Axiomatic semantics

Semantics and Application to Program Verification

Antoine Miné

École normale supérieure, Paris
year 2013–2014

Course 4
12 March 2014

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)

Aximoatic 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)

Overview

- Specifications (informal examples)
- Floyd–Hoare logic
- Dijkstra's predicate transformers
  (weakest precondition, strongest postcondition)
- Verification conditions
  (partially automated program verification)
- Advanced topics
  - Auxiliary variables
  - Non-determinism
  - Total correctness (termination)
  - Arrays

Specifications

Example: function specification

example in C + ACSL

```
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

example in C + ACSL

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

- express the intended behavior of the function (returned value)

Example: function specification

example in C + ACSL

```
//@ 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

example in C + ACSL

```
//@ 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

example with full assertions

```
//@ 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 $\exists q: a = qb + r \wedge 0 \leq r < b$)

Example: ghost variables

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;
}
```

Program annotations can be more complex than the program

Example: ghost variables

example with ghost variables

```
//@ 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;
}
```

Program annotations can be more complex than the program and require reasoning on enriched states (ghost variables)

Example: class invariants

example in ESC/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: $\{P\}\ prog\ \{Q\}$

- *prog* is a program fragment
- P and Q are logical assertions over program variables
  (e.g. $P \stackrel{\text{def}}{=} (X \geq 0 \wedge Y \geq 0) \vee (X < 0 \wedge Y < 0)$)

A triple means:

- if P holds before *prog* is executed
- then Q holds after the execution of *prog*
- unless *prog* does not terminates or encounters an error

P is the precondition, Q is the postcondition

$\{P\}\ prog\ \{Q\}$ expresses partial correctness
(does not rule out errors and non-termination)

Hoare triples serve as judgements in a proof system
(introduced in [Hoare69])

Language

$$
\begin{array}{lcll}
\textit{stat} & ::= & X \leftarrow \textit{expr} & (\textit{assignment}) \\
 & | & \textbf{skip} & (\textit{do nothing}) \\
 & | & \textbf{fail} & (\textit{error}) \\
 & | & \textit{stat};\ \textit{stat} & (\textit{sequence}) \\
 & | & \textbf{if}\ \textit{expr}\ \textbf{then}\ \textit{stat}\ \textbf{else}\ \textit{stat} & (\textit{conditional}) \\
 & | & \textbf{while}\ \textit{expr}\ \textbf{do}\ \textit{stat} & (\textit{loop})
\end{array}
$$

- $X \in \mathbb{V}$: integer-valued variables
- *expr*: integer arithmetic expressions

  we assume that:

  - expressions are deterministic (for now)
  - expression evaluation do 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.: **if** $X = 0$ **then fail else** $\cdots/X\cdots$)

Hoare rules: axioms

Axioms:

$$\frac{}{\{P\}\ \textbf{skip}\ \{P\}} \qquad\qquad \frac{}{\{P\}\ \textbf{fail}\ \{Q\}}$$

- any property true before **skip** is true afterwards
- any property is true after **fail**

Hoare rules: axioms

Assignment axiom:

