CS 456

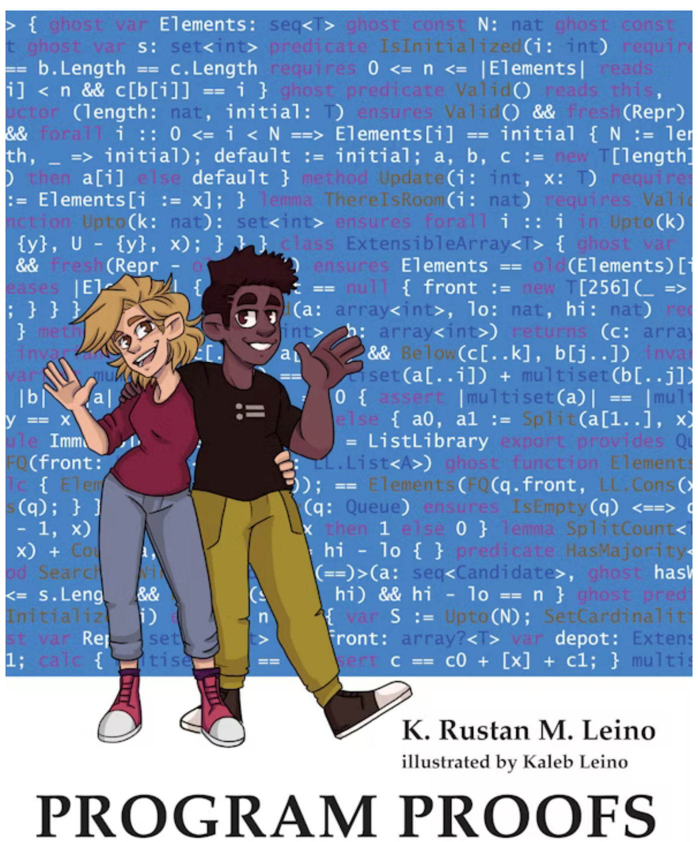
Programming Languages
Fall 2024

Week 12
Axiomatic Semantics and Hoare Logic

Homework

Install Dafny:

see www.dafny.org



Semantics

- Operational Semantics
  - ★ Simple abstract machine shows *how* to evaluate expression
- Denotational Semantics
  - ★ Map language construct to mathematical domains (e.g., sets) to describe what expressions mean

Can Prove:

Metatheoretic Properties

- Determinism of Evaluation
- Soundness of Program Transformations
- Program Equivalence

Axiomatic Semantics

Axiomatic Semantics

- Meaning given by proof rules
- Useful for reasoning about properties of *specific* programs

- Step 1: Define a language of claims
- Step 2: Define a set of rules (axioms) to build proofs of claims
- Step 3: Verify specific programs

Assertions

- Not unusual to see pre- and post-conditions in code comments:

```
/*Precondition: 0 <= i <= A.length
  Postcondition: returns A[i]*/
public int get(int i) {
  return A[i]
}
```

- Step 1A: Define a language of assertions to capture these sorts of claims

Assertions

- Step 1A: Define a language of *assertions* to capture these claims about states
- Examples:
  - The value of the variable X is greater than 4
  - The variable Y holds an even number
  - The value of X is half of the value of Z
- Formalize claims in some logic with variables
  - Proof Assistant (Coq, Isabelle, Agda, …)
  - smt-lib (many automated verifiers)
  - First-order logic: $\forall$, $\exists$, $\wedge$, $\rightarrow$, $X = Y$

Hoare Triple

- Step 1B: Define a judgement for claims about programs involving assertions
- Partial Correctness Triple:

$$\{P\}\ c\ \{Q\}$$

If we start in a state satisfying P

And c terminates in a state,

then that final state satisfies Q

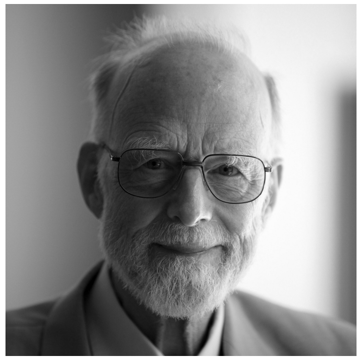
Hoare Triple

An Axiomatic Basis for Computer Programming

C. A. R. Hoare
The Queen's University of Belfast, Northern Ireland*

In this paper an attempt is made to explore the logical foundations of computer programming by use of techniques which were first applied in the study of geometry and have later been extended to other branches of mathematics. This involves the elucidation of sets of axioms and rules of inference which can be used in proofs of the properties of computer programs. Examples are given of such axioms and rules, and a formal proof of a simple theorem is displayed. Finally, it is argued that important advantages, both theoretical and practical, may follow from a pursuance of these topics.

