Lecture Notes: Axiomatic Semantics and Hoare-style Verification

17-355/17-665/17-819: Program Analysis (Spring 2020)
Claire Le Goues
clegoues@cs.cmu.edu

> It has been found a serious problem to define these languages [ALGOL, FORTRAN, COBOL] with sufficient rigor to ensure compatibility among all implementations...One way to achieve this would be to insist that all implementations of the language shall satisfy the axioms and rules of inference which underlie proofs of properties of programs expressed in the language. In effect, this is equivalent to accepting the axioms and rules of inference as the ultimately definitive specification of the meaning of the language.
>
> *C.A.R Hoare, An Axiomatic Basis for Computer Programming,1969*

So far in this course we have largely represented and reasoned about programs (and analysis of those programs) in terms of operational semantics, which gives meaning to programs based on what happens when we execute them. Now, we turn our attention to a different kind of representation, which in turn enables a different kind of static reasoning about program correctness.

1 Axiomatic Semantics

Axiomatic semantics (or Hoare-style logic) defines the meaning of a statement in terms of its effects on assertions of truth that can be made about the associated program. This provides a formal system for reasoning about correctness. An axiomatic semantics fundamentally consists of: (1) a language for stating assertions about programs (where an assertion is something like "if this function terminates, $x > 0$ upon termination"), coupled with (2) rules for establishing the truth of assertions. Various logics have been used to encode such assertions; for simplicity, we will begin by focusing on first-order logic.

In this system, a *Hoare Triple* encodes such assertions:

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

P is the precondition, Q is the postcondition, and S is a piece of code of interest. Relating this back to our earlier understanding of program semantics, this can be read as "if P holds in some state E and if $\langle S, E\rangle \Downarrow E'$, then Q holds in E'." We distinguish between partial ($\{P\}\ S\ \{Q\}$) and total ($[P]\ S\ [Q]$) correctness by saying that total correctness means that, given precondition P, S will terminate, and Q will hold; partial correctness does not make termination guarantees. We primarily focus on partial correctness.

1.1 Assertion judgements using operational semantics

Consider a simple assertion language adding first-order predicate logic to WHILE expressions:

$$\begin{array}{rcllll} A & ::= & \text{true} & \mid \text{false} & \mid e_1 = e_2 & \mid e_1 \geq e_2 \quad \mid A_1 \wedge A_2 \\ & \mid & A_1 \vee A_2 & \mid A_1 \Rightarrow A_2 & \mid \forall x.A & \mid \exists x.A \end{array}$$

Note that we are somewhat sloppy in mixing logical variables and program variables; all WHILE variables implicitly range over integers, and all WHILE boolean expressions are also assertions.

We now define an assertion judgement $E \vDash A$, read "A is true in E". The $\vDash$ judgment is defined inductively on the structure of assertions, and relies on the operational semantics of WHILE arithmetic expressions. For example:

$$\begin{array}{ll} E \vDash \text{true} & \textit{always} \\ E \vDash e_1 = e_2 & \textit{iff } \langle e_1, E\rangle \Downarrow n = \langle e_2, E\rangle \Downarrow n \\ E \vDash e_1 \geq e_2 & \textit{iff } \langle e_1, E\rangle \Downarrow n \geq \langle e_2, E\rangle \Downarrow n \\ E \vDash A_1 \wedge A_2 & \textit{iff } E \vDash A_1 \text{ and } E \vDash A_2 \\ \ldots & \\ E \vDash \forall x.A & \textit{iff } \forall n \in \mathbb{Z}.E[x := n] \vDash A \\ E \vDash \exists x.A & \textit{iff } \exists n \in \mathbb{Z}.E[x := n] \vDash A \end{array}$$

Now we can define formally the meaning of a partial correctness assertion $\vDash \{P\}\ S\ \{Q\}$:

$$\forall E \in \mathcal{E}.\forall E' \in \mathcal{E}.(E \vDash P \wedge \langle S, E\rangle \Downarrow E') \Rightarrow E' \vDash Q$$

Question: *What about total correctness?*

This gives us a formal, but unsatisfactory, mechanism to decide $\vDash \{P\}\ S\ \{Q\}$. By defining the judgement in terms of the operational semantics, we practically have to run the program to verify an assertion! It's also awkward/impossible to effectively verify the truth of a $\forall x.A$ assertion (check every integer?!). This motivates a new symbolic technique for deriving valid assertions from others that are known to be valid.

