Axiomatic semantics
Semantics and Application to Program Verification

Antoine Miné

École normale supérieure, Paris
year 2014–2015

Course 7
25 March 2015
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)

Axiomatiec 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 transformers
  (weakest precondition, strongest postcondition)
- Verification conditions
  (partially automated program verification)
- Advanced topics
  - Total correctness (termination)
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
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;
    }
    // \exists Q: A = QB + R and 0 ≤ R < B
    return R;
}
```

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

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

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++;
    }
}
```

Course 7
Axiomatic semantics
Antoine Miné
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])
Floyd–Hoare logic

Language

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

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

we assume that:

- expressions are deterministic (for now)
- expression evaluation does not cause error

for instance, to avoid division by zero, we can:
- either define \( 1/0 \) to be a valid value, such as 0
- or systematically guard divisions
  (e.g.: \( \text{if } X = 0 \text{ then fail else } \cdots /X \cdots \)
Hoare rules: axioms

**Axioms:**

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

- any property true before **skip** is true afterwards
- any property is true after **fail**
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 free occurrences of \( X \) are replaced with \( e \)
- \( e \) must be deterministic
- the rule is “backwards” \((P \) appears as a postcondition\)

examples:

- \( \{ \text{true} \} \ X \leftarrow 5 \ \{ X = 5 \} \)
- \( \{ Y = 5 \} \ X \leftarrow Y \ \{ X = 5 \} \)
- \( \{ X + 1 \geq 0 \} \ X \leftarrow X + 1 \ \{ X \geq 0 \} \)
- \( \{ \text{false} \} \ X \leftarrow Y + 3 \ \{ Y = 0 \land X = 12 \} \)
- \( \{ Y \in [0, 10] \} \ X \leftarrow Y + 3 \ \{ X = Y + 3 \land Y \in [0, 10] \} \)
Hoare rules: consequence

**Rule of consequence:**

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

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

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 \textbf{fail} can be replaced with \( \{\text{true}\} \textbf{fail} \{\text{false}\} \) (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:

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

\[
\frac{\{ P \land \neg e \} \ t \ \{ Q \}}{\{ P \} \ \text{if} \ e \ \text{then} \ s \ \text{else} \ t \ \{ Q \}}
\]

to prove that \( Q \) holds after the test

we prove that it holds after each branch \((s, t)\)

under the assumption that it is executed \((e, \neg e)\)

example:

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

\[
\begin{align*}
\{ X > 0 \} & \ \text{skip} \ \{ X > 0 \} \\
\{ (X \neq 0) \land (X \geq 0) \} & \ \text{skip} \ \{ X > 0 \}
\end{align*}
\]
Floyd–Hoare logic

**Hoare rules: sequences**

Sequences:

\[
\begin{array}{c}
\{P\} \, s \, \{R\} \\
\{R\} \, t \, \{Q\} \\
\{P\} \, s; \, t \, \{Q\}
\end{array}
\]

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:

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

Hoare rules: loops

Loops:

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

\( P \) 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: \( ( I \land \neg e ) \Rightarrow Q \)

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

we can derive the rule:

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

\[
\begin{align*}
\{P\} \text{skip} \{P\} & & \{\text{true}\} \text{fail} \{\text{false}\} & & \{P[e/X]\} \ X \leftarrow e \ \{P\} \\
\{P\} s \{R\} & & \{R\} t \{Q\} & & \{P \land e\} s \{Q\} & & \{P \land \neg e\} t \{Q\} \\
\{P\} s; t \{Q\} & & \{P \land e\} s \{P\} & & \{P\} \text{if } e \text{ then } s \text{ else } t \{Q\} \\
\{P\} \text{while } e \text{ do } s \{P \land \neg e\} & & P \Rightarrow P' & & Q' \Rightarrow Q & & \{P'\} c \{Q'\} & & \{P\} c \{Q\}
\end{align*}
\]
Proof tree example

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

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

\[
A \quad B \quad \{P_1\} \ s \quad \{P_1 \land I \geq N\} \quad \{X = 1 \land I = 0 \land N \geq 0\} \ s \quad \{X = 2^N \land N = I \land N \geq 0\}
\]

\[
P_1 \overset{\text{def}}{=} X = 2^I \land I \leq N \land N \geq 0
\]
\[
P_2 \overset{\text{def}}{=} X = 2^{I+1} \land I+1 \leq N \land N \geq 0
\]
\[
P_3 \overset{\text{def}}{=} 2X = 2^{I+1} \land I+1 \leq N \land N \geq 0 \quad \equiv X = 2^I \land I < N \land N \geq 0
\]

\[
A : (X = 1 \land I = 0 \land N \geq 0) \Rightarrow P_1
\]
\[
B : (P_1 \land I \geq N) \Rightarrow (X = 2^N \land N = I \land N \geq 0)
\]
\[
C : P_3 \iff (P_1 \land I < N)
\]
Proof tree example

