Lectures Notes on Hoare Logic

Construction and Verification of Software
FCT-NOVA
Bernardo Toninho

15 March, 2022

These notes are based on lecture notes by João Seco.

1 Introduction

The axiomatic approach of *Hoare Logic* gives meaning to program fragments by predicting the effects of the execution of each statement in the state of a program. Such effects are expressed by pairs of assertions on the state of a program. Assertions are pure logical expressions, often using first-order logic, where *program variables* may occur. The main construction of this logic is the *Hoare triple*, where P is a pre-condition, S is a program (fragment) and Q is a post-condition:

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

A Hoare triple is understood as: "if P holds in a given program state, then the *terminating* execution of program fragment S produces a state where Q holds". This is a *partial correctness* result since the termination of S is not assured by the triple. *Total correctness* is achieved when termination is also guaranteed.

In the setting of Hoare logic, if a program fragment is executed in a state where the pre-condition holds, then the logic ensures that execution succeeds and that the post-condition holds in the final state, if it exists. If a program fragment is executed starting from an initial state where the pre-condition does not hold, no guarantees can be given about the absence of runtime errors, or about the resulting state (or its existence). For instance, if we consider the program:

$$P \triangleq \textsf{if } x > y \textsf{ then } z := x \textsf{ else } z := y$$

we can prove, using *Hoare Logic*, the following triple:

$$\{\textsf{true}\}\ P\ \{(z = x \vee z = y) \wedge (z \geq x \vee z \geq y)\}$$

This triple denotes that for all possible values of x, y, and z, specified by the pre-condition $\textsf{true}$, the value of z after the execution of program P is equal to the value of x or to the value of y.

In order to provide effective reasoning techniques about composite programs, Hoare logic allows us to compose triples in various ways, just as one composes a larger program from smaller program fragments. Moreover, Hoare logic also provides reasoning principles that allow one to strengthen pre-conditions and/or weaken post-conditions. We now cover some preliminary definitions and concepts that are key to understanding the principles of Hoare logic.

2 Preliminary Material

As stated during lecture, a key contribution of Hoare logic over other approaches at the time was its language-based approach to verification. Hoare logic [Hoa69] provides *axioms* and *inference rules* for the constructs of a small imperative programming language.

Many authors over the years have extended Hoare logic with various features such as procedures, global variables, pointers and even concurrency (ultimately giving rise to separation logic [Rey02]), but the original presentation of Hoare logic is based on a minimal language of pure *expressions* and *commands*.

2.1 Language

The core language syntax which we will use in these notes (following Hoare's original presentation) is given below:

$$\begin{array}{lcll}
A & ::= & & \text{Assertions} \\
 & & \mathsf{true} \mid \mathsf{false} & \\
 & \mid & E_1 = E_2 & \\
 & \mid & A_1 \wedge A_2 \mid A_1 \vee A_2 & \\
 & \mid & A_1 \Rightarrow A_2 & \\
 & \mid & \forall x.A \mid \exists x.A & \\
 & & & \\
E & ::= & & \text{Expressions} \\
 & & num & \text{Integer} \\
 & \mid & x & \text{Variable} \\
 & \mid & E + E \mid ... & \text{Integer operations} \\
 & \mid & E < E \mid ... & \text{Relational operations} \\
 & \mid & E \;\mathsf{and}\; E \mid ... & \text{Boolean operations} \\
 & & & \\
P & ::= & & \text{Programs} \\
 & & \mathsf{skip} & \text{No operation} \\
 & \mid & x := E & \text{Assignment} \\
 & \mid & P; P & \text{Sequential Composition} \\
 & \mid & \mathsf{if}\; E \;\mathsf{then}\; P \;\mathsf{else}\; P & \text{Conditional} \\
 & \mid & \mathsf{while}\; E \;\mathsf{do}\; P & \text{Iteration}
\end{array}$$

The language includes assertions, pure expressions, and statements. Programs are defined over a set of state variables, that can be referred to in assertions. Expressions are pure, since they cannot change the state of the program. Statements define state change operations and control flow. We will assume that programs and expressions under consideration are well-typed in the expected sense: operations are used with operands of the appropriate type; the expression E in a conditional or an iteration statement evaluates to a boolean value, etc. We will not develop the type system for the language formally.

