Notes on Program Verification

This document provides a quick reference to Floyd-Hoare logic for program verification. The document lists only a few rules with some examples. It was created for the sole purpose of teaching the fundamentals of program verification and is meant for the students in C-S 743 - Software Verification and Validation. More on Floyd-Hoare Logic can be found in several books on program verification and axiomatic semantics. This document uses the syntax and examples taken from the notes published by Professor Mike Gordon at the University of Cambridge, UK.

`http://www.cl.cam.ac.uk/~mjcg/Teaching/2011/Hoare/Notes/Notes.pdf`

Assignment rule:

$$\{A[E/V]\} \qquad V = E \qquad \{A\}$$

where A is the assignment statement that tries to assign the value of the expression E to the variable V. The rule says that "the value of V after the assignment must be equal to the evaluated value of the expression E". Formally, if the statement A is true before the assignment, then the statement obtained by substituting V by the evaluated value of E must be true after the statement execution. The precondition ensures that the validity of substitution must be checked before the assignment. One such validation constraint is type checking.

Example 1: $\{x == 0 \wedge y == 10\}\ x = y;\ \ \{x == 10 \wedge y == 10\}$

In this example, it asserts that the value of x is changed by the assignment statement while the value of y remains the same. Types of x and y are the same as indicated by the pre-condition.

Is the following assignment statement true?
$\{y == 10\}\ x = y;\ \ \{x == 10 \wedge y == 10\}$
Explain.

Example 2: $\{x + 1 == n + 1\}\ x = x + 1;\ \ \{x == n + 1\}$

In this example, the precondition helps evaluating the value of x before the assignment and the postcondition asserts what its value will be after the statement execution.

Sequencing rule

$$\frac{\{P\}\ S_1\ \{Q\},\qquad \{Q\}\ S_2\ \{R\}}{\{P\}\ S_1;\ S_2\ \{R\}}$$

Informally, the sequence rule describes that the execution of two consecutive statements can be combined into one composite statement as long as the postcondition of the first statement becomes or implies the precondition of the second statement.

Example 6: Swapping two values.
Given the following assignment statements

$$\{x == 10 \wedge y == 5\}\ r = x\ \{x == 10 \wedge y == 5 \wedge r == 10\},$$
$$\{x == 10 \wedge y == 5 \wedge r == 10\}\ x = y\ \{x == 5 \wedge y == 5 \wedge r == 10\},$$
$$\{x == 5 \wedge y == 5 \wedge r == 10\}\ y = r\ \{x == 5 \wedge y == 10 \wedge r == 10\}$$

one can deduce that

$$\{x == 10 \wedge y == 5\}\ r = x;\ x = y;\ y = r\{x == 5 \wedge y == 10 \wedge r == 10\}$$

Derived Sequencing rule

$$\frac{\begin{array}{ll} & P \Rightarrow P_1, \\ \{P_1\}\ S_1\ \{Q_1\}, & Q_1 \Rightarrow P_2, \\ \{P_2\}\ S_2\ \{Q_2\}, & Q_2 \Rightarrow P_3, \\ \ldots & \\ \{P_n\}\ S_n\ \{Q_n\}, & Q_n \Rightarrow Q \end{array}}{\{P\}\ S_1;\ S_2;\ \ldots;\ S_n\ \{Q\}}$$

The derived sequencing rule is an extension of the sequencing rule and is applicable for a block of statements. In fact, this rule is used to prove that if the individual statements of a program are correct, and their sequencing is correct, then the whole program is correct.

Conditional rules

$$\frac{\{P \wedge C\}\ S\ \{Q\}, \qquad P \wedge \neg\, C \Rightarrow Q}{\{P\}\ \textbf{if}\ C\ \textbf{then}\ S\ \{Q\}}$$

$$\frac{\{P \wedge C\}\ S_1\ \{Q\}, \qquad \{P \wedge \neg\, C\}\ S_2\ \{Q\}}{\{P\}\ \textbf{if}\ C\ \textbf{then}\ S_1\ \textbf{else}\ S_2\ \{Q\}}$$

In the above rules, C refers to the condition in the **if** statement.

While rule

A **while** statement includes a condition C and a body of statements S. In addition, there is an invariant I (a property expressed like a condition) which must be true (i) before entering the while loop, (ii) at the end of each iteration of the while loop and (iii) immediately after the while loop is terminated. A more formal definition of 'while rule' follows:

$$\frac{\{P \wedge I \wedge C\}\ S\ \{I \wedge Q\}}{\{P \wedge I\}\ \textbf{while}\ C\ \textbf{do}\ S\ \{P \wedge I \wedge Q \wedge \neg\, C\}}$$

In the above rule, P indicates a precondition to arrive at the **while** loop, and Q refers to the post-condition that indicates the consequence of executing the while loop.

