Chapter 8

Reasoning about program correctness

> "*How do we convince people that in programming simplicity and clarity — in short: what mathematicians call "elegance" — are not a dispensable luxury, but a crucial matter that decides between success and failure?* — Edsger W. Dijkstra [3].

In this chapter, we will look at a very powerful piece of work, known as *Hoare logic*, named after its creator, Tony Hoare, who is currently a principal researcher at Microsoft. This work dates back as far as 1969, when Hoare proposed the first axiomatic basis for computer programming [4]. Taking only a set of few primitives programming constructs (now considered to be the basis of structured programming), Hoare demonstrated a simple logic that enabled programs to be *proved* correctness. Since that time, much research has gone into building on Hoare's initial work, with considerable success.

We will look at Hoare logic and show how to use it to reason about the correctness of programs, allowing us to *prove* that a program meets its contract/specification. This is much more powerful than running tests, which constitute nothing more than a handful of observations about our program, and also more powerful that using review to check for a larger set of cases. For some systems, testing and review are not enough.

By the mid-1970s, there was a general belief that this type of reasoning would pervade software engineering within a decade or two. However, three things happened:

1. The speed and power of computers continued to increase so quickly that the types of systems that could be constructed using software outgrew the techniques for such complex reasoning.
2. Software began to pervade homes and businesses as useful tools, and the *correctness* of these software systems was not as important as it was for systems in which software had previously been applied, such as digital components. Further, time to market became more important than it had previously, so there was an acceptance of lower quality for a faster turnaround.
3. The spread of the Internet has meant that fixing many software applications could be done by users downloading patches, or in the case of server-side problems, developers just uploading a new version to the server. This is not sufficient for many safety-critical system, in which the software resides in a non-PC and non-server environment (e.g. on a car ECU).

It is my belief that the techniques presented this chapter are the most important of the subject — yet are also the techniques that I can confidently state none of you will ever apply as they are applied in these

notes. The importance of this chapter is to change the way that you think about programs, the science of programming, and of software design and verification, in order to make you better programmers and software engineers.

Learning outcomes

The learning outcomes of this chapter are:

1. To critique the difference between program proof and other forms of verification, such as testing.
2. To be able to prove the correctness of simple programs against their contracts.
3. To think about programming in a more systematic and mathematical manner, with the result of improving our programming skills.

8.1 Introduction to reasoning about programs

The first thing to ask is why we would want to reason about programs at all. In previous chapters, we have looked at how to formally specify and design (by contract) our system, and to prove that the specification and design satisfy certain properties. Proof of formal statements are possible because we are operating in space of logic and set theory, which have well-founded proof theories.

However, for high integrity systems, we want to ensure that our designs are implemented correctly. That is, we want to show that our code corresponds to our design.

Quality assurance techniques such as peer review and testing seem to offer very good reliability for software. However, with respect to high-integrity constraints, they do not offer the level of assurance that is required. What if we have missed a few cases that can cause catastrophic consequences?

This chapter covers the theory of how we can *prove* that a program meets its contract. This is a powerful ability that provides us with assurance far above what is possible with testing and review.

8.1.1 The correctness of binary search

Let's motivate the idea of reasoning about programs using the following story from Jon Bentley:

> "Binary search solves the problem [of searching within a pre-sorted array] by keeping track of a range within the array in which T [i.e. the sought value] must be if it is anywhere in the array. Initially, the range is the entire array. The range is shrunk by comparing its middle element to T and discarding half the range. The process continues until T is discovered in the array, or until the range in which it must lie is known to be empty. In an N-element table, the search uses roughly log(2) N comparisons.
>
> Most programmers think that with the above description in hand, writing the code is easy; they're wrong. The only way you'll believe this is by putting down this column right now and writing the code yourself. Try it.
>
> I've assigned this problem in courses at Bell Labs and IBM. Professional programmers had a couple of hours to convert the above description into a program in the language of their

> choice; a high-level pseudocode was fine. At the end of the specified time, almost all the programmers reported that they had correct code for the task. We would then take thirty minutes to examine their code, which the programmers did with test cases. In several classes and with over a hundred programmers, the results varied little: ninety percent of the programmers found bugs in their programs (and I wasn't always convinced of the correctness of the code in which no bugs were found).
>
> I was amazed: given ample time, only about ten percent of professional programmers were able to get this small program right. But they aren't the only ones to find this task difficult: in the history in Section 6.2.1 of his Sorting and Searching, Knuth points out that while the first binary search was published in 1946, the first published binary search without bugs did not appear until 1962."
>
> — Jon Bentley, Programming Pearls (1st edition), pp. 35-36 [1].

Donald Knuth mentions in *The Art of Computer Programming* [6] that the first version of the binary search algorithm appeared in 1946, but it was not until 1962 that the first *correct* version appeared. All previous versions contained faults.

Further, the most well-known implementation, written and "*proved*" by Jon Bentley, which has been adapted many times, contains a fault. The truth is, very few correct versions of the algorithm exist.

Exercise 8.1
There is a fault in the following widely-tested and widely-used version of binary search. Find this fault. TIP: Converting to SPARK and using the SPARK tools to verify the program will reveal the fault.

```
1  int binarySearch(int [] list, int target)
2  {
3    int low = 0;
4    int high = len(list) - 1;
5    int mid;
6
7    while(low <= high) {
8      mid = (low + high)/2;
9      if (list[mid] < target) {
10       low = mid + 1;
11     }
12     else if (list[mid] > target) {
13       high = mid - 1;
14     }
15     else {
16       return mid;
17     }
18   }
19   return -1;
20 }
```

□

The question then is: if some of the best and brightest minds in computer science cannot get a seemingly simple algorithm correct using testing, and dozens of versions in textbooks can be published with faults that remain undetected for decades, what hope do we have for producing correct software the first time?

In this chapter, we will look at Hoare logic as one approach for proving a program correct with respect to its specification. In workshops, we will look at powerful support provided by the SPARK tools that can assist us in this goal.

8.2 A small programming language

Before we look at Hoare logic, we'll define a small structured programming language that is Turing compatible, and which will form the basis of the mathematical objects in Hoare's logic. The language consists of procedures, variables, expressions (numerical expressions and arrays of expressions), variable assignment, sequencing, conditionals, loops, and procedure calls:

Procedures:

P := **procedure** $p(v_1, \ldots, v_n) \mathrel{\widehat{=}} S$

where $v_1, \ldots, v_n$ are variable names, p is a procedure name, and S is a program statement (defined below).

