Formal Methods in Software Development
Course 12. Hoare Logic

Mădălina Eraşcu

Content based on the book Leino, K. Rustan M. Program Proofs. MIT Press, 2023; lecture Formal Methods in Software Development by Wolfgang Schreiner, Johannes Kepler University, Linz, Austria

Thanks to Costel Anghel, 3rd year Bachelor student, Applied Informatics

June 5, 2024

Examples

Example

Semantics of `assert`

```
method Triple(x: int) returns (r: int)
{
    var y := 2 * x;
    r := x + y;
    assert r == 10 * x;
    assert r < 5;
    assert false;
}
```

Replace the condition `r<5` with `r==0`, then with any $r \neq 0$.

Example

Weak/strong formulae

```
method Triple(x: int) returns (r: int)
requires x % 2 == 0
ensures r == 3 * x
{
    var y := x / 2;
    r := 6 * y;
}
```

Write two stronger alternatives to the precondition which make the method `Triple` verify.

Definition

Consider the formula $A \Rightarrow B$. We say that A is stronger than B and B is weaker than A.

Examples (cont'd)

Example

Control paths and verification conditions

```
method Triple(x: int) returns (r: int)
{
    if {
        case x < 18 =>
            var a, b := 2 * x, 4 * x;
            r := (a+b) / 2;
        case 0 <= x =>
            var y := 2 * x;
            r := x + y;
        }
    assert r == 3 * x;
}
```

Contents

Program State

The variables that are used at a point in a program are called in scope. The state at a specific point in a program refers to the assignment of values to the variables that are currently within scope at that particular point in the program's execution.

```
method MyMethod(x: int) returns (y: int)
requires x >= 20
ensures y >= 50
(0)
{ (1)
var a, b;
(2)
a := x-1;
(3)
if x < 30 {
    (4)
    b := 51 - x;
    (5)
} else {
    (6)
    b := 31;
    (7)
}
(8)
y := a + b;
(9)
}
(10)
```

Contents

Program State

```
method MyMethod(x: int) returns (y: int)
requires x >= 20
ensures y >= 50
(0)
{ (1)
var a := x - 1;
(2)
var b := 31;
(3)
y := a + b;
(4)
}
(5)
```

We can write an assertion, which, in principle is true, at each program point. Analyzing it in a *forward direction*, we have:

$$
\begin{aligned}
&(0) x \geq 20 \\
&(1) x \geq 20 \\
&(2) x \geq 20 \wedge a = x - 1 \\
&(3) x \geq 20 \wedge a = x - 1 \wedge b = 31 \\
&(4) x \geq 20 \wedge a = x - 1 \wedge b = 31 \wedge y = a + b \\
&(5) y \geq 50
\end{aligned}
$$

Moreover, the postcondition must be shown true, i.e.

$$x \geq 20 \ \wedge \ a = x - 1 \ \wedge \ b = 31 \ \wedge \ y = a + b \quad \Longrightarrow \quad y \geq 50$$

Program State (cont'd)

```
method MyMethod(x: int) returns (y: int)
requires x >= 20
ensures y >= 50
(0)
{ (1)
var a := x - 1;
(2)
var b := 31;
(3)
y := a + b;
(4)
}
(5)
```

The program can be analyzed also in *backward direction*, i.e.

$$
\begin{aligned}
&(5) y \geq 50 \\
&(4) y \geq 50 \\
&(3) a + b \geq 50 \\
&(2) a + 31 \geq 50 \\
&(1) x - 1 + 31 \geq 50 \\
&(0) x \geq 20
\end{aligned}
$$

Moreover the precondition must be shown true, i.e

$$x \geq 20 \quad \Longrightarrow \quad x - 1 + 31 \geq 50$$

Contents

Hoare Calculus

A Hoare triple is a set of logical rules that refers rigorously about the correctness of computer programs. It is typically written as follows:

$$\{\{P\}\}S\{\{Q\}\}$$

P represents the precondition, S stands for the program statements whose behaviour you are specifying, and Q represents the postcondition.

It can read it as *If you start with the precondition P to hold and execute the code provided in S, the program guarantees that the postcondition Q will hold.*

Example

The Hoare triple:

$$\{\{x == 4\}\}x := x * 2\{\{x == 8\}\}$$

expresses the fact that if the program starts in a state where x is 4, and executes $x = x * 2$, it is guaranteed to terminate in a state where x is 8.

Other examples:

- $\{\{x < 18\}\}y := 5\{\{y >= 0\}\}$
- $\{\{x < 18\}\}y := 18 - x\{\{y >= 0\}\}$

Counterexamples:

- $\{\{x < 18\}\}y := x\{\{y >= 0\}\}$
- $\{\{x < 18\}\}y := 2 * (x + 3)\{\{y >= 0\}\}$

Hoare Calculus (cont'd)

Partial correctness interpretation of $\{\{P\}\}c\{\{Q\}\}$: *If c is executed in a state in which P holds, then it terminates in a state in which Q holds unless it aborts or runs forever.*

- Program does not produce wrong result.
- But program also need not produce any result. Abortion and non-termination are not (yet) ruled out.

