Hoare logic

From Wikipedia, the free encyclopedia

Hoare logic (also known as **Floyd–Hoare logic** or **Hoare rules**) is a formal system with a set of logical rules for reasoning rigorously about the correctness of computer programs. It was proposed in 1969 by the British computer scientist and logician C. A. R. Hoare, and subsequently refined by Hoare and other researchers.[1] The original ideas were seeded by the work of Robert Floyd, who had published a similar system[2] for flowcharts.

Contents

Hoare triple

The central feature of **Hoare logic** is the **Hoare triple**. A triple describes how the execution of a piece of code changes the state of the computation. A Hoare triple is of the form

$$\{P\}\ C\ \{Q\}$$

where P and Q are *assertions* and C is a *command*.[note 1] P is named the *precondition* and Q the *postcondition*: when the precondition is met, executing the command establishes the postcondition. Assertions are formulae in predicate logic.

Hoare logic provides axioms and inference rules for all the constructs of a simple imperative programming language. In addition to the rules for the simple language in Hoare's original paper, rules for other language constructs have been developed since then by Hoare and many other researchers. There are rules for concurrency, procedures, jumps, and pointers.

Partial and total correctness

Using standard Hoare logic, only partial correctness can be proven, while termination needs to be proved separately. Thus the intuitive reading of a Hoare triple is: Whenever P holds of the state before the execution of C, then Q will hold afterwards, or C does not terminate. In the latter case, there is no "after", so Q can be any statement at all. Indeed, one can choose Q to be false to express that C does not terminate.

Total correctness can also be proven with an extended version of the While rule.

In his 1969 paper, Hoare used a narrower notion of termination which also entailed absence of any run-time errors: "*Failure to terminate may be due to an infinite loop; or it may be due to violation of an implementation-defined limit, for example, the range of numeric operands, the size of storage, or an operating system time limit.*"[3]

Rules

Empty statement axiom schema

The empty statement rule asserts that the **skip** statement does not change the state of the program, thus whatever holds true before **skip** also holds true afterwards.[note 2]

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

Assignment axiom schema

The assignment axiom states that after the assignment any predicate holds for the variable that was previously true for the right-hand side of the assignment. Formally, let P be an assertion in which the variable x is free. Then:

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

where $P[E/x]$ denotes the assertion P in which each free occurrence of x has been replaced by the expression E.

The assignment axiom scheme means that the truth of $P[E/x]$ is equivalent to the after-assignment truth of P. Thus were $P[E/x]$ true prior to the assignment, by the assignment axiom, then P would be true subsequent to which. Conversely, were $P[E/x]$ false (i.e. $\neg P[E/x]$ true) prior to the assignment statement, P must then be false afterwards.

Examples of valid triples include:

- $\{\ x+1 = 43\ \}\quad y := x + 1\quad \{\ y = 43\ \}$
- $\{\ x + 1 \leq N\ \}\quad x := x + 1\quad \{\ x \leq N\ \}$

The assignment axiom scheme is equivalent to saying that to find the precondition, first take the post-condition and replace all occurrences of the left-hand side of the assignment with the right-hand side of the assignment. Be careful not to try to do this backwards by following this *incorrect* way of thinking: $\{P\}\ x{:=}E\ \{P[E/x]\}$; this rule leads to nonsensical examples like:

$$\{\ x = 5\ \}\quad x := 3\quad \{\ 3 = 5\ \}$$

Another *incorrect* rule looking tempting at first glance is $\{P\}\ x:=E\ \{P \text{ and } x=E\}$; it leads to nonsensical examples like:

$$\{\ x = 5\ \}\quad x := x + 1\quad \{\ x = 5 \text{ and } x = x + 1\ \}$$

While a given postcondition P uniquely determines the precondition $P[E/x]$, the converse is not true. For example:

- $\{\ 0 \le y*y \wedge y*y \le 9\ \}\quad x := y * y\quad \{\ 0 \le x \wedge x \le 9\ \}$,
- $\{\ 0 \le y*y \wedge y*y \le 9\ \}\quad x := y * y\quad \{\ 0 \le x \wedge y*y \le 9\ \}$,
- $\{\ 0 \le y*y \wedge y*y \le 9\ \}\quad x := y * y\quad \{\ 0 \le y*y \wedge x \le 9\ \}$, and
- $\{\ 0 \le y*y \wedge y*y \le 9\ \}\quad x := y * y\quad \{\ 0 \le y*y \wedge y*y \le 9\ \}$

are valid instances of the assignment axiom scheme.

