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)

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 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
//@ 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
Specifications

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 \( \exists q: a = qb + r \land 0 \leq r < b \))
Specifications

Example: ghost variables

```plaintext
example with ghost variables

//@ 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
Language support

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

- 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 prog is executed
- then \( Q \) holds after the execution of prog
- unless 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

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

- \(X \in V\): integer-valued variables
- \(\text{expr}\): 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.: \(\text{if } X = 0 \text{ then fail else } \cdots /X \cdots\))
Floyd–Hoare logic

Hoare rules: axioms

**Axioms:**

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

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

- 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” \((P \text{ appears as a postcondition})\)

**Examples:**

- $\{\text{true}\} \ X \leftarrow 5 \ \{X = 5\}$
- $\{Y = 5\} \ X \leftarrow Y \ \{X = 5\}$
- $\{X + 1 \geq 0\} \ X \leftarrow X + 1 \ \{X \geq 0\}$
- $\{\text{false}\} \ X \leftarrow Y + 3 \ \{Y = 0 \land X = 12\}$
- $\{Y \in [0, 10]\} \ X \leftarrow Y + 3 \ \{X = Y + 3 \land Y \in [0, 10]\}$
Floyd–Hoare logic

Hoare rules: consequence

Rule of consequence:

\[
\begin{align*}
P & \Rightarrow P' \\
Q' & \Rightarrow Q \\
\{P', c\} & \Rightarrow \{Q'\} \\
\{P\} & \Rightarrow \{Q\}
\end{align*}
\]

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:**

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

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:

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

Hoare rules: sequences

Sequences:

\[
\begin{align*}
\{P\} s \{R\} & \quad \{R\} t \{Q\} \\
\{P\} s; t \{Q\} & 
\end{align*}
\]

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:

\[
\begin{align*}
\{X = 1 \land Y = 1\} & \quad X \leftarrow X + 1 \quad \{X = 2 \land Y = 1\} \quad Y \leftarrow Y - 1 \quad \{X = 2 \land Y = 0\}
\end{align*}
\]
Loops: 

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

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

P is a loop invariant

P holds before each loop iteration, before even testing e

Practical use:

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

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:

\[ \{ P \} \ while \ e \ do \ s \ \{ Q \} \]
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 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

\[
\begin{aligned}
\{P\} \text{skip} \{P\} & & \{\text{true}\} \text{fail} \{\text{false}\} & & \{P[e/X]\} X \leftarrow e \{P\} \\
\{P\} s \{R\} & & \{R\} t \{Q\} & & \{P \land e\} s \{Q\} & & \{P \land \neg e\} t \{Q\} \\
\{P\} s; t \{Q\} & & \{P \land e\} s \{P\} & & \{P\} \text{while } e \text{ do } s \{P \land \neg e\} \\
\end{aligned}
\]
Floyd–Hoare logic

Proof tree example

\[ S \overset{\text{def}}{=} \text{while } l < N \text{ do } (X \gets 2X; \ l \gets l + 1) \]

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

\[
\begin{array}{c}
\text{A} \quad \text{B} \quad \{P_1\} \quad \text{s} \quad \{P_1 \land l \geq N\}
\end{array}
\]

\[
\{X = 1 \land l = 0 \land N \geq 0\} \quad \text{s} \quad \{X = 2^N \land N = l \land N \geq 0\}
\]

\[
\begin{align*}
P_1 & \overset{\text{def}}{=} X = 2^l \land l \leq N \land N \geq 0 \\
P_2 & \overset{\text{def}}{=} X = 2^{l+1} \land l+1 \leq N \land N \geq 0 \\
P_3 & \overset{\text{def}}{=} 2X = 2^{l+1} \land l+1 \leq N \land N \geq 0 \quad \equiv X = 2^l \land l < N \land N \geq 0
\end{align*}
\]

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

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

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

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

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

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

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

\[
\{I = 1\} \quad \text{while } I \neq 0 \text{ do } J \leftarrow J - 1 \quad \{\text{false}\}
\]
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 \( (P \text{ is an invariant}) \)
- but \( \{ P \land (X < 10) \} \ X \leftarrow X + 1; \ Y \leftarrow Y + 1 \ \{ P \} \)

\( P \land X < 10 \) 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 ← X + 1; Y ← Y + 1) \{X = Y = 10\}

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

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

