Formal Correctness: An Introduction to Hoare Logic & GCL

Proving and Deriving Provably Correct Programs

Amirreza Khakpour

September, 2025

Outline

Introduction to Hoare Logic

Why Do We Need Formal Verification?

How can we be *certain* that a program works as intended?

Why Do We Need Formal Verification?

How can we be *certain* that a program works as intended?

- Testing can reveal the presence of bugs, but can it ever prove their absence?

Why Do We Need Formal Verification?

How can we be *certain* that a program works as intended?

- Testing can reveal the presence of bugs, but can it ever prove their absence?
- For mission-critical systems in avionics, medicine, or finance, we need a mathematical guarantee of correctness.

Why Do We Need Formal Verification?

How can we be *certain* that a program works as intended?

- Testing can reveal the presence of bugs, but can it ever prove their absence?
- For mission-critical systems in avionics, medicine, or finance, we need a mathematical guarantee of correctness.

The Goal

To establish a formal framework for reasoning about program correctness with the same rigor that mathematicians use to prove theorems.

The Hoare Triple: Specifying Program Behavior

The core concept of Hoare Logic is the Hoare Triple, a formal statement about a piece of code.

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

P Precondition: A logical assertion that is true *before* the command executes.

The Hoare Triple: Specifying Program Behavior

The core concept of Hoare Logic is the Hoare Triple, a formal statement about a piece of code.

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

- **P Precondition**: A logical assertion that is true *before* the command executes.
- **C Command**: The segment of code being analyzed.

The Hoare Triple: Specifying Program Behavior

The core concept of Hoare Logic is the Hoare Triple, a formal statement about a piece of code.

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

- **P** **Precondition**: A logical assertion that is true *before* the command executes.
- **C** **Command**: The segment of code being analyzed.
- **Q** **Postcondition**: A logical assertion that is true *after* the command terminates.

The Hoare Triple: Specifying Program Behavior

The core concept of Hoare Logic is the Hoare Triple, a formal statement about a piece of code.

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

- **P** **Precondition**: A logical assertion that is true *before* the command executes.
- **C** **Command**: The segment of code being analyzed.
- **Q** **Postcondition**: A logical assertion that is true *after* the command terminates.

Interpretation: Partial Correctness
"If precondition P holds and command C executes, then **upon termination**, postcondition Q will hold."

Hoare Logic: Core Rules

A Simple Imperative Language

Hoare Logic provides a set of rules for reasoning about the correctness of programs written in a simple, structured imperative language. The language consists of a few basic command types.

Basic Commands

- skip: Do nothing.
- x := E: Assignment

Control Structures

- C1; C2: Sequence
- if B then C1 else C2: Conditional
- while B do C: Iteration

The Strategy

We will introduce a proof rule for each of these command structures, allowing us to reason about any program built from them.

Language Building Blocks

The components of our simple language are defined as follows:

- **Variables** $x, y, z, \ldots$ represent program variables (e.g., integers, booleans).

Language Building Blocks

The components of our simple language are defined as follows:

- **Variables** $x, y, z, \ldots$ represent program variables (e.g., integers, booleans).
- **Expressions (E)** Arithmetic or logical expressions that evaluate to a value.
  - Examples: $x + 1$, $y * 2$, $z - 5$

Language Building Blocks

The components of our simple language are defined as follows:

- **Variables** $x, y, z, \ldots$ represent program variables (e.g., integers, booleans).
- **Expressions (E)** Arithmetic or logical expressions that evaluate to a value.
  - Examples: `x + 1`, `y * 2`, `z - 5`
- **Boolean Cond. (B)** Expressions that evaluate to true or false. These are used as guards in conditionals and loops.
  - Examples: `x > 0`, `y <= r`, `z = 10`

Language Building Blocks

The components of our simple language are defined as follows:

- **Variables** $x, y, z, \ldots$ represent program variables (e.g., integers, booleans).
- **Expressions (E)** Arithmetic or logical expressions that evaluate to a value.
  - Examples: `x + 1`, `y * 2`, `z - 5`
- **Boolean Cond. (B)** Expressions that evaluate to true or false. These are used as guards in conditionals and loops.
  - Examples: `x > 0`, `y <= r`, `z = 10`
- **Commands (C)** A statement in the language, built from the constructs on the previous slide.

