Hoare Logic

Program Verification

Your Name

August 1, 2025

Outline

Syntax of the Language

Based on Backus-Naur Form (BNF)

Expressions:

$$E ::= N \mid V \mid E_1 + E_2 \mid E_1 - E_2 \mid E_1 \times E_2 \mid \ldots$$

Boolean expressions:

$$B ::= \mathbf{T} \mid \mathbf{F} \mid E_1 = E_2 \mid E_1 \leq E_2 \mid \ldots$$

Commands:

$$\begin{array}{lcl} C & ::= & V := E \\ & \mid & C_1;\ C_2 \\ & \mid & \text{IF } B \text{ THEN } C_1 \text{ ELSE } C_2 \\ & \mid & \text{WHILE } B \text{ DO } C' \end{array}$$

Example Programs - 1

Illustrating the language syntax

Factorial of a number ‘n‘

This program computes $n!$ and stores the result in the variable ‘fact‘. It assumes the variable ‘n‘ holds a non-negative integer. The body of the ‘while‘ loop is a sequence of two assignment commands.

```
fact := 1;
i := n;
while i > 0 do
    fact := fact * i;
    i := i - 1
```

Example Programs - 2

Maximum of two numbers ‘x‘ and ‘y‘

This program uses a conditional statement to find the maximum of two numbers, ‘x‘ and ‘y‘, and stores the result in ‘max‘.

```
if x <= y then
    max := y
else
    max := x
```

What is a Program Specification?

The Contract

A program specification acts as a formal contract. It precisely describes the expected behavior of a piece of code.

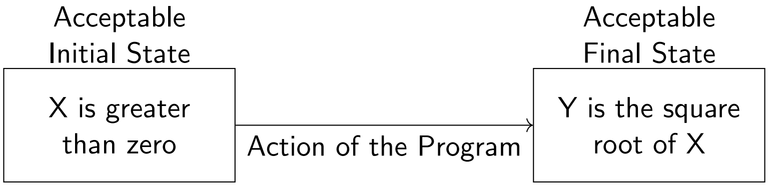
- It does **not** describe *how* the program works.
- It **does** describe *what* the program must accomplish.

Key Components

A specification consists of two main parts:

- **Precondition:** A condition that must be true *before* the program is executed.
- **Postcondition:** A condition that is guaranteed to be true *after* the program terminates.

Visualizing a Specification
From Initial to Final State



Hoare's Notation

Historical Context

C.A.R. Hoare introduced the following notation called a **partial correctness specification** for specifying what a program does:

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

Components

- C is a command (a program or program fragment)
- P and Q are conditions on the program variables used in C
- P is called the **precondition**
- Q is called the **postcondition**

The Precondition (P)

Acceptable Initial State

The **precondition** defines the set of initial states for which the program is guaranteed to work correctly.

- It's an assumption about the values of program variables before execution.
- If the precondition is not met, the program has no obligations. It can crash, loop forever, or produce a wrong answer.
- Note: Reasoning about memory layout and heap requires *Separation Logic*, an extension of Hoare Logic that can reason about pointer structures and memory allocation

Example

For a program that calculates the square root of X:

Informal: "X is greater than zero"

Formal: $\{X > 0\}$

The Postcondition (Q)

Acceptable Final State

The **postcondition** describes the state of the program after it has finished executing.

- It's the "promise" or "guarantee" of the specification.
- It typically relates the final values of variables to their initial values.

Example

For the square root program:

Informal: "Y is the square root of X"

Formal: $\{Y \times Y = X \land Y \geq 0\}$

(Note: we relate the final value of Y to the initial value of X).

Writing Conditions

Mathematical Notation

Conditions on program variables will be written using standard mathematical notations together with **logical operators**:

- $\wedge$ (and)
- $\vee$ (or)
- $\neg$ (not)
- $\Rightarrow$ (implies)

Example

Some example conditions:

- $x > 0 \wedge y \geq 0$ (x is positive AND y is non-negative)
- $x = 0 \vee y = 0$ (x equals zero OR y equals zero)
- $x > 0 \Rightarrow x^2 > 0$ (if x is positive, then x squared is positive)

Formal Specification: The Hoare Triple

Combining Pre- and Postconditions

Hoare Logic provides a formal notation to write specifications, called a **Hoare Triple**.

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

This is read as:

> *If the precondition P is true before executing the program S, and if S terminates, then the postcondition Q will be true afterward.*

Example (Square Root Specification)

Combining our previous examples, the specification for a square root program S is:

$$\{X > 0\}\ S\ \{Y \times Y = X \wedge Y \geq 0\}$$

Here, S is the placeholder for the actual program code (the "Action").

Evolution of Notation

Historical Note

Hoare's original notation was $P\ \{C\}\ Q$ not $\{P\}\ C\ \{Q\}$, but the latter form is now more widely used.

Alternative Notations

You may encounter different notations in the literature:

- Original: $P\ \{C\}\ Q$
- Modern: $\{P\}\ C\ \{Q\}$
- Some texts: $\{P\}\ C\ \{Q\}$ (without special formatting)

All represent the same concept: a partial correctness specification.

Partial Correctness

What is Partial Correctness?

A Hoare triple $\{P\}\ C\ \{Q\}$ expresses **partial correctness**:

> *If the precondition P is true before executing command C, and if C terminates, then the postcondition Q will be true after execution.*

Important: Termination Not Guaranteed

Partial correctness does **not** guarantee that the program terminates!

- It only says what must be true *if* the program terminates
- A program that loops forever can still be partially correct
- Total correctness = Partial correctness + Termination

Reading Hoare Triples

How to Read $\{P\}\ C\ \{Q\}$

The triple $\{P\}\ C\ \{Q\}$ can be read as:

1. "If P is true, then after C executes, Q will be true"
2. "C transforms states satisfying P into states satisfying Q"
3. "Starting from P, command C establishes Q"

Example (Simple Assignment)

$\{x = 5\}\ y := x + 1\ \{y = 6\}$

This reads as: "If x equals 5 before the assignment, then y will equal 6 after the assignment."

Meaning of Hoare's Notation

Formal Definition

$\{P\}\ C\ \{Q\}$ is true if:

- whenever C is executed in a state satisfying P
- and *if* the execution of C terminates
- then the state in which C terminates satisfies Q

Example (Assignment Command)

Consider: $\{X = 1\}\ X := X + 1\ \{X = 2\}$

- P is the condition that the value of X is 1
- Q is the condition that the value of X is 2
- C is the assignment command $X := X + 1$ (i.e. 'X becomes X+1')

Truth and Falsity of Hoare Triples

Example (True Triple)

$\{X = 1\}\ X := X + 1\ \{X = 2\}$ is **true**

Why? Starting from a state where $X = 1$, executing $X := X + 1$ results in $X = 2$.

Example (False Triple)

$\{X = 1\}\ X := X + 1\ \{X = 3\}$ is **false**

Why? Starting from $X = 1$, executing $X := X + 1$ results in $X = 2$, not $X = 3$.

Key Insight

A Hoare triple is a mathematical statement that can be either true or false. It makes a claim about what happens when a program executes.

Hoare Logic and Verification Conditions

What is Hoare Logic?