1.2 Derivation rules for Hoare triples

We write $\vdash A$ (read "we can prove A") when A can be derived from basic axioms. The derivation rules for $\vdash\ A$ are the usual ones from first-order logic with arithmetic, like (but obviously not limited to):

$$\frac{\vdash A \quad \vdash B}{\vdash A \wedge B}\ and$$

We can now write $\vdash \{P\}\ S\ \{Q\}$ when we can derive a triple using derivation rules. There is one derivation rule for each statement type in the language (sound familiar?):

$$\frac{}{\vdash \{P\}\ \text{skip}\ \{P\}}\ skip \qquad \frac{}{\vdash \{[e/x]P\}\ x{:=}e\ \{P\}}\ assign$$

$$\frac{\vdash \{P\}\ S_1\ \{P'\} \qquad \vdash \{P'\}\ S_2\ \{Q\}}{\vdash \{P\}\ S_1;\ S_2\ \{Q\}}\ seq \qquad \frac{\vdash \{P \wedge b\} S_1 \{Q\} \qquad \vdash \{P \wedge \neg b\}\ S_2\ \{Q\}}{\vdash \{P\}\ \text{if } b \text{ then } S_1 \text{ else } S_2\ \{Q\}}\ if$$

Question: *What can we do for while?*

There is also the *rule of consequence*:

$$\frac{\vdash P' \Rightarrow P \qquad \vdash \{P\}\ S\ \{Q\} \qquad \vdash Q \Rightarrow Q'}{\vdash \{P'\}\ S\ \{Q'\}}\ consq$$

This rule is important because it lets us make progress even when the pre/post conditions in our program don't exactly match what we need (even if they're logically equivalent) or are stronger or weaker logically than ideal.

We can use this system to prove that triples hold. Consider $\{\text{true}\}\ x := e\ \{x = e\}$, using (in this case) the assignment rule plus the rule of consequence:

$$\frac{\vdash \text{true} \Rightarrow e = e \qquad \overline{\{e = e\}\ x := e\ \{x = e\}}}{\vdash \{\text{true}\} x := e \{x = e\}}$$

We elide a formal statement of the soundness of this system. Intuitively, it expresses that the axiomatic proof we can derive using these rules is equivalent to the operational semantics derivation (or that they are sound and relatively complete, that is, as complete as the underlying logic).

2 Proofs of a Program

Hoare-style verification is based on the idea of a specification as a contract between the implementation of a function and its clients. The specification consists of the precondition and a postcondition. The precondition is a predicate describing the condition the code/function relies on for correct operation; the client must fulfill this condition. The postcondition is a predicate describing the condition the function establishes after correctly running; the client can rely on this condition being true after the call to the function.

Note that if a client calls a function without fulfilling its precondition, the function can behave in any way at all and still be correct. Therefore, if a function must be robust to errors, the precondition should include the possibility of erroneous input, and the postcondition should describe what should happen in case of that input (e.g. an exception being thrown).

The goal in Hoare-style verification is thus to (statically!) prove that, given a pre-condition, a particular post-condition will hold after a piece of code executes. We do this by generating a logical formula known as a *verification condition*, constructed such that, if true, we know that the program behaves as specified. The general strategy for doing this, introduced by Dijkstra, relies on the idea of a *weakest precondition* of a statement with respect to the desired post-condition. We then show that the given precondition implies it. However, loops, as ever, complicate this strategy.

2.1 Strongest postconditions and weakest pre-conditions

We can write any number of perfectly valid Hoare triples. Consider the Hoare triple $\{x = 5\}\ x := x * 2\ \{x > 0\}$. This triple is clearly correct, because if $x = 5$ and we multiply x by 2, we get $x = 10$ which clearly implies that $x > 0$. However, although correct, this Hoare triple is not a precise as we might like. Specifically, we could write a stronger postcondition, i.e. one that implies $x > 0$. For example, $x > 5 \wedge x < 20$ is stronger because it is more informative; it pins down the value of x more precisely than $x > 0$. The strongest postcondition possible is $x = 10$; this is the most useful

postcondition. Formally, if $\{P\}\ S\ \{Q\}$ and for all Q' such that $\{P\}\ S\ \{Q'\}$, $Q \Rightarrow Q'$, then Q is the strongest postcondition of S with respect to P.

