Static Analysis of Programs with Hoare Logic
15-453 Project

Alex Cappiello (acappiel)

May 1, 2014

1 Introduction

Software bugs may effectively cost up to $312 billion per year[4]. So, why is the status quo so bad? Even a beginner computer science student will have heard repeatedly that successful test cases will never demonstrate correctness. On the other hand, Hoare logic (also Floyd-Hoare logic) provides a formal logical means of reasoning about the correctness of programs, which has been around since 1969[1]. In spite of this, unit testing dominates the industry and formal reasoning rarely makes an appearance. To make a long story short, it turns out that formally reasoning about code gets ugly fast. Accompanied with a general lack of tools to aid this kind of formal analysis, it would seem that this status quo is unlikely to change significantly.

In spite of this, the goal of this paper is to explore the role of Hoare logic in computer science and to look at recent efforts in the area. In its canonical form, Hoare logic strongly resembles sequential imperative code. In an effort to extend its usefulness, recent research has endeavored to expand into the realm of both synchronous computing and quantum computing, developing extensions of Hoare logic in those areas.

2 Hoare Logic Principles

At its core, Hoare logic is simply a formal set of logical rules (axioms and inference rules) to demonstrate correctness. These rules are expressed in terms of the Hoare triple, an expression of the form

$$\{P\}\ C\ \{Q\} \tag{1}$$

where P and Q are boolean-valued assertions and C is a command. The semantics of this expression are that when P (the precondition) is satisfied, then executing C leads to Q (the postcondition) being satisfied. There are generally considered to be two axioms and five inference rules, shown in Figure 1, where they are given shorthand names. Even though ⟨IF⟩ did not appear in Hoare's original paper, it can be derived from ⟨WHILE⟩ and for simplicity is generally included.

Another important concept is the difference between partial and total correctness. Partial correctness is used to show that if a loop terminates, then the postcondition will hold, but makes no guarantee that it will. Total correctness, on the other hand, is an augmented rule that is able to show total correctness. This is the distinction between ⟨WHILE⟩ and ⟨WHILE′⟩. The key idea behind

Empty axiom:

$$\langle \textsc{skip} \rangle \ \frac{*}{\{P\}\ \text{skip}\ \{P\}}$$

Assignment axiom:

$$\langle := \rangle \ \frac{*}{\{P[E/x]\}\ x := E\ \{P\}}$$

Rule of composition:

$$\langle ; \rangle \ \frac{\{P\}\ S\ \{Q\} \qquad \{Q\}\ T\ \{R\}}{\{P\}\ S; T\ \{R\}}$$

Conditional rule:

$$\langle \textsc{if} \rangle \ \frac{\{B \wedge P\}\ S\ \{Q\} \qquad \{\neg B \wedge P\}\ T\ \{Q\}}{\{P\}\ \textbf{if}\ B\ \textbf{then}\ S\ \textbf{else}\ T\ \textbf{endif}\ \{Q\}}$$

Consequence rule:

$$\langle \rightarrow \rangle \ \frac{P_1 \rightarrow P_2 \qquad \{P_2\}\ S\ \{Q_2\} \qquad Q_2 \rightarrow Q_1}{\{P_1\}\ S\ \{Q_1\}}$$

While rule:

$$\langle \textsc{while} \rangle \ \frac{\{P \wedge B\}\ S\ \{P\}}{\{P\}\ \textbf{while}\ B\ \textbf{do}\ S\ \textbf{done}\ \{\neg B \wedge P\}}$$

While rule for total correctness:

$$\langle \textsc{while}' \rangle \ \frac{D \text{ is well-ordered} \qquad [P \wedge B \wedge t \in D \wedge t = z]\ S\ [P \wedge t \in D \wedge t < z]}{[P \wedge t \in D]\ \textbf{while}\ B\ \textbf{do}\ S\ \textbf{done}\ [\neg B \wedge P \wedge t \in D]}$$

Figure 1: Axioms and inference rules.

total correctness is establishing some finite quantity that is strictly decreasing across loop iterations.

A brief description of each rule is as follows:

$\langle$**skip**$\rangle$**:** skip does not change the state of the program, so whatever was true beforehand remains true afterward.