Hoare Logic is a **deductive proof system** for Hoare triples $\{P\}\ C\ \{Q\}$

- Provides **axioms** (basic facts about simple commands) and **inference rules** (ways to combine proofs)
- Example: Assignment axiom, sequence rule, while loop rule
- Forms the theoretical foundation for program verification

Direct Verification with Hoare Logic

Advantages:

- Original proposal by Hoare
- Provides complete formal proofs

Disadvantages:

- Tedious and error-prone for humans
- Impractical for large programs

Verification Conditions

Definition: What is a Verification Condition?

A **verification condition** is a mathematical formula (without program constructs) whose truth implies the correctness of a program.

- Generated from Hoare triples by analyzing the program structure
- Expressed purely in terms of logic and mathematics
- No references to program execution or state changes

Verification Conditions -2

Modern Approach: Verification Conditions

Can 'compile' proving $\{P\}\ C\ \{Q\}$ to **verification conditions**

- More natural for automated reasoning
- Basis for computer-assisted verification
- Separates program logic from mathematical reasoning

Verification Conditions -3

Key Property

Proof of verification conditions is **equivalent** to proof with Hoare Logic

- Hoare Logic can be used to *explain* verification conditions
- Both approaches prove the same correctness properties
- Verification conditions are more amenable to automation

Verification Condition Example

Example (Simple Verification Condition)

To prove $\{x > 0\}\ y := x + 1\ \{y > 1\}$:

Step 1: Analyze what the program does

- The assignment $y := x + 1$ sets y to the value of $x + 1$

Step 2: Generate the verification condition

- We need: if $x > 0$ initially, then $y > 1$ after assignment
- Since y will equal $x + 1$, we need: $x > 0 \Rightarrow (x + 1) > 1$

Step 3: The verification condition is:

$$x > 0 \Rightarrow (x + 1) > 1$$

This is a pure mathematical statement that can be proved using algebra, without any reference to program execution!

Partial Correctness Specification

Definition

An expression $\{P\}\ C\ \{Q\}$ is called a **partial correctness specification**

- P is called its **precondition**
- Q its **postcondition**

When is $\{P\}\ C\ \{Q\}$ true?

$\{P\}\ C\ \{Q\}$ is true if:

- whenever C is executed in a state satisfying P
- and *if* the execution of C terminates
- then the state in which C's execution terminates satisfies Q

Why "Partial" Correctness?

The Key Point

These specifications are 'partial' because for $\{P\}\ C\ \{Q\}$ to be true it is **not** necessary for the execution of C to terminate when started in a state satisfying P

What is Required

It is only required that *if* the execution terminates, *then* Q holds

Example (Infinite Loop)

$\{X = 1\}$ WHILE T DO $X := X$ $\{Y = 2\}$ – this specification is true!
Why? The loop never terminates, so we never need to check if $Y = 2$. The specification only makes a claim about what happens *if* the program terminates.

Total Correctness Specification

Definition

A stronger kind of specification is a **total correctness specification**

- There is no standard notation for such specifications
- We shall use $[P]\ C\ [Q]$

When is $[P]\ C\ [Q]$ true?

A total correctness specification $[P]\ C\ [Q]$ is true if and only if:

- whenever C is executed in a state satisfying P the execution of C terminates
- after C terminates Q holds

Total Correctness Example

Example (False Total Correctness)

$[X = 1]\ Y := X;\ \mathrm{WHILE\ T\ DO}\ X := X\ [Y = 1]$

This says that:

- the execution of $Y := X;\ \mathrm{WHILE\ T\ DO}\ X := X$ terminates when started in a state satisfying $X = 1$
- after which $Y = 1$ will hold

This is clearly false because the while loop never terminates!

Key Difference

- Partial correctness: $\{P\}\ C\ \{Q\}$ – "If it terminates, then..."
- Total correctness: $[P]\ C\ [Q]$ – "It terminates and then..."

Relationship Between Partial and Total Correctness

Mathematical Relationship

Total correctness = Partial correctness + Termination

$[P]\ C\ [Q] \equiv \{P\}\ C\ \{Q\} \wedge$ "C terminates when started in a state satisfying P"

Practical Implications

- Proving partial correctness is often easier
- Proving termination requires additional techniques
  - **Variant functions**: expressions that decrease with each loop iteration and are bounded below
  - Also called *ranking functions* or *termination measures*
- Many verification tools focus on partial correctness first
- Total correctness is needed for critical systems

Auxiliary Variables

Example (Variable Swap)

$\{X = x \land Y = y\}\ R := X;\ X := Y;\ Y := R\ \{X = y \land Y = x\}$

This says that *if* the execution of

```
R:=X; X:=Y; Y:=R
```

terminates (which it does)

then the values of X and Y are exchanged

Key Observation

The variables x and y, which don't occur in the command and are used to name the initial values of program variables X and Y

What are Auxiliary Variables?

Definition

Variables that appear in specifications but not in the program code are called:

- **Auxiliary variables**
- **Ghost variables**
- **Specification variables**

Purpose

Auxiliary variables allow us to:

- Refer to initial values of program variables in postconditions
- Express relationships between initial and final states
- Write more expressive specifications

Naming Convention

Informal Convention

To distinguish between program variables and auxiliary variables:

- **Program variables** are UPPER CASE (e.g., X, Y, Z)
- **Auxiliary variables** are lower case (e.g., x, y, z)

Example (More Examples)

- $\{X = x\}\ X := X + 1\ \{X = x + 1\}$ – x remembers initial value
- $\{X = x \land Y = y\}\ X := X + Y\ \{X = x + y \land Y = y\}$
- $\{A[i] = a\}\ A[i] := 0\ \{A[i] = 0 \land \text{“old } A[i] = a\text{”}\}$

Why Auxiliary Variables Matter

Without Auxiliary Variables

Consider trying to specify variable swap without auxiliary variables:

- $\{?\}\ R := X;\ X := Y;\ Y := R\ \{?\}$
- How do we say "X gets Y's initial value"?
- We can't refer to initial values!

With Auxiliary Variables

We can express complex relationships:

- Maximum: $\{X = x \wedge Y = y\}$ ... $\{M = \max(x, y)\}$
- Sorting: $\{A = a\}$ ... {"A is a sorted permutation of a"}
- Any computation relating initial and final states

Important Notes about Auxiliary Variables

Key Properties

1. Auxiliary variables are **immutable** – they never change value during program execution
2. They exist only in specifications, not in the actual program
3. They are universally quantified (implicitly)

Formal Interpretation

The specification $\{X = x\}\ C\ \{Q(x)\}$ actually means:

$$\forall x.\ \{X = x\}\ C\ \{Q(x)\}$$

"For all values x, if X starts with value x, then after C, property $Q(x)$ holds"

Floyd-Hoare Logic

The Need for Formal Proofs

To construct formal proofs of partial correctness specifications, *axioms* and *rules of inference* are needed

What Floyd-Hoare Logic Provides

This is what Floyd-Hoare logic provides:

- The formulation of the deductive system is due to Hoare
- Some of the underlying ideas originated with Floyd

Structure of Proofs in Floyd-Hoare Logic

Proof Definition

A proof in Floyd-Hoare logic is a sequence of lines, each of which is either:

- An **axiom** of the logic, or
- Follows from earlier lines by a **rule of inference** of the logic

Note: Proofs can also be trees, if you prefer

Purpose of Formal Proofs

A formal proof makes explicit what axioms and rules of inference are used to arrive at a conclusion

Components of Floyd-Hoare Logic

Axioms

Axioms are basic facts about specific programming constructs that require no proof:

- Assignment axiom
- Skip axiom (for the empty command)
- Other basic command axioms

Rules of Inference

Rules of inference allow us to derive new facts from existing ones:

- Sequence rule (composition)
- Conditional rule (if-then-else)
- While loop rule
- Consequence rule

Historical Context

Robert W. Floyd (1936-2001)

- Introduced flowchart-based verification methods (1967)
- Pioneered the use of loop invariants
- Developed techniques for proving program termination

C.A.R. Hoare (1934-)

- Formalized Floyd's ideas into a logical system (1969)
- Introduced the triple notation $\{P\}C\{Q\}$
- Created the axiomatic semantics approach

Note

The system is called "Floyd-Hoare Logic" to honor both contributors

Example: What a Proof Looks Like

Example (Simple Proof Structure)

To prove $\{x = 5\}\ y := x + 1; z := y\ \{z = 6\}$:

1. $\{x = 5\}\ y := x + 1\ \{y = 6\}$ (Assignment axiom)
2. $\{y = 6\}\ z := y\ \{z = 6\}$ (Assignment axiom)
3. $\{x = 5\}\ y := x + 1; z := y\ \{z = 6\}$ (Sequence rule on 1,2)

Each line is justified by an axiom or rule!

Key Insight

Floyd-Hoare Logic provides a *systematic* way to prove program correctness, not just intuitive arguments

Judgements

Three Kinds of Things That Could Be True or False

- **Statements of mathematics**, e.g., $(X+1)^2 = X^2 + 2 \times X + 1$
- **Partial correctness specifications** $\{P\}C\{Q\}$
- **Total correctness specifications** $[P]C[Q]$

What Are Judgements?

These three kinds of things are examples of *judgements*

- A logical system gives rules for proving judgements
- Floyd-Hoare logic provides rules for proving partial correctness specifications
- The laws of arithmetic provide ways of proving statements about integers

Proving Judgements

The Turnstile Notation

$\vdash S$ means statement S can be proved

- How to prove predicate calculus statements assumed known
- This course covers axioms and rules for proving *program correctness statements*

Note

We will introduce the specific axioms and inference rules of Floyd-Hoare logic in detail in the following sections

Example (Different Types of Provable Judgements)

- $\vdash (x + y)^2 = x^2 + 2xy + y^2$ (mathematical)
- $\vdash \{x = 5\} y := x + 1 \{y = 6\}$ (program correctness)
- $\vdash [x \geq 0] y := \sqrt{x} [y^2 = x]$ (total correctness)

Why Judgements Matter

Formal vs Informal Reasoning

- **Informal**: "Obviously, if x=5 then after y:=x+1, y will be 6"
- **Formal**: Use axioms and rules to derive $\vdash \{x = 5\} y := x + 1 \{y = 6\}$

Benefits of Formal Judgements

1. **Precision**: No ambiguity about what needs to be proved
2. **Mechanization**: Can be checked by computers
3. **Composability**: Complex proofs built from simpler ones
4. **Confidence**: Mathematical certainty about correctness

Types of Logical Systems

Different Logical Systems for Different Judgements

Judgement Type	**Logical System**
Mathematical statements	Predicate logic, arithmetic
Partial correctness	Floyd-Hoare logic
Total correctness	Extended Hoare logic
Type checking	Type systems

Focus of This Course

This course focuses on Floyd-Hoare logic for proving partial correctness specifications

- We'll learn the axioms (basic facts)
- We'll learn the inference rules (ways to combine facts)
- We'll practice constructing formal proofs

Reminder of our Little Programming Language

Axiomatic Semantics

The proof rules that follow constitute an *axiomatic semantics* of our programming language

Expressions

$$E ::= N \mid V \mid E_1 + E_2 \mid E_1 - E_2 \mid E_1 \times E_2 \mid \ldots$$

Boolean expressions

$$B ::= \mathbf{T} \mid \mathbf{F} \mid E_1 = E_2 \mid E_1 \leq E_2 \mid \ldots$$

Reminder of our Little Programming Language - 2

Commands

$$
\begin{array}{rll}
C ::= & V := E & \text{Assignments} \\
 & \mid C_1 ; C_2 & \text{Sequences} \\
 & \mid \text{IF } B \text{ THEN } C_1 \text{ ELSE } C_2 & \text{Conditionals} \\
 & \mid \text{WHILE } B \text{ DO } C & \text{WHILE-commands}
\end{array}
$$

Substitution Notation

Definition

$Q[E/V]$ is the result of replacing all occurrences of V in Q by E

- Read $Q[E/V]$ as 'Q with E for V'
- For example: $(X + 1 > X)[Y + Z/X] = ((Y + Z) + 1 > Y + Z)$
- Ignoring issues with bound variables for now (e.g. variable capture)

Substitution in Terms

Same notation for substituting into terms, e.g. $E_1[E_2/V]$

The Cancellation Law

Substitution as Cancellation

Think of this notation as the 'cancellation law'

$$V[E/V] = E$$

which is analogous to the cancellation property of fractions

$$v \times (e/v) = e$$

Important Property

Note that $Q[x/V]$ doesn't contain V (if $V \neq x$)

The Cancellation Law - 2

Example (Substitution Examples)

- $(X + Y > 0)[5/X] = (5 + Y > 0)$
- $(X \times X = Y)[X + 1/X] = ((X + 1) \times (X + 1) = Y)$
- $(X > Y \wedge Y > Z)[W/Y] = (X > W \wedge W > Z)$

Why Substitution Matters

Connection to Assignment

Substitution notation is crucial for understanding the assignment axiom:

- If we want Q to be true after $V := E$
- Then $Q[E/V]$ must be true before
- Because after assignment, V will have the value that E had before

Preview: Assignment Axiom

This leads to the assignment axiom (details coming next)

Reading Inference Rules

Inference Rule Notation

Before we see the axioms and rules, let's understand the notation:

$$\frac{\text{premises}}{\text{conclusion}}$$

- The line is read as "implies" or "allows us to derive"
- Above the line: what we need to prove (premises)
- Below the line: what we can conclude
- If nothing above the line: it's an **axiom** (needs no proof)

Reading Inference Rules - 2

Example (Reading an Inference Rule)

$$\frac{A \quad B}{C}$$

This means: "If we can prove A and we can prove B, then we can conclude C"

The Assignment Axiom

Assignment Axiom

Now we can understand the assignment axiom:

$$\frac{}{\{Q[E/V]\}\ V := E\ \{Q\}}$$

- Nothing above the line = this is an axiom
- Below the line = what we can always conclude
- Read: "We can always derive that $\{Q[E/V]\}\ V := E\ \{Q\}$ is true"

Understanding the Axiom

Read backwards: to achieve Q after $V := E$, need $Q[E/V]$ before

The Assignment Axiom (Hoare)

Assignment Syntax and Semantics

- **Syntax**: $V := E$
- **Semantics**: value of V in final state is value of E in initial state
- **Example**: $X := X + 1$ (adds one to the value of the variable X)

The Assignment Axiom

$$\vdash \{Q[E/V]\}\ V := E\ \{Q\}$$

Where V is any variable, E is any expression, Q is any statement.