The assignment axiom proposed by Hoare *does not apply* when more than one name may refer to the same stored value. For example,

$$\{\ y = 3\ \}\quad x := 2\quad \{\ y = 3\ \}$$

is wrong if x and y refer to the same variable (aliasing), although it is a proper instance of the assignment axiom scheme (with both $\{P\}$ and $\{P[2/x]\}$ being $\{y=3\}$).

Rule of composition

Hoare's rule of composition applies to sequentially executed programs S and T, where S executes prior to T and is written $S;T$ (Q is called the *midcondition*):[4]

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

For example, consider the following two instances of the assignment axiom:

$$\{\ x + 1 = 43\ \}\quad y := x + 1\quad \{\ y = 43\ \}$$

and

$$\{\ y = 43\ \}\quad z := y\quad \{\ z = 43\ \}$$

By the sequencing rule, one concludes:

$$\{\ x + 1 = 43\ \}\quad y := x + 1; z := y\quad \{\ z = 43\ \}$$

Another example is shown in the right box.

Verifying swap-code without auxiliary variables

The three statements below (line 2, 4, 6) exchange the values of the variables a and b, without needing an auxiliary variable. In the verification proof, the initial value of a and b is denoted by the constant A and B, respectively. The proof is best read backwards, starting from line 7; for example, line 5 is obtained from line 7 by replacing a (target expression in line 6) by $a–b$ (source expression in line 6). Some arithmetical simplifications are used tacitly, viz. $a–(a–b) = b$ (line 5→3), and $a+b–b = a$ (line 3→1).

Nr	Code	Assertions
1:		$\{\ a = A \wedge b = B\ \}$
2:	$a := a + b;$	
3:		$\{\ a - b = A \wedge b = B\ \}$
4:	$b := a - b;$	
5:		$\{\ b = A \wedge a - b = B\ \}$
6:	$a := a - b$	
7:		$\{\ b = A \wedge a = B\ \}$

Conditional rule

$$\frac{\{B \wedge P\}\ S\ \{Q\}\quad ,\quad \{\neg B \wedge P\ \}\ T\ \{Q\}}{\{P\}\ \textbf{if}\ B\ \textbf{then}\ S\ \textbf{else}\ T\ \textbf{endif}\ \{Q\}}$$

The conditional rule states that a postcondition Q common to **then** and **else** part is also a postcondition of the whole **if**...**endif** statement. In the **then** and the **else** part, the unnegated and negated condition B can be added to the precondition P, respectively. The condition, B, must not have side effects. An example is given in the next section.

This rule was not contained in Hoare's original publication.[1] However, since a statement

if B **then** S **else** T **endif**

has the same effect as a one-time loop construct

bool b:=**true**; **while** $B \wedge b$ **do** S; b:=**false done**; b:=**true**; **while** $\neg B \wedge b$ **do** T; b:=**false done**

the conditional rule can be derived from the other Hoare rules. In a similar way, rules for other derived program constructs, like **for** loop, **do**...**until** loop, **switch**, **break**, **continue** can be reduced by program transformation to the rules from Hoare's original paper.

Consequence rule

$$\frac{P_1 \rightarrow P_2 \quad , \quad \{P_2\}\ S\ \{Q_2\} \quad , \quad Q_2 \rightarrow Q_1}{\{P_1\}\ \mathrm{S}\ \{Q_1\}}$$

This rule allows to strengthen the precondition and/or to weaken the postcondition. It is used e.g. to achieve literally identical postconditions for the **then** and the **else** part.

For example, a proof of

$\{0 \le \mathrm{x} \le 15\}$ **if** $x < 15$ **then** $x := x + 1$ **else** $x := 0$ **endif** $\{0 \le x \le 15\}$

needs to apply the conditional rule, which in turn requires to prove

$\{0 \le x \le 15 \wedge x < 15\}$ x:=x+1 $\{0 \le x \le 15\}$, or simplified
$\{0 \le x < 15\}$ x:=x+1 $\{0 \le x \le 15\}$

for the **then** part, and

$\{0 \le x \le 15 \wedge x \ge 15\}$ x:=0 $\{0 \le x \le 15\}$, or simplified
$\{x=15\}$ x:=0 $\{0 \le x \le 15\}$

for the **else** part.

