An Introduction to axiomatic semantics for CS471

Axiomatic semantics were introduced by Tony Hoare and others as a way of defining the semantics of a programming language independently of the syntax and also of any particular way of implementing the language. It concentrates on the idea of program state and what must be true about the values of the variables that make up the state. Since program state only changes through assignment each statement about program state refers to the states between assignments. A logical language of assertions is used to make statements about program states. Essentially this language, in its simplest form, allows expression of equality and inequality of any arithmetic expression, and logical combinations of these. Variables are all considered to contain integers. Typical assertions are:

x = 0 (the value of variable is zero in this state)
x > 0 & y <= 0 (x is greater than zero and y is less than or equal to zero in this state)
x = y * 2 + 1 (the value of x equals one more than the value of y multiplied by two in this state)

Not only do the axioms below define the semantics of the language, they trail of assertions through the program can be used as a *proof* of program correctness. Essentially by using a standard decomposition of a program into individual statements we can ensure that every program state follows correctly from the previous one. Building up the individual statements into the whole program allows us to state that the program produces a final state from an initial state correctly. That is, that the program as a whole is correct, relative to these two assertions. The initial and final assertions are called the *specification* of the program.

The axioms

Although axiomatic semantics can be defined for any language, taking a very simple imperative language is the easiest. This language only has assignment, if-then-else and a while loop. Each of these forms is defined semantically by an axiom that is unprovable but can be justified either intuitively or by more sophisticated techniques in mathematical logic (that we will not consider).

The first axiom is the basic one of assignment:

1. assignment $\{p[x \rightarrow E]\}$ x = E {p}

The postcondition is any logical assertion p, between the braces. The precondition (actually the *weakest* precondition since there is an infinite number of assertions which are true about the state before an assignment) is p with x substituted by E. Note it is a straight text substitution, not a value substitution).

For instance take the assignment x = y*2 + 1 with the postcondition {x>10}. The current precondition is give in:

$\{y*2 + 1 > 10\}\ x = y*2 + 1\ \{x > 10\}$

The rule for the conditional form needs two parts, since both branches of the form must be included. The general form is if B then S_1 else S_2, where B is any Boolean expression and S_1 and S_2 are statements.

2. composition
$$\frac{\{p\}\ S\ \{q\},\ \{q\}\ S_2\ \{r\}}{\{p\}\ S_1\ ;\ S_2\ \{r\}}$$
This rule expresses the fact that a sequence of two statements has an intermediate assertion between the two statements that is the postcondition for the first one and the precondition for the second.

3. conditional
$$\frac{\{p\ \&\ B\}S_1\{q\},\ \{p\ \&\ {\sim}B\}S_2\{q\}}{\{p\}\ \text{if } B \text{ then } S_1 \text{ else } S_2\ \{q\}}$$

Note that the axiom includes components for both branches and that this is the structured program type of if-then-else that results in the same state so that we can make one assertion q about that final state.

4. loop
$$\frac{\{I\ \&\ B\}\ S\ \{I\}}{\{I\}\ \text{while } B \text{ do } S\ \{I\ \&\ {\sim}B\}}$$

The loop has the most problematic axiom since a) the statement S is repeated until the loop condition B is false, and b) it may never terminate. This axiom takes care of these problems by insisting that an assertion I is true before the loop starts, after every execution of the body S, and after the loop terminates. This is an *invariant* assertion. Loop termination is handled by making the loop condition part of the precondition (above the line in the rule) and its negation after the loop finishes. However, this, *by itself*, does not guarantee that the loop terminates, merely that *if* the loop terminates then it will terminate with ~B being true. Hoare's methods therefore define *partial* correctness. *Complete* correctness that ensures that loops will terminate needs additional techniques that we will not discuss here.

5. dummy $\{p\}\ d\ \{p\}$
where d is the dummy statement. This is needed in the case where an if-then-else really has no else branch and is written if B then S else d.

