Lecture Notes: Deductive Program Verification using Hoare Logic*

Andrei Arusoaie

Contents

*The content of this document is not claimed to be original or to be published as original research. It is meant to serve as a learning resource for students.

1 Introduction to Deductive Program Verification

What is Deductive Program Verification?

Deductive program verification is a formal method for proving the correctness of programs using mathematical logic. Unlike testing, which can only demonstrate the presence of bugs for specific inputs, deductive verification aims to prove that a program satisfies its specification for *all* possible inputs.

At its core, deductive verification involves three essential components. First, we need **formal specifications** that express what a program should do using logical formulas. Second, we require a **formal semantics** that provides a precise mathematical model of program execution. Finally, we use a **proof system** consisting of logical rules to prove that the program satisfies its specification.

Why Use Deductive Program Verification?

Deductive program verification offers significant advantages over traditional testing approaches. Most importantly, it provides **correctness guarantees**: while testing only checks specific test cases, verification provides mathematical proof that a program is correct for all inputs satisfying the precondition. This is particularly valuable for **safety-critical systems** where failures can be catastrophic—think of aviation control systems, medical devices, or financial trading systems. In these domains, formal verification provides the highest level of assurance possible.

Beyond safety, deductive methods excel at **bug detection**. Formal methods can detect subtle edge cases and corner cases that are easily missed during testing, especially in complex control flow or arithmetic operations. Additionally, formal specifications serve as precise, unambiguous **documentation** of what a program does, which can be invaluable for maintenance and evolution.

However, deductive verification is not without its challenges. It requires significant **expertise** in logic, proof techniques, and tool usage. The process can be **time-consuming**, especially for complex programs, and writing correct and complete specifications is itself a non-trivial task. Despite these drawbacks, for systems where correctness is paramount, the investment is often worthwhile.

Applications of Deductive Verification

Deductive verification is commonly used to prove various types of properties. Safety properties ensure that bad things never happen—for example, that array bounds are never violated or that null pointers are never dereferenced. We can also verify the absence of arithmetic overflow or underflow, which can lead to subtle security vulnerabilities. For algorithms, we might verify functional correctness, such as proving that a sorting algorithm always produces a sorted array with the same elements as the input. Security properties, including information flow control and absence of timing channels, can also be verified using deductive methods.

2 Historical Background: Floyd-Hoare Logic

Robert Floyd (1967)

Robert Floyd laid the foundation for axiomatic semantics of programs in his seminal 1967 paper. Floyd introduced the concept of using assertions (logical formulas) at various points in a program to reason about its correctness.

Reference: `https://people.eecs.berkeley.edu/~necula/Papers/FloydMeaning.pdf`

C.A.R. Hoare (1969)

Building on Floyd's work, Tony Hoare formalized what is now known as Hoare Logic in 1969. Hoare provided:

- A systematic notation for specifications (Hoare triples)
- A complete set of proof rules for imperative programs
- Proofs of soundness and relative completeness

Reference: `https://dl.acm.org/doi/pdf/10.1145/363235.363259`

Tony Hoare is also famous for inventing the quicksort algorithm and is considered one of the fathers of formal verification. He received the Turing Award in 1980 for his fundamental contributions to programming languages and formal methods.

3 The IMP Language

To study Hoare Logic, we work with a simple imperative programming language called IMP. Despite its simplicity, IMP captures the essential features of imperative programming.

3.1 Syntax of IMP

3.1.1 Arithmetic Expressions

$$\text{AExp} ::= \text{Var} \mid \text{Int} \mid \text{AExp} + \text{AExp} \mid \text{AExp}/\text{AExp}$$

Arithmetic expressions consist of variables, integer constants, and arithmetic operations. For simplicity, we show only addition and division, but other operations (subtraction, multiplication, modulo) can be added similarly.

3.1.2 Boolean Expressions

$$\begin{aligned}\text{BExp} ::= &\texttt{true} \mid \texttt{false} \mid \text{AExp} < \text{AExp} \\ &\mid \texttt{not}\ \text{BExp} \mid \text{BExp}\ \texttt{and}\ \text{BExp}\end{aligned}$$

Boolean expressions include boolean constants, comparisons, and logical operations.