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

\[
\begin{align*}
\{ \text{true} \} &\quad l \leftarrow l - 1 \quad \{ \text{true} \} \\
\{ l \neq 0 \} &\quad l \leftarrow l - 1 \quad \{ \text{true} \} \\
\{ \text{true} \} &\quad \textbf{while } l \neq 0 \textbf{ do } l \leftarrow l - 1 \quad \{ \text{true} \land \neg (l \neq 0) \} \\
\{ \text{true} \} &\quad \textbf{while } l \neq 0 \textbf{ do } l \leftarrow l - 1 \quad \{ l = 0 \}
\end{align*}
\]

- in some cases, the program does not terminate
  (if the program starts with \( l < 0 \))
- the same proof holds for: \( \{ \text{true} \} \textbf{ while } l \neq 0 \textbf{ do } J \leftarrow J - 1 \quad \{ l = 0 \} \)
- anything can be proven of a program that never terminates:

\[
\begin{align*}
\{ l = 1 \land l \neq 0 \} &\quad J \leftarrow J - 1 \quad \{ l = 1 \} \\
\{ l = 1 \} &\quad \textbf{while } l \neq 0 \textbf{ do } J \leftarrow J - 1 \quad \{ l = 1 \land l = 0 \} \\
\{ l = 1 \} &\quad \textbf{while } l \neq 0 \textbf{ do } J \leftarrow J - 1 \quad \{ \text{false} \}
\end{align*}
\]
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}\n
we need to find an invariant assertion \(P\) for the \textbf{while} rule

\textbf{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 (\( P' \) 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}
  (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
Soundness and completeness

Validity:
\[
\{ P \} \ c \ \{ Q \} \ \text{is valid} \quad \overset{\text{def}}{\iff} \quad \text{executions starting in a state satisfying } P \\
\text{and terminating} \quad \text{end in a state satisfying } Q
\]

(it is an operational notion)

- **soundness**
  
  a proof tree exists for \( \{ P \} \ c \ \{ Q \} \implies \{ P \} \ c \ \{ Q \} \) is valid

- **completeness**
  
  \( \{ P \} \ c \ \{ Q \} \) is valid \implies a proof tree exists for \( \{ P \} \ c \ \{ Q \} \)

(technically, by Gödel's incompleteness theorem, \( P \implies Q \) is not always provable for strong theories; hence, Hoare logic is incomplete; we consider relative completeness by adding all valid properties \( P \implies Q \) on assertions as axioms)

**Theorem (Cook 1974)**

Hoare logic is sound (and relatively complete)

Completeness no longer holds for more complex languages (Clarke 1976)
Link with denotational semantics

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

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

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

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

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

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

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

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

---

**Theorem**

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

\( (A \models P \text{ means } \forall \rho \in A, \text{ the formula } P \text{ is true on the variable assignment } \rho) \)
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}\} \text{ is always valid}\]
- \([^\{[R]\}\ c\ {\{[R']\}} \Rightarrow S[\ c\ ]\ R \subseteq R'\]

\[\iff\]

\([S[\ c\ ]\ R]\) provides the best valid postcondition
**Loop invariants**

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

- **Denotational semantics:**
  
  we must find \(\text{lfp } F\) where \(F(X) \overset{\text{def}}{=} R \cup S[s] \{ \rho \in X | \rho \models e \}\)

  - \(\text{lfp } F = \cap \{ X | F(X) \subseteq X \}\) (Tarski’s theorem)
  
  - \(F(X) \subseteq X \iff ([R] \Rightarrow [X]) \land \{[X \land e]\} \ s \ \{[X]\}\)

  \(R \subseteq X\) means \([R] \Rightarrow [X]\),
  
  \(S[s] \{ \rho \in X | \rho \models e \}\) \subseteq X means \(\{[X \land e]\} \ s \ \{[X]\}\)

  As a consequence:
  
  - any \(X\) such that \(F(X) \subseteq X\) gives an inductive invariant \([X]\)
  
  - \(\text{lfp } F\) gives the best inductive invariant
  
  - any \(X\) such that \(\text{lfp } F \subseteq X\) gives an invariant
    
    (not necessarily inductive)

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

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

(introduced in [Dijkstra75])
**Predicate transformers**

**Dijkstra’s weakest liberal preconditions**

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

(introduced in [Dijkstra75])
A calculus for \( wlp \)

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

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

- \( e \Rightarrow Q \) 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 \( I \)
  - intuitively, \( wlp \) checks that \( I \) is an inductive invariant implying \( P \)
  - if so, it returns \( I \); otherwise, it returns false

\( wlp \) is the weakest precondition only if \( I \) is well-chosen...
Predicate transformers

Alternate form for loops

**Unrolling** of the loop \( \text{while } e \text{ do } s \):

- \( L_0 \overset{\text{def}}{=} \text{fail} \)
- \( L_{i+1} \overset{\text{def}}{=} \text{if } e \text{ then } (s; L_i) \text{ else skip} \)
- \( L_i \) runs the loop and fails after \( i \) iterations

we have:

\[
\begin{aligned}
\text{wlp}(L_0, P) &= \text{true} \\
\text{wlp}(L_{i+1}, P) &= (e \Rightarrow \text{wlp}(s, \text{wlp}(L_i, P))) \land (\neg e \Rightarrow P)
\end{aligned}
\]