However, the assignment rule for the **then** part requires to choose P as $0 \le x \le 15$; rule application hence yields

$\{0 \le x+1 \le 15\}$ x:=x+1 $\{0 \le x \le 15\}$, which is logically equivalent to
$\{-1 \le x < 15\}$ x:=x+1 $\{0 \le x \le 15\}$.

The consequence rule is needed to strengthen the precondition $\{-1 \le x < 15\}$ obtained from the assignment rule to $\{0 \le x < 15\}$ required for the conditional rule.

Similarly, for the **else** part, the assignment rule yields

$\{0 \le 0 \le 15\}$ x:=0 $\{0 \le x \le 15\}$, or equivalently
{**true**} x:=0 $\{0 \le x \le 15\}$,

hence the consequence rule has to be applied with P_1 and P_2 being $\{x{=}15\}$ and $\{\textbf{true}\}$, respectively, to strengthen again the precondition. Informally, the effect of the consequence rule is to "forget" that $x{=}15$ is known at the entry of the **else** part, since the assignment rule used for the **else** part doesn't need that information.

While rule

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

Here P is the loop invariant, which is to be preserved by the loop body S. After the loop is finished, this invariant P still holds, and moreover $\neg B$ must have caused the loop to end. As in the conditional rule, B must not have side effects.

For example, a proof of

$$\{\mathrm{x} \le 10\}\quad \textbf{while}\ x < 10\ \textbf{do}\ x := x + 1\ \textbf{done}\quad \{\neg x < 10 \wedge \mathrm{x} \le 10\}$$

by the while rule requires to prove

$$\{x \le 10 \wedge x < 10\}\quad x := x + 1\quad \{x \le 10\},\quad \text{or simplified}$$
$$\{x < 10\}\quad x := x + 1\quad \{x \le 10\},$$

which is easily obtained by the assignment rule. Finally, the postcondition $\{\neg x{<}10 \wedge x{\le}10\}$ can be simplified to $\{x{=}10\}$.

For another example, the while rule can be used to formally verify the following strange program to compute the exact square root x of an arbitrary number a - even if x is an integer variable and a is not a square number:

$$\{\textbf{true}\}\quad \textbf{while}\ x{*}x \neq a\ \textbf{do skip done}\quad \{x * x = a \wedge \textbf{true}\}$$

After applying the while rule with P being **true**, it remains to prove

$$\{\textbf{true} \wedge x{*}x \neq a\}\quad \textbf{skip}\quad \{\textbf{true}\},$$

which follows from the skip rule and the consequence rule.

In fact, the strange program is *partially* correct: if it happened to terminate, it is certain that x must have contained (by chance) the value of a 's square root. In all other cases, it will not terminate; therefore it is not *totally* correct.

While rule for total correctness

If the above ordinary while rule is replaced by the following one, the Hoare calculus can also be used to prove total correctness, i.e. termination[note 3] as well as partial correctness. Commonly, square brackets are used here instead of curly braces to indicate the different notion of program correctness.

$$\frac{< \text{ is a well-founded ordering on the set } D\quad ,\quad [P \wedge B \wedge t \in D \wedge t = z]\quad S\quad [P \wedge t \in D \wedge t < z\ ]}{[P \wedge t \in D]\quad \textbf{while}\ B\ \textbf{do}\ S\ \textbf{done}\quad [\neg B \wedge P \wedge t \in D]}$$

In this rule, in addition to maintaining the loop invariant, one also proves termination by way of an expression t, called the loop variant, whose value strictly decreases with respect to a well-founded relation $<$ on some domain set D during each iteration. Since $<$ is well-founded, a strictly decreasing chain of members of D can have only finite length, so t cannot keep decreasing forever. (For example, the usual order $<$ is well-founded on positive integers $\mathbb{N}$, but neither on the integers $\mathbb{Z}$ nor on positive real numbers $\mathbb{R}^+$; all these sets are meant in the mathematical, not in the computing sense, they are all infinite in particular.)