3.1.3 Statements

$$\begin{aligned}\text{Stmt} ::= &\text{Var} := \text{AExp} \\ &\mid \texttt{if}\ \text{BExp}\ \texttt{then}\ \text{Stmt}\ \texttt{else}\ \text{Stmt} \\ &\mid \texttt{while}\ \text{BExp}\ \texttt{do}\ \text{Stmt}\ \texttt{end} \\ &\mid \text{Stmt} ; \text{Stmt} \\ &\mid \texttt{skip}\end{aligned}$$

The statement constructs include:

- Assignment: Changes the value of a variable
- Conditional: Executes one of two branches based on a condition
- While loop: Repeatedly executes a statement while a condition holds
- Sequencing: Executes two statements in order
- Skip: Does nothing (identity statement)

4 Hoare Triples and Specifications

4.1 Hoare Triples

Definition 4.1 (Hoare Triple)**.** *A Hoare triple has the form:*

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

where:

- P *is the* precondition *(a logical formula describing the state before execution)*
- S *is a program statement*
- Q *is the* postcondition *(a logical formula describing the state after execution)*

4.2 Semantics of Hoare Triples

4.2.1 Partial Correctness

Definition 4.2 (Partial Correctness)**.** *A Hoare triple* $\{P\}\ S\ \{Q\}$ *is valid in the sense of* partial correctness*, written* $\models \{P\}\ S\ \{Q\}$*, if:*

Whenever S *is executed in a state satisfying* P*,* ***if*** *the execution terminates, then the resulting state satisfies* Q*.*

Note the key word "if" — partial correctness does not guarantee termination. It only says that *if* the program terminates, then the postcondition holds.

4.2.2 Total Correctness

Definition 4.3 (Total Correctness)**.** *A Hoare triple* $[P]\ S\ [Q]$ *is valid in the sense of* total correctness*, written* $\models [P]\ S\ [Q]$*, if:*

Whenever S *is executed in a state satisfying* P*, the execution* ***must*** *terminate, and the resulting state satisfies* Q*.*

Total correctness = Partial correctness + Termination.

In these notes, we focus on partial correctness. Total correctness requires additional techniques to prove termination (e.g., variant functions for loops).

4.3 Example: Sum Program

Consider the following program that computes the sum of integers from 1 to n:

```
sum := 0;
i := 1;
while (i < n + 1) do
    sum := sum + i;
    i := i + 1
end
```

We can specify this program with the Hoare triple:

$$\{n \geq 0\}\ \texttt{sumPgm}\ \{sum = \tfrac{n(n+1)}{2}\}$$

This says: if n is non-negative before execution, then after execution (if it terminates), sum will equal the formula for the sum of integers from 1 to n.

4.4 Understanding Hoare Triples: Special Cases

Let's examine some special cases to understand Hoare triples better:

1. $\{\texttt{true}\}\ S\ \{Q\}$: The postcondition Q holds after executing S, regardless of the initial state. This is the strongest specification.

2. $\{P\}\ S\ \{\texttt{true}\}$: The postcondition always holds (trivially). This is the weakest, least informative postcondition.

3. $[P]\ S\ [\texttt{true}]$: For total correctness, this says S terminates from any state satisfying P.

4. $\{\texttt{true}\}\ S\ \{\texttt{false}\}$: The program never terminates from any initial state (or executes in a way that makes false true, which is impossible).

5. $\{\texttt{false}\}\ S\ \{Q\}$: Vacuously true for any S and Q, since the precondition is never satisfied.

4.5 Examples of Valid and Invalid Triples

Example 4.4. *Is* $\{i = 0\}$ `while` $i < n$ `do` $i := i + 1$ `end` $\{i = n\}$ *valid?*

This depends on whether n is non-negative. If $n \geq 0$, then yes. If n can be negative, then no (the loop never executes and $i = 0 \neq n$).

Example 4.5. *Is* $\{i = 0\}$ `while` $i < n$ `do` $i := i + 1$ `end` $\{i \geq n\}$ *valid?*

Yes, this is valid (assuming n is a natural number). After the loop, either $i = n$ (if $n \geq 0$) or $i = 0$ and the loop never executed (if $n < 0$), but in both cases $i \geq n$ when the loop condition is false.