The Simplest Rule: The Axiom of Assignment

This is the foundational axiom for reasoning about state changes.

Axiom of Assignment: $\{Q[x/E]\}$ `x := E` $\{Q\}$
To find the precondition, take the postcondition Q and substitute every free occurrence of ‘x‘ with the expression ‘E‘. We work *backwards*.

Assignment Example

Example (Simple Example)
Goal: Prove $\{P\}$ x := 5 $\{x = 5\}$

- Substitute ‘x‘ with ‘5‘ in the postcondition ‘x = 5‘.
- $P \equiv (5 = 5) \equiv$ **True**
- **Result:** $\{True\}$ x := 5 $\{x = 5\}$

Assignment Example

Example (Simple Example)
Goal: Prove $\{P\}$ x := 5 $\{x = 5\}$

- Substitute ‘x‘ with ‘5‘ in the postcondition ‘x = 5‘.
- $P \equiv (5 = 5) \equiv$ **True**
- **Result:** $\{True\}$ x := 5 $\{x = 5\}$

Example (More Involved Example)
Goal: Prove $\{x = k\}$ x := x + 1 $\{x = k + 1\}$

- The precondition is $(x = k + 1)[x/(x + 1)]$.
- This simplifies to $(x + 1 = k + 1) \equiv (x = k)$.
- **Result:** $\{x = k\}$ x := x + 1 $\{x = k + 1\}$

The Rule of Consequence: Strengthening and Weakening

Often, the precondition we have isn't syntactically identical to the one required by a rule. The Rule of Consequence allows us to bridge this logical gap.

Rule of Consequence

$$\frac{P \to P' \quad \{P'\}\ \mathrm{C}\ \{Q'\} \quad Q' \to Q}{\{P\}\ \mathrm{C}\ \{Q\}}$$

This rule allows us to:

1. **Strengthen the Precondition**: Use a precondition P that is logically stronger than the one required (P').
2. **Weaken the Postcondition**: Settle for a postcondition Q that is logically weaker than the one we proved (Q').

Assignment: Full Proof Example using Consequence

Goal: Prove the triple $\{x = 5\}$ `x := x * 2` $\{x > 8\}$

Assignment: Full Proof Example using Consequence

Goal: Prove the triple $\{x = 5\}$ `x := x * 2` $\{x > 8\}$

Proof Strategy
1. First, use the Assignment Rule to find the *weakest precondition* (P') that guarantees the postcondition $\{x > 8\}$. 2. Then, use the Rule of Consequence to show that our *given* precondition $\{x = 5\}$ is strong enough to satisfy P'.

Step-by-step Proof

1. **Find the weakest precondition (P'):**
   - Command: x := x $*$ 2, Postcondition Q: $\{x > 8\}$
   - Using the Assignment Rule, $P' \equiv Q[x/(x * 2)] \equiv (x * 2 > 8)$.
   - Simplifying P' gives us $\{x > 4\}$.
   - This establishes the validity of the triple: $\{x > 4\}$ x := x $*$ 2 $\{x > 8\}$.
2. **Apply the Rule of Consequence:**
   - Our *given* precondition is $P \equiv \{x = 5\}$.
   - The *required* weakest precondition is $P' \equiv \{x > 4\}$.
   - We must show that $P \to P'$. The statement $(x = 5) \to (x > 4)$ is true.
3. **Conclusion:** Since $(x = 5) \to (x > 4)$ and we have proven $\{x > 4\}$ x := x $*$ 2 $\{x > 8\}$, by the Rule of Consequence, we have proven the original goal. ■

Rules for Program Structure: Sequencing

The Composition Rule

$$\frac{\{P\}\ \texttt{C\_1}\ \{R\} \quad \{R\}\ \texttt{C\_2}\ \{Q\}}{\{P\}\ \texttt{C\_1;}\ \texttt{C\_2}\ \{Q\}}$$

To prove a sequence of commands, find an intermediate assertion R that connects them.

Example

Goal: Prove $\{x = A \wedge y = B\}$ t := x; x := y; y := t $\{x = B \wedge y = A\}$

- Working backward from the goal $\{x = B \wedge y = A\}$:
- Before y := t, we need $\{x = B \wedge t = A\}$.
- Before x := y, we need $\{y = B \wedge t = A\}$.
- Before t := x, we need $\{y = B \wedge x = A\}$, which is logically equivalent to the given precondition.

