Universität Bremen

dfki Deutsches Forschungszentrum für Künstliche Intelligenz
German Research Center for Artificial Intelligence

Systems of High Safety and Security

Lecture 9 from 17.12.2025: Floyd-Hoare Logic: Proving Correctness

Winter term 2025/26

Christoph Lüth

Roadmap

- Introduction
- Legal Requirements - Norms and Standards
- The Development Process
- Hazard Analysis
- The Big Picture: Hybrid Systems
- Temporal Logic with LTL and CTL
- Operational Semantics
- Axiomatic Semantics - Specifying Correctness
- Floyd-Hoare Logic - Proving Correctness
- A Simple Compiler and its Correctness
- TinyRV32I
- Hardware Verification & Conclusions

Validity and Derivability

- Last week: **Semantic validity** $\models\{P\}\, c\, \{Q\}$
  - Defined by denotational semantics:

$$\models\{P\}\, c\, \{Q\} \iff \forall I.\, \forall \sigma.\, \sigma \models^I P \land \exists \tau.\, (\sigma, \tau) \in [\![c]\!]_{\mathcal{C}} \implies \tau \models^I Q$$

  - Problem: would have to calculate semantics of c

Validity and Derivability

- Last week: **Semantic validity** $\models \{P\}\, c\, \{Q\}$
  - Defined by denotational semantics:

$$\models \{P\}\, c\, \{Q\} \iff \forall I.\, \forall \sigma.\, \sigma \models^I P \land \exists \tau.\, (\sigma, \tau) \in [\![c]\!]_{\mathcal{C}} \implies \tau \models^I Q$$

  - Problem: would have to calculate semantics of c
- Today: **Syntactic derivability** $\vdash \{P\}\, c\, \{Q\}$
  - Defined by **rules**
  - Can be **derived**
  - Must be **proven correct** with respect to semantic validity
- General approach in logic.

Rules of the Floyd-Hoare Calculus

- The Floyd-Hoare calculus allows us to derive syntactically **triples** of the form $\vdash \{P\}\, c\, \{Q\}$.
- The **calculus** of logic consists of six rules of the form

$$\frac{\vdash \{P_1\}\, c_1\, \{Q_1\} \ldots \vdash \{P_n\}\, c_n\, \{Q_n\}}{\vdash \{P\}\, c\, \{Q\}}$$

- For each construct of the programming language, there is a rule:

$$c \quad ::= \quad \textbf{Idt} = \textbf{Exp} \mid c_1; c_2 \mid \textbf{skip} \mid \textbf{if}\ (e)\ \textbf{then}\ c_1\ \textbf{else}\ c_2 \mid \textbf{while}\ (e)\ c$$

Rules of the Floyd-Hoare Calculus: Assignment

$$\frac{}{\vdash \{P[e/x]\}\, x = e\, \{P\}}$$

- An assignment x=e changes the state to update position x with the value of e. In order for the predicate P to hold **afterwards**, the predicate must hold **beforehand** when we replace x with e.
- $P[e/x]$ denotes the **syntactic substitution** of x with e in P.
- The rule is read (and used) from the **back**, i.e. we start with a **postcondition**.
- Examples:

```
// {?}
x = 5
// {x < 10}
```

Rules of the Floyd-Hoare Calculus: Assignment

$$\frac{}{\vdash \{P[e/x]\}\, x = e\, \{P\}}$$

- An assignment x=e changes the state to update position x with the value of e. In order for the predicate P to hold **afterwards**, the predicate must hold **beforehand** when we replace x with e.
- $P[e/x]$ denotes the **syntactic substitution** of x with e in P.
- The rule is read (and used) from the **back**, i.e. we start with a **postcondition**.
- Examples:

```
// {(x < 10)[5/x]}
x = 5
// {x < 10}
```

Rules of the Floyd-Hoare Calculus: Assignment

$$\frac{}{\vdash \{P[e/x]\}\, x = e\, \{P\}}$$

- An assignment x=e changes the state to update position x with the value of e. In order for the predicate P to hold **afterwards**, the predicate must hold **beforehand** when we replace x with e.
- $P[e/x]$ denotes the **syntactic substitution** of x with e in P.
- The rule is read (and used) from the **back**, i.e. we start with a **postcondition**.
- Examples:

```
// {5 < 10}
x = 5
// {x < 10}
```

Rules of the Floyd-Hoare Calculus: Assignment

$$\frac{}{\vdash \{P[e/x]\}\, x = e\, \{P\}}$$

- An assignment x=e changes the state to update position x with the value of e. In order for the predicate P to hold **afterwards**, the predicate must hold **beforehand** when we replace x with e.
- $P[e/x]$ denotes the **syntactic substitution** of x with e in P.
- The rule is read (and used) from the **back**, i.e. we start with a **postcondition**.
- Examples:

```
// {5 < 10}
x = 5
// {x < 10}
```

```
//{(x < 10)[x + 1/x]}
x = x+ 1
//{x < 10}
```

Rules of the Floyd-Hoare Calculus: Assignment

$$\frac{}{\vdash \{P[e/x]\}\, x = e\, \{P\}}$$

- An assignment x=e changes the state to update position x with the value of e. In order for the predicate P to hold **afterwards**, the predicate must hold **beforehand** when we replace x with e.
- $P[e/x]$ denotes the **syntactic substitution** of x with e in P.
- The rule is read (and used) from the **back**, i.e. we start with a **postcondition**.
- Examples:

```
// {5 < 10}
x = 5
// {x < 10}
```

```
//{x + 1 < 10 ⟺ x < 9}
x = x+ 1
//{x < 10}
```

Rules of the Floyd-Hoare Calculus: Sequencing

$$\frac{\vdash \{A\}\, c_1\, \{B\} \qquad \vdash \{B\}\, c_2\, \{C\}}{\vdash \{A\}\, c_1; c_2\, \{C\}}$$

- Here, an intermediate assertion B is required.
- This rules allows the chaining of derivations, deriving a precondition from a given postcondition (or the other way around).

$$\frac{}{\vdash \{A\}\ \textbf{skip}\ \{A\}}$$

- Trivial.