We can compute the strongest postcondition for a given statement and precondition using the function $sp(S, P)$. Consider the case of a statement of the form $x := E$. If the condition P held before the statement, we now know that P still holds and that $x = E$—where P and E are now in terms of the old, pre-assigned value of x. For example, if P is $x+y = 5$, and S is $x := x+z$, then we should know that $x' + y = 5$ and $x = x' + z$, where x' is the old value of x. The program semantics doesn't keep track of the old value of x, but we can express it by introducing a fresh, existentially quantified variable x'. This gives us the following strongest postcondition for assignment:[1]

$$sp(x := E, P) \quad = \exists x'.[x'/x]P \wedge x = [x'/x]E$$

While this scheme is workable, it is awkward to existentially quantify over a fresh variable at every statement; the formulas produced become unnecessarily complicated, and if we want to use automated theorem provers, the additional quantification tends to cause problems. Dijkstra proposed reasoning instead in terms of *weakest preconditions*, which turns out to work better. If $\{P\}\ S\ \{Q\}$ and for all P' such that $\{P'\}\ S\ \{Q\}$, $P' \Rightarrow P$, then P is the weakest precondition $wp(S, Q)$ of S with respect to Q.

We can define a function yielding the weakest precondition inductively, following the Hoare rules. For for assignments, sequences, and if statements, this yields:

$$\begin{aligned}
&wp(x := E, P) && = [E/x]P \\
&wp(S;\ T, Q) && = wp(S,\ wp(T, Q)) \\
&wp(\text{if } B \text{ then } S \text{ else } T, Q) && = B \Rightarrow wp(S, Q) \wedge \neg B \Rightarrow wp(T, Q)
\end{aligned}$$

2.2 Loops

As usual, things get tricky when we get to loops. Consider:

$$\{P\}\ \text{while}(i < x)\ \text{do}\ f = f * i; i := i + 1\ \ \text{done}\{f = x!\}$$

What is the weakest precondition here? Fundamentally, we need to prove by induction that the property we care about will generalize across an arbitrary number of loop iterations. Thus, P is the base case, and we need some inductive hypothesis that is preserved when executing loop body an arbitrary number of times. We commonly refer to this hypothesis as a *loop invariant*, because it represents a condition that is always true (i.e. invariant) before and after each execution of the loop.

Computing weakest preconditions on loops is very difficult on real languages. Instead, we assume the provision of that loop invariant. A loop invariant must fulfill the following conditions:

- $P \Rightarrow I$: The invariant is initially true. This condition is necessary as a base case, to establish the induction hypothesis.
- $\{Inv \wedge B\}\ S\ \{Inv\}$: Each execution of the loop preserves the invariant. This is the inductive case of the proof.

[1]Recall that the operation $[x'/x]E$ denotes the capture-avoiding substitution of x' for x in E; we rename bound variables as we do the substitution so as to avoid conflicts.

- $(Inv \wedge \neg B) \Rightarrow Q$: The invariant and the loop exit condition imply the postcondition. This condition is simply demonstrating that the induction hypothesis/loop invariant we have chosen is sufficiently strong to prove our postcondition Q.

The procedure outlined above only verifies partial correctness, because it does not reason about how many times the loop may execute. Verifying full correctness involves placing an upper bound on the number of remaining times the loop body will execute, typically called a *variant function*, written v, because it is variant: we must prove that it decreases each time we go through the loop. We mention this for the interested reader; we will not spend much time on it.

2.3 Proving programs

Assume a version of WHILE that annotates loops with invariants: $\text{while}_{inv}\ b$ do S. Given such a program, and associated pre- and post-conditions:

$$\{P\}\ S_{inv}\ \{Q\}$$

The proof strategy constructs a verification condition $VC(S_{annot}, Q)$ that we seek to prove true (usually with the help of a theorem prover). VC is guaranteed to be stronger than $wp(S_{annot}, Q)$ but still weaker than P: $P \Rightarrow VC(S_{annot}, Q) \Rightarrow wp(S_{annot}, Q)$ We compute VC using a verification condition generation procedure $VCGen$, which mostly follows the definition of the wp function discussed above:

$$\begin{array}{lcl}
VCGen(\text{skip}, Q) & = & Q \\
VCGen(S_1; S_2, Q) & = & VCGen(S_1, VCGen(S_2, Q)) \\
VCGen(\text{if } b \text{ then } S_1 \text{ else } S_2, Q) & = & b \Rightarrow VCGen(S_1, Q) \wedge \neg b \Rightarrow VCGen(S_2, Q) \\
VCGen(x := e, Q) & = & [e/x]Q
\end{array}$$

The one major point of difference is in the handling of loops:

$$VCGen(\text{while}_{inv}\ e \text{ do } S, Q) = Inv \wedge (\forall x_1 ... x_n . Inv \Rightarrow (e \Rightarrow VCGen(S, Inv) \wedge \neg e \Rightarrow Q))$$

To see this in action, consider the following WHILE program:

$$\begin{array}{l}
r := 1; \\
i := 0; \\
\text{while } i < m \text{ do} \\
\qquad r := r * n; \\
\qquad i := i + 1
\end{array}$$

We wish to prove that this function computes the nth power of m and leaves the result in r. We can state this with the postcondition $r = n^m$.