2.2 Axioms and Inference Rules

Hoare logic is supported by a system of inference rules where each language construct is characterized by a single inference rule or axiom. Some inference rules are *structural* in the sense that they do not characterize any specific language construct, but rather allow for the manipulation of the pre- and/or post- condition of an arbitrary Hoare triple.

In the remainder of these notes, we make no distinction between an axiom and an inference rule: an axiom is just an inference rule with an empty premise. Logical assertions are subject to the "usual" classical first-order logic reasoning principles. We refrain from presenting a full system of inference rules for classical first-order logic. Inference rules for classical propositional logic are given below, for reference:

$$\frac{\begin{array}{c}A\\ \vdots\\ B\end{array}}{A \Rightarrow B}\ (\Rightarrow I) \qquad \frac{A \quad B}{A \wedge B}\ (\wedge I) \qquad \frac{A}{A \vee B}\ (\vee I_1) \qquad \frac{B}{A \vee B}\ (\vee I_2) \qquad \frac{\begin{array}{c}A\\ \vdots\\ \bot\end{array}}{\neg A}\ (\neg I)$$

$$\frac{A \quad A \Rightarrow B}{B}\ (\Rightarrow E) \qquad \frac{A \wedge B}{A}\ (\wedge E_1) \qquad \frac{A \wedge B}{B}\ (\wedge E_2) \qquad \frac{A \vee B \quad \begin{array}{c}A\\ \vdots\\ C\end{array} \quad \begin{array}{c}B\\ \vdots\\ C\end{array}}{C}\ (\vee E)$$

$$\frac{\neg A \quad A}{C}\ (\neg E) \qquad \frac{}{A \vee \neg A}\ (LEM)$$

Rules marked with I are dubbed introduction rules and state how to prove a certain proposition. Rule $(\Rightarrow I)$ states that we can prove $A \Rightarrow B$ if we assume A and deduce B. Similarly, we can prove $\neg A$ if we assume A and deduce falsehood. Rules marked with E are dubbed elimination rules and state how to make use of the fact that a certain proposition holds during inference. For instance, rule $(\neg E)$ states that if we can derive $\neg A$ and also A then we can prove an arbitrary proposition C (since *ex falso quodlibet*). Finally, classical logic requires the law of excluded middle, marked as LEM above, stating that for any proposition A, either it or its negation must hold.

3 The Rules of Hoare Logic

We now present the rules of Hoare logic.

Trivial or Empty Statement. For the skip statement/program, the valid triple is the one that registers no effects. So it accepts any assertion as pre- and post-condition. This is captured by the rule:

$$\{A\}\ \mathsf{skip}\ \{A\} \quad (\mathit{Skip})$$

Sequencing. The rule that captures the effect of a sequence of statements is the following

$$\frac{\{A\}\ P\ \{B\} \quad \{B\}\ Q\ \{C\}}{\{A\}\ P; Q\ \{C\}}\ (\mathit{Sequence})$$

In this case, if we can prove valid triples for the statements P and Q, such that the post-condition of the triple for P is *exactly* the pre-condition of the triple for Q, then we can derive a valid triple for the sequence of statements $(P; Q)$, with pre-condition A (the one of the triple for P) and post-condition C (the post-condition of the triple for Q).

It is often the case that while deriving a Hoare triple for a larger program, the post-conditions and pre-conditions of sequenced program statements don't align exactly, but are logically related. We can account for this with the following rule.

Consequence.

$$\frac{A' \Rightarrow A \quad \{A\}\ P\ \{B\} \quad B \Rightarrow B'}{\{A'\}\ P\ \{B'\}} \quad (\mathit{Consequence})$$

Given a derivation of a triple of the form $\{A\}\ P\ \{B\}$, the rule of consequence allows us to *strengthen* the pre-condition A and *weaken* the post-condition B. If we can show that the pre-condition A is implied by A' and that the post-condition B implies B', we are justified in deriving the triple $\{A'\}\ P\ \{B'\}$. Note that since $B \Rightarrow B$ and $A \Rightarrow A$, the following two specialized rules follow from the more general rule above. In proof theory we say that the following two rules are *admissible*:

$$\frac{A' \Rightarrow A \quad \{A\}\ P\ \{B\}}{\{A'\}\ P\ \{B\}} \qquad \frac{\{A\}\ P\ \{B\} \quad B \Rightarrow B'}{\{A\}\ P\ \{B'\}}$$

