Axiomatic Semantics

Angelo Gargantini: Appunti per le lezioni di Informatica prese dal libro di Sebesta

10th May 2005

Axiomatic semantics was defined in conjunction with the development of a method to prove the correctness of programs. Such correctness proofs, when they can be constructed, show that a program performs the computation described by its specification. In a proof, each statement of a program is both preceded and followed by a logical expression that specifies constraints on program variables. These, rather than the entire state of an abstract machine (as with operational semantics), are used to specify the meaning of the statement. The notation used to describe constraints, indeed the language of axiomatic semantics, is predicate calculus. Although simple Boolean expressions are often adequate to express constraints, in some cases they are not.

Assertions

Axiomatic semantics is based on mathematical logic. The logical expressions are called predicates, or assertions. An assertion immediately preceding a program statement describes the constraints on the program variables at that point in the program. An assertion immediately following a statement describes the new constraints on those variables (and possibly others, after exertion of the statement. These assertions are called the precondition and postcondition, respectively, of the statement. Developing an axiomatic description or proof of a given program requires that every statement in the program have both a precondition and a postcondition.

In the following sections, we examine assertions from the point of view that preconditions for statements are computed from given postconditions althouh it is possible to consider these in the opposite sense. We assume all variables are integer type. As a simple example, consider the following assignment statement and postcondition:

```
sum = 2 * x + 1 {sum > 1}
```

Precondition and postcondition assertions are presented in braces to distinguish them from program statements. One possible precondition for this statement is {x > 10}.

Weakest Preconditions

The *weakest precondition* is the least restrictive precondition that will guarantee the validity of the associated postcondition. For example, in the above statement and postcondition, {x > 10}, {x > 50}, and {x > 1000} are all valid preconditions. The weakest of all preconditions in this case is {x > 0}.

If the weakest precondition can be computed from the given postcondition for each statement of a language, then correctness proofs can be constructed for programs in that language. The proof is begun by using the desired results of the program's execution as the postcondition of the last statement of the program. This postcondition, along with the last statement, is used to compute the weakest precondition for the last statement. This precondition is then used as the postcondition for the second last statement. This process continues until the beginning of the program is reached. At that point. the precondition of the first statement states the conditions under which the program will compute the desired results.

For some program statements, the computation of a weakest precondition from the statement and a postcondition is simple and can be specified by an axiom. In most cases, however, the weakest precondition can be computed only by an inference rule. An axiom is a logical statement that is assumed to be true; an inference rule is a method of inferring the truth of one assertion on the basis of the values of other assertions.

To use axiomatic semantics with a given programming language, whether for correctness proofs or for formal semantics specifications, either an axiom or an inference rule must be available for each kind of statement in the language. In the following subsections, we present an axiom for assigning-statements and inference rules for statement sequences, selection statement and logical pretest loops. Note that we assume that neither arithmetic nor Boolean expressions have side effects

Assignment Statements

Let x = E be a general assignment statement and Q be its postcondition. Then its precondition, P, is defined by the axiom

P = Q $_{[x \rightarrow E]}$

which means that P is computed as Q with all instances of x replaced by E. For example, if we have the assignment statement and postcondition

```
a = b / 2 - 1 {a < 10}
```

the weakest precondition is computed by substituting b / 2 - 1 in the assertion {a < 10}, as follows:

```
b / 2 - 1 < 10
b < 22
```

Thus the weakest precondition for the given assignment and postcondition is {b < 22}. Recall that the assignment axiom is guaranteed to be true only in the absence of side effects. An assignment statement has a side effect if it changes some variable other than its left side.

The usual notation for specifying the axiomatic semantics of a given statement form is

{P} S {Q}

where P is the precondition, Q is the postcondition, and S is the statement form. In the case of the assignment statement, the notation is

{Q $_{[x \rightarrow E]}$} x=E {Q}

As another example of computing a precondition for an assignment statement, consider the following:

```
x = 2 * y - 3 {x > 25}
```

The precondition is computed as follows:

```
2 * y - 3 > 25
y > 14
```

So {y > 14} is the weakest precondition for this assignment statement and postcondition.

