Axiomatic Semantics of Programming Languages

Robert D. Cameron

January 23, 2002

The axiomatic approach to semantics takes a rather unusual view of the meaning of "meaning", i.e., that the meaning of a program is the set of true statements about it. The idea is that the semantic definition of a language should allow true properties (and only true properties) to be proved about programs of the language. For example, one might wish to prove that a program has certain outputs given certain inputs, or that a program always runs to completion (terminates).

In order to prove properties about programs axiomatic semantics deals in *assertions* about the values of program variables. Allowable assertions include all the standard operators of mathematics and logic. For example, $x < 5$ is an assertion that the variable x has a value less than 5, and $y = 0 \vee x/y > 2$ asserts that either y has the value 0 or the ratio of x to y is greater than 2. In addition to these standard types of assertion, axiomatic semantics has one special form of assertion called an *asserted program* or *pre-post formula* which has the form $\{\mathcal{P}\}$ **S** $\{\mathcal{Q}\}$ where $\mathcal{P}$ and $\mathcal{Q}$ are simple logical assertions as above and **S** is a statement of the programming language. The meaning of such an assertion is that if $\mathcal{P}$ is true before execution of **S** and **S** executes to completion, then $\mathcal{Q}$ is true after execution of **S**. $\mathcal{P}$ and $\mathcal{Q}$ are called, respectively, the *precondition* and *postcondition* of the assertion.

The axiomatic semantics of the language is defined by specifying, for each type of statement **S** of the language, the conditions under which asserted programs involving **S** can be assumed to be true. For example, the rule for assignment statements (typically) is that all assertions of the form $\{\mathcal{P}[x \leftarrow t]\}$ x `:=` t $\{\mathcal{P}\}$ are true, where $\mathcal{P}[x \leftarrow t]$ means the assertion obtained by substituting the expression t for all occurrences of x in the assertion $\mathcal{P}$. According to this rule, the following assertions are true.

$$\{y = 7\}\ x\ \texttt{:=}\ x\ \texttt{+}\ 1\ \{y = 7\}$$
$$\{x + 1 = 8\}\ x\ \texttt{:=}\ x\ \texttt{+}\ 1\ \{x = 8\}$$

In the first case, the postcondition does not involve x, so no substitutions have to be made to derive the precondition. In the second case, a substitution is required: x in the postcondition is replaced by $x + 1$ in the precondition. Both of these assertions correspond with what we expect to be true about assignment statements. The rule $\{\mathcal{P}[x \leftarrow t]\}$ `x := t` $\{\mathcal{P}\}$ is called the *assignment axiom*.

The condition $\mathcal{P}[x \leftarrow t]$ is called the *weakest precondition* for the statement $x := t$ and the postcondition $\mathcal{P}$. In general, the weakest precondition is the one which places the fewest restrictions on program variables and still gives the validity of a given postcondition. Consider, for example, the statement $x := y + 3$ and the postcondition $x > 0$. This postcondition is true with the precondition $y > 0$, and also with the weakest precondition $y + 3 > 0$. Note, however, that $y > 0$ makes a stronger statement about the value of y than does $y + 3 > 0$, since the former excludes the values -2 and -1 for y, whereas the latter allows those values.

Finding the weakest precondition allows proofs for the most general case (fewest restrictions on input values) to be found. In general, the axiomatic definition of programming languages seeks to

provide rules for determining the weakest precondition given any language statement **S** and any postcondition $\mathcal{Q}$.

Many of the rules in axiomatic semantics require that the truth of one asserted program be based on the truths of other asserted programs. For example, the truth of $\{\mathcal{P}\}$ **S1; S2** $\{\mathcal{Q}\}$ follows if and only if there is an assertion $\mathcal{R}$ such that $\{\mathcal{P}\}$ **S1** $\{\mathcal{R}\}$ and $\{\mathcal{R}\}$ **S2** $\{\mathcal{Q}\}$ are both true. A rule of this type is known as a *rule of inference* and is usually expressed using the following notation.

$$\frac{\{\mathcal{P}\}\ \mathbf{S1}\ \{\mathcal{R}\}, \{\mathcal{R}\}\ \mathbf{S2}\ \{\mathcal{Q}\}}{\{\mathcal{P}\}\ \mathbf{S1;}\ \mathbf{S2}\ \{\mathcal{Q}\}}$$

If each of the conditions listed above the line holds, it is then valid to infer the truth of the assertion below the line. This particular rule is usually called the *composition rule* and describes how proofs about sequences of statements can be derived from proofs about individual statements (in this formulation, a sequence of statements is treated as a type of statement itself).