Conditional The rule that captures the effects of a conditional statement is:

$$\frac{\{A \wedge E\}\ P\ \{B\} \quad \{A \wedge \neg E\}\ Q\ \{B\}}{\{A\}\ \textsf{if}\ E\ \textsf{then}\ P\ \textsf{else}\ Q\ \{B\}} \quad (\mathit{Conditional})$$

The rule follows the general structure that a triple for a compound statement can be derived from valid triples about its constituent parts (sub-statements). So, if one can derive a valid triple for the then branch of an if statement, assuming that the condition E is true, and derive a valid triple for the else branch, assuming that E is false, with both triples sharing a pre-condition A and a post-condition B, Then the full if statement can be validated with the pre-condition A and post-condition B.

Assignment. The effects of an assignment are captured by the following rule:

$$\{A[{}^{E}\!/\!{}_{x}]\}\ x := E\ \{A\} \quad (\mathit{Assign})$$

In this case, an assertion A that may refer to x holds as post-condition of an assignment, if that same assertion holds as pre-condition when referring to the new value of x instead, given by expression E. To illustrate the use of rule (Assign), consider the Hoare triple

$$\{x + 1 > 0\}\ x := x + 1\ \{x > 0\}$$

which can be shown to be valid by direct applicaton of the rule, where we obtain the pre-condition $(x > 0)[{}^{x+1}\!/\!{}_{x}] = x + 1 > 0$. The following Hoare triple can therefore be deduced by rule ($\mathit{Consequence}$), since $x \geq 0 \Rightarrow (x + 1 > 0)$:

$$\{x \geq 0\}\ x := x + 1\ \{x > 0\}$$

In another example, consider two arbitrary predicates on y and z, $P(y)$ and $Q(z)$, and the following triple:

$$\{P(y) \wedge Q(z)\}\ x := y\ ;\ y := z\ ;\ z := x\ \{P(z) \wedge Q(y)\}$$

while it may be not obvious how to derive the triple above, it is intuitively true: since we merely swapped y and z, $P(z)$ and $Q(y)$ must hold after performing the three assignments. Here is a proof of this triple:

$$\frac{\dfrac{\{P(y) \wedge Q(z)\}\ x := y\ \{P(x) \wedge Q(z)\} \quad \{P(x) \wedge Q(z)\}\ y := z\ \{P(x) \wedge Q(y)\}}{\{P(y) \wedge Q(z)\}\ x := y;\ y := z\ \{P(x) \wedge Q(y)\}} \quad \{P(x) \wedge Q(y)\}\ z := x\ \{P(z) \wedge Q(y)\}}{\{P(y) \wedge Q(z)\}\ x := y;\ y := z;\ z := x\ \{P(z) \wedge Q(y)\}}$$

In general, rule ($Assign$) also allows for the deduction

$$\{true\}\ x := E\ \{x = E\}$$

whose proof results from combining the rule ($Assign$) and rule ($Consequence$) in a derivation with the form

$$\frac{(E = E) \Rightarrow true \qquad \{E = E\}\ x := E\ \{x = E\}}{\{true\}\ x := E\ \{x = E\}}$$

Consider the example program fragment P defined above, and the proof sketch below, where assertions are interspersed with the statement structure.

```
{ true }
if (x > y) {
     { (x > y) }
     z := x;
     { (x > y) && (z == x) }
}
else {
     { (x <= y) }
     z := y;
     { (x <= y) && (z == y) }
}
{ (x>y) && (z == x) || (x<=y) && (z == y)}
```

As an exercise, we explore in class the building of this proof, by going over each step using a forward strategy (from the pre-condition to the post-condition).

There is also a Hoare logic rule that covers the use of loops in a program. We will explore that rule and the verification of loops in the next lecture.

