Software Verification

Bertrand Meyer

Lecture 2: Axiomatic semantics
Program Verification: the very idea

P: a program

max (a, b: INTEGER): INTEGER
  do
    if a > b then
      Result := a
    else
      Result := b
  end
end

S: a specification

require true

ensure Result >= a
ensure Result >= b

Does P ⊨ S hold?

The Program Verification problem:

- **Given**: a program P and a specification S
- **Determine**: if every execution of P, for every value of input parameters, satisfies S
What is a theory?

(Think of any mathematical example, e.g. elementary arithmetic)

A theory is a mathematical framework for proving properties about a certain object domain

Such properties are called theorems

Components of a theory:

- **Grammar** (e.g. BNF), defines well-formed formulae (WFF)
- **Axioms**: formulae asserted to be theorems
- **Inference rules**: ways to derive new theorems from previously obtained theorems, which can be applied mechanically
How do we know that an axiomatic semantics (or logic) is “right”?

- **Sound**: every theorem (i.e., deduced property) is a true formula

- **Complete**: every true formula can be established as a theorem (i.e., by applying the inference rules).

- **Decidable**: there exists an effective (terminating) process to establish whether an arbitrary formula is a theorem.
Let $f$ be a well-formed formula

Then

$$\vdash f$$

expresses that $f$ is a theorem
Inference rule

An inference rule is written

\[
\begin{array}{c}
  f_1, f_2, \ldots, f_n \\
\hline
  \quad f_0
\end{array}
\]

It expresses that if \( f_1, f_2, \ldots, f_n \) are theorems, we may infer \( f_0 \) as another theorem.
Example inference rule

“Modus Ponens” (common to many theories):

\[
p, \quad p \Rightarrow q
\]

\[
q
\]
How to obtain theorems

Theorems are obtained from the axioms by zero or more* applications of the inference rules.

*Finite of course
Example: a simple theory of integers

Grammar: Well-Formed Formulae are boolean expressions

- $i_1 = i_2$
- $i_1 < i_2$
- $\neg b_1$
- $b_1 \implies b_2$

where $b_1$ and $b_2$ are boolean expressions, $i_1$ and $i_2$ integer expressions

An integer expression is one of

- 0
- A variable $n$
- $f'$ where $f$ is an integer expression (represents “successor”)
An axiom and axiom schema

\[ \vdash 0 < 0' \]

\[ \vdash f < g \Rightarrow f' < g' \]
An inference rule

\[ P(0), P(f) \Rightarrow P(f') \]

\[ \underline{P(f)} \]
Axiomatic semantics


Purpose:

- Describe the effect of programs through a theory of the underlying programming language, allowing proofs
The theories of interest

Grammar: a well-formed formula is a “Hoare triple”

Informal meaning: A, started in any state satisfying P, will satisfy Q on termination
Software correctness (a quiz)

Consider

\{P\} A \{Q\}

Take this as a job ad in the classifieds

Should a lazy employment candidate hope for a weak or strong $P$? What about $Q$?

Two “special offers”:

1. \{False\} A \{...\}
2. \{...\} A \{True\}
Axiomatic semantics

“Hoare semantics” or “Hoare logic”: a theory describing the partial correctness of programs, plus termination rules
What is an assertion?

Predicate (boolean-valued function) on the set of computation states

**True**: Function that yields True for all states

**False**: Function that yields False for all states

P implies Q: means ∀ s : State, P (s) ⇒ Q (s)

and so on for other boolean operators
Another view of assertions

We may equivalently view an assertion $P$ as a subset of the set of states (the subset where the assertion yields True):

True: Full *State* set
False: Empty subset
implies: subset (inclusion) relation
and: intersection or: union
**Application to a programming language: Eiffel**

\[
\text{extend (new : } G \ ; \ key : H) \\
\quad -- \text{Assuming there is no item of key } key, \\
\quad -- \text{insert } new \text{ with } key; \text{ set } inserted.
\]

**require**

\[
\text{key_not_present: not has (key)}
\]

