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 ≃ 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: function specification

```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

```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
- 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:** \( \{ P \} \; \text{prog} \; \{ Q \} \)

- \( \text{prog} \) is a program fragment
- \( P \) and \( Q \) are logical assertions over program variables
  
  (e.g. \( P \overset{\text{def}}{=} (X \geq 0 \land Y \geq 0) \lor (X < 0 \land Y < 0) \))

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

\( P \) is the **precondition**, \( Q \) is the **postcondition**

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

(does not rule out errors and non-termination)

Hoare triples serve as **judgements** in a proof system

(introduced in [Hoare69])
## Language

\[
\text{stat} ::= \quad X \leftarrow \text{expr} \quad \text{(assignment)} \\
| \quad \text{skip} \quad \text{(do nothing)} \\
| \quad \text{fail} \quad \text{(error)} \\
| \quad \text{stat}; \text{stat} \quad \text{(sequence)} \\
| \quad \text{if expr then stat else stat} \quad \text{(conditional)} \\
| \quad \text{while expr do stat} \quad \text{(loop)} \\
\]

- \( X \in \mathbb{V} \): integer-valued variables
- \( expr \): integer arithmetic expressions

We assume that:

- expressions are deterministic (for now)
- expression evaluation does not cause error (only \text{fail} does)

For instance, to avoid divisions by zero, we assume that all divisions are explicitly guarded as in: \( \text{if } X = 0 \text{ then fail else } \cdots /X \cdots \)
Axioms:

\[
\begin{align*}
\{P\} \text{skip} \{P\} \\
\{P\} \text{fail} \{Q\}
\end{align*}
\]

- any property true before \textbf{skip} is true afterwards
- any property is true after \textbf{fail}
Assignment axiom:

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

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:

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

**Rule of consequence:**

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

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 \( \{\text{true}\} \ \text{fail} \ \{\text{false}\} \) (as \( P \Rightarrow \text{true} \) and \( \text{false} \Rightarrow \) 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:**

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

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:

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

\[
\{X = 1 \land Y = 1\} \ X \leftarrow X + 1 \ \{X = 2 \land Y = 1\} \ Y \leftarrow Y - 1 \ \{X = 2 \land Y = 0\}
\]
Hoare rules: loops

Loops:

\[
\begin{align*}
\{ P \land e \} & \ s \ \{ P \} \\
\{ P \} & \ \text{while} \ e \ \text{do} \ s \ \{ P \land \lnot e \}
\end{align*}
\]

\( P \) is a loop invariant:
\( P \) holds before each loop iteration, before even testing \( e \)

Practical use:
actually, we would rather prove the triple: \( \{ P \} \ \text{while} \ e \ \text{do} \ s \ \{ Q \} \)
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: \( (I \land \lnot e) \Rightarrow Q \)

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

we can derive the rule:
\[ \{ P \} \ \text{while} \ e \ \text{do} \ s \ \{ Q \} \]
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 sense: $\Leftrightarrow$)

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**

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

\[\{\text{true}\} \text{fail} \{\text{false}\}\]

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

\[\{P\} s \{R\} \quad \{R\} t \{Q\}\]

\[\{P\} \text{;} s; t \{Q\}\]

\[\{P \land e\} s \{Q\}\]

\[\text{if } e \text{ then } s \text{ else } t \{Q\}\]

\[\{P \land e\} s \{P\}\]

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

\[P \Rightarrow P'\]

\[Q' \Rightarrow Q\]

\[\{P'\} c \{Q'\}\]

\[\{P\} c \{Q\}\]
Proof tree example

\[
\begin{align*}
& s \overset{\text{def}}{=} \text{while } l < N \text{ do } (X \leftarrow 2X; \ l \leftarrow l + 1) \\
& \begin{array}{c}
C \\
\{ P_3 \} X \leftarrow 2X \quad \{ P_2 \} \\
\{ P_1 \land l < N \} X \leftarrow 2X; \ l \leftarrow l + 1 \quad \{ P_1 \}
\end{array} \\
& \begin{array}{c}
A \quad B \\
\{ P_1 \} s \quad \{ P_1 \land l \geq N \} \\
\{ X = 1 \land l = 0 \land N \geq 0 \} s \quad \{ X = 2^N \land N = l \land N \geq 0 \}
\end{array}
\end{align*}
\]

\[
\begin{align*}
P_1 & \overset{\text{def}}{=} X = 2^l \land l \leq N \land N \geq 0 \\
P_2 & \overset{\text{def}}{=} X = 2^{l+1} \land l+1 \leq N \land N \geq 0 \\
P_3 & \overset{\text{def}}{=} 2X = 2^{l+1} \land l+1 \leq N \land N \geq 0 \quad \equiv X = 2^l \land l < N \land N \geq 0 \\
A : & (X = 1 \land l = 0 \land N \geq 0) \Rightarrow P_1 \\
B : & (P_1 \land l \geq N) \Rightarrow (X = 2^N \land N = l \land N \geq 0) \\
C : & P_3 \iff (P_1 \land l < N)
\end{align*}
\]
**Proof tree example**