Example 8: Consider the following while loop. Assume that $r \in \mathbb{N} \wedge y \in \mathbb{N} \wedge q \in \mathbb{N}$.

$q = 0;$
while $y \leq r$ **do**
begin
    $r = r - y;$
    $q = q + 1;$
end

The keywords **begin** and **end** were used instead of the curly parentheses (Pascal style as opposed to C, C++ and Java style) in order to resolve ambiguities in using the parentheses.

For this while loop, one can derive the following:

$C \equiv y \leq r$

$S \equiv r = r - y;\ q = q + 1$

$P \equiv$ *true* since it is not specifically stated

$Q \equiv q * y + r = x$ where x is the initial value of r.

The invariant for this loop is actually the same as the post-condition, namely $x == q * y + r$. Thus,

$P \equiv x == q * y + r$

The details of the proof is given in the exercises section.

Finding a suitable invariant is a challenge in the verification of a loop. Once found, it is used in all the three instances (just before entering into the loop, at the end of each iteration and at the termination of the loop) to verify the correctness of the loop.

Switch rule

For simplicity, we assume that the case selector is always an integer. The following structure for the **switch** statement is assumed.

switch $<<$ *INTEGEREXPRESSION* $>>$
BEGIN
CASE $<<$ $VALUE_1$ $>>$: $<<$ $STATEMENT_1$ $>>$;
break;
CASE $<<$ $VALUE_2$ $>>$: $<<$ $STATEMENT_2$ $>>$;
break;
...
CASE $<<$ $VALUE_n$ $>>$: $<<$ $STATEMENT_n$ $>>$;
break;
END

It is required that the value of the case selector must be within the range $VALUE_1 .. VALUE_n$, both inclusive.

The rule for **switch** statement follows:

$$T \equiv e \in \mathbb{Z} \wedge v_1 \in \mathbb{Z} \wedge v_2 \in \mathbb{Z} \wedge \ldots \wedge v_n \in \mathbb{Z} \wedge$$
$$(e == v_1 \vee e == v_2 \vee \ldots e == v_n) \wedge$$
$$(v_1 \neq v_2 \neq \ldots \neq v_n)$$

$$\{T \wedge P \wedge e == v_1\} S_1 \{Q\}$$
$$\{T \wedge P \wedge e == v_2\} S_2 \{Q\}$$
$$\ldots$$
$$\{T \wedge P \wedge e == v_n\} S_n \{Q\}$$

$\{P\}$ **switch** *e BEGIN*
CASE v_1 : S_1; *break*;
CASE v_2 : S_2; *break*;
...
CASE v_n : S_n; *break*;
END $\{Q\}$

P and Q are the pre- and post-condition respectively for the switch statement.

Exercises

1. Devise an axiom and/or rule of inference for a statement *SKIP* that has no effect. Show that **if** C **then** S is regarded as a simplification of **if** C **then** S **else** *SKIP*; i.e., the rule for one-armed conditional statement is derivable from the rule for the two-armed conditional statement and the axiom/rule for *SKIP*.

   The rule for *SKIP* follows:

   $$\{P\}\ SKIP\ \{P\}$$

   This rule asserts that the precondition is the same as the post-condition meaning that the statement *SKIP* has no effect.

   The rule for **if-then** statement is given as (already given in the notes)

   $$\frac{\{P \wedge C\}\ S\ \{Q\} \qquad\qquad P \wedge \neg\, C \Rightarrow Q}{\{P\}\ \textbf{if}\ C\ \textbf{then}\ S\ \{Q\}}$$

   and the rule for **if-then-else** statement is given as (already given in the notes)

   $$\frac{\{P \wedge C\}\ S_1\ \{Q\} \qquad\qquad \{P \wedge \neg\, C\}\ S_2\ \{Q\}}{\{P\}\ \textbf{if}\ C\ \textbf{then}\ S_1\ \textbf{else}\ S_2\ \{Q\}}$$

   Substituting S for S_1 and *SKIP* for S_2 in the rule for **if-then-else**, we get

   $$\frac{\{P \wedge C\}\ S\ \{Q\} \qquad\qquad \{P \wedge \neg\, C\}\ SKIP\ \{Q\}}{\{P\}\ \textbf{if}\ C\ \textbf{then}\ S\ \textbf{else}\ SKIP\ \{Q\}}$$

   The term $\{P \wedge \neg\, C\}\ SKIP$ can be rewritten as $\{P \wedge \neg\, C\}\ SKIP\{P \wedge \neg\, C\}$ using the rule for *SKIP*. Thus,

   $$\frac{\{P \wedge C\}\ S\ \{Q\} \qquad\qquad \{P \wedge \neg\, C\}\ SKIP\ \{P \wedge \neg\, C\}\ \{Q\}}{\{P\}\ \textbf{if}\ C\ \textbf{then}\ S\ \textbf{else}\ SKIP\ \{Q\}}$$

   This shows that the conditions $P \wedge \neg\, C$ and Q must both be true at the same time without executing any statement. This is possible only if $P \wedge \neg\, C \Rightarrow Q$ which is what stated in the rule for **if-then**. Therefore, it is proved that the one-armed rule for conditions is derived from its two-armed rule and the rule for the *SKIP* statement.

