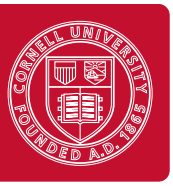
CS 4110 – Programming Languages and Logics
Lecture #7: Denotational → Axiomatic Semantics



Based on material by Stephen Chong, Greg Morrisett, Andrew Myers, George Necula, and Radu Rugina

1 More denotational semantics

Returning to our original problem from the last lecture... our goal is to find a partial function from **Store** to **Stores** to serve as the denotation of loops **while** b **do** c. We will express $\mathcal{C}[\![\textbf{while } b \textbf{ do } c]\!]$ as the fixed point of a higher-order function F:

$$\begin{aligned} F &\in (\mathbf{Store} \rightharpoonup \mathbf{Store}) \to (\mathbf{Store} \rightharpoonup \mathbf{Store}) \\ F(f) &= \{(\sigma,\sigma) \mid (\sigma, \textbf{false}) \in \mathcal{B}[\![b]\!]\} \cup \\ &\qquad \{(\sigma,\sigma') \mid (\sigma, \textbf{true}) \in \mathcal{B}[\![b]\!] \wedge \exists \sigma''.\ ((\sigma,\sigma'') \in \mathcal{C}[\![c]\!] \wedge (\sigma'',\sigma') \in f)\} \end{aligned}$$

Compare this definition with the recursive equation for $\mathcal{C}[\![\textbf{while } b \textbf{ do } c]\!]$ we wrote last time:

$$\begin{aligned} \mathcal{C}[\![\textbf{while } b \textbf{ do } c]\!] = &\{(\sigma,\sigma) \mid (\sigma, \textbf{false}) \in \mathcal{B}[\![b]\!]\} \cup \\ &\{(\sigma,\sigma') \mid (\sigma, \textbf{true}) \in \mathcal{B}[\![b]\!] \wedge \exists \sigma''.\ ((\sigma,\sigma'') \in \mathcal{C}[\![c]\!] \wedge (\sigma'',\sigma') \in \mathcal{C}[\![\textbf{while } b \textbf{ do } c]\!])\} \end{aligned}$$

The higher-order function F takes a partial function f and computes a single iteration of the loop. However, rather than invoking itself when it needs to go around the loop again, it calls f instead.

We define the denotation of the loop to be the least fixed point of F:

$$\begin{aligned} \mathcal{C}[\![\textbf{while } b \textbf{ do } c]\!] &= \bigcup_{i \geq 0} F^i(\emptyset) \\ &= \emptyset \cup F(\emptyset) \cup F(F(\emptyset)) \cup F(F(F(\emptyset))) \cup \ldots \\ &= \text{fix}(F) \end{aligned}$$

For this definition to make sense, the least fixed point $\text{fix}(F)$ must exist and capture the function we intuitively expect. The reason this works is that F (along with all of the other operations used to define our denotational semantics) is *monotone*, in that $f \subseteq g$ implies $F(f) \subseteq F(g)$, and *continuous*, in that for every chain $f_0 \subseteq f_1 \subseteq \ldots$ we have that $F(\bigcup_{i \geq 0} f_i) = \bigcup_{i \geq 0} F(f_i)$. Explaining these details further requires delving into *domain theory*, which is unfortunately beyond the scope of this course. (If you're interested in understanding them, come take CS 6110 in the spring!)

1.1 Example

To see how this works, let's work out the denotation of **while** foo $<$ bar **do** foo $:=$ foo $+ 1$.

$$\begin{aligned} F(\emptyset) &= \{(\sigma,\sigma) \mid (\sigma, \textbf{false}) \in \mathcal{B}[\![b]\!]\} \cup \\ &\qquad \{(\sigma,\sigma') \mid (\sigma, \textbf{true}) \in \mathcal{B}[\![b]\!] \wedge \exists \sigma''.\ ((\sigma,\sigma'') \in \mathcal{C}[\![c]\!] \wedge (\sigma'',\sigma') \in \emptyset)\} \\ &= \{(\sigma,\sigma) \mid \sigma(\mathsf{foo}) \geq \sigma(\mathsf{bar})\} \end{aligned}$$

$$
\begin{aligned}
F^2(\emptyset) &= \{(\sigma,\sigma) \mid (\sigma, \textbf{false}) \in \mathcal{B}[\![b]\!]\} \cup \\
&\quad \{(\sigma,\sigma') \mid (\sigma, \textbf{true}) \in \mathcal{B}[\![b]\!] \wedge \exists \sigma''.\ ((\sigma,\sigma'') \in \mathcal{C}[\![c]\!] \wedge (\sigma'',\sigma') \in F(\emptyset))\} \\
&= \{(\sigma,\sigma) \mid \sigma(\mathsf{foo}) \geq \sigma(\mathsf{bar})\} \cup \\
&\quad \{(\sigma, \sigma[\mathsf{foo} \mapsto \sigma(\mathsf{foo}) + 1]) \mid \sigma(\mathsf{foo}) < \sigma(\mathsf{bar}) \wedge \sigma(\mathsf{foo}) + 1 \geq \sigma(\mathsf{bar})\}
\end{aligned}
$$