These rules define the validity relation for Hoare triples, but are not well-suited for an actual implementation. As any inference system, algorithms have been proposed to implement forward and backward reasoning with this relation. The least intuitive rules to apply algorithmically (in a backward or forward fashion) are the rules for assignment and the rule of consequence, where we must often "guess" intermediate assertions. We will develop more convenient and algorithmic uses of these rules in later lectures.

4 Procedures and Method Invocation

In lecture, we saw an extension of the core imperative programming language that serves as the basis for Hoare logic with method definitions and invocation. There are many ways to extend Hoare logic with procedures. We opt for a simple variant, where method invocation is built into an assignment statement and where method definitions are annotated with their corresponding pre- and post-conditions. The syntax of the language is extended as follows:

$$
\begin{array}{rcll}
E & ::= & \text{Expressions} & \\
 & | & \ldots & \\
 & & & \\
S & ::= & \text{Statements} & \\
 & | & \ldots & \\
 & | & x := m(E_1, \ldots, E_n) & \text{Call + Assignment} \\
 & & & \\
D & ::= & \text{Declarations} & \\
 & | & \texttt{method}\ m(x_1, \ldots, x_n)\ \texttt{returns}\ (r) & \\
 & & \texttt{requires}\ \mathit{Pre}(x_1, \ldots, x_n) & \\
 & & \texttt{ensures}\ \mathit{Post}(x_1, \ldots, x_n, r) & \\
 & & \{S\} & \\
 & & & \\
P & ::= & \text{Program} & \\
 & | & \overline{D} &
\end{array}
$$

A program essentially a sequence of declarations, which we assume are all checked for validity in the sense of Hoare logic. A method declaration for a method m taking arguments x_1 through x_n and returning some value r, specifies a pre-condition of the form $\mathit{Pre}(x_1, \ldots, x_n)$, indicating that the pre-condition may refer to the method *parameters*. Similarly, the post-condition $\mathit{Post}(x_1, \ldots, x_n, r)$ can refer to the method parameters and its return value r.

A valid program P is a sequence of valid method declarations. To validate a method declaration, we use the following rule:

$$
\frac{\{\mathit{Pre}(x_1, \ldots, x_n)\}\ S\ \{\mathit{Post}(x_1, \ldots, x_n, r)\}}{\begin{array}{l}\texttt{method}\ m(x_1, \ldots, x_n)\ \texttt{returns}\ (r) \\ \texttt{requires}\ \mathit{Pre}(x_1, \ldots, x_n) \\ \texttt{ensures}\ \mathit{Post}(x_1, \ldots, x_n, r)\ \{S\}\ \mathit{valid}\end{array}}
$$

To account for method invocation in Hoare logic, we have the following (assuming r does not occur in B):

$$
\frac{\begin{array}{c}\begin{array}{l}\texttt{method}\ m(x_1, \ldots, x_n)\ \texttt{returns}\ (r) \\ \texttt{requires}\ \mathit{Pre}(x_1, \ldots, x_n) \\ \texttt{ensures}\ \mathit{Post}(x_1, \ldots, x_n, r)\ \{S\}\end{array} \in P \\ A \Rightarrow \mathit{Pre}(E_1, \ldots, E_n) \qquad \mathit{Post}(E_1, \ldots, E_n, r) \Rightarrow B[{}^{r}/{}_{x}]\end{array}}{\{A\}\ x := m(E_1, \ldots, E_n)\ \{B\}}\ (\mathit{Assign})
$$

In the rule above, P refers to the whole program. As noted during lecture, the verification of method calls is compositional in the sense that we only *check* that the pre-condition of the method holds (i.e. it follows from A), when appropriately instantiated with the method arguments $E_1, \ldots E_n$, which warrants us to *assume* the post-condition holds when proving B. The substitution on B in the premise may appear strange: note that B (where r is restricted to not occur) will typically refer to the variable x, which stores the return value of the method. However, we reason about this value symbolically in the post-condition of the method, simply referring to it as r. Therefore, when proving the post-condition of the triple B from the post-condition of the method $\mathit{Post}(E_1, \ldots, E_n, r)$, we substitute the occurrence of x in B for r, so that the names line-up correctly.