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

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:

$\{\text{true}\}\ X \leftarrow 5\ \{X = 5\}$
$\{Y = 5\}\ X \leftarrow Y\ \{X = 5\}$
$\{X + 1 \geq 0\}\ X \leftarrow X + 1\ \{X \geq 0\}$
$\{\text{false}\}\ X \leftarrow Y + 3\ \{Y = 0 \wedge X = 12\}$
$\{Y \in [0, 10]\}\ X \leftarrow Y + 3\ \{X = Y + 3 \wedge Y \in [0, 10]\}$

Hoare rules: consequence

Rule of consequence:

$$\frac{P \Rightarrow P' \qquad Q' \Rightarrow Q \qquad \{P'\}\ c\ \{Q'\}}{\{P\}\ c\ \{Q\}}$$

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 $\dfrac{}{\{\text{true}\}\ \textbf{fail}\ \{\text{false}\}}$

  (as $P \Rightarrow$ true and false $\Rightarrow Q$ always hold)
- $\{X = 99 \wedge Y \in [1, 10]\}\ X \leftarrow Y + 10\ \{X = Y + 10 \wedge Y \in [1, 10]\}$

  (as $\{Y \in [1, 10]\}\ X \leftarrow Y + 10\ \{X = Y + 10 \wedge Y \in [1, 10]\}$ and $X = 99 \wedge Y \in [1, 10] \Rightarrow Y \in [1, 10]$)

Hoare rules: tests

Tests:

$$\frac{\{P \wedge e\}\ s\ \{Q\} \qquad \{P \wedge \neg e\}\ t\ \{Q\}}{\{P\}\ \textbf{if}\ e\ \textbf{then}\ s\ \textbf{else}\ t\ \{Q\}}$$

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:

$$\dfrac{\dfrac{\overline{\{X<0\}\ X \leftarrow -X\ \{X>0\}}}{\{(X \neq 0) \wedge (X<0)\}\ X \leftarrow -X\ \{X>0\}} \qquad \dfrac{\overline{\{X>0\}\ \textbf{skip}\ \{X>0\}}}{\{(X \neq 0) \wedge (X \geq 0)\}\ \textbf{skip}\ \{X>0\}}}{\{X \neq 0\}\ \textbf{if}\ X<0\ \textbf{then}\ X \leftarrow -X\ \textbf{else skip}\ \{X>0\}}$$

Hoare rules: sequences

Sequences:
$$\frac{\{P\}\ s\ \{R\} \qquad \{R\}\ t\ \{Q\}}{\{P\}\ s;t\ \{Q\}}$$

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:

$$\frac{\{P \wedge e\}\ s\ \{P\}}{\{P\}\ \textbf{while}\ e\ \textbf{do}\ s\ \{P \wedge \neg e\}}$$

P is a loop invariant

P holds before each loop iteration, before even testing e

Practical use:

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

it is sufficient to invent an assertion I that:

holds when the loop start: $P \Rightarrow I$

is invariant by the body s: $\{I \wedge e\}\ s\ \{I\}$

implies the assertion when the loop stops: $(I \wedge \neg e) \Rightarrow Q$

we can derive the rule:

$$\frac{P \Rightarrow I \qquad I \wedge \neg e \Rightarrow Q \qquad \dfrac{\{I \wedge e\}\ s\ \{I\}}{\{I\}\ \textbf{while}\ e\ \textbf{do}\ s\ \{I \wedge \neg e\}}}{\{P\}\ \textbf{while}\ e\ \textbf{do}\ s\ \{Q\}}$$

Hoare rules: logical part

Hoare logic is parameterized by the choice of logical theory of assertions

the logical theory is used to:

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

Examples: (generally first order theories)

- booleans $(\mathbb{B}, \neg, \wedge, \vee)$
- bit-vectors $(\mathbb{B}^n, \neg, \wedge, \vee)$
- 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

$$\frac{}{\{P\}\ \textbf{skip}\ \{P\}} \qquad \frac{}{\{\text{true}\}\ \textbf{fail}\ \{\text{false}\}} \qquad \frac{}{\{P[e/X]\}\ X \leftarrow e\ \{P\}}$$

$$\frac{\{P\}\ s\ \{R\} \qquad \{R\}\ t\ \{Q\}}{\{P\}\ s;t\ \{Q\}} \qquad \frac{\{P \land e\}\ s\ \{Q\} \qquad \{P \land \neg e\}\ t\ \{Q\}}{\{P\}\ \textbf{if}\ e\ \textbf{then}\ s\ \textbf{else}\ t\ \{Q\}}$$

$$\frac{\{P \land e\}\ s\ \{P\}}{\{P\}\ \textbf{while}\ e\ \textbf{do}\ s\ \{P \land \neg e\}}$$

$$\frac{P \Rightarrow P' \qquad Q' \Rightarrow Q \qquad \{P'\}\ c\ \{Q'\}}{\{P\}\ c\ \{Q\}}$$