Expressions:

E := $NE \mid a[NE]$

NE := $n \mid v \mid E_1 + E_2 \mid E_1 - E_2 \mid E_1 \times E_2 \mid E_1/E_2 \mid$ etc...

where n is a number, v is a variable, a is an array variable, NE is a numerical expression, and E an expression.

Booleans

B := **true** | **false** | $\neg B \mid E_1 = E_2 \mid E_1 \leq E_2 \mid$ etc...

Statements

S := **skip** | $v := E \mid a[NE] := E \mid p(E_1, \ldots, E_n) \mid S_1;\ S_2 \mid$
if B **then** S_1 **else** S_2 **endif** | **while** B **do** C **done**

where v is a variable, a is an array variable, and p is a procedure name.

The **skip** statement is a statement that has no effect and always executes successfully, the same as `null` in Ada.

A *program* in this language consists of a (possibly empty) sequence of procedure definitions, and a final "main" program; e.g.:

procedure $p_1(v_1, \ldots, v_m) \mathrel{\widehat{=}} S_1$
...
procedure $p_n(v_1, \ldots, v_n) \mathrel{\widehat{=}} S_n$
S

where S is the main program, which may reference procedures $p_1, \ldots, p_n$.

Using this simple language, we can construct any program that possible with more expressive languages, such as those with references, but the program may be more long-winded.

Example 8.1
Using this language, we can construct a program that calculates the factorial of 10 an assigns it to a variable f (where n is an input variable and f is an output variable):

□

Algorithm 1 A program for calculating the factorial of a number

```
procedure FACTORIAL(in n, out f)
    f := 1
    i := 0
    while i ≠ n do
        i := i + 1
        f := f × I
    end while
end procedure
FACTORIAL(10,f)
```

8.3 Hoare logic

Hoare logic [4] is an axiomatisation of programming. Hoare's aim was to treat computer programs as mathematical objects, and to provide a sound and complete set of rules for proving things about these objects.

8.3.1 Introduction to Hoare logic

Hoare logic is a set of rules for reasoning about programs. The rules are organised around the program statements in the simple programming language defined above. Any program in the language is a collection of statements, as defined by the grammar S, with a (possibly empty) set of procedures that it references.

The rules are applied to the program in a top down manner, until the final result is the application of rules to atomic statements: **skip** or variable assignments ($V := E$). The rules for compound program statements, such as sequencing and branching, require us to prove properties about the statements that make up the compound statement.

Statements about programs are written using *Hoare triples*, which take the following form:

$$\{P\}\ S\ \{Q\}$$

This states that, given a program S with precondition P, if P is true when S is executed, then S will establish the postcondition Q. In other words, a Hoare triple describes how executing a program changes the variables in the program.

Hoare logic is a collection of axioms and rules of this form that can be used to reason about the correctness of programs; similar to using Peano's axioms to reason about numbers. To prove correctness of our program, we need to show that $\{P\}S\{Q\}$ is true, using Hoare's axioms and rules.

The rules are written as inference rules, of the form:

$$\frac{H_1, \ldots, H_n}{C}$$

where $H_1, \ldots, H_n$ are a collection of hypotheses, and C is the conclusion. An inference rule should be read as: if hypotheses $H_1, \ldots, H_n$ can all be proved, then the conclusion C is true. Thus, to prove C, we must prove $H_1, \ldots, H_n$.

For example, consider the most famous inference rule of all — *modus ponens*:

$$\frac{A, A \Rightarrow B}{B}$$

This states that, if A is true and $A \Rightarrow B$ is true, then we can infer that B must be true.

An inference rule with no hypotheses is called an *axiom*, and in often just written as the conclusion with no line. For example, the following is an axiom of propositional logic:

$$A \Rightarrow (B \Rightarrow A)$$

That is, this proposition is always true. This could be written as:

$$\frac{\mathit{true}}{A \Rightarrow (B \Rightarrow A)}$$

but the line and premise are often omitted for readability.

In Hoare logic, a hypothesis can be either a predicate or a Hoare triple, and the conclusion is always a Hoare triple.

8.3.2 An introductory example — Incrementing an integer

Consider the following SPARK program used to increment a variable:

```
1  procedure Inc (X: in out Integer)
2  --# pre X >= 0;
3  --# post X > 0;
4  is begin
5     X := X + 1;
6  end Inc;
```

Can we *prove* that this program establishes its postcondition for all `X` that satisfy the precondition? It may seem quite easy for us to reason about this in our head to convince ourselves that the program is correct, however, how could we go about automating or mechanising this?

8.4 The rules of Hoare logic

In this section, we present the fundamental rules of Hoare logic, and present several examples.

8.4.1 Assignment axiom

The assignment axiom is specified as a triple:

$$\{P[E/x]\}\ x := E\ \{P\}$$

where $P[E/x]$ represents substituting all occurrences of the variable x with expression E in P. Note that this rule has no hypotheses, just a conclusion, so it is an axiom rather than an inference rule.

This rule says that, if we want to prove that P is true after executing the statement $x := E$, then we need to prove that $P[E/x]$ is true before executing $x := E$.

When using this rule on a program proof, we *derive* $P[E/x]$ from the program and the postcondition. $P[E/x]$ is called the *weakest precondition* of S that establishes P. That is, it is the weakest, or more general, predicate for which program S establishes the postcondition P.

Example 8.2
The following are two simple Hoare triples that are true:

$$\{y = 0\}\ x := y\ \{x = 0\}$$
$$\{5 + y \geq 0\}\ x := 5\ \{x + y \geq 0\}$$

In both triples above, the variable x has been substituted by the assigned expression to form the precondition.

□

Example 8.3
So, using the assignment axiom, can we prove that our program `Inc` establishes its postcondition?

To do this, we need to prove the following:

$$\{x \geq 0\}\ x := x + 1\ \{x > 0\}$$

From Hoare's assignment axiom, we derive the weakest precondition of the program $x := x + 1$ from the postcondition $x > 0$. If we substitute $x + 1$ for all occurrences of x in $x > 0$, we are left with $x + 1 > 0$, and the resulting proof we need to show is:

$$\{x + 1 > 0\}\ x := x + 1\ \{x > 0\}$$

However, the precondition of this statement is not the same (syntactically) as the precondition of the original program. To finish the proof, we need to introduce another rule, which we introduce now.