KEY WORDS AND PHRASES: axiomatic method, theory of programming' proofs of programs, formal language definition, programming language design, machine-independent programming, program documentation
CR CATEGORY: 4.0, 4.21, 4.22, 5.20, 5.21, 5.23, 5.24



of axioms it is possible to deduce such simple theorems as:

$$x = x + y \times 0$$

$$y \leqslant r \supset r + y \times q = (r - y) + y \times (1 + q)$$

The proof of the second of these is:

A5 $(r - y) + y \times (1 + q)$

$= (r - y) + (y \times 1 + y \times q)$

A9 $= (r - y) + (y + y \times q)$

A3 $= ((r - y) + y) + y \times q$

A6 $= r + y \times q$ provided $y \leqslant r$

The axioms A1 to A9 are, of course, true of the traditional infinite set of integers in mathematics. However, they are also true of the finite sets of "integers" which are manipulated by computers provided that they are confined to *nonnegative* numbers. Their truth is independent of the size of the set; furthermore, it is largely independent of the choice of technique applied in the event of "overflow"; for example:

(1) Strict interpretation: the result of an overflowing operation does not exist; when overflow occurs, the offending program never completes its operation. Note that in this case, the equalities of A1 to A9 are strict, in the sense that both sides exist or fail to exist together.

(2) Firm boundary: the result of an overflowing operation is taken as the maximum value represented.

C. A. R. Hoare. 1969. An axiomatic basis for computer programming. *Commun. ACM* 12, 10 (Oct. 1969), 576–580.

Hoare Triple

- Step 1B: Define a judgement for claims about programs involving assertions
- Partial Correctness Triple:
- $$\{P\}\ c\ \{Q\}$$
- Total Correctness Triple:
- $$[P]\ c\ [Q]$$
- A triple that makes a true claim is said to be *valid*

Hoare Triples

What *should* these mean:

```
{True} c {X = 5}
∀m. {X = m} c {X = m + 5}
[X <= Y] c [Y <= X]
```

Concept Check

Which of these should be valid?

```
{X = 2} X := X + 1 {X = 3}
{X = 2} X := 5; Y := 3 {X = 5}
{False} skip {True}
[Y = 5] X := Y + 3 [X = 5]
{True} while true do SKIP end {False}
[True] while true do SKIP end [False]
[True] while true do SKIP end [True]
```

Axiomatic Semantics

- Step 1: Define a language of claims
- Step 2: Define a set of rules (axioms) to build proofs of claims
- Step 3: Verify specific programs

Imp Assertions

One assertion language for Imp commands is:

$$X \in \mathtt{Id} \qquad\qquad N \in \mathbb{N}$$

$$A ::= N \mid A + A \mid A - A \mid A * A \mid X$$

$$P, Q ::= T \mid \perp \mid A < A \mid A = A \mid P \wedge Q \mid P \vee Q \mid \neg P$$

Examples Assertions:

The value of the variable X is greater than 4
The variable Y holds an even number
The value of X is half of the value of Z

Satisfiability

- We define a semantics for this language to identify when a state σ *satisfies* an assertion P:

$$\sigma \models P$$

$$\frac{}{\sigma \models T}$$

$$\frac{\sigma, a_1 \Downarrow v_1 \qquad \sigma, a_2 \Downarrow v_2 \qquad v_1 <_{\mathbb{N}} v_2}{\sigma \models a_1 < a_2}$$

$$\frac{\sigma, a_1 \Downarrow v_1 \qquad \sigma, a_2 \Downarrow v_2 \qquad v_1 =_{\mathbb{N}} v_2}{\sigma \models a_1 = a_2}$$

Satisfability

We define a semantics for this language to identify when a state σ *satisfies* an assertion P:

$$\boxed{\sigma \models P}$$

$$\frac{\sigma \models P \qquad \sigma \models Q}{\sigma \models P \wedge Q}$$

$$\frac{\sigma \models P}{\sigma \models P \vee Q} \qquad \frac{\sigma \models Q}{\sigma \models P \vee Q} \qquad \frac{\sigma \not\models P}{\sigma \models \neg P}$$