Proof tree example

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

$$\dfrac{A \qquad B \qquad \dfrac{\dfrac{C \qquad \dfrac{}{\{P_3\}\ X \leftarrow 2X\ \{P_2\}} \qquad \dfrac{}{\{P_2\}\ I \leftarrow I+1\ \{P_1\}}}{\{P_1 \wedge I < N\}\ X \leftarrow 2X;\ I \leftarrow I+1\ \{P_1\}}}{\{P_1\}\ s\ \{P_1 \wedge I \geq N\}}}{\{X = 1 \wedge I = 0 \wedge N \geq 0\}\ s\ \{X = 2^N \wedge N = I \wedge N \geq 0\}}$$

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

$A : (X = 1 \wedge I = 0 \wedge N \geq 0) \Rightarrow P_1$

$B : (P_1 \wedge I \geq N) \Rightarrow (X = 2^N \wedge N = I \wedge N \geq 0)$

$C : P_3 \iff (P_1 \wedge I < N)$

Proof tree example

$s \stackrel{\text{def}}{=} \textbf{while } I \neq 0 \textbf{ do } I \leftarrow I - 1$

$$\cfrac{\cfrac{\cfrac{}{\{\text{true}\}\ I \leftarrow I - 1\ \{\text{true}\}}}{\{I \neq 0\}\ I \leftarrow I - 1\ \{\text{true}\}}}{\cfrac{\{\text{true}\}\ \textbf{while } I \neq 0 \textbf{ do } I \leftarrow I - 1\ \{\text{true} \wedge \neg(I \neq 0)\}}{\{\text{true}\}\ \textbf{while } I \neq 0 \textbf{ do } I \leftarrow I - 1\ \{I = 0\}}}$$

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

$$\cfrac{\cfrac{\cfrac{}{\{I = 1 \wedge I \neq 0\}\ J \leftarrow J - 1\ \{I = 1\}}}{\{I = 1\}\ \textbf{while } I \neq 0 \textbf{ do } J \leftarrow J - 1\ \{I = 1 \wedge I = 0\}}}{\{I = 1\}\ \textbf{while } I \neq 0 \textbf{ do } J \leftarrow J - 1\ \{\text{false}\}}$$

Invariants and inductive invariants

Example: we wish to prove:

$\{X = Y = 0\}$ **while** $X < 10$ **do** $(X \leftarrow X + 1;\ Y \leftarrow Y + 1)$ $\{X = Y = 10\}$

we need to find an invariant assertion P for the **while** rule

Incorrect invariant: $P \stackrel{\text{def}}{=} X, Y \in [0, 10]$

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

  $P \wedge X < 10$ does not prevent $Y = 10$
  after $Y \leftarrow Y + 1$, P does not hold anymore

Invariants and inductive invariants

Example: we wish to prove:

$\{X = Y = 0\}$ **while** $X < 10$ **do** $(X \leftarrow X + 1;\; Y \leftarrow Y + 1)$ $\{X = Y = 10\}$

we need to find an invariant assertion P for the **while** rule

Correct invariant: $P' \stackrel{\text{def}}{=} X \in [0, 10] \wedge X = Y$

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

$\Longrightarrow$ to prove a loop invariant
it is often necessary to find a stronger loop invariant

Soundness and completeness

Validity:

$\{P\}\ c\ \{Q\}$ is valid $\stackrel{\text{def}}{\Longleftrightarrow}$ 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)

Link with denotational semantics

Reminder: $\mathsf{S}[\![\, \mathit{stat} \,]\!] : \mathcal{P}(\mathcal{E}) \to \mathcal{P}(\mathcal{E})$ where $\mathcal{E} \stackrel{\text{def}}{=} \mathbb{V} \mapsto \mathbb{I}$

$\mathsf{S}[\![\, \textbf{skip} \,]\!]\, R \stackrel{\text{def}}{=} R$

$\mathsf{S}[\![\, \textbf{fail} \,]\!]\, R \stackrel{\text{def}}{=} \emptyset$

$\mathsf{S}[\![\, s_1; s_2 \,]\!] \stackrel{\text{def}}{=} \mathsf{S}[\![\, s_2 \,]\!] \circ \mathsf{S}[\![\, s_1 \,]\!]$

$\mathsf{S}[\![\, X \leftarrow e \,]\!]\, R \stackrel{\text{def}}{=} \{\, \rho[X \mapsto v] \mid \rho \in R,\ v \in \mathsf{E}[\![\, e \,]\!]\, \rho \,\}$