A First Example

```
z= x;
x= y;
y= z;
```

- What does this programme compute?

A First Example

```
z= x;
x= y;
y= z;
```

- What does this programme compute?
- The values of x and y are swapped.
- How do we specify this?

A First Example

```
z= x;
x= y;
y= z;
```

- What does this programme compute?
- The values of x and y are swapped.
- How do we specify this?
- $\vdash \{x = X \wedge y = Y\}\, p\, \{y = X \wedge x = Y\}$

Derivation:

A First Example

```
z= x;
x= y;
y= z;
```

- What does this programme compute?
- The values of x and y are swapped.
- How do we specify this?
- $\vdash \{x = X \wedge y = Y\}\, p\, \{y = X \wedge x = Y\}$

Derivation:

$$\vdash \{x = X \wedge y = Y\}$$
$$z = x; x = y; y = z;$$
$$\{y = X \wedge x = Y\}$$

A First Example

```
z= x;
x= y;
y= z;
```

- What does this programme compute?
- The values of x and y are swapped.
- How do we specify this?
- $\vdash \{x = X \wedge y = Y\}\, p\, \{y = X \wedge x = Y\}$

Derivation:

$$\dfrac{\dfrac{}{\begin{array}{l}\vdash \{x = X \wedge y = Y\} \\ \quad z = x; x = y; \\ \quad \{?\}\end{array}} \quad \dfrac{}{\begin{array}{l}\vdash \{?\} \\ \quad y = z; \\ \quad \{y = X \wedge x = Y\}\end{array}}}{\begin{array}{l}\vdash \{x = X \wedge y = Y\} \\ \quad z = x; x = y; y = z; \\ \quad \{y = X \wedge x = Y\}\end{array}}$$

A First Example

```
z= x;
x= y;
y= z;
```

- What does this programme compute?
- The values of x and y are swapped.
- How do we specify this?
- $\vdash \{x = X \wedge y = Y\}\, p\, \{y = X \wedge x = Y\}$

Derivation:

$$\dfrac{\dfrac{}{\vdash \{x = X \wedge y = Y\}\ z = x; x = y;\ \{z = X \wedge x = Y\}} \qquad \dfrac{}{\vdash \{z = X \wedge x = Y\}\ y = z;\ \{y = X \wedge x = Y\}}}{\vdash \{x = X \wedge y = Y\}\ z = x; x = y; y = z;\ \{y = X \wedge x = Y\}}$$

A First Example

```
z= x;
x= y;
y= z;
```

- What does this programme compute?
- The values of x and y are swapped.
- How do we specify this?
- $\vdash \{x = X \wedge y = Y\}\, p\, \{y = X \wedge x = Y\}$

Derivation:

$$\dfrac{\dfrac{\dfrac{\vdash \{x = X \wedge y = Y\}\ z = x;\ \{?\} \qquad \vdash \{?\}\ x = y;\ \{z = X \wedge x = Y\}}{\vdash \{x = X \wedge y = Y\}\ z = x; x = y;\ \{z = X \wedge x = Y\}} \qquad \vdash \{z = X \wedge x = Y\}\ y = z;\ \{y = X \wedge x = Y\}}{\vdash \{x = X \wedge y = Y\}\ z = x; x = y; y = z;\ \{y = X \wedge x = Y\}}$$

A First Example

```
z= x;
x= y;
y= z;
```

- What does this programme compute?
- The values of x and y are swapped.
- How do we specify this?
- $\vdash \{x = X \wedge y = Y\}\, p\, \{y = X \wedge x = Y\}$

Derivation:

$$\dfrac{\dfrac{\dfrac{}{\vdash \{x = X \wedge y = Y\}\ z = x;\ \{z = X \wedge y = Y\}} \quad \dfrac{}{\vdash \{z = X \wedge y = Y\}\ x = y;\ \{z = X \wedge x = Y\}}}{\vdash \{x = X \wedge y = Y\}\ z = x; x = y;\ \{z = X \wedge x = Y\}} \quad \dfrac{}{\vdash \{z = X \wedge x = Y\}\ y = z;\ \{y = X \wedge x = Y\}}}{\vdash \{x = X \wedge y = Y\}\ z = x; x = y; y = z;\ \{y = X \wedge x = Y\}}$$

Simplified Notation for Sequences

```
// {y = Y ∧ x = X}
z= x;
//
x= y;
//
y= z;
// {x = Y ∧ y = X}
```

- **Same** information as the derivation tree, but in a **compact** form.
- Proof is carried out **backwards** (starting from the last assignment)

Simplified Notation for Sequences

```
// {y = Y ∧ x = X}
z= x;
//
x= y;
// {x = Y ∧ z = X}
y= z;
// {x = Y ∧ y = X}
```

- **Same** information as the derivation tree, but in a **compact** form.
- Proof is carried out **backwards** (starting from the last assignment)

Simplified Notation for Sequences

```
// {y = Y ∧ x = X}
z= x;
// {y = Y ∧ z = X}
x= y;
// {x = Y ∧ z = X}
y= z;
// {x = Y ∧ y = X}
```

- **Same** information as the derivation tree, but in a **compact** form.
- Proof is carried out **backwards** (starting from the last assignment)

Example: A First Proof

Consider the body of the factorial programme:

```
// (B)
p= p* c;
// (A)
c= c+ 1;
// {p = (c − 1)!}
```

- Which assertions hold
  1. at position (A)?
  2. at position (B)?

Example: A First Proof

Consider the body of the factorial programme:

```
// (B)
p= p* c;
// (A)
c= c+ 1;
// {p = (c − 1)!}
```

- Which assertions hold

  1. at position (A)? $p = ((c + 1) - 1)! \iff p = c!$
  2. at position (B)?