$\langle := \rangle$**:** Let $P[E/x]$ be the assertion P where every occurrence of x is replaced by E. Then, the truth of $P[E/x]$ is equivalent to the truth of P after the assignment.

$\langle ; \rangle$**:** For some new intermediate condition Q, both $\{P\}\ S\ \{Q\}$ and $\{Q\}\ T\ \{R\}$ hold.

$\langle$**if**$\rangle$**:** Q must hold regardless of the truth of B.

$\langle \rightarrow \rangle$**:** It is permissible to strengthen the precondition and weaken the postcondition.

$\langle$**while**$\rangle$**:** The loop body must preserve the loop invariant.

$\langle$**while**$'\rangle$**:** The loop body preserves the loop invariant and the value t decreases.

Naturally, these rules beg the question of how to formulate these preconditions, postconditions, and loop invariants. In the general case, these do not merely come from the program directly and rather must be supplied by the programmer directly–a mixed blessing. At the expense of some extra effort from the programmer, extra information that is not needed for the program's execution is utilized to show correctness.

An example proof is given on the following page in Figure 2.

Suppose we have a program that adds the first n nonnegative integers, $0 + 1 + \cdots + n - 1$ in a loop. We would like to show that the result is $\frac{n(n-1)}{2}$. One such program is

$$\{n \geq 0\} s := 0; i := 0;\ \textbf{while}\ i < n\ \textbf{do}\ s := s + i; i := i + 1\ \textbf{done}\ \{s = n(n-1)/2\}$$

For convenience, let

$$\begin{aligned} LI &\equiv 0 \leq i \leq n \wedge s = i(i-1)/2 \\ LI' &\equiv 0 \leq i < n \wedge s = i(i+1)/2 \\ PC &\equiv \neg(i < n) \wedge LI \end{aligned}$$

Then, a proof is given by:

$$\dfrac{\langle;\rangle\ \dfrac{\langle{:=}\rangle\ \dfrac{*}{\{n \geq 0\} s := 0 \{n \geq 0 \wedge s = 0\}} \qquad \langle;\rangle\ \dfrac{\langle{:=}\rangle\ \dfrac{*}{\{n \geq 0 \wedge s = 0\} i := 0 \{LI\}} \qquad \langle\text{WHILE}\rangle\ \dfrac{\langle;\rangle\ \dfrac{\langle{:=}\rangle\ \dfrac{*}{\{LI \wedge i < n\} s := s + i \{LI'\}} \qquad \langle{:=}\rangle\ \dfrac{*}{\{LI'\} i := i + 1 \{LI\}}}{\{LI \wedge i < n\} s := s + i; i := i + 1 \{LI\}}}{\{LI\}\ \textbf{while}\ i < n\ \textbf{do}\ s := s + i; i := i + 1\ \textbf{done}\ \{PC\}}}{\{n \geq 0 \wedge s = 0\} i := 0;\ \textbf{while}\ i < n\ \textbf{do}\ s := s + i; i := i + 1\ \textbf{done}\ \{PC\}}}{\{n \geq 0\} s := 0; i := 0;\ \textbf{while}\ i < n\ \textbf{do}\ s := s + i; i := i + 1\ \textbf{done}\ \{PC\}}$$

Note that our goal of showing that $s = n(n-1)/2$ follows directly from simplifying PC, $\neg(i < n) \wedge 0 \leq i \leq n \wedge s = i(i-1)/2$.

Figure 2: Sample Hoare calculus proof.

3 Verification of Synchronous Languages

Hoare calculus is fundamentally structured around imperative languages, especially $\langle := \rangle$ and $\langle ; \rangle$, thus cannot be directly applied to synchronous programs. Hoare calculus is appealing to apply to synchronous programs since it is not affected by the state space explosion that hinders the model checking approach[2]. In order to adapt synchronous programs to fit this imperative model, a program is broken up into macro and micro steps. A macro step consists of many micro steps and represent logical breaks in the program, e.x. synchronization points. Micro steps can be thought of as atomic operations, of which many may be executed concurrently. Thus, assignments occurring in the micro steps of a single macro step can be collected into a single assignment step in the Hoare calculus. For this to work, all micro steps within a macro step must occur in the same execution environment.