**Alternate \text{wlp} for loops:** \( \text{wlp(while } e \text{ do } s, P) \overset{\text{def}}{=} \forall i: X_i \)

where

\[
\begin{aligned}
X_0 &\overset{\text{def}}{=} \text{true} \\
X_{i+1} &\overset{\text{def}}{=} (e \Rightarrow \text{wlp}(s, X_i)) \land (\neg e \Rightarrow P)
\end{aligned}
\]

\( X_i \leftarrow X_{i+1} \): sequence of assertions of increasing strength

(\( \forall i: X_i \)) is the limit, with an arbitrary number of iterations

(\( \forall i: X_i \)) is a closed form guaranteed to be the weakest precondition

(no need for a user-specified invariant)

(\( \forall i: X_i \)) is the fixpoint of a second-order formula

\( \Rightarrow \) very difficult to handle
\[ \text{wlp(}\text{if } X < 0 \text{ then } Y \leftarrow -X \text{ else } Y \leftarrow X, \ Y \geq 10) = \]
\[ (X < 0 \Rightarrow \text{wlp}(Y \leftarrow -X, \ Y \geq 10)) \land (X \geq 0 \Rightarrow \text{wlp}(Y \leftarrow X, \ Y \geq 10)) \]
\[ (X < 0 \Rightarrow -X \geq 10) \land (X \geq 0 \Rightarrow X \geq 10) = \]
\[ (X \geq 0 \lor -X \geq 10) \land (X < 0 \lor X \geq 10) = \]
\[ X \geq 10 \lor X \leq -10 \]

\[ \text{wlp generates complex formulas} \]
\[ \text{it is important to simplify them from time to time} \]
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)
we can define \( slp : (\text{Prop} \times \text{prog}) \to \text{Prop} \)

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

- \( \{P\} \ c \ \{Q\} \iff (slp(P, c) \Rightarrow Q) \)  
  (strongest postcondition)

- \( slp(P, c) \) 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

\[
\begin{align*}
\text{slp}(P, \text{skip}) & \overset{\text{def}}{=} P \\
\text{slp}(P, \text{fail}) & \overset{\text{def}}{=} \text{false} \\
\text{slp}(P, X \leftarrow e) & \overset{\text{def}}{=} \exists v : P[v/X] \land X = e[v/X] \\
\text{slp}(P, s; t) & \overset{\text{def}}{=} \text{slp}(\text{slp}(P, s), t) \\
\text{slp}(P, \text{if } e \text{ then } s \text{ else } t) & \overset{\text{def}}{=} \text{slp}(P \land e, s) \lor \text{slp}(P \land \neg e, t) \\
\text{slp}(P, \text{while } e \text{ do } s) & \overset{\text{def}}{=} (P \Rightarrow I) \land (\text{slp}(I \land e, s) \Rightarrow I) \land (\neg e \land I)
\end{align*}
\]

(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**: \textit{wlp}, \textit{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

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

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

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

by induction on the syntax of statements

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

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

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

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

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

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

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

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

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

- we use \text{wlp} to infer assertions automatically when possible
- \text{vcg}_s(c, P) = (P', C) propagates postconditions backwards (P into P') and accumulates into C verification conditions (from loops)
- we could do the same using \text{slp} instead of \text{wlp} (symbolic execution)
Consider the program:

\[
\begin{align*}
\{N \geq 0\} & \quad X \leftarrow 1; I \leftarrow 0; \\
& \textbf{while} \{X = 2^I \land 0 \leq I \leq N\} \quad I < N \textbf{ do} \\
& \quad (X \leftarrow 2X; I \leftarrow I + 1) \\
\{X = 2^N\}
\end{align*}
\]

we get three verification conditions:

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

which can be checked independently.
Verification conditions

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 operators 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)
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


Extensions
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 **while**

\[
\forall t \in W: [P \land e \land u = t] s [P \land u \prec t] \quad \text{([}W, \prec\text{) is well-founded)}
\]

- \((W, \prec)\) 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”
    - often \(\neg e \implies u = 0\): \(u\) counts the number of steps until termination
Total correctness

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 \( ((W, \prec) \stackrel{\text{def}}{=} (\mathbb{N}, \leq)) \) using an integer expression \( r \) over program variables:

\[
\forall n: [P \land e \land (r = n)] \quad s [P \land (r < n)] \quad (P \land e) \Rightarrow (r \geq 0) \\
[P] \quad \text{while } e \text{ do } s [P \land \neg e]
\]

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

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

\[
\forall n: [I < N \land N - I = n] \quad I \leftarrow I + 1; \ X \leftarrow 2X \quad [N - I = n - 1] \\
I < N \Rightarrow N - I \geq 0 \\
[\text{true}] \quad p \quad [I \geq N]
\]
Weakest precondition

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

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

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

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

the invariant predicate \( I \) is combined with a 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)