Validity

We can now precisely define whe
a partial Hoare Triple is valid:

$\sigma \models P$

VALIDIT

$$\{P\}\ c\ \{Q\} \equiv$$
$$\forall \sigma.\ \sigma \models P \rightarrow$$

If we start in a state satisfying P

$$\forall \sigma'.\ \sigma, c \Downarrow \sigma'$$

And c terminates

$$\rightarrow$$
$$\sigma' \models Q$$

then that final state satisfies Q

Proving Validity

- That gives us the first part of axiomatic semantics
  - Step 1: Define a language of claims
- How to prove that {P} c {Q} is valid?
  - Could reason directly about the semantics of c
  - Step 2: Define a set of rules (axioms) to build proofs of claims without reasoning directly about states and executions

$$\boxed{\vdash \{P\}\ c\ \{Q\}}$$

Proof Rules

How to prove that {P} c {Q} is valid?

- Define a set of rules (axioms) to build proofs of claims without reasoning directly about states and executions

$$\boxed{\vdash \{P\}\ c\ \{Q\}}$$

Hoare Skip

Use our intuition about what we want to be able to prove to guide definition of rules

$$\{P\}\, c\, \{Q\} \equiv$$
$$\forall \sigma.\ \sigma \models P \rightarrow\ \ \forall \sigma'.\ \sigma, c \Downarrow \sigma' \rightarrow \sigma' \models Q$$

Hoare Skip?

$\{?\}\ \mathsf{skip}\ \{Q\} \equiv$

$\forall \sigma.\ \sigma \models ? \rightarrow\ \forall \sigma'.\ \sigma, \mathsf{skip} \Downarrow \sigma' \rightarrow \sigma' \models Q$

$$\frac{}{\vdash \{?\}\ \mathsf{skip}\ \{Q\}}$$

Hoare Skip!

$\{Q\}\ \mathsf{skip}\ \{Q\} \equiv$

$\forall \sigma.\ \sigma \models Q \rightarrow\ \forall \sigma'.\ \sigma, \mathsf{skip} \Downarrow \sigma' \rightarrow \sigma' \models Q$

$$\frac{}{\vdash \{Q\}\ \mathsf{skip}\ \{Q\}}\ \textsc{HLSkip}$$

Hoare Assign?

$\{\ ??\ \}\, X := a\, \{Q\} \equiv$

$\forall\sigma.\ \sigma \models\ ??\ \rightarrow$

$\quad \forall\sigma'.\ \sigma, X{:=}a \Downarrow \sigma' \rightarrow \sigma' \models Q$

$$\frac{}{\vdash \{\ ??\ \}\, X{:=}a\, \{Q\}}$$

Hoare Assign!

$$\{[X := a]Q\}\ X := a\ \{Q\} \equiv$$

$$\forall \sigma.\ \sigma \models [X := a]Q \rightarrow$$

$$\quad \forall \sigma'.\ \sigma, X := a \Downarrow \sigma' \rightarrow \sigma' \models Q$$

$$\frac{}{\vdash \{[X := a]Q\}\ X := a\ \{Q\}}\ \textsc{HLAssign}$$

Hoare Assign^{bad}

- Why not this “forward” rule?

$$\frac{}{\vdash \{P\}\ X := a\ \{[X := a]P\}}$$

Hoare Assign!

$$\{[X{\coloneqq}a]Q\}\ X \mathrel{:\!\underline{=}} a\ \{Q\} \equiv$$

$$\forall\sigma.\ \sigma \models [X{\coloneqq}a]Q \rightarrow$$

$$\quad \forall\sigma'.\ \sigma,\ X \mathrel{:\!\underline{=}} a \Downarrow \sigma' \rightarrow \sigma' \models Q$$

$$\frac{}{\vdash \{[X{\coloneqq}a]Q\}\ X := a\ \{Q\}}\ \textsc{HLAssign}$$

Hoare Seq?

