Verifying Algorithms

We continue the discussion how to validate correctness statements for the central programming language constructs. Next we consider conditional statements and loops, see Sections 2.8 and 2.9 in the textbook.

The inference rule for if-statements is as follows:

$$\frac{\texttt{P\&\&B\{C}_0\texttt{\}Q} \quad \texttt{P\&\&!B\{C}_1\texttt{\}Q}}{\texttt{P\{if(B)C}_0 \text{ else } \texttt{C}_1\texttt{\}Q}}$$

Here we need the side condition that the evaluation of `P` should not change the state.

Example. Verify the validity of the following correctness statement:

```
ASSERT( y==y0 && z==z0)
if (y <= z) { y=z+1; z=z+2; } else { z=y+2; y=y+1;}
ASSERT( max( y0, z0) < y < z )
```

The inference rule of if-statements without an else-part is obtained by choosing C_1 to be an empty code fragment.

$$\frac{\texttt{P\&\&B\{C}_0\texttt{\}Q} \quad \texttt{P\&\&!B\{\}Q}}{\texttt{P\{if(B)C}_0\texttt{ \}Q}}$$

How can we make the 2nd premise of the inference rule more understandable?

Example. Verify

```
ASSERT( z <= y )
if ( x > z || y > x ) { w = z-1; x = y; }
ASSERT( w <= z <= y <= x )
```

Handling loops is more difficult. At first sight it could seem that we have to treat each iteration separately and use large numbers of intermediate assertions. A single well-chosen assertion is normally sufficient.

The inference rule for while statements is:

$$\frac{\texttt{I\&\&B}\ \{\texttt{C}\}\ \texttt{I}}{\texttt{I}\ \{\texttt{while(B)C}\}\ \texttt{I\&\&!B}}$$

Above assertion I is called the loop invariant and it occurs four times in the inference rule. Below we discuss justification for this rule, for more details see Section 2.9.

```
/* other part of the program */
                                  <-- (1)
while(B) {
              <-- (2)
           C
              <-- (3)
}
                           <-- (4)
/* rest of the program */
```

(1) Here the program is in a state satisfying the pre-condition `I`.

(2) Here the program is in a state satisfying the assertion `I&&B`.

(3) Here `B` may or may not be true.

(4) Here the program is in a state satisfying `I&&!B` (assuming the code C preserves the assertion `I`.

Hence a proof tableau schema can be written as:

```
/* other part of the program */


ASSERT(I)
while(B) {
          ASSERT(I && B)


          C


          ASSERT(I)
}
ASSERT(I && !B)


/* other part of the program */
```

Similarly as in the inference rule for conditional statements, above we need to assume that evaluating the expression `B` does not change the state.

The essential question in loop verification is choosing a suitable loop invariant. Roughly speaking, the invariant has to satisfy the following:

- the invariant is true before entering the loop,
- the loop body preserves the invariant, and,
- the invariant together with the negation of the while-condition should imply the post-condition.

Example. Choose a suitable loop invariant and complete the proof tableau for the following correctness statement:

```
ASSERT(true)
i = 0;
j = 100;
while( i <= 100 ) {
            i = i+1;
            j = j-1;
}
ASSERT( i == 101 && j == -1)
```

In class we will go through further examples of choosing a loop invariant and completing a proof tableau for program fragments involving loops.

Termination

Above we have discussed correctness statements of the form

`P { while(B) C } Q`

However, the corresponding inference rule establishes only that if the program is started in a state satifying `P`, then always when it terminates the state satisfies assertion `Q`. That is, we cannot exclude the possibility of non-termination and thus the inference rule establishes only *partial correctness.*

In order to prove *total correctness,* we need to prove that the execution of the loop always terminates. A typical way to do this is to use a suitable "variant" integer expression that is

- strictly decreased (respectively, increased) by each iteration of the loop, and
- must remain greater than a given lower bound (respectively, less than some upper bound).

Example. The following code to compute powers is partially correct. How should the pre-condition and the invariant be modified in order to guarantee total correctness.

```
ASSERT( n >= 0 )
i = 0; y = 1;
while ( i != n )
INVAR( i >= 0 && y == power(x, i))
{
  y = y*x*x;
  i = i+2;
}
ASSERT( y == power(x, n))
```

Uncomputability of termination. There is *no general algorithm* to determine whether or not a given (while) loop terminates. We will come back to this question at the end of the course when discussing computability. It may be interesting to note that there exist very innocent looking while-loops for which *no one knows* whether or not the loop terminates for all variable values, for example, see Program 2.7 on page 55 in the text.