Example: A First Proof

Consider the body of the factorial programme:

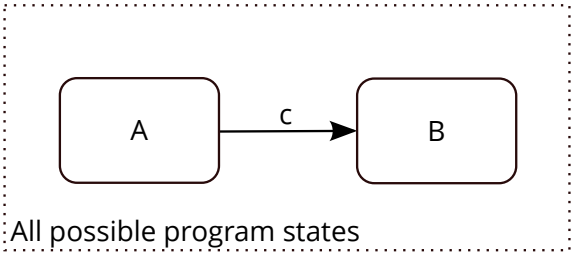
```
// (B)
p= p* c;
// (A)
c= c+ 1;
// {p = (c − 1)!}
```

- Which assertions hold

1. at position (A)? $p = ((c + 1) - 1)! \iff p = c!$
2. at position (B)? $p \cdot c = c! \iff p \cdot c = c \cdot (c - 1)! \iff p = (c - 1)!$

Rules of the Floyd-Hoare Calculus: Weakening

$$\frac{A' \Longrightarrow A \qquad \vdash \{A\}\, c\, \{B\} \qquad B \Longrightarrow B'}{\vdash \{A'\}\, c\, \{B'\}}$$

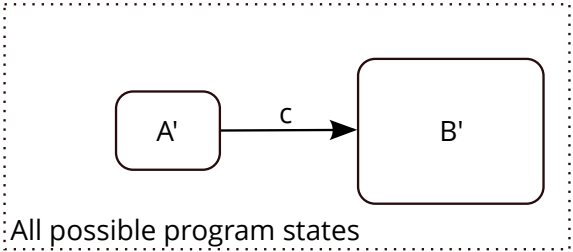


- $\models \{A\}\, c\, \{B\}$: execution of c starts in a state where A holds, and ends (if at all) in a state where B holds.
- State predicates describe sets of states:

$$\{\sigma \in \Sigma | \sigma \models' P\} \subseteq \{\sigma \in \Sigma | \sigma \models' Q\} \text{ if } P \Longrightarrow Q$$

Rules of the Floyd-Hoare Calculus: Weakening

$$\frac{A' \Longrightarrow A \qquad \vdash \{A\}\, c\, \{B\} \qquad B \Longrightarrow B'}{\vdash \{A'\}\, c\, \{B'\}}$$



- $\models \{A\}\, c\, \{B\}$: execution of c starts in a state where A holds, and ends (if at all) in a state where B holds.
- State predicates describe sets of states:

$$\{\sigma \in \Sigma | \sigma \models' P\} \subseteq \{\sigma \in \Sigma | \sigma \models' Q\} \text{ if } P \Longrightarrow Q$$

Example: A Second Proof

We consider again the swapping programme, this time without a temporary variable:

```
// {x = X ∧ y = Y}
// (A)
x= x+y;
// (B)
y= x−y;
// (C)
x= x−y;
// {y = X ∧ x = Y}
```

- Which assertions hold at positions (A), (B), (C) and how are they simplified so that the pre-condition arises?
  1. (C)?
  2. (B)?
  3. (A)?

Example: A Second Proof

We consider again the swapping programme, this time without a temporary variable:

```
// {x = X ∧ y = Y}
// (A)
x= x+y ;
// (B)
y= x−y ;
// (C)
x= x−y ;
// {y = X ∧ x = Y}
```

- Which assertions hold at positions (A), (B), (C) and how are they simplified so that the pre-condition arises?
  1. (C)? $y = X \wedge x - y = Y$
  2. (B)?
  3. (A)?

Example: A Second Proof

We consider again the swapping programme, this time without a temporary variable:

```
// {x = X ∧ y = Y}
// (A)
x= x+y;
// (B)
y= x−y;
// (C)
x= x−y;
// {y = X ∧ x = Y}
```

- Which assertions hold at positions (A), (B), (C) and how are they simplified so that the pre-condition arises?
  1. (C)? $y = X \wedge x - y = Y$
  2. (B)? $x - y = X \wedge x - (x - y) = Y \iff x - y = X \wedge y = Y$
  3. (A)?

Example: A Second Proof

We consider again the swapping programme, this time without a temporary variable:

```
// {x = X ∧ y = Y}
// (A)
x= x+y;
// (B)
y= x−y;
// (C)
x= x−y;
// {y = X ∧ x = Y}
```

- Which assertions hold at positions (A), (B), (C) and how are they simplified so that the pre-condition arises?
  1. (C)? $y = X \wedge x - y = Y$
  2. (B)? $x - y = X \wedge x - (x - y) = Y \iff x - y = X \wedge y = Y$
  3. (A)? $(x + y) - y = X \wedge y = Y \iff x = X \wedge y = Y$

A Second Proof – Alternative Notation

We express weakening through directly consecutive assertions:

```
//
//
x= x+y;
//
//
y= x-y;
//
x= x-y;
// {y = X ∧ x = Y}
```

```
//
//
//
x= x+y;
//
y= x-y;
//
x= x-y;
// {y = X ∧ x = Y}
```

A Second Proof – Alternative Notation

We express weakening through directly consecutive assertions:

```
//
//
x= x+y;
//
//
y= x-y;
// {y = X ∧ x - y = Y}
x= x-y;
// {y = X ∧ x = Y}
```

```
//
//
//
x= x+y;
//
y= x-y;
// {y = X ∧ x - y = Y}
x= x-y;
// {y = X ∧ x = Y}
```

A Second Proof – Alternative Notation

We express weakening through directly consecutive assertions:

```
//
//
x= x+y;
//
// {x − y = X ∧ x − (x − y) = Y}
y= x−y;
// {y = X ∧ x − y = Y}
x= x−y;
// {y = X ∧ x = Y}
```

```
//
//
//
x= x+y;
// {x − y = X ∧ x − (x − y) = Y}
y= x−y;
// {y = X ∧ x − y = Y}
x= x−y;
// {y = X ∧ x = Y}
```