Next, we need to determine a precondition for the program. We cannot simply compute it with wp because we do not yet know the loop invariant is—and in fact, different loop invariants could lead to different preconditions. However, a bit of reasoning will help. We must have $m \geq 0$ because we have no provision for dividing by n, and we avoid the problematic computation of 0^0 by assuming $n > 0$. Thus our precondition will be $m \geq 0 \wedge n > 0$.

A good heuristic for choosing a loop invariant is often to modify the postcondition of the loop to make it depend on the loop index instead of some other variable. Since the loop index runs from i to m, we can guess that we should replace m with i in the postcondition $r = n^m$. This gives us a first guess that the loop invariant should include $r = n^i$.

This loop invariant is not strong enough, however, because the loop invariant conjoined with the loop exit condition should imply the postcondition. The loop exit condition is $i \geq m$, but we need to know that $i = m$. We can get this if we add $i \leq m$ to the loop invariant. In addition, for proving the loop body correct, we will also need to add $0 \leq i$ and $n > 0$ to the loop invariant. Thus our full loop invariant will be $r = n^i \wedge 0 \leq i \leq m \wedge n > 0$.

Our next task is to use weakest preconditions to generate proof obligations that will verify the correctness of the specification. We will first ensure that the invariant is initially true when the loop is reached, by propagating that invariant past the first two statements in the program:

$$\begin{array}{l}\{m \geq 0 \wedge n > 0\} \\ r := 1; \\ i := 0; \\ \{r = n^i \wedge 0 \leq i \leq m \wedge n > 0\}\end{array}$$

We propagate the loop invariant past $i := 0$ to get $r = n^0 \wedge 0 \leq 0 \leq m \wedge n > 0$. We propagate this past $r := 1$ to get $1 = n^0 \wedge 0 \leq 0 \leq m \wedge n > 0$. Thus our proof obligation is to show that:

$$m \geq 0 \wedge n > 0 \Rightarrow 1 = n^0 \wedge 0 \leq 0 \leq m \wedge n > 0$$

We prove this with the following logic:

$m \geq 0 \wedge n > 0$	by assumption
$1 = n^0$	because $n^0 = 1$ for all $n > 0$ and we know $n > 0$
$0 \leq 0$	by definition of $\leq$
$0 \leq m$	because $m \geq 0$ by assumption
$n > 0$	by the assumption above
$1 = n^0 \wedge 0 \leq 0 \leq m \wedge n > 0$	by conjunction of the above

To show the loop invariant is preserved, we have:

$$\begin{array}{l}\{r = n^i \wedge 0 \leq i \leq m \wedge n > 0 \wedge i < m\} \\ r := r * n; \\ i := i + 1; \\ \{r = n^i \wedge 0 \leq i \leq m \wedge n > 0\}\end{array}$$

We propagate the invariant past $i := i+1$ to get $r = n^{i+1} \wedge 0 \leq i+1 \leq m \wedge n > 0$. We propagate this past $r := r * n$ to get: $r * n = n^{i+1} \wedge 0 \leq i + 1 \leq m \wedge n > 0$. Our proof obligation is therefore:

$$\begin{array}{l}r = n^i \wedge 0 \leq i \leq m \wedge n > 0 \wedge i < m \\ \Rightarrow r * n = n^{i+1} \wedge 0 \leq i + 1 \leq m \wedge n > 0\end{array}$$

We can prove this as follows:

$r = n^i \wedge 0 \leq i \leq m \wedge n > 0 \wedge i < m$	by assumption
$r * n = n^i * n$	multiplying by n
$r * n = n^{i+1}$	definition of exponentiation
$0 \leq i + 1$	because $0 \leq i$
$i + 1 < m + 1$	by adding 1 to inequality
$i + 1 \leq m$	by definition of $\leq$
$n > 0$	by assumption
$r * n = n^{i+1} \wedge 0 \leq i + 1 \leq m \wedge n > 0$	by conjunction of the above

Last, we need to prove that the postcondition holds when we exit the loop. We have already hinted at why this will be so when we chose the loop invariant. However, we can state the proof obligation formally:

$$r = n^i \wedge 0 \leq i \leq m \wedge n > 0 \wedge i \geq m$$
$$\Rightarrow r = n^m$$

We can prove it as follows:

$r = n^i \wedge 0 \leq i \leq m \wedge n > 0 \wedge i \geq m$	by assumption
$i = m$	because $i \leq m$ and $i \geq m$
$r = n^m$	substituting m for i in assumption