Axiomatic semantics

Lecture 19 Tuesday, April 4, 2023

1 Introduction to axiomatic semantics

The idea in axiomatic semantics is to give specifications for what programs are supposed to compute. This contrasts with operational model (which show how programs execute) or denotational models (which show what programs compute). Axiomatic semantics defines the meaning of programs in terms of logical formulas satisfied by the program.

This approach to reasoning about programs and expressing program semantics was originally proposed by Floyd and Hoare, and then pushed further by Dijkstra and Gries.

Program specifications can be expressed using pre-conditions and post-conditions:

$$\{Pre\}\ c\ \{Post\}$$

where c is a program, and Pre and $Post$ are logical formulas that describe properties of the program state (usually referred to as assertions). Such a triple is referred to as a *partial correctness statement* (or partial correctness assertion triple) and has the following meaning:

"If Pre holds before c, and c terminates, then $Post$ holds after c."

In other words, if we start with a store σ where Pre holds, and the execution of c in store σ terminates and yields store σ', then $Post$ holds in store σ'.

Pre- and post-conditions can be regarded as interfaces or contracts between the program and its clients. They help users to understand what the program is supposed to yield without needing to understand how the program executes. Typically, programmers write them as comments for procedures and functions, for better program understanding, and to make it easier to maintain programs. Such specifications are especially useful for library functions, for which the source code is, in many cases, unavailable to the users. In this case, pre- and post-conditions serve as contracts between the library developers and users of the library.

However, there is no guarantee that pre- and post-conditions written as informal code comments are actually correct: the comments specify the intent, but give no correctness guarantees. Axiomatic semantics addresses this problem: we will look at how to rigorously describe partial correctness statements and how to prove and reason about program correctness.

Note that partial correctness doesn't ensure that the given program will terminate – this is why it is called "partial correctness". In contrast, total correctness statements ensure that the program terminates whenever the precondition holds. Such statements are denoted using square brackets:

$$[\ Pre\ ]\ c\ [\ Post\ ]$$

meaning:

"If Pre holds before c then c will terminate and $Post$ will hold after c."

In general a pre-condition specifies what the program expects before execution; and the post-conditions specifies what guarantees the program provides when the program terminates. Here is a simple example:

$$\{\mathsf{foo} = 0 \wedge \mathsf{bar} = i\}\ \mathsf{baz} := 0; \textbf{while}\ \mathsf{foo} \neq \mathsf{bar}\ \textbf{do}\ (\mathsf{baz} := \mathsf{baz} - 2; \mathsf{foo} := \mathsf{foo} + 1)\ \{\mathsf{baz} = -2i\}$$

If, before the program executes, the store maps foo to zero, and maps bar to i, then, if the program terminates, then the final store will map baz to $-2i$, that is, -2 times the initial value of bar. Note that i is a logical variable: it doesn't occur in the program, and is just used to express the initial value of bar.

This partial correctness statement is valid. That is, it is indeed the case that if we have any store σ such that $\sigma(\mathsf{foo}) = 0$, and

$$\mathcal{C}[\![\mathsf{baz} := 0; \textbf{while } \mathsf{foo} \neq \mathsf{bar} \textbf{ do } (\mathsf{baz} := \mathsf{baz} - 2; \mathsf{foo} := \mathsf{foo} + 1)]\!]\sigma = \sigma',$$

then $\sigma'(\mathsf{baz}) = -2\sigma(\mathsf{bar})$.

Note that this is a *partial* correctness statement: if the pre-condition is true before c, **and** c **terminates** then the post-condition holds after c. For some initial stores, the program will fail to terminate.

The following total correctness statement is true.

$$[\mathsf{foo} = 0 \wedge \mathsf{bar} = i \wedge i \geq 0] \;\; \mathsf{baz} := 0; \textbf{while } \mathsf{foo} \neq \mathsf{bar} \textbf{ do } (\mathsf{baz} := \mathsf{baz} - 2; \mathsf{foo} := \mathsf{foo} + 1) \;\; [\mathsf{baz} = -2i]$$

That is, if we have a store σ such that $\sigma(\mathsf{foo}) = 0$, and $\sigma(\mathsf{bar})$ is non-negative, then the execution of the command will terminate, and if σ' is the final store then $\sigma'(\mathsf{baz}) = -2\sigma(\mathsf{bar})$.

The following partial correctness statement is not valid. (Why not?)

$$\{\mathsf{foo} = 0 \wedge \mathsf{bar} = i\} \; \mathsf{baz} := 0; \textbf{while } \mathsf{foo} \neq \mathsf{bar} \textbf{ do } (\mathsf{baz} := \mathsf{baz} + \mathsf{foo}; \mathsf{foo} := \mathsf{foo} + 1) \; \{\mathsf{baz} = i\}$$

In the rest of our presentation of axiomatic semantics we will focus exclusively on partial correctness. We will formalize and address the following:

- What logic do we use for writing assertions? That is, what can we express in pre- and post-condition?
- What does it mean that an assertion is valid? What does it mean that a partial correctness statement $\{Pre\}\ c\ \{Post\}$ is valid?
- How can we prove that a partial correctness statement is valid?