Instances of the Assignment Axiom

Examples

Instances of the assignment axiom are:

- $\vdash \{E = x\}\ V := E\ \{V = x\}$
- $\vdash \{Y = 2\}\ X := 2\ \{Y = X\}$
- $\vdash \{X + 1 = n + 1\}\ X := X + 1\ \{X = n + 1\}$
- $\vdash \{E = E\}\ X := E\ \{X = E\}$ (if X does not occur in E)

Key Insight

The precondition is obtained by substituting E for V in the postcondition!

Understanding the Assignment Axiom

Why Does This Work?

Let's think step by step:

1. We want property Q to hold after executing $V := E$
2. After the assignment, V has the value that E had before
3. So if we want Q to be true about V after...
4. Then Q must have been true about E before!
5. That's exactly what $Q[E/V]$ expresses

Example (Step-by-Step)

Want: $\{?\}\ X := Y + 1\ \{X > 5\}$

- After: $X > 5$ must be true
- Before: $(Y + 1) > 5$ must be true
- So: $\{Y + 1 > 5\}\ X := Y + 1\ \{X > 5\}$

The Backwards Fallacy

Common Misconception

Many people feel the assignment axiom is ‘backwards’

First Erroneous Intuition

One common erroneous intuition is that it should be:

$$\vdash \{P\}\ V := E\ \{P[V/E]\}$$

where $P[V/E]$ denotes the result of substituting V for E in P

Why This is Wrong

This has the false consequence $\vdash \{X = 0\}\ X := 1\ \{X = 0\}$

- Since $(X = 0)[X/1]$ is equal to $(X = 0)$
- Because 1 doesn’t occur in $(X = 0)$
- But clearly X cannot equal 0 after we set it to 1!

The Backwards Fallacy - 2

Second Erroneous Intuition

Another erroneous intuition is that it should be:

$$\vdash \{P\}\ V := E\ \{P[E/V]\}$$

Why This is Also Wrong

This has the false consequence $\vdash \{X = 0\}\ X := 1\ \{1 = 0\}$

- Taking P to be $X = 0$, V to be X, and E to be 1
- We get $(X = 0)[1/X] = (1 = 0)$
- But $1 = 0$ is always false!

The Correct Direction

The assignment axiom goes "backwards" because we substitute in the *precondition*, not the postcondition!

Why "Backwards" is Actually Forward

Think About Information Flow

The assignment axiom seems backwards but it's actually forward-thinking:

- We start with what we *want* (the postcondition Q)
- We work out what we *need* (the precondition $Q[E/V]$)
- This is called **weakest precondition reasoning**

Example (Working Backwards)

Goal: Ensure $Y = 10$ after $Y := X \times 2$

- Postcondition: $Y = 10$
- Substitute: $(Y = 10)[X \times 2/Y] = (X \times 2 = 10)$
- Simplify: $X = 5$
- Result: $\{X = 5\}\ Y := X \times 2\ \{Y = 10\}$

Validity

The Importance of Validity

Important to establish the validity of axioms and rules

Formal Semantics and Soundness

Later will give a *formal semantics* of our little programming language

- Then *prove* axioms and rules of inference of Floyd-Hoare logic are sound
- This will only increase our confidence in the axioms and rules to the extent that we believe the correctness of the formal semantics!

The Assignment Axiom in Real Languages

Important Limitation

The Assignment Axiom is not valid for ‘real’ programming languages

Historical Note

In an early PhD on Hoare Logic, G. Ligler showed that the assignment axiom can fail to hold in six different ways for the language Algol 60

Why This Matters

- Our simple language has carefully chosen features
- Real languages have complications that break the axiom
- Understanding these limitations helps us apply Hoare Logic correctly

Expressions with Side Effects

The Hidden Assumption

The validity of the assignment axiom depends on expressions not having side effects

Example (Block Expression)

Suppose our language were extended to contain the 'block expression':

```
BEGIN Y:=1; 2 END
```

- This expression has value 2
- But its evaluation also 'side effects' the variable Y by storing 1 in it

Why Side Effects Break the Assignment Axiom

The Problem

If the assignment axiom applied to block expressions, then it could be used to deduce:

$$\vdash \{Y = 0\}\ X := \texttt{BEGIN Y:=1; 2 END}\ \{Y = 0\}$$

The Faulty Reasoning

- Since $(Y = 0)[E/X] = (Y = 0)$ (because X does not occur in $(Y = 0)$)
- By the assignment axiom, we'd conclude the above
- This is clearly false: after the assignment Y will have the value 1!

The Lesson

The assignment axiom only works when expressions are **pure** (no side effects)

Other Ways the Assignment Axiom Can Fail

Real Language Complications

In real programming languages, the assignment axiom can fail due to:

1. **Side effects in expressions** (as we just saw)
2. **Aliasing**: multiple names for the same location
3. **Call by reference**: procedure parameters that modify variables
4. **Global variables**: hidden dependencies between parts of code
5. **Undefined behavior**: division by zero, array bounds violations
6. **Concurrent modification**: other threads changing variables

Defensive Programming

Understanding these limitations helps us:

- Design better programming languages
- Write more verifiable code
- Know when we can trust our formal proofs

Example: Aliasing Breaking the Assignment Axiom

Example (Aliasing Problem)

Consider if our language had arrays and we tried:

$$\vdash \{A[i] = 5\}\ A[j] := 0\ \{A[i] = 5\}$$

The assignment axiom would suggest this is valid because:

- $(A[i] = 5)[0/A[j]]$ might seem to be just $(A[i] = 5)$
- But what if $i = j$? Then $A[i]$ and $A[j]$ are the same location!
- After the assignment, $A[i] = 0$, not 5

The Solution

In real verification:

- Must track when different expressions might refer to same location
- Need more sophisticated rules for arrays and pointers
- This leads to *separation logic* and other advanced techniques

A Forwards Assignment Axiom (Floyd)

Floyd's Original Formulation

This is the original semantics of assignment due to Floyd:

$$\vdash \{P\}\ V := E\ \{\exists v.\ V = E[v/V] \wedge P[v/V]\}$$

where v is a new variable (i.e., doesn't equal V or occur in P or E)

What This Means

- We start with precondition P
- After assignment, V has the value that E had (with old V replaced by v)
- The old properties still hold (with old V replaced by v)
- We use existential quantification to "remember" the old value

Example of the Forwards Axiom

Example (Forwards Assignment)

$$\vdash \{X = 1\}\ X := X + 1\ \{\exists v.\ X = X + 1[v/X] \land X = 1[v/X]\}$$