□

8.4.2 Consequence rule

Hoare's consequence rule is as follows:

$$\frac{P' \Rightarrow P,\ Q \Rightarrow Q',\ \{P\}\ S\ \{Q\}}{\{P'\}\ S\ \{Q'\}}$$

This states that, if we can establish that $\{P\}\ S\ \{Q\}$ is true, then any precondition that is at least as *strong* as P, and any postcondition that is at least as *weak* as Q, are also valid preconditions/postconditions for the program S.

Example 8.4
Some examples:

$\{x \geq 0\}\ x := x + 1\ \{x > 0\}$ (because $x \geq 0\ \Rightarrow\ x + 1 > 0$)
$\{5 + y \geq 0 \wedge z = 0\}\ x := 5\ \{x + y \geq 0\}$ (a strengthened postcondition)
$\{5 + y \geq 0\}\ x := 5\ \{\mathit{true}\}$ (a weakened postcondition)

□

Proving assignment statements

To prove that an assignment statement establishes its postcondition for some precondition other than the weakest precondition, we need to prove that $Q \Rightarrow P[E/x]$, where Q is the precondition. For example, to prove the following triple:

$$\{\text{true}\}\ x := 5\ \{x \geq 5\}$$

Apply the assignment axiom:

$$\{5 \geq 5\}\ x := 5\ \{x \geq 5\},$$

and then prove *true* $\Rightarrow 5 \geq 5$, which is trivially true.

Example 8.5
Let's return to our example of the `Inc` procedure. We need to prove the following:

$$\{x \geq 0\}\ x := x + 1\ \{x > 0\}$$

Applying the assignment rule, we get:

$$\{x + 1 > 0\}\ x := x + 1\ \{x > 0\}$$

Now, we need to prove that the precondition of the original program implies the weakest precondition above. That is, we know that the program establishes its postcondition if the precondition is $x + 1 > 0$, but does it prove it if $x \geq 0$? We need to prove:

$$x \geq 0 \Rightarrow x + 1 > 0,$$

which holds trivially: if x itself is greater than or equal to 0, then $x + 1$ must be strictly greater then 0. Therefore, we have proved that our program establishes its postcondition when the precondition holds.

□

8.4.3 Sequential composition rule

Consider the following SPARK program for swapping two numbers:

```
1  procedure Swap (X, Y : in out Float)
2  --# global T;
3  --# derives X from Y & Y from X;
4  --# post X = Y~ and Y = X~
5  is begin
6      T := X;
7      X := Y;
8      Y := T;
9  end Swap;
```

Using Hoare logic, we need to prove the triple:

$\{\text{true}\}$
$t := x;$
$x := y;$
$y := t$
$\{x = Y \wedge y = X\}$

Auxiliary variables

Before we tackle this proof, consider using a programming language that allows simultaneous assignment; e.g. $x, y := y, x;$ which means that x gets the value of y and y gets the value of x simultaneously, eliminating our need for the "temp" variable. If our swap program is implemented using that, we have to prove the triple:

$$\{\text{true}\}\ x, y := y, x\ \{x = Y \wedge y = X\},$$

in which X and Y are the values of x and y before the program is executed — like `X` in Ada.

Applying the assignment axiom:

$$\{y = Y \wedge x = X\}\ x, y := y, x\ \{x = Y \wedge y = X\}$$

How do we prove $\mathit{true} \Rightarrow y = Y \wedge x = X$? The trick here is that we don't have to! The variables X and Y cannot occur in the program: they are *auxiliary variables*, and represent the value of the corresponding variable *before* the program is executed. All free variables in a program have an auxiliary variable that specifies its pre-execution value.

At any point in a program, x represents the current value of the variable x, where "current" means immediately after the previous statement is executed, and X represents the pre-execution value. Therefore, at the start of the program, before any statement has been executed, it must be that $x = X$. That is, a variable is always equal to its pre-execution value before any part of the program has executed. As a result, $x = X$ is an implicit part of every precondition, so we do not need to prove propositions of the form $x = X$ in the precondition — they always hold.

Reasoning about swap using the assignment axiom — incorrectly

Now, let's return to the original definition of the swap program.

The assignment axiom on its own is not sufficient to prove this program is correct. If we simply substitute in the assignment expressions for the variables, we end up with the following:

$$\{y = Y \wedge t = X\}\ t := x;\ x := y;\ y := t\ \{x = Y \wedge y = X\}$$

Here, we cannot prove $t = X$. And in fact, this does not need to be true for the program to establish its postcondition. The problem here is that we attempted to apply the assignment rule to a *sequence* of (non-simultaneous) assignment statements, which does not work.

Instead, we have to "work backwards" through the program sequential composition used in the program.

Sequential composition rule

To solve this problem, we use Hoare's sequential composition rule:

$$\frac{\{P\}\ S_1\ \{R\},\ \ \{R\}\ S_2\ \{Q\}}{\{P\}\ S_1;\ S_2\ \{Q\}}$$

This states that, to prove that the program $S_1;\ S_2$ will achieve the postcondition Q from precondition P, we need to prove that the subprogram S_1 can get to some intermediate state R, and from R, subprogram S_2 will establish Q.

But how do we determine what R should be? The weakest precondition of S_2!

Example 8.6
Finally, we can reason about the correctness of the "swap" program correctly. First, we apply the sequential composition axiom:

$\{\text{true}\}$
$t := x;\ x := y$
$\{R\}$
$y := t$
$\{x = Y \wedge y = X\}$

Note here that we are concatenating two Hoare triples as shorthand, such that

$$\{P\}\ S_1\ \{R\}, \{R\}\ S_2\ \{Q\}$$

is equivalent to

$$\{P\}\ S_1\ \{R\}\ S_2\ \{Q\}.$$

Then, we find the weakest precondition, R, using the assignment axiom:

$\{\text{true}\}$
$t := x;\ x := y$
$\{x = Y \wedge t = X\}$
$y := t$
$\{x = Y \wedge y = X\}$

And then apply the sequential composition rule to the top triple:

$\{\text{true}\}$
$t := x$
$\{y = Y \wedge t = X\}$
$x := y$
$\{x = Y \wedge t = X\},$
$y := t$
$\{x = Y \wedge y = X\}$

The bottom two triples are trivially true from the assignment axiom, so we need to just prove the top one. Again, we apply the assignment axiom to get:

$$\{y = Y \wedge x = X\}\ t := x\ \{y = Y \wedge t = X\}.$$

The above holds trivially from the assignment axiom, so all that is left is to prove that the weakest precondition above proves the precondition (*true*) of the swap program (the consequence rule). For this, we need to prove:

$$y = Y \wedge x = X \ \Rightarrow\ true$$

which holds trivially — everything proves *true*. In fact, the prove is equivalent to proving $true \Rightarrow true$, because both $y = Y$ and $x = X$ hold trivially at the start of the program.