Note that the appearance of the left side of the assignment statement in its right side does not affect the process of computing the weakest precondition. For example, for

```
x=x+y-3 {x> 10}
```

the weakest precondition is

```
x + y - 3 > 10
y > 13 - x
```

At the beginning of our discussion of axiomatic semantics, we stated that axiomatic semantics was developed to prove the correctness of programs. In light of that, it is natural at this point to wonder how the axiom for ment statements can be used to prove anything. Here is how: A given assignment statement with both a precondition and a postcondition can be considered a theorem. If the assignment axiom, when applied to the postcondition and the assignment statement, produces the given precondition, the theorem is proved. For example, consider the logical statement

{x > 3} x = x - 3 {x > 0}

Using the assignment axiom on x = x - 3 {x > 0}

produces {x > 3}, which is the given precondition. Therefore we have proven the logical statement above.

Next, consider the logical statement

{x > 5} x = x - 3 {x > 0}

In this case, the given precondition, {x > 5}, is not the same as the assertion produced by the axiom. However, it is obvious that {x > 5} implies {x > 3} To use this in a proof, we need an inference rule, named the *rule of consequence*. The general form of an inference rule is

$$\frac{S1, S2, ..., Sn}{S}$$

which states that if S1, S2, ..., and Sn are true, then the truth of S can be inferred.

(AG) For example, for the substitution, we can write the following rule:

$$\frac{P = Q_{[x \rightarrow E]}}{\{P\}x = E\{Q\}}$$

The form of the rule of consequence is

$$\frac{\{P\}S\{Q\}, P' \rightarrow P, Q \rightarrow Q'}{\{P'\}S\{Q'\}}$$

The $\rightarrow$ symbol means "implies," and S can be any program statement.' rule can be stated as follows: If the logical statement {P} S {Q} is true, the assertion P implies the assertion P, and the assertion Q implies the assertion Q' , then it can be inferred that {P'} S {Q'}. In other words, the rule of consequence says that a postcondition can always be weakened and a precondition can always be strengthened. This is quite useful in program proofs. For example it allows the completion of the proof of the last logical statement example above. If we let P be {x > 3}, Q and Q' be {x > 0}, P be {x > 5}, we have

$$\frac{\{x > 3\}x = x - 3\{x > 0\}, (x > 5) \rightarrow (x > 3), (x > 0) \rightarrow (x > 0)}{\{x > 5\}x = x - 3\{x > 0\}}$$

This completes the proof.

Sequences

The weakest precondition for a sequence of statements cannot be described by an axiom, because the precondition depends on the particular kinds of statements in the sequence. In this case, the precondition can only be described with an inference rule. Let SI and S2 be adjacent program statements. If S1 and S2 have the following pre- and postconditions

{PI} S1 {P2} {P2} S2 {P3}

the inference rule for such a two-statement sequence is

$$\frac{\{PI\}S1\{P2\}, \{P2\}S2\{P3\}}{\{P1\}S1; S2\{P3\}}$$

So, for the above example, {PI} S1; S2 {P3} describes the axiomatic semantics of the sequence S1; S2. If S1 and S2 are the assignment statements

xl = El

and

x2 = E2 then we have

{P3$_{[x2 \rightarrow E2]}$} x2 = E2 {P3}

{(P3$_{[x2 \rightarrow E2]}$)$_{[x1 \rightarrow E1]}$} x1 = E1{P3$_{[x2 \rightarrow E2]}$}

Therefore, the weakest precondition for the sequence xl = El; x2 = E2 with postcondition P3 is {(P3$_{[x2 \rightarrow E2]}$)$_{[x1 \rightarrow E1]}$}.

For example, consider the following sequence and postcondition:

```
y = 3 * x + 1;
x = y + 3; {x < 10}
```

The precondition for the last assignment statement is

```
y < 7
```

This is then used as the postcondition for the first. The precondition for the first assignment statement can now be computed:

```
3 * x + 1 < 7
x < 2
```

Selection

We next consider the inference rule for selection statements. We consider only selections that include else clauses. The inference rule is

$$\frac{\{B \; and \; P\}S1\{Q\}, \{(not \; B) \; and \; P\}S2\{Q\}}{\{P\}\texttt{if}\; B \;\texttt{then}\; S1 \;\texttt{else}\; S2\{Q\}}$$

This rule indicates that selection statements must be proven for both of their cases. The first logical statement above the line is the then clause; the second is the else clause.