A Second Proof – Alternative Notation

We express weakening through directly consecutive assertions:

```
//
//
x= x+y;
// {x − y = X ∧ y = Y}
// {x − y = X ∧ x − (x − y) = Y}
y= x−y;
// {y = X ∧ x − y = Y}
x= x−y;
// {y = X ∧ x = Y}
```

```
//
//
//
x= x+y;
// {x − y = X ∧ x − (x − y) = Y}
y= x−y;
// {y = X ∧ x − y = Y}
x= x−y;
// {y = X ∧ x = Y}
```

A Second Proof – Alternative Notation

We express weakening through directly consecutive assertions:

```
//
// {(x + y) − y = X ∧ y = Y}
x= x+y ;
// {x − y = X ∧ y = Y}
// {x − y = X ∧ x − (x − y) = Y}
y= x−y ;
// {y = X ∧ x − y = Y}
x= x−y ;
// {y = X ∧ x = Y}
```

```
//
//
//
x= x+y ;
// {x − y = X ∧ x − (x − y) = Y}
y= x−y ;
// {y = X ∧ x − y = Y}
x= x−y ;
// {y = X ∧ x = Y}
```

A Second Proof – Alternative Notation

We express weakening through directly consecutive assertions:

```
// {x = X ∧ y = Y}
// {(x + y) − y = X ∧ y = Y}
x= x+y ;
// {x − y = X ∧ y = Y}
// {x − y = X ∧ x − (x − y) = Y}
y= x−y ;
// {y = X ∧ x − y = Y}
x= x−y ;
// {y = X ∧ x = Y}
```

```
//
//
//
x= x+y ;
// {x − y = X ∧ x − (x − y) = Y}
y= x−y ;
// {y = X ∧ x − y = Y}
x= x−y ;
// {y = X ∧ x = Y}
```

A Second Proof – Alternative Notation

We express weakening through directly consecutive assertions:

$// \ \{x = X \wedge y = Y\}$
$// \ \{(x + y) - y = X \wedge y = Y\}$
x= x+y ;
$// \ \{x - y = X \wedge y = Y\}$
$// \ \{x - y = X \wedge x - (x - y) = Y\}$
y= x−y ;
$// \ \{y = X \wedge x - y = Y\}$
x= x−y ;
$// \ \{y = X \wedge x = Y\}$

$//$
$//$
$// \ \{(x + y) - y = X \wedge (x + y) - ((x + y) - y) = Y\}$
x= x+y ;
$// \ \{x - y = X \wedge x - (x - y) = Y\}$
y= x−y ;
$// \ \{y = X \wedge x - y = Y\}$
x= x−y ;
$// \ \{y = X \wedge x = Y\}$

A Second Proof – Alternative Notation

We express weakening through directly consecutive assertions:

$// \ \{x = X \wedge y = Y\}$
$// \ \{(x + y) - y = X \wedge y = Y\}$
x= x+y ;
$// \ \{x - y = X \wedge y = Y\}$
$// \ \{x - y = X \wedge x - (x - y) = Y\}$
y= x−y ;
$// \ \{y = X \wedge x - y = Y\}$
x= x−y ;
$// \ \{y = X \wedge x = Y\}$

$//$
$// \ \{x = X \wedge x + y - x - y + y = Y\}$
$// \ \{(x + y) - y = X \wedge (x + y) - ((x + y) - y) = Y\}$
x= x+y ;
$// \ \{x - y = X \wedge x - (x - y) = Y\}$
y= x−y ;
$// \ \{y = X \wedge x - y = Y\}$
x= x−y ;
$// \ \{y = X \wedge x = Y\}$

A Second Proof – Alternative Notation

We express weakening through directly consecutive assertions:

$// \ \{x = X \wedge y = Y\}$
$// \ \{(x + y) - y = X \wedge y = Y\}$
x= x+y ;
$// \ \{x - y = X \wedge y = Y\}$
$// \ \{x - y = X \wedge x - (x - y) = Y\}$
y= x−y ;
$// \ \{y = X \wedge x - y = Y\}$
x= x−y ;
$// \ \{y = X \wedge x = Y\}$

$// \ \{x = X \wedge y = Y\}$
$// \ \{x = X \wedge x + y - x - y + y = Y\}$
$// \ \{(x + y) - y = X \wedge (x + y) - ((x + y) - y) = Y\}$
x= x+y ;
$// \ \{x - y = X \wedge x - (x - y) = Y\}$
y= x−y ;
$// \ \{y = X \wedge x - y = Y\}$
x= x−y ;
$// \ \{y = X \wedge x = Y\}$

Rules of the Floyd-Hoare Calculus: Conditional

$$\frac{\vdash \{A \wedge b\}\, c_0\, \{B\} \qquad \vdash \{A \wedge \neg b\}\, c_1\, \{B\}}{\vdash \{A\}\ \textbf{if}\ (b)\ c_0\ \textbf{else}\ c_1\, \{B\}}$$

- In the pre-condition of the **if**-branch, the condition b holds, and in the **else**-branch, the negation $\neg b$ holds.
- Both branches must end with the same post-condition.

A Third Example: Conditionals

Consider the following programme:

```
// (F)
if (x < y) {
  // (E)
  // ...
  z = x;
  // (C)
} else {
  // (D)
  // ...
  z= y;
  // (B)
}
// (A)
```

- What does this programme compute?

A Third Example: Conditionals

Consider the following programme:

```
// (F)
if (x < y) {
  // (E)
  // ...
  z = x;
  // (C)
} else {
  // (D)
  // ...
  z= y;
  // (B)
}
// (A)
```

- What does this programme compute?
- The **minimum** of x and y (in z)

A Third Example: Conditionals

Consider the following programme:

```
// (F)
if (x < y) {
  // (E)
  // ...
  z = x;
  // (C)
} else {
  // (D)
  // ...
  z= y;
  // (B)
}
// (A)
```

- What does this programme compute?
- The **minimum** of x and y (in z)
- How do we specify this?