$\{\ ?\ \}\ c_1 ; c_2\ \{Q\} \equiv$

$\forall\sigma.\ \sigma \models\ ?\ \rightarrow$

$\quad \forall\sigma'.\ \sigma, c_1 ; c_2 \Downarrow \sigma' \rightarrow \sigma' \models Q$

$$\frac{}{\vdash \{\ ?\ \}\ c_1 ; c_2\ \{Q\}}$$

Hoare Seq?

$$\{\ ?\ \}\ c_1; c_2\ \{Q\} \equiv$$
$$\forall \sigma_1.\ \sigma_1 \models\ ? \rightarrow \forall \sigma_3.$$
$$(\exists \sigma_2.\ \sigma, c_1 \Downarrow \sigma_2 \land \sigma, c_2 \Downarrow \sigma_3) \rightarrow$$
$$\sigma_3 \models Q$$

$$\dfrac{}{\vdash \{\ ?\ \}\ c_1; c_2\ \{Q\}}$$

Hoare Seq?

$\{\ ?_1\ \}\ c_1; c_2\ \{Q\} \equiv$

$\forall \sigma_1.\ \sigma_1 \models\ ?_1 \rightarrow \forall \sigma_3.$

$(\exists \sigma_2.\ \sigma, c_1 \Downarrow \sigma_2 \land \sigma, c_2 \Downarrow \sigma_3) \rightarrow$

$\sigma_3 \models Q$

$$\frac{\vdash \{?_1\} c_1 \{?_2\} \quad \vdash \{?_2\} c_2 \{Q\}}{\vdash \{\ ?_1\ \}\ c_1; c_2\ \{Q\}}$$

Hoare Seq!

$\{ P \}\ c_1 ; c_2\ \{Q\} \equiv$

$\forall\sigma.\ \sigma \models P \rightarrow$

$\quad \forall\sigma'.\ \sigma, c_1 ; c_2 \Downarrow \sigma' \rightarrow \sigma' \models Q$

$$\frac{\vdash \{P\} c_1 \{R\} \quad \vdash \{R\} c_2 \{Q\}}{\vdash \{P\}\ c_1 ; c_2\ \{Q\}}\ \text{HLSEQ}$$

Hoare Seq!

$$\frac{\vdash \{P\}\, c_1 \,\{R\} \quad \vdash \{R\}\, c_2 \,\{Q\}}{\vdash \{P\}\; c_1 ; c_2 \;\{Q\}} \;\textsc{HLSeq}$$

Hoare If!

$$\frac{\vdash \{P \wedge b\}\, c_1\, \{Q\} \quad \vdash \{P \wedge \neg b\}\, c_2\, \{Q\}}{\vdash \{P\}\ \text{if}\ b\ \text{then}\ c_1\ \text{else}\ c_2\ \text{end}\ \{Q\}}$$

HLIF

Proof Rules

- What if Assertions don't align?

```
{X=2} X ≔ X + 1 {X = 3}
```

- Have rule for weakening postconditions and strengthening preconditions

$$\frac{\vdash \{P_W\}\ c\ \{Q_S\} \qquad P \rightarrow P_W \qquad Q_S \rightarrow Q}{\vdash \{P\}\ c\ \{Q\}}\ \textsc{HLConseq}$$

$$\frac{\dfrac{}{\vdash \{X+1=3\}\ X \coloneqq X + 1\ \{X = 3\}}\ \textsc{HLAssign} \qquad \dfrac{}{X=2 \rightarrow X+1=3} \qquad \dfrac{}{X=3 \rightarrow X=3}}{\vdash \{X=2\}\ X \coloneqq X + 1\ \{X = 3\}}\ \textsc{HLConseq}$$

Rule Review

$$\frac{}{\vdash \{Q[X{:=}a]\}\, X{:=}a\, \{Q\}}\ \textsc{HLAssign} \qquad \frac{}{\vdash \{Q\}\ \text{skip}\ \{Q\}}\ \textsc{HLSkip}$$

$$\frac{\vdash \{P\}\ c_1\ \{R\} \qquad \vdash \{R\}\ c_2\ \{Q\}}{\vdash \{P\}\ c_1;c_2\ \{Q\}}\ \textsc{HLSeq}$$

$$\frac{\vdash \{P \wedge b\}\ c_1\ \{Q\} \qquad \vdash \{P \wedge \neg b\}\ c_2\ \{Q\}}{\vdash \{P\}\ \textbf{if}\ b\ \textbf{then}\ c_1\ \textbf{else}\ c_2\ \{Q\}}\ \textsc{HLIf}$$

$$\frac{\vdash \{P_W\}\ c\ \{Q_S\} \qquad P \rightarrow P_W \qquad Q_S \rightarrow Q}{\vdash \{P\}\ c\ \{Q\}}\ \textsc{HLConseq}$$

Hoare While?