Example 1. Using the assignment axiom and the rule of composition, prove that the following sequence of statements exchanges the values of variables x and y.

$$\begin{aligned} &x \texttt{ := } x \texttt{ - } y\texttt{;}\\ &y \texttt{ := } x \texttt{ + } y\texttt{;}\\ &x \texttt{ := } y \texttt{ - } x \end{aligned}$$

In order to do the proof, we must first establish appropriate pre- and postconditions. The precondition is simply that x and y have some particular values, say A and B: $\{x = A \wedge y = B\}$. The postcondition is that these values are exchanged: $\{x = B \wedge y = A\}$. The problem is thus to prove the validity of the following asserted program.

$$\begin{aligned} &\{x = A \wedge y = B\}\\ &x \texttt{ := } x \texttt{ - } y\texttt{;}\\ &y \texttt{ := } x \texttt{ + } y\texttt{;}\\ &x \texttt{ := } y \texttt{ - } x\\ &\{x = B \wedge y = A\} \end{aligned} \tag{1}$$

A standard strategy in such proofs is to work backwards from the desired result. Start with the final statement and the given postcondition and derive a valid asserted program involving that statement, i.e., find $\mathcal{P}1$ in the following.

$$\begin{aligned} &\{P1\}\\ &x \texttt{ := } y \texttt{ - } x\texttt{;}\\ &\{x = B \wedge y = A\} \end{aligned}$$

Using the assignment axiom, the desired precondition is derived by back substitution of $y - x$ for x in the postcondition, giving the following.

$$\begin{aligned} &\{y - x = B \wedge y = A\}\\ &x \texttt{ := } y \texttt{ - } x\texttt{;}\\ &\{x = B \wedge y = A\} \end{aligned} \tag{2}$$

A similar process is then carried out using the newly derived precondition of the last statement as the postcondition for the penultimate [second last] statement. Using the assignment axiom once more, the precondition is again found by back substitution, this time of $x + y$ for y.

$$\begin{array}{l} \{x + y - x = B \wedge x + y = A\} \\ y \texttt{ := } x \texttt{ + } y\texttt{;} \\ \{y - x = B \wedge y = A\} \end{array} \tag{3}$$

The precondition here can be simplified to $\{y = B \wedge x + y = A\}$, and then the rule of composition can be applied to (3) and (2) to yield the following valid asserted program.

$$\begin{array}{l} \{y = B \wedge x + y = A\} \\ y \texttt{ := } x \texttt{ + } y\texttt{;} \\ x \texttt{ := } y \texttt{ - } x\texttt{;} \\ \{x = B \wedge y = A\} \end{array} \tag{4}$$

Now the precondition of (4) can be used as the postcondition of the first statement. By the assignment axiom, the following asserted program is derived.

$$\begin{array}{l} \{y = B \wedge x - y + y = A\} \\ x \texttt{ := } x \texttt{ - } y\texttt{;} \\ \{y = B \wedge x + y = A\} \end{array} \tag{5}$$

The precondition can be simplified to $\{y = B \wedge x = A\}$. Finally, the rule of composition allows us to combine (5) and (4) proving the validity of the original asserted program (1) and hence completing the proof.