A Third Example: Conditionals

Consider the following programme:

```
// (F)
if (x < y) {
  // (E)
  // ...
  z = x;
  // (C)
} else {
  // (D)
  // ...
  z= y;
  // (B)
}
// (A)
```

- What does this programme compute?
- The **minimum** of x and y (in z)
- How do we specify this?
- $z \leq x \wedge z \leq y \wedge (z = x \vee z = y)$

A Third Example: Conditionals

Consider the following programme:

```
// (F)
if (x < y) {
  // (E)
  // ...
  z = x;
  // (C)
} else {
  // (D)
  // ...
  z= y;
  // (B)
}
// (A)
```

- What does this programme compute?
- The **minimum** of x and y (in z)
- How do we specify this?
- $z \leq x \wedge z \leq y \wedge (z = x \vee z = y)$
- Which assertions must hold at positions (A) – (F)?

A Third Example: Conditionals

Consider the following programme:

```
// (F)
if (x < y) {
  // (E)
  // ...
  z = x;
  // (C)
} else {
  // (D)
  // ...
  z= y;
  // (B)
}
// (A)
```

- What does this programme compute?
- The **minimum** of x and y (in z)
- How do we specify this?
- $z \leq x \wedge z \leq y \wedge (z = x \vee z = y)$
- Which assertions must hold at positions (A) – (F)?
- Where must we use which logical transformations?

Solution: A Third Example

The programme with complete annotations

```
//
if (x < y) {
  //
  //
  //
  //
  z = x;
  //
} else {
  //
  //
  //
  z= y;
  //
}
// {z ≤ x ∧ z ≤ y ∧ (z = x ∨ z = y)}
```

Solution: A Third Example

The programme with complete annotations

```
//
if (x < y) {
  //
  //
  //
  //
  z = x;
  // {z ≤ x ∧ z ≤ y ∧ (z = x ∨ z = y)}
} else {
  //
  //
  //
  z= y;
  // {z ≤ x ∧ z ≤ y ∧ (z = x ∨ z = y)}
}
// {z ≤ x ∧ z ≤ y ∧ (z = x ∨ z = y)}
```

Solution: A Third Example

The programme with complete annotations

```
//
if (x < y) {
  //
  //
  //
  //
  z = x;
  // {z ≤ x ∧ z ≤ y ∧ (z = x ∨ z = y)}
} else {
  //
  //
  // {y ≤ x ∧ y ≤ y ∧ (y = x ∨ y = y)}
  z= y;
  // {z ≤ x ∧ z ≤ y ∧ (z = x ∨ z = y)}
}
// {z ≤ x ∧ z ≤ y ∧ (z = x ∨ z = y)}
```

Solution: A Third Example

The programme with complete annotations

```
//
if (x < y) {
  //
  //
  //
  // {x ≤ x ∧ x ≤ y ∧ (x = x ∨ x = y)}
  z = x;
  // {z ≤ x ∧ z ≤ y ∧ (z = x ∨ z = y)}
} else {
  //
  //
  // {y ≤ x ∧ y ≤ y ∧ (y = x ∨ y = y)}
  z= y;
  // {z ≤ x ∧ z ≤ y ∧ (z = x ∨ z = y)}
}
// {z ≤ x ∧ z ≤ y ∧ (z = x ∨ z = y)}
```

Solution: A Third Example

The programme with complete annotations

```
//
if (x < y) {
  //
  //
  //
  // {x ≤ x ∧ x ≤ y ∧ (x = x ∨ x = y)}
  z = x;
  // {z ≤ x ∧ z ≤ y ∧ (z = x ∨ z = y)}
} else {
  //
  // {y ≤ x ∧ true ∧ (y = x ∨ true}
  // {y ≤ x ∧ y ≤ y ∧ (y = x ∨ y = y)}
  z= y;
  // {z ≤ x ∧ z ≤ y ∧ (z = x ∨ z = y)}
}
// {z ≤ x ∧ z ≤ y ∧ (z = x ∨ z = y)}
```

Solution: A Third Example

The programme with complete annotations

```
//
if (x < y) {
  //
  //
  //
  // {x ≤ x ∧ x ≤ y ∧ (x = x ∨ x = y)}
  z = x;
  // {z ≤ x ∧ z ≤ y ∧ (z = x ∨ z = y)}
} else {
  // {¬(x < y) ∧ true}
  // {y ≤ x ∧ true ∧ (y = x ∨ true}
  // {y ≤ x ∧ y ≤ y ∧ (y = x ∨ y = y)}
  z= y;
  // {z ≤ x ∧ z ≤ y ∧ (z = x ∨ z = y)}
}
// {z ≤ x ∧ z ≤ y ∧ (z = x ∨ z = y)}
```

Solution: A Third Example

The programme with complete annotations

```
//
if (x < y) {
  //
  //
  // {true ∧ x ≤ y ∧ (true ∨ x = y)}
  // {x ≤ x ∧ x ≤ y ∧ (x = x ∨ x = y)}
  z = x;
  // {z ≤ x ∧ z ≤ y ∧ (z = x ∨ z = y)}
} else {
  // {¬(x < y) ∧ true}
  // {y ≤ x ∧ true ∧ (y = x ∨ true}
  // {y ≤ x ∧ y ≤ y ∧ (y = x ∨ y = y)}
  z= y;
  // {z ≤ x ∧ z ≤ y ∧ (z = x ∨ z = y)}
}
// {z ≤ x ∧ z ≤ y ∧ (z = x ∨ z = y)}
```