Consider the following example of the computation using the selection inference rule. The example selection statement is

```
if (x > 0)   y = y - 1
else y = y + 1
```

Suppose the postcondition for this selection statement is {y > 0}. We can use the axiom for assignment on the then clause

```
y = y - 1 {y > 0}
```

This produces {y - 1 > 0} or {y > 1}. Now we apply the same axiom to the else clause

```
y = y + 1 {y > 0}
```

This produces the precondition {y + 1 > 0} or {y > -l}. Because {y > 1} → {y > -1}, the rule of consequence allows us to use {y > 1} for the precondition of the selection statement.

Logical Pretest Loops - While loops

Another essential construct of an imperative programming language is logical pretest, or *while loop*. Computing the weakest precondition for a while loop is inherently more difficult than for a sequence because the number of iterations cannot always be predetermined. In a case where the number of iterations is known, the loop can be treated as a sequence.

The problem of computing the weakest precondition for loops is similar to the problem of proving a theorem about all positive integers. In the latter case, induction is normally used, and the same inductive method can be used for loops. The principal step in induction is finding an *inductive hypothesis*. The corresponding step in the axiomatic semantics of a while loop is finding an assertion called a loop invariant, which is crucial to finding the weakest precondition.

The inference rule for computing the precondition for a while loop is

$$\frac{\{I \; and \; B\} S \{I\}}{\{I\} \; \texttt{while} \; B \; \texttt{do} \; S \; \texttt{end} \; \{I \; and \; (not \; B)\}}$$

where I is the *loop invariant*.

The axiomatic description of a while loop is written as

$\{P\}$ `while` B `do` S `end`$\{Q\}$

The loop invariant must satisfy a number of requirements to be useful. First, the weakest precondition for the while must guarantee the truth of the loop invariant. In turn, the loop invariant must guarantee the truth of the postcondition upon loop termination. These constraints move us from the inference rule to the axiomatic description. During execution of the loop, the truth of the loop invariant must be unaffected by the evaluation of the loop-controlling Boolean expression and the loop body statements. Hence the name invariant.

Another complicating factor for while loops is the question of loop termination. If Q is the postcondition that holds immediately after loop exit, then a precondition P for the loop is one that guarantees Q at loop exit and also guarantees that the loop terminates.

The complete axiomatic description ot a while construct requires all of the following to be true, in which I is the loop invariant:

- P $\rightarrow$ I
- { I and B} S { I}
- { I and (not B)} $\rightarrow$Q
- the loop terminates

To find a loop invariant, we can use a method similar to that used for determining the inductive hypothesis in mathematical induction, which is as follows: the relationship for a few cases is computed, with the hope that a pattern emerges that will apply to the general case. It is helpful to treat the process of producing a weakest precondition as a function, `wp`. In general

```
wp(statement, postcondition) = precondition
```

(ndr) In case of an assignement, we have:

```
wp(x = E, Q ) = Q [x→E]
```

To find I, we use the loop postcondition Q to compute preconditions for several different numbers of iterations of the loop body, starting with none. If the loop body contains a single assignment statement, the axiom for assignment statements can be used to compute these cases. Consider the example loop

```
while y <> x do y = y + 1 end {y = x}
```

Remember that the equal sign is being used for two different purposes here. In assertions, it means mathematical equality; outside assertions, it means the assignment operator.

For zero iterations, the weakest precondition is, obviously,

```
{y = x}
```

For one iteration, it is

```
wp(y = y + 1, {y = x}) = {y + 1 = x} ≡ {y = x - 1}
```

For two iterations, it is

```
wp(y = y + 1, {y = x - 1}) = {y + 1 = x - 1} ≡ {y = x - 2}
```

For three iterations, it is

```
wp(y = y + 1, {y = x - 2}) = {y + 1 = x - 2} ≡ {y = x - 3}
```

It is now obvious that {y < x} will suffice for cases of one or more iterations. Combining this with {y = x} for the zero iterations case, we get {y <= x}, which can be used for the loop invariant. A precondition for the while statement can be determined from the loop invariant. In this example, P = I can be used.