Simplifying the Postcondition

$$\vdash \{X = 1\}\ X := X + 1\ \{\exists v.\ X = X + 1[v/X] \land X = 1[v/X]\}$$
$$\vdash \{X = 1\}\ X := X + 1\ \{\exists v.\ X = v + 1 \land v = 1\}$$
$$\vdash \{X = 1\}\ X := X + 1\ \{\exists v.\ X = 1 + 1 \land v = 1\}$$
$$\vdash \{X = 1\}\ X := X + 1\ \{X = 1 + 1 \land \exists v.\ v = 1\}$$
$$\vdash \{X = 1\}\ X := X + 1\ \{X = 2 \land \mathbf{T}\}$$
$$\vdash \{X = 1\}\ X := X + 1\ \{X = 2\}$$

Comparing Forward and Backward Axioms

Key Observation

The forwards axiom is equivalent to the standard (backwards) one but harder to use

Backwards (Hoare)

$$\vdash \{Q[E/V]\}\ V := E\ \{Q\}$$

- Direct: substitute in precondition
- Natural for verification
- No existential quantifiers

Forwards (Floyd)

$$\vdash \{P\}\ V := E\ \{\exists v.\ V = E[v/V] \wedge P[v$$

- Requires existential elimination
- More complex postconditions
- Natural for symbolic execution

Why Have Two Forms?

Different Use Cases

- **Backwards (Hoare)**: Better for *verification*
  - Start with desired postcondition
  - Work backwards to find required precondition
  - Natural for proving programs meet specifications
- **Forwards (Floyd)**: Better for *analysis*
  - Start with known precondition
  - Work forwards to compute postcondition
  - Natural for symbolic execution and program analysis

In Practice

Most verification systems use Hoare's backwards form because:

- Simpler to work with (no existential quantifiers)
- More direct for common verification tasks
- Easier to automate

Precondition Strengthening

Recall

Recall that

$$\frac{\vdash S_1, \ldots, \vdash S_n}{\vdash S}$$

means $\vdash S$ can be deduced from $\vdash S_1, \ldots, \vdash S_n$

Precondition Strengthening

The Rule

Using this notation, the rule of **precondition strengthening** is:

$$\frac{\vdash P \Rightarrow P', \quad \vdash \{P'\}\ C\ \{Q\}}{\vdash \{P\}\ C\ \{Q\}}$$

Note

The two hypotheses are different kinds of judgements:

- $\vdash P \Rightarrow P'$ is a mathematical/logical judgement
- $\vdash \{P'\}\ C\ \{Q\}$ is a program correctness judgement

Understanding Precondition Strengthening

What Does This Rule Mean?

- If P implies P' (i.e., P is stronger than P')
- And we know that $\{P'\}\ C\ \{Q\}$ holds
- Then $\{P\}\ C\ \{Q\}$ also holds

Intuition

- A stronger precondition gives us more information
- If the program works correctly with less information (P')
- It will certainly work with more information (P)
- "Demanding more from the input never hurts"

Understanding Precondition Strengthening

Example (Simple Example)

- Know: $\vdash \{x > 0\}\ y := x\ \{y > 0\}$
- Have: $x = 5 \Rightarrow x > 0$
- Conclude: $\vdash \{x = 5\}\ y := x\ \{y > 0\}$

Postcondition Weakening

The Dual Rule

Just as the previous rule allows the precondition of a partial correctness specification to be strengthened, the following one allows us to weaken the postcondition

Postcondition Weakening Rule

$$\frac{\vdash \{P\}\ C\ \{Q'\}, \quad \vdash Q' \Rightarrow Q}{\vdash \{P\}\ C\ \{Q\}}$$

Understanding Postcondition Weakening

What Does This Rule Mean?

- If we can establish $\{P\}\ C\ \{Q'\}$
- And Q' implies Q (i.e., Q' is stronger than Q)
- Then $\{P\}\ C\ \{Q\}$ also holds

Intuition

- If the program establishes a strong property (Q')
- It automatically establishes any weaker property (Q)
- "Promising less in the output is always safe"

Example (Simple Example)

- Know: $\vdash \{x = 5\}\ y := x + 1\ \{y = 6\}$
- Have: $y = 6 \Rightarrow y > 0$
- Conclude: $\vdash \{x = 5\}\ y := x + 1\ \{y > 0\}$

The Rule of Consequence

Combining Both Rules

Often we use both precondition strengthening and postcondition weakening together. This gives us the general **rule of consequence**:

$$\frac{\vdash P \Rightarrow P', \quad \vdash \{P'\}\ C\ \{Q'\}, \quad \vdash Q' \Rightarrow Q}{\vdash \{P\}\ C\ \{Q\}}$$

When to Use

This rule is essential for:

- Adapting existing proofs to new situations
- Simplifying complex preconditions or postconditions
- Connecting different parts of a larger proof

Example: Using the Rule of Consequence

Example (Complete Example)

Want to prove: $\vdash \{x = 10 \wedge y = 5\}\ z := x - y\ \{z > 0\}$

1. By assignment axiom:

$$\vdash \{x - y > 0\}\ z := x - y\ \{z > 0\}$$

2. We need to show: $x = 10 \wedge y = 5 \Rightarrow x - y > 0$
   - If $x = 10$ and $y = 5$, then $x - y = 5$
   - And $5 > 0$ is true
3. By precondition strengthening:

$$\vdash \{x = 10 \wedge y = 5\}\ z := x - y\ \{z > 0\}$$

Why These Rules Matter

Practical Importance

The rules of consequence are crucial because:

- Real programs rarely have specifications that match axioms exactly
- We need to adapt and combine different proof rules
- They allow modular reasoning about programs

Key Insight

These rules formalize the intuition that:

- **Preconditions**: "If it works with less, it works with more"
- **Postconditions**: "If it achieves more, it achieves less"

Why These Rules Matter

Remember

The direction matters:

- Preconditions can be *strengthened* (made more specific)
- Postconditions can be *weakened* (made more general)

An Example Formal Proof

A Little Formal Proof

Here is a little formal proof:

1. $\vdash \{R = X \wedge 0 = 0\}\ Q := 0\ \{R = X \wedge Q = 0\}$ By the assignment axiom
2. $\vdash R = X \Rightarrow R = X \wedge 0 = 0$ By pure logic
3. $\vdash \{R = X\}\ Q := 0\ \{R = X \wedge Q = 0\}$ By precondition strengthening
4. $\vdash R = X \wedge Q = 0 \Rightarrow R = X + (Y \times Q)$ By laws of arithmetic
5. $\vdash \{R = X\}\ Q := 0\ \{R = X + (Y \times Q)\}$ By postcondition weakening

Note

The rules precondition strengthening and postcondition weakening are sometimes called the *rules of consequence*

Analyzing the Example Proof

What This Proof Shows

We proved: $\vdash \{R = X\}\ Q := 0\ \{R = X + (Y \times Q)\}$

- Starting with $R = X$
- After setting Q to 0
- We have $R = X + (Y \times 0) = X$

Key Steps

1. Started with assignment axiom for $Q := 0$
2. Strengthened precondition from $R = X \wedge 0 = 0$ to just $R = X$
3. Weakened postcondition using arithmetic ($Y \times 0 = 0$)

Lesson

Even simple proofs often require the rules of consequence to connect axioms with desired specifications

