Axiomatic semantics
Semantics and Application to Program Verification

Antoine Miné

École normale supérieure, Paris
year 2015–2016

Course 6
18 March 2016
Introduction

**Operational semantics**
Models precisely program execution as low-level transitions between internal states
(transition systems, execution traces, big-step semantics)

**Denotational semantics**
Maps programs into objects in a mathematical domain
(higher level, compositional, domain oriented)

**Axiomatic semantics** (today)
Prove properties about programs

- programs are annotated with logical assertions
- a rule-system defines the validity of assertions (logical proofs)
- clearly separates programs from specifications
  (specification $\simeq$ user-provided abstraction of the behavior, it is not unique)
- enables the use of logic tools (partial automation, increased confidence)
Overview

- **Specifications** (informal examples)
- Floyd–Hoare logic
- Dijkstra’s predicate calculus
  (weakest precondition, strongest postcondition)
- Verification conditions
  (partially automated program verification)
- Total correctness (termination)
- Non-determinism
- Arrays
- Concurrency
Specifications
Example: function specification

```c
int mod(int A, int B) {
    int Q = 0;
    int R = A;
    while (R >= B) {
        R = R - B;
        Q = Q + 1;
    }
    return R;
}
```
Example in C + ACSL

```c
//@ ensures \result == A mod B;
int mod(int A, int B) {
    int Q = 0;
    int R = A;
    while (R >= B) {
        R = R - B;
        Q = Q + 1;
    }
    return R;
}
```

- express the intended behavior of the function (returned value)
Example: function specification

Example in C + ACSL

```c
//@ requires A>=0 && B>=0;
//@ ensures result == A mod B;
int mod(int A, int B) {
    int Q = 0;
    int R = A;
    while (R >= B) {
        R = R - B;
        Q = Q + 1;
    }
    return R;
}
```

- express the intended behavior of the function (returned value)
- add requirements for the function to actually behave as intended
  (a requires/ensures pair is a function contract)
Example: function specification

```c
//@ requires A>=0 && B>0;
//@ ensures \result == A mod B;
int mod(int A, int B) {
    int Q = 0;
    int R = A;
    while (R >= B) {
        R = R - B;
        Q = Q + 1;
    }
    return R;
}
```

- express the intended behavior of the function (returned value)
- add requirements for the function to actually behave as intended (a requires/ensures pair is a function contract)
- strengthen the requirements to ensure termination
Example: program annotations

```c
//@ requires A>=0 && B>0;
//@ ensures \result == A mod B;
int mod(int A, int B) {
    int Q = 0;
    int R = A;
    //@ assert A>=0 && B>0 && Q=0 && R==A;
    while (R >= B) {
        //@ assert A>=0 && B>0 && R>=B && A==Q*B+R;
        R = R - B;
        Q = Q + 1;
    }
    //@ assert A>=0 && B>0 && R>=0 && R<B && A==Q*B+R;
    return R;
}
```

Assertions give detail about the internal computations why and how contracts are fulfilled

(Note: \( r = a \mod b \) means \( \exists q: a = qb + r \land 0 \leq r < b \))
Example: ghost variables

```c
//@ requires A>=0 && B>0;
//@ ensures \result == A mod B;
int mod(int A, int B) {
    int R = A;
    while (R >= B) {
        R = R - B;
    }
    // ∃Q: A = QB + R and 0 ≤ R < B
    return R;
}
```

The annotations can be more complex than the program itself
Example: ghost variables

```c
//@ requires A>=0 && B>0;
//@ ensures \result == A mod B;
int mod(int A, int B) {
  //@ ghost int q = 0;
  int R = A;
  //@ assert A>=0 && B>0 && q=0 && R==A;
  while (R >= B) {
    //@ assert A>=0 && B>0 && R>=B && A==q*B+R;
    R = R - B;
    //@ ghost q = q + 1;
  }
  //@ assert A>=0 && B>0 && R>=0 && R<B && A==q*B+R;
  return R;
}
```

The annotations can be more complex than the program itself and require reasoning on enriched states (ghost variables)
Example: class invariants

**Example in ESC/Java**

```java
public class OrderedArray {
    int a[];
    int nb;
    //@invariant nb >= 0 && nb <= 20
    //@invariant (\forall int i; (i >= 0 && i < nb-1) ==> a[i] <= a[i+1])

    public OrderedArray() { a = new int[20]; nb = 0; }

    public void add(int v) {
        if (nb >= 20) return;
        int i; for (i=nb; i > 0 && a[i-1] > v; i--) a[i] = a[i-1];
        a[i] = v; nb++;
    }
}
```

**class invariant:** property of the fields true outside all methods

it can be temporarily broken within a method
but it must be restored before exiting the method
Contracts (and class invariants):
- built in few languages (Eiffel)
- available as a library / external tool (C, Java, C#, etc.)

Contracts can be:
- checked dynamically
- **checked statically** (Frama-C, Why, ESC/Java)
- inferred statically (CodeContracts)

**In this course:**
deductive methods (logic) to check (prove) statically (at compile-time) partially automatically (with user help) that contracts hold
Floyd–Hoare logic
**Hoare triples**

**Hoare triple:** \( \{ P \} \text{ prog } \{ Q \} \)

- \textit{prog} is a program fragment
- \( P \) and \( Q \) are \textit{logical assertions} over program variables
  
  \( P \) def \((X \geq 0 \land Y \geq 0) \lor (X < 0 \land Y < 0))\)

A triple means:

- if \( P \) holds before \textit{prog} is executed
- then \( Q \) holds after the execution of \textit{prog}
- unless \textit{prog} does not terminate or encounters an error

\( P \) is the \textit{precondition}, \( Q \) is the \textit{postcondition}

\( \{ P \} \text{ prog } \{ Q \} \) expresses \textit{partial correctness}

(do not rule out errors and non-termination)

Hoare triples serve as \textit{judgements} in a proof system

(introduced in [Hoare69])
## Language

\[
stat ::= \begin{array}{ll}
X \leftarrow expr & \text{(assignment)} \\
\mid \text{skip} & \text{(do nothing)} \\
\mid \text{fail} & \text{(error)} \\
\mid \text{stat; stat} & \text{(sequence)} \\
\mid \text{if expr then stat else stat} & \text{(conditional)} \\
\mid \text{while expr do stat} & \text{(loop)}
\end{array}
\]