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

\(\implies\)

to prove a loop invariant
it is often necessary to find a \textit{stronger} inductive loop invariant
Soundness and completeness

Validity:

{P} c {Q} is valid $\iff$ executions starting in a state satisfying P and terminating 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, $P \Rightarrow Q$ is not always provable for strong theories; hence, Hoare logic is incomplete; we consider relative completeness by adding all valid properties $P \Rightarrow Q$ on assertions as axioms)

**Theorem (Cook 1974)**

Hoare logic is sound (and relatively complete)

Completeness no longer holds for more complex languages (Clarke 1976)
Floyd–Hoare logic

Link with denotational semantics

Reminder: $S[\text{stat}] : \mathcal{P}(\mathcal{E}) \rightarrow \mathcal{P}(\mathcal{E})$ where $\mathcal{E} \overset{\text{def}}{=} \mathcal{V} \mapsto \bot$

- $S[\text{skip}] R \overset{\text{def}}{=} R$
- $S[\text{fail}] R \overset{\text{def}}{=} \emptyset$
- $S[\text{s}_1; \text{s}_2] \overset{\text{def}}{=} S[\text{s}_2] \circ S[\text{s}_1]$
- $S[\text{X} \leftarrow \text{e}] R \overset{\text{def}}{=} \{ \rho[X \mapsto v] | \rho \in R, v \in E[\text{e}] \rho \}$
- $S[\text{if } \text{e} \text{ then } \text{s}_1 \text{ else } \text{s}_2] R \overset{\text{def}}{=} S[\text{s}_1] \{ \rho \in R | \text{true} \in E[\text{e}] \rho \} \cup S[\text{s}_2] \{ \rho \in R | \text{false} \in E[\text{e}] \rho \}$
- $S[\text{while } \text{e} \text{ do } \text{s}] R \overset{\text{def}}{=} \{ \rho \in \text{lfp } F | \text{false} \in E[\text{e}] \rho \}$
  where $F(X) \overset{\text{def}}{=} R \cup S[\text{s}] \{ \rho \in X | \text{true} \in E[\text{e}] \rho \}$

Theorem

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

($A \models P$ means $\forall \rho \in A$, the formula $P$ 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
**Loop invariants**

**Hoare:**

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

**Denotational semantics:**

we must find lfp \( F \) where \( F(X) \overset{\text{def}}{=} R \cup S[e] \{ \rho \in X \mid \rho \models e \} \)

\( \text{lfp } F = \cap \{ X \mid F(X) \subseteq X \} \) \hspace{1cm} \text{(Tarski’s theorem)}

\( F(X) \subseteq X \iff ([R] \Rightarrow [X]) \land \{ [X \land e] \} s \{ [X] \} \)

\( R \subseteq X \) means \([R] \Rightarrow [X],\)

\( S[e] \{ \rho \in X \mid \rho \models e \} \subseteq X \) means \( \{ [X \land e] \} s \{ [X] \} \)

As a consequence:

- any \( X \) such that \( F(X) \subseteq X \) gives an inductive invariant \([X]\)
- lfp \( F \) gives the best inductive invariant
- any \( X \) such that lfp \( F \subseteq X \) gives an invariant
  (not necessarily inductive)

(see [Cousot02])
Predicate transformers
**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** \( wlp : (\text{prog} \times \text{Prop}) \rightarrow \text{Prop} \)

\( wlp(c, P) \) 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:

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

(introduced in [Dijkstra75])
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** $\text{wlp} : (\text{prog} \times \text{Prop}) \rightarrow \text{Prop}$

$\text{wlp}(c, P)$ 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:**

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

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

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

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

**Unrolling** of the loop `while e 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: \( \text{wlp}(\text{while } e \text{ do } s, P) \overset{\text{def}}{=} \forall i : X_i \)

where \( X_0 \overset{\text{def}}{=} \text{true} \)

\[
\begin{aligned}
X_i &\overset{\text{def}}{=} (e \Rightarrow \text{wlp}(s, X_i)) \land (\neg e \Rightarrow P) \\
X_i &\Leftarrow X_{i+1}: \text{sequence of assertions of increasing strength}
\end{aligned}
\]

\( (\forall i : X_i) \) 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
\( \implies \) very difficult to handle
**Wlp** computation example

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

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

- \( \text{wlp}(c, \text{false}) \equiv \text{false} \) \hspace{1cm} \text{(excluded miracle)}
- \( \text{wlp}(c, P) \land \text{wlp}(d, Q) \equiv \text{wlp}(c, P \land Q) \) \hspace{1cm} \text{(distributivity)}
- \( \text{wlp}(c, P) \lor \text{wlp}(d, Q) \equiv \text{wlp}(c, P \lor Q) \) \hspace{1cm} \text{(distributivity)}
  \hspace{1cm} \text{(⇒ always true, ⇐ only true for deterministic, error-free programs)}
- \text{if } P \Rightarrow Q, \text{ then } \text{wlp}(c, P) \Rightarrow \text{wlp}(c, Q) \hspace{1cm} \text{(monotonicity)}

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