1.1 The Language of Assertions

What can we say in the pre- and post-conditions? In the examples we saw above, we used program variables, equality, logical variables (e.g., i), and conjunction ($\wedge$). What we allow in pre- and post-conditions will influence what properties of a program we can describe using partial correctness statements.

The language that we will use for writing assertion is the set of logical formulas that include comparisons of arithmetic expressions, standard logical operators (and, or, implication, negation), as well as quantifiers (universal and existential). Assertions may use additional logical variables, different than the variables that occur in the program. Note that arithmetic expressions may also contain logical variables.

assertions	$P, Q \in$ **Assn**	$P ::= \textbf{true} \mid \textbf{false} \mid a_1 < a_2 \mid a_1 = a_2$
		$\mid P_1 \wedge P_2 \mid P_1 \vee P_2 \mid P_1 \Rightarrow P_2 \mid \neg P$
		$\mid \forall i.\ P \mid \exists i.\ P$
arithmetic expressions	$a \in$ **Aexp**	$a ::= n \mid x \mid a_1 + a_2 \mid a_1 \times a_2 \mid i$
logical variables	$i, j \in$ **LVar**	

One can notice that the domain of boolean expressions **Bexp** is a subset of the domain of assertions. Notable additions over the syntax of boolean expression are quantifiers ($\forall$ and $\exists$). For instance, one can express the fact that variable x divides variable y using existential quantification: $\exists i.\ x \times i = y$.

1.2 Validity of assertions

Now we would like to describe what we mean by "assertion P holds in store σ''. But to determine whether P holds or not, we need more than just the store σ (which maps program variables to their values); we also need to know the values of the logical variables. We describe those values using an interpretation I:

$$I : \textbf{LVar} \to \textbf{Int}$$

Now we can express the validity (or satisfiability) of assertion as a relation

$$\sigma \vDash_I P$$

read as "P is valid in store σ under interpretation I," or "store σ satisfies assertion P under interpretation I." We will write $\sigma \nvDash_I P$ whenever $\sigma \vDash_I P$ doesn't hold.

We proceed to define the validity relation inductively:

$\sigma \vDash_I \textbf{true}$	(always)
$\sigma \vDash_I a_1 < a_2$	if $\mathcal{A}_{\text{Interp}}[\![a_1]\!](\sigma, I) < \mathcal{A}_{\text{Interp}}[\![a_2]\!](\sigma, I)$
$\sigma \vDash_I a_1 = a_2$	if $\mathcal{A}_{\text{Interp}}[\![a_1]\!](\sigma, I) = \mathcal{A}_{\text{Interp}}[\![a_2]\!](\sigma, I)$
$\sigma \vDash_I P_1 \wedge P_2$	if $\sigma \vDash_I P_1$ and $\sigma \vDash_I P_2$
$\sigma \vDash_I P_1 \vee P_2$	if $\sigma \vDash_I P_1$ or $\sigma \vDash_I P_2$
$\sigma \vDash_I P_1 \Rightarrow P_2$	if $\sigma \nvDash_I P_1$ or $\sigma \vDash_I P_2$
$\sigma \vDash_I \neg P$	if $\sigma \nvDash_I P$
$\sigma \vDash_I \forall i.\ P$	if $\forall k \in Int.\ \sigma \vDash_{I[i \mapsto k]} P$
$\sigma \vDash_I \exists i.\ P$	if $\exists k \in Int.\ \sigma \vDash_{I[i \mapsto k]} P$

The evaluation function $\mathcal{A}_{\text{Interp}}[\![a]\!]$ is similar to the denotation of expressions, but also deals with logical variables:

$$\begin{aligned}
\mathcal{A}_{\text{Interp}}[\![n]\!](\sigma, I) &= n \\
\mathcal{A}_{\text{Interp}}[\![x]\!](\sigma, I) &= \sigma(x) \\
\mathcal{A}_{\text{Interp}}[\![i]\!](\sigma, I) &= I(i) \\
\mathcal{A}_{\text{Interp}}[\![a_1 + a_2]\!](\sigma, I) &= \mathcal{A}_{\text{Interp}}[\![a_1]\!](\sigma, I) + \mathcal{A}_{\text{Interp}}[\![a_2]\!](\sigma, I)
\end{aligned}$$

We can now say that an assertion P is valid (written $\vDash P$) if it is valid in any store, under any interpretation: $\forall \sigma, I.\ \sigma \vDash_I P$.

Having defined validity for assertions, we can now define the validity of partial correctness statements. We say that $\{P\}\ c\ \{Q\}$ is valid in store σ and interpretation I, written $\sigma \vDash_I \{P\}\ c\ \{Q\}$, if:

$$\forall \sigma'.\ \text{if } \sigma \vDash_I P \text{ and } \mathcal{C}[\![c]\!]\sigma = \sigma' \text{ then } \sigma' \vDash_I Q$$

Note that this definition talks about the execution of program c in the initial store σ, described using the denotation $\mathcal{C}[\![c]\!]$.