- $X \in \mathbb{V}$: integer-valued variables

- \textit{expr}: integer arithmetic expressions

we assume that:

- expressions are deterministic (for now)

- expression evaluation does not cause error (only \texttt{fail} does)

for instance, to avoid divisions by zero, we assume that all divisions are \textit{explicitly} guarded

as in: \texttt{if $X = 0$ then fail else \ldots /$X$\ldots}
Hoare rules: axioms

Axioms:

\[
\{ P \} \text{skip} \{ P \} \\
\{ P \} \text{fail} \{ Q \}
\]

- any property true before \text{skip} is true afterwards
- any property is true after \text{fail}
Hoare rules: axioms

**Assignment axiom:**

\[
\{ P[e/X] \} \ X \leftarrow e \ \{ P \}
\]

for \( P \) over \( X \) to be true after \( X \leftarrow e \)

\( P \) must be true over \( e \) before the assignment

- \( P[e/X] \) is \( P \) where all free occurrences of \( X \) are replaced with \( e \)
- \( e \) must be deterministic
- the rule is “backwards”: \( P \) appears as a postcondition

**Examples:**

\[
\begin{align*}
\{ \text{true} \} \ X & \leftarrow 5 \ \{ X = 5 \} \\
\{ Y = 5 \} \ X & \leftarrow Y \ \{ X = 5 \} \\
\{ X + 1 \geq 0 \} \ X & \leftarrow X + 1 \ \{ X \geq 0 \} \\
\{ \text{false} \} \ X & \leftarrow Y + 3 \ \{ Y = 0 \land X = 12 \} \\
\{ Y \in [0, 10] \} \ X & \leftarrow Y + 3 \ \{ X = Y + 3 \land Y \in [0, 10] \}
\end{align*}
\]
Floyd–Hoare logic

Hoare rules: consequence

**Rule of consequence:**

\[
P \Rightarrow P' \quad Q' \Rightarrow Q \quad \{P'\} \leftarrow \{Q'\}
\]

\[
\{P\} \leftarrow \{Q\}
\]

we can weaken a Hoare triple by:

- **weakening its postcondition** \( Q \leftarrow Q' \)
- **strengthening its precondition** \( P \Rightarrow P' \)

we assume a logic system to be available to prove formulas on assertions, such as \( P \Rightarrow P' \) (e.g., arithmetic, set theory, etc.)

**examples:**

- the axiom for **fail** can be replaced with \( \{true\} \) **fail** \( \{false\} \)
  (as \( P \Rightarrow true \) and \( false \Rightarrow Q \) always hold)

- \( \{X = 99 \land Y \in [1,10]\} \ X \leftarrow Y + 10 \ \{X = Y + 10 \land Y \in [1,10]\} \)
  (as \( \{Y \in [1,10]\} \ X \leftarrow Y + 10 \ \{X = Y + 10 \land Y \in [1,10]\} \) and \( X = 99 \land Y \in [1,10] \Rightarrow Y \in [1,10] \))
Hoare rules: tests

Tests:

\[ \begin{align*}
\{ P \land e \} & \quad s \quad \{ Q \} \\
\{ P \land \neg e \} & \quad t \quad \{ Q \} \\
\{ P \} & \quad \text{if } e \text{ then } s \text{ else } t \quad \{ Q \}
\end{align*} \]

to prove that \( Q \) holds after the test
we prove that it holds after each branch \((s, t)\)
under the assumption that the branch is executed \((e, \neg e)\)

example:

\[
\begin{align*}
\{ X < 0 \} & \quad X \leftarrow -X \quad \{ X > 0 \} \\
\{ X \neq 0 \} \land (X < 0) & \quad X \leftarrow -X \quad \{ X > 0 \} \\
\{ X \neq 0 \} & \quad \text{if } X < 0 \text{ then } X \leftarrow -X \text{ else skip } \{ X > 0 \} \\
\{ X > 0 \} & \quad \text{skip } \{ X > 0 \} \\
\{ X \neq 0 \} \land (X \geq 0) & \quad \text{skip } \{ X > 0 \}
\end{align*}
\]
Hoare rules: sequences

Sequences: \[
\begin{array}{c}
\{P\} \ s \ \{R\} \quad \{R\} \ t \ \{Q\} \\
\{P\} \ s; \ t \ \{Q\}
\end{array}
\]

to prove a sequence \( s; t \)
we must invent an intermediate assertion \( R \)
 implied by \( P \) after \( s \), and implying \( Q \) after \( t \)
(often denoted \( \{P\} \ s \ \{R\} \ t \ \{Q\} \))

example:
\[
\{X = 1 \land Y = 1\} \ X \leftarrow X + 1 \ \{X = 2 \land Y = 1\} \ Y \leftarrow Y - 1 \ \{X = 2 \land Y = 0\}
\]
Floyd–Hoare logic

Hoare rules: loops

**Loops:**

\[
\begin{align*}
\{ P \land e \} \ s \ \{ P \} \\
\{ P \} \ \textbf{while} \ e \ \textbf{do} \ s \ \{ P \land \neg e \}
\end{align*}
\]

\( P \) is a loop invariant:

\( P \) holds before each loop iteration, before even testing \( e \)

Practical use:

actually, we would rather prove the triple: \( \{ P \} \ \textbf{while} \ e \ \textbf{do} \ s \ \{ Q \} \)

it is sufficient to invent an assertion \( I \) that:

- holds when the loop start: \( P \Rightarrow I \)
- is invariant by the body \( s \): \( \{ I \land e \} \ s \ \{ I \} \)
- implies the assertion when the loop stops: \( (I \land \neg e) \Rightarrow Q \)

we can derive the rule:

\[
\begin{align*}
P \Rightarrow I & \quad I \land \neg e \Rightarrow Q \\
\{ I \land e \} \ s \ \{ I \} \\
\{ I \} \ \textbf{while} \ e \ \textbf{do} \ s \ \{ I \land \neg e \}
\end{align*}
\]

\( \{ P \} \ \textbf{while} \ e \ \textbf{do} \ s \ \{ Q \} \)
Hoare rules: logical part

