

CS256: Programming Languages and Semantics

Introduction to Axiomatic Semantics

<http://www.eecs.harvard.edu/~greg/cs256sp2006/>

Greg Morrisett
greg@eecs.harvard.edu
327 Maxwell Dworkin

February 15, 2006

1 Overview

The topic for today is axiomatic (or relational) semantics. The goal is to use an assertion language to characterize the behavior of program fragments in a compositional fashion. The use of relations (a.k.a. assertions) allows us to abstract from details that may be irrelevant to the outside world.

Note: Winskell's book covers this topic particularly well, so I'm only going to sketch what's in the book.

We start by defining an assertion language, which corresponds to first-order, predicate logic, over the sort of integer expressions. Formally, the abstract syntax of assertions A is given by the following BNF:

$$\begin{aligned} v &\in \text{LogVar} \\ t &\in \text{Term} & ::= v \mid \mathbf{x} \mid i \mid e_1 + e_2 \mid e_1 * e_2 \\ A &\in \text{Assn} & ::= \text{true} \mid \text{false} \mid t_1 = t_2 \mid t_1 \leq t_2 \mid A_1 \wedge A_2 \mid A_1 \vee A_2 \mid \neg A \mid \\ & & A_1 \supset A_2 \mid \forall v. A \mid \exists v. A \end{aligned}$$

You'll notice that terms are essentially IMP expressions extended with one extra case: v which ranges over so-called logical variables (as opposed to \mathbf{x} which ranges over program variables, and i which ranges over integer constants.) Assertions then consist of the constants true, false, equality and inequality of terms, conjunctions of assertions, disjunctions of assertions, the negation of an assertion, and universally and existentially quantified assertions. Note that the quantified variables range over integers. Throughout, I'll use liberal encodings of various assertions (e.g., $t_1 < t_2$ abbreviates $t_1 + 1 \leq t_2$).

Next, we need to talk about when a given store s satisfies an assertion A (written $s \models A$.) However, we must account for the free logic variables that may appear within an assertion. To do so, we introduce the notion of an *interpretation* (\mathcal{I}) which is a mapping from logic variables to integer values:

$$\mathcal{I} \in \text{LogVar} \rightarrow Z$$

and we talk about a store and interpretation pair satisfying an assertion (written $s \models_{\mathcal{I}} A$). This is defined inductively based on the abstract syntax of assertions:

$$\begin{aligned} s \models_{\mathcal{I}} \text{true} & \quad (\text{i.e., all stores satisfy true.}) \\ s \not\models_{\mathcal{I}} \text{false} & \quad (\text{i.e., no store satisfies false.}) \\ s \models_{\mathcal{I}} t_1 = t_2 & \quad \text{if } (Et_1 s \mathcal{I}) = (Et_2 s \mathcal{I}) \\ s \models_{\mathcal{I}} t_1 \leq t_2 & \quad \text{if } (Et_1 s \mathcal{I}) \leq (Et_2 s \mathcal{I}) \\ s \models_{\mathcal{I}} A_1 \wedge A_2 & \quad \text{if } s \models_{\mathcal{I}} A_1 \text{ and } s \models_{\mathcal{I}} A_2 \\ s \models_{\mathcal{I}} A_1 \vee A_2 & \quad \text{if } s \models_{\mathcal{I}} A_1 \text{ or } s \models_{\mathcal{I}} A_2 \\ s \models_{\mathcal{I}} \neg A & \quad \text{if } s \not\models_{\mathcal{I}} A \\ s \models_{\mathcal{I}} A_1 \supset A_2 & \quad \text{if } s \models_{\mathcal{I}} \neg A_1 \text{ or } s \models_{\mathcal{I}} A_2 \\ s \models_{\mathcal{I}} \forall v. A & \quad \text{if for all } i, s \models_{\mathcal{I}[v \mapsto i]} A \\ s \models_{\mathcal{I}} \exists v. A & \quad \text{if there exists an } i \text{ such that } s \models_{\mathcal{I}[v \mapsto i]} A \end{aligned}$$

This is one of those annoying definitions that doesn't seem to go anywhere because it's defining the meaning of the satisfaction relation using English to reinterpret the various symbols. (Question: what goes wrong if we try to formalize these rules as a deductive system in something like Twelf?)