5 Proof System for Hoare Logic

5.1 Syntactic Derivability vs Semantic Validity

We distinguish between two concepts:

- $\models \{P\}\ S\ \{Q\}$: The triple is *semantically valid* (true according to the operational semantics)
- $\vdash \{P\}\ S\ \{Q\}$: The triple is *syntactically derivable* (provable using our proof rules)

A proof system consists of *inference rules* that allow us to derive new true statements from known true statements.

5.2 Inference Rules

An inference rule has the form:

$$\frac{\vdash \{P_1\}\ S_1\ \{Q_1\} \quad \ldots \quad \vdash \{P_n\}\ S_n\ \{Q_n\}}{\vdash \{P\}\ S\ \{Q\}}$$

This reads: "If the sequents above the line are provable, then the sequent below the line is also provable."

Rules with no premises (nothing above the line) are called *axioms*.

6 Hoare Logic Proof Rules

6.1 Assignment Rule

$$\frac{\cdot}{\vdash \{Q[e/x]\}\ x := e\ \{Q\}}$$

The assignment rule works *backwards*. To establish postcondition Q after assignment $x := e$, we need to show that $Q[e/x]$ holds before the assignment.

$Q[e/x]$ denotes *substitution*: replace every free occurrence of x in Q with the expression e.

Example 6.1. *Is* $\vdash \{\texttt{true}\}\ i := 2\ \{i = 2\}$ *valid?*

Using the assignment rule with $Q = (i = 2)$ *and* $e = 2$*:*

$$Q[e/i] = (i = 2)[2/i] = (2 = 2) = \texttt{true}$$

So we get:

$$\frac{\cdot}{\vdash \{\texttt{true}\}\ i := 2\ \{i = 2\}}$$

Example 6.2. *Is* $\vdash \{i = k\}\ i := i + 1\ \{i = k + 1\}$ *valid?*

Using the assignment rule with $Q = (i = k + 1)$ *and* $e = i + 1$*:*

$$Q[e/i] = (i = k + 1)[i + 1/i] = (i + 1 = k + 1) = (i = k)$$

So we get:

$$\frac{\cdot}{\vdash \{i = k\}\ i := i + 1\ \{i = k + 1\}}$$

Example 6.3. *Is* $\vdash \{s = k\}\ s := s + i\ \{s = k + i\}$ *valid?*

Using the assignment rule with $Q = (s = k + i)$ *and* $e = s + i$*:*

$$Q[e/s] = (s = k + i)[s + i/s] = (s + i = k + i) = (s = k)$$

So we get:

$$\frac{\cdot}{\vdash \{s = k\}\ s := s + i\ \{s = k + i\}}$$

Example 6.4. *Is* $\vdash \{i \geq 0\}\ i := i + 1\ \{i \geq 1\}$ *valid?*

Using the assignment rule with $Q = (i \geq 1)$ *and* $e = i + 1$*:*

$$Q[e/i] = (i \geq 1)[i + 1/i] = (i + 1 \geq 1) = (i \geq 0)$$

So we get:

$$\frac{\cdot}{\vdash \{i \geq 0\}\ i := i + 1\ \{i \geq 1\}}$$

6.2 Precondition Strengthening (Rule of Consequence)

$$\frac{\vdash \{P'\}\ S\ \{Q\} \qquad P \to P'}{\vdash \{P\}\ S\ \{Q\}}$$

If we can prove the triple with precondition P', and P implies P' (i.e., P is stronger than P'), then the triple with P also holds.

Intuition: A stronger precondition (one that is harder to satisfy) can always be used in place of a weaker one.

Example 6.5 (Precondition Strengthening)**.** *Is* $\vdash \{n > 0\}\ n := n + 1\ \{n \geq 1\}$ *valid?*

From Example 6.4, we know $\vdash \{n \geq 0\}\ n := n + 1\ \{n \geq 1\}$*.*

Since $n > 0 \to n \geq 0$*, we can apply precondition strengthening:*

$$\frac{\vdash \{n \geq 0\}\ n := n + 1\ \{n \geq 1\} \qquad n > 0 \to n \geq 0}{\vdash \{n > 0\}\ n := n + 1\ \{n \geq 1\}}$$

6.3 Postcondition Weakening (Rule of Consequence)

$$\frac{\vdash \{P\}\ S\ \{Q'\} \qquad Q' \to Q}{\vdash \{P\}\ S\ \{Q\}}$$

If we can prove the triple with postcondition Q', and Q' implies Q (i.e., Q is weaker than Q'), then the triple with Q also holds.

