CS655 Lecture 7 | Axiomatic Semantics 2: With A Vengeance | 9 February, 2006

Scribe: Daniel Dunbar | Lecturer: Wes Weimer

1 Introduction

In the last lecture axiomatic semantics were introduced for use in proving properties of programs. The Hoare rules were given so that derivations could be on the basis of judgments and not only on the denotational or operation semantics. In order to ensure that these rules correspond with the notion of truth it is necessary to show that the Hoare rules are *sound*, i.e. that it is not possible to prove an assertion that is false. It is also desirable that this implication can also be reversed, that is that any true assertion can be proved using the Hoare rules.

2 Soundness

A formal system is sound by definition if $\vdash \{A\}c\{B\}$ implies that $\forall_{\sigma\in\Sigma} \models \{A\}c\{B\}$. By expanding the notion of an assertion judgment then this holds if, for all states $\sigma \in \Sigma$,

$$\frac{\sigma \models A \qquad \mathsf{Op} :: \langle c, \sigma\rangle \Downarrow \sigma' \qquad \mathsf{Pr} :: \vdash \{A\}c\{B\}}{\sigma' \models B}$$

where Op is a derivation in operational semantics and Pr is an axiomatic semantics proof.

In order prove this it is not sufficient to use one of the previous techniques for induction. Inducting on the structure of c or Op will encounter problems with the while rule. Similarly, inducting on the structure of Pr will encounter problems with the rule of consequence. However, by inducting on the structure of both Op and Pr simultaneously the proof can be carried out.

This induction will be established using well-founded induction over the lexicographic ordering of $(\mathsf{Op}, \mathsf{Pr})$ where both of Op and Pr are ordered by substructure. That is, for $\mathsf{Op} < \mathsf{Op}'$ ($\mathsf{Pr} < \mathsf{Pr}'$) iff Op (Pr) is a substructure of Op' (Pr'). The well-founded order is defined by $(\mathsf{Op}, \mathsf{Pr}) \prec (\mathsf{Op}', \mathsf{Pr}')$ iff $\mathsf{Op} < \mathsf{Op}'$ or $\mathsf{Op} = \mathsf{Op}'$ and $\mathsf{Pr} < \mathsf{Pr}'$.

Using this induction method the proof is mostly straightforward except for the while rule. If the last rule in Pr was a while rule then Pr is

$$\frac{\mathsf{Pr}_1 :: \vdash \{A \wedge b\}c\{A\}}{\vdash \{A\} \text{ while } b \text{ do } c\{A \wedge \neg b\}}$$

and there are two possible rules for the last rule in Op by inversion and assume $\sigma \models A$. In the false case then Op has the form

$$\frac{\mathsf{Op}_1 :: \langle b, \sigma\rangle \Downarrow \mathit{false}}{\langle \text{while } b \text{ do } c\rangle \Downarrow \sigma}$$

and by the soundness of booleans and Op_1 then $\sigma \models A \wedge \neg b$ and so $\models \{A\}\text{while } b \text{ do } c\{A \wedge \neg b\}$. In the true case Op is of the form

$$\frac{\mathsf{Op}_1 :: \langle b, \sigma\rangle \Downarrow \mathit{true} \qquad \mathsf{Op}_2 :: \langle c, \sigma\rangle \Downarrow \sigma' \qquad \mathsf{Op}_3 :: \langle \text{while } b \text{ do } c, \sigma'\rangle \Downarrow \sigma''}{\langle \text{while } b \text{ do } c, \sigma\rangle \Downarrow \sigma''}$$

and as before by the soundness of booleans $\sigma \models A \wedge b$. By the induction hypothesis on Pr_1 and Op_2 this implies that $\sigma' \models A$. Since $\mathsf{Op}_3 < \mathsf{Op}$ then $(\mathsf{Op}_3, \mathsf{Pr}) \prec (\mathsf{Op}, \mathsf{Pr})$ and so by the induction hypothesis $\sigma'' \models A \wedge \neg b$ and therefore $\models \{A\}$ while b do $c\{A \wedge \neg b\}$.

3 Completeness

Axiomatic semantics is said to be complete if $\models \{A\}c\{B\}$ implies that there exists a proof $\mathsf{Pr} \vdash \{A\}c\{B\}$. Completeness ensures that the Hoare rules are sufficient to verify all valid properties of a program. Because

there is a logic embedded in the choice of an assertion language, axiomatic semantics could only be complete if the chosen logic was also complete. This is unsatisfactory since we would like to use first-order logic, and first-order logic is known to be incomplete.