Hoare logic is **parameterized** by the choice of logical theory of assertions. The logical theory is used to:

- **prove** properties of the form $P \Rightarrow Q$ (rule of consequence)
- **simplify** formulas
  (replace a formula with a simpler one, equivalent in a logical sens: $\Leftrightarrow$)

**Examples:** (generally first order theories)

- booleans ($\mathbb{B}, \neg, \land, \lor$)
- bit-vectors ($\mathbb{B}^n, \neg, \land, \lor$)
- Presburger arithmetic ($\mathbb{N}, +$)
- Peano arithmetic ($\mathbb{N}, +, \times$)
- linear arithmetic on $\mathbb{R}$
- Zermelo-Fraenkel set theory ($\in, \{\}$)
- theory of arrays (lookup, update)

Theories have different expressiveness, decidability and complexity results. This is an important factor when trying to automate program verification.
Hoare rules: summary

\[
\begin{align*}
\{P\} \text{ skip } \{P\} & \\
\{\text{true}\} \text{ fail } \{\text{false}\} & \\
\{P[e/X]\} X \leftarrow e \{P\} & \\
\{P\} s \{R\} & \quad \{R\} t \{Q\} \quad \{P\} s; t \{Q\} \\
\{P \land e\} s \{Q\} & \quad \{P \land \neg e\} t \{Q\} \quad \{P\} \text{ if } e \text{ then } s \text{ else } t \{Q\} \\
\{P \land e\} s \{P\} & \quad \{P\} \text{ while } e \text{ do } s \{P \land \neg e\} \\
\end{align*}
\]

\[
\begin{align*}
P \Rightarrow P' & \quad Q' \Rightarrow Q & \quad \{P'\} c \{Q'\} \\
\{P\} c \{Q\} & 
\end{align*}
\]
Proof tree example

\[ s \overset{\text{def}}{=} \text{while } I < N \text{ do } (X \leftarrow 2X; \ I \leftarrow I + 1) \]

\[
\begin{array}{c}
\{ P_3 \} \ X \leftarrow 2X & \{ P_2 \} \\
\{ P_2 \} \ I \leftarrow I + 1 & \{ P_1 \} \\
\{ P_1 \land I < N \} \ X \leftarrow 2X; \ I \leftarrow I + 1 & \{ P_1 \} \\
\end{array}
\]

\[
\begin{array}{c}
\{ P_1 \} \ s & \{ P_1 \land I \geq N \} \\
\{ X = 1 \land I = 0 \land N \geq 0 \} \ s & \{ X = 2^N \land N = I \land N \geq 0 \}
\end{array}
\]

\[
P_1 \overset{\text{def}}{=} X = 2^I \land I \leq N \land N \geq 0
\]

\[
P_2 \overset{\text{def}}{=} X = 2^{I+1} \land I + 1 \leq N \land N \geq 0
\]

\[
P_3 \overset{\text{def}}{=} 2X = 2^{I+1} \land I + 1 \leq N \land N \geq 0 \quad \equiv \quad X = 2^I \land I < N \land N \geq 0
\]

\[
A : (X = 1 \land I = 0 \land N \geq 0) \Rightarrow P_1
\]

\[
B : (P_1 \land I \geq N) \Rightarrow (X = 2^N \land N = I \land N \geq 0)
\]

\[
C : P_3 \iff (P_1 \land I < N)
\]
Proof tree example

\[ s \overset{\text{def}}{=} \text{while } l \neq 0 \text{ do } l \leftarrow l - 1 \]

\[
\begin{align*}
\{\text{true}\} & \quad l \leftarrow l - 1 \quad \{\text{true}\} \\
\{l \neq 0\} & \quad l \leftarrow l - 1 \quad \{\text{true}\}
\end{align*}
\]

\[
\{\text{true}\} \quad \text{while } l \neq 0 \text{ do } l \leftarrow l - 1 \quad \{\text{true} \land \neg(l \neq 0)\}
\]

\[
\{\text{true}\} \quad \text{while } l \neq 0 \text{ do } l \leftarrow l - 1 \quad \{l = 0\}
\]

- in some cases, the program does not terminate
  (if the program starts with \( l < 0 \))

- the same proof holds for:
  \( \{\text{true}\} \quad \text{while } l \neq 0 \text{ do } J \leftarrow J - 1 \quad \{l = 0\} \)

- anything can be proven of a program that never terminates:

\[
\begin{align*}
\{l = 1 \land l \neq 0\} & \quad J \leftarrow J - 1 \quad \{l = 1\} \\
\{l = 1\} & \quad \text{while } l \neq 0 \text{ do } J \leftarrow J - 1 \quad \{l = 1 \land l = 0\}
\end{align*}
\]

\[
\{l = 1\} \quad \text{while } l \neq 0 \text{ do } J \leftarrow J - 1 \quad \{\text{false}\}
\]
Example: we wish to prove:

\[
\{ X = Y = 0 \} \textbf{ while } X < 10 \textbf{ do } (X \leftarrow X + 1; \ Y \leftarrow Y + 1) \{ X = Y = 10 \}
\]

we need to find an invariant assertion \( P \) for the \textbf{while} rule

**Incorrect invariant:** \( P \overset{\text{def}}{=} X, Y \in [0, 10] \)

- \( P \) indeed holds at each loop iteration \((P \) is an invariant\)
- but \( \{ P \land (X < 10) \} \ X \leftarrow X + 1; \ Y \leftarrow Y + 1 \{ P \} \) does not hold

\( P \land X < 10 \) does not prevent \( Y = 10 \) after \( Y \leftarrow Y + 1 \), \( P \) does not hold anymore
Example: we wish to prove:

\[ \{ X = Y = 0 \} \textbf{while } X < 10 \textbf{ do } (X \leftarrow X + 1; \ Y \leftarrow Y + 1) \{ X = Y = 10 \} \]

we need to find an invariant assertion \( P \) for the \textbf{while} rule

**Correct invariant:** \( P' \overset{\text{def}}{=} X \in [0, 10] \land X = Y \)

- \( P' \) also holds at each loop iteration \( (P' \text{ is an invariant}) \)
- \( \{ P' \land (X < 10) \} X \leftarrow X + 1; \ Y \leftarrow Y + 1 \{ P' \} \) can be proven
- \( P' \) is an \textbf{inductive invariant}  
  \( \text{(passes to the induction, stable by a loop iteration)} \)

\[ \implies \]

to prove a loop invariant  
 it is often necessary to find a \textbf{stronger} inductive loop invariant
Auxiliary variables:

Auxiliary variables: mathematical variables that do not appear in the program they are constant during program execution

Applications:

- simplify proofs
- express more properties (contracts, input-output relations)
- achieve (relative) completeness on extended languages (concurrency, recursive procedures)

Example:

\[ \{ X = x \land Y = y \} \text{ if } X < Y \text{ then } Y \leftarrow X \text{ else skip } \{ Y = \min(x, y) \} \]

- \( x \) and \( y \) retain the values of \( X \) and \( Y \) from the program entry
- \( Y = \min(X, Y) \) is much less useful as a specification of a \( \min \) function

"\{true\} \text{ if } X < Y \text{ then } Y \leftarrow X \text{ else skip } \{ Y = \min(X, Y) \}" holds, but "\{true\} \( X \leftarrow Y + 1 \text{ } \{ Y = \min(X, Y) \} \)" also holds
Floyd–Hoare logic

Link with denotational semantics

**Reminder:** \( S[\text{ stat }] : \mathcal{P}(\mathcal{E}) \rightarrow \mathcal{P}(\mathcal{E}) \) where \( \mathcal{E} \overset{\text{def}}{=} \forall \leftrightarrow \emptyset \)

\( S[\text{ skip }] R \overset{\text{def}}{=} R \)

\( S[\text{ fail }] R \overset{\text{def}}{=} \emptyset \)

\( S[ s_1; s_2 ] \overset{\text{def}}{=} S[ s_2 ] \circ S[ s_1 ] \)

\( S[ X \leftarrow e ] R \overset{\text{def}}{=} \{ \rho[X \mapsto v] | \rho \in R, v \in E[ e ] \rho \} \)

\( S[ \text{ if } e \text{ then } s_1 \text{ else } s_2 ] R \overset{\text{def}}{=} S[ s_1 ] \{ \rho \in R | \text{true} \in E[ e ] \rho \} \cup S[ s_2 ] \{ \rho \in R | \text{false} \in E[ e ] \rho \} \)

\( S[ \text{ while } e \text{ do } s ] R \overset{\text{def}}{=} \{ \rho \in \text{lf} \rho F | \text{false} \in E[ e ] \rho \} \)

where \( F(X) \overset{\text{def}}{=} R \cup S[ s ] \{ \rho \in X | \text{true} \in E[ e ] \rho \} \)

**Theorem**

\( \{ P \} \ c \ \{ Q \} \overset{\text{def}}{\leftrightarrow} \forall R \subseteq \mathcal{E}: R \models P \implies S[ c ] R \models Q \)

\( (A \models P \text{ means } \forall \rho \in A, \text{ the formula } P \text{ is true on the variable assignment } \rho) \)
Floyd–Hoare logic

Link with denotational semantics

- Hoare logic reasons on formulas
- denotational semantics reasons on state sets

we can assimilate assertion formulas and state sets
(logical abuse: we assimilate formulas and models)

let \([R]\) be any formula representing the set \(R\), then:

- \([[R]] \ c \ {[S\[c]\] R}]\) is always valid
- \([[R]] \ c \ {[R']}\) \(\Rightarrow\) \(S\[c]\ R \subseteq R'\)

\(\Rightarrow\) \([S\[c]\ R]\) provides the best valid postcondition
Link with denotational semantics

**Loop invariants**

- **Hoare:**
  to prove \( \{ P \} \textbf{while} e \textbf{do} s \{ P \land \neg e \} \) we must prove \( \{ P \land e \} \textbf{do} s \{ P \} \)
  i.e., \( P \) is an **inductive invariant**

- **Denotational semantics:**
  we must find \( \text{lfp} \ F \) where \( F(X) \stackrel{\text{def}}{=} R \cup S[\{s\}] \{ \rho \in X \mid \rho \models e \} \)
  - \( \text{lfp} \ F = \cap \{ X \mid F(X) \subseteq X \} \) (Tarski’s theorem)
  - \( F(X) \subseteq X \iff ([R] \Rightarrow [X]) \land \{[X \land e]\} \textbf{do} s \{[X]\} \)
    - \( R \subseteq X \) means \([R] \Rightarrow [X]\),
    - \( S[\{s\}] \{ \rho \in X \mid \rho \models e \} \subseteq X \) means \([X \land e]\) \textbf{do} s \{[X]\}

As a consequence:

- any \( X \) such that \( F(X) \subseteq X \) gives an inductive invariant
- \( \text{lfp} \ F \) gives the best inductive invariant
- any \( X \) such that \( \text{lfp} \ F \subseteq X \) gives an invariant
  (not necessarily inductive)

(see [Cousot02])
Predicate calculus
Dijkstra’s weakest liberal preconditions

**Principle:** predicate calculus
- calculus to derive preconditions from postconditions
- order and mechanize the search for intermediate assertions
  (easier to go backwards, mainly due to assignments)

**Weakest liberal precondition** \( \text{wlp} : (\text{prog} \times \text{Prop}) \rightarrow \text{Prop} \)

\( \text{wlp}(c, P) \) is the weakest, i.e. most general, precondition ensuring that \( \{ \text{wlp}(c, P) \} \ c \ \{P\} \) is a Hoare triple

(greatest state set that ensures that the computation ends up in \( P \))

formally: \( \{P\} \ c \ \{Q\} \iff (P \Rightarrow \text{wlp}(c, Q)) \)

“liberal” means that we do not care about termination and errors

Examples:

\[
\begin{align*}
\text{wlp}(X \leftarrow X + 1, X = 1) &= \\
\text{wlp(while } X < 0 \ X \leftarrow X + 1, X \geq 0) &= \\
\text{wlp(while } X \neq 0 \ X \leftarrow X + 1, X \geq 0) &=
\end{align*}
\]

(introduced in [Dijkstra75])
Dijkstra’s weakest liberal preconditions

**Principle:** predicate calculus
- calculus to derive preconditions from postconditions
- order and mechanize the search for intermediate assertions
  (easier to go backwards, mainly due to assignments)