Of course, the definition depends upon an evaluation function E which maps a term t , a store s , and an interpretation \mathcal{I} to an integer. It's defined in the same fashion as our denotational evaluation function for IMP expressions, except we have the extra case for logic variables which is handled by the interpretation:

$$\begin{aligned} E \ v \ s \ \mathcal{I} &= \mathcal{I}(v) \\ E \ \mathbf{x} \ s \ \mathcal{I} &= s(\mathbf{x}) \\ E \ i \ s \ \mathcal{I} &= i \\ E \ (e_1 + e_2) \ s \ \mathcal{I} &= (E \ e_1 \ s \ \mathcal{I}) + (E \ e_2 \ s \ \mathcal{I}) \\ E \ (e_1 * e_2) \ s \ \mathcal{I} &= (E \ e_1 \ s \ \mathcal{I}) * (E \ e_2 \ s \ \mathcal{I}) \end{aligned}$$

We say $s \models A$ when for any interpretation \mathcal{I} , we can show $s \models_{\mathcal{I}} A$. We say that an assertion A is valid (written $\models A$) if for all stores s , $s \models A$.

Next we can define the meaning of the Hoare *partial correctness triple*, which is written $\{A_1\}c\{A_2\}$. Intuitively, this is an assertion which is meant to claim that if we start with a store s_1 which satisfies the assertion A_1 , and if we run the configuration $\langle c, s_1 \rangle$, then *if* the configuration terminates with an output store s_2 , then s_2 will satisfy A_2 . So, we write:

$$\models \{A_1\}c\{A_2\} \quad \text{iff} \quad \forall s_1, s_2. (s_1 \models A_1 \wedge \langle c, s_1 \rangle \Downarrow s_2) \supset (s_2 \models A_2).$$

This definition is in contrast to the *total correctness triple* $[A_1]c[A_2]$ whose meaning is given by:

$$\models [A_1]c[A_2] \quad \text{iff} \quad \forall s_1. (s_1 \models A_1) \supset (\exists s_2. \langle c, s_1 \rangle \Downarrow s_2 \wedge s_2 \models A_2).$$

2 Hoare Proof Rules

The previous section defined the meaning of Hoare-style specifications where we associate pre- and post-conditions with a given command. The assertions capture properties of the store that must hold upon input and output of the command. But using the definition directly to prove properties about code is not so easy. Fortunately, Hoare constructed a proof system that allows us to (mostly) mechanize the construction of a proof that a given Hoare-triple is valid.

These rules are given below, again based on the structure of the abstract syntax for IMP commands:

$$\begin{aligned} (H1) \quad & \{A\}\mathbf{skip}\{A\} \\ (H2) \quad & \{A[e/\mathbf{x}]\}\mathbf{x} := e\{A\} \\ (H3) \quad & \frac{\{A_1\}c_1\{A_2\} \quad \{A_2\}c_2\{A_3\}}{\{A_1\}c_1; c_2\{A_3\}} \\ (H4) \quad & \frac{\{A_1 \wedge e \neq 0\}c_1\{A_2\} \quad \{A_1 \wedge e = 0\}c_2\{A_2\}}{\{A_1\}\mathbf{if} \ e \ \mathbf{then} \ c_1 \ \mathbf{else} \ c_2\{A_2\}} \\ (H5) \quad & \frac{\{A \wedge e \neq 0\}c\{A\}}{\{A\}\mathbf{while} \ e \ \mathbf{do} \ c\{A \wedge e = 0\}} \\ (H6) \quad & \frac{\models A_1 \supset A'_1 \quad \{A'_1\}c\{A'_2\} \quad \models A'_2 \supset A_1}{\{A_1\}c\{A_2\}} \end{aligned}$$

The rules for skip and sequential composition should be fairly self-evident. The rule for assignment is always confusing to me. In essence, it says that if we want A to hold after the assignment, then it's sufficient to

show that A with e substituted for x holds before the assignment. For example, if we want to ensure that $x > y$ then we must show $e > y$.

The rule for conditionals is also relatively straightforward: if we start knowing A_1 is true, then in the case that c_1 is executed, we know in addition to A_1 remaining true (since expressions have no side effects) that also the expression evaluated to a non-zero value. If we can take those two facts and show that executing c_1 in any state satisfying them yields our desired post-condition A_2 , then this branch is okay. Dually, for c_2 , we must add in the assumption that the expression evaluated to zero and from this establish the post-condition.