The remaining axiom helps with a proof of program correctness. They help with "gluing" assertions together where they are not exactly the same.

6. consequence
$$\frac{p \rightarrow p',\ \{p'\}\ S\ \{q'\},\ q' \rightarrow q}{\{p\}\ S\ \{q\}}$$

This rule expresses the fact that a precondition that is different from the weakest precondition must logically *imply* the weakest precondition for progress to be made in the proof. On the other side, the postcondition must imply the precondition for the next statement if it is different from it.

A Sample Proof

We will illustrate the use of the axioms to prove a simple program correct. The program calculates the product of two integers by using repeated addition. x and y are the two numbers, and the strategy is to add the value of y into a product variable, p, x times. Since x is used as a counter, it must be greater than or equal to zero. This is the precondition part of the specification. The postcondition is simply p = x * y. The program, with specification added is:

```
{x >= 0}
p = 0
n = x
while n > 0 do
  p = p + y
  n = n – 1
end
{p = x * y}
```

The proof of correctness we will present is informal, with may steps where we rely on algebraic manipulations of logical expressions that strictly speaking should also be proved. These proofs are hard, and will not be shown here.

The first step is to find the invariant of the loop. Unless this is accurate, the rest of the proof will be difficult or impossible. In this case, we can find the invariant by following the values of p and n round the loop. It is fairly easy to see that after every execution of the loop body the value of p is y * (x – n). Even though this is an invariant we will not be able to proved the program without including the termination condition in the invariant as well. Since n starts out positive and goes down to zero, this portion is $n \geq 0$. The invariant is thus:

$I \equiv p = y * (x - n) \ \& \ n \geq 0$

This is true before the loop starts, after every execution of the body and after the loop finishes. In order to prove that the loop body preserves the invariant we need to apply the assignment axiom twice for the two statements in the body. We do this by assuming the postcondition (the invariant and "pushing it back" through each assignment. For the second one this is:

$\{p = y * (x - (n - 1)) \ \& \ n - 1 \geq 0\} \ n = n - 1 \ \{ p = y * (x - n) \ \& \ n \geq 0\}$

Pushing this back one more gives:

$\{p + y = y * (x – (n – 1)) \ \& \ n – 1 \geq 0\}$ p = p + y $\{ p = y * (x – (n – 1)) \ \& \ n – 1 \geq 0\}$

By manipulating this precondition, we have:

$\{p = y * (x – n) \ \& \ n > 0\}$

We can see that this is equivalent to I & B since $n > 0$ implies $n \geq 0$. Using the axiom for composition we have:

$\{I \ \& \ n > 0\}$ p = p + y; n = n –1 $\{I\}$

We thus have the prerequisite for applying the loop axiom, which allows us to write:

$\{I\}$ while n > 0 do p + p + y; n = n –1 end $\{I \ \& \ \sim(n > 0)\}$

Applying the same technique with the initialization statement before the loop we can write:

$\{p = y * (x – x) \ \& \ x \geq 0\}$ n = x $\{p = y *(x – n) \ \& \ n \geq 0\}$

and then

$\{0 = y *(x – x) \ \& \ x \geq 0\}$ p = 0 $\{p = y * (x – x) \ \& \ x \geq 0\}$

This precondition reduces to $x \geq 0$ since the first part is simply true $(0 = 0)$. Again applying composition we have:

$\{x \geq 0\}$ p = 0; n = x $\{I\}$

Applying composition to this and the loop gives:

$\{x \geq 0\}$ P $\{I \ \& \ \sim(n > 0\}$, where P is the whole program.

The final step is to show that I & ~(n >0) implies $p = x * y$ to “glue” this assertion to the final assertion from the specification. Since the invariant contains $n \geq 0$ and $\sim(n > 0)$ is equivalent to $n \leq 0$ this must mean that $n = 0$ (*proving* this is hard). Since the first part of the invariant with $n = 0$ is equivalent to $p = x * y$, we have the desired glue, and the proof is complete.