In addition to the rules for assignment and composition, there are also some general purpose rules of inference that are often needed for axiomatic proofs. These rules are not related to any particular language constructs and apply in general to axiomatic proofs in any language. The *consequence rule* allows us to make substitutions of stronger preconditions and/or weaker postconditions, as follows.

$$\frac{\mathcal{P} \supset \mathcal{P}1, \{\mathcal{P}1\}\ \mathbf{S}\ \{\mathcal{Q}1\}, \mathcal{Q}1 \supset \mathcal{Q}}{\{\mathcal{P}\}\ \mathbf{S}\ \{\mathcal{Q}\}}$$

This rule states that, for any given asserted program $\{\mathcal{P}1\}\ \ \mathbf{S}\ \ \{\mathcal{Q}1\}$, $\mathcal{P}1$ can be replaced by any condition $\mathcal{P}$ from which $\mathcal{P}1$ can be inferred ($\mathcal{P} \supset \mathcal{P}1$) and $\mathcal{Q}1$ can be replaced by any condition $\mathcal{Q}$ which it implies. This rule just allows us to formally make the substitutions that we informally recognize as valid. In addition, the *and rule* and the *or rule* have similar functions and are described respectively as follows.

$$\frac{\{\mathcal{P}1\}\ \mathbf{S}\ \{\mathcal{Q}1\}, \{\mathcal{P}2\}\ \mathbf{S}\ \{\mathcal{Q}2\}}{\{\mathcal{P}1 \wedge \mathcal{P}2\}\ \mathbf{S}\ \{\mathcal{Q}1 \wedge \mathcal{Q}2\}}$$

$$\frac{\{\mathcal{P}1\}\ \mathbf{S}\ \{\mathcal{Q}1\}, \{\mathcal{P}2\}\ \mathbf{S}\ \{\mathcal{Q}2\}}{\{\mathcal{P}1 \vee \mathcal{P}2\}\ \mathbf{S}\ \{\mathcal{Q}1 \vee \mathcal{Q}2\}}$$

In practice, the latter two rules are infrequently used, but the consequence rule is often applied.

In addition to assignment and statement sequencing, there are two other common types of control structure in programming languages, namely conditional statements and repetitive statements.

As a typical representative for the conditional constructs of many languages, the `if` statement can be specified by the following rule of inference.[1]

$$\frac{\{\mathcal{P} \wedge \mathbf{E}\}\ \mathbf{S1}\ \{\mathcal{Q}\}, \{\mathcal{P} \wedge \sim \mathbf{E}\}\ \mathbf{S2}\ \{\mathcal{Q}\}}{\{\mathcal{P}\}\ \texttt{if}\ \mathbf{E}\ \texttt{then}\ \mathbf{S1}\ \texttt{else}\ \mathbf{S2}\ \texttt{fi}\ \{\mathcal{Q}\}}$$

This rule states the conditions under which an assertion about an `if` statement can be inferred from assertions for the `then` and `else` branches of the statement. It can be used in the proof of an `if-then-else` asserted program by breaking the proof into two subsidiary proofs, one each for the `then` and `else` branches. Note that along the `then` branch, the precondition is augmented to include the statement predicate (the `if` part); this reflects the fact that the `then` branch is not executed unless the predicate is true. Similarly, the precondition along the `else` branch includes the logical negation of the predicate.

The axiomatic treatment for iterative constructs can be illustrated by the rule for `while` loops.

$$\frac{\{\mathcal{P} \wedge \mathbf{E}\}\ \mathbf{S}\ \{\mathcal{P}\}}{\{\mathcal{P}\}\ \texttt{while}\ \mathbf{E}\ \texttt{do}\ \mathbf{S}\ \texttt{od}\ \{\mathcal{P} \wedge \sim \mathbf{E}\}}$$

Note that, in this rule, the precondition $\mathcal{P}$ is also part of the postcondition; $\mathcal{P}$ is said to be a *loop invariant*.

Finding an appropriate loop invariant is often the critical step in proofs. Of course, there are many conditions which are trivially invariant through the execution of a loop, for example, any condition which does not depend on any loop variables. In general, however, such conditions are of no use in doing axiomatic proofs; what is needed is a condition which is some sort of invariant relationship between various loop variables. Such a condition must be true of the initial values of loop variables and true after each execution of the loop body. The nature of such loop invariants will be better illustrated in the example below.