Total correctness interpretation of $\{\{P\}\}c\{\{Q\}\}$: *If c is executed in a state in which P holds, then it terminates in a state in which Q holds.*

- Program produces the correct result.

We will use the partial correctness interpretation for the moment.

Hoare Calculus (cont'd)

Hoare calculus rules are inference rules with Hoare triples as proof goals.

$$\frac{\{\!\{P1\}\!\}c_1\{\!\{Q_1\}\!\} \quad \ldots \quad \{\!\{Pn\}\!\}c_n\{\!\{Qn\}\!\} \quad VC_1, \ldots, VC_m}{\{\!\{P\}\!\}c\{\!\{Q\}\!\}}$$

Application of a rule to a triple $\{\!\{P\}\!\}c\{\!\{Q\}\!\}$ to be verified

- other triples $\{\!\{P_1\}\!\}c_1\{\!\{Q_1\}\!\}, \ldots, \{\!\{P_n\}\!\}c_n\{\!\{Q_n\}\!\}$ to be verified, and
- formulas VC_1, ..., VC_m (the verification conditions) to be proved.

Given a Hoare triple $\{\!\{P\}\!\}c\{\!\{Q\}\!\}$ as the root of the verification tree:

- The rules are repeatedly applied until the leaves of the tree do not contain any more Hoare triples.
- If all verification conditions in the tree can be proved, the root of the tree represents a valid Hoare triple.

Remark

The Hoare calculus generates verification conditions such that the validity of the conditions implies the validity of the original Hoare triple

Contents

Contents

Special Commands

- `skip` command

$$\{\{P\}\}\ \texttt{skip}\ \{\{P\}\}$$

It says that the precondition and postcondition remain the same after executing the `skip` command

- `abort` command

$$\{\{\mathit{true}\}\}\ \texttt{abort}\ \{\mathit{false}\}$$

It signifies abnormal termination of the program such as dividing by 0 or invalid memory access. Its output should be an error or something trivially false.

Contents

Scalar Assignments

$$\{\!\{Q[e/x]\}\!\}\ x := e\ \{\!\{Q\}\!\}$$

To make sure that Q holds for x after the assignment of e to x, it suffices to make sure that Q holds for e before the assignment.

Example

$\{\!\{x + 3 < 5\}\!\} x := x + 3 \{\!\{x < 5\}\!\}$

Contents

Array Assignment

$$\{\!\{Q[a[i \rightarrowtail e]/a]\}\!\}\ a[i] := e\ \ \{\!\{Q\}\!\}$$

- An array is modelled as a function $a : I \Rightarrow V$.
  - Index set I, value set V.
  - $a[i] = e$ array a contains at index i the value e.
- Term $a[i \mapsto e]$ ("array a updated by assigning value e to index i").
  - A new array that contains at index i the value e.
  - All other elements of the array are the same as in a.

Thus array assignment becomes a special case of scalar assignment.

- Think of "$a[i] := e$" as "$a := a[i \mapsto e]$".

Example

$$\{\!\{a[i \mapsto x][1] > 0\}\!\} a[i] := x \{\!\{a[1] > 0\}\!\}$$

How to reason about $a[i \mapsto e]$?

$$Q[a[i \mapsto e][j]]\ \rightsquigarrow (i = j \Rightarrow Q[e]) \wedge (i \neq j \Rightarrow Q[a[j]])$$

Above, we used array axioms

- $i = j \Rightarrow a[i \mapsto e][j] = e$
- $i \neq j \Rightarrow a[i \mapsto e][j] = a[j]$

Example

$\{\!\{a[i \mapsto x][1] > 0\}\!\}$ $a[i] := x$ $\{\!\{a[1] > 0\}\!\}$ is equivalent to

$\{\!\{(i = 1 \Rightarrow x > 0) \wedge (i \neq 1 \Rightarrow a[1] > 0)\}\!\}$ $a[i] := x$ $\{\!\{a[1] > 0\}\!\}$

Contents

Command Sequences

$$\frac{\{\!\{P\}\!\}\, c_1 \,\{\!\{R\}\!\} \quad \{\!\{R\}\!\}\, c_2 \,\{\!\{Q\}\!\}}{\{\!\{P\}\!\}\, c_1 \,;\, c_2 \,\{\!\{Q\}\!\}}$$

To show that, if P holds before the execution of $c1; c2$, then Q holds afterwards, it suffices to show for some R that

- if P holds before $c1$, that R holds afterwards, and that
- if R holds before $c2$, then Q holds afterwards.

