Hoare Logic

James Koppel

January 2, 2023

1 Introduction

Hoare logic lies at the foundation of formally verifying imperative programs. Although verification is rarely performed outside of research settings, Hoare logic is still interesting from a software-design perspective because it provides a simple way to make the assumptions and guarantees of programs (the "logical layer") more concrete, and hence help build intuition for the role of the assumptions and guarantees of programs in software design.

2 Assertions

The big leap of working with Hoare triples is to go from just seeing lines of code, to seeing everything that must be true between each line of code. Let's use this piece of code as an example:

```
x = 10;
y = 42;
z = x + y;
```

The central object of study in Hoare logic are called *assertions*, logical facts that must be true at a certain point in the program. We typically write assertions in curly braces. So, we can write assertions before and after each line above as follows:

```
{ true }
x = 10;
{ x = 10 }
y = 42;
{ x = 10, y = 42 }
z = x + y;
{ x = 10, y = 42, z = 52 }
```

Here are a couple things to point out about the assertions:

First, they are written in the language of logic, rather than the language of programming. For instance, the = within the assertions is mathematical equality, not the assignment operator in programming. We can replace any

assertion with anything that's logically equivalent. For instance, for the last assertion, we could also write $x = 10, z = 52, y = z - x$.

Second, some terminology: The assertion at the top is the *precondition*, the facts that must be true before the code runs. This guarantees that, after the code runs, the last assertion, or *postcondition*, must be true. The assertions in the middle are intermediate facts.

Third, you may be wondering what it means for the precondition to be "true." It means the code may run only if "true is true," which is a tautology. This means the code may be run in any context.

Conversely, a precondition of "false" means that the following code may never run. So the following program and assertions are valid:

```
{ false }
x = 1;
{ x = 2 }
```

This can be read "If false is true, then, after running the code `x = 1`, it will be true that $x = 2$." The ability to prove any postcondition from a precondition of "false" is important when analyzing if statements where the condition is always true (or false).

Just from these examples, we can already learn some interesting things. Notice how, even though each line is simple, the assertions grow in complexity, reflecting how the current state of the program grows in complexity.

However, the assertions given above are not the only ones that can be given for that example. As one alternative, suppose there is a fourth variable, w, which must be set to 27 before this code runs. Then we can alter the assertions to "remember" the value of w.

```
{ w = 27 }
x = 10;
{ x = 10, w = 27 }
y = 42;
{ x = 10, y = 42, w = 27 }
z = x + y;
{ x = 10, y = 42, z = 52, w = 27 }
```

Conversely, suppose that, in the code we intend to write after this example code, it is only necessary that $z > 1$, and x and y will not be used. Then we can "forget" about x and y, and about the exact value of z.

```
{ true }
x = 10;
{ x > 1 }
y = 42;
{ x > 1, y > 1 }
z = x + y;
{ z > 1 }
```

Notice how, this time, the assertions do not grow in complexity. This corresponds to a programmer needing to remember less about this code in order to read whatever comes after it.

Exercise 1 *Conceptual question: In the latter example, how could we change the code without altering the final postcondition? How does the "forgetting" of assertions correspond to a form of modularity?*

Clearly, there are infinitely many preconditions and postconditions that can be given to a piece of code. In the first example, we could change the precondition "true" to $p = 80$, for instance, but then forget about p and keep the rest the same. This is clearly extraneous. "true" is the *weakest precondition* for the postcondition of $x = 10, y = 42, z = 52$, in that all other preconditions imply it. Conversely, starting with a precondition of "true," "$x = 10, y = 42, z = 52$" is the strongest postcondition, in that it implies all other possible postconditions such as "$z > 1$". So, for a given precondition, there is a canonical postcondition, and vice versa. But note that, if we only have a program, there is not both a canonical precondition and postcondition. Adding w to the precondition changes the strongest postcondition, and adding w to the postcondition changes the weakest precondition.

So, in order to get a canonical weakest precondition, we need both the code and a postcondition. In order to get a canonical strongest postcondition, we need both the code and a precondition. We cannot get something logical without already having something logical. These assertions are central examples of Level 3 "logical" constructs, while the code is the central example of a Level 2 construct. So, this is another example of a central theme of this course: while it can help us infer some things, having Level 2 information alone is never enough to reason about the design of a program.