The Sequencing Rule

Syntax and Semantics

- **Syntax**: $C_1; \cdots ; C_n$
- **Semantics**: the commands $C_1, \cdots, C_n$ are executed in that order
- **Example**: $R := X;\ X := Y;\ Y := R$
  - The values of X and Y are swapped using R as a temporary variable
  - Note *side effect*: value of R changed to the old value of X

The Sequencing Rule

$$\frac{\vdash \{P\}\ C_1\ \{Q\}, \quad \vdash \{Q\}\ C_2\ \{R\}}{\vdash \{P\}\ C_1; C_2\ \{R\}}$$

Understanding the Sequencing Rule

What the Rule Says

- If C_1 transforms state from P to Q
- And C_2 transforms state from Q to R
- Then $C_1; C_2$ transforms state from P to R

The Middle Condition

- Q acts as a "glue" between the two commands
- It must be the postcondition of C_1
- And the precondition of C_2
- Finding the right Q is often the key to sequencing proofs

Generalization

For n commands: need $n - 1$ intermediate conditions

$$\{P\}\ C_1\ \{Q_1\}\ C_2\ \{Q_2\}\ \cdots\ C_{n-1}\ \{Q_{n-1}\}\ C_n\ \{R\}$$

Example Proof: Variable Swap

Goal

Prove the variable swap works correctly

Example: By the assignment axiom:

$$
\begin{array}{rl}
\text{(i)} & \vdash \{X = x \wedge Y = y\}\ R := X\ \{R = x \wedge Y = y\} \\
\text{(ii)} & \vdash \{R = x \wedge Y = y\}\ X := Y\ \{R = x \wedge X = y\} \\
\text{(iii)} & \vdash \{R = x \wedge X = y\}\ Y := R\ \{Y = x \wedge X = y\}
\end{array}
$$

Hence by (i), (ii) and the sequencing rule:

$$
\text{(iv)} \quad \vdash \{X = x \wedge Y = y\}\ R := X;\ X := Y\ \{R = x \wedge X = y\}
$$

Hence by (iv) and (iii) and the sequencing rule:

$$
\text{(v)} \quad \vdash \{X = x \wedge Y = y\}\ R := X;\ X := Y;\ Y := R\ \{Y = x \wedge X = y\}
$$

Breaking Down the Swap Proof

Step-by-Step Analysis

Starting with $X = x$ and $Y = y$:

1. After $R := X$: we have $R = x$, $X = x$, $Y = y$
2. After $X := Y$: we have $R = x$, $X = y$, $Y = y$
3. After $Y := R$: we have $R = x$, $X = y$, $Y = x$

Final result: X and Y are swapped!

Key Observation

- Each intermediate assertion captures the exact state
- We track all variables, including the temporary R
- The proof is compositional: we prove each step separately

Note on Auxiliary Variables

The lowercase x and y are auxiliary variables that remember the initial values

Conditionals

Syntax and Semantics

- **Syntax**: `IF` S `THEN` C_1 `ELSE` C_2
- **Semantics**:
  - If the statement S is true in the current state, then C_1 is executed
  - If S is false, then C_2 is executed
- **Example**: `IF X<Y THEN MAX:=Y ELSE MAX:=X`
  - The value of the variable MAX is set to the maximum of the values of X and Y

The Conditional Rule

The Conditional Rule

$$\frac{\vdash \{P \wedge S\}\ C_1\ \{Q\}, \quad \vdash \{P \wedge \neg S\}\ C_2\ \{Q\}}{\vdash \{P\}\ \texttt{IF}\ S\ \texttt{THEN}\ C_1\ \texttt{ELSE}\ C_2\ \{Q\}}$$

Understanding the Rule

- We need to prove two things:
  - When S is true, C_1 transforms $P \wedge S$ to Q
  - When S is false, C_2 transforms $P \wedge \neg S$ to Q
- Both branches must establish the same postcondition Q
- The precondition P is strengthened by the branch condition

Example: Finding the Maximum

Goal

Prove:

$\vdash \{\mathbf{T}\}$ `IF X≥Y THEN MAX:=X ELSE MAX:=Y` $\{MAX = \max(X, Y)\}$

Example: Finding the Maximum

Step 1: Logical Facts

From Assignment Axiom + Precondition Strengthening:

- $\vdash (X \geq Y \Rightarrow X = \max(X, Y)) \wedge (\neg(X \geq Y) \Rightarrow Y = \max(X, Y))$

Step 2: Prove Each Branch

It follows that:

- $\vdash \{\mathbf{T} \wedge X \geq Y\}\ MAX := X\ \{MAX = \max(X, Y)\}$
- $\vdash \{\mathbf{T} \wedge \neg(X \geq Y)\}\ MAX := Y\ \{MAX = \max(X, Y)\}$

Step 3: Apply Conditional Rule

Then by the conditional rule:

$$\vdash \{\mathbf{T}\}\ \texttt{IF X}{\geq}\texttt{Y THEN MAX:=X ELSE MAX:=Y}\ \{MAX = \max(X, Y)\}$$

Key Points about Conditionals

Important Observations

- Both branches must end in the same postcondition
- The branch condition provides extra information in each case
- We can use this extra information to prove different things in each branch

Key Points about Conditionals

Common Pattern

When proving conditional statements:

1. Identify what you know in each branch ($P \wedge S$ vs $P \wedge \neg S$)
2. Use assignment axiom for each branch separately
3. Apply precondition strengthening if needed
4. Combine using the conditional rule

Note

The conditional rule requires the same postcondition Q for both branches. If branches naturally lead to different postconditions, you may need to weaken them to a common Q.

WHILE-commands

Syntax and Semantics

- **Syntax**: `WHILE` S `DO` C
- **Semantics**:
  - If the statement S is true in the current state, then C is executed and the WHILE-command is repeated
  - If S is false, then nothing is done
  - Thus C is repeatedly executed until the value of S becomes false
  - If S never becomes false, then the execution of the command never terminates

Example (Simple WHILE Loop)

`WHILE` $\neg(X = 0)$ `DO X:= X-2`

- If the value of X is non-zero, then its value is decreased by 2 and then the process is repeated
- This WHILE-command will terminate (with X having value 0) if the value of X is an even non-negative number

The Challenge of WHILE Loops

Why WHILE Loops are Difficult

- Unlike sequence and conditionals, we don't know how many times the loop will execute
- We need to reason about *all possible* number of iterations
- The loop might not terminate at all!
- We need a way to capture what stays true throughout the loop

The Key Insight: Invariants

- An **invariant** is a property that remains true before and after each iteration
- If we can find an appropriate invariant, we can reason about the loop
- The invariant captures the "essence" of what the loop does

Invariants

Definition

Suppose $\vdash \{P \wedge S\}\ C\ \{P\}$

P is said to be an *invariant of C whenever S holds*

The WHILE-rule Intuition

The WHILE-rule says that:

- *if* P is an invariant of the body of a WHILE-command whenever the test condition holds
- *then* P is an invariant of the whole WHILE-command

In Other Words

- If executing C *once* preserves the truth of P
- Then executing C *any number of times* also preserves the truth of P

After Termination

What Happens When the Loop Exits?