Intuition: A weaker postcondition (one that is easier to satisfy) can always be used in place of a stronger one.

Example 6.6 (Postcondition Weakening)**.** *Is* $\vdash \{n > 0\}\ n := n + 1\ \{n > 0\}$ *valid?*

From Example 6.4 and precondition strengthening (Example 6.5), we know $\vdash \{n > 0\}\ n := n + 1\ \{n \geq 1\}$.

Since $n \geq 1 \to n > 0$*, we can apply postcondition weakening:*

$$\frac{\vdash \{n > 0\}\ n := n + 1\ \{n \geq 1\} \qquad n \geq 1 \to n > 0}{\vdash \{n > 0\}\ n := n + 1\ \{n > 0\}}$$

6.4 Composition Rule (Sequential Composition)

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

To verify a sequence of two statements, we verify each statement separately. The postcondition of the first statement must match the precondition of the second statement.

Remark 6.7. *In practice, the postcondition of* S_1 *and precondition of* S_2 *might not match exactly. We can use precondition strengthening or postcondition weakening to make them match.*

Example 6.8 (Composition Rule)**.** *Is* $\vdash \{\texttt{true}\}\ x := 2\,;\, y := x\ \{y = 2 \wedge x = 2\}$ *a valid triple?*

We need to find an intermediate assertion Q*. Let's choose* $Q = (x = 2)$*.*

First: $\vdash \{\texttt{true}\}\ x := 2\ \{x = 2\}$ *(from Example 6.1)*

Second: We need $\vdash \{x = 2\}\ y := x\ \{y = 2 \wedge x = 2\}$

Using the assignment rule: $(y = 2 \wedge x = 2)[x/y] = (x = 2 \wedge x = 2) = (x = 2)$

So: $\vdash \{x = 2\}\ y := x\ \{y = 2 \wedge x = 2\}$

Now we can apply the composition rule:

$$\frac{\vdash \{\texttt{true}\}\ x := 2\ \{x = 2\} \qquad \vdash \{x = 2\}\ y := x\ \{y = 2 \wedge x = 2\}}{\vdash \{\texttt{true}\}\ x := 2\,;\, y := x\ \{y = 2 \wedge x = 2\}}$$

6.5 If Statement Rule

$$\frac{\vdash \{P \wedge C\}\ S_1\ \{Q\} \qquad \vdash \{P \wedge \neg C\}\ S_2\ \{Q\}}{\vdash \{P\}\ \texttt{if}\ C\ \texttt{then}\ S_1\ \texttt{else}\ S_2\ \{Q\}}$$

To verify a conditional, we verify both branches:

- For the then-branch: assume P and C both hold, prove Q after S_1
- For the else-branch: assume P and $\neg C$ both hold, prove Q after S_2

Example 6.9 (If Statement)**.** *Is the follwing triple valid:*

$$\vdash \{\texttt{true}\}\ \texttt{if}\ x < 0\ \texttt{then}\ m := -x\ \texttt{else}\ m := x\ \{m \geq 0\}?$$

We need to prove two branches:

Then-branch: $\vdash \{\texttt{true} \wedge x < 0\}\ m := -x\ \{m \geq 0\}$

Using assignment rule: $(m \geq 0)[-x/m] = (-x \geq 0)$

We need to show: $\texttt{true} \wedge x < 0 \rightarrow -x \geq 0$*, which is true.*

So with precondition strengthening: $\vdash \{\texttt{true} \wedge x < 0\}\ m := -x\ \{m \geq 0\}$

Else-branch: $\vdash \{\texttt{true} \wedge \neg(x < 0)\}\ m := x\ \{m \geq 0\}$

Using assignment rule: $(m \geq 0)[x/m] = (x \geq 0)$

We need to show: $\texttt{true} \wedge \neg(x < 0) \rightarrow x \geq 0$*, which is true.*

So with precondition strengthening: $\vdash \{\texttt{true} \wedge \neg(x < 0)\}\ m := x\ \{m \geq 0\}$

Both branches are provable, so the entire if statement is provable.