The rule for while-loops is, in some sense, where all the real action is, but to understand it, let's first talk about the last rule which is sometimes called the *rule of consequence*. It says that we can always strengthen the pre-condition (i.e., rule out more initial states) and weaken the post-condition (i.e., throw in more potential final states) and the triple is still valid.

This rule gets used heavily in conjunction with loops. In particular, to verify a pre- and post-condition for a loop, we must first strengthen the pre-condition to a loop invariant A . Then we must show that this invariant is preserved by each iteration of the loop. Then we use consequence to throw away (i.e., weaken) the loop invariant information that is unneeded for the post-condition.

For example, suppose we wish to establish that:

$$\{x = n \wedge y = m \wedge s = 0\} \quad \text{while } x \text{ do } (s := s + y; x := x - 1) \quad \{s = n * m\}$$

We cannot use the pre-condition as the invariant for the while loop because both x and s change. However, we can use the rule of consequence to weaken the pre-condition to:

$$\{\exists v. x = n - v \wedge y = m \wedge s = m * v\} \quad \text{while } x \text{ do } (s := s + y; x := x - 1) \quad \{s = n * m\}$$

since it's relatively easy to show that:

$$(x = n \wedge y = m \wedge s = 0) \supset (\exists v. x = n - v \wedge y = m \wedge s = m * v)$$

choosing as the witness for v the value 0. We still can't use the while rule directly since the post-condition doesn't match the invariant. We can use the while-rule on the following triple:

$$\begin{array}{l} \{\exists v. x = n - v \wedge y = m \wedge s = m * v\} \\ \text{while } x \text{ do } (s := s + y; x := x - 1) \\ \{\exists v. x = n - v \wedge y = m \wedge s = m * v \wedge x = 0\} \end{array}$$

So again, we can use the rule of consequence to strengthen the post-condition as long as we can show:

$$(\exists v. x = n - v \wedge y = m \wedge s = m * v \wedge x = 0) \supset s = n * m$$

which follows pretty directly since when x is both 0 and $n - v$ we can conclude $v = n$, and thus $s = m * v$ implies $s = m * n$.

Now we must show that our guess is indeed an invariant for the loop. That is, we must check that:

$$\begin{array}{l} \{\exists v. x = n - v \wedge y = m \wedge s = m * v \wedge x \neq 0\} \\ s := s + y; x := x - 1 \\ \{\exists v. x = n - v \wedge y = m \wedge s = m * v\} \end{array}$$

We can break this down with the rule of composition by first finding an intermediate assertion A such that:

$$\{A\} \quad x := x - 1 \quad \{\exists v. x = n - v \wedge y = m \wedge s = m * v\}$$

and then try to establish that:

$$\{\exists v. x = n - v \wedge y = m \wedge s = m * v \wedge x \neq 0\} \quad s := s + y \quad \{A\}$$

Using the assignment rule, we can solve for A :

$$\{\exists v. x - 1 = n - v \wedge y = m \wedge s = m * v\} \quad x := x - 1 \quad \{\exists v. x = n - v \wedge y = m \wedge s = m * v\}$$

Taking this pre-condition as the post-condition of the first assignment, we can use the assignment rule again to establish:

$$\{\exists v. x - 1 = n - v \wedge y = m \wedge s + y = m * v\} \quad s := s + y \quad \{\exists v. x - 1 = n - v \wedge y = m \wedge s = m * v\}$$

So we've established that:

$$\begin{aligned} & \{\exists v. x - 1 = n - v \wedge y = m \wedge s + y = m * v\} \\ & s := s + y; x := x - 1 \\ & \{\exists v. x = n - v \wedge y = m \wedge s = m * v\} \end{aligned}$$

This doesn't quite match our invariant, so we need to use consequence again which obligates us to show that our invariant implies the pre-condition we calculated for the loop body:

$$\begin{aligned} & \{\exists v. x = n - v \wedge y = m \wedge s = m * v \wedge x \neq 0\} \supset \\ & \quad \{\exists v'. x - 1 = n - v' \wedge y = m \wedge s + y = m * v'\} \end{aligned}$$

I've taken the liberty of α -converting (i.e., renaming) v to v' to avoid confusion. It's fairly easy to see that the assertion above is logically equivalent to:

$$\begin{aligned} & \{\exists v. x = n - v \wedge y = m \wedge s = m * v \wedge x \neq 0\} \supset \\ & \quad \{\exists v'. x = n - (v' + 1) \wedge y = m \wedge s = m * (v' + 1)\} \end{aligned}$$

and picking $v - 1$ as the witness for v' establishes the result.

