CS403: Programming Languages

Axiomatic Semantics (part 2)

Loops

Previously, we've seen example proofs for assignment and conditionals. We continue in this vein by looking at proofs concerning loops. The proof rule for while loops, assuming that the loop terminates, is:

`<<requires: P>> while E do S <<ensures: Q>>`	:	$P \rightarrow I$
		and `<<requires: I>> E <<ensures: I>>`
		and `<<requires: (I AND E)>> S <<ensures: I>>`
		and $(I \wedge \neg E) \rightarrow Q$

The process for proving a while loop correct is to tackle each part of the proof rule in turn. There are four steps, once the proper loop invariant I has been found:

1. show $P \rightarrow I$, meaning the invariant follows from the precondition
2. show `<<requires: I>> E <<ensures: I>>`, meaning the invariant survives the loop test
3. show `<<requires: I` $\wedge$ `E>> S <<ensures: I>>`, meaning the invariant survives the loop body
4. show $(I \wedge \neg E) \rightarrow Q$, meaning the postcondition follows from the loop exiting and the invariant

Let's prove the correctness of the following program:

```
<<requires: x >= 0 AND y == 0>>
while (x > 0)
    {
    x = x - 1;
    y = y + 1;
    }
<<ensures: x == 0 AND y == #x>>
```

The notation `#x` in the code denotes the original value of *x*. To prepare for the proof, we identify all the variables in the proof rule (plus the invariant):

P:	$x \geq 0 \wedge y \equiv 0$
E:	`x > 0`
S:	`{x = x - 1, y = y + 1}`
Q:	$x \equiv 0 \wedge y \equiv \#x$
I:	$x \geq 0 \wedge x + y \equiv \#x$

Note that for the invariant, the sum of x and y is a constant. For the *first* step in the proof, we need to show that the implication $P \rightarrow I$ is true. This is because, at the point of the precondition, $x \equiv \#x$ and $y \equiv 0$. Therefore, $x + y \equiv \#x$. For the *second* step, we need to show that $I \rightarrow (I \otimes E)$.

$Z = I \otimes E$
$Z = (x \geq 0 \wedge x + y \equiv \#x) \otimes$ `{x > 0}` $Z = (x \geq 0 \wedge x + y \equiv \#x)$

Does $I \rightarrow Z$? Yes. Generally, if the loop condition does not cause a change of state, it is sufficient to say "loop condition has no side-effects" for this part of the proof.

For the *third* step, we push I back through S to see if $(I \wedge E)$ implies the result:

$Z = I \otimes S$
$Z = (x \geq 0 \wedge x + y \equiv \#x) \otimes$ `{x = x - 1; y = y + 1;}` $Z = ((x \geq 0 \wedge x + y \equiv \#x) \otimes$ `{y = y + 1}`$) \otimes$ `{x = x - 1}`
$Z = (x \geq 0 \wedge x + y + 1 \equiv \#x) \otimes$ `{x = x - 1}` $Z = x - 1 \geq 0 \wedge x - 1 + y + 1 \equiv \#x$
$Z = x \geq 1 \wedge x + y \equiv \#x$

Given that $I \wedge E$ is:

$x \geq 0 \wedge x + y \equiv \#x \wedge x > 0$

which is equivalent to:

$x > 0 \wedge x + y \equiv \#x$

is it true that $(I \wedge E) \rightarrow Z$, or:

$(x > 0 \wedge x + y \equiv \#x) \rightarrow (x \geq 1 \wedge x + y \equiv \#x)$

Since $x > 0$ is equivalent to $x \geq 1$, we see that it is. For the *fourth* and final step, we need to show that $I \wedge \neg E \rightarrow Q$. We have $x \geq 0$ (from I) and $x \leq 0$ (from the negation of E), so it must be the case that $x \equiv 0$. That, with $x + y \equiv \#x$ (also from I), we deduce that $y \equiv \#x$, which is indeed Q. The proof is complete (even more whew!).

The above proof is known as a *partial correctness* proof, because we assumed the loop terminated. For a full correctness proof, one would need to prove that the loop terminates.