6.6 While Loop Rule

$$\frac{\vdash \{I \wedge C\}\ S\ \{I\}}{\vdash \{I\}\ \texttt{while}\ C\ \texttt{do}\ S\ \texttt{end}\ \{I \wedge \neg C\}}$$

This is the most complex rule. It requires finding a *loop invariant* I.

Definition 6.10 (Loop Invariant)**.** *A loop invariant is a logical formula I that:*

1. *Holds before the loop starts*
2. *Is preserved by each iteration of the loop (if I and C hold before the loop body, then I holds after)*
3. *Together with the negation of the loop condition, implies the desired postcondition*

The while rule says: if I is an invariant (preserved by the loop body when the condition holds), then after the loop, $I \wedge \neg C$ holds.

Example 6.11 (While Loop)**.** *Is*

$$\vdash \{i = 0\}\ \texttt{while}\ i < n\ \texttt{do}\ i := i + 1\ \texttt{end}\ \{i = n\}$$

valid (assuming n is a natural number)?

We need to find an invariant. Let's try $I = (i \leq n)$.

Check invariant is preserved: *We need* $\vdash \{i \leq n \wedge i < n\}\ i := i + 1\ \{i \leq n\}$

Using assignment rule: $(i \leq n)[i + 1/i] = (i + 1 \leq n)$

We need: $i \leq n \wedge i < n \rightarrow i + 1 \leq n$*, which is true.*

So the invariant is preserved.

Check the invariant initially holds: $i = 0 \rightarrow i \leq n$ *(true for* $n \geq 0$*)*

Check the invariant and loop exit imply postcondition: $i \leq n \wedge \neg(i < n) \rightarrow i = n$

This simplifies to: $i \leq n \wedge i \geq n \rightarrow i = n$*, which is true.*

7 Soundness and Completeness

7.1 Soundness

Theorem 7.1 (Soundness)**.** *The proof system of Hoare Logic is sound:*

$$\textit{If } \vdash \{P\}\ S\ \{Q\}\textit{, then } \models \{P\}\ S\ \{Q\}$$

This means: if we can prove a triple using the proof rules, then it is semantically valid (true in all executions).

Soundness is the minimum requirement for any useful proof system — we don't want to prove false things!

7.2 Completeness

One might hope for completeness:

$$\text{If } \models \{P\}\ S\ \{Q\}\text{, then } \vdash \{P\}\ S\ \{Q\}$$

Unfortunately, **full completeness does not hold** for Hoare Logic.

The reason is that the rules for precondition strengthening and postcondition weakening require proving implications $\varphi \rightarrow \psi$ in the assertion language. If the assertion language is expressive enough (e.g., includes arithmetic), then by Gödel's incompleteness theorem, there are valid implications that cannot be proven.

7.3 Relative Completeness

However, Hoare proved a weaker result called *relative completeness*:

Theorem 7.2 (Relative Completeness)**.** *If we assume an oracle that can decide the validity of all implications* $\varphi \rightarrow \psi$ *in the assertion language, then the proof system is complete:*

$$\textit{If } \models \{P\}\ S\ \{Q\}\textit{, then } \vdash \{P\}\ S\ \{Q\}$$

This means the proof system is complete "relative to" the ability to prove implications in the assertion logic. The incompleteness comes from the assertion logic (e.g., arithmetic), not from the program logic itself.

8 Summary of Hoare Logic Rules

Here is a complete summary of all the proof rules:

Assignment

$$\frac{}{\vdash \{Q[e/x]\}\ x := e\ \{Q\}}$$

Precondition Strengthening

$$\frac{\vdash \{P'\}\ S\ \{Q\} \quad P \to P'}{\vdash \{P\}\ S\ \{Q\}}$$

Postcondition Weakening

$$\frac{\vdash \{P\}\ S\ \{Q'\} \quad Q' \to Q}{\vdash \{P\}\ S\ \{Q\}}$$

Composition

$$\frac{\vdash \{P\}\ S_1\ \{Q\} \quad \vdash \{Q\}\ S_2\ \{R\}}{\vdash \{P\}\ S_1\ ;\ S_2\ \{R\}}$$

If

$$\frac{\vdash \{P \wedge C\}\ S_1\ \{Q\} \quad \vdash \{P \wedge \neg C\}\ S_2\ \{Q\}}{\vdash \{P\}\ \texttt{if}\ C\ \texttt{then}\ S_1\ \texttt{else}\ S_2\ \{Q\}}$$