$\vdash \{X < 3\}$ **while** $(X < 3)$ **do** $X := X + 1$ **end** $\{X = 3\}$

$$\frac{\vdash \{?\}\ c\ \{?\}}{\vdash \{?\}\ \text{while}\ b\ \text{do}\ c\ \text{end}\ \{Q\}}$$

Hoare While?

$$\frac{\dfrac{\vdash \{X < 4\}\ X := X + 1\ \{X < 4\}}{\vdash \{X < 4\}\ \textbf{while}\ (X < 3)\ \textbf{do}\ \ X := X + 1\ \textbf{end}\ \{X < 4\}}}{\vdash \{X < 3\}\ \textbf{while}\ (X < 3)\ \textbf{do}\ \ X := X + 1\ \textbf{end}\ \{X = 3\}}$$

$$\frac{\vdash \{Q \quad\}\ c\ \{Q\}}{\vdash \{Q\}\ \text{while}\ b\ \text{do}\ c\ \text{end}\ \{Q \quad\}}$$

Hoare While?

$$\frac{\dfrac{\vdash \{X < 4 \land X < 3\}\ X := X + 1\ \{X < 4\}}{\vdash \{X < 4\}\ \textbf{while}\ (X < 3)\ \textbf{do}\ \ X := X + 1\ \textbf{end}\ \{X < 4\}}}{\vdash \{X < 3\}\ \textbf{while}\ (X < 3)\ \textbf{do}\ \ X := X + 1\ \textbf{end}\ \{X = 3\}}$$

$$\frac{\vdash \{Q \land b\}\ c\ \{Q\}}{\vdash \{Q\}\ \text{while}\ b\ \text{do}\ c\ \text{end}\ \{Q \quad\ \}}$$

Hoare While?

$$\frac{\dfrac{\vdash \{X < 4 \land X < 3\}\ X := X + 1\ \{X < 4\}}{\vdash \{X < 4\}\ \textbf{while}\ (X < 3)\ \textbf{do}\ \ X := X + 1\ \textbf{end}\ \{X < 4 \land \neg X < 3\ \}}}{\vdash \{X < 3\}\ \textbf{while}\ (X < 3)\ \textbf{do}\ \ X := X + 1\ \textbf{end}\ \{X = 3\}}$$

$$\frac{\vdash \{Q \land b\}\ c\ \{Q\}}{\vdash \{Q\}\ \text{while}\ b\ \text{do}\ c\ \text{end}\ \{Q \land \neg b\}}$$

Hoare While!

I is a *loop invariant*:

-Holds before loop

-Holds after each loop iteration

-Holds when the loop exits

$$\frac{\vdash \{I \wedge b\}\ c\ \{I\}}{\vdash \{I\}\ \text{while}\ b\ \text{do}\ c\ \text{end}\ \{I \wedge \neg b\}}\ \textsc{HLWhile}$$

Loop Invariants

Hoare Logic is a structural model-theoretic proof system

- Rules characterize a set of states consistent with the requirements imposed by the pre- and post-conditions
- Highly mechanical: intermediate states can almost always be automatically constructed
- One major exception:

$$\frac{\vdash \{I \wedge b\}\ c\ \{I\}}{\vdash \{I\}\ \textbf{while}\ b\ \textbf{do}\ c\ \textbf{end}\ \{I \wedge \neg b\}}\ \text{HLW}\small{\text{HILE}}$$

The invariant must:

- be weak enough to be implied by the precondition
- hold across each iteration
- be strong enough to imply the postcondition

Rule Review

$$\frac{}{\vdash \{Q[X:=a]\}X:=a\{Q\}}\ \textsc{HLAssign}$$

$$\frac{}{\vdash \{Q\}\ \mathrm{skip}\ \{Q\}}\ \textsc{HLSkip}$$

$$\frac{\vdash \{P\}\ c_1\ \{R\} \qquad \vdash \{R\}\ c_2\ \{Q\}}{\vdash \{P\}\ c_1;c_2\ \{Q\}}\ \textsc{HLSeq}$$

$$\frac{\vdash \{P \wedge b\}\ c_1\ \{Q\} \qquad \vdash \{P \wedge \neg b\}\ c_2\ \{Q\}}{\vdash \{P\}\ \mathbf{if}\ b\ \mathbf{then}\ c_1\ \mathbf{else}\ c_2\ \{Q\}}\ \textsc{HLIf}$$

$$\frac{\vdash \{I \wedge b\}\ c\ \{I\}}{\vdash \{I\}\ \mathbf{while}\ b\ \mathbf{do}\ c\ \mathbf{end}\ \{I \wedge \neg b\}}\ \textsc{HLWhile}$$