Later in this course, we'll see some examples of how to mechanically transform the precondition and postcondition into code, showing how the logic is more fundamental than the code, in some sense.

3 Hoare Triples

In the last section, we showed lines of code, with assertions between each one. We can think about all the lines of code as being combined into a single statement, also called a *command*, and discarding the intermediate assertions. Then what's left is the precondition, command, and postcondition. This triple of a precondition, command, and postcondition is called a *Hoare triple*. We write them like this:

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

A Hoare triple $\{P\}S\{Q\}$ should be read "if P is true, then, after S executes, Q will be true."

In the remainder of the worksheet, we will give the formal rules for inferring Hoare triples for a simple imperative programming language with assignments.This is sufficient to give some intuition for treating the logical layer of a program as a concrete entity. Loops and conditionals are explained in the appendix. Hoare logics have also been developed for many more complicated language features, such as concurrency and heap-allocation.

4 Rules

Like other logical systems, Hoare Logic is defined by a set of *inference rules*. Inference rules are given in the form

$$\frac{premise1 \quad premise2 \quad \dots \quad premiseN}{conclusion}\ rule - name$$

This is read to say: if the premises of the rule hold, then one can deduce that the conclusions hold. For example, here's an inference rule for "if the glove doesn't fit, you must acquit":

$$\frac{\text{the glove doesn't fit}}{\text{you must acquit}}\ glove - acquit$$

These premises are often in turn proven with other rules. Proofs thus take the form of "derivation trees," like this:

$$\frac{\dfrac{\text{hand size} = x \quad \text{glove size} = y \quad x > y}{\text{the glove doesn't fit}}}{\text{you must acquit}}$$

Inference rule notation can take some getting used to. If you want some more exposure, a fun way of getting it is from Logitext (`http://logitext.mit.edu/tutorial`), an interaction which explains another formal system, the sequent calculus.

4.1 Substitutions

In the next subsection, we'll explain how to infer Hoare triples for assignments, perhaps the most fundamental rule. But first, we'll have to explain substitution.

Substitutions are a generalization of the notion of "plugging in for a variable" in mathematics. So, for instance, in the expression $E = \frac{1}{2}at^2 + vt + x$, which you may recognize as the displacement of an object moving at constant acceleration,

we can "plug in" 5 for x, and get $\frac{1}{2}at^2 + vt + 5$. We write this in the notation $[5/x]E$, read as "substitute 5 for x in E," or just "5 for x in E."

Here are some more example substitutions:

$$\begin{aligned} [x+1/x](x*x-y) &= (x+1)*(x+1)-y \\ [y/x](z := x) &= (z := y) \\ [y/x](z := x; w = z+z) &= (z := y; w = z+z) \end{aligned}$$

4.2 Assignment

Now that we've explained substitutions, the rule for assignments is easy to state:

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

For instance, here's an example of using this rule, using the fact that $[y/x](x = 5) = (y = 5)$:

$$\frac{}{\{y = 5\}\ x := y\ \{x = 5\}}\ assign$$

Here are a couple more applications:

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

$$\frac{}{\{(x+1)*(x+1) - y = 25\}\ x := x + 1\ \{x*x - y = 25\}}\ assign$$

Note that it's simple to compute the precondition given the command and the postcondition: we just do a substitution. But, to compute the postcondition from the command and the precondition, it's much harder: we'd need do compute an "unsubstitution." So, the version of the rule we've given is primarily appropriate for computing preconditions from postconditions.

