Axiomatic Semantics

So far, we’ve given an operational semantics for imperative statements:

- we can translate them into instructions for a simple abstract machine instructions; or

- we can interpret them directly in an existing language (we’ll see a more formal treatment of this approach soon).

An important alternative approach is to give a logical interpretation to statements.

- The state of an imperative program is defined by the values of all its variables.

- We can characterize a state by giving a logical predicate (or assertion), mentioning the variables, which is satisfied by the values of the variables in that state.

- We can define the semantics of statements by saying how they affect (arbitrary) predicates.
TRIPLES INVOLVING ASSERTIONS

We write a Hoare triple

\[ \{ P \} \; S \; \{ Q \} \]

to mean that if the precondition \( P \) is true before the execution of \( S \) then the postcondition \( Q \) will be true after the execution of \( S \).

Note that the triple says nothing about what happens if \( S \) doesn’t terminate. So we are only characterizing statements that terminate.

Examples of triples (not all stating true things!)

\[ \{ y \geq 3 \} \; x := y + 1 \; \{ x \geq 4 \} \]

\[ \{ x + y = c \} \quad \text{while} \; x > 0 \; \text{do} \]
\[ \quad \quad \quad y := y + 1; \]
\[ \quad \quad \quad x := x - 1 \]
\[ \quad \quad \quad \text{end} \; \{ x + y = c \} \]

\[ \{ y = 2 \} \; x := y + 1 \; \{ x = 4 \} \]

\[ \{ y = 2 \} \; x := y + z \; \{ x = 4 \} \]
How do we distinguish true triples from false?
Who’s to say which ones are true?
It all depends on the **semantics** of statements!

If we work in a suitably structured language, we can give a fixed set of **axioms** and **rules of inference**, one for each kind of statement. We then take as true the set of triples that can be logically **deduced** from these axioms and rules.

Of course, we want to choose axioms and rules that capture our intuitive understanding of what the statements do, and they need to be as **strong** as possible.
ASSIGNMENT AXIOM

\{ P[E/x] \} x := E \{ P \}

where $P[E/x]$ means $P$ with all instances of $x$ replaced by $E$.

This axiom may seem backwards at first, but it makes sense if we start from the postcondition. For example, if we want to show $x \geq 4$ after the execution of

$$x := y + 1$$

then the necessary precondition is $y + 1 \geq 4$, i.e., $y \geq 3$. 
**More Rules for Statements**

**Conditional Rule**

\[
\{ P \land E \} \ S_1 \ { Q \}, \ \{ P \land \neg E \} \ S_2 \ { Q \}
\]

\[\begin{array}{c}
\{ P \} \text{ if } E \text{ then } S_1 \text{ else } S_2 \text{ endif } \ { Q } \\
\end{array}\]

**Composition Rule**

\[
\{ P \} \ S_1 \ { Q \}, \ \{ Q \} \ S_2 \ { R \}
\]

\[\begin{array}{c}
\{ P \} \ S_1; \ S_2 \ { R } \\
\end{array}\]

**While Rule**

\[
\{ P \land E \} \ S \ { P \}
\]

\[\begin{array}{c}
\{ P \} \text{ while } E \text{ do } S \ { P \land \neg E } \\
\end{array}\]
BOOKKEEPING RULES

Consequence Rule

\[ P \Rightarrow P', \{ P' \} \models \{ Q' \}, Q' \Rightarrow Q \]

\[ \{ P \} \models \{ Q \} \]

Here \( P \Rightarrow Q \) means that “\( P \) implies \( Q \),” i.e., “\( Q \) is true whenever \( P \) is true,” i.e. “\( P \) is false or \( Q \) is true.” Hence we always have \( False \Rightarrow Q \) for any \( Q \)!
PROOF TREE EXAMPLE
\[
\{ x + y = c \} \\
\quad y := y + 1 \\
\quad \{ x + y = c + 1 \}
\]

\[
\{ x + y = c \land x > 0 \} \\
\quad y := y + 1 \\
\quad \{ x + y = c + 1 \}
\]

\[
\{ x + y = c \land x > 0 \} \\
\quad y := y + 1; x := x - 1 \\
\quad \{ x + y = c \}
\]

\[
\{ x + y = c \} \\
\quad \text{while } x > 0 \text{ do } y := y + 1; x := x - 1 \text{ end} \\
\quad \{ x + y = c \land x \leq 0 \}
\]

\[
\{ x + y = c \} \\
\quad \text{while } x > 0 \text{ do } y := y + 1; x := x - 1 \text{ end} \\
\quad \{ x + y = c \}
\]
Proof trees can be unwieldy. Because the structure of the tree corresponds directly to the structure of the program code, it is common to use an alternative representation of proofs in which we annotate programs with assertions.

\[
\begin{align*}
\{x + y = c\} \\
\text{while } x > 0 \text{ do} \\
\{x + y = c \land x > 0\} \\
\{x + y = c\} \\
y := y + 1; \\
\{x + y = c + 1\} \\
x := x - 1 \\
\{x + y = c\} \\
\end{align*}
\]

To verify that this is a valid proof, we have to check that the annotations are consistent with each other and with the rules and axioms.
MERITS AND PROBLEMS OF AXIOMATIC SEMANTICS

Gives a very clean semantics for structured statements.

But things get more complicated if we add features like:

- expressions with side-effects
- statements that break out of loops
- procedures
- non-trivial data structures and aliases

Useful for formal proofs of program properties (though these are seldom done).

Thinking in terms of assertions is good for informal reasoning about programs. (And there are beginning to be useful automated theorem proving support tools too.)

Other forms of semantic definition, e.g., natural semantics, also use similar logical structures.