This problem is solved by considering the *relative* completeness of axiomatic semantics. We will say that axiomatic semantics are *relatively complete* if they would be complete under the assumption that the underlying logic is also complete. Under this definition the Hoare rules are relatively complete.

For example, consider the assertion judgment $\models \{x < 5 \wedge z = 2\}y := x + 2\{y < 7\}$. Since $\vdash \{x + 2 < 6\}y := x + 2\{y < 7\}$ and $\vdash x < 5 \wedge z = 2 \Rightarrow x + 2 < 7$ and clearly $\vdash \{y < 7\} \Rightarrow \{y < 7\}$ so by the rule of consequence $\vdash \{x < 5 \wedge z = 2\}y := x + 2\{y < 7\}$. The difficulty here is in determining the appropriate assertion which can be derived while supporting the conclusion.

Dijkstra's insight into this problem was that in order to verify that $\{A\}c\{B\}$ first consider all predicates A' such that $\models \{A'\}c\{B\}$. This set is denoted $\mathsf{Pre}(c, B)$ and contains all the preconditions of B for command c. Additionally, it is necessary to verify that for some $A' \in \mathsf{Pre}(c, B)$ then $A \Rightarrow A'$. Any condition in this set would suffice to prove that verify the assertion, but most of these conditions will be overly strong (e.g. *false*!) and not provable. However, the conditions in this set can be ordered by implication and so the proof strategy is to compute the weakest precondition $\mathsf{wp}(c, B)$ in this set and prove that $A \Rightarrow \mathsf{wp}(c, B)$.

The conditions for $\mathsf{wp}(c, B)$ can be expressed by $\vdash \{\mathsf{wp}(c, B)\}c\{B\}$ (wp is a precondition according to the Hoare rules) and $\models \{A\}c\{B\}$ implies that $\models A \Rightarrow \mathsf{wp}(c, B)$. If these properties hold and A is complete ($\models A$ implies $\vdash A$) then

$$\frac{\vdash A \Rightarrow \mathsf{wp}(c, B) \qquad \vdash \{\mathsf{wp}(c, B)\}c\{B\}}{\vdash \{A\}c\{B\}}.$$

The weakest precondition is defined inductively on c following the Hoare rules.

- $\mathsf{wp}(c_1;\ c_2, B) = \mathsf{wp}(c_1, \mathsf{wp}(c_2, B))$
- $\mathsf{wp}(x := e, B) = [e/x]B$
- $\mathsf{wp}(\mathsf{if}\ e\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2, B) = (e \Rightarrow \mathsf{wp}(c_1, B) \wedge \neg e \Rightarrow \mathsf{wp}(c_2, B))$

To derive the weakest precondition for loops we start with the unwinding equivalence

$$\mathsf{while}\ b\ \mathsf{do}\ c = \mathsf{if}\ b\ \mathsf{then}\ c;\ \mathsf{while}\ b\ \mathsf{do}\ c\ \mathsf{else}\ \mathsf{skip}.$$

Letting $w = \mathsf{while}\ b\ \mathsf{do}\ c$ and $W = \mathsf{wp}(w, B)$ then

$$W = b \Rightarrow \mathsf{wp}(c, W) \wedge \neg b \Rightarrow B.$$

This is a recursive equation that can be solved using domain theory if a complete partial order over assertions can be defined.

Define an ordering $A \sqsubseteq A'$ iff $\models A' \Rightarrow A$. This defines a partial order, and for any chain

$$A_1 \sqsubseteq A_2 \sqsubseteq \ldots$$

then the infinite conjunction $\wedge A_i$ is a least upper bound for the chain. Define

$$F(A) = b \Rightarrow \mathsf{wp}(c, A) \wedge \neg b \Rightarrow B;$$

this function is both monotonic and continuous, and the least-fixed point is

$$\mathsf{wp}(w, B) = \wedge F^i(true).$$

Alternatively, we can define a family of weakest preconditions $\mathsf{wp_k}(\mathsf{while}\ e\ \mathsf{do}\ c, B)$ as the weakest precondition for which the loop terminates with assertion B holding *if* it terminates in k or fewer iterations. With this definition

$$\mathsf{wp_0} = \neg b \Rightarrow B,$$

$$\mathsf{wp_{i+1}} = e \Rightarrow \mathsf{wp}(c, \mathsf{wp_i}) \wedge \neg e \Rightarrow B,$$

and

$$\mathsf{wp}(\mathsf{while}\ e\ \mathsf{do}\ c, B) = \wedge \mathsf{wp_k} = lub\{\mathsf{wp_k} : k \geq 0\}.$$

4 Verification Conditions

Although the mathematical theory is sound and suffices to show completeness (see Necula reference), in general weakest preconditions are impossible to compute. However, the actual weakest precondition was not necessary in order to verify the condition. Define a verification condition $\mathsf{VC}(c, B)$ as any precondition of B for command c for which it is possible to verify that given $\{A\}c\{B\}$ then $A \Rightarrow \mathsf{VC}(c, B) \rightarrow \mathsf{wp}(c, B)$.