$$\mathsf{S}[\![\, \textbf{if } e \textbf{ then } s_1 \textbf{ else } s_2 \,]\!]\, R \stackrel{\text{def}}{=} \begin{array}{l} \mathsf{S}[\![\, s_1 \,]\!]\, \{\, \rho \in R \mid \text{true} \in \mathsf{E}[\![\, e \,]\!]\, \rho \,\} \cup \\ \mathsf{S}[\![\, s_2 \,]\!]\, \{\, \rho \in R \mid \text{false} \in \mathsf{E}[\![\, e \,]\!]\, \rho \,\} \end{array}$$

$\mathsf{S}[\![\, \textbf{while } e \textbf{ do } s \,]\!]\, R \stackrel{\text{def}}{=} \{\, \rho \in \text{lfp } F \mid \text{false} \in \mathsf{E}[\![\, e \,]\!]\, \rho \,\}$
where $F(X) \stackrel{\text{def}}{=} R \cup \mathsf{S}[\![\, s \,]\!]\, \{\, \rho \in X \mid \text{true} \in \mathsf{E}[\![\, e \,]\!]\, \rho \,\}$

Theorem

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

($A \models P$ means $\forall \rho \in A$, the formula P is true on the variable assignment ρ)

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:

- $\{[R]\}\ c\ \{[\mathsf{S}[\![\, c \,]\!]\, R]\}$ is always valid
- $\{[R]\}\ c\ \{[R']\} \Rightarrow \mathsf{S}[\![\, c \,]\!]\, R \subseteq R'$

$\Longrightarrow [\mathsf{S}[\![\, c \,]\!]\, R]$ provides the best valid postcondition

Link with denotational semantics

Loop invariants

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

- **Denotational semantics:**
  we must find lfp F where $F(X) \stackrel{\text{def}}{=} R \cup \mathsf{S}[\![s]\!] \{ \rho \in X \mid \rho \models e \}$
  - lfp $F = \cap \{ X \mid F(X) \subseteq X \}$ (Tarski's theorem)
  - $F(X) \subseteq X \iff ([R] \Rightarrow [X]) \wedge \{[X \wedge e]\}\ s\ \{[X]\}$

    $R \subseteq X$ means $[R] \Rightarrow [X]$,
    $\mathsf{S}[\![s]\!] \{ \rho \in X \mid \rho \models e \} \subseteq X$ means $\{[X \wedge e]\}\ s\ \{[X]\}$

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

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 $\quad wlp : (prog \times Prop) \to Prop$

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

"liberal" means that we do not care about termination and errors

Examples:

$wlp(X \leftarrow X + 1, X = 1) =$

$wlp(\textbf{while } X < 0\ X \leftarrow X + 1, X \geq 0) =$

$wlp(\textbf{while } X \neq 0\ X \leftarrow X + 1, X \geq 0) =$

(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 $\quad wlp : (prog \times Prop) \to Prop$

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

"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:

$$wlp(\textbf{skip},\ P) \stackrel{\text{def}}{=} P$$

$$wlp(\textbf{fail},\ P) \stackrel{\text{def}}{=} \text{true}$$

$$wlp(X \leftarrow e,\ P) \stackrel{\text{def}}{=} P[e/X]$$

$$wlp(s; t,\ P) \stackrel{\text{def}}{=} wlp(s, wlp(t, P))$$

$$wlp(\textbf{if } e \textbf{ then } s \textbf{ else } t,\ P) \stackrel{\text{def}}{=} (e \Rightarrow wlp(s, P)) \land (\neg e \Rightarrow wlp(t, P))$$

$$wlp(\textbf{while } e \textbf{ do } s,\ P) \stackrel{\text{def}}{=} I \land ((e \land I) \Rightarrow wlp(s, I)) \land ((\neg e \land I) \Rightarrow P)$$

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

- **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...

Alternate form for loops

Unrolling of the loop **while** e **do** s:

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

we have: $\begin{cases} wlp(L_0, P) = \text{true} \\ wlp(L_{i+1}, P) = (e \Rightarrow wlp(s, wlp(L_i, P))) \wedge (\neg e \Rightarrow P) \end{cases}$

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

where $X_0 \stackrel{\text{def}}{=} \text{true}$

$X_{i+1} \stackrel{\text{def}}{=} (e \Rightarrow wlp(s, X_i)) \wedge (\neg e \Rightarrow P)$

