There are two main problems with the operational approach to semantics that we have studied. One is that the rules we used are very coarse in their application. All we were really able to do was to discuss the broad progress of the program through its state transitions, and we were not able to discuss things at any finer level of detail. That is why it is called “big step” semantics. There is a finer application of the method called “small step” semantics, but both suffer from a similar defect that the form of the rules are too closely bound up with the syntax; one of the goals of formal semantics is a clear separation of syntax and semantics. A minor point is that we were forced to use a mixture of two representations: logic for the rules, and functions for program state and state change. I would be nicer to have a uniform language across the whole system. The good consequence of this would be simpler proofs and it would lead to a more useful system overall. Interestingly, the other two main methods lead us to the extremes. Denotational semantics are based purely on sets and functions, whereas axiomatic semantics are based purely on logic. One of major tasks is to show that the three methods are equivalent and lead to similar inferences about general programs. We shall not do that in this course, but it has been done by many people in many different ways. Really it comes down to personal preference which method to pick, but it nice at least to have a choice.

We start, as before, talking about program state and changes to it. In operational semantics we were obliged to model the state explicitly as a function and to use function update to change it; any semantic method needs a way to handle changes of state. The store modeled as a function will reappear in denotational semantics, but the axiomatic method takes a different approach. Instead of using explicit modeling, we will use an implicit model, where the store is described through a set of constraints on the values stored in it.

A constraint on a program state is an assertion about the values of variables in the state. The language of assertions is logic (with a bit of arithmetic) and then instead of writing \([x \mapsto 1]\) to indicate that \(x\) is bound to 1, we might write simply (it is true that) \(x = 1\). This is more like an equation than an assignment; it constrains \(x\) to be 1. Other constraints might be \(x > 1\) or even \(x > y + 1\), where the values of \(x\) and \(y\) might be unknown, but their values are constrained by the inequality. Assertions about program state do not model values and bindings explicitly, they only give looser constraints on what those values might be. The usefulness of this approach will only become apparent when we look at the details, but briefly, assertions can be used to annotate a program, and the truth of the assertions checked. In particular, let us write an assertion for the final state of a program, when it terminates. This might look this (actually this is the notation we shall use):

\[
\{a_{\text{initial}}\} P \{a_{\text{final}}\}
\]

\(P\) stands for the program (the actual source code). \(a_{\text{initial}}\) and \(a_{\text{final}}\) are assertions. The initial assertion might be empty (which is always true) but might contain initial constraints on the variables in the program. The final assertion is the interesting one, however. This assertion could tell us what the program is intended to do. We will call this the specification of the program. As a simple example consider a program that adds a list of numbers to produce a total.

The specification of this program might be \(\{\text{total} = \sum_i x_i\}\) where the \(x_i\)'s are the elements of an array. The initial constraint might be \(\{x_0 = 5 \land x_1 = 12 \land \ldots\}\). Now we make a very interesting statement which is at the heart of axiomatic semantics. Assuming the initial assertion is true and the program runs to completion (i.e. it terminates) then the final assertion must be true. In logic this is:

\[a_{\text{initial}} \land \text{terminates}(P) \Rightarrow a_{\text{final}}\]

Of course this statement will only be true if both the left hand and the right hand of the implication are true. (We will not be interested when the program does not terminate, when \(a_{\text{final}}\) could be anything and the implication would still be true.) The question is how do we know if the implication is true? The answer is if we can choose the right assertions, and write the program correctly, then we can show that the implication is true. To put it another way, taking program \(P\), and the two assertions, if we can prove that the implication is true, using only the rules of logic, then we have shown that the program is correct relative to the two assertions. This is the great utility of this style of semantics.
It has long been a goal to give programmers tools such that they can write correct programs. Now at least we have the possibility that, having written the program, we can check whether it is correct or not, by trying to prove the implication above. Unfortunately, it is well known that just because a proof exists, it does not mean that we can find that proof; we may be too stupid to find it. However, if we do find a proof, then it is a rock solid certainty that the program is correct. This is a substantial improvement on the traditional ways of testing programs, which can never prove correctness. One word of warning; we may think we have a proof, but we did not follow the rules correctly, so, in fact, we have proved nothing. The history of mathematics is littered with ‘proofs’ that were later shown to be flawed. Luckily the proofs we will look at are much simpler than the famous unsolved problems in math (e.g. Fermat’s last theorem).

In order to achieve the goal we will need three things:

1. a language in which to express assertions about program state
2. a way to break down the program into its parts and make assertions about those parts (actually about the program state between commands)
3. a technique to take 1 and 2 and build a proof using only axioms and rules of inference

Of course, we need to be as formal as possible in order to have the certainty about proofs. Central to this is the discovery of axioms to help us with annotation of program parts. It was the discovery of these by Hoare, and others, that led to the deep study of logic applied to programs.