We must ensure that our choice satisfies the four criteria for I for our example loop. First, because P = I, P $\rightarrow$ I. The second requirement is that it must be true that

$$\{I \, and \, B\} S \{I\}$$

In our example, we have

```
{y <= x and y <> x} y = y + 1 {y <= x}
```

Applying the assignment axiom to

```
y = y + 1 {y <= x}
```

we get {y + l <= x}, which is equivalent to {y < x}, which is implied by {y <= x and y <> x). So, the statement above is proven. Next, we must have

$$\{I \, and \, (not \, B)\} \rightarrow Q$$

In our example, we have

```
{(y <= x) and not (y <> x)} → {y = x},
{(y <= x) and (y = x)} → {y = x}
{y = x} → {y = x}
```

So this is obviously true. Next, loop termination must be considered. In this example, the question is whether the loop

```
{y <= x} while y <> x do y = y + 1 end {y = x}
```

terminates. Recalling that x and y are assumed to be integer variables, we can easily see drat this loop does terminate. The precondition guarantees that y initially is not larger than x. The loop body increases y with each iteration, until y is equal to x. No matter how much smaller y is than x initially, it will eventually become equal to x. So the loop will terminate. Because our choice of I satisfies all four criteria, it is adequate for the loop invariant and the loop precondition.

The process used above to compute the invariant for a loop does not always produce an assertion that is the weakest precondition (although it does in the example above).

As another example of finding a loop invariant, consider the following loop statement:

```
while s > l do s = s/2 end {s = 1}
```

As above, we use the assignment axiom to try to find a loop invariant and a precondition for the loop. For zero iterations, the weakest precondition is {s = l}. For one iteration, it is

```
wp(s = s / 2, {s = 1}) = {s / 2 = l}≡{s = 2}
```

For two iterations, it is

```
wp(s = s / 2, {s = 2}) = {s / 2 = 2}≡{s = 4}
```

For three iterations, it is

```
wp(s = s / 2, {s = 4}) = {s / 2 = 4}≡{s = 8}
```

From these cases, we can see clearly that the invariant is

s is a nonnegative power of 2

Once again, the computed I can serve as P, and I passes the four requirements. Unlike our earlier example of finding a loop precondition, this one clearly is not a weakest precondition. Consider using the precondition {s > l}. The logical statement

```
{s > 1} while s > l do s = s/2 end {s = 1}
```

can easily be proven, and this precondition is significantly broader than the one computed above. The loop and precondition are satisfied for any positive value for s, not just powers of 2 as the process indicates. Because of the rule of consequence, using a precondition that is stronger than the weakest precondition does not invalidate a proof. (But normally it is more difficult to prove the Hoare triple with weaker preconditions *ndr*)

Finding loop invariants is not always easy. It is helpful to understand the nature of these invariants. First, a loop invariant is a weakened version of the loop postcondition and also a precondition for the loop. So I must be weak enough to be satisfied prior to the beginning of loop execution, but when combined with the loop exit condition, it must be strong enough to force the truth of the postcondition.

Because of the difficulty of proving loop termination, that requirement is often ignored. If loop termination can be shown, the axiomatic description ot the loop is called *total correctness*. If the other conditions can be met but termination is not guaranteed, it is called *partial correctness*.

In more complex loops, finding a suitable loop invariant, even for partial correctness requires a good deal of ingenuity. Because computing the precondition for a while loop depends on finding a loop invariant, proving the correctness of programs with while loops using axiomatic semantics can be difficult.