But if $\sigma(\mathsf{foo}) < \sigma(\mathsf{bar}) \wedge \sigma(\mathsf{foo}) + 1 \geq \sigma(\mathsf{bar})$ then $\sigma(\mathsf{foo}) + 1 = \sigma(\mathsf{bar})$, so we can simplify further:

$$
\begin{aligned}
&= \{(\sigma,\sigma) \mid \sigma(\mathsf{foo}) \geq \sigma(\mathsf{bar})\} \cup \\
&\quad \{(\sigma, \sigma[\mathsf{foo} \mapsto \sigma(\mathsf{foo}) + 1]) \mid \sigma(\mathsf{foo}) < \sigma(\mathsf{bar}) \wedge \sigma(\mathsf{foo}) + 1 = \sigma(\mathsf{bar})\}
\end{aligned}
$$

$$
\begin{aligned}
F^3(\emptyset) &= \{(\sigma,\sigma) \mid (\sigma, \textbf{false}) \in \mathcal{B}[\![b]\!]\} \cup \\
&\quad \{(\sigma,\sigma') \mid (\sigma, \textbf{true}) \in \mathcal{B}[\![b]\!] \wedge \exists \sigma''.\ ((\sigma,\sigma'') \in \mathcal{C}[\![c]\!] \wedge (\sigma'',\sigma') \in F^2(\emptyset))\} \\
&= \{(\sigma,\sigma) \mid \sigma(\mathsf{foo}) \geq \sigma(\mathsf{bar})\} \cup \\
&\quad \{(\sigma, \sigma[\mathsf{foo} \mapsto \sigma(\mathsf{foo}) + 1]) \mid \sigma(\mathsf{foo}) < \sigma(\mathsf{bar}) \wedge \sigma(\mathsf{foo}) + 1 = \sigma(\mathsf{bar})\} \cup \\
&\quad \{(\sigma, \sigma[\mathsf{foo} \mapsto \sigma(\mathsf{foo}) + 2]) \mid \sigma(\mathsf{foo}) < \sigma(\mathsf{bar}) \wedge \sigma(\mathsf{foo}) + 2 = \sigma(\mathsf{bar})\}
\end{aligned}
$$

$$
\begin{aligned}
F^4(\emptyset) &= \{(\sigma,\sigma) \mid \sigma(\mathsf{foo}) \geq \sigma(\mathsf{bar})\} \cup \\
&\quad \{(\sigma, \sigma[\mathsf{foo} \mapsto \sigma(\mathsf{foo}) + 1]) \mid \sigma(\mathsf{foo}) < \sigma(\mathsf{bar}) \wedge \sigma(\mathsf{foo}) + 1 = \sigma(\mathsf{bar})\} \cup \\
&\quad \{(\sigma, \sigma[\mathsf{foo} \mapsto \sigma(\mathsf{foo}) + 2]) \mid \sigma(\mathsf{foo}) < \sigma(\mathsf{bar}) \wedge \sigma(\mathsf{foo}) + 2 = \sigma(\mathsf{bar})\} \cup \\
&\quad \{(\sigma, \sigma[\mathsf{foo} \mapsto \sigma(\mathsf{foo}) + 3]) \mid \sigma(\mathsf{foo}) < \sigma(\mathsf{bar}) \wedge \sigma(\mathsf{foo}) + 3 = \sigma(\mathsf{bar})\}
\end{aligned}
$$

If we take the union of all $F^i(\emptyset)$, we get the expected semantics of the loop.

$$
\begin{aligned}
&\mathcal{C}[\![\textbf{while } \mathsf{foo} < \mathsf{bar} \textbf{ do } \mathsf{foo} := \mathsf{foo} + 1]\!] \\
&\qquad \{(\sigma,\sigma) \mid \sigma(\mathsf{foo}) \geq \sigma(\mathsf{bar})\} \cup \\
&\qquad \{(\sigma, \sigma[\mathsf{foo} \mapsto \sigma(\mathsf{foo}) + n]) \mid \sigma(\mathsf{foo}) < \sigma(\mathsf{bar}) \wedge \sigma(\mathsf{foo}) + n = \sigma(\mathsf{bar})\}
\end{aligned}
$$

2 Introduction to axiomatic semantics

Now we turn to the third and last 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 exclusively on partial correctness, addressing 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?