv \text{lookup}(i, Q, 0) \wedge \text{lookup}(z, Q, 3!)$$

That is, the program is correct iff

$$\{\text{pre}\} p \{\text{post}\}$$

¹Pre- and post conditions cannot be used to show program correctness for all values of the loop index - you would need some sort of inductive argument as we will see a little later on.

This works because for specific values for i we can *unfold* the iteration and view p essentially as a sequence of statements without iteration:

$$p' \equiv \text{assign}(z, \text{mult}(z, i)) \text{ seq } \text{assign}(i, \text{sub}(i, 1)) \text{ seq} \\ \text{assign}(z, \text{mult}(z, i)) \text{ seq } \text{assign}(i, \text{sub}(i, 1)) \text{ seq} \\ \text{assign}(z, \text{mult}(z, i)) \text{ seq } \text{assign}(i, \text{sub}(i, 1))$$

And is clear that our proof rule

$$\{\text{pre}\} p' \{\text{post}\}$$

applies.²

²One way to show this is to develop pre and post conditions for each statement and then compose them to get the overall correctness proof.

Program Correctness

Now consider our program for any value of i :

$$p \equiv \text{while}(le(1, i), \text{assign}(z, \text{mult}(z, i)) \text{ seq } \text{assign}(i, \text{sub}(i, 1)))$$

with the precondition

$$\text{pre}(S) \equiv \text{lookup}(i, S, vi) \wedge \text{lookup}(z, S, 1)$$

and the postcondition

$$\text{post}(Q) \equiv \text{lookup}(i, S, 0) \wedge \text{lookup}(z, S, vi!)$$

Now, in order to prove p correct we would have to show that our proof rule

$$\{\text{pre}\} p \{\text{post}\}$$

holds for every value of vi , that is, every possible unfolding of the while loop. Clearly that is not possible.

To get around this we could probably come up with some sort of inductive argument on the loop iteration (recall our induction on states a while back).


Turns out that there is a very clever way to do this without induction: *loop invariants*

Loop invariants allow us to perform "inductionless induction".

Definition of *loop invariant*,³

In computer science, a loop invariant is a property of a program loop that is true before (and after) each iteration. It is a logical assertion, sometimes checked within the code by an assertion call. Knowing its invariant(s) is essential in understanding the effect of a loop.

In formal program verification loop invariants are expressed by formal predicate logic and used to prove properties of loops and by extension algorithms that employ loops (usually correctness properties). The loop invariants will be true on entry into a loop and following each iteration, so that on exit from the loop both the loop invariants and the loop termination condition can be guaranteed.

³https://en.wikipedia.org/wiki/Loop_invariant 

In this new approach we divide programs with loops into parts:

$$\{pre\} \text{ init } \mathbf{seq\ while\ } b \mathbf{ do\ } c \mathbf{ end\ } \{post\}$$

where *init* is code that sets up the loop.

We now introduce a new predicate called *inv* that captures the loop invariant in such a way that it relates one iteration to the next in a similar sense as does the inductive step during an inductive argument.

Program Correctness & Iteration

We augment our proof procedure with an additional predicate on states: inv .

With this we can set up the predicates for the loop as follows:

```
{pre}
init seq
{inv}
while  $b$  do
  { $inv \wedge b$ }
   $c$ 
  {inv}
{ $inv \wedge \neg b$ }
{post}
```

NOTE: *We now have pre- and postconditions for each statement in this iterative program. These conditions will hold in all iterations of the loop.*

This gives us a new proof rule for *partial correctness* of loops:

if

$$\begin{aligned} & \{\text{pre}\} \text{ init } \{\text{inv}\} \wedge \\ & \{\text{inv} \wedge b\} c \{\text{inv}\} \wedge \\ & (\text{inv} \wedge \neg b) \Rightarrow \text{post} \end{aligned}$$

then

'init **seq while** *b* **do** *c* **end**' is correct

NOTE: We call this partial correctness because we make no assertions about termination. All we assert is that, if the computation terminates, then it will be correct.

Or written in our notation, partial correctness of loops:

if

$$\begin{aligned} &(\text{init}, S) \dashrightarrow Q \wedge [\text{pre}(S) \Rightarrow \text{inv}(Q)] \wedge \\ &(c, S) \dashrightarrow Q \wedge (b, S) \dashrightarrow B \wedge [(\text{inv}(s) \wedge B) \Rightarrow \text{inv}(Q)] \wedge \\ &(b, T) \dashrightarrow B \wedge [(\text{inv}(T) \wedge \neg B) \Rightarrow \text{post}(T)] \end{aligned}$$

then **'init seq while b do c end'** is correct

Finding loop invariants automatically is an active research, good candidates for loop invariants are usually expression that involve the loop counter.