Composition: Full Proof Example

Goal: Prove $\{x = k\}$ y := x+1; x := y*2 $\{x > 2k\}$

Composition: Full Proof Example

Goal: Prove $\{x = k\}$ `y := x+1; x := y*2` $\{x > 2k\}$

Proof Strategy
1. Work backward from the final postcondition, using the Assignment Rule twice to find the weakest precondition for the entire sequence (P'). 2. Use the Rule of Consequence to show our given precondition $\{x = k\}$ is sufficient.

Step-by-step proof

1. **Analyze x := y*2:**
   - The weakest precondition for this command to establish $\{x > 2k\}$ is the intermediate assertion $R \equiv (y * 2 > 2k) \equiv \{y > k\}$.
   - This establishes: $\{y > k\}$ x := y*2 $\{x > 2k\}$.
2. **Analyze y := x+1:**
   - The postcondition for this command must be $R \equiv \{y > k\}$.
   - The weakest precondition for the whole sequence (P') is $R[y/(x+1)] \equiv ((x+1) > k) \equiv \{x > k-1\}$.
3. **Apply Rule of Consequence:**
   - We have derived the valid triple: $\{x > k-1\}$ y := x+1; x := y*2 $\{x > 2k\}$.
   - Our *given* precondition is $P \equiv \{x = k\}$.
   - We must show that $P \to P'$. The statement $(x = k) \to (x > k-1)$ is true.
4. **Conclusion:** The original goal is valid. ■

Rules for Program Structure: Conditionals

The Conditional Rule (If/Else)

$$\frac{\{P \wedge B\}\ \texttt{C\_1}\ \{Q\} \quad \{P \wedge \neg B\}\ \texttt{C\_2}\ \{Q\}}{\{P\}\ \texttt{if}\ B\ \texttt{then C\_1 else C\_2}\ \{Q\}}$$

You must prove that the postcondition Q holds for both the 'then' and the 'else' branches, starting from the precondition P and the fact that the branch was taken.

Example

Goal: Prove $\{True\}$ `if x < 0 then y := -x else y := x` $\{y = |x|\}$

- **Then path:** Prove $\{x < 0\}$ `y:=-x` $\{y = |x|\}$. Weakest precondition is $(-x = |x|)$, which is true if $x < 0$. Valid.
- **Else path:** Prove $\{x \geq 0\}$ `y:=x` $\{y = |x|\}$. Weakest precondition is $(x = |x|)$, which is true if $x \geq 0$. Valid.
- Both paths are valid, so the triple is correct.

Conditional: Full Proof Example

Let $Q \equiv (m = x \vee m = y) \wedge m \geq x \wedge m \geq y$.

Goal: Prove $\{True\}$ `if x > y then m := x else m := y` $\{Q\}$

Proof Strategy
We use the conditional rule, which requires proving the correctness of the 'then' branch and the 'else' branch separately.

Step-by-step Proof

1. **The "Then" Branch ($x > y$ is true)**
   - **Goal:** Prove $\{x > y\}$ m := x $\{Q\}$.
   - We find the weakest precondition for m := x to establish Q: $Q[m/x] \equiv (x = x \vee x = y) \wedge x \geq x \wedge x \geq y$.
   - This simplifies to $True \wedge True \wedge x \geq y$, which is just $x \geq y$.
   - By the Rule of Consequence, since $(x > y) \rightarrow (x \geq y)$, this branch is valid.

2. The ”Else” Branch ($x \leq y$ is true)

- **Goal:** Prove $\{x \leq y\}$ m := y $\{Q\}$.
- We find the weakest precondition for m := y to establish Q: $Q[m/y] \equiv (y = x \vee y = y) \wedge y \geq x \wedge y \geq y$.
- This simplifies to $\mathit{True} \wedge y \geq x \wedge \mathit{True}$, which is just $y \geq x$.
- By the Rule of Consequence, since $(x \leq y) \rightarrow (y \geq x)$, this branch is valid.

Conclusion: Since both branches guarantee the postcondition Q, the entire conditional statement is correct. ■

The While Rule and Loop Invariants

Reasoning about loops is the most complex part of Hoare Logic.

The Rule of Iteration (While)

$$\frac{\{I \land B\}\ C\ \{I\}}{\{I\}\ \ \{I \land \neg B\}}$$

The key is finding a **Loop Invariant** I. A property I is a valid loop invariant if:

1. **Initialization**: I is true before the loop begins.