**ensure**

\[
\text{insertion_done: item (key) = new} \\
\text{key_present: has (key)} \\
\text{inserted: inserted} \\
\text{one_more: count = old count + 1}
\]
The case of postconditions

Postconditions are often predicates on two states

Example (Eiffel, in a class \textit{COUNTER}):
Partial vs total correctness

\{P\} \quad A \quad \{Q\}

Total correctness:
- \(A\), started in any state satisfying \(P\), will terminate in a state satisfying \(Q\)

Partial correctness:
- \(A\), started in any state satisfying \(P\), will, \textit{if it terminates}, yield a state satisfying \(Q\)
Elementary mathematics

Assume we want to prove, on integers

\[ \{ x > 0 \} \land \{ y \geq 0 \} \quad [1] \]

but have actually proved

\[ \{ x > 0 \} \land \{ y = z^2 \} \quad [2] \]

We need properties from other theories, e.g. arithmetic
“EM”: Elementary Mathematics

The mark [EM] will denote results from other theories, taken (in this discussion) without proof

Example:

\[ y = z^2 \quad \text{implies} \quad y \geq 0 \quad \quad [\text{EM}] \]
Rule of consequence

\{P\} A \{Q\}, \quad P' \text{ implies } P, \quad Q \text{ implies } Q'

\hline

\{P'\} A \{Q'\}

Example: \( \{x > 0\} y := x + 2 \{y > 0\} \)
Rule of conjunction

\[
\{P\} A \{Q\}, \quad \{P\} A \{R\}
\]

\[
\underline{\{P\} \quad A \quad \{Q \text{ and } R\}}
\]

Example: \(\{True\} x := 3 \quad \{x > 1 \text{ and } x > 2\}\)
Axiomatic semantics for a programming language

Example language: Graal (from *Introduction to the theory of Programming Languages*)

Scheme: give an axiom or inference rule for every language construct
Abort

\{\text{False} \} \text{ abort } \{P\}
Sequential composition

\[ \{P\} A \{Q\}, \quad \{Q\} B \{R\} \]

\[ \overline{\{P\} \quad A \quad ; \quad B} \quad \{R\} \]

Example:

\[ \{x > 0\} \ x := x + 3 \ ; \ x := x + 1 \ \{x > 4\} \]
Assignment axiom (schema)

\[
\{P [e / x]\} \quad x := e \quad \{P\}
\]

\(P [e/x]\) is the expression obtained from \(P\) by replacing (substituting) every occurrence of \(x\) by \(e\).
Substitution

\[
x \ [x/x] = \\
x \ [y/x] = \\
x \ [x/y] = \\
x \ [z/y] = \\
3 \times x + 1 \ [y/x] =
\]
Applying the assignment axiom

\{y > z - 2\} \ x := x + 1 \ \{y > z - 2\}

\{2 + 2 = 5\} \ x := x + 1 \ \{2 + 2 = 5\}

\{y > 0\} \ x := y \ \{x > 0\}

\{x + 1 > 0\} \ x := x + 1 \ \{x > 0\}
Limits to the assignment axiom

No side effects in expressions!

```plaintext
asquiring_for_trouble (x: in out INTEGER): INTEGER
    do
        x := x + 1;
        global := global + 1;
    Result := 0
    end
```

Do the following hold?

- \{global = 0\} u := asking_for_trouble (a) \{global = 0\}
- \{a = 0\} u := asking_for_trouble (a) \{a = 0\}
The rule of constancy

\{P\} A \{Q\}, \text{ FV}(R) \cap \text{modifies}(A) = \emptyset

\text{FV}(F) = \text{variables free in formula } F

\text{modifies}(A) = \text{variables assigned to in code } A

"Whatever } A \text{ doesn’t modify stays the same"
The rule of constancy: examples

\{ y = 3 \} x := x + 1 \{ y = 3 \}

\{ \forall y \neq 0: y^2 > 0 \} y := y + 1 \{ \forall y \neq 0: y^2 > 0 \}

\{ y = 3 \} x := \sqrt{y} \{ y = 3 \}


\{ bob.age = 65 \} tony.age := 78 \{ bob.age = 65 \}
The rule of constancy: caveats

\{ y = 3 \} x := x + 1 \{ y = 3 \}

\{ \forall y \neq 0: y^2 > 0 \} y := y + 1 \{ \forall y \neq 0: y^2 > 0 \}

\{ y = 3 \} x := \sqrt{y} \{ y = 3 \}

Only if \sqrt{\cdot} doesn’t have side effects on \( y \)!