$X_i \Leftarrow X_{i+1}$: sequence of assertions of increasing strength
$(\forall i \colon X_i)$ is the limit, with an arbitrary number of iterations

$(\forall i \colon X_i)$ is a closed form guaranteed to be the weakest precondition
(no need for a user-specified invariant)

$(\forall i \colon X_i)$ is the fixpoint of a second-order formula
$\Longrightarrow$ very difficult to handle

Wlp computation example

$$
\begin{aligned}
& wlp(\textbf{if } X < 0 \textbf{ then } Y \leftarrow -X \textbf{ else } Y \leftarrow X,\ Y \geq 10) = \\
& \quad (X < 0 \Rightarrow wlp(Y \leftarrow -X, Y \geq 10)) \wedge (X \geq 0 \Rightarrow wlp(Y \leftarrow X, Y \geq 10)) \\
& \quad (X < 0 \Rightarrow -X \geq 10) \wedge (X \geq 0 \Rightarrow X \geq 10) = \\
& \quad (X \geq 0 \vee -X \geq 10) \wedge (X < 0 \vee X \geq 10) = \\
& \quad X \geq 10 \vee X \leq -10
\end{aligned}
$$

wlp generates complex formulas

it is important to simplify them from time to time

Properties of wlp

- $wlp(c, \text{false}) \equiv \text{false}$ (excluded miracle)
- $wlp(c, P) \wedge wlp(d, Q) \equiv wlp(c, P \wedge Q)$ (distributivity)
- $wlp(c, P) \vee wlp(d, Q) \equiv wlp(c, P \vee Q)$ (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 that 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

$$slp(P, \textbf{skip}) \stackrel{\text{def}}{=} P$$

$$slp(P, \textbf{fail}) \stackrel{\text{def}}{=} \text{false}$$

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

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

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

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

(the rule for $X \leftarrow e$ makes *slp* much less attractive than *wlp*)

Verification conditions

Verification condition approch to program verification

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

$$\begin{array}{lcl} \mathit{stat} & ::= & X \leftarrow \mathit{expr} \\ & | & \textbf{skip} \\ & | & \mathit{stat};\ \mathit{stat} \\ & | & \textbf{if } \mathit{expr} \textbf{ then } \mathit{stat} \textbf{ else } \mathit{stat} \\ & | & \textbf{while } \{\mathit{Prop}\}\ \mathit{expr} \textbf{ do } \mathit{stat} \\ & | & \textbf{assert } \mathit{expr} \end{array}$$

$$\mathit{prog} \ ::= \ \{\mathit{Prop}\}\ \mathit{stat}\ \{\mathit{Prop}\}$$

- 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

$\underline{\mathrm{vcg}_p : prog \to \mathcal{P}(Prop)}$

$\mathrm{vcg}_p(\{P\}\, c\, \{Q\}) \stackrel{\text{def}}{=}$
$\quad \text{let } (R, C) = \mathrm{vcg}_s(c,\ Q) \text{ in } C \cup \{P \Rightarrow R\}$

$\underline{\mathrm{vcg}_s : (stat \times Prop) \to (Prop \times \mathcal{P}(Prop))}$

$\mathrm{vcg}_s(\textbf{skip},\ Q) \stackrel{\text{def}}{=} (Q, \emptyset)$

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

$\mathrm{vcg}_s(s; t,\ Q) \stackrel{\text{def}}{=}$
$\quad \text{let } (R, C) = \mathrm{vcg}_s(t,\ Q) \text{ in let } (P, D) = \mathrm{vcg}_s(s,\ R) \text{ in } (P, C \cup D)$

$\mathrm{vcg}_s(\textbf{if } e \textbf{ then } s \textbf{ else } t,\ Q) \stackrel{\text{def}}{=}$
$\quad \text{let } (S, C) = \mathrm{vcg}_s(s,\ Q) \text{ in let } (T, D) = \mathrm{vcg}_s(t,\ Q) \text{ in } ((e \Rightarrow S) \wedge (\neg e \Rightarrow T), C \cup D)$

$\mathrm{vcg}_s(\textbf{while } \{I\}\ e \textbf{ do } s,\ Q) \stackrel{\text{def}}{=}$