An interesting fact is that of course, the loop might not terminate (in particular when $n < 0$). So the point is that the fact which we established cannot be extended to a total correctness assertion. The primary difference for the proof system of total assertions is that in addition to identifying a loop invariant, we must also identify a loop metric (essentially a mapping from the machine state to a natural number) that always decreases and thus eventually reaches zero. If we strengthened the pre-condition of the loop to assume that $n \geq 0$, then we could pick the value of x as the loop metric.

3 Soundness of the Proof System

We say that a triple $\{A_1\}c\{A_2\}$ is provable (written $\vdash \{A_1\}c\{A_2\}$) if we can construct a derivation using the Hoare proof rules the relation holds. Of course, we need to establish the *soundness* of the rules in order to conclude that $\vdash \{A_1\}c\{A_2\} \supset \models \{A_1\}c\{A_2\}$. That is, we'd like to check that the proof rules really do align with our definition of what the partial correctness assertions mean, which in turn appeals to the operational semantics for the language.

This can be achieved by structural induction on Hoare derivations and is a fairly easy exercise, so I'll leave that as a homework for you. Perhaps the trickiest part is establishing the soundness of the assignment rule...

4 [In]completeness of the Hoare Rules

Whenever you write down a proof system for something, in addition to establishing *soundness* (which shows that you can only prove true things), you'd love to also get a notion of *completeness*. That is, you'd like to show that anything which is true is provable. In some sense, this would allow us to say that this is the "right" proof system in that we'll never come up short when it comes to the provability of some fact.

Sadly, the Hoare rules are not complete, and indeed, there's nothing that we can do try to make them complete. Notice that the rule of consequence demands that we show $\models A_1 \supset A'_1$ instead of $\vdash A_1 \supset A'_1$.

That is, we can use the rule as long as $A_1 \supset A'_1$ is true for all stores, even if we can't (formally) prove that this implication holds! Now you might hope to write down a proof system that allows us to establish this implication. But Goedel's incompleteness theorem tells us that for this particular logic (which is powerful enough to "encode" itself), there is no finite axiomatization that is complete. That is, we cannot write down a proof system that allows us to prove all of the true assertions.

So what does that tell us? That tells us that there is some Hoare triple that is valid, but for which we cannot hope to construct a (formal) proof that shows its validity.

So in what sense are the Hoare rules "good" or "right"? If I wrote down a different proof system, and established its soundness, how could I possibly claim that it was better or worse than Hoare's rules?

Well, it turns out that the great thing about Hoare's rules is that they isolate all of the incompleteness in the rule of consequence. That is, it turns out that these rules are *relatively complete* in the sense that if we have an oracle which can decide whether or not assertions are valid, then by golly, we can "prove" any valid triple using Hoare's deduction system.

Furthermore, and somewhat surprisingly, we only need to use this oracle *once* in any derivation. That is, we can *mechanically* reduce whether or not $\models \{A_1\}c\{A_2\}$ is true to a single question in first-order logic of the form $A_1 \supset A$ for some assertion A . Of course, what we can't do is mechanically determine whether or not $A_1 \supset A$.

The proofs of these two facts are closely connected and revolve around the fact that given a command c and a post-condition A_2 , we can automatically compute a pre-condition A_1 that *precisely* describes the set of stores s such that $\langle c, s \rangle \Downarrow s'$ implies $s' \models A_2$. By precise, I mean that A_1 includes only those stores s which result in outputs that satisfy the post-condition. In this sense, A_1 is the *weakest liberal pre-condition* of the command c and the post-condition A_2 .

In some sense, the ability to mechanically compute weakest pre-conditions is like the "penultimate" dataflow analysis, where the abstract domain that we're using is first-order predicate logic, and we can reduce just about any question about the program to a single logical implication. This sounds great until you realize that all we've really done is *encoded* the semantics of the program as an assertion, so all we've done is reduced the question about a program to a question about an alternative encoding of its semantics. Still, because the assertion language is "pure" in a certain sense, doing symbolic manipulation of that question is often easier than trying to do it directly on the program text.

5 Homework

1. Prove (on paper) the soundness of the Hoare rules.
2. Encode the assertion language and Hoare proof rules in Twelf.