Axiomatic Semantics

Programming Languages

William Killian

Millersville University

Outline

- Axiomatic Semantics
  - History
  - Application
  - Hoare Triple
- Weakest Preconditions
- Loop Invariants

Axiomatic Semantics

Definition

- Formal correctness of a program
- An axiomatic semantics consists of
  - A language for stating assertions about programs
  - Rules for establishing the truth of assertions
- Some typical kinds of assertions:
  - This program terminates
  - If this program terminates, the variables **x** and **y** have the same value throughout the execution of the program
  - The array accesses are within the array bounds

History

Program verification is almost as old as programming

“Checking a Large Routine”

Alan Turing, 1949

“Thus the practice of proving programs would seem to lead to solution of three of the most pressing problems in software and programming, namely, reliability, documentation, and compatibility.”

“An Axiomatic Basis for Computer Programming”

C.A.R. Hoare, 1969

“Program testing can be used to show the presence of bugs, but never to show their absence!”

Edgar Dijkstra

C.A.R. Hoare's Perspective

- Very challenging to define languages at the time (namely FORTRAN, ALGOL, and COBOL) that ensured compatibility between all implementations
- Insist that all implementations of the language shall **satisfy the axioms** […] which underlie proofs of properties of programs expressed in the language.

Key Idea

Accept the axioms and rules of inference as the ultimately **definitive specification** of the meaning of the language

Applications of Axiomatic Semantics

- ***Proving has not replaced testing and debugging***
- Proving the correctness of algorithms
- Proving the correctness of hardware descriptions
- Extended static checking
  - Checking array bounds
  - Checking access of uninitialized data
  - Use-after-free detection
- Documentation of programs and interfaces

Axiomatic Semantics Terms

Program State

The values of all variables in an instance of a running program (held in memory)

Assertion

What we expect to be true about a subset of the ***program state*** during execution

Precondition

An ***assertion*** that we expect must hold before executing a statement or procedure

Postcondition

An ***assertion*** that we expect must hold after executing a statement or procedure

Hoare Triple

{A} s {B}

Where

- **A** is the precondition
  - Where **A** holds in some initial state, **σ**
- **s** is the statement(s) being run
  - Where **s** changes the initial state, **σ**, to a new state, **σ'**
- **B** is the postcondition
  - Where **B** holds in the new state, **σ'**

Example

{ y <= x } z := x ; z := z +1 { y < z }

Weakest Preconditions

Weakest Preconditions

{A} s {B}

A is the **least restrictive assertion** that will guarantee **B** after executing **s** where

- **A** is a precondition
- **s** is the statement being executed
- **B** is a postcondition

Weakest Precondition Example

a = b + 1 **{ a > 1 }**

One possible precondition: { b > 10 }

The ***weakest precondition***: { b > 0 }

How? Variable substitution!

a = b + 1 *where* **a > 1**

substitute a for b + 1

b + 1 > 1

b > 0

Proof of Program Correctness

- Given:
  - A postcondition that must be upheld
  - A sequence of statements being executed
- Then:
  - Work back through the program to the first statement.
  - Substitute any/all variables with their assignment expressions
  - If the precondition on the first statement is the same as the program specification, the program is correct.

Sequence Statements

{ Q } s1; s2; s3 **{ Q' }**

- **Q** is the weakest precondition for all statements
- **Q'** is the postcondition

Introduce new assertions (Q_1, Q_2) that serve as pre/post conditions for each statement

{ Q } s1 **{ Q_1 }**

{ Q_1 } s2 **{ Q_2 }**

{ Q_2 } s3 **{ Q' }**

Selection Statement

{ Q } if B then S1 else S2 { Q' }

Consider if the Boolean predicate is true and false:

- **{ Q and B }** S1 **{ Q' }**
- **{ Q and !B }** S2 **{ Q' }**

```
if (x < y) min = x
else       min = y
```

{ min is the smaller value }

Examples

x = x + y

{ y > x }

y = 2 * y

{ y < 5 }

b = a + 2

a = 4b + 2

{ a > 10 }

z = x

if (y < z) z = y

{ z <= x && z <= y }

Loops?

{ Q } while B **do** S **end { Q' }**

Q and **Q'** can share a common logical component, called the **loop invariant** or *inductive hypothesis*

{ I } while B **do** S **end { I and !B }**

Which can infer for each loop body execution:

{ I and B } S **{ I }**

Loop Invariants

The Loop Invariant

- **Q => I**
  - the loop invariant must be true initially
- **{I}** B **{I}**
  - evaluation of the Boolean must not change the validity of I
- **{I and B}** S **{I}**
  - I is not changed by executing the body of the loop
- **(I and (not B)) => Q'**
  - if I is true and B is false, Q' is implied
- The loop terminates
  - can be difficult to prove

Reminders about Loop Invariants

- Serves as a precondition
- *Weakened* form of the postcondition
  - When combined with !B serves as the postcondition
- Not easy to figure out initially

Loop Invariant Example

```
i = 0;
{ B and I }
while (i < 10) {
   { I }
   i = i + 1;
   { I }
}
{ !B and I }
```

```
 B: i <  10
!B: i >= 10
```

Loop Invariant Example

```
A[N] = ?
i = 1;
min = A[0]
{ B and I }
while (i < N) {
  { I }
  if (A[i] < min)
    min = A[i]
  i = i + 1;
  { I }
}
{ !B and I }
```

```
 B: i <  N
!B: i >= N
```