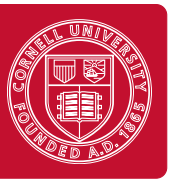
CS 4110 – Programming Languages and Logics
Lecture #10: Axiomatic Semantics



1 Introduction to axiomatic semantics

Now we turn to the third and final main style of semantics, *axiomatic semantics*. The idea in axiomatic semantics is to define meaning in terms of logical specifications that programs satisfy. This is in contrast to operational models (which show *how* programs execute) and denotational models (which show *what* programs compute). This approach to reasoning about programs and expressing program semantics was originally proposed by Floyd and Hoare and was developed further by Dijkstra and Gries.

A common way to express program specifications is in terms of pre-conditions and post-conditions:

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

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

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

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

Pre-conditions 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 as documentation and to make it easier to maintain programs. Such specifications are especially useful for library functions for which the source code is often not available to the users. In this case, pre-conditions and post-conditions serve as contracts between the library developers and users of the library.

However, there is no guarantee that pre-conditions and post-conditions written informally in comments are correct: the comments describe the intent of the developer, but they do not give a guarantee of correctness. Axiomatic semantics addresses this problem. It shows how to rigorously describe partial correctness statements and how to establish correctness using formal reasoning.

Note that partial correctness specifications don't ensure that the program will terminate—this is why they are called "partial". 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 (if the program terminates). Here is a simple example:

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

It says that if the store maps foo to 0 and bar to i before execution, then, if the program terminates, the final store will map baz to $-2i$ (i.e., -2 times the initial value of bar). Note that i is a logical variable that doesn't occur in the program and is only used to express the initial value of bar. Such variables are sometimes called *ghost variables*.

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. There are some initial stores for which the program will not 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 start with a store σ that maps foo to 0 and bar to a non-negative integer, then the execution of the command will terminate in a final store σ' with $\sigma'(\mathsf{baz}) = -2\sigma(\mathsf{bar})$.

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 discussion of axiomatic semantics we will focus almost exclusively on partial correctness assertions.

2 Assertions

Now we turn to the following issues:

- What logic do we use for writing assertions? That is, what can we express in pre-conditions 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?

What can we say in pre-conditions and post-conditions? In the examples so far, we have used program variables, equality, logical variables (e.g., i), and conjunction ($\wedge$). What we allow in pre-conditions and post-conditions directly influences the sorts of program properties we can describe using partial correctness statements. We will use the set of logical formulas including comparisons between arithmetic expressions,

standard logical operators (and, or, implication, negation), as well as quantifiers (universal and existential). Assertions may also introduce logical variables that are different than the variables appearing in the program.

$$
\begin{aligned}
i, j &\in \mathbf{LVar} \\
a \in \mathbf{Aexp} &::= x \mid i \mid n \mid a_1 + a_2 \mid a_1 \times a_2 \\
P, Q \in \mathbf{Assn} &::= \mathsf{true} \mid \mathsf{false} \mid a_1 < a_2 \mid P_1 \wedge P_2 \mid P_1 \vee P_2 \mid P_1 \Rightarrow P_2 \mid \neg P \mid \forall i.\, P \mid \exists i.\, P
\end{aligned}
$$

Observe that the domain of boolean expressions $\mathbf{Bexp}$ 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: $\exists i.\, x \times i = y$.

3 Satisfaction and Validity

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,

$$I : \mathbf{LVar} \to \mathbf{Int},$$

and define the function $\mathcal{A}_i[\![a]\!]$, which is like the denotation of expressions extended to logical variables in the obvious way:

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

Now we can express the satisfiability of assertions as a relation $\sigma \vDash_I P$ read as "P is satisfied 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.

$\sigma \vDash_I \mathsf{true}$	(always)
$\sigma \vDash_I a_1 < a_2$	if $\mathcal{A}_i[\![a_1]\!](\sigma, I) < \mathcal{A}_i[\![a_2]\!](\sigma, I)$
$\sigma \vDash_I P_1 \wedge P_2$	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 $s \nvDash_I P_1$ or $\sigma \vDash_I P_2$
$\sigma \vDash_I \neg P$	if $s \nvDash_I P$
$\sigma \vDash_I \forall i.\, P$	if $\forall k \in Int.\, \sigma \vDash_{I[i \mapsto k]} P$
$\sigma \vDash_I \exists i.\, P$	if $\exists k \in Int.\, \sigma \vDash_{I[i \mapsto k]} P$

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 individual assertions, we now turn to partial correctness statements. We say that a partial correctness statement $\{P\}\ c\ \{Q\}$ is satisfied 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 depends on the execution of c in the initial store σ.

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."