Hoare in Action

- Want to build proof trees:

⊢{(z - 1) - (x - 1) = p - m ∧ x <> 0 } z := z - 1; x := x - 1{(z - 1) - (x - 1) = p - m}

⊢{(z - 1) - (x - 1) = p - m } **while** x … z = p - m ∧ (x = 0)}

⊢{p=p} z := p {z = p} … {z = p - m ∧ (x = 0)}

… {z = p - m ∧ (x = 0)}

⊢{ m = m } x := m {x = m} … {z = p - m ∧ (x = 0)}

⊢{ m = m } x := m; z := p … - m ∧ (x = 0)}

⊢{ True } x := m; z := p; **while** x ≠ 0 **do** z := z - 1; x := x - 1 **end** {z = p - m }

Proof is *compositional*: it follows structure of program!

Decorated Programs

Idea: include assertions in program

$\{ \text{True} \} \rightarrow \{ m = m \}$

X := m;

$\{ X = m \} \rightarrow \{ X = m \wedge p = p \}$

Z := p;

$\{ X = m \wedge Z = p \} \rightarrow \{ Z - X = p - m \}$

while $X \neq 0$ **do**

$\{ Z - X = p - m \wedge X \neq 0 \} \rightarrow \{ (Z - 1) - (X - 1) = p - m \}$

Z := Z - 1;

$\{ Z - (X - 1) = p - m \}$

X := X - 1

$\{ Z - X = p - m \}$

end;

$\{ Z - X = p - m \wedge \neg (X \neq 0) \} \rightarrow \{ Z = p - m \}$

Decorated Programs

- Idea: include assertions in program
- If each individual command is correct, so is the program

$$
\begin{array}{l}
\{\ X = m \wedge Y = n\ \} \\
\quad X := X + Y \\
\{\ ??\ \} \\
\quad Y := X - Y \\
\{\ ??\ \} \\
\quad X := X - Y \\
\{\ X = n \wedge Y = m\ \}
\end{array}
$$

Decorated Programs

- Idea: include assertions in program
- If each individual command is correct, so is the program

$$
\begin{array}{l}
\{\, X = m \wedge Y = n \,\} \\
\quad X := X + Y \\
\{\, ?? \,\} \\
\quad Y := X - Y \\
\{\, X - Y = n \wedge Y = m \,\} \\
\quad X := X - Y \\
\{\, X = n \wedge Y = m \,\}
\end{array}
$$

Decorated Programs

- Idea: include assertions in program
- If each individual command is correct, so is the program

$$
\begin{array}{l}
\{\ X = m \wedge Y = n\ \} \\
\quad X := X + Y \\
\{X - (X - Y) = n \wedge X - Y = m\} \\
\quad Y := X - Y \\
\{\ X - Y = n \wedge Y = m\ \} \\
\quad X := X - Y \\
\{\ X = n \wedge Y = m\ \}
\end{array}
$$

Decorated Programs

- Idea: include assertions in program
- If each individual command is correct, so is the program

$$\{ X = m \wedge Y = n\} \rightarrow$$
$$\{(X + Y) - ((X + Y) - Y) = n \wedge (X + Y) - Y = m\}$$
$$X := X + Y$$
$$\{X - (X - Y) = n \wedge X - Y = m\}$$
$$Y := X - Y$$
$$\{ X - Y = n \wedge Y = m \}$$
$$X := X - Y$$
$$\{ X = n \wedge Y = m \}$$

Example


```
{{ True }}
  if X <= Y then
                                                  }}
          {{
    Z := Y - X
          {{                                      }}
  else
          {{                                      }}
    Y := X + Z
          {{                                      }}
  end
{{ Y = X + Z }}
```

Our proof rules provide a systematic way of generating intermediate assertions. The fully decorated program constitutes a proof that the program when executed in a state that satisfies the precondition, will produce a state satisfying the postcondition.