For instance, consider the following method:

```
method max(x:int,y:int) returns (r:int)
  requires true
  ensures x > y ⟹ r=x ∧ x ≤ y ⟹ r=y
{
  if (x > y) { return  x; } else { return y; }
}
```

We can prove the triple $\{true\}$ $z :=$ `max`$(10, 20)$ $\{z = 20\}$, by checking that $true \Rightarrow true$ and that $(10 > 20 \Rightarrow (r = 10) \wedge 10 \leq 20 \Rightarrow r = 20) \Rightarrow r = 20$.

Note that the verification of methods and method calls in Dafny is actually more involved than what is detailed here. On one hand, the rules above don't really say much about how to check recursive methods. More importantly, Dafny as a programming language includes heap-allocated data (e.g. through objects) and methods can refer to class fields. The model sketched in lecture and in these notes has no account of global variables nor of the heap. In later lectures we will see how we can reason precisely about these features in Dafny, but giving a formal account of these features in Hoare logic is beyond the scope of this course.

5 Exercises

1. Implement, specify and verify the following methods. Try to define the strongest post-condition and the weakest pre-condition possible.

   (a) `method Abs(x:int) returns (y:int)`

   (b) `method Max2(x:int,y:int) returns (w:int)`.

      i. Use a function to specify the post-condition.

      ii. Don't use a function to specify the post-condition.

   (c) `method Max3(x:int,y:int,z:int) returns (w:int)`. Use the function and methods above to specify and implement method `Max3` (As in Exercise 1b, do a version with and without the use of a function in the specification).

   (d) `method CompareTo(x:int, y:int) returns (c:int)`

   Write as many intermediate assertions in the code as you can, to illustrate the proof behind a verified method in Dafny.

2. Implement and verify (i.e. define suitable pre- and post- conditions) a method:

   ```
   method IntervalContains(lowA:nat, highA:nat, lowB:nat, highB:nat)
       returns (bool)
   ```

   That takes four natural numbers, specifying two time intervals A and B during a day. The numbers denote hours. The method tests whether (i.e. returns `true` iff) interval A fully contains interval B.

3. Consider the following Dafny methods:

   ```
   method mystery1(n:nat,m:nat) returns (res:nat)
      ensures true
   {
      if (n=0) {
            return m;
   ```

```
      } else {
         var aux := mystery1 (n-1,m);
         return 1+aux;
      }
   }
method mystery2(n:nat,m:nat) returns (res:nat)
      ensures true
{
      if (n=0) {
         return 0;
      }
      else {
         var aux := mystery2(n-1,m);
         var aux2 := mystery1(m,aux);
         return aux2;
      }
}
```

Change the post-condition of both methods so that it is the *strongest* possible.

4. Consider the following specification of a `min` method:

```
method min2(a:int, b:int) returns (m:int)
        ensures (m=a) ⟺ (a≤b)
{
        if (a≥b) {m:=b;} else {m:=a;}
        return m;
    }
```

Produce a code snippet that uses `min2` according to its specification but contains an assertion on the result of the method that fails to verify, despite being *a true statement*. Fix the specification of `min2` to make the assertion go through.

5. Implement the simplest method possible that satisfies the given specification:

(a)
```
method m1(x:int,y:int) returns (z:int)
requires 0 < x < y
ensures z ≥ 0 ∧ z < y ∧ z ≠ x
```

(b)
```
method m2(x:nat) returns (y:int)
requires x ≤ -1
ensures y > x ∧ y < x
```

(c)
```
method m3(x:int,y:int) returns (z:bool)
ensures z ⇒ x=y
```

(d)
```
method m4(x:int,y:int) returns (z:bool)
ensures z ⟺ x=y
```

6. [⋆ ⋆ ⋆] Consider the following Dafny method:

```
function square(n:int) : int { n * n }

method Q(n:nat) returns (r:int)
  ensures true
{
  var count := 0;
  var sum := 1;
  while (sum ≤ n)
    invariant 0 ≤ count
    invariant square(count) ≤ n
    invariant sum = square(count+1)
  {
    count := count + 1;
    sum := sum + 2 * count + 1;
  }
  return count;
}
```

What is the strongest post condition for method `Q`?

References

[Hoa69] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, 1969.

[Rey02] John C. Reynolds. Separation logic: A logic for shared mutable data structures. In *LICS*, pages 55–74. IEEE Computer Society, 2002.