The three triples are all true, so our program is now proved to establish its postcondition.

□

8.4.4 Empty statement axiom

The empty statement axiom asserts that the **skip** statement changes nothing, so whatever holds before **skip** is executed will hold afterwards:

$$\{P\}\ \textbf{skip}\ \{P\}$$

This axiom is trivial, but is important for completeness of the axioms.

8.4.5 Conditional rule

The next rule we introduce is for proving programs containing conditional statements. We assume that if-then-else statements are the only conditionals, but other conditionals are just shorthand for if-then-else, and even then, the reasoning rules are much the same.

The conditional rule is:

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

This states that, to prove that the program **if** B **then** S_1 **else** S_2 **endif** establishes the postcondition Q from precondition P, we need to prove that each branch establishes the same. However, we assume that B holds for the true branch, and that $\neg B$ holds for the false branch.

Example 8.7
Conditional swapping

The illustrate this, consider our swap program again, except that it only swaps the values of x and y if $x < y$. In other words, it establishes the postcondition $x \geq y$:

$\{true\}$
if $x < y$ **then**
    $x := t;$
    $x := y;$
    $y := t$

else
    skip
end if
$\{x \geq y\}$

As a side note, the statement **if** B **then** S **else skip endif** is equivalent to just **if** B **then** S **endif** (that is, removing the **else** branch — a one-armed branch statement), so there is no need to a rule covering this case.

For our conditional swap program, we must prove the following two triples:

$\{x < y\}\ t := x;\ x := y;\ y := t\ \{x \geq y\}$
$\{x \geq y\}$ **skip** $\{x \geq y\}$

Going backwards through the first triple, we can reason about this as follows:

$\{y \geq x\}$
$t := x$
$\{y \geq t\}$
$x := y$
$\{x \geq t\}$
$y := t$
$\{x \geq y\}$

All that remains is to apply the consequence rule to show that the weakest precondition implies the precondition of the first Hoare triple; that is, $x < y \Rightarrow y \geq x$. This holds trivially, therefore, the first triple is proved.

What remains is to prove the second triple:

$\{x \geq y\}$ **skip** $\{x \geq y\}$,

which holds trivially from the empty statement axiom.

We have proved that both branches of the statement establish the postcondition, and therefore, the entire program establishes its postcondition.

□

8.4.6 Iteration rule

Now, we get to the most difficult-to-apply rule in Hoare logic: the *iteration* rule. As it name says, it is used to reason about iteration in programs. As with conditionals, we assume only one basic iteration operator, the *while* loop, but other type of loops, such as *for* and *do-while* loops, are straightforward extensions from this.

The iteration rule is:

$$\frac{\{P \wedge B\}\ S\ \{P\}}{\{P\}\ \textbf{while}\ B\ \textbf{do}\ S\ \textbf{done}\ \{\neg B \wedge P\}}$$

In this rule, the predicate P is the *loop invariant*. This rule states that, provided B is true, if the loop invariant holds before the loop body S executes, and holds after S executes, then the while loop will preserve the loop invariant as well. In addition, the while loop will establish the condition $\neg B$ — the branch condition B must be false for the loop to exit. Therefore, a loop invariant is a property that holds before the loop, at the end of each iteration, and after the loop.

Reasoning about loops

To reason about a while loop, we must actually calculate the loop invariant. It turns out that this is by far the hardest part about applying Hoare logic.

The loop body change the variables of the program. Our job is to find a relationship between the values of the variables that are preserved by the execution of the loop body. That is, the individual values of the variables may change, but the relationship between them does not.

If we want to prove a postcondition that is different to the loop invariant; e.g.:

$$\{P\}\ \textbf{while}\ B\ \textbf{do}\ S\ \textbf{done}\ \{Q\}$$

we use the iteration rule, and then prove that $(\neg B \wedge P) \Rightarrow Q$ using the consequence rule.

Example 8.8
To demonstrate the rule as well as the concept of loop invariants, we'll consider the following SPARK program for calculating the factorial of a number:

```
1  procedure Factorial (N : in Integer,
2                       F : out Integer)
3  --# pre N >= 0
4  --# post F = Fact(N)
5  is
6    I : Integer;
7  begin
8    F := 1;
9    I := 0;
10   while I /= N do
11     I := I + 1;
12     F := F * I;
13   done;
14 end Factorial;
```

In this procedure, the expression `Fact(N)` refers to the factorial mathematical function.

To prove this program establishes its postcondition under its precondition, we need to prove the following triple:

$$\begin{array}{l}\{n \geq 0\}\\ f := 1;\\ i := 0;\\ \textbf{while}\ i \neq n\ \textbf{do}\\ \quad i := i + 1;\\ \quad f := f * i;\\ \textbf{end while}\end{array}$$

$$\{f = n!\}$$

where $n!$ is defined as the factorial function:

$$\begin{array}{lcl} 0! & = & 1 \\ n! & = & n \times (n-1)! \quad \text{where } n > 0 \end{array}$$

First, we need to establish the loop invariant.

The factorial loop invariant

The loop invariant for the factorial program is:

$$f = i!$$

That is, f is the factorial of i before the loop, and this is re-established after each iteration, including the final iteration. In this program, as with others, the loop invariant is generally based on the postcondition that the loop needs to establish.