Example


```
{{ True }}
  if X <= Y then
            {{                          }}
    Z := Y - X
            {{                          }}
  else
            {{                          }}
    Y := X + Z
            {{Y = X + Z}}
  end
{{ Y = X + Z }}
```

follows from the postcondition

Example


```
{{ True }}
  if X <= Y then
            {{                          }}
    Z := Y - X
            {{Y = X + Z}}
  else
            {{                          }}
    Y := X + Z
            {{ Y = X + Z}}
  end
{{ Y = X + Z }}
```

Example


```
{{ True }}
  if X <= Y then
              {{X <= Y}}
    Z := Y - X
              {{Y = X + Z}}
  else
              {{X > Y}}
    Y := X + Z
              {{Y = X + Z}}
  end
{{ Y = X + Z }}
```

- *By If Rule*
- *But, shape of precondition in assignments does not match the shape demanded by the Assgn rule*

Example


```
{{ True }}
  if X <= Y then
                {{X <= Y}} ->
                {{Y = X + (Y - X)}}
    Z := Y - X
                {{Y = X + Z}}
  else
                {{X > Y}} ->
                {{X + Z = X + Z}}
    Y := X + Z
                {{Y = X + Z}}
  end
{{ Y = X + Z }}
```

Update precondition using rule of consequence and Assgn rule

Non-trivial implication

Proof can be constructed automatically, reasoning backwards from the postcondition

Loop Invariants

- Largely straightforward
- **Except** for loops!

```
{ X = m }
   while X ≠ 0 do
    X := X - 1;
   end
{ X = 0 }
```

Loops


```
{{ True }} -->
      {{                         }}
   X := a;
      {{                         }}
   Y := b;
      {{                         }}
   Z := 0;
      {{                         }}
   while X <> 0 && Y <> 0 do
      {{                         }}
    X := X - 1;
      {{                         }}
    Y := Y - 1;
      {{                         }}
    Z := Z + 1;
      {{                         }}
   end
{{ Z = min a b }}
```

Loops


```
{{ True }}
      {{                    }}
   X := a;
      {{                    }}
   Y := b;
      {{                    }}
   Z := 0;
      {{                    }}
   while X <> 0 && Y <> 0 do
      {{                    }}
    X := X - 1;
      {{                    }}
    Y := Y - 1;
      {{                    }}
    Z := Z + 1;
      {{                    }}
   end
{{ Z = min a b }}
```

postcondition given in terms of inputs a and b

loop invariant expresses constraints on local variables `X` and `Y`

candidate invariant:

`Z + min X Y = min a b`

Loops


```
{{ True }}
      {{  min a b = min a b  }}
   X := a;
      {{  min X b = min a b  }}
   Y := b;
      {{  min X Y = min a b  }}
   Z := 0;
      {{  Inv                }}
   while X <> 0 && Y <> 0 do
      {{ Z + 1 + min (X - 1) (Y - 1) = min a b }}
    X := X - 1;
      {{ Z + 1 + min X (Y - 1) = min a b }}
    Y := Y - 1;
      {{ Z + 1 + min X Y = min a b }}
    Z := Z + 1;
      {{ Inv  }}
   end
{{ ~(X <> 0 /\ Y <> 0) /\ Inv) }} ->
{{ Z = min a b }}
```

Inv == (Z + min X Y = min a b)

Precondition Inference


```
{{ True }} ->
      {{  min a b = min a b   }}
   X := a;
      {{  min X b = min a b   }}
   Y := b;
      {{  min X Y = min a b   }}
   Z := 0;
      {{  Inv                 }}
   while X <> 0 && Y <> 0 do
      {{ Inv /\ (X <> 0) /\ Y <> 0) }} ->
      {{ Z + 1 + min (X - 1) (Y - 1) = min a b }}
    X := X - 1;
      {{ Z + 1 + min X (Y - 1) = min a b }}
    Y := Y - 1;
      {{ Z + 1 + min X Y = min a b }}
    Z := Z + 1;
      {{ Inv  }}
   end
{{ ~(X <> 0 /\ Y <> 0) /\ Inv) }} ->
{{ Z = min a b }}
```

This style of proof construction is known as weakest precondition inference

Identify a precondition that satisfies the largest set of states that still enable verification of the postcondition