Only if \( i \neq 3 \)!

\{ bob.age = 65 \} tony.age := 78 \{ bob.age = 65 \}

Only if \( bob \neq tony \), i.e., they are not aliases!
The assignment axiom for arrays

\{ P [ if k = i then e else a[k] / a[k] ] \} \quad a[i] := e \quad \{ P \}

Example:

\{ 3 = i \text{ or } (3 \neq i \text{ and } a[3] = 2) \}

\quad a[i] := 2

\{ a[3] = 2 \}
Conditional rule

\[ \{P \text{ and } c\} A \{Q\}, \quad \{P \text{ and not } c\} B \{Q\} \]

\[ \{P\} \text{ if } c \text{ then } A \text{ else } B \text{ end } \{Q\} \]

Example:
\[ \{y > 0\} \]

\[
\text{if } x > 0 \text{ then } y := y + x \text{ else } y := y - x
\]

\[ \{y > 0\} \]
Prove:

\[ \{ m, n, x, y > 0 \text{ and } x \neq y \text{ and } \gcd(x, y) = \gcd(m, n) \} \]

if \( x > y \) then
  \( x := x - y \)
else
  \( y := y - x \)
end

\[ \{ m, n, x, y > 0 \text{ and } \gcd(x, y) = \gcd(m, n) \} \]
Loop rule (partial correctness)

\[
\{P\} \ A \ \{I\} \quad \{I \text{ and not } c\} \ B \ \{I\}
\]

\[
\{P\} \ \text{from } A \text{ until } c \ \text{loop } B \ \text{end} \ {I \text{ and } c}
\]

\{P\} A \ {I}\) proves initiation: the invariant holds initially

\{I \text{ and not } c\} B \ {I}\) proves consecution (or inductiveness): the invariant is preserved by an arbitrary iteration of the loop
Loop rule (partial correctness, variant)

\[ \{P\} A \{l\}, \ {l \ and \ not \ c} \ B \{l\}, \ {(l \ and \ c) \ implies \ Q} \]

\[
\begin{align*}
\{P\} & \quad \text{from } A \text{ until } c \text{ loop } B \text{ end } \{Q\}
\end{align*}
\]

Example:
{\(y > 3\ and \ n > 0\)}

\[
\text{from } i := 0 \text{ until } i = n \text{ loop}
\]
\[
i := i + 1
\]
\[
y := y + 1
\]

end

{\(y > 3 + n\)}
Loop termination

Must show there is a variant:

An expression $v$ of type INTEGER such that (for a loop \texttt{from A until c loop B end} with precondition $P$):

1. \{P\} A {v \geq 0}
2. {v \geq 0} is an invariant of the loop
3. $v$ decreases with every iteration:
   \[ \forall v_0 > 0: \{v = v_0 \text{ and not } c\} \ B \ {v < v_0}\]

You can reuse invariants used for partial correctness to prove 1, 2, and 3.
Loop termination: example

\{y > 3 \text{ and } n > 0\}

\begin{verbatim}
from i := 0 until i = n loop
  i := i + 1
  y := y + 1
variant
  ??
end
\end{verbatim}

\{y > 3 + n\}
Computing the maximum of an array

from

\[ i := 0 ; \text{Result} := a[1] \]

until

\[ i = a.\text{upper} \]

loop

\[ i := i + 1 \]

Result := \text{max}(\text{Result}, a[i])

end
Loop as approximation strategy

Result = a₁ = Max (a₁ .. a₁)

Result = Max (a₁ .. a₂)

Result = Max (a₁ .. aᵢ)

Result = Max (a₁ .. aₙ)

Loop body:

i := i + 1
Result := max (Result, a [i ])

The loop invariant