Solution: A Third Example

The programme with complete annotations

```
//
if (x < y) {
  //
  // {x ≤ y}
  // {true ∧ x ≤ y ∧ (true ∨ x = y)}
  // {x ≤ x ∧ x ≤ y ∧ (x = x ∨ x = y)}
  z = x;
  // {z ≤ x ∧ z ≤ y ∧ (z = x ∨ z = y)}
} else {
  // {¬(x < y) ∧ true}
  // {y ≤ x ∧ true ∧ (y = x ∨ true}
  // {y ≤ x ∧ y ≤ y ∧ (y = x ∨ y = y)}
  z= y;
  // {z ≤ x ∧ z ≤ y ∧ (z = x ∨ z = y)}
}
// {z ≤ x ∧ z ≤ y ∧ (z = x ∨ z = y)}
```

Solution: A Third Example

The programme with complete annotations

```
//
if (x < y) {
  // {x < y ∧ true}
  // {x ≤ y}
  // {true ∧ x ≤ y ∧ (true ∨ x = y)}
  // {x ≤ x ∧ x ≤ y ∧ (x = x ∨ x = y)}
  z = x;
  // {z ≤ x ∧ z ≤ y ∧ (z = x ∨ z = y)}
} else {
  // {¬(x < y) ∧ true}
  // {y ≤ x ∧ true ∧ (y = x ∨ true}
  // {y ≤ x ∧ y ≤ y ∧ (y = x ∨ y = y)}
  z= y;
  // {z ≤ x ∧ z ≤ y ∧ (z = x ∨ z = y)}
}
// {z ≤ x ∧ z ≤ y ∧ (z = x ∨ z = y)}
```

Solution: A Third Example

The programme with complete annotations

```
// {true}
if (x < y) {
  // {x < y ∧ true}
  // {x ≤ y}
  // {true ∧ x ≤ y ∧ (true ∨ x = y)}
  // {x ≤ x ∧ x ≤ y ∧ (x = x ∨ x = y)}
  z = x;
  // {z ≤ x ∧ z ≤ y ∧ (z = x ∨ z = y)}
} else {
  // {¬(x < y) ∧ true}
  // {y ≤ x ∧ true ∧ (y = x ∨ true}
  // {y ≤ x ∧ y ≤ y ∧ (y = x ∨ y = y)}
  z= y;
  // {z ≤ x ∧ z ≤ y ∧ (z = x ∨ z = y)}
}
// {z ≤ x ∧ z ≤ y ∧ (z = x ∨ z = y)}
```

Rules of the Floyd-Hoare Calculus: Iteration

$$\frac{\vdash \{A \wedge b\}\ c\ \{A\}}{\vdash \{A\}\ \textbf{while}(b)\ c\ \{A \wedge \neg b\}}$$

- Iteration corresponds to **induction**.
- With (well-founded) induction, we show that the **same** property holds for all x, $P(x)$, if it holds for all smaller y — i.e., if y becomes larger, the property must still hold.
- Analogously, we need an **invariant** A which holds **before** and **after** the loop body.
- The **pre-condition** of the **loop** is the invariant A.
- In the **pre-condition** of the **loop body**, we can assume the loop condition b.
- After the loop has terminated, A and the **negation** of the loop condition b hold — this is the **post-condition**.

Finding Invariants

- Finding invariants can be tricky.
- We calculate the invariant from the post-condition (when going backwards).
- A lot of while-loops are **counting** loops (e.g. in C, **for** is just syntactic sugar). In this case, if we have $\psi[n]$ as the postcondition, try $\psi[i-1]$ as invariant, where i is the counting variable.
- If the loop goes up to $n-1$ (typical for arrays, loop condition $i < n$), you may need $i \leq n$ as additional condition, so in the end you can deduce $i = n$ to get $\psi[n-1]$.
- If the loop goes up to n (loop condition $i \leq n$), you may need $i - 1 leqn$ as additional condition, so you can deduce $i - 1 = n$ to get $\psi[n]$.

How to Write Floyd-Hoare Proofs

```
// {P}
// {P_2[e/x]}
x= e;
// {P_3}
while (x< n) {
    // {P_3 ∧ x < n}
    // {P_3[a/z]}
    z= a;
    // {P_3}
    }
// {P_3 ∧ ¬(x < n)}
// {Q}
```

- Example shows: $\vdash \{P\}\, c\, \{Q\}$
- Programme is annotated with valid assertions.
- Before a line stands the pre-condition, afterwards the post-condition.
  - Must match exactly with the instruction.
- Implicit application of the sequencing rule.
- Weakening is noted by multiple assertions and must be **proven**.
  - In the example: $P \Longrightarrow P_2[e/x]$, $P_2 \Longrightarrow P_3$, $P_3 \wedge x < n \Longrightarrow P_4$, $P_3 \wedge \neg(x < n) \Longrightarrow Q$.

The Complete Factorial

This program computes the factorial of n:

```
p= 1;
c= 1;
while (c <= n) {
  p = p * c;
  c = c + 1;
}
```

- What is the specification (postcondition)?
- What is the **invariant**?
- What kind of loop do we have?

Factorial Revisited

This program computes the factorial of n:

```
// {n = N ∧ 0 ≤ n}
//
//
p= 1;
//
while (0 < n) {
 //
 //
 //
 p= n*p;
 //
 //
 n= n-1;
 //
}
//
//
// {p = N!}
```