Back to the proof

The first step is to apply the sequential composition rule to split the program into its constituent parts:

```
1:  {n ≥ 0}
2:  f := 1;
3:  i := 0;
4:  {R}                  ← new annotation (from composition rule)
5:  while i ≠ n do
6:      i := i + 1;
7:      f := f * i;
8:  end while
9:  {f = n!}
```

To find R, we need to calculate the weakest precondition of the while loop using the iteration rule. From the iteration rule, we know that the final result will be $\neg B \wedge P$ — the negation of the branch (the loop has exited) and the loop invariant. We annotate our proof such that after the loop, the branch is false and the invariant holds:

```
1:  {n ≥ 0}
2:  f := 1;
3:  i := 0;
4:  {R}
5:  while i ≠ n do
6:      i := i + 1;
7:      f := f * i;
8:  end while
9:  {i = n ∧ f = i!}          ← new annotation
10: {f = n!}
```

We need to prove that this new annotation, $(f = i! \wedge i = n)$, implies the postcondition. That is, we need to show:

$$(f = i! \wedge i = n) \Rightarrow f = n!.$$

We can substitute n for i in the premise using the *one-point rule*, leaving $f = n! \Rightarrow f = n!$, which is true. Essentially, we can now eliminate the postcondition from the proof (although it will remain for completeness in these notes).

The other thing we know from the iteration rule is that the invariant must hold at the both the start and the end of the loop. Also, at the start of the loop, the Boolean expression in the branch must be true. So, we annotate the start and end of the loop body with the invariant, and just the start of the loop with the negated branch condition:

```
1:   {n ≥ 0}
2:   f := 1;
3:   i := 0;
4:   {R}
5:   while i ≠ n do
6:       {f = i! ∧ i ≠ n}          ← new annotation (loop invariant and negated branch condition)
7:       i := i + 1;
8:       f := f * i;
9:       {f = i!}                  ← new annotation (loop invariant)
10:  end while
11:  {f = i! ∧ i = n}
12:  {f = n!}
```

We have already established that the annotation at line 11 implies the postcondition at line 12. The next thing to prove is that the loop body preserves the loop invariant. This involves proving the following Hoare triple:

$$\{f = i! \wedge i \neq n\}$$
$$i := i + 1;$$
$$f := f * i;$$
$$\{f = i!\}$$

We apply the sequential composition rule and the assignment axiom to establish the weakest precondition of the sequential composition:

```
1:   {f = i! ∧ i ≠ n}
2:   {f × (i + 1) = (i + 1)!}   ← new annotation (weakest precondition of i := i + 1)
3:   i := i + 1;
4:   {f * i = i!}               ← new annotation (weakest precondition of f := f * i)
5:   f := f * i;
6:   {f = i!}
```

We now need to prove that the loop invariant and the guard establish the weakest precondition of the loop body. For the factorial program, this means we must prove the following:

$$(f = i! \wedge i \neq n) \Rightarrow f \times (i + 1) = (i + 1)!$$

From the definition of the ! operator, we know that $(i+1)! = (i+1) \times i!$. If we take the right-hand size of the implication and substitute this, we get:

$$(f = i! \wedge i \neq n) \Rightarrow f \times (i+1) = (i+1) \times i!$$

Dividing both sides by $i+1$ leaves us with

$$(f = i! \wedge i \neq n) \Rightarrow f = i!$$

which holds. Therefore, we have established that the loop preserves the loop invariant. Further to this, we have successfully applied the iteration rule, means that we now have the conclusion: $\{P\}$ **while** B **do** S **done** $\{Q\}$. This means that we have established that the intermediate predicate R can be replaced with just P, leaving our proof in the following state:

1: $\{n \geq 0\}$
2: $f := 1;$
3: $i := 0;$
4: $\{f = i!\}$
5: **while** $i \neq n$ **do**
6: $\{f = i! \wedge i \neq n\}$
7: $\{f \times (i+1) = (i+1)!\}$
8: $i := i + 1;$
9: $\{f * i = i!\}$
10: $f := f * i;$
11: $\{f = i!\}$
12: **end while**
13: $\{f = i! \wedge i = n\}$
14: $\{f = n!\}$

The iteration rule has been successfully applied, completing the first part of the proof of the sequential composition rule. What remains is to prove the first part of the sequential composition:

$\{n \geq 0\}$
$f := 1;$
$i := 0;$
$\{f = i!\}$

That is, we need to prove that the first part of the program (the assignment of initial values to the variables) establishes the loop invariant. We do this by applying the sequential composition rule, and two applications of the assignment rule:

$\{n \geq 0\}$
$\{1 = 0!\}$
$f := 1;$
$\{f = 0!\}$
$i := 0;$
$\{f = i!\}$

To complete the proof, we need to show that the precondition implies the weakest precondition above:

$$n \geq 0 \Rightarrow 1 = 0!$$

$1 = 0!$ is true from the definition of factorial, so this is true, which completes our proof of the second part of the sequential composition rule, meaning that the entire sequential composition (the variable initialisation composed with the while loop — the factorial program) establishes its postcondition whenever its precondition it true.

A re-cap

To recap the proof of the factorial program, we did the following (in reverse order):

1. We proved that the loop invariant was established just prior to executing the loop; that is, the initial variable assignment established the loop invariant.
2. We proved that the loop maintains the loop invariant; that is, the loop invariant holds at both the start of the loop body and the end of the loop body.
3. Using our loop invariant, we proved that the loop establishes the postcondition $f = n!$ after the loop finishes executing.

These are the necessary conditions to prove the factorial program using the iteration rule, so this proves that the factorial program establishes its postcondition.

□

Loops and termination

The iteration rule presented in this section is not valid for all while loops: it does not consider the case in which the loop does not terminate.

Example 8.9
Consider the following non-terminating while loop, with precondition and postcondition:

$\{y \geq 0\}$
while $y \geq 0$ **do**
    $x := 0;$
end while
$\{y \geq 0\}$

Applying the iteration rule, we can prove the hypothesis:

$$\{y \geq 0\}\ x := 0\{y \geq 0\}$$

which allows us to conclude the following:

$\{y \geq 0\}$
while $y \geq 0$ **do**
    $x := 0;$
end while

$$\{y \geq 0 \wedge \neg y \geq 0\}$$

The concluded postcondition is a contradiction. Not only does our loop achieve no postcondition because it never terminates, if it did, it could not possibly achieve the postcondition $y \geq 0 \wedge \neg y \geq 0$.