The While Rule and Loop Invariants

Reasoning about loops is the most complex part of Hoare Logic.

The Rule of Iteration (While)

$$\frac{\{I \land B\}\ C\ \{I\}}{\{I\}\ \ \{I \land \neg B\}}$$

The key is finding a **Loop Invariant** I. A property I is a valid loop invariant if:

1. **Initialization**: I is true before the loop begins.
2. **Maintenance**: If I and the loop condition B are true at the start of an iteration, I remains true after the loop body C executes.

The While Rule and Loop Invariants

Reasoning about loops is the most complex part of Hoare Logic.

The Rule of Iteration (While)

$$\frac{\{I \wedge B\}\ C\ \{I\}}{\{I\}\ \ \{I \wedge \neg B\}}$$

The key is finding a **Loop Invariant** I. A property I is a valid loop invariant if:

1. **Initialization**: I is true before the loop begins.
2. **Maintenance**: If I and the loop condition B are true at the start of an iteration, I remains true after the loop body C executes.
3. **Termination**: When the loop ends (i.e., $\neg B$), the invariant I helps prove the final postcondition.

While Rule: Full Proof Example (Summation)

Goal: Prove $\{n \geq 1\} \leq$ s:=0; i :=1; while i 0 n do (s:=s+i ; i := i +1) $\{s =$n(n+1)$_{\frac{}{2}}\}$

Step-by-step Proof

1. **Analyze the Loop:** while i <= n do ...
   - **Propose Invariant** I: s is the sum of integers up to $i-1$, and i hasn't gone too far. $I \equiv (s = \frac{(i-1)i}{2} \wedge i \leq n+1)$.
   - **Maintenance:** We must prove $\{I \wedge i \leq n\}$ s:=s+i ; i:=i+1 $\{I\}$. Working backward from I, the weakest precondition for the loop body is $(s+i = \frac{((i+1)-1)(i+1)}{2} \wedge i+1 \leq n+1) \equiv (s+i = \frac{i(i+1)}{2} \wedge i \leq n)$. The left part simplifies to $s = \frac{i(i+1)}{2} - i = \frac{i(i-1)}{2}$. The required precondition is $(s = \frac{i(i-1)}{2} \wedge i \leq n)$. This is exactly what $I \wedge i \leq n$ simplifies to. The invariant is maintained.
   - **Loop Conclusion:** By the While Rule, we have proven: $\{I\}$ while loop $\{I \wedge \neg(i \leq n)\}$. The postcondition $I \wedge i > n$, combined with $i \leq n+1$ from I, implies $i = n+1$. Substituting $i = n+1$ into I gives $s = \frac{((n+1)-1)(n+1)}{2} = \frac{n(n+1)}{2}$.

2. Analyze the Full Program: s :=0; i :=1; while_loop

- We work backwards from the loop's required precondition, which is *I*.
- Weakest precondition before i :=1 is
  $I[i/1] \equiv (s = \frac{(1-1)1}{2} \wedge 1 \leq n+1) \equiv (s = 0 \wedge n \geq 0)$.
- Weakest precondition before s:=0 is $(0 = 0 \wedge n \geq 0) \equiv (n \geq 0)$.

3. Final Conclusion: We have proven
$\{n \geq 0\}$ full program $\{s = n(n+1)_{\frac{}{2}}\}$.*Sinceourgivenprecondition*$(n \geq 1) \rightarrow (n \geq 0)$,
the original goal is proven by the Rule of Consequence. ■

Dijkstra’s Guarded Commands

A New Idea: Nondeterminism

Dijkstra introduced a language to make program derivation more systematic. A core idea was to embrace **nondeterminism**.

The basic building block is the **Guarded Command**:

$$B \rightarrow S$$

Interpretation
The statement list S is eligible for execution *only if* its guard B is true.

GCL Constructs: The Alternative

Alternative Construct: if ... fi

if B1 -> S1 [] ... [] Bn -> Sn fi

- If one or more guards are true, one of the corresponding statements is chosen *arbitrarily* and executed.
- If no guards are true, the program aborts!