While

$$\frac{\vdash \{I \wedge C\}\ S\ \{I\}}{\vdash \{I\}\ \texttt{while}\ C\ \texttt{do}\ S\ \texttt{end}\ \{I \wedge \neg C\}}$$

9 Practical Considerations

9.1 Finding Loop Invariants

Finding appropriate loop invariants is often the most challenging part of using Hoare Logic. Some strategies:

- **Generalize the postcondition**: Start with the desired postcondition and weaken it to something that can be maintained throughout the loop
- **Use the loop body**: Look at what the loop body does and what relationships it maintains
- **Mathematical insight**: For algorithms with known mathematical properties (e.g., summation), use those properties
- **Iteration**: Often finding the right invariant requires several attempts

9.2 Verification Workflow

The typical workflow for deductive program verification is:

1. **Write specifications**: Determine preconditions and postconditions
2. **Annotate the program**: Add loop invariants and other intermediate assertions
3. **Generate verification conditions**: Use the Hoare Logic rules to generate logical formulas that must be proven
4. **Discharge verification conditions**: Use automated theorem provers (SMT solvers) or interactive proof assistants to prove the verification conditions

Modern verification tools (e.g., Dafny, Frama-C, Why3) automate much of this process, but understanding Hoare Logic is essential for using these tools effectively.

10 Conclusion

Hoare Logic provides a rigorous foundation for reasoning about program correctness. Key takeaways:

- Hoare triples $\{P\}\ S\ \{Q\}$ specify program behavior
- Partial correctness deals with what happens *if* a program terminates
- The proof system provides compositional rules for each language construct
- The system is sound and relatively complete
- Finding loop invariants is the main challenge in practice
- Modern tools build on these theoretical foundations

11 Extended Example: The Sum Program Revisited

Let's work through a complete verification of the sum program from earlier:

```
sum := 0;
i := 1;
while (i < n + 1) do
    sum := sum + i;
    i := i + 1
end
```

Specification: $\vdash \{n \geq 0\}$ `sumPgm` $\{sum = \frac{n(n+1)}{2}\}$

11.1 Step 1: Finding the Loop Invariant

The desired postcondition is $sum = \frac{n(n+1)}{2}$. We need an invariant that:

- Is true before the loop
- Is maintained by the loop body
- Together with the loop exit condition, implies the postcondition

Let's consider what happens during execution:

- Initially: $sum = 0, i = 1$
- After 1 iteration: $sum = 1, i = 2$
- After 2 iterations: $sum = 1 + 2 = 3, i = 3$
- After k iterations: $sum = 1 + 2 + \ldots + k, i = k + 1$

So after k iterations, $sum = \frac{k(k+1)}{2}$ and $i = k + 1$.
This suggests the invariant: $sum = \frac{i \cdot (i-1)}{2} \wedge 1 \leq i \leq n + 1$

11.2 Step 2: Check the invariant

Initially: After $sum := 0; i := 1$:
We have $sum = 0 = \frac{1 \cdot 0}{2}$ and $1 \leq 1 \leq n + 1$ (for $n \geq 0$), so the invariant holds.

Preservation: We need to show that if $I \wedge i < n + 1$ holds, then after the loop body, I holds.
Before the loop body: $sum = \frac{i(i-1)}{2} \wedge 1 \leq i \leq n + 1 \wedge i < n + 1$
After $sum := sum + i; i := i + 1$:

Let $i' = i + 1$ and $sum' = sum + i$. We need to show:

$$sum' = \frac{i'(i'-1)}{2}$$

Substituting:

$$sum + i = \frac{(i+1)i}{2}$$

From the precondition, $sum = \frac{i(i-1)}{2}$, so:

$$\frac{i(i-1)}{2} + i = \frac{i(i-1)+2i}{2} = \frac{i^2 - i + 2i}{2} = \frac{i^2+i}{2} = \frac{i(i+1)}{2}$$

This matches our requirement.

Also, $i' = i + 1 \leq n + 1$ (since $i < n + 1$), so the bounds part of the invariant is preserved as well.

11.3 Step 4: Verify the Postcondition

After the loop exits, we have $I \wedge \neg(i < n + 1)$, i.e.:

$$sum = \frac{i(i-1)}{2} \wedge 1 \leq i \leq n + 1 \wedge i \geq n + 1$$

From $i \leq n + 1$ and $i \geq n + 1$, we get $i = n + 1$.