Factorial Revisited

This program computes the factorial of n:

```
// {n = N ∧ 0 ≤ n}
//
//
p= 1;
//
while (0 < n) {
 //
 //
 //
 p= n*p;
 //
 //
 n= n-1;
 //
}
//
// {n! · p = N! ∧ 0 ≤ n ∧ n ≥ 0}
// {p = N!}
```

$$n! \cdot p = N!$$

Factorial Revisited

This program computes the factorial of n:

```
// {n = N ∧ 0 ≤ n}
//
//
p= 1;
//
while (0 < n) {
 //
 //
 //
 p= n*p;
 //
 //
 n= n-1;
 //
}
// {n! · p = N! ∧ n ≤ N ∧ 0 ≤ n ∧ ¬(0 < n)}
// {n! · p = N! ∧ 0 ≤ n ∧ n ≥ 0}
// {p = N!}
```

$$n! \cdot p = N! \\ \wedge\ 0 \leq n$$

Factorial Revisited

This program computes the factorial of n:

```
// {n = N ∧ 0 ≤ n}
//
//
p= 1;
//
while (0 < n) {
 //
 //
 //
 p= n*p;
 //
 //
 n= n-1;
 // {n! · p = N! ∧ n ≤ N ∧ 0 ≤ n}
}
// {n! · p = N! ∧ n ≤ N ∧ 0 ≤ n ∧ ¬(0 < n)}
// {n! · p = N! ∧ 0 ≤ n ∧ n ≥ 0}
// {p = N!}
```

$$n! \cdot p = N! \\ \wedge\ 0 \leq n$$

Factorial Revisited

This program computes the factorial of n:

```
// {n = N ∧ 0 ≤ n}
//
//
p= 1;
//
while (0 < n) {
 //
 //
 //
 p= n*p;
 //
 // {(n − 1)! · p = N! ∧ n − 1 ≤ N ∧ 0 ≤ n − 1}
 n= n−1;
 // {n! · p = N! ∧ n ≤ N ∧ 0 ≤ n}
}
// {n! · p = N! ∧ n ≤ N ∧ 0 ≤ n ∧ ¬(0 < n)}
// {n! · p = N! ∧ 0 ≤ n ∧ n ≥ 0}
// {p = N!}
```

$$n! \cdot p = N! \\ \wedge\ 0 \leq n$$

Factorial Revisited

This program computes the factorial of n:

```
// {n = N ∧ 0 ≤ n}
//
//
p= 1;
//
while (0 < n) {
 //
 //
 //
 p= n*p;
 // {(n − 1)! · p = N! ∧ n ≤ N ∧ 0 < n}
 // {(n − 1)! · p = N! ∧ n − 1 ≤ N ∧ 0 ≤ n − 1}
 n= n−1;
 // {n! · p = N! ∧ n ≤ N ∧ 0 ≤ n}
}
// {n! · p = N! ∧ n ≤ N ∧ 0 ≤ n ∧ ¬(0 < n)}
// {n! · p = N! ∧ 0 ≤ n ∧ n ≥ 0}
// {p = N!}
```

$$n! \cdot p = N! \\ \wedge\ 0 \leq n$$

Factorial Revisited

This program computes the factorial of n:

```
// {n = N ∧ 0 ≤ n}
//
//
p= 1;
//
while (0 < n) {
 //
 //
 // {(n − 1)! · n · p = N! ∧ n ≤ N ∧ 0 < n}
 p= n*p;
 // {(n − 1)! · p = N! ∧ n ≤ N ∧ 0 < n}
 // {(n − 1)! · p = N! ∧ n − 1 ≤ N ∧ 0 ≤ n − 1}
 n= n−1;
 // {n! · p = N! ∧ n ≤ N ∧ 0 ≤ n}
}
// {n! · p = N! ∧ n ≤ N ∧ 0 ≤ n ∧ ¬(0 < n)}
// {n! · p = N! ∧ 0 ≤ n ∧ n ≥ 0}
// {p = N!}
```

$$
\begin{aligned}
& n! \cdot p = N! \\
\wedge\ & 0 \leq n \\
\wedge\ & n \leq n
\end{aligned}
$$

Factorial Revisited

This program computes the factorial of n:

```
// {n = N ∧ 0 ≤ n}
//
//
p= 1;
//
while (0 < n) {
 //
 // {n! · p = N! ∧ n ≤ N ∧ 0 < n}
 // {(n − 1)! · n · p = N! ∧ n ≤ N ∧ 0 < n}
 p= n*p;
 // {(n − 1)! · p = N! ∧ n ≤ N ∧ 0 < n}
 // {(n − 1)! · p = N! ∧ n − 1 ≤ N ∧ 0 ≤ n − 1}
 n= n−1;
 // {n! · p = N! ∧ n ≤ N ∧ 0 ≤ n}
}
// {n! · p = N! ∧ n ≤ N ∧ 0 ≤ n ∧ ¬(0 < n)}
// {n! · p = N! ∧ 0 ≤ n ∧ n ≥ 0}
// {p = N!}
```

$$
\begin{aligned}
& n! \cdot p = N! \\
\wedge\ & 0 \leq n \\
\wedge\ & n \leq n
\end{aligned}
$$

Factorial Revisited

This program computes the factorial of n:

```
// {n = N ∧ 0 ≤ n}
//
//
p= 1;
//
while (0 < n) {
 // {n! · p = N! ∧ n ≤ N ∧ 0 ≤ n ∧ 0 < n}
 // {n! · p = N! ∧ n ≤ N ∧ 0 < n}
 // {(n − 1)! · n · p = N! ∧ n ≤ N ∧ 0 < n}
 p= n*p;
 // {(n − 1)! · p = N! ∧ n ≤ N ∧ 0 < n}
 // {(n − 1)! · p = N! ∧ n − 1 ≤ N ∧ 0 ≤ n − 1}
 n= n−1;
 // {n! · p = N! ∧ n ≤ N ∧ 0 ≤ n}
}
// {n! · p = N! ∧ n ≤ N ∧ 0 ≤ n ∧ ¬(0 < n)}
// {n! · p = N! ∧ 0 ≤ n ∧ n ≥ 0}
// {p = N!}
```

$$
\begin{aligned}
& n! \cdot p = N! \\
\wedge\ & 0 \leq n \\
\wedge\ & n \leq n
\end{aligned}
$$