2. Show that, using the rules for conditional statements, the following two code fragments are equivalent:

```
Code fragment 1:

if (n >= '0' && n <= '9') {
    print ("n is s digit");
else
    print ("n is not a digit");

Code fragment 2:
if (n >= 0) {
   if (n <= 9) {
     print ("n is digit");
   else print ("n is not a digit");
else print ("n is not a digit");
```

Let
P - the precondition for the fragment of code (same for both)
Q - the post-condition for the fragment of code (same for both)
of the second **if** block.
C_1 - the condition $n \geq 0$
C_2 - the condition $n \leq 9$
S_1 - the statement `print ("n is digit")`
S_2 - the statement `print ("n is not a digit")`

For the first code fragment,

$$\{P \wedge C_1 \wedge C_2\} S_1 \{Q\} \quad (1)$$

$$\{P \wedge \neg\, (C_1 \wedge C_2)\} S_2 \{Q\} \quad (2)$$

For the second code fragment,
$\{P \wedge C_1\}$ inner block $\{Q\}$
$\{P \wedge \neg\, C_1\} S_2 \{Q\}$
which will be expanded as
$\{P \wedge C_1\}$ $[\{P \wedge C_1 \wedge C_2\} S_1 \{Q\}$
$\{P \wedge C_1 \wedge \neg\, C_2\} S_2 \{Q\}]$ $\{Q\}$
$\{P \wedge \neg\, C_1\} S_2 \{Q\}$
After simplification,

$$\{P \wedge C_1 \wedge C_2\} S_1 \{Q\} \quad (3)$$

$$\{P \wedge C_1 \wedge \neg\, C_2\} S_2 \{Q\} \quad (4)$$

$$\{P \wedge \neg\, C_1\} S_2 \{Q\} \quad (5)$$

Equations (1) and (3) are equivalent and describe the same situation (same precondition, same statement executed resulting in the same post-condition). We will now prove that equation (2) is equivalent to the combination of equations (4) and (5).

Equation (2) has the precondition $\neg\,(C_1 \wedge C_2) \equiv \neg\, C_1 \vee \neg\, C_2$ DeMorgan's law. Since equations (4) and (5) indicate that the same statement is executed and result in the same post-condition, it becomes obvious that one or both of the preconditions can be true in order to execute the statement S_2. That is,

$(C_1 \wedge \neg\, C_2) \vee \neg\, C_1$
$\equiv (C_1 \vee \neg\, C_1) \wedge (\neg\, C_2 \vee \neg\, C_1)$ Distributive law
$\equiv \mathit{true} \wedge (\neg\, C_2 \vee \neg\, C_1)$
$\equiv (\neg\, C_2 \vee \neg\, C_1)$

Therefore, equation (2) describes the same situation as the combined effects of equations (4) and (5). Hence, both code fragments are equivalent.

3. Given $r \in \mathbb{Z}$, $q \in \mathbb{Z}$, and $y \in \mathbb{Z}$, and the code

```
q = 0;
while (y <= r) {
   r = r - y;
   q = q + 1;
}
```

prove that $x == q * y + r$ is an invariant for this while loop where x is equal to the initial value of r.

Let $r == r_0$, $y == y_0$ initially.
At the beginning of the loop (before entering the loop),
$x == q_0 * y_0 + r_0$
$== 0 + r_0$
Therefore, the invariant is true before entering the loop.

At the end of the first iteration, $q == 1$, $r == r_0 - y_0$.
$x == 1 * y_0 + (r_0 - y_0) == r_0$
Therefore, the invariant is true at the end of the first iteration.

This can be generalized to k iterations as well.
At the end of the k^{th} iteration, $q == k$, $r == r_0 - k * y_0$.
$x == k * y_0 + (r_0 - k * y_0) == r_0$
Hence, the invariant is true at the end of the k^{th} iteration as well.

At the exit of the loop, assume that the loop has been executed N times.
Substitute $k == N$ in the previous step; this will prove that the invariant is true at the exit of the loop.

Therefore, $x == q * y + r$ is an invariant for this while loop.