4.3 Sequence

The SEQUENCE rule simply chains two Hoare triples together, checking that the postcondition of the first matches the precondition of the second.

$$\frac{\{P\}S\{Q\} \quad \{Q\}T\{R\}}{\{P\}S;T\{R\}}\ seq$$

Here is an example of using it in conjuction with the ASSIGNMENT rule to verify the program $x := y; x := x + 1$.

```
1 | {                    }
2 | b := 2 - a
3 | {                    }
4 | c := b * 2
5 | {                    }
6 | d := c + 1
7 | {d = 5}
```

Figure 1: Code listing for Exercise 2

$$\dfrac{\dfrac{}{\{y+1>0\}\ x := y\ \{x+1>0\}}\ assign \qquad \dfrac{}{\{x+1>0\}\ x := x+1\ \{x>0\}}\ assign}{\{y+1>0\} x := y;\ x := x+1 \{x>0\}}\ seq$$

In doing this proof, we had to come up with an assertion which is true between each statement of the program. This rule motivates the notation we used in the beginning of this document, where we simply wrote an assertion between each line of the program.

```
{ y + 1 > 0 }
x := y
{ x + 1 > 0 }
x := x + 1
{ x > 0 }
```

Exercise 2 *Look at the code in Figure 1. Fill in all assertions.*

Exercise 3 *We haven't given you any rules other than the ones for straight-line code. But we can still write some interesting programs if we use integer division*[1]*, which we introduce in this exercise. In this exercise, assume all variables are integers, and all division is integer division (rounds toward 0).*

Your task: Fill in the assertions of Figure 2.

You may have some intuition about this code, and it will be tempting to use it. But this must be resisted, because it can lead you to the wrong answer. In particular, intuitions you may have learned about division of real numbers may not carry over. Terry Tao calls this the "pre-rigorous" stage. Turn your intuition off and follow the rules mechanically (rigorous stage) so that, in time, you will build a new intuition which matches the rules (post-rigorous stage).

***Note on notation**: /\ is ASCII for ∧, which denotes the logical "and." => is ASCII for ⇒ and is read "implies." A ⇒ B means "if A is true, then B is true."*

[1]For the student with a more advanced manthematical background, the reason integer division allows more interesting programs is that it is a non-linear operator.

```
1 | { x > 0                                                    }
2 | y := (x / 2) * 2
3 | {                                                          }
4 | z := x - y
5 | {                                                          }
6 | a := z * 5 + (1 - z) * 12
7 | { ((x is odd) => a = 5)  /\ ((x is even) => a = 12) }
```

Figure 2: Code listing for exercise 3

Exercise 4 *This exercise is much harder, but is meant to illustrate two important points in software design: that conditionals are not simply "if-statements" and can appear even in straight-line code, and that Hoare logic gives an objective measure of complexity which can motivate refactoring. We will do part of this example now, and revisit in again in the next section.*

In this exercise, as before, assume all variables are integers, and all division is integer division (rounds toward 0). ***Also assume that x/0 == 0 for all x.***

1. *Look at the code in Figure 3. Fill in the assertions between each line. We have done the last and first ones for you.*

2. *In what sense does this code contain a conditional?*

Hint 1*: It will help if you start at the end and work backwards.*
Hint 2*: Think carefully about the expression* $(2-(a+1)/a)/2$ *and what it does. Try it on different values, and remember that division rounds towards 0, and* $x/0 == 0$.

4.4 Precondition strengthening, postcondition weakening

As we discussed earlier, there can be many preconditions for the same statement and postcondition. For example, for the statement $x := x+1$ with postcondition $x > 0$, one possible precondition is $x > -1$, but another one is $x > -1 \wedge x < 10$. We say that $x > -1$ is the *weakest precondition*, as it is implied by any other valid precondition. Convesely, $x > 0$ is the *strongest postcondition*.

The rules we gave above only allow for a single precondition for each statement and postcondition. To get others, we need to use the CONSEQUENCE rule, which can strengthen preconditions and weaken postconditions.

$$\frac{P' \Rightarrow P \quad \{P\}S\{Q\} \quad Q \Rightarrow Q'}{\{P'\}S\{Q'\}}\ \textit{consequence}$$

Here's an example application:

```
1  | { true }
2  | d := (2-(a+1)/a)/2;
3  | {                                                          }
4  | m := d * 2 + (1-d) * 3;
5  | {                                                          }
6  | x := b * 2;
7  | {                                                          }
8  | x := x * 2;
9  | {                                                          }
10 | x := m * x;
11 | {                                                          }
12 | x := x + 1;
13 | { ((a <= 0) => x = 8*b+1) /\ ((a > 0) => x = 12*b+1)   }
```