To express theses concepts into the language, some new language constructs are required:

next$(x) = \tau$**:** delayed assignment. τ is evaluated in the current macro step, but x does not take this new value until the start of the following macro step.

pause: Used to mark the end of a macro step.

Gesell and Schneider proceed to identify two main approaches to extending Hoare calculus to this language[2]:

1. Add new rules for each statement of the synchronous language.
2. Translate the synchronous program to a sequential one and then apply classic Hoare calculus.

Both approaches have drawbacks. There are still several serious challenges that prevent simply writing rules for synchronous Hoare calculus. For example, determining which parts of the program affect one another (i.e. where to draw the macro step boundaries) quickly becomes nontrivial, to the point that a single statement may consist of several macro steps. This leads to increasingly complex rules. It may be tempting at this point to consider, rather than adapting Hoare calculus, to transform the program itself into an equivalent sequential program. However, doing so fundamentally changes the control flow of the program, so may not be trivial to do in the first place and the tradeoff comes with the need to define more elaborate invariants.

Consequently, Gesell and Schneider take a hybrid approach[2]:

> *Similar to the translation to sequential programs, we first identify all assignments of a macro step. Having the assignments of each macro step at hand, we then rewrite the program so that all assignments made in a macro step are done in a single assignment to a tuple of the writeable variables of the program. We thereby try to retain the rest of the syntactic structure of the synchronous program.*

Programs written in this manner are said to be in synchronous tuple assignment (STA) normal form. More precisely, the assignments $x_1 = \tau_1, \ldots, x_m = \tau_m$, such that τ_i only has occurrences of

```
module Sum(nat ?n, s, event !rdy) {
  nat i;
  (s, i).() = (0, 0).();
  pause;
  while (i < n) {
    ().(s, i) = ().(s + i, i + 1);
  }
  pause;
  (rdy).() = (true).();
}
```

Figure 3: STA sum code.

STA Assign:

$$\langle \text{STA} \rangle \frac{*}{\{\left[\cdots\left[[P]^{\pi_1,\ldots,\pi_n}_{y'_1,\ldots,y'_n}\right]^{\tau_n}_{x_n}\cdots\right]^{\tau_1}_{x_1}\}(x_1,\ldots,x_m).(y_1,\ldots,y_n) = (\tau_1\ldots,\tau_m).(\pi_1,\ldots,\pi_n)\{P\}}$$

Pause:

$$\langle \text{PAUSE} \rangle \frac{*}{\{\left[[\ldots P\ldots]^{\tau_1,\ldots,\tau_n}_{i_1 i,\ldots,i'_n}\right]^{y'_1\ldots y'_n}_{y_1\ldots y_n}\}\ \texttt{pause}\ \{P\}}$$

Figure 4: Synchronous assignment axioms.

$x_1,\ldots,x_{i-1}$, and $\mathbf{next}(y_1) = \pi_1,\ldots,\mathbf{next}(y_n) = \pi_n$ provides the single synchronous tuple assignment

$$(x_1,\ldots,x_m).(y_1,\ldots,y_n) = (\tau_1\ldots,\tau_m).(\pi_1,\ldots,\pi_n) \tag{2}$$

A synchronous program is in STA form if all actions are STAs and a **pause** appears between each STA. Our program from before can be written as the STA program in Figure 3.

In order to analyze these programs, axioms for STA assignment and `pause` are added to replace the original $\langle :=\rangle$ axiom. These axioms are given in Figure 4. STA assignment, $\langle \text{STA} \rangle$, handles each type of assignment separately. Immediate assignments are straightforward, as we simply substitute in the same manner as before. To handle delayed assignments, intermediate 'carrier' variables are introduced. When `pause` is reached, these carrier variables are finally bound to the original variable, as shown in $\langle \text{PAUSE} \rangle$. This step also updates the input variables, i_k.