Example (Max function)

```
if x >= y -> m := x
[] y >= x -> m := y
fi
```

GCL Constructs: The Repetitive

Repetitive Construct: do ... od

do B1 -> S1 [] ... [] Bn -> Sn od

- As long as at least one guard is true, the machine nondeterministically picks one and executes its command.
- The loop terminates only when all guards are false.

Example (Euclidean Algorithm)

```
do x > y -> x := x - y
[] y > x -> y := y - x
od
```

Dijkstra's Weakest Precondition Calculus

Dijkstra provided a formal semantics for GCL using a *predicate transformer*.

$$\mathrm{wp}(S, Q)$$

wp$(\mathbf{S}, Q)$ The weakest precondition for command S to establish postcondition Q.

Interpretation: Total Correctness
$\mathrm{wp}(S, Q)$ is the set of *all* initial states from which S is **guaranteed to terminate** in a state satisfying Q. This is stronger than Hoare's partial correctness.

We can define 'wp' for each program construct, allowing us to calculate the precondition for an entire program.

WP Rule: Assignment

The ‘wp‘ rule for assignment is identical to the Hoare axiom.

Rule for Assignment

$$\mathrm{wp}(\mathrm{x} := \mathrm{E}, Q) \equiv Q[x/E]$$

Example

Calculate $\mathrm{wp}(\mathrm{x} := \mathrm{x} + 5, x > 20)$.

- We substitute x with $x + 5$ in the postcondition $x > 20$.
- This yields $x + 5 > 20$.
- The weakest precondition is $x > 15$.

WP Rule: Composition

For a sequence of commands, we calculate the ‘wp‘ backwards through the program.

Rule for Composition

$$\text{wp}(\text{S1};\ \text{S2}, Q) \equiv \text{wp}(\text{S1}, \text{wp}(\text{S2}, Q))$$

Example

Calculate wp(y := x * 2; z := y - 3, $z > 10$).

- First, calculate the ‘wp‘ for the second command:

$$\text{wp}(\text{z := y - 3}, z > 10) \equiv (y - 3 > 10) \equiv (y > 13)$$

- Now, use this result as the postcondition for the first command:

$$\text{wp}(\text{y := x * 2}, y > 13) \equiv (x * 2 > 13) \equiv (x > 6.5)$$

WP Rule: Alternative (‘if...fi‘)

The ‘wp‘ for an alternative construct has two conditions.

Rule for Alternative

$$\text{wp}(\text{if...fi0}, Q) \equiv (\vee B_i) \wedge (\wedge (B_i \implies \text{wp}(\text{S\_i}, Q)))$$

1. $\vee B_i$: At least one guard must be true (to prevent abortion).
2. $\wedge (B_i \implies \text{wp}(\text{S\_i}, Q))$: Every enabled path must lead to the desired postcondition.

Alternative Example

Example
For if x > 0 -> y := 1 [] x <= 0 -> y := -1 fi and postcondition $y^2 = 1$:

- Condition 1 (Guards): $(x > 0) \vee (x \leq 0) \equiv \textbf{True}$.
- Condition 2 (Paths):
  - $x > 0 \implies \text{wp(y := 1}, y^2 = 1) \equiv (1^2 = 1) \equiv \textbf{True}$.
  - $x \leq 0 \implies \text{wp(y := -1}, y^2 = 1) \equiv ((-1)^2 = 1) \equiv \textbf{True}$.
- The final ‘wp‘ is $\textbf{True} \wedge \textbf{True} \wedge \textbf{True} \equiv \textbf{True}$.

Comparison and Conclusion

Hoare Logic vs. Dijkstra's Calculus

Concept	Hoare Logic	Dijkstra's WP Calculus
Correctness	Partial Correctness (Assumes termination)	Total Correctness (Guarantees termination)
Nondeterminism	Not naturally handled.	A central, simplifying feature.
Direction	Verification (checking).	Derivation (synthesis).
Primary Use	"Prove this program is correct."	"Derive a correct program."

Conclusion

- **C.A.R. Hoare** gave us the first great logical system for program *verification*. The **Hoare Triple** and the **Loop Invariant** are cornerstones of computer science.

- **Edsger W. Dijkstra** gave us a system for program *derivation*. **Nondeterminism** and the **Weakest Precondition** calculus allow us to construct correct programs from their specifications.

- Both systems demonstrate that programming can be a rigorous, mathematical discipline. They form the bedrock of modern formal methods used in safety-critical systems today.