□

This means that the iteration rule only allows us to prove *partial correctness*. The correctness is partial because we have not proved that the program terminates, and thus, it may not even achieve its postcondition. What we have proved is that, *if the program does terminate*, then it will satisfy the postcondition.

The solution is a modified rule that must include a proof that the loop terminates. Including the modified rule gives us the ability to prove *total correctness*. We can define total correctness as:

$$\textit{total correctness} := \textit{partial correctness} + \textit{termination}$$

We will not consider total correctness in completeness in these notes, however, for the interested reader, the rule is as follows:

$$\frac{[P \wedge B \wedge E = n]\ S\ [P \wedge E < n], \quad P \wedge S \Rightarrow E \geq 0}{[P]\ \textbf{while}\ B\ \textbf{do}\ S\ \textbf{done}\ [\neg B \wedge P]}$$

Note that the Hoare triple is now expressed with square brackets instead of curly brackets — this denotes a rule of total correctness.

In the above rule, the expression E refers to a *loop variant*. The idea behind the rule is to demonstrate that the loop variant E decreases on each iteration of the loop, implying that the loop must terminate.

We will not consider total correctness further in these notes. However, the distinction between partial and total correctness is an important one.

8.4.7 Establishing loop invariants

Establishing loop invariants automatically is an open and challenging research question. This is good news for researchers in the area, but bad news for those of us that want to prove programs correct.

Loop invariants are important because knowing the loop invariant provides you with a stronger property to work with when showing that the loop establishes the required postcondition. There are three properties that we require in a loop invariant, I:

1. It should hold before the loop executes. That is, if the annotation before the loop is P, then $P \Rightarrow I$. It must be *weaker* than its precondition.
2. With the negated branch condition, $\neg B$, it should establish the result that we want. That is, if the result we want is Q, then $I \wedge \neg B \Rightarrow Q$. It must be *stronger* than its postcondition.
3. With the branch condition, B, the body must re-establish the invariant. That is, $\{B \wedge I\}\ S\ \{I\}$.

Heuristics for finding loop invariants

The above properties are not enough to find loop invariants. While finding loop invariants requires some creativity, there are heuristics to help with this. The following heuristics can help us to infer loop invariants:

1. Loop invariants generally contain most of the variables from the loop condition, loop body, the precondition, and the postcondition (where we mean the precondition/postcondition of the loop, not necessarily the entire program).
2. They generally state a relationship between the variables.
3. They generally contain a predicate that is closely related to the postcondition, but is not the same.
4. They hold even if the loop condition is false.

Example 8.10
In the factorial program, we have the following:

1. Initially, $f = 1$ and $i = 0$.
2. After termination, we want that $f = n!$ and $i = n$.
3. At each loop, i and f are both increased.

From this, we can infer that a good loop invariant is $f = i!$.

□

Example 8.11
As an example of loop invariants and the iteration rule, let's consider the following program that sums the values in an array of length N:

```
1  procedure Summation (N : in Integer; A : in IntArray; Sum : out Integer)
2  --# derives Sum from A & N;
3  --# pre N >= 0
4  --# post Sum = ∑_{j=0}^{N-1} A(J)
5  is begin
6     I := 0;
7     Sum := 0;
8     while I /= N do
9        Sum := Sum + A(I);
10       I := I + 1;
11    done;
12 end Summation;
```

What is the loop invariant for this program? We can eyeball the program to see that it will need to support precondition of $i = 0 \wedge sum = 0$, and the postcondition $sum = \sum_{j=0}^{N-1} a[j]$.

What we need to look for is a predicate that is close to the postcondition, but describes a relationship between *sum* and *i*. We can use what we know about the program to help us. The loop body is summing up elements from an array, adding each element, $a[i]$, one at a time, and using the variable *sum* to record the result. We know that *i* is 0 initially and is incremented until it reaches N. The loop exits when $i = N$, so a good guess is to replace *i* with N in the postcondition to get:

$$sum = \sum_{j=0}^{i-1}$$

This becomes our loop invariant. Now, we can annotate our program with the loop invariant in the right places (before/after the loop, and before/after the loop body), and work backwards to get the following annotations:

$\{N \geq 0\}$
$\{0 = \sum_{j=0}^{-1}\}$
$i := 0;$
$\{0 = \sum_{j=0}^{i-1}\}$
$sum := 0;$
$\{sum = \sum_{j=0}^{i-1}\}$
while $i \neq N$ **do**
    $\{i \neq N \wedge sum = \sum_{j=0}^{i-1}\}$
    $\{sum + a[i] = \sum_{j=0}^{i}\}$
    $sum := sum + a[i];$
    $\{sum = \sum_{j=0}^{i}\}$
    $i := i + 1;$
    $\{sum = \sum_{j=0}^{i-1}\}$
end while
$\{i = N \wedge sum = \sum_{j=0}^{i-1}\}$
$\{sum = \sum_{j=0}^{N-1} a[j]\}$

Using these, we need to prove:

1. That the precondition entails $0 = \sum_{j=0}^{-1}$ (the consequence rule — strengthening the precondition):

   The predicate $0 = \sum_{j=0}^{-1}$ holds from the definition of $\sum$, because we are summing a list from index 0 to index -1 (an empty list), and the sum of an empty list is 0.

2. That the loop invariant is established at the start of the loop body:

   $$i \neq N \wedge sum = \sum_{j=0}^{i-1} \quad \Rightarrow \quad sum + a[i] = \sum_{j=0}^{i},$$

   which holds by reasoning that if *sum* is the sum of the array from 0 to $i - 1$, then the sum of the array from 0 to *i* must be $sum + a[i]$.

3. That the invariant and negation of the loop condition establish the postcondition:

   $$i = N \wedge sum = \sum_{j=0}^{i-1} \quad \Rightarrow \quad sum = \sum_{j=0}^{N-1} a[j],$$

   which holds be using the *one-point rule* on $i = N$ and substituting N for the value of i in the premise.

Proving these three conditions means that our summation algorithm conforms to its contract.

□

8.4.8 Array assignment rule

Our simple programming language permits arrays. The values of array variables and their indices can be treated as any other expression. That is, $a[1]$ or $a[x]$ are just expressions. However, array assignment must be treated separately from variable assignment.

An array assignment is of the form:

$$a[NE] := E$$

in which a is an array variable, NE is a numerical expression, and E is an expression. For example, $a[x] := a[x+1]$ assigns the value of the $(x+1)$-th element of a to the index x.