Given the loop invariant P, the condition B must imply that t is not a minimal element of D, for otherwise the body S could not decrease t any further, i.e. the premise of the rule would be false. (This is one of various notations for total correctness.) [note 4]

Resuming the first example of the previous section, for a total-correctness proof of

$$[x \le 10] \quad \textbf{while } x < 10 \textbf{ do } x{:=}x{+}1 \textbf{ done} \quad [\neg\, x < 10 \wedge x \le 10]$$

the while rule for total correctness can be applied with e.g. D being the positive integers with the usual order, and the expression t being $10 - x$, which then in turn requires to prove

$$[x \le 10 \wedge x < 10 \wedge 10{-}x \ge 0 \wedge 10{-}x = z] \quad x{:=}x{+}1 \quad [x \le 10 \wedge 10{-}x \ge 0 \wedge 10{-}x < z]$$

Informally speaking, we have to prove that the distance $10{-}x$ decreases in every loop cycle, while it always remains non-negative; this process can go on only for a finite number of cycles.

The previous proof goal can be simplified to

$$[x < 10 \wedge 10{-}x = z] \quad x{:=}x{+}1 \quad [x \le 10 \wedge 10{-}x < z],$$

which can be proven as follows:

$[x{+}1 \le 10 \wedge 10{-}x{-}1 < z] \quad x{:=}x{+}1 \quad [x \le 10 \wedge 10{-}x < z]$ is obtained by the assignment rule, and
$[x{+}1 \le 10 \wedge 10{-}x{-}1 < z]$ can be strengthened to $[x < 10 \wedge 10{-}x = z]$ by the consequence rule.

For the second example of the previous section, of course no expression t can be found that is decreased by the empty loop body, hence termination cannot be proved.

See also

- Assertion (computing)
- Communicating sequential processes
- Design by contract
- Denotational semantics
- Dynamic logic
- Edsger W. Dijkstra
- Loop invariant
- Predicate transformer semantics
- Program verification
- Refinement calculus
- Separation logic
- Sequent calculus
- Static code analysis

Notes

1. Hoare originally wrote "$P\ \{C\}\ Q$" rather than "$\{P\}\ C\ \{Q\}$".
2. This article uses a natural deduction style notation for rules. For example, $\frac{\alpha\ ,\ \beta}{\phi}$ informally means "If both α and β hold, then also ϕ holds"; α and β are called antecedents of the rule, ϕ is called its succedent. A rule without antecedents is called an axiom, and written as $\frac{}{\phi}$.
3. "Termination" here is meant in the broader sense that computation will eventually be finished; it does **not** imply that no limit violation (e.g. zero divide) can stop the program prematurely.
4. Hoare's 1969 paper didn't provide a total correctness rule; cf. his discussion on p.579 (top left). For example Reynolds' textbook (John C. Reynolds (2009). *Theory of Programming Languages*. Cambridge University Press.), Sect.3.4, p.64 gives the following version of a total correctness rule: $\frac{P \wedge B \Rightarrow 0 \leq t \quad , \quad [P \wedge B \wedge t = z]\ S\ [P \wedge t < z]}{[P]\ \textbf{while}\ B\ \textbf{do}\ S\ \textbf{done}\ [P \wedge \neg B]}$ when z is an integer variable that doesn't occur free in P, B, S, or t, and t is an integer expression (Reynolds' variables renamed to fit with this article's settings).

References

1. Hoare, C. A. R. (October 1969). "An axiomatic basis for computer programming" (PDF). *Communications of the ACM* **12** (10): 576–580. doi:10.1145/363235.363259.
2. R. W. Floyd. "Assigning meanings to programs. (http://www.cs.virginia.edu/~weimer/2007-615/reading/FloydMeaning.pdf)" Proceedings of the American Mathematical Society Symposia on Applied Mathematics. Vol. 19, pp. 19–31. 1967.
3. p.579 upper left
4. Huth, Michael; Ryan, Mark. *Logic in Computer Science* (second ed.). CUP. p. 276. ISBN 052154310X.

Further reading

- Robert D. Tennent. *Specifying Software (http://www.cs.queensu.ca/home/specsoft/)* (a textbook that includes an introduction to Hoare logic, written in 2002) ISBN 0-521-00401-2

External links

- KeY-Hoare (http://www.key-project.org/download/hoare/) is a semi-automatic verification system built on top of the KeY theorem prover. It features a Hoare calculus for a simple while language.
- j-Algo-modul Hoare calculus (http://j-algo.binaervarianz.de/index.php?language=en) — A visualisation of the Hoare calculus in the algorithm visualisation program j-Algo

Categories: 1969 in computer science | Program logic | Static program analysis

---