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

\[
\begin{array}{c}
\{\text{true}\} \quad l \leftarrow l - 1 \quad \{\text{true}\} \\
\{l \neq 0\} \quad l \leftarrow l - 1 \quad \{\text{true}\}
\end{array}
\]

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

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

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

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

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

\[
\begin{array}{c}
\{l = 1 \land l \neq 0\} \quad J \leftarrow J - 1 \quad \{l = 1\} \\
\{l = 1\} \quad \textbf{while} l \neq 0 \textbf{ do } J \leftarrow J - 1 \quad \{l = 1 \land l = 0\}
\end{array}
\]

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

\textbf{Incorrect invariant:} \quad P \overset{\text{def}}{=} X, Y \in [0, 10]

- \( P \) indeed holds at each loop iteration \((P \text{ is an invariant})\)
- but \( \{P \land (X < 10)\} X \leftarrow X + 1; \ Y \leftarrow Y + 1 \{P\} \)
  does not hold
  \[ P \land X < 10 \text{ does not prevent } Y = 10 \]
  after \( Y \leftarrow Y + 1, \ P \text{ does not hold anymore} \)
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 \( (P' \) is an invariant)\)
- \( \{ P' \land (X < 10) \} X \leftarrow X + 1; \ Y \leftarrow Y + 1 \{ P' \} \) can be proven
- \( P' \) is an inductive invariant \( (passes \ to \ the \ induction, \ stable \ by \ a \ loop \ iteration) \)

\[ \implies \]

to prove a loop invariant

it is often necessary to find a stronger inductive loop invariant
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 \{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 \( \mathcal{E} \triangleq \forall \leftrightarrow \emptyset \)

\[ S[\text{skip}] R \overset{\text{def}}{=} R \]
\[ S[\text{fail}] R \overset{\text{def}}{=} \emptyset \]
\[ S[ s_1; s_2 ] \overset{\text{def}}{=} S[ s_2 ] \circ S[ s_1 ] \]
\[ S[ X \leftarrow e ] R \overset{\text{def}}{=} \{ \rho[X \mapsto v] \mid \rho \in R, v \in E[e] \rho \} \]
\[ S[ \text{if } e \text{ then } s_1 \text{ else } s_2 ] R \overset{\text{def}}{=} S[ s_1 ] \{ \rho \in R \mid \text{true} \in E[e] \rho \} \cup S[ s_2 ] \{ \rho \in R \mid \text{false} \in E[e] \rho \} \]
\[ S[ \text{while } e \text{ do } s ] R \overset{\text{def}}{=} \{ \rho \in \text{lfp } F \mid \text{false} \in E[e] \rho \} \]
where \( F(X) \overset{\text{def}}{=} R \cup S[ s ] \{ \rho \in X \mid \text{true} \in E[e] \rho \} \)

Theorem

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

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

Link with denotational semantics

Loop invariants

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

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

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** \( wlp : (prog \times Prop) \rightarrow Prop \)

\( wlp(c, P) \) 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:**

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

(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:**
- \( wlp(X \leftarrow X + 1, X = 1) = (X = 0) \)
- \( wlp(\text{while } X < 0 X \leftarrow X + 1, X \geq 0) = \text{true} \)
- \( wlp(\text{while } X \neq 0 X \leftarrow X + 1, X \geq 0) = \text{true} \)

(introduced in [Dijkstra75])
A calculus for wlp

\( \text{wlp} \) is defined by induction on the syntax of programs:

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

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

- **while** loops require providing an invariant predicate \( l \)
  - intuitively, \( \text{wlp} \) checks that \( l \) is an inductive invariant implying \( P \)
  - if so, it returns \( l \); otherwise, it returns false
  - \( \text{wlp} \) is the weakest precondition only if \( l \) is well-chosen...
\begin{align*}
\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
\end{align*}

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

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

- $\text{wlp}(c, P) \land \text{wlp}(d, Q) \equiv \text{wlp}(c, P \land Q)$  
  (distributivity)

- $\text{wlp}(c, P) \lor \text{wlp}(c, Q) \equiv \text{wlp}(c, P \lor Q)$  
  (distributivity)

  ($\Rightarrow$ always true, $\Leftarrow$ only true for deterministic, error-free programs)

- if $P \Rightarrow Q$, then $\text{wlp}(c, P) \Rightarrow \text{wlp}(c, Q)$  
  (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 \( slp : (Prop \times prog) \rightarrow Prop \)

- \( \{ P \} \ c \ \{ slp(P, c) \} \)  
  (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

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

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

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

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

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

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

(the rule for \( X \leftarrow e \) makes \( \text{slp} \) much less attractive than \( \text{wlp} \))
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 [King69])
Language

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

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

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

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

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

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

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

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

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

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

\[ \text{let} \; (S, C) = \text{vcg}_s(s, Q) \; \text{in} \; \text{let} \; (T, D) = \text{vcg}_s(t, Q) \; \text{in} \]

\[ (((e \Rightarrow S) \land (\neg e \Rightarrow T)), C \cup D) \]