4 Verification of Quantum Programs

4.1 Fundamentals of Quantum Programs

Canonical Hoare calculus is even less suited for analyzing quantum programs. In 2011, Ming proposed a complete Floyd-Hoare logic for reasoning about quantum programs[3]. One key motivating factor for its development is that quantum algorithms are generally not very intuitive, compared to their conventional counterparts, increasing the likelihood of programmer error. Before taking a look at this, it is first necessary to understand the model of quantum computation it is based on. A few definitions are needed to get things started:

Qbit: The fundamental unit of data in a quantum system is the qbit (or qubit), which may exist as a 0, 1, or a superposition of both. Using Dirac notation, the qbit ψ is written as a linear combination of this basis: $|\psi\rangle = \alpha|0\rangle + \beta|1\rangle \equiv \begin{pmatrix}\alpha\\ \beta\end{pmatrix}$, where $\alpha, \beta \in \mathbb{C}$ are probability amplitudes. This is simply a description of the 2-dimensional Hilbert space $\mathcal{H}_2$.

Quantum register: A sequence of qbits, analogous to a classical CPU register. Denoted $\bar{q} = q_1, \ldots, q_n$, where each q_i is a qbit.

Unitary Operator: A bounded linear operator $U : \mathcal{H} \to \mathcal{H}$ that satisfies $U^*U = UU^* = I$, where U^* is the adjoint (a generalization of complex conjugate) of U and $I : \mathcal{H} \to \mathcal{H}$ is the identity operator. In the statement $\bar{q} := U\bar{q}$, U is a unitary operator on $\mathcal{H}_{\bar{q}}$. Note that Ying uses $U^\dagger$ instead of U^*

Löwner Partial Order: For any $A, B \in \mathcal{L}(\mathcal{H})$, $A \sqsubseteq B$ if $B - A$ is a positive operator.

Trace Class Operator: A is a trace class operator if $\{\langle\psi_i|A|\psi_i\rangle\}_{i\in I}$ is summable for any orthonormal basis $\{|\psi_i\rangle\}_{i\in I}$ of $\mathcal{H}$. The trace of A, is $tr(A) = \sum_i \langle\psi_i|A|\psi_i\rangle$.

Density Operator: A positive operator, ρ, where $tr(\rho) = 1$.

Partial Density Operator: A positive operator, ρ, where $tr(\rho) \leq 1$. The set of these operators is referred to as $\mathcal{D}^-(\mathcal{H})$.

More generally, quantum data is defined over some Hilbert space. This system only deals with booleans and integers, which are definted over $\mathcal{H}_2$and $\mathcal{H}_\infty$, respectively.

The evolution of a closed quantum system at times t_0 and t are given by $|\psi_0\rangle$ and $|\psi\rangle$, respectively. Quantum mechanics tells us that these two states are related to each other by a unitary operator that only depends on t_0 and t. Specifically,

$$|\psi\rangle = U|\psi_i\rangle$$

This can be equivalently given by density operators given the states ρ_0 and ρ:

$$\rho = U\rho_0 U^*$$

The operational semantics of a quantum program is defined as

$$\langle S, \rho\rangle \to \langle S', \rho'\rangle$$

$$S ::= \textbf{skip} \mid q := 0 \mid \bar{q} := U\bar{q} \mid S_1; S_2 \mid \textbf{measure}\ M[\bar{q}] : \bar{S} \mid \textbf{while}\ M[\bar{q}] = 1\ \textbf{do}\ S$$

Figure 5: Grammar for quantum programs[3].

where the program S is proceeds one step forward, starting from state ρ to give state ρ' with S' being the remainder of the program. This is generalized for many state changes by letting

$$\langle S, \rho\rangle \to \langle S_1, \rho_1\rangle \to \cdots \to \langle S_{n-1}, \rho_{n-1}\rangle \to \langle S', \rho'\rangle = \langle S, \rho\rangle \to^n \langle S', \rho'\rangle$$

meaning that $\langle S', \rho'\rangle$ can be reached in n steps starting from $\langle S, \rho\rangle$ using the transition relation $\to$ (defined in Figure 6, using the grammar in Figure 5). A computation is said to diverge if the number of steps between two states is infinite. If it is finite and the final state is $\langle E, \rho'\rangle$, it is said to terminate at ρ'.