Factorial Revisited

This program computes the factorial of n:

```
// {n = N ∧ 0 ≤ n}
//
//
p= 1;
// {n! · p = N! ∧ n ≤ N ∧ 0 ≤ n}
while (0 < n) {
 // {n! · p = N! ∧ n ≤ N ∧ 0 ≤ n ∧ 0 < n}
 // {n! · p = N! ∧ n ≤ N ∧ 0 < n}
 // {(n − 1)! · n · p = N! ∧ n ≤ N ∧ 0 < n}
 p= n*p;
 // {(n − 1)! · p = N! ∧ n ≤ N ∧ 0 < n}
 // {(n − 1)! · p = N! ∧ n − 1 ≤ N ∧ 0 ≤ n − 1}
 n= n−1;
 // {n! · p = N! ∧ n ≤ N ∧ 0 ≤ n}
}
// {n! · p = N! ∧ n ≤ N ∧ 0 ≤ n ∧ ¬(0 < n)}
// {n! · p = N! ∧ 0 ≤ n ∧ n ≥ 0}
// {p = N!}
```

$$
\begin{aligned}
& n! \cdot p = N! \\
\wedge\ & 0 \leq n \\
\wedge\ & n \leq n
\end{aligned}
$$

Factorial Revisited

This program computes the factorial of n:

```
// {n = N ∧ 0 ≤ n}
//
// {n! · 1 = N! ∧ n ≤ N ∧ 0 ≤ n}
p= 1;
// {n! · p = N! ∧ n ≤ N ∧ 0 ≤ n}
while (0 < n) {
 // {n! · p = N! ∧ n ≤ N ∧ 0 ≤ n ∧ 0 < n}
 // {n! · p = N! ∧ n ≤ N ∧ 0 < n}
 // {(n − 1)! · n · p = N! ∧ n ≤ N ∧ 0 < n}
 p= n*p;
 // {(n − 1)! · p = N! ∧ n ≤ N ∧ 0 < n}
 // {(n − 1)! · p = N! ∧ n − 1 ≤ N ∧ 0 ≤ n − 1}
 n= n−1;
 // {n! · p = N! ∧ n ≤ N ∧ 0 ≤ n}
}
// {n! · p = N! ∧ n ≤ N ∧ 0 ≤ n ∧ ¬(0 < n)}
// {n! · p = N! ∧ 0 ≤ n ∧ n ≥ 0}
// {p = N!}
```

$$
\begin{aligned}
& n! \cdot p = N! \\
\wedge\ & 0 \leq n \\
\wedge\ & n \leq n
\end{aligned}
$$

Factorial Revisited

This program computes the factorial of n:

```
// {n = N ∧ 0 ≤ n}
// {n! = N! ∧ n = N ∧ 0 ≤ n}
// {n! · 1 = N! ∧ n ≤ N ∧ 0 ≤ n}
p= 1;
// {n! · p = N! ∧ n ≤ N ∧ 0 ≤ n}
while (0 < n) {
 // {n! · p = N! ∧ n ≤ N ∧ 0 ≤ n ∧ 0 < n}
 // {n! · p = N! ∧ n ≤ N ∧ 0 < n}
 // {(n − 1)! · n · p = N! ∧ n ≤ N ∧ 0 < n}
 p= n*p;
 // {(n − 1)! · p = N! ∧ n ≤ N ∧ 0 < n}
 // {(n − 1)! · p = N! ∧ n − 1 ≤ N ∧ 0 ≤ n − 1}
 n= n−1;
 // {n! · p = N! ∧ n ≤ N ∧ 0 ≤ n}
}
// {n! · p = N! ∧ n ≤ N ∧ 0 ≤ n ∧ ¬(0 < n)}
// {n! · p = N! ∧ 0 ≤ n ∧ n ≥ 0}
// {p = N!}
```

$$
\begin{aligned}
& n! \cdot p = N! \\
\wedge\ & 0 \leq n \\
\wedge\ & n \leq n
\end{aligned}
$$

Cheat Sheet: Rules of the Floyd-Hoare-Logic

$$\frac{}{\vdash \{P[e/x]\}\, x = e\, \{P\}}$$

$$\frac{\vdash \{A \wedge b\}\, c_0\, \{B\} \qquad \vdash \{A \wedge \neg b\}\, c_1\, \{B\}}{\vdash \{A\}\ \textbf{if}\ (b)\ c_0\ \textbf{else}\ c_1\, \{B\}}$$

$$\frac{\vdash \{A \wedge b\}\, c\, \{A\}}{\vdash \{A\}\ \textbf{while}(b)\ c\, \{A \wedge \neg b\}}$$

$$\frac{}{\vdash \{A\}\ \textbf{skip}\, \{A\}} \qquad \frac{\vdash \{A\}\, c_1\, \{B\} \qquad \vdash \{B\}\, c_2\, \{C\}}{\vdash \{A\}\, c_1; c_2\, \{C\}}$$

$$\frac{A' \Longrightarrow A \qquad \vdash \{A\}\, c\, \{B\} \qquad B \Longrightarrow B'}{\vdash \{A'\}\, c\, \{B'\}}$$

Correctness and Completeness

- What about the relation between $\models\{P\}\, c\, \{Q\} \overset{?}{\iff} \vdash\{P\}\, c\, \{Q\}$?
- We have the following:

Theorem: Correctness

$$\vdash\{P\}\, c\, \{Q\} \implies \models\{P\}\, c\, \{Q\}$$

- All syntactically derivable Hoare triples are semantically valid.

Theorem: Relative Completeness

$$\models\{P\}\, c\, \{Q\} \implies \vdash\{P\}\, c\, \{Q\} \quad \textbf{except} \text{ for weakening proofs and invariants.}$$

- We cannot **automatically** prove program correctness.

Summary of Floyd-Hoare Logic

- The logic abstracts over concrete system states through **assertions**
- Assertions are boolean expressions enriched with logical variables.
- **Hoare triples** $\{P\}\, c\, \{Q\}$ abstract the semantics of c
  - Semantic **validity** of Hoare triples: $\models \{P\}\, c\, \{Q\}$.
  - Syntactic **derivability** of Hoare triples: $\vdash \{P\}\, c\, \{Q\}$
- **Assignments** are modelled through **substitution**, i.e., the set of valid statements changes.
- For iterations, an **invariant** is required (which **cannot** be derived).