Finally, we can say that a partial correctness triple is valid (written $\vDash \{P\}\ c\ \{Q\}$), if it is valid in any store and interpretation:

$$\forall \sigma, I.\ \sigma \vDash_I \{P\}\ c\ \{Q\}.$$

Now we know what we mean when we say "assertion P holds" or "partial correctness statement $\{P\}\ c\ \{Q\}$ is valid."

1.3 Hoare logic and program correctness

How do we show that a partial correctness statement $\{P\}\ c\ \{Q\}$ holds? We know that $\{P\}\ c\ \{Q\}$ is valid if it holds for all stores and interpretations: $\forall \sigma, I.\ \sigma \vDash_I \{P\}\ c\ \{Q\}$. Furthermore, showing that $\sigma \vDash_I \{P\}\ c\ \{Q\}$ requires reasoning about the execution of command c (that is, $\mathcal{C}[\![c]\!]$), as indicated by the definition of validity.

It turns out that there is an elegant way of deriving valid partial correctness statements, without having to reason about stores, interpretations, and the execution of c. We can use a set of inference rules and axioms, called *Hoare* rules, to directly derive valid partial correctness statements. The set of rules forms a proof system known as Hoare logic.

$$\textsc{Skip}\ \frac{}{\vdash \{P\}\ \textbf{skip}\ \{P\}} \qquad \textsc{Assign}\ \frac{}{\vdash \{P[a/x]\}\ x := a\ \{P\}}$$

$$\textsc{Seq}\ \frac{\vdash \{P\}\ c_1\ \{R\} \qquad \vdash \{R\}\ c_2\ \{Q\}}{\vdash \{P\}\ c_1; c_2\ \{Q\}} \qquad \textsc{If}\ \frac{\vdash \{P \wedge b\}\ c_1\ \{Q\} \qquad \vdash \{P \wedge \neg b\}\ c_2\ \{Q\}}{\vdash \{P\}\ \textbf{if}\ b\ \textbf{then}\ c_1\ \textbf{else}\ c_2\ \{Q\}}$$

$$\textsc{While}\ \frac{\vdash \{P \wedge b\}\ c\ \{P\}}{\vdash \{P\}\ \textbf{while}\ b\ \textbf{do}\ c\ \{P \wedge \neg b\}}$$

Note that in the rule for assignment, we write $P[a/x]$ for the predicate P where every occurrence of the program variable x is replaced with the expression a.

The assertion P in the rule for while loops is essentially a loop invariant; it is an assertion that holds before and after each iteration, as shown in the premise of the rule. Therefore, it is both a pre-condition for the loop (because it holds before the first iteration); and also a post-condition for the loop (because it holds after the last iteration). The fact that P is both a pre- and post-condition for the while loop is reflected in the conclusion of the rule.

There is one more rule, the rule of consequence, which allows to strengthen pre-conditions and weaken post-conditions:

$$\textsc{Consequence}\ \frac{\vDash (P \Rightarrow P') \qquad \vdash \{P'\}\ c\ \{Q'\} \qquad \vDash (Q' \Rightarrow Q)}{\vdash \{P\}\ c\ \{Q\}}$$

These set of Hoare rules represent an inductive definition for a set of partial correctness statements $\{P\}\ c\ \{Q\}$. We will say that $\{P\}\ c\ \{Q\}$ is a theorem in Hoare logic, written $\vdash \{P\}\ c\ \{Q\}$, if we can build a finite proof tree for it.

1.4 Soundness and Completeness

At this point we have two kinds of partial correctness assertions:

a) valid partial correctness statements $\vDash \{P\}\ c\ \{Q\}$, which hold for all stores and interpretations, according to the semantics of c; and

b) Hoare logic theorems $\vdash \{P\}\ c\ \{Q\}$, that is, a partial correctness statement that can be derived using Hoare rules.

The question is how do these sets relate to each other? More precisely, we have to answer two questions. First, is each Hoare logic theorem guaranteed to be valid partial correctness triple? In other words,

$$\text{does} \quad \vdash \{P\}\ c\ \{Q\} \quad \text{imply} \quad \vDash \{P\}\ c\ \{Q\}?$$

The answer is yes, and it shows that Hoare logic is sound. Soundness is important because it says that Hoare logic doesn't allow us to derive partial correctness assertions that actually don't hold. The proof of soundness requires induction on the derivations in $\vdash \{P\}\ c\ \{Q\}$ (but we will omit this proof).

The second question refers to the expressiveness and power of Hoare rules: can we always build a Hoare logic proof for each valid assertion? In other words,

$$\text{does} \quad \vDash \{P\}\ c\ \{Q\} \quad \text{imply} \quad \vdash \{P\}\ c\ \{Q\}?$$

The answer is a qualified yes: if $\vDash \{P\}\ c\ \{Q\}$ then there is a proof of $\{P\}\ c\ \{Q\}$ using the rules of Hoare logic, provided there are proofs for the validity of assertions that occur in the rule of consequence $\vDash (P \Rightarrow P')$ and $\vDash (Q' \Rightarrow Q)$. This result is known as the *relative completeness of Hoare logic* and is due to Cook (1978); the proof is fairly complex and we will omit it (but you can find details in Winskel).