The most significant difference between classical and quantum computing is the idea of the quantum measurement. One of the fundamental aspects of quantum mechanics is that a system in superposition collapses upon being observed (think Schrödinger's Cat). Therefore, we cannot simply read a qbit without altering the system. In a sense, a quantum measurement can be thought of as a probabilistic choice in the program. Formally, we measure a system with state space $\mathcal{H}$ described by a collection $\{M_n\}$ of measurement operators that satisfy the normalization condition

$$\sum_m M_m^* M_m = I_{\mathcal{H}}$$

If the system is initially in the pure state $|\psi\rangle$, then the probability of result m is

$$p(m) = \langle\psi|M_m^* M_M|\psi\rangle$$

and the resulting state is

$$|\psi_m\rangle = \frac{M_m|\psi\rangle}{\sqrt{p(m)}}$$

In the case that a measurement M has only two outcomes (i.e. 0 and 1) then $M = \{M_0, M_1\}$ and is like the evaluation of a boolean statement. In the statement **measure** $M[\bar{q}] : \bar{S}$, $M = \{M_m\}$ is a measurement on the state space of $\bar{q}$, $\mathcal{H}_{\bar{q}}$, and $\bar{S} = \{S_m\}$ is the set of quantum programs where an outcome m of the measurement corresponds to S_m. In other words, the result of the measurement determines some subprogram S_m that will be executed next.

4.2 Correctness and Verification

The Hoare triple is extended to quantum programs by using quantum predicates to express the input and output states. Specifically, for some $X \subseteq Var$, where Var is some countably infinite set of quantum variables, a quantum predicate on $\mathcal{H}_X$ is a Hermitian operator P such that

$$0_{\mathcal{H}_X} \sqsubseteq P \sqsubseteq I_{\mathcal{H}_X}$$

(*Skip*) $$\frac{}{\langle \mathbf{skip}, \rho\rangle \to \langle E, \rho\rangle}$$

(*Initialization*) $$\frac{}{\langle q := 0, \rho\rangle \to \langle E, \rho_0^q\rangle}$$

where :

$$\rho_0^q = |0\rangle_q\langle 0|\rho|0\rangle_q\langle 0| + |0\rangle_q\langle 1|\rho|1\rangle_q\langle 0|$$

if $type(q) = \mathbf{Boolean}$, and

$$\rho_0^q = \sum_{n=-\infty}^{\infty} |0\rangle_q\langle n|\rho|n\rangle_q\langle 0|$$

if $type(q) = \mathbf{integer}$.

(*Unitary Transformation*) $$\frac{}{\langle \overline{q} := U\overline{q}, \rho\rangle \to \langle E, U\rho U^\dagger\rangle}$$

(*Sequential Composition*) $$\frac{\langle S_1, \rho\rangle \to \langle S_1', \rho'\rangle}{\langle S_1; S_2, \rho\rangle \to \langle S_1'; S_2, \rho\rangle}$$

where we make the convention that $E; S_2 \equiv S_2$.

(*Measurement*) $$\frac{}{\langle \mathbf{measure}\ M[\overline{q}] : \overline{S}, \rho\rangle \to \langle S_m, M_m\rho M_m^\dagger\rangle}$$

for each outcome m of measurement $M = \{M_m\}$

(*Loop 0*) $$\frac{}{\langle \mathbf{while}\ M[\overline{q}] = 1\ \mathbf{do}\ S, \rho\rangle \to \langle E, M_0\rho M_0^\dagger\rangle}$$

(*Loop 1*) $$\frac{}{\langle \mathbf{while}\ M[\overline{q}] = 1\ \mathbf{do}\ S, \rho\rangle \to \langle S; \mathbf{while}\ M[\overline{q}] = 1\ \mathbf{do}\ S, M_1\rho M_1^\dagger\rangle}$$

Figure 6: Transition semantics ($\to$) for quantum programs[3].