Figure 3: Code listing for Exercise 5

$$\frac{(x+1>11)\Rightarrow(x+1>10)\quad \{x+1>10\}x:=x+1\{x>10\}\quad (x>10)\Rightarrow(x>9)}{\{x+1>11\}x:=x+1\{x>9\}}\ \textit{consequence}$$

When we express Hoare logic as assertions between each line of a program, the consequence rule lets us modify an assertion using purely logical reasoning, without any intervening code. So a use of the consequence rule involves two assertions in a row, where the latter/former is a weakening/strengthening of the former/latter. For example:

```
{ true }
x = 10;
{ x = 10 }
{ x > 1 }
y = 42;
{ x > 1, y = 42 }
{ x > 1, y > 1 }
z = x + y;
{ x > 1, y > 1, z = x+y}
{ z > 1 }
```

Suppose a program consists of two subprograms, A followed by B. **If the strongest postcondition of A is stronger than the weakest precondition of B, the program has modularity**, because there are many ways to change A without changing B.

Exercise 5 *Revisit the code in Figure 3. Notice how it required you to track a lot of information between some of the intermediate lines. How might you be able to reorder the statements to make the code simpler? The precondition and postcondition of the code as a whole should remain unchanged.*

***Note**: Doing so can also involve a very subtle use of the consequence rule. After you finish, check the official solution for how.*

A Loop Invariants (Optional)

The challenge of verifying with loops is that a single inference must capture the behavior of the loop, no matter how many times the loop runs. The following rule does the trick:

$$\frac{\{P \wedge B\} S \{P\}}{\{P\} \text{ while } B \text{ do } S \{\neg B \wedge P\}} \textit{ while}$$

This rule is a bit different than the others in that P appears on both sides of the premise. Typically, this P will need to be chosen using the CONSEQUENCE rule, and this rule gives no guidance how to do so. While using the ASSIGN or SEQUENCE rules can be quite mechanical, coming up with a loop invariant typically requires more creativity, and is among the hardest tasks in verifying programs.

P is called a *loop invariant*, because it must be true at the start of each iteration of the loop, as well as after the loop runs.

As an example, in this factorial program, the loop has invariant $fac = i! \wedge i <= n$. At the end of the loop, we have $i = n$, and hence this is sufficient to prove that $fac = n!$, so the program correctly computes the factorial, as desired. Note that it is not sufficient to merely have $fac = i!$; then we would not be able to prove that $i = n$ at the end of the loop.

```
i := 0
fac := 1
while i < n do
  i := i + 1
  fac := fac * i
end
{ fac = n! }
```

If you are familiar with proofs by induction, you may recognize the similarities between a loop invariant and an induction hypothesis. For a more thorough explanation of loop invariants, check out the Wikipedia articles on loop invariants and Hoare logic. We are also collecting resources to help students understand/practice loop invariants at `https://docs.google.com/document/d/1nUngjbSlyJQfMfvAaxKfn3aF7IeHi3va5yENWeqlo7k/edit`.

As a sidenote, the rule we presented only gives *partial correctness*, meaning it does not prove termination. A loop `while (true)` would have a postcondition of *false*, meaning it never terminates. Consult Wikipedia if you want to learn about termination proofs.

Exercise 6 ***Optional**: Consider the following code for a sequential search procedure:*

```
{ true }
i := 0
{                                   }
while i < n && arr[i] != val do
  {                                 }
  i := i + 1
  {                                 }
end
{ arr[i] == val || (forall j, (j >= 0 && j < n) => arr[j] != val) }
```

1. *Prove this sequential search procedure correct by choosing a proper loop invariant.*
2. *State the loop invariant you chose*

B Conditionals

$$\frac{\{P\}S\{R\} \quad \{Q\}T\{R\}}{\{(B \Rightarrow P) \wedge (\neg B \Rightarrow Q)\} \text{ if } B \text{ then } S \text{ else } T\{R\}} \; if$$

Note that the IF rule doubles the size of the precondition, so that a chain of them can cause exponential blowup in complexity. This corresponds to how a chain of if statements can have exponentially many paths. Being able to forget which path was taken using the CONSEQUENCE rule is paramount to making programs with conditionals tractable to reason about.