**Weakest liberal precondition** \( wlp : (\text{prog} \times \text{Prop}) \rightarrow \text{Prop} \)

\( wlp(c, P) \) is the weakest, i.e. most general, precondition ensuring that \( \{ wlp(c, P) \} \ c \ \{ P \} \) is a Hoare triple
(greatest state set that ensures that the computation ends up in \( P \))

formally: \( \{ P \} \ c \ \{ Q \} \iff (P \Rightarrow wlp(c, Q)) \)

“liberal” means that we do not care about termination and errors

Examples:
\[
\begin{align*}
wlp(X \leftarrow X + 1, \ X = 1) &= (X = 0) \\
wlp(\text{while } X < 0 \ X \leftarrow X + 1, \ X \geq 0) &= \text{true} \\
wlp(\text{while } X \neq 0 \ X \leftarrow X + 1, \ X \geq 0) &= \text{true}
\end{align*}
\]
(introduced in [Dijkstra75])
A calculus for \textit{wlp}

\textit{wlp} is defined by induction on the syntax of programs:

\[
\text{wlp}(\text{skip}, P) \overset{\text{def}}{=} P \\
\text{wlp}(\text{fail}, P) \overset{\text{def}}{=} \text{true} \\
\text{wlp}(X \leftarrow e, P) \overset{\text{def}}{=} P[e/X] \\
\text{wlp}(s; t, P) \overset{\text{def}}{=} \text{wlp}(s, \text{wlp}(t, P)) \\
\text{wlp}(\text{if } e \text{ then } s \text{ else } t, P) \overset{\text{def}}{=} (e \Rightarrow \text{wlp}(s, P)) \land (\neg e \Rightarrow \text{wlp}(t, P)) \\
\text{wlp}(\text{while } e \text{ do } s, P) \overset{\text{def}}{=} I \land ((e \land I) \Rightarrow \text{wlp}(s, I)) \land ((\neg e \land I) \Rightarrow P)
\]

- \( e \Rightarrow Q \) is equivalent to \( Q \lor \neg e \)
  - weakest property that matches \( Q \) when \( e \) holds
  - but says nothing when \( e \) does not hold

- \textbf{while} loops require providing an \textbf{invariant predicate} \( I \)
  - intuitively, \textit{wlp} checks that \( I \) is an inductive invariant implying \( P \)
  - if so, it returns \( I \); otherwise, it returns false
  - \textit{wlp} is the weakest precondition only if \( I \) is well-chosen...
\[ \text{wlp}(\text{if } X < 0 \text{ then } Y \leftarrow -X \text{ else } Y \leftarrow X, \ Y \geq 10) = \]

\[ (X < 0 \Rightarrow \text{wlp}(Y \leftarrow -X, \ Y \geq 10)) \land (X \geq 0 \Rightarrow \text{wlp}(Y \leftarrow X, \ Y \geq 10)) \]

\[ (X < 0 \Rightarrow -X \geq 10) \land (X \geq 0 \Rightarrow X \geq 10) = \]

\[ (X \geq 0 \lor -X \geq 10) \land (X < 0 \lor X \geq 10) = \]

\[ X \geq 10 \lor X \leq -10 \]

\text{wlp} \text{ generates complex formulas}
\text{it is important to simplify them from time to time}
Properties of \( wlp \)

- \( wlp(c, \text{false}) \equiv \text{false} \) (excluded miracle)

- \( wlp(c, P) \land wlp(d, Q) \equiv wlp(c, P \land Q) \) (distributivity)

- \( wlp(c, P) \lor wlp(d, Q) \equiv wlp(c, P \lor Q) \) (distributivity)
  
    (\( \Rightarrow \) always true, \( \Leftarrow \) only true for deterministic, error-free programs)

- if \( P \Rightarrow Q \), then \( wlp(c, P) \Rightarrow wlp(c, Q) \) (monotonicity)

\( A \equiv B \) means that the formulas \( A \) and \( B \) are equivalent

i.e., \( \forall \rho: \rho \models A \iff \rho \models B \)

(stronger than syntactic equality)
Strongest liberal postconditions

we can define \( slp : (\text{Prop} \times \text{prog}) \rightarrow \text{Prop} \)

\[
\{ P \} \ c \ \{ slp(P, c) \}
\]

(postcondition)

\[
\{ P \} \ c \ \{ Q \} \iff (slp(P, c) \Rightarrow Q)
\]

(strongest postcondition)

(corresponds to the smallest state set)

\( slp(P, c) \) does not care about non-termination

(liberal)

allows forward reasoning

we have a duality:

\[
(P \Rightarrow wlp(c, Q)) \iff (slp(P, c) \Rightarrow Q)
\]

proof: \( (P \Rightarrow wlp(c, Q)) \iff \{ P \} \ c \ \{ Q \} \iff (slp(P, c) \Rightarrow Q) \)
Calculus for slp

\[ slp(P, \text{skip}) \stackrel{\text{def}}{=} P \]

\[ slp(P, \text{fail}) \stackrel{\text{def}}{=} \text{false} \]

\[ slp(P, X \leftarrow e) \stackrel{\text{def}}{=} \exists v: P[v/X] \land X = e[v/X] \]

\[ slp(P, s; t) \stackrel{\text{def}}{=} slp(slp(P, s), t) \]

\[ slp(P, \text{if } e \text{ then } s \text{ else } t) \stackrel{\text{def}}{=} slp(P \land e, s) \lor slp(P \land \neg e, t) \]

\[ slp(P, \text{while } e \text{ do } s) \stackrel{\text{def}}{=} (P \Rightarrow I) \land (slp(I \land e, s) \Rightarrow I) \land (\neg e \land I) \]

(the rule for \( X \leftarrow e \) makes \( slp \) much less attractive than \( wlp \))
Verification conditions
How can we automate program verification using logic?