The WHILE-rule also expresses the fact that after a WHILE-command has terminated, the test must be false

- Otherwise, it wouldn't have terminated
- So we know both:
  - The invariant P still holds
  - The test condition S is false (i.e., $\neg S$ is true)

The Power of Invariants

This gives us a powerful way to reason about loops:

1. Find an invariant P that captures the essential property
2. Prove that P is preserved by the loop body when S is true
3. Conclude that after the loop, we have $P \wedge \neg S$

The WHILE-Rule

The WHILE-rule

$$\frac{\vdash \{P \wedge S\}\ C\ \{P\}}{\vdash \{P\}\ \texttt{WHILE}\ S\ \texttt{DO}\ C\ \{P \wedge \neg S\}}$$

Understanding the Rule

- **Premise**: If P and S are both true, then after executing C, P is still true
- **Conclusion**: Starting with P true, after the WHILE loop, P is still true AND S is false
- The invariant P is maintained throughout all iterations
- When the loop exits, we additionally know that S is false

Example: Integer Division

Goal

Prove that the following computes integer division:

Example (Division by Repeated Subtraction)

It is easy to show:

$$\vdash \{X = R+(Y\times Q)\wedge Y \leq R\}\ R := R-Y;\ Q := Q+1\ \{X = R+(Y\times Q)\}$$

Hence by the WHILE-rule with $P =$ '$X = R + (Y \times Q)$' and $S =$ '$Y \leq R$':

$$\begin{aligned} &\vdash \{X = R + (Y \times Q)\} \\ &\quad \mathtt{WHILE}\ Y \leq R\ \mathtt{DO} \\ &\quad (R := R - Y;\ Q := Q + 1) \\ &\{X = R + (Y \times Q) \wedge \neg(Y \leq R)\} \end{aligned}$$

Analyzing the Division Example

The Invariant

$P : X = R + (Y \times Q)$ captures the relationship between:

- X: the original dividend
- R: the current remainder
- Y: the divisor
- Q: the quotient being computed

Why This Works

- Initially: $R = X$ and $Q = 0$, so $X = R + (Y \times 0) = R$
- Each iteration: We subtract Y from R and add 1 to Q
- The invariant $X = R + (Y \times Q)$ is preserved
- When done: $\neg(Y \leq R)$ means $R < Y$
- So we have: $X = R + (Y \times Q)$ with $0 \leq R < Y$
- This is exactly the definition of integer division!

Finding Good Invariants

The Art of Finding Invariants

Finding the right invariant is often the hardest part:

- It must be true initially (before the loop starts)
- It must be preserved by each iteration
- Combined with the negated test, it must imply the desired postcondition

Common Patterns

- **Accumulation**: Invariant tracks partial results (like sum so far)
- **Bounds**: Invariant maintains bounds on variables
- **Relationships**: Invariant preserves relationships between variables
- **Progress**: Invariant shows we're making progress toward goal

Remember

The invariant doesn't say what changes—it says what stays the same!

Example: Complete Division Program

From the Previous Slide

$$\begin{aligned} \vdash \{X = R + (Y \times Q)\} \\ \texttt{WHILE}\ \ Y \leq R\ \ \texttt{DO} \\ (R := R - Y;\ Q := Q + 1) \\ \{X = R + (Y \times Q) \wedge \neg(Y \leq R)\} \end{aligned}$$

Setting Up the Division

It is easy to deduce that:

$$\vdash \{\mathbf{T}\}\ R := X;\ Q := 0\ \{X = R + (Y \times Q)\}$$

Example: Complete Division Program

Complete Program

Hence by the sequencing rule and postcondition weakening:

$$
\begin{aligned}
&\vdash \{\mathbf{T}\}\\
&R := X;\\
&Q := 0;\\
&\texttt{WHILE } Y \leq R \texttt{ DO}\\
&(R := R - Y;\ Q := Q + 1)\\
&\{R < Y \wedge X = R + (Y \times Q)\}
\end{aligned}
$$

Summary

What We Have Given

- A notation for specifying what a program does
- A way of proving that it meets its specification

Next Topics

Now we look at ways of finding proofs and organizing them:

- Finding invariants
- Derived rules
- Backwards proofs
- Annotating programs prior to proof

Automation

Then we see how to automate program verification:

- The automation mechanizes some of these ideas

How Does One Find an Invariant?

The WHILE-rule

$$\frac{\vdash \{P \wedge S\}\ C\ \{P\}}{\vdash \{P\}\ \texttt{WHILE}\ \ S\ \ \texttt{DO}\ \ C\ \{P \wedge \neg S\}}$$

Look at the Facts

- Invariant P must hold initially
- With the negated test $\neg S$ the invariant P must establish the result
- When the test S holds, the body must leave the invariant P unchanged

How Does One Find an Invariant?

Think About How the Loop Works

The invariant should say that:

- What *has been done so far* together with what *remains to be done*
- Holds *at each iteration* of the loop
- And gives *the desired result* when the loop terminates

Example: Factorial Program

Consider a Factorial Program

$$\{X = n \wedge Y = 1\}$$
$$\texttt{WHILE } X \neq 0 \texttt{ DO}$$
$$(Y := Y \times X;\ X := X - 1)$$
$$\{X = 0 \wedge Y = n!\}$$

Look at the Facts

- Initially $X = n$ and $Y = 1$
- Finally $X = 0$ and $Y = n!$
- On each loop Y is increased and X is decreased

Example: Factorial Program

Think How the Loop Works

- Y holds the result so far
- $X!$ is what remains to be computed
- $n!$ is the desired result

The Invariant

The invariant is $X! \times Y = n!$

- 'stuff to be done' $\times$ 'result so far' $=$ 'desired result'
- Decrease in X combines with increase in Y to make invariant

Related Example

Another Factorial-like Program

$$
\begin{aligned}
&\{X = 0 \land Y = 1\} \\
\texttt{WHILE } X < N \texttt{ DO } &(X := X + 1;\ Y := Y \times X) \\
&\{Y = N!\}
\end{aligned}
$$

Look at the Facts

- Initially $X = 0$ and $Y = 1$
- Finally $X = N$ and $Y = N!$
- On each iteration both X and Y increase: X by 1 and Y by X

First Attempt

- An invariant is $Y = X!$
- At end need $Y = N!$, but WHILE-rule only gives $\neg(X < N)$

Related Example

Ah Ha!