A first naïve attempt at an array assignment would look similar to the assignment axiom:

$$\{P[E/a[NE]]\}\ a[NE] := E\ \{P\}$$

That is, replace all occurrences of $a[NE]$ with E. However, this does not take into account that NE may be a variable, and there may be other variables with the same value as NE that can reference a in the precondition or postcondition. This is known as the *aliasing* problem.

Example 8.12
Consider the following counterexample for the rule above:

$$\{x = y \wedge a[x] = 0\}\ a[y] := 5\ \{a[x] = 0\}$$

Because $a[x]$ is not referenced in the program, we must assume that its value has not changed. However, this is not the case, because $x = y$, and therefore $a[x]$ should be 5 after the assignment.

□

The solution for this is to actually treat array assignment as an ordinary assignment statement, except that we consider the array in its entirety. That is, we treat the assignment:

$$a[NE] := E$$

as the assignment:

$$a := a\{NE \rightarrow E\}$$

in which $a\{NE \rightarrow E\}$ represents the array that is identical to a, except with the NE-th component has the value E — the same as array updates in the SPARK annotation language, discussed in Section 7.3.1.

Therefore, the array assignment rule is simply:

$$\{P[a\{NE \rightarrow E\}/a]\}\ a[NE] := E\ \{P\},$$

which is the same as the assignment rule.

To reason fully about arrays, we need to provide axioms for the notation $a\{NE \rightarrow E\}$. The following two axioms are the *array* axioms in Hoare logic:

$$a\{NE \rightarrow E\}[NE] = E$$
$$a\{NE_1 \rightarrow E\}[NE_2] = a[NE_2] \qquad \text{where } NE_1 \neq NE_2$$

That is, the new value at the overridden index is E, and the values remain the same at all other indices.

Example 8.13
As a demonstration of applying this rule, we use the counterexample of the naïve attempt, which also tests out the validity of the new rule. Using the naïve rule, the following Hoare triple is valid:

$$\{x = y \wedge a[x] = 0\}\ a[y] := 5\ \{a[x] = 0\}$$

Using the updated array assignment axiom, we prove that the program does *not* achieve the postcondition $a[x] = 0$. Applying the array assignment axiom and working backwards, we get the triple:

$$\{a\{y \to 5\}[x] = 0\}\ a[y] := 5\ \{a[x] = 0\}$$

Now, we need to prove that the original precondition applies the weakest precondition above:

$$x = y \wedge a[x] = 0 \ \Rightarrow\ a\{y \to 5\}[x] = 0$$

From the array axioms, we can infer that $a[y] = 5$, however, from the premise ($x = y \wedge a[x] = 0$), we can also infer that $a[y] = 0$, leaving us with:

$$x = y \wedge a[x] = 0 \ \Rightarrow\ a\{y \to 5\}[x] = 0 \wedge a[y] = 0 \wedge a[y] = 5$$

which is a contradiction. The proof fails, so the program does not establish its postcondition.

□

8.4.9 Procedure call rule

In the first instance, we will consider only calls to non-recursive procedures — either direct or indirect. The rule for this is straightforward, but comes with some restrictions.

The procedure call rule is:

$$\frac{\{P\}\ S\ \{Q\}}{\{P[E_1/v_1, \ldots, E_n/v_n]\}\ p(E_1, \ldots, E_n)\ \{Q[E_1/v_1, \ldots, E_n/v_n]\}} \qquad \text{where } p(v_1, \ldots, v_n) \mathrel{\widehat{=}} S$$

This states that, if the body, S, of a procedure p, establishes a postcondition, Q, under the precondition P, then a call to that procedure will establish the same postcondition, but with all parameters replaced by the supplied arguments.

To prove a Hoare triple containing a procedure call, we first prove the correctness of the procedure call in terms of its formal parameters, and then replace the formal parameters with the actual arguments in the precondition and postcondition.

Unfortunately, this rule is not valid for all procedure calls. If our language only had procedures with no parameters, the simplified version of this rule (with no renaming) would be sufficient. However, the rule is not valid for cases in which the same variable is used as an argument to the procedure. The following example, taken from Hoare [5], demonstrates a counter example to the rule.

Example 8.14

Assume:	$p(x, y) \mathrel{\widehat{=}} x := y + 1$		(1)
	$\{y + 1 = y + 1\}\ x := y + 1\ \{x = y + 1\}$	[Assignment axiom]	(2)
	$\mathit{true} \ \Rightarrow\ y + 1 = y + 1$	[Logical theorem]	(3)
From 2, 3:	$\{\mathit{true}\}\ x := y + 1\ \{x = y + 1\}$	[Consequence rule]	(4)
From 1, 4:	$\{\mathit{true}\}\ p(x, y)\ \{x = y + 1\}$	[Procedure rule]	(5)
From 5:	$\{\mathit{true}\}\ p(z, z)\ \{z = z + 1\}$	[Procedure rule]	(6)

Clearly, the final line (6) contains a contradiction because $z = z + 1$ is not satisfiable, showing that the procedure rule is not sound for all programs.

□

Parameters and arguments

In many languages, procedures take two types of arguments: *value* variables, which are values passed to the procedure that cannot be changed; and *value-result* variables, which are *variables* passed to the procedure that can be changed. In Ada, these are labelled using the keywords `in` and `out`.

From the counterexample above, it is clear that certain combinations of parameters and arguments can cause problems in some programs with respect to Hoare logic. To ensure that the procedure call rule is valid, the set of programs accepted by Hoare logic is restricted to the following:

1. The number of arguments in a procedure call must be equal to the number of parameters in the procedure.
2. The formal parameter names must be distinct from each other.
3. The arguments for value-result variables must be distinct from each other. It is this rule that prevents the problem outlined in the counterexample above — the procedure call would be illegal.
4. Value parameters cannot be changed in the procedure body.
5. No recursion.

In languages such as SPARK, these rules apply, and the SPARK toolset will raise errors when typechecking programs that violate the above rules.

Example 8.15
As an example, consider the factorial program from Example 8.8. In that example, we established that the factorial program conforms to its contract.