- **Hoare logic**: deductive system
  - can only automate the checking of proofs

- **Predicate transformers**: \( wlp \), \( slp \) calculus
  - construct (big) formulas mechanically
  - invention is still needed for loops

- **Verification condition generation**
  - take as input a program with annotations
    (at least contracts and loop invariants)
  - generate mechanically logic formulas ensuring the correctness
    (reduction to a mathematical problem, no longer any reference to a program)
  - use an automatic SAT/SMT solver to prove (discharge) the formulas
  or an interactive theorem prover

(the idea of logic-based automated verification appears as early as \([\text{King69}]\))
Language

\[
\begin{align*}
\text{stat} & ::= \ X \leftarrow \text{expr} \\
& \mid \text{skip} \\
& \mid \text{stat; stat} \\
& \mid \text{if expr then stat else stat} \\
& \mid \text{while \{ Prop\} expr do stat} \\
& \mid \text{assert expr}
\end{align*}
\]

\[
\begin{align*}
\text{prog} & ::= \ \{ \text{Prop}\} \text{stat} \{ \text{Prop}\}
\end{align*}
\]

- loops are annotated with loop invariants
- optional assertions at any point
- programs are annotated with a contract
  (precondition and postcondition)
Verification condition generation algorithm

\[ \text{vcg}_p : \text{prog} \rightarrow \mathcal{P}(\text{Prop}) \]

\[ \text{vcg}_p(\{P\} \ c \ \{Q\}) \overset{\text{def}}{=} \text{let } (R, C) = \text{vcg}_s(c, Q) \text{ in } C \cup \{P \Rightarrow R\} \]

\[ \text{vcg}_s : (\text{stat} \times \text{Prop}) \rightarrow (\text{Prop} \times \mathcal{P}(\text{Prop})) \]

\[ \text{vcg}_s(\text{skip}, Q) \overset{\text{def}}{=} (Q, \emptyset) \]

\[ \text{vcg}_s(X \leftarrow e, Q) \overset{\text{def}}{=} (Q[e/X], \emptyset) \]

\[ \text{vcg}_s(s; t, Q) \overset{\text{def}}{=} \text{let } (R, C) = \text{vcg}_s(t, Q) \text{ in let } (P, D) = \text{vcg}_s(s, R) \text{ in } (P, C \cup D) \]

\[ \text{vcg}_s(\text{if } e \text{ then } s \text{ else } t, Q) \overset{\text{def}}{=} \]

\[ \text{let } (S, C) = \text{vcg}_s(s, Q) \text{ in let } (T, D) = \text{vcg}_s(t, Q) \text{ in } ((e \Rightarrow S) \land (\neg e \Rightarrow T), C \cup D) \]

\[ \text{vcg}_s(\text{while } \{I\} e \text{ do } s, Q) \overset{\text{def}}{=} \]

\[ \text{let } (R, C) = \text{vcg}_s(s, I) \text{ in } (I, C \cup \{(I \land e) \Rightarrow R, (I \land \neg e) \Rightarrow Q\}) \]

\[ \text{vcg}_s(\text{assert } e, Q) \overset{\text{def}}{=} (e \Rightarrow Q, \emptyset) \]

We use \textit{wlp} to infer assertions automatically when possible.

\[ \text{vcg}_s(c, P) = (P', C) \text{ propagates postconditions backwards and accumulates into } C \text{ verification conditions (from loops).} \]
Verification condition generation example

Consider the program:

$$\{N \geq 0\} \quad X \leftarrow 1; \ I \leftarrow 0;$$

while $$\{X = 2^I \land 0 \leq I \leq N\} \ I < N$$ do

$$\quad (X \leftarrow 2X; \ I \leftarrow I + 1)$$

$$\{X = 2^N\}$$

we get three verification conditions:

$$C_1 \overset{\text{def}}{=} (X = 2^I \land 0 \leq I \leq N) \land I \geq N \Rightarrow X = 2^N$$

$$C_2 \overset{\text{def}}{=} (X = 2^I \land 0 \leq I \leq N) \land I < N \Rightarrow 2X = 2^{I+1} \land 0 \leq I + 1 \leq N$$

(from $$(X = 2^I \land 0 \leq I \leq N)[I + 1/I, 2X/X]$$)

$$C_3 \overset{\text{def}}{=} N \geq 0 \Rightarrow 1 = 2^0 \land 0 \leq 0 \leq N$$

(from $$(X = 2^I \land 0 \leq I \leq N)[0/I, 1/X]$$)

which can be checked independently
What about real languages?

In a real language such as C, the rules are not so simple

Example: the assignment rule

\[
\{ P[e/X] \} \; X \leftarrow e \{ P \}
\]

requires that

- \( e \) has no effect (memory write, function calls)
- there is no pointer aliasing
- \( e \) has no run-time error

moreover, the operations in the program and in the logic may not match:

- integers: logic models \( \mathbb{Z} \), computers use \( \mathbb{Z}/2^n\mathbb{Z} \) (wrap-around)
- continuous:
  - logic models \( \mathbb{Q} \) or \( \mathbb{R} \), programs use floating-point numbers (rounding error)
  - a logic for pointers and dynamic allocation is also required (separation logic)

(see for instance the tool Why, to see how some problems can be circumvented)
Termination
Total correctness

**Hoare triple:** \([P] \text{ prog } [Q]\)
- if \(P\) holds before \text{ prog} is executed
- then \text{ prog} always terminates
- and \(Q\) holds after the execution of \text{ prog}

**Rules:** we only need to change the rule for \textbf{while}

\[
\forall t \in W: [P \land e \land u = t] \implies [P \land u < t] \\
\hline
[|P|] \text{ while } e \text{ do } s [P \land \neg e]
\]

\((W, \prec)\) well-founded \(\iff\) every \(V \subseteq W, V \neq \emptyset\) has a minimal element for \(\prec\)
ensures that we cannot decrease infinitely by \(\prec\) in \(W\)
generally, we simply use \((\mathbb{N}, <)\)
(also useful: lexicographic orders, ordinals)

- in addition to the loop invariant \(P\)
we invent an expression \(u\) that strictly decreases by \(s\)
\(u\) is called a “ranking function”
only often \(\neg e \implies u = 0\): \(u\) counts the number of steps until termination
To simplify, we can decompose a proof of total correctness into:

- a proof of partial correctness $\{P\} \ c \ \{Q\}$ ignoring termination
- a proof of termination $[P] \ c \ [true]$ ignoring the specification

we must still include the precondition $P$ as the program may not terminate for all inputs.