Finding invariants in loops

Every proper loop has an invariant. Consider this loop, which halves even numbers:

```
<<requires: x % 2 == 0 AND y == 0 AND x >= 0>>

//invariant is true here
while (x > 0)
    {
    //and here
    x = x - 2;
    y = y + 1;
    //and here
    }
//and finally here

<<ensures: x == 0 AND y = #x / 2>>
```

The comments indicate where the invariant must be true. This implies that the invariant need not be true in the middle of the loop body.
A loop invariant describes some computational value that is begin preserved at certain points in the loop. Here are some of the invariants for this particular loop:

```
x % 2 == 0
x + 2 * y == #x
x >= 0
y >= 0
```

Which particular invariants are necessary for proving partial correctness of this loop? One could use the conjunction of all of these:

```
x % 2 == 0 AND x + 2 * y == #x AND x >= 0 AND y >= 0
```

This should always work, although some clauses are likely irrelevant or redundant. The art of invariant finding is determining minimal invariants for the task.
Here is another example:

```
<<requires: a >= 0 AND b > 0 AND t == 1>>
while (b > 0)
    {
    if (b % 2 == 0)
        {
        a = a * a;
        b = b / 2;
        }
    else
        {
        t = t * a;
        b = b - 1;
        }
    }
<<ensures: b == 0 AND t == #a ^ #b>>
```

The invariants for this loop include:

```
 t * a ^ b = a# ^ b#
 a >= 0
 b >= 0
 t >= 0
```

It takes some calculation, but it can be shown that these invariants hold at all the right places. The trickiest of these is the first. Obviously it holds just before the loop is entered since *t* is 1 and *a* and *b* are *a#* and *b<sharp/>*, respectively. Since the loop test does not cause a change of state, the invariant is true at the top of the loop body as well. To show that the invariant is true at the bottom of the loop, one must look at the two cases, when *b* is even and when *b* is odd.

For the first case, we can see that *t* remains unchanged, but *a* is squared and *b* is halved. So is it true that:

```
t * a ^ b == t * (a * a) ^ (b / 2) ?
```

We can rewrite the right-hand side as:

```
 t * (a ^ 2) ^ (b / 2)
```

By the one of the laws of exponents, this simplifies to:

```
t * a ^ (2 * b / 2)
```

Canceling the twos yields are original invariant:

```
t * a ^ b
```

For the second case, when *b* is odd, we ask:

```
t * a ^ b == (t * a) * a ^ (b - 1) ?
```

We can rewrite the right-hand side as:

```
t * a ^ 1 * a ^ (b - 1)
```

By another law of exponents, this simplifies to:

```
t * a ^ (1 + b - 1)
```

Canceling the ones yields:

```
t * a ^ b
```

as desired. Finally, the invariant must be true immediately after the loop exits since, as before, the loop condition does not cause a change of state.
Invariant quantities need not be mathematical in nature; they can be logical as well. For example, consider this loop:

```
<<requires: length(a) > 0 AND i == 1 AND max == a[0]>>

while (i < n)
    {
    if (a[i] > max)
        max = a[i];
    else
        max = max;
    i = i + 1;
    }

<<ensures: i == n AND max == MAX(a,0,n)>>
```

where `MAX` is a mythical function that returns the true maximum in array *a* in the range 0 to *n* (exclusive).
Invariant expressions for this loop include:

```
max == MAX(a,0,i)
i <= n
i > 0
```

As before, let's investigate the most complicated invariant expression, the first one listed. We see that it is true just before the loop is entered since *max* is equal to the only value in the range 0..*i* as *i* is equal to 1. Because the loop test causes no change of state, the invariant is true at the top of the loop. Note that the loop body extends the range by one and that the newly allowed value in the range (located at `a[i-1]` when at the bottom of the loop) has been checked to see if it the new maximum value. If it is, *max* is updated; if not, *max* stays the same. Therefore, *max* remains equal to `MAX(a,0,i)` when at the bottom of the loop. Finally, the invariant is true when the loop exits since the loop condition does not cause a change of state.