Now, we want to prove the following *call* to the FACTORIAL procedure:

$$\{N = 10\}\ \text{FACTORIAL}(10, \text{F})\ \{F = 10!\}$$

We have already proved the triple $\{N \geq 0\}$ FACTORIAL(N, F) $\{F = N!\}$, which is the hypothesis of the procedure call rule. So, applying the procedure call rule, we substitute in 10 for N to get:

$$\{10 \geq 0\}\ \text{FACTORIAL}(10, \text{F})\ \{F = 10!\}$$

The postcondition is achieved, but not the precondition. We use the consequence rule to show that the precondition of our program implies the weakest precondition above: $N = 10 \Rightarrow 10 \geq 0$. This is trivially true because $10 \geq 0$ is true.

□

Recursive procedures — the Induction rule

Reasoning about recursive procedures is not possible using the procedure call rule above. This is because the procedure call rule "unfolds" a procedure call and proves its body. If the procedure is recursive, then the body contains the same procedure call, so the unfolding will continue *ad infinitum*.

Reasoning about recursive procedures is not so difficult, although it is not straightforward to automate. As it turns out, the solution is simple yet extremely powerful: in the proof of the procedure body, we assume that recursive calls to the procedure already hold. Formally, this can be stated:

$$\frac{\{P\}\ p()\ \{Q\} \vdash \{P\}\ S\ \{Q\}}{\{P\}\ p()\ \{Q\}} \qquad \text{where } p() \mathrel{\widehat{=}} S$$

in which we have omitted parameters to improve readability. In the above, the symbol $\vdash$ means "entails" or "proves"; so $P \vdash Q$ means that Q is provable if we assume that P holds.

This rule is called the *induction* rule. It says: if we can prove that the procedure body satisfies the precondition and postcondition under assumption that all recursive calls to p also do so, then p satisfies the postcondition under the precondition.

How can this be?! Assuming the conclusion as part of the program body proof seems strange, but the following example provides a concrete case on which it holds.

Example 8.16
The following program shows a recursive procedure:

procedure p(**in** x, **out** y)
    if $x \leq 5$ **then**
        $y := 1$
    else
        $x := x - 1$;
        $p(x, y)$
    end if
end procedure

It is clear to reason that if $x > 5$, it will be decremented until it reaches 5, and the program will terminate. The program will always terminate with $y = 1$, so the fact that the second branch contains a recursive call is inconsequential: all that matters are the terminating states.

□

Recursive procedure calls can never be terminating statements themselves: the statements within the call must be executed. The terminating statements in the procedure are the only statements with end states, we need to only prove that these statements establish the postcondition. For the recursive statements, we assume that these are true for the purposes of establishing the proof.

This is analogous to the use of induction in logic: we prove the base case, and then the inductive case. In the recursive procedure rule above, we offer only the inductive step, because the base case corresponds to the zero-th unfolding of the procedure, which is achieved by default: the zero-th unfolding is not executing the procedure at all, which results in a terminating state of **false**, and **false** implies any postcondition.

8.4.10 Other rules

Many extensions of the basic Hoare logic have been proposed, including extensions that consider pointers and references, concurrency, goto statements, object-oriented programs, and real-time programs. However, the basic Hoare logic here is sufficient to reason about a wide range of programs – including all programs written in the SPARK language.

8.5 Mechanising Hoare logic

It is clear that proofs about Hoare triple are long and tedious, even for small programs like the factorial program, with many small details that need to be done correctly. There are many opportunities to make mistakes, which may lead us to "prove" a correct program is incorrect, or even worse, that an incorrect program in correct.

However, with the exception of finding loop invariants, many of the steps can be mechanised, improving both the speed at which we can prove programs, and the correctness of the proofs themselves.

The procedure for mechanising proofs is straightforward, and it mimics that of how we tackle the proof manually. If we consider that the input to the process is a Hoare triple, then the following steps are undertaken:

1. Annotate the program with the necessary predicates; for example, the annotations between sequentially composed statements, and the loop invariants. Many of these can be done automatically using the idea of weakest preconditions, but loop invariants are difficult. This is an active research area, even here at the University of Melbourne in the Department of Computing and Information Systems.
2. Use a program to generate *verification conditions*. The verification conditions can be generated automatically by applying the Hoare logic rules. The parts of the Hoare logic proofs that remain are the verification conditions; for example, that a program precondition implies the weakest precondition.
3. Prove as many verifications conditions correct as possible using a theorem prover.
4. Leave the remainder of the verification conditions for the user, who can try to prove manually, or more likely, with a proof assistant tool.
5. If all verification conditions can be proved, the program is correct. If at least one is proved incorrect, the program is incorrect. If some cannot be proved or disproved, we cannot be certain of the program's correctness.

In workshops, we will look at the SPARK tools, which use logics quite similar to Hoare logic and processes such as the one described above to prove that SPARK programs conform to their contracts.

8.6 Additional reading

1. Hoare's original 1969 paper "An axiomatic basis for computer programming" [4] is a truly brilliant article that I highly recommend reading. It contains only a subset of the rules presented in this chapter.

A PDF copy is available on the LMS and in the subject repository.

2. Mike Gordon from Cambridge University has written an excellent article on Hoare logic titled "Background reading on Hoare Logic", which provides additional material and examples. This can be downloaded from `http://www.cl.cam.ac.uk/~mjcg/Teaching/2011/Hoare/Notes/Notes.pdf`

3. Dijkstra presented a formal reasoning framework called *predicate transformer semantics* [2]. The semantics are a re-formulation of Hoare logic (and are accessible to anyone that understands Hoare logic) that are essentially an algorithm to reduce a proof about Hoare logic to a formula in first-order logic.

Bibliography

[1] J. Bentley. *Programming pearls*. Addison-Wesley Professional, 2000.

[2] E. W. Dijkstra. Guarded commands, nondeterminacy, and formal derivation of programs. In *Programming Methodology*, pages 166–175. Springer, 1978.

[3] E. W. Dijkstra. "Why is software so expensive?": An explanation to the hardware designer. In *Selected Writings on Computing: A Personal Perspective*, pages 338–348. Springer, 1982.

[4] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 1969.

[5] C. A. R. Hoare. Procedures and parameters: An axiomatic approach. In *Symposium on Semantics of Algorithmic Languages*, pages 102–116. Springer, 1971.

[6] Donald E Knuth. *The Art of Computer Programming, 3: Sorting and Searching*. Addison Wesley, Reading, MA, 2nd edition, 1998.