The simple language defined to include only assignments, compound statements, `if-then-else` statements and `while` loops is called SLINT. It is not a real language, but is useful to demonstrate basic axiomatic techniques without the complexities in larger languages. Its axiomatic semantics consist exactly of the rules introduced above, i.e., the assignment axiom and composition, `if-then-else` and `while` rules plus the three general purpose rules for consequence, "and" and "or". Although the SLINT is simple in nature it can be used to illustrate nontrivial proofs of program correctness.

Example 2. Consider the following program **S** for performing integer division of the natural numbers x and y.

```
q := 0;
r := x;
while r >= y do
  r := r - y;
  q := q + 1
od
```

Show that $\{\mathcal{P}0\}$ **S** $\{\mathcal{Q}0\}$ holds where $\mathcal{P}0$ is $x \geq 0 \wedge y \geq 0$ and $\mathcal{Q}0$ is $q * y + r = x \wedge 0 \leq r < y$. That is, if x and y are nonnegative integers and the program runs to completion then q and r are respectively the quotient and remainder of x divided by y.

[1] Here, the `fi` keyword is used to mark the end of the `if` statement in the style of Algol 68.

Again, we work backwards from the desired result. By the composition rule, the proof is complete if we can find an assertion $\mathcal{P}1$ such that the following asserted programs hold.

$$
\begin{array}{l}
\{x \geq 0 \wedge y \geq 0\} \\
q \texttt{ := } 0\texttt{;} \qquad (1) \\
r \texttt{ := } x\texttt{;} \\
\{\mathcal{P}1\}
\end{array}
$$

$$
\begin{array}{l}
\{\mathcal{P}1\} \\
\texttt{while } r \texttt{ >= } y \texttt{ do} \qquad (2) \\
\quad r \texttt{ := } r \texttt{ - } y\texttt{;} \\
\quad q \texttt{ := } q \texttt{ + 1} \\
\texttt{od} \\
\{q * y + r = x \wedge 0 \leq r < y\}
\end{array}
$$

Now $\mathcal{P}1$ can be analyzed as the precondition of the `while` loop, based on the `while` rule and the given postcondition. $\mathcal{P}1$ is, in fact, the loop invariant, and its determination is the essential step in doing the proof.

The nature of $\mathcal{P}1$ can be discovered by looking at the general form of asserted programs for `while` loops as taken from the `while` rule.

$$\{\mathcal{P}\} \texttt{ while } \mathbf{E} \texttt{ do } \mathbf{S} \texttt{ od } \{\mathcal{P} \wedge \sim \mathbf{E}\}$$

By matching $\{\mathcal{P} \wedge \sim \mathbf{E}\}$ against the postcondition of (2), we might guess that $\mathcal{P}1$ should be taken as $q * y + r = x \wedge 0 \leq r$, since the assertion $r < y$ corresponds to $\sim \mathbf{E}$. Note that this condition is true before the loop is executed, since $q * y + r = 0 * y + r = r = x$ and $x \geq 0$ is given implying $r \geq 0$. Also note that each time through the loop the value of $q * y$ increases by y, while the value of r decreases by y, so their sum, which is initially x, remains invariant. The value of r always remains greater than 0 since it is only decremented by y each time and the loop is not entered if r is less than y. Thus, our guess is correct and the condition

$$q * y + r = x \wedge 0 \leq r$$

is in fact the desired loop invariant $\mathcal{P}1$. However, our argument here is an informal one, so we must complete the proof using the formal steps.

Knowing $\mathcal{P}1$, then, we can now use the rule of inference for `while` loops to show the truth of (2) if we can show the truth of the following asserted program.

$$
\begin{array}{l}
\{q * y + r = x \wedge 0 \leq r \wedge r \geq y\} \\
r \texttt{ := } r \texttt{ - } y\texttt{;} \qquad (3) \\
q \texttt{ := } q \texttt{ + 1} \\
\{q * y + r = x \wedge 0 \leq r\}
\end{array}
$$

Note that we have added the condition $r \geq y$ to the precondition of the loop body as specified by the `while` rule. Working backwards, we know that the following assertion is valid by the assignment axiom.