Indeed, we have:

$$
\frac{\{P\} \ c \ \{Q\} \quad [P] \ c \ [true]}{[P] \ c \ [Q]}
$$
Total correctness example

We use a simpler rule for integer ranking functions \( ((W, \prec) \overset{\text{def}}{=} (\mathbb{N}, \leq)) \) using an integer expression \( r \) over program variables:

\[
\forall n: [P \land e \land (r = n)] \quad s \quad [P \land (r < n)] \quad (P \land e) \Rightarrow (r \geq 0)
\]

\[
[P] \quad \textbf{while} \quad e \quad \textbf{do} \quad s \quad [P \land \neg e]
\]

Example: \( p \overset{\text{def}}{=} \textbf{while} \quad I < N \quad \textbf{do} \quad I \leftarrow I + 1; \quad X \leftarrow 2X \quad \textbf{done} \)

we use \( r \overset{\text{def}}{=} N - I \) and \( P \overset{\text{def}}{=} \text{true} \)

\[
\forall n: [I \land N - I = n] \quad I \leftarrow I + 1; \quad X \leftarrow 2X \quad [N - I = n - 1]
\]

\[
I \land N \Rightarrow N - I \geq 0
\]

\[
\text{[true]} \quad p \quad [I \geq N]
\]
Weakest precondition

Weakest precondition \( wp(prog, \text{Prop}) : \text{Prop} \)

- similar to \( wlp \), but also additionally imposes termination
- \( [P] c [Q] \iff (P \Rightarrow wp(c, Q)) \)

As before, only the definition for \textbf{while} needs to be modified:

\[
wp(\text{while } e \text{ do } s, P) \overset{\text{def}}{=} I \land (I \Rightarrow v \geq 0) \land \\
\forall n: ((e \land I \land v = n) \Rightarrow wp(s, I \land v < n)) \land \\
((\neg e \land I) \Rightarrow P)
\]

the invariant predicate \( I \) is combined with a \textbf{variant expression} \( v \)

- \( v \) is positive (this is an invariant: \( I \Rightarrow v \geq 0 \))
- \( v \) decreases at each loop iteration

and similarly for strongest postconditions
Non-determinism
Non-determinism in Hoare logic

We model non-determinism with the statement $X \leftarrow ?$ meaning: $X$ is assigned a random value

($X \leftarrow [a, b]$ can be modeled as: $X \leftarrow ?;\text{ if } X < a \lor X > b \text{ then fail;}$)

**Hoare axiom:**

$$\{\forall X : P\} \quad X \leftarrow ? \quad \{P\}$$

If $P$ is true after assigning $X$ to random then $P$ must hold whatever the value of $X$ before

Often, $X$ does not appear in $P$ and we get simply:

$$\{P\} \quad X \leftarrow ? \quad \{P\}$$

**Example:**

$$\begin{align*}
\{X = x\} \quad Y &\leftarrow X \quad \{Y = x\} \\
\{Y = x\} \quad X &\leftarrow ? \quad \{Y = x\} \\
\{Y = x\} \quad X &\leftarrow Y \quad \{X = x\} \\
\{X = x\} \quad Y &\leftarrow X; \quad X \leftarrow ?; \quad X \leftarrow Y \quad \{X = x\}
\end{align*}$$
Non-determinism in predicate calculus

**Predicate transformers:**

- \( \text{wlp}(X \leftarrow ?, P) \overset{\text{def}}{=} \forall X: P \)  
  
  \((P \text{ must hold whatever the value of } X \text{ before the assignment})\)

- \( \text{slp}(P, X \leftarrow ?) \overset{\text{def}}{=} \exists X: P \)  
  
  \((\text{if } P \text{ held for one value of } X, P \text{ holds for all values of } X \text{ after the assignment})\)

**Link with operational semantics** (as transition systems)

predicates \( P \) as sets of states \( P \subseteq \Sigma \)
commands \( c \) as transition relations \( c \subseteq \Sigma \times \Sigma \)

we define:  
\[
\text{post}[\tau](P) \overset{\text{def}}{=} \{ \sigma' | \exists \sigma \in P: (\sigma, \sigma') \in \tau \} \\
\text{pre}[\tau](P) \overset{\text{def}}{=} \{ \sigma | \forall \sigma' \in \Sigma: (\sigma, \sigma') \in \tau \implies \sigma' \in P \}
\]

then:  
\( \text{slp}(P, c) = \text{post}[c](P) \)  
\( \text{wlp}(c, P) = \text{pre}[c](P) \)
Arrays
Arrays

Array syntax

We enrich our language with:

- a set $\mathcal{A}$ of array variables
- array access in expressions: $A(expr)$, $A \in \mathcal{A}$
- array assignment: $A(expr) \leftarrow expr$, $A \in \mathcal{A}$
  (arrays have unbounded size here, we do not care about overflow)

**Issue:**

a natural idea is to generalize the assignment axiom:

$$\{P[f/A(e)]\} \ A(e) \leftarrow f \ \{P\}$$

but this is not sound, due to aliasing

example:

we would derive the invalid triple: $$\{A(J) = 1 \land I = J\} \ A(I) \leftarrow 0 \ \{A(J) = 1 \land I = J\}$$
as $(A(J) = 1)[0/A(I)] = (A(J) = 1)$
Solution: use a specific theory of arrays (McCarthy 1962)

- enrich the assertion language with expressions $A\{e \mapsto f\}$
  meaning: the array equal to $A$ except that index $e$ maps to value $f$

- add the axiom

\[
\{P[A\{e \mapsto f\} / A]\} \quad A(e) \leftarrow f \quad \{P\}
\]

intuitively, we use “functional arrays” in the logic world

- add logical axioms to reason about our arrays in assertions

\[
A\{e \mapsto f\}(e) = f \quad (e \neq e') \Rightarrow (A\{e \mapsto f\}(e') = A(e'))
\]
Example: swap

given the program \( p \overset{\text{def}}{=} T \leftarrow A(I); \ A(I) \leftarrow A(J); \ A(J) \leftarrow T \)

we wish to prove: \( \{A(I) = x \land A(J) = y\} \ p \ \{A(I) = y \land A(J) = x\} \)

by propagating \( A(I) = y \) backwards by the assignment rule, we get
\[
\begin{align*}
A\{ J \mapsto T \}(I) &= y \\
A\{ I \mapsto A(J) \}\{ J \mapsto T \}(I) &= y \\
A\{ I \mapsto A(J) \}\{ J \mapsto A(I) \}(I) &= y
\end{align*}
\]

we consider two cases:

if \( I = J \), then \( A\{ I \mapsto A(J) \}\{ J \mapsto A(I) \} = A \)
so, \( A\{ I \mapsto A(J) \}\{ J \mapsto A(I) \}(I) = A(I) = A(J) \)

if \( I \neq J \), then \( A\{ I \mapsto A(J) \}\{ J \mapsto A(I) \}(I) = A\{ I \mapsto A(J) \}(I) = A(J) \)
in both cases, we get \( A(J) = y \) in the precondition

likewise, \( A(I) = x \) in the precondition
Concurrent programs
Concurrent programs