This is not inherently simpler, but it turns out that the hard part of determining the verification conditions in practice is discovering the loop invariants. By introducing a new form of while that includes a loop invariant, and requiring the programmer to specify this invariant, then the process of computing verification conditions can be automated. A process for computing $\mathsf{VC}(c, B)$ is called VCGen.

The new while command will have the form

$$\mathsf{while}_{inv}\ e\ \mathsf{do}\ c$$

where the invariant *inv* should hold every time before e is evaluated.

4.1 Verification Condition Generation

The generation of $\mathsf{VC}(c, B)$ is defined in a manner much like that for wp:

- $\mathsf{VC}(\mathsf{skip}, B) = B$
- $\mathsf{VC}(c_1;\ c_2, B) = \mathsf{VC}(c_1, \mathsf{VC}(c_2, B))$
- $\mathsf{VC}(\mathsf{if}\ b\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2, B) = (b \Rightarrow \mathsf{VC}(c_1, B) \wedge \neg b \Rightarrow \mathsf{VC}(c_2, B)$
- $\mathsf{VC}(x := e, B) = [e/x]B$
- $\mathsf{VC}(\mathsf{while}_{inv}\ e\ \mathsf{do}\ c, B) = inv \wedge \forall_{x_0,\ldots,x_n} inv \rightarrow (e \rightarrow \mathsf{VC}(c, inv) \wedge \neg e \Rightarrow B)))$

For the while rule $x_1, \ldots, x_n$ represent all the variables that are modified in c. The condition can be interpreted as:

- *inv* holds on entry,
- for any arbitrary iteration,

  inv is preserved by the command, and

  B holds if the loop terminates.

4.2 Example

Suppose we wish to compute the verification condition of the following program P with the post-condition $x \neq 0$:

$$\begin{array}{l} x = 0; \\ y = 2; \\ \mathsf{while}_{x+y=2}\ y > 0\ \mathsf{do} \\ \quad y := y - 1; \\ \quad x := x + 1 \end{array}$$

By the sequencing rule $\mathsf{VC}(P, x \neq 0)$ first computes the verification condition of the while loop; if this loop command is w then

$$\mathsf{VC}(w, x \neq 0) = x + y = 2 \wedge \forall_{x,y} x + y = 2 \Rightarrow (y\rangle 0 \Rightarrow \mathsf{VC}(c, x + y = 2) \wedge y \leq 0 \Rightarrow x \neq 0)$$

and

$$\mathsf{VC}(y := y - 1; x := x + 1, x + y = 2) = (x + 1) + (y - 1) = 2$$

so

$$\mathsf{VC}(w, x \neq 0) = x + y = 2 \wedge \forall_{x,y} x + y = 2 \Rightarrow (y\rangle 0 \Rightarrow (x+1) + (y-1) = 2 \wedge y \leq 0 \Rightarrow x \neq 0)$$

and finally,

$$\mathsf{VC}(x := 0; y := 2; w, x \neq 0) = 0 + 2 = 2 \wedge \forall_{x,y} x + y = 2 \Rightarrow (y\rangle 0 \Rightarrow (x+1) + (y-1) = 2 \wedge y \leq 0 \Rightarrow x \neq 0).$$

The resulting verification condition is a predicate involcing quantified variables and arithmetic; it can be verified by an automated theorem prover such as Simplify.

Note that simply providing an invalid loop invariant, either one that is too strong (like *false*), or too weak (like *true*) will not introduce an error in this process; the resulting condition will simply be unsatisfiable. VCGen can in fact be shown to be sound by induction on the structure of c, and soundness holds for any choice of loop invariants.

4.3 Forward VCGen

The verification condition is traditionally computed backwards, which works well for structured code, but it can also be computer forwards. This technique works even for unstructured languages (e.g. assembly language) by using symbolic execution. The PREfix tool in use at Microsoft uses this technique.

5 Summary

The axiomatic semantics have been shown to be sound and (relatively) complete. The proof of soundness required a new technique of simultaneous induction. The proof of completeness required developing the concept of a weakest precondition. By shifting the burden of the work of determining invariants over to the program we were able to compute verification conditions that were sound and could be computed in practice.