$$
\begin{array}{l}
\{(q + 1) * y + r = x \wedge 0 \leq r\} \\
q \texttt{ := } q \texttt{ + 1} \\
\{q * y + r = x \wedge 0 \leq r\}
\end{array}
$$

Using the assignment axiom one more time we get the following valid assertion.

$$\begin{array}{l}\{(q+1)*y+(r-y)=x \wedge 0 \leq (r-y)\} \\ r \texttt{ := } r \texttt{ - } y\texttt{;} \\ \{(q+1)*y+r=x \wedge 0 \leq r\}\end{array}$$

Using the rule of composition on these last two asserted programs gives us the following.

$$\begin{array}{l}\{(q+1)*y+(r-y)=x \wedge 0 \leq (r-y)\} \\ r \texttt{ := } r \texttt{ - } y\texttt{;} \\ q \texttt{ := } q \texttt{ + } 1 \\ \{q*y+r=x \wedge 0 \leq r\}\end{array} \tag{4}$$

The precondition here is not the same as in (3), but can be simplified to $\{q*y+r=x \wedge r \geq y\}$ which is weaker than the precondition of (3) (and hence is logically implied by that condition). Using this fact, the validity of (3) follows from that of (4) by the rule of consequence. Application of the `while` rule establishes (2) from (3).

To complete the proof, it now remains to show the validity of the following asserted program.

$$\begin{array}{l}\{x \geq 0 \wedge y \geq 0\} \\ q \texttt{ := } 0\texttt{;} \\ r \texttt{ := } x \\ \{q*y+r=x \wedge 0 \leq r\}\end{array} \tag{5}$$

This is just (1) above, with the appropriate substitution made for the postcondition $\mathcal{P}1$. Again using the assignment axiom and working backwards, we have

$$\begin{array}{l}\{q*y+x=x \wedge 0 \leq x\} \\ r \texttt{ := } x \\ \{q*y+r=x \wedge 0 \leq r\}\end{array}$$

and

$$\begin{array}{l}\{0*y+x=x \wedge 0 \leq x\} \\ q \texttt{ := } 0 \\ \{q*y+x=x \wedge 0 \leq x\}\end{array}$$

The precondition here reduces to $\{x \geq 0\}$ and can be replaced by $\{x \geq 0 \wedge y \geq 0\}$ by the consequence rule. The composition rule is then applied to these latter two asserted programs to establish (5) and complete the proof.

In this example, note that the division is performed correctly whenever $x \geq 0$ and $y > 0$, but that the program does not terminate if $y = 0$. Nevertheless, the asserted program is valid in both cases. In general, proving an asserted program only means that *if* the program terminates, it gives the correct result. Proving such an asserted program is often said to be a proof of *partial correctness*; a proof of *total correctness* also requires a proof that the program terminates. Such proofs require additional axiomatic techniques that we will not go into here.

Viewpoint on Axiomatic Semantics

We have introduced axiomatic semantics for a simple programming language with a small number of constructs. Researchers in the area have also developed techniques for handling more difficult sorts of language constructs such as procedure calling with parameters, recursion and subscripted variables. Unfortunately, axiomatic semantics cannot be practically applied to most current programming languages, however, as these languages generally violate certain necessary assumptions for the validity of axiomatic semantics.

From an axiomatic viewpoint, the biggest problems with most current languages are the possibilities of side-effects in expressions and aliasing of variables. Side-effects in expressions typically arise when a function call within the expression results in the value of a global variable being changed. Such a possibility would invalidate the assignment axiom and the rules of inference for `if-then-else` statements and `while` loops. Aliasing of variables refers to the possiblity that two different names may be used to denote the same memory location. This typically happens using call-by-reference parameter passing (`var` parameters in Pascal). Aliasing invalidates the assignment axiom; the precondition under aliasing would have to be modified to substitute the expression for all variables that named the same location as the variable to which the assignment was being made.

The problems of applying axiomatic semantics to current languages suggest new language design principles to support axiomatic definitions. Two positive steps would be to prohibit side effects in expressions and eliminate aliasing. This, coupled with ongoing research into automated program verification systems, could lead to programming languages and environments supporting the development of highly reliable programs.