Concurrent program syntax

Language

add a parallel composition statement: $\text{stat} || \text{stat}$

semantics: $s_1 || s_2$

- execute $s_1$ and $s_2$ in parallel
- allowing an arbitrary interleaving of atomic statements
  (expression evaluation or assignments)
- terminates when both $s_1$ and $s_2$ terminate

Hoare logic: extended by Owicki and Gries [Owicki76]

first idea: $\begin{align*}
\{P_1\} s_1 \{Q_1\} \quad \{P_2\} s_2 \{Q_2\} \\
\{P_1 \land P_2\} s_1 || s_2 \{Q_1 \land Q_2\}
\end{align*}$

but this is unsound
Concurrent programs: rule soundness

**Issue:**

\[
\begin{array}{c}
\{ P_1 \} \ s_1 \ \{ Q_1 \} \\
\{ P_2 \} \ s_2 \ \{ Q_2 \}
\end{array}
\]

\[
\{ P_1 \land P_2 \} \ s_1 \ || \ s_2 \ \{ Q_1 \land Q_2 \}
\]

is not always sound

example:

given \( s_1 \ \overset{\text{def}}{=} \ X \leftarrow 1 \) and \( s_2 \ \overset{\text{def}}{=} \ \text{if } X = 0 \text{ then } Y \leftarrow 1 \), we derive:

\[
\begin{array}{c}
\{ X = Y = 0 \} \ s_1 \ \{ X = 1 \land Y = 0 \} \\
\{ X = 1 \land Y = 0 \} \ s_1 \ || \ s_2 \ \{ X = Y = 0 \land Y = 1 \}
\end{array}
\]

\[
\{ X = Y = 0 \} \ s_1 \ || \ s_2 \ \{ \text{false} \}
\]

**Solution:**

the proofs of \( \{ P_1 \} \ s_1 \ \{ Q_1 \} \) and \( \{ P_2 \} \ s_2 \ \{ Q_2 \} \) must not interfere
Concurrent programs: rule soundness

interference freedom

given proofs $\Delta_1$ and $\Delta_2$ of $\{P_1\} s_1 \{Q_1\}$ and $\{P_2\} s_2 \{Q_2\}$

$\Delta_1$ does not interfere with $\Delta_2$ if:

for any $\Phi$ appearing before a statement in $\Delta_1$

for any $\{P'_2\} s'_2 \{Q'_2\}$ appearing in $\Delta_2$

$\{\Phi \land P'_2\} s'_2 \{\Phi\}$ holds

and moreover $\{Q_1 \land P'_2\} s'_2 \{Q_1\}$

i.e.: the assertions used to prove $\{P_1\} s_1 \{Q_1\}$ are stable by $s_2$

example:

given $s_1 \overset{\text{def}}{=} X \leftarrow 1$ and $s_2 \overset{\text{def}}{=} \text{if } X = 0 \text{ then } Y \leftarrow 1$, we derive:

$\{X = 0 \land Y \in [0, 1]\} s_1 \{X = 1 \land Y \in [0, 1]\}$

$\{X \in [0, 1] \land Y = 0\} s_2 \{X \in [0, 1] \land Y \in [0, 1]\}$

$\{X = Y = 0\} s_1 \parallel s_2 \{X = 1 \land Y \in [0, 1]\}$
Concurrent programs: rule completeness

**Issue:** incompleteness

\[
\{X = 0\} \ X \leftarrow X + 1 \ || \ X \leftarrow X + 1 \ \{X = 2\} \text{ is valid}
\]

but no proof of it can be derived

**Solution:** auxiliary variables

introduce explicitly program points and program counters

example:

\[
\ell_1 X \leftarrow X + 1 \ || \ \ell_2 X \leftarrow X + 1 \ \ell_3 X \leftarrow X + 1 \ \ell_4
\]

with auxiliary variables \(pc_1 \in \{1, 2\}\), \(pc_2 \in \{3, 4\}\)

we can now express that a process is at a given control point and distinguish assertions based on the location of other processes

\[
s_1 \overset{\text{def}}{=} \ell_1 X \leftarrow X + 1 \ \ell_2, \ s_2 \overset{\text{def}}{=} \ell_3 X \leftarrow X + 1 \ \ell_4
\]

\[
\{(pc_2 = 3 \land X = 0) \lor (pc_2 = 4 \land X = 1)\} \ s_1 \ \{(pc_2 = 3 \land X = 1) \lor (pc_2 = 4 \land X = 2)\}
\]

\[
\{(pc_1 = 1 \land X = 0) \lor (pc_1 = 2 \land X = 1)\} \ s_2 \ \{(pc_1 = 1 \land X = 1) \lor (pc_1 = 2 \land X = 2)\}
\]

\[
\implies \{pc_1 = 1 \land pc_2 = 3 \land X = 0\} \ s_1 \ || \ s_2 \ \{pc_1 = 2 \land pc_2 = 4 \land X = 1\}
\]

in fact, auxiliary variables make the proof method complete
Conclusion
Conclusion

- logic allows us to reason about program correctness
- verification can be reduced to proofs of simple logic statements

**Issue: automation**

- annotations are required (loop invariants, contracts)
- verification conditions must be proven

To scale up to realistic programs, we need to automate as much as possible

**Some solutions:**

- automatic logic solvers to discharge proof obligations
  - SAT / SMT solvers
- abstract interpretation to approximate the semantics
  - fully automatic
  - able to infer invariants