Substituting:

$$sum = \frac{(n+1) \cdot n}{2} = \frac{n(n+1)}{2}$$

This is exactly our postcondition.

The complete formal proof tree is quite large, but the structure is:

1. Use the composition rule to handle $sum := 0; i := 1;$ while...
2. Use the assignment rule twice for the initialization
3. Use the while rule with invariant $I = (sum = \frac{i(i-1)}{2} \wedge 1 \leq i \leq n + 1)$
4. Within the while rule, use composition and assignment rules for the loop body
5. Use precondition strengthening and postcondition weakening as needed

This example illustrates the key challenges and techniques in program verification using Hoare Logic.

12 Exercises

12.1 Exercise 1: Simple Assignments

Prove the following Hoare triples:

1. $\vdash \{x = 5\}\ y := x + 3\ \{y = 8\}$
2. $\vdash \{x > 0\}\ x := x \cdot 2\ \{x > 0\}$
3. $\vdash \{x = y\}\ x := x + 1; y := y + 1\ \{x = y\}$

12.2 Exercise 2: Conditionals

Prove the following:

$$\vdash \{\texttt{true}\}\ \texttt{if}\ x > y\ \texttt{then}\ max := x\ \texttt{else}\ max := y\ \{max \geq x \wedge max \geq y\}$$

12.3 Exercise 3: Finding Loop Invariants

Find suitable loop invariants for the following programs:

1. Factorial:

```
f := 1;
i := 1;
while (i <= n) do
    f := f * i;
    i := i + 1
end
```

   Postcondition: $f = n!$

2. Array search (assuming arrays exist in our language):

```
i := 0;
found := false;
while (i < length(a) and not found) do
    if a[i] = x then
        found := true
    else
        i := i + 1
end
```

   Postcondition: If $found$, then $a[i] = x$

3. GCD (greatest common divisor):

```
while (b != 0) do
    temp := b;
    b := a mod b;
    a := temp
end
```

   Postcondition: $a = \gcd(a_0, b_0)$ where a_0, b_0 are initial values

12.4 Modern Verification Tools

Several practical tools build on Hoare Logic foundations:

- **Dafny**: A programming language with built-in verification. Programmers write specifications as part of the code, and Dafny automatically generates and proves verification conditions.
- **Frama-C**: A framework for analyzing C programs. The WP (weakest precondition) plugin uses Hoare Logic to verify C code annotated with ACSL specifications.

- **Why3**: A platform for deductive program verification that supports multiple programming languages and theorem provers.
- **VeriFast**: A tool for verifying C and Java programs, with strong support for concurrent programs using separation logic.

12.5 Limitations and Extensions

While Hoare Logic is foundational, it has limitations for real-world programs:

- **Pointers and heap**: Classical Hoare Logic doesn't handle pointers well. *Separation Logic* extends Hoare Logic for reasoning about heap-manipulating programs.
- **Concurrency**: Concurrent programs require extensions like *Concurrent Separation Logic* or *Rely-Guarantee reasoning*.
- **Procedures and recursion**: Need additional rules for function calls and recursion.
- **Object-oriented features**: Require reasoning about dynamic dispatch, inheritance, etc.
- **Scale**: Verifying large programs requires modular reasoning and abstraction techniques.

Despite these challenges, Hoare Logic remains the theoretical foundation for modern program verification.

13 Conclusion

This course has covered the fundamental concepts of deductive program verification using Hoare Logic:

- The concept of formal specifications using preconditions and postconditions
- Hoare triples as a way to express program correctness
- A compositional proof system with rules for each language construct
- The critical role of loop invariants in reasoning about loops
- Soundness and relative completeness of the proof system
- Connections to modern verification tools and techniques

Hoare Logic represents one of the great achievements in computer science: a rigorous, mathematical foundation for proving program correctness. While challenges remain in scaling these techniques to large, complex systems, the fundamental insights continue to guide research and practice in program verification.