Can automate this inference once we know the loop invariant

Concept Check


```
{{ ? }}  skip  {{ X = 5 }}                        X = 5

{{ ? }}  X := Y + Z {{ X = 5 }}                   Y + Z = 5

{{ ? }}  X := Y  {{ X = Y }}                      True

{{ ? }}
  if X = 0 then                   (X = 0 /\ Z = 4) \/ (X <> 0 /\ W = 3)
       Y := Z + 1
  else Y := W + 2
{{ Y = 5 }}

{{ ? }} X := 5 {{ X = 0 }}                        False

{{ ? }} while true do
          X := 0                                  True
        end
{{ X = 0 }}
```

Loop Invariants

- Largely straightforward
- **Except** for loo

? needs to

1. be weak enough to be implied by the loop's precondition,
2. be strong enough to imply the loop's postcondition
3. be preserved by one iteration of the loop

$\{ X = m \wedge Y = n \} \rightarrow \{ \textbf{?} \}$

while $X \neq 0$ do

$\{ \textbf{?} \wedge X \neq 0 \} \rightarrow \{ [X{:=}X\text{-}1]$

Y := Y - 1;

$\{ [X{:=}X\text{-}1] \ ? \}$

X := X - 1

$\{ \textbf{?} \}$

end

$\{ \textbf{?} \wedge X = 0 \} \rightarrow \{ Y = n - m \}$

Loop Invariants

- Largely straightfo
- **Except** for loo

$\{ X = m \wedge Y = n \} \rightarrow \{$ **Tru**

while $X \neq 0$ do

$\{$ **True** $\wedge X \neq 0 \} \rightarrow \{ [X{:=}X\text{-}1]\ [Y{:=}Y\text{-}1]$ **True** $\}$

$Y := Y - 1;$

$\{ [X{:=}X\text{-}1]$ **True** $\}$

$X := X - 1$

$\{$ **True** $\}$

end

$\{$ **True** $\wedge X = 0 \} \rightarrow \{ Y = n - m \}$

? needs to

1. be weak enough to be implied by the loop's precondition,
2. be strong enough to imply the loop's postcondition
3. be preserved by one iteration of the loop

Loop Invariants

- Largely straight
- **Except** for lo

? needs to

1. be weak enough to be implied by the loop's precondition,
2. be strong enough to imply the loop's postcondition
3. be preserved by one iteration of the loop

{ X = m ∧ Y = n } → { **Tru**
while X ≠ 0 do
{ **True** ∧ X ≠ 0 } → { [X≔X-1] [Y≔Y-1] **True**}
Y := Y - 1;
{ [X≔X-1] **True** }
X := X - 1
{ **True** }
end
{ **True** ∧ X = 0 } → { Y = n - m }

What fails to hold when ? is **True?**

Loop Invariants

? needs to

1. be weak enough to be implied by the loop's precondition,
2. be strong enough to imply the loop's postcondition
3. be preserved by one iteration of the loop

- Largely straightf
- **Except** for loop

{ X = m ∧ Y = n } → { **Y = n - m** }
while X ≠ 0 do
{**Y = n - m** ∧ X ≠ 0} → {[X≔X-1] [Y≔Y-1] **Y = n - m**}
Y := Y - 1;
{ **Y =** [X≔X-1] **n - m** }
X := X - 1
{ **Y = n - m** }
end
{ **Y = n - m** ∧ X = 0 } → { Y = n - m }

What fails to hold when

? is **Postcondition?**

Loop Invariants

- ★ Largely straightforwar
- ★ **Except** for loops!

? needs to

1. be weak enough to be implied by the loop's precondition,
2. be strong enough to imply the loop's postcondition
3. be preserved by one iteration of the loop

{ X = m ∧ Y = n } → {**Y-X = n**

while X ≠ 0 do

{**Y-X = n - m** ∧ X ≠ 0}→{[X≔ ...][...]**Y-X = n - m**}

Y := Y - 1;

{ **Y - X = n - m** [X≔X-1] }

X := X - 1

{ **Y - X= n - m** }

end

{ **Y - X = n - m** ∧ X = 0 } → { Y = n - m }

Success!

Recap

- Developed a logic for proving that {P} c {Q} is valid
  - We defined a set of rules (axioms) to build proofs of claims without reasoning directly about states and executions
- Saw how to verify specific programs

$$\boxed{\vdash \{P\}\ c\ \{Q\}}$$