The following is an example of a proof ot correctness of a pseudocode program that computes the factorial function.

```
{n >= 0}
count = n;
fact = 1;
while count <> 0 do
  fact = fact * count;
  count = count - 1;
end
{fact = n!}
```

The method described earlier for finding the loop invariant does not work for the loop in this example. Some ingenuity is required here, which can be aided by a brief study of the code. The loop computes the factorial function in order of the last multiplication first; that is, (n - 1) * n is done first, assuming n is greater than l. So, part of the invariant can be

```
fact = (count + 1) * (count + 2) *... * (n - 1) * n
```

But we must also ensure that count is always nonnegative, which we can do by adding that to the part above, to get

```
I= (fact = (count + 1) * ... * n) AND (count >= 0)
```

Next, we must check that this I meets the requirements for invariants. We once again let I also be used for P, so P clearly implies I. The next question is

{ I and B} S { I}

I and B is

```
((fact = (count + 1) * ... * n) AND (count >= 0)) AND (count <> 0)
```

which reduces to

```
(fact = (count + 1) * ... * n) AND (count > 0)
```

In our case, we must compute the precondition of the body of the loop, using the invariant for the postcondition. For

{P} count = count - 1 {I}

we compute P to be

```
{(fact = count * (count + 1) * ... * n) AND (count >= 1)}
```

Using this as the postcondition for the first assignment in the loop body,

```
{P} fact = fact * count {(fact = count * (count + 1) * ... * n) AND (count >= 1)}
```

In this case, P is

```
{(fact = (count + 1) * ... * n) AND (count >= 1)}
```

It is clear that I and B implies this P, so by the rule of consequence,

{I and B} S {I}

is true. Finally, the last test of I is

I and (not B) → Q

For our example, this is

```
((fact = (count +1) * ... * n) AND (count >= 0) ) AND (count = 0)) →  fact = n!
```

This is clearly true, for when count = 0, the first part is precisely the definition of factorial. So, our choice of I meets the requirements for a loop invariant. Now we can use our P (which is the same as I) from the while as the postcondition on the second assignment of the program

```
{P} fact = 1 {(fact = (count + 1) * ... * n) AND (count >= 0) }
```

which yields for P

```
(1 = (count + 1) * ... * n) AND (count >= 0) )
```

Using this as the postcondition for the first assignment in the code

```
{P} count = n {(1 = (count + 1) * ... * n) AND (count >= 0)) }
```

produces for P

```
{(n+l)*...*n=1 AND (n >= 0) >
```

The left operand of the AND operator is true (because 1 = l) and the right operand is exactly the precondition of the whole code segment, {n >= 0}. Therefore, the program has been proven to be correct.

Evaluation

To define the semantics of a complete programming language using the axiomatic method we must define an axiom or an inference rule for each statement type in the language. Defining axioms and inference rules for some of the statements of programming languages has proven to be a difficult task. An obvious solution to this problem is to design the language with the axiomatic method in mind, so that only statements for which axioms or inference rules can be written are included. Unfortunately, such a language would be quite small and simple, given the state of the science of axiomatic semantics.

Axiomatic semantics is a powerful tool for research into program correctness proofs, and it provides an excellent framework in which to reason about programs, both during their construction and later. Its usefulness in describing the meaning of programming languages to either language users or compiler writers is, however, highly limited.

Esercizi

1. What if the factorial had been defined as follows:

```
{n >= 0}
count = 0;
fact = 1;
while count < n do
  fact = fact * count;
  count = count + 1;
end
{fact = n!}
```

How would you define I in this case? How would you modify the proof?

2. Aximatic semantics and recursive definitions. Assume that you have a function fact(n) and you want to prove that:

{n>=0} x = fact(n) {x=n!}

fact(n) è definita ricorsivamente:

if n = 0 fact(n) = 1 else fact(n) = fact(n-1)

Come procederesti nella dimostrazione?

3. (esercizio fatto in classe)

Dimostra che il seguente programma è corretto:

```
{x = V_x and y = V_y}
temp = x;
x=y;
y=temp;
{x = V_y and y = V_x}
```

4. Dimostra che il seguente programma è corretto:

```
{n>0}
count = n;
sum = 0;
while count <> 0 do
    sum = sum + count;
    count = count - 1;
end
{sum = 1 +2 + ... + n = Σ_{i=1}^{n} i}
```

Soluzione: prendi come invariante {sum = $\Sigma_{i=count+1}^{n} i$ and count ≥ 0}

Index