$\quad \text{let } (R, C) = \mathrm{vcg}_s(s,\ I) \text{ in } (I, C \cup \{(I \wedge e) \Rightarrow R, (I \wedge \neg e) \Rightarrow Q\})$

$\mathrm{vcg}_s(\textbf{assert } e,\ Q) \stackrel{\text{def}}{=} (e \Rightarrow Q, \emptyset)$

- we use *wlp* to infer assertions automatically when possible
- $\mathrm{vcg}_s(c, P) = (P', C)$ 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)

Verification condition generation example

Consider the program:

$$\{N \geq 0\} \qquad \begin{array}{l} X \leftarrow 1; I \leftarrow 0; \\ \textbf{while } \{X = 2^I \wedge 0 \leq I \leq N\} \; I < N \textbf{ do} \\ \qquad (X \leftarrow 2X; \; I \leftarrow I + 1) \end{array}$$

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

we get three verification conditions:

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

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

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

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

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

which can be checked independently

Extensions

Auxiliary variables

Auxiliary variables:

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

Applications:

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

Example: $\{X = x \wedge Y = y\}$ **if** $X < Y$ **then** $Y \leftarrow X$ **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
  "$\{\text{true}\}$ **if** $X < Y$ **then** $Y \leftarrow X$ **else skip** $\{Y = \min(X, Y)\}$" holds, but
  "$\{\text{true}\}$ $X \leftarrow Y + 1$ $\{Y = \min(X, Y)\}$" also holds

Non-determinism

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

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

Hoare axiom: $$\frac{}{\{\forall X : P\}\ X \leftarrow ?\ \{P\}}$$

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: $\frac{}{\{P\}\ X \leftarrow ?\ \{P\}}$

Example:

$$\frac{\{X = x\}\ Y \leftarrow X\ \{Y = x\} \qquad \dfrac{\{Y = x\}\ X \leftarrow ?\ \{Y = x\} \qquad \{Y = x\}\ X \leftarrow Y\ \{X = x\}}{}}{\{X = x\}\ Y \leftarrow X; X \leftarrow ?; X \leftarrow Y\ \{X = x\}}$$

Non-determinism

Predicate transformers:

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

- $slp(P,\ X \leftarrow ?) \stackrel{\text{def}}{=} \exists X\colon P$
  (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:
$$\mathrm{post}[\tau](P) \stackrel{\text{def}}{=} \{\, \sigma' \mid \exists \sigma \in P\colon (\sigma, \sigma') \in \tau \,\}$$
$$\widetilde{\mathrm{pre}}[\tau](P) \stackrel{\text{def}}{=} \{\, \sigma \mid \forall \sigma' \in \Sigma\colon (\sigma, \sigma') \in \tau \implies \sigma' \in P \,\}$$

then:
$$slp(P, c) = \mathrm{post}[c](P)$$
$$wlp(c, P) = \widetilde{\mathrm{pre}}[c](P)$$

Total correctness

Hoare triple: $[P]\ prog\ [Q]$

- if P holds before *prog* is executed
- then *prog* always terminates
- and Q holds after the execution of *prog*

Rules: we only need to change the rule for **while**

$$\frac{\forall t \in W\colon [P \wedge e \wedge u = t]\ s\ [P \wedge u \prec t]}{[P]\ \textbf{while}\ e\ \textbf{do}\ s\ [P \wedge \neg e]} \quad ((W,\prec) \text{ is well-founded})$$

- $(W,\prec)$ well-founded $\overset{\text{def}}{\iff}$ 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:

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

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:

$$\frac{\forall n \colon [P \wedge e \wedge (r = n)]\ s\ [P \wedge (r < n)] \qquad (P \wedge e) \Rightarrow (r \geq 0)}{[P]\ \textbf{while}\ e\ \textbf{do}\ s\ [P \wedge \neg e]}$$

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

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

$$\frac{\begin{array}{c}\forall n \colon [I < N \wedge N - I = n]\ I \leftarrow I + 1;\ X \leftarrow 2X\ [N - I = n - 1] \\ I < N \Rightarrow N - I \geq 0\end{array}}{[\text{true}]\ p\ [I \geq N]}$$

Weakest precondition

Weakest precondition $wp(prog, Prop) : Prop$

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

As before, only the definition for **while** needs to be modified:

$$\begin{aligned} wp(\textbf{while } e \textbf{ do } s,\ P) \stackrel{\text{def}}{=}\ & I\ \wedge \\ & (I \Rightarrow v \geq 0)\ \wedge \\ & \forall n\colon ((e \wedge I \wedge v = n) \Rightarrow wp(s, I \wedge v < n))\ \wedge \\ & ((\neg e \wedge I) \Rightarrow P) \end{aligned}$$

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)

Arrays

We enrich our language with:

- a set $\mathbb{A}$ of array variables
- array access in expressions: $A(expr)$, $A \in \mathbb{A}$
- array assignment: $A(expr) \leftarrow expr$, $A \in \mathbb{A}$

  (arrays have unbounded size here, we do not care about overflow)

Issue:

a natural idea is to generalize the assignment axiom:

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

but this is not sound, due to aliasing

example:

we would derive the invalid triple: $\{A(J) = 1 \wedge I = J\}\ A(I) \leftarrow 0\ \{A(J) = 1 \wedge I = J\}$
as $(A(J) = 1)[0/A(I)] = (A(J) = 1)$

Arrays

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

  (intuitively, we use "functional arrays" in the logic world)

- add logical axioms to reason about our arrays in assertions

$$\overline{A\{e \mapsto f\}(e) = f} \qquad \overline{(e \neq e') \Rightarrow (A\{e \mapsto f\}(e') = A(e'))}$$

Arrays: example

Example: swap

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

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

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

What about real languages?

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

Example: the assignment rule $\dfrac{}{\{P[e/X]\}\ X \leftarrow e\ \{P\}}$ 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: in the following courses

- automatic logic solvers to discharge proof obligations

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

Bibliography

[Apt81] **K. Apt**. *Ten Years of Hoare's logic: A survey* In ACM TOPLAS, 3(4):431–483, 1981.

[Cousot02] **P. Cousot**. *Constructive design of a hierarchy of semantics of a transition system by abstract interpretation.* In TCS, 277(1–2):47–103, 2002.

[Dijkstra76] **E.W. Dijkstra**. *Guarded commands, nondeterminacy and formal derivation of program* In Comm. ACM, 18(8):453–457, 1975.

[Floyd67] **R. Floyd**. *Assigning meanings to programs* In In Proc. Sympos. Appl. Math., Vol. XIX, pages 19–32, 1967.

[Hoare69] **C.A.R. Hoare**. *An axiomatic basis for computer programming* In Commun. ACM 12(10), 1969.

[King69] **J.C. King**. *A program verifier* In PhD thesis, Dept. of Computer Science, Carnegie-Mellon University, 1969.

[Owicki76] **S. Owicki & D. Gries**. *An axiomatic proof technique for parallel programs I* In Acta Informatica, 6(4):319–340, 1976.