Invariant needed: $Y = X! \wedge X \leq N$

Why This Works

- At end: $X \leq N \wedge \neg(X < N) \Rightarrow X = N$
- Often need to strengthen invariants to get them to work
- Typical to add stuff to 'carry along' like $X \leq N$

Conjunction and Disjunction

Specification Conjunction and Disjunction

Specification conjunction

$$\frac{\vdash \{P_1\}\ C\ \{Q_1\}, \quad \vdash \{P_2\}\ C\ \{Q_2\}}{\vdash \{P_1 \wedge P_2\}\ C\ \{Q_1 \wedge Q_2\}}$$

Specification disjunction

$$\frac{\vdash \{P_1\}\ C\ \{Q_1\}, \quad \vdash \{P_2\}\ C\ \{Q_2\}}{\vdash \{P_1 \vee P_2\}\ C\ \{Q_1 \vee Q_2\}}$$

Use of These Rules

These rules are useful for splitting a proof into independent bits:

- They enable $\vdash \{P\}\ C\ \{Q_1 \wedge Q_2\}$ to be proved by proving separately that both $\vdash \{P\}\ C\ \{Q_1\}$ and also that $\vdash \{P\}\ C\ \{Q_2\}$

Theoretical vs Practical Considerations

Theoretical Status

Any proof with these rules could be done without using them:

- i.e., they are theoretically redundant (proof omitted)
- However, useful in practice

Why These Rules Matter in Practice

- They make proofs more modular
- Allow separate verification of different properties
- Can simplify complex specifications
- Make proof structure clearer

Derived Rules for Finding Proofs

The Goal-Directed Approach

Suppose the goal is to prove $\{Precondition\}\ Command\ \{Postcondition\}$
If there were a rule of the form:

$$\frac{\vdash H_1, \ldots, \vdash H_n}{\vdash \{P\}\ C\ \{Q\}}$$

then we could instantiate:

- $P \mapsto Precondition$, $C \mapsto Command$, $Q \mapsto Postcondition$
- to get instances of $H_1, \ldots, H_n$ as subgoals

The Key Insight

- Some rules are already in this form (e.g., the sequencing rule)
- We will derive rules of this form for all commands
- Then we use these derived rules for mechanizing Hoare Logic proofs

Understanding Goal-Directed Proof

What is Goal-Directed Proof?

Instead of building proofs from axioms up (forward), we:

1. Start with what we want to prove (the goal)
2. Find a rule whose conclusion matches our goal
3. The premises of that rule become our new subgoals
4. Repeat until we reach axioms or known facts

Example (Sequencing Example)

To prove $\{P\}\ C_1; C_2\ \{Q\}$:

- Apply sequencing rule backwards
- New subgoals: find R such that:
  - $\vdash \{P\}\ C_1\ \{R\}$
  - $\vdash \{R\}\ C_2\ \{Q\}$

Establishing Derived Rules for All Commands

We will establish derived rules for all commands:

$$\frac{\ldots}{\vdash \{P\}\ V := E\ \{Q\}}$$

$$\frac{\ldots}{\vdash \{P\}\ C_1; C_2\ \{Q\}}$$

$$\frac{\ldots}{\vdash \{P\}\ \texttt{IF}\ S\ \texttt{THEN}\ C_1\ \texttt{ELSE}\ C_2\ \{Q\}}$$

$$\frac{\ldots}{\vdash \{P\}\ \texttt{WHILE}\ S\ \texttt{DO}\ C\ \{Q\}}$$

Derived Rules

Purpose

These support 'backwards proof' starting from a goal $\{P\}\ C\ \{Q\}$

The Derived Assignment Rule

An Example Proof

Let's revisit our earlier proof from Section 12:

1. $\vdash \{R = X \wedge 0 = 0\}\ Q := 0\ \{R = X \wedge Q = 0\}$ By assignment axiom
2. $\vdash R = X \Rightarrow R = X \wedge 0 = 0$ By pure logic
3. $\vdash \{R = X\}\ Q := 0\ \{R = X \wedge Q = 0\}$ By precondition strengthening

Generalizing to a Proof Schema

We can generalize this pattern:

1. $\vdash \{Q[E/V]\}\ V := E\ \{Q\}$ By assignment axiom
2. $\vdash P \Rightarrow Q[E/V]$ By assumption
3. $\vdash \{P\}\ V := E\ \{Q\}$ By precondition strengthening

The Derived Assignment Rule

The Rule

This proof schema justifies:

Derived Assignment Rule

$$\frac{\vdash P \Rightarrow Q[E/V]}{\vdash \{P\}\ V := E\ \{Q\}}$$

Key Insight

- $Q[E/V]$ is the **weakest liberal precondition** $wlp(V := E, Q)$
- This is the weakest condition that guarantees Q after $V := E$
- Links back to our discussion of substitution in Section 9

Understanding the Derived Assignment Rule

Why This Rule is Powerful

- **Goal-directed**: Start with desired postcondition Q
- **Systematic**: Compute $Q[E/V]$ mechanically
- **Complete**: Can derive any valid assignment triple

Example (Using the Rule)

Original proof required 3 steps:

1. $\vdash R = X \Rightarrow R = X \wedge 0 = 0$ By pure logic
2. $\vdash \{R = X\}\ Q := 0\ \{R = X \wedge Q = 0\}$ By derived assignment

Now only 2 steps! We saved one step by using the derived rule.

Why Do We Need Derived Rules?

The Problem with Forward Proof

Using just the assignment axiom:

- We must guess the right precondition
- Often requires multiple attempts
- May need complex logical manipulations
- Hard to mechanize or automate

The Solution: Work Backwards

Derived rules let us:

- Start with what we want to prove (the goal)
- Systematically compute what we need
- No guessing required
- Can be automated by computers

What is Weakest Liberal Precondition?

The Intuition

Given: $\{?\}\ V := E\ \{Q\}$

We ask: "What must be true before the assignment so that Q is true after?"

Answer: Whatever Q says about V, must have been true about E before!

Example (Simple Example)

- Want: $\{?\}\ X := X + 1\ \{X > 0\}$
- After: X must be greater than 0
- Before: $X + 1$ must be greater than 0
- So: $X > -1$ before the assignment
- We compute: $(X > 0)[X + 1/X] = X + 1 > 0 = X > -1$

Computing WLP Step by Step

The Substitution Process

$Q[E/V]$ means: Replace every occurrence of V in Q with E

Example (More Examples)

Postcondition Q	**Assignment**	**WLP:** $Q[E/V]$
$Y = 5$	$Y := X + 2$	$(X + 2) = 5$, i.e., $X = 3$
$X = Y$	$X := Y + 1$	$(Y + 1) = Y$, i.e., $\mathbf{F}$
$X^2 > 0$	$X := Y - 3$	$(Y - 3)^2 > 0$, i.e., $Y \neq 3$

Key Insight

The wlp is exactly what the assignment axiom gives us - but now we can compute it mechanically!

Array Assignment - Corrected

Example (Array Assignment)

Goal: $\{?\}\ A[i] := v\ \{A[j] = w\}$

The substitution for arrays is tricky:

- $A[j]$ after assignment equals:
  - v if $i = j$ (we just assigned it!)
  - $A[j]$ if $i \neq j$ (unchanged)

Therefore, the wlp is:

- If we can prove $i = j$: need $v = w$
- If we can prove $i \neq j$: need $A[j] = w$
- In general: $(i = j \Rightarrow v = w) \wedge (i \neq j \Rightarrow A[j] = w)$

Why This Matters

This connects to our discussion of aliasing (Section 10) - array indices might refer to the same location!

Why Backwards Proof?

Advantages of Working Backwards

- **Goal-focused**: Always know what you're trying to prove
- **Systematic**: Each command type has a specific strategy
- **Mechanizable**: Can be automated more easily
- **Natural**: Matches how humans often think about proofs

The Process

1. Look at the command structure
2. Apply the corresponding derived rule
3. Generate simpler subgoals
4. Continue until reaching assignment axioms