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

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

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

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

\[ \text{vcg}_s(c, P) = (P', C) \] propagates postconditions backwards and accumulates into \( C \) verification conditions (from loops).
Consider the program:

\{ N \geq 0 \} \quad X \leftarrow 1; \; I \leftarrow 0;
\textbf{while} \; \{ X = 2^I \land 0 \leq I \leq N \} \; I < N \; \textbf{do}
\quad (X \leftarrow 2X; \; I \leftarrow I + 1)
\{ X = 2^N \}

we get three verification conditions:

\begin{align*}
C_1 & \equiv (X = 2^I \land 0 \leq I \leq N) \land I \geq N \Rightarrow X = 2^N \\
C_2 & \equiv (X = 2^I \land 0 \leq I \leq N) \land I < N \Rightarrow 2X = 2^{I+1} \land 0 \leq I + 1 \leq N \\
& \text{(from \; (X = 2^I \land 0 \leq I \leq N)[I + 1/I, 2X/X])} \\
C_3 & \equiv N \geq 0 \Rightarrow 1 = 2^0 \land 0 \leq 0 \leq N \\
& \text{(from \; (X = 2^I \land 0 \leq I \leq N)[0/I, 1/X])}
\end{align*}

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

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

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 $\text{ while}$

$$
\forall t \in W : [P \land e \land u = t] \ s \ [P \land u \prec t] \\
[Q] \text{ while } e \text{ do } s \ [P \land \neg e]
$$

- $(W, \prec)$ 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 $(\mathbb{N}, \prec)$
- (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”
- 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 $[P] \ c \ [\text{true}]$
  ignoring the specification

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

indeed, we have:

$$\frac{\{P\} \ c \ \{Q\} \quad [P] \ c \ [\text{true}]}{[P] \ c \ [Q]}$$
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:

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

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

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

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

\[
\forall n:\ [l < N \land N - l = n] \models l \leftarrow l + 1; \ X \leftarrow 2X \ [N - l = n - 1]
\]

\[
l < N \implies N - l \geq 0
\]

\[
[\text{true}] \ [l \geq N]
\]
Weakest precondition

\[ \text{wp}(\text{prog}, \text{Prop} : \text{Prop}) \]

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

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

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

the \textit{invariant predicate} \( I \) is combined with a \textit{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:**

\[
\begin{align*}
\{X = x\} & \quad Y \leftarrow X \quad \{Y = x\} \\
\{Y = x\} & \quad X \leftarrow ? \quad \{Y = x\} \quad \{Y = x\} \quad X \leftarrow Y \quad \{X = x\}
\end{align*}
\]

\[
\{X = x\} \quad Y \leftarrow X; X \leftarrow ?; X \leftarrow Y \quad \{X = x\}
\]
Non-determinism in predicate calculus

Predicate transformers:

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

- \( \text{slp}(P, X \leftarrow ?) \equiv \exists X : P \)
  
  (if \( P \) held for one value of \( X \), \( P \) holds for all values of \( X \) 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:

\[
\text{post}[\tau](P) \equiv \{ \sigma' | \exists \sigma \in P: (\sigma, \sigma') \in \tau \} \\
\text{pre}[\tau](P) \equiv \{ \sigma | \forall \sigma' \in \Sigma: (\sigma, \sigma') \in \tau \implies \sigma' \in P \}
\]

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

\[
\begin{align*}
\{P[A\{e \mapsto f\}/A]\} & A(e) \leftarrow f \{P\}
\end{align*}
\]

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

- add logical axioms to reason about our arrays in assertions

\[
\begin{align*}
A\{e \mapsto f\}(e) &= f \\
(e \neq e') \Rightarrow (A\{e \mapsto f\}(e') &= A(e'))
\end{align*}
\]
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

\[
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
\]

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: \( \text{stat} \parallel \text{stat} \)

semantics: \( s_1 \parallel s_2 \)

- 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 \cite{Owicki76}

first idea:

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

but this is unsound
Concurrent programs: rule soundness

**Issue:**

\[
\frac{\{P_1\} s_1 \{Q_1\} \quad \{P_2\} s_2 \{Q_2\}}{\{P_1 \land P_2\} s_1 \parallel s_2 \{Q_1 \land Q_2\}} \text{ is not always sound}
\]

equation:

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

\[
\frac{\{X = Y = 0\} s_1 \{X = 1 \land Y = 0\} \quad \{X = Y = 0\} s_2 \{X = 0 \land Y = 1\}}{\{X = Y = 0\} s_1 \parallel s_2 \{\text{false}\}}
\]

**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:

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

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

**Issue:** incompleteness

\[\{X = 0\} X \leftarrow X + 1 \ || \ X \leftarrow X + 1 \{X = 2\}\] is valid but no proof of it can be derived

**Solution:** auxiliary variables introduce explicitly program points and program counters

Example:

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

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

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

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

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

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

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


