CS403: Programming Languages

Axiomatic Semantics (part 1)

Axiomatic semantics

Axiomatic semantics can be thought of as a formal treatment of the *preconditions and postconditions (or invariants)* However, this system is intended for proving a program correct rather than checking invariants at run time in order to detect a program running amok. In general, given a set of preconditions, a program, and a set of postconditions, a proof of correctness consists of showing that given the set of preconditions, executing the program code will not violate any of the postconditions. One of the simplest meaningful proofs one can perform deals with a basic assignment statement.

Assignments

Consider this program, complete with precondition and postcondition:

```
<<requires: x > 1>> x = x - 1 <<ensures: x > 0>>
```

Note that a precondition *requires* certain facts to be true at the beginning while a postcondition *ensures* that other facts are true after the interstitial code has be executed.

In the example above, the pre- and post-conditions are enclosed in angle brackets and the program code resides between the conditions. From inspection, it can be deduced that this simple program is correct. The formal method for proving correctness is to "push back" the postcondition through the statement to see if the precondition can be generated. To push back a condition, one replaces occurrences of the LHS of the program statement in the condition with the RHS. For example, `x > 0` becomes `x - 1 > 0` which is equivalent to `x > 1` by algebraic manipulation. Note that we have recovered the precondition, so we have proved our program correct.

In the above example, the pushed-backed post-condition was algebraically equivalent to the precondition. Not all correct programs will have this property. Consider a similar program

```
<<requires: x = 100>> x = x - 1 <<ensures: x > 0>>
```

Surely this program is correct, yet the precondition does not match the result of pushing the postcondition back through the program code. All is not lost, however, since the pushed back condition does not need to match the precondition exactly. The precondition need only *imply* the pushed back postcondition. In this case, `x = 100` implies `x > 0`. The formal notation of the proof rule for simple assignment statements is:

$$\texttt{<<requires: P>> x = e <<ensures: Q>>} : P \rightarrow Q\,x \setminus e$$

where P is the precondition, $x = e$ is the assignment statement in question, Q is the postcondition, $Q \setminus e$ is the postcondition with each occurrence of x replaced by the expression e, and $\rightarrow$ is a logical operator which means *implies*.

In general, we can prove the correctness of any statement S. The proof rule for statements is:

$$\texttt{<<requires: P>> S <<ensures: Q>>} : P \rightarrow Q \otimes S$$

where $Q \otimes S$ is read as "Q pushed back through S".

Conditionals

Proving conditionals correct is a bit trickier. Consider this program:

```
<<requires: true>>                 // P, the precondition
if (x >= y)                        // E, the test expression
    max = x;                       // S, the if-true branch
else
    max = y;                       // T, the if-false branch
<<ensures: (max == x && x >= y)
        || (max == y && x < y)>>   // Q, the postcondition
```

The proof rule for conditionals whose test has no side effects is:

`<<requires: P>> if (E) S else T <<ensures: Q>>` : $P \wedge E \rightarrow Q \otimes S$ *and* $P \wedge \neg E \rightarrow Q \otimes T$

where *and* separates the parts of the rule that must be proved individually and $\otimes$ is the "push-back" operator and $\neg$ is logical negation. In the example, our proof starts by showing the **if-true** branch is correct. We begin by letting:

$Z = Q \otimes S$

which yields:

$Z = (x \equiv x \wedge x \geq y) \vee (x \equiv y \wedge x < y)$

or:

$Z = x \geq y \vee (x \equiv y \wedge x < y)$

or (since $x \equiv y \wedge x < y$ is a contradiction),

$Z = x \geq y$

To complete this half of the proof, we need to determine, $P \wedge E$, which is:

$true \wedge x \geq y$
$x \geq y$

Since:

$x \geq y \rightarrow Z$
$x \geq y \rightarrow x \geq y$ #obvious

the first half of the proof is complete. The second half is similar; we prove the **if-false** branch:

$Z = Q \otimes T$

which yields:

$Z = (y \equiv x \wedge x \geq y) \vee (y \equiv y \wedge x < y)$

or:

$Z = (y \equiv x \wedge x \geq y) \vee x < y$

or (since $x \equiv y \wedge x \geq y$ can only be true if $x \equiv y$):

$Z = y \equiv x \vee x < y$

To complete this half of the proof, we need to determine, $P \wedge \neg E$, which is $x < y$. Since Z is equivalent to $x \leq y$, it is easy to see that:

$x < y \rightarrow Z$
$x < y \rightarrow x \leq y$ `//LHS is consistent/more restrictive`

The proof is complete (whew!).

The weakest precondition

Often, we wish to have the *weakest* precondition in our code that guarantees the code is correct. There is a way for us to determine if the given precondition is the weakest possible; the weakest precondition of a conditional is given by:

$((Q \otimes S) \wedge E) \vee ((Q \otimes T) \wedge \neg E)$

which is:

$(x \leq y \wedge x \leq y) \vee ((y \equiv x \vee x < y) \wedge x < y)$

This simplifies to:

$x \leq y \vee ((y \equiv x \vee x < y) \wedge x < y)$
$x \leq y \vee x < y$
true

Thus, the given precondition is the weakest, as one would suspect.

Generating the weakest precondition for assignments is even simpler. One merely pushes the postcondition back through the assignment to generate the weakest precondition.