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

\( (\text{stronger than syntactic equality}) \)
we can define \( \text{slp} : (\text{Prop} \times \text{prog}) \rightarrow \text{Prop} \)

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

- \( \{P\} \ c \ \{Q\} \iff (\text{slp}(P, c) \Rightarrow Q) \)  
  (strongest postcondition)
  (corresponds to the smallest state set)

- \( \text{slp}(P, c) \) does not care about non-termination  
  (liberal)

- allows forward reasoning

we have a duality:

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

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

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

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

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

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

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

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

(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: \(\text{wlp}, \text{slp}\) 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])
Verification conditions

Language

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

\[
\text{prog} ::= \{\text{Prop}\} \text{stat} \{\text{Prop}\}
\]

- 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:

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

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

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

\[
\begin{align*}
\text{vcg}_s (\text{skip}, Q) &\overset{\text{def}}{=} (Q, \emptyset) \\
\text{vcg}_s (X \gets e, Q) &\overset{\text{def}}{=} (Q[e/X], \emptyset) \\
\text{vcg}_s (s; t, Q) &\overset{\text{def}}{=} \\
&\begin{align*}
\text{let } (R, C) &= \text{vcg}_s (t, Q) \text{ in } \begin{align*}
\text{let } (P, D) &= \text{vcg}_s (s, R) \text{ in } (P, C \cup D) \\
\end{align*} \\
\end{align*} \\
\text{vcg}_s (\text{if } e \text{ then } s \text{ else } t, Q) &\overset{\text{def}}{=} \\
&\begin{align*}
\text{let } (S, C) &= \text{vcg}_s (s, Q) \text{ in } \begin{align*}
\text{let } (T, D) &= \text{vcg}_s (t, Q) \text{ in } ((e \Rightarrow S) \land (\neg e \Rightarrow T), C \cup D) \\
\end{align*} \\
\end{align*} \\
\text{vcg}_s (\text{while } \{ I \} e \text{ do } s, Q) &\overset{\text{def}}{=} \\
&\begin{align*}
\text{let } (R, C) &= \text{vcg}_s (s, I) \text{ in } (I, C \cup \{(I \land e) \Rightarrow R, (I \land \neg e) \Rightarrow Q\}) \\
\end{align*} \\
\text{vcg}_s (\text{assert } e, Q) &\overset{\text{def}}{=} (e \Rightarrow Q, \emptyset)
\end{align*}
\]

- We use \( wlp \) 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 \( slp \) instead of \( wlp \) (symbolic execution).
Consider the program:
\[
\{N \geq 0\} \quad X \leftarrow 1; I \leftarrow 0;
\text{while} \ \{X = 2^I \land 0 \leq I \leq N\} \ I < N \text{ do}
\quad (X \leftarrow 2X; \ I \leftarrow I + 1)
\{X = 2^N\}
\]

we get three verification conditions:
\[
C_1 \overset{\text{def}}{=} (X = 2^I \land 0 \leq I \leq N) \land I \geq N \Rightarrow X = 2^N
\]
\[
C_2 \overset{\text{def}}{=} (X = 2^I \land 0 \leq I \leq N) \land I < N \Rightarrow 2X = 2^{I+1} \land 0 \leq I + 1 \leq N
\]
\[
\text{(from } (X = 2^I \land 0 \leq I \leq N)[I + 1/I, 2X/X])
\]
\[
C_3 \overset{\text{def}}{=} N \geq 0 \Rightarrow 1 = 2^0 \land 0 \leq 0 \leq N
\]
\[
\text{(from } (X = 2^I \land 0 \leq I \leq N)[0/I, 1/X])
\]

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 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


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 \(\text{while}\)

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

\((W, \prec)\) 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”
  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 $\{P\} \mathbf{c} \{Q\}$
  ignoring termination

- a proof of termination $[P] \mathbf{c} [\text{true}]$
  ignoring the specification
  (we must still include the precondition $P$
  as the program may not terminate for all inputs)

indeed, we have: $\frac{\{P\} \mathbf{c} \{Q\} \quad [P] \mathbf{c} [\text{true}]}{[P] \mathbf{c} [Q]}$
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:

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

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

Example:

\[
p \overset{\text{def}}{=} \text{while } l < N \text{ do } l \leftarrow l + 1; \ X \leftarrow 2X \text{ done}
\]

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

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

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

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

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

- 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:

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

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)