For students interested in pursuing this topic further, recommended next steps include:

- Hands-on experience with verification tools (Dafny is particularly beginner-friendly)
- Study of separation logic for reasoning about pointers and concurrency
- Exploration of automated invariant inference techniques
- Understanding of SMT solvers and their role in automated verification
- Investigation of certified programming with proof assistants (Coq, Isabelle)

13.1 How Dafny Uses Hoare Logic Internally

When you write a Dafny program with specifications, the following happens behind the scenes:

1. **Verification Condition Generation**: Dafny applies the Hoare Logic rules (assignment, composition, if, while) to transform your program into a set of logical formulas called verification conditions (VCs).

2. **Weakest Precondition Calculation**: For each statement, Dafny computes the weakest precondition that guarantees the postcondition holds, similar to what we do manually when applying the assignment rule backwards.

3. **SMT Solving**: The generated VCs are passed to the Z3 SMT solver, which attempts to prove them automatically. If Z3 succeeds, your program is verified. If it fails, Dafny reports which assertion or postcondition couldn't be proven.

4. **Modular Verification**: Dafny verifies each method independently using only its specification, not its implementation. This corresponds to the compositional nature of Hoare Logic.

13.2 The Sum Program in Dafny

Let's see our running example (the sum program) in Dafny:

```
method mysum(n: nat) returns (sum: int)
    requires n >= 0
    ensures sum == n * (n + 1) / 2
{
    sum := 0;
    var i := 1;

    while i < n + 1
        invariant 1 <= i <= n + 1
        invariant sum == i * (i - 1) / 2
    {
        sum := sum + i;
        i := i + 1;
    }
}
```

This is a direct translation of our Hoare Logic proof into executable, verified code. Dafny will automatically:

- Check that the invariant holds after initialization
- Verify that the loop body preserves the invariant
- Confirm that the invariant plus loop exit implies the postcondition
- Handle all the arithmetic reasoning using Z3

13.3 Advantages of Tool-Assisted Verification

Using Dafny (or similar tools) provides several advantages over manual Hoare Logic proofs:

- **Automation**: The tool automatically generates and checks verification conditions
- **Fast feedback**: You get immediate feedback when a proof fails

- **Executable specifications**: Your verified code can actually run
- **Better error messages**: When verification fails, tools can often point to the problematic line
- **Scalability**: Tools can handle larger programs than manual proofs

However, the human insight required to find good loop invariants and specifications remains essential. The tool can check proofs automatically, but it cannot (yet) discover the key insights needed for verification.

13.4 Other tools

While Dafny is an excellent learning tool and practical verification system, the principles of Hoare Logic extend to many other tools:

- **Frama-C with WP plugin**: Verifies C code using weakest precondition calculus
- **VeriFast**: Uses separation logic (an extension of Hoare Logic) for C and Java
- **Why3**: A general verification platform supporting multiple languages and provers
- **SPARK**: A subset of Ada with built-in verification support
- **Viper**: An intermediate verification language used by several tools

All of these tools are built on the same foundational ideas: preconditions, postconditions, invariants, and the compositional proof rules of Hoare Logic.

13.5 Conclusion

Hoare Logic is not just a theoretical framework—it's the foundation of practical program verification tools used in industry today. By understanding Hoare Logic deeply, you gain the conceptual tools needed to:

- Write precise specifications
- Reason about program correctness
- Find appropriate loop invariants
- Use modern verification tools effectively
- Understand why verification succeeds or fails

Dafny and other tools automate the mechanical aspects of proof checking, but the creative insights—what should the specifications be? what is the right invariant?—still require human intelligence and understanding of the underlying Hoare Logic principles.

14 References and Further Reading

- Floyd, R. W. (1967). "Assigning meanings to programs." *Proceedings of Symposia in Applied Mathematics*, 19, 19-32.
- Hoare, C. A. R. (1969). "An axiomatic basis for computer programming." *Communications of the ACM*, 12(10), 576-580.

- Dijkstra, E. W. (1975). "Guarded commands, nondeterminacy and formal derivation of programs." *Communications of the ACM*, 18(8), 453-457.
- Apt, K. R., & Olderog, E. R. (2019). *Verification of Sequential and Concurrent Programs*. Springer.
- Reynolds, J. C. (2002). "Separation logic: A logic for shared mutable data structures." *Proceedings of LICS*, 55-74.
- Leino, K. R. M. (2010). "Dafny: An automatic program verifier for functional correctness." *LPAR*, 348-370.