How to find R? Easy in most of the cases.

Example

$$\frac{\{\!\{x + y - 1 > 0\}\!\} y := y - 1 \{\!\{x + y > 0\}\!\} \quad \{\!\{x + y > 0\}\!\} x := x + y \{\!\{x > 0\}\!\}}{\{\!\{x + y - 1 > 0\}\!\} y := y - 1; x := x + y \{\!\{x > 0\}\!\}}$$

Contents

Conditionals

- $$\frac{\{\{P \land b\}\}\ c_1\ \{\{Q\}\} \quad \{\{P \land \neg b\}\}\ c_2\ \{\{Q\}\}}{\{\{P\}\}\ \textbf{if}\ b\ \textbf{then}\ c_1\ \textbf{else}\ c_2\ \{\{Q\}\}}$$
- $$\frac{\{\{P \land b\}\}\ c\ \{\{Q\}\} \quad \{\{P \land \neg b\}\} \implies \{\{Q\}\}}{\{\{P\}\}\ \textbf{if}\ b\ \textbf{then}\ c\ \{\{Q\}\}}$$

This notation allows verification of both branches of an ”if-else” statement satisfy the postcondition Q based on a condition b (true and false).

Example

$$\frac{\{\{x \neq 0 \land x \geq 0\}\} y := x \{\{y > 0\}\} \quad \{\{x \neq 0 \land x \not\geq 0\}\} y := -x \{\{y > 0\}\}}{\{\{x \neq 0\}\}\ \textbf{if}\ x \geq 0\ \textbf{then}\ y := x\ \textbf{else}\ y := -x \{\{y > 0\}\}}$$

Contents

Loops

Loops

$$\frac{\{\!\{I \wedge b\}\!\}\ c\ \{\!\{I\}\!\}}{\{\!\{I\}\!\}\ \textbf{while}\ b\ \textbf{do}\ c\ \{\!\{I \wedge \neg b\}\!\}}$$

If it is the case that

- I holds before the execution of the **while**-loop and
- I also holds after every iteration of the loop body,

then I holds also after the execution of the loop (together with the negation of the loop condition b).

Loops (Generalized)

$$\frac{P \Longrightarrow I \quad \{\!\{I \wedge b\}\!\}\ c\ \{\!\{I\}\!\} \quad (I \wedge \neg b) \Longrightarrow Q}{\{\!\{P\}\!\}\ \textbf{while}\ b\ \textbf{do}\ c\ \{\!\{Q\}\!\}}$$

To show that, if before the execution of a while-loop the property P holds, after its termination the property Q holds, it suffices to show for some property I (the loop invariant) that

- I holds before the loop is executed (i.e. that P implies I),
- if I holds when the loop body is entered (i.e. if also b holds), that after the execution of the loop body I still holds,
- when the loop terminates (i.e. if b does not hold), I implies Q.

Loops (cont'd)

Example

$$I :\iff s = \sum_{j=1}^{i-1} j \wedge 1 \leq i \leq n+1$$

$$\frac{\begin{array}{c}(n \geq 0 \wedge s = 0 \wedge i = 1) \implies I \\ \{\{I \wedge i \leq n\}\}\ s := s + i; i := i + 1\ \{\{I\}\} \\ (I \wedge i \not\leq n) \implies s = \sum_{j=1}^{n} j\end{array}}{\{\{n \geq 0 \wedge s = 0 \wedge i = 1\}\}\textbf{while } i \leq n \textbf{ do } (s := s + i; i := i + 1)\{\{s = \sum_{j=1}^{n} j\}\}}$$

Termination of Loops

$$\frac{I \Rightarrow t \geq 0 \quad \{\!\{I \wedge b \wedge t = N\}\!\}\ c\ \{\!\{I \wedge t < N\}\!\}}{\{\!\{I \wedge t < N\}\!\}\ \textbf{while}\ b\ \textbf{do}\ c\ \{\!\{I \wedge \neg b\}\!\}}$$

$$\frac{P \Longrightarrow I \quad I \Rightarrow t \geq 0 \quad \{\!\{I \wedge b \wedge t = N\}\!\}\ c\ \{\!\{I \wedge t < N\}\!\} \quad (I \wedge \neg b) \Longrightarrow Q}{\{\!\{P\}\!\}\ \textbf{while}\ b\ \textbf{do}\ c\ \{\!\{Q\}\!\}}$$

New interpretation of $\{\!\{P\}\!\}c\{\!\{Q\}\!\}$ which takes into account termination of the loop.

- If execution of c starts in a state where P holds, then execution terminates in a state where Q holds, unless it aborts.
- Non-termination is ruled out.

Termination term/function t (term type-checked to denote an integer).

- Becomes smaller by every iteration of the loop.
- But does not become negative.
- Consequently, the loop must eventually terminate.
- The initial value of t limits the number of loop iterations.