(*Axiom Skip*) $\{P\}\mathbf{Skip}\{P\}$

(*Axiom Initialization*) (1) If $type(q) = \mathbf{Boolean}$, then

$$\{|0\rangle_q\langle 0|P|0\rangle_q\langle 0| + |1\rangle_q\langle 0|P|0\rangle_q\langle 1|\}q := 0\{P\}$$

(2) If $type(q) = \mathbf{integer}$, then

$$\{\sum_{n=-\infty}^{\infty} |n\rangle_q\langle 0|P|0\rangle_q\langle n|\}q := 0\{P\}$$

(*Axiom Unitary Transformation*) $\{U^\dagger PU\}\overline{q} := U\overline{q}\{P\}$

(*Rule Sequential Composition*) $$\frac{\{P\}S_1\{Q\} \quad \{Q\}S_2\{R\}}{\{P\}S_1; S_2\{R\}}$$

(*Rule Measurement*) $$\frac{\{P_m\}S_m\{Q\} \text{ for all } m}{\{\sum_m M_m^\dagger P_m M_m\}\mathbf{measure}\ M[\overline{q}] : \overline{S}\{Q\}}$$

(*Rule Loop Partial*) $$\frac{\{Q\}S\{M_0^\dagger PM_0 + M_1^\dagger QM_1\}}{\{M_0^\dagger PM_0 + M_1^\dagger QM_1\}\mathbf{while}\ M[\overline{q}] = 1\ \mathbf{do}\ S\{P\}}$$

(*Rule Order*) $$\frac{P \sqsubseteq P' \quad \{P'\}S\{Q'\} \quad Q' \sqsubseteq Q}{\{P\}S\{Q\}}$$

Figure 7: Proof system qPD of partial correctness for quantum programs[3].

A Hoare triple, $\{P\}S\{Q\}$, is true for partial correctness if it holds that

$$tr(P\rho) \leq tr(Q[\![S]\!](\rho)) + [tr(\rho) - tr([\![S]\!](\rho))] \tag{3}$$

where $\rho \in \mathcal{D}^-(\mathcal{H})$. This can be interpreted as meaning that if input ρ satisfies P then either S terminates on it and the output $[\![S]\!](\rho)$ satisfies Q or S diverges.

Ying provides a complete proof system of partial correctness, qPD, reproduced in Figure 7. In the interest of brevity, I will not discuss the proof system in detail and I will not discuss total correctness. If the reader is interested, then refer to Ying's paper.

5 Summary

Above all else, these case studies demonstrate that Hoare logic is flexible, in that it can be extended to provide proof logic for languages well beyond its original intent. This merely comes down to

adapting the axioms and inference rules accordingly. As was seen in the analysis of synchronous programs, this comes at a tradeoff of modifying the language versus modifying the Hoare calculus. Modifying the language risks losing some of the features of the language whereas modifying the Hoare calculus risks increasing the complexity of the rules. For synchronous programs, an intermediate strategy was a suitable compromise for these tradeoffs. For quantum programs, their radically different nature precluded simplifying the language. As a result, the same ideas were employed to produce the Hoare calculus, but with everything redefined in terms of the operators for quantum computation. While not discussed, Hoare logic is fairly challenging to effectively take advantage of in practice. As a result, it is unlikely that this sort of program analysis will become ubiquitous. However, it seems to be better suited towards programs that are especially prone to human error, such as quantum programs, where the extra effort may still pay off in the long run.

6 References

1. C. A. R. Hoare. 1969. An axiomatic basis for computer programming. *Commun.* ACM 12, 10 (October 1969), 576-580. DOI=10.1145/363235.363259 `http://doi.acm.org/10.1145/363235.363259`

2. Manuel Gesell and Klaus Schneider. 2012. A hoare calculus for the verification of synchronous languages. In *Proceedings of the sixth workshop on Programming languages meets program verification* (PLPV '12). ACM, New York, NY, USA, 37-48. DOI=10.1145/2103776.2103782 `http://doi.acm.org/10.1145/2103776.2103782`

3. Mingsheng Ying. 2011. Floyd-Hoare logic for quantum programs. *ACM Trans. Program. Lang. Syst.* 33, 6, Article 19 (January 2012), 49 pages. DOI = 10.1145/2049706.2049708 `http://doi.acm.org/10.1145/2049706.2049708`

4. PRWeb. January 8, 2013. Cambridge University Study States Software Bugs Cost Economy $312 Billion Per Year. `http://www.prweb.com/releases/2013/1/prweb10298185.htm`