Modeling Programming Languages Formally

CPSC 509: Programming Language Principles

Ronald Garcia∗

8 January 2014

(Time Stamp: 13:13, Monday 12th September, 2016)

This course focuses on the design and analysis of programming languages. One of our key tools for this endeavor will be mathematics, though not in the sense of arithmetic or calculus, like you see in high school or early university studies. Instead we dive deeper into the foundations of mathematics: logic and set theory. These topics often appear in undergraduate discrete mathematics courses or theory of computation courses for computer science students. Don’t worry too much if you haven’t taken one of these courses, we’ll build up the necessary material as we go along. We will model programs as mathematical objects in set theory, and programming languages as sets of programs and their meanings. In these notes, we start small: ridiculously small. Our first programming language gives us an opportunity to introduce the most basic concepts. We’ll build on these concepts for the remainder of the course.

A side-note: below you will see some mathematical machinery (e.g., sets, propositions, and proofs) introduced without much explanation. Don’t be frightened: we will cover these in more depth in class. I expect that students in this class will come from a variety of backgrounds, so we will take time in the course to get you up to speed. My goal here is to expose you to the material early and then work to improve your understanding in class and through exercises.

1 How Do We Model Programming Languages Mathematically?

Consider the following transcript of interacting with an extremely simplistic programming language, which we’ll call Vapid version 0.0

Vapid Programming Language v0.0

> 1
2
> 2
1

This language has only two programs: 1, and 2. The next version of this language, Vapid version 1.0, adds a new program, 3:

> 3

However this program doesn’t ever produce an answer: it just hangs. This is different from trying to run, say 4:

> 4
Error: bad program.
While 3 is a well-formed program with defined behavior—nontermination—4 is not a program whatsoever, so has no defined behaviour.

Our last version of Vapid, version 2.0, adds one last program, 5 (we skip 4 because that’s a terrible idea for a program, amiright? 😎):

>5
Exception

Wait a second! If I try to run 4 I get that it’s a bad program, but if I run 5 I get an exception: what’s the difference?? Good question! It’s hard to tell the difference (except for the message) in an interpreter, because it simultaneously:

- Decides if the program is well-formed (i.e., a legal, meaningful, program)
- Evaluates it to produce some result.

We can more easily see the difference between these two behaviours if we write a compiler for Vapid 2.0. We’ll call it vcc. The relevant interactions then look this:

```shell
home> vcc -o one one.vpd
done
home> ./one
2
done
home> vcc -o two two.vpd
done
home> ./two
1
done
home> vcc -o three three.vpd
done
home> ./three
^C
done
home> vcc -o four four.vpd
done
four.vpd:1:1: error: bad program
4
done
home> ./four
-bash: ./four: No such file or directory
done
home> vcc -o five five.vpd
done
home> ./five
Exception
done
```

The vcc Vapid compiler separates checking for well-formedness from execution. Here we see that every program in Vapid 2.0 compiles successfully and runs. However, the program 4 stored in four.vpd does not compile because it is not meaningful. The program 5 compiles fine, but it throws an exception.

Hopefully these examples, even in such a vapid context, can give you an idea of some of the subtleties that arise (and that you as a PL theorist must keep in mind) when building and analyzing models of programming languages, whether you are designing a whole new language, or analyzing an existing language that appears in the wild (and there are plenty of untamed languages out there!).

### 1.1 Semantics for the Vapids

These two languages give us a chance to introduce the basic mathematical framework for specifying languages. A language is specified in three parts:

1. A set of legal programs;
2. A set of possible observable results;
3. A mapping from programs to observable results, which we’ll call its evaluator.
Let’s dive right in! Here is a definition of the Vapid 0.0 language in this framework:

\[
\begin{align*}
\text{PGM} &= \{ 1, 2 \} \\
\text{OBS} &= \{ 1, 2 \} \\
\mathit{eval} : & \text{PGM} \rightarrow \text{OBS} \\
\mathit{eval}(1) &= 2 \\
\mathit{eval}(2) &= 1
\end{align*}
\]

We can immediately define its evaluator by cases (i.e., we list out each input-output pair, like a lookup table) since the language is finite (i.e., has a finite number of programs). In fact, we could have equivalently written the definition down literally as a table, which makes it quite evident that \( \mathit{eval} \) is just a set of pairs:

\[
\mathit{eval} = \{ \langle 2, 1 \rangle, \langle 1, 2 \rangle \}.
\]

A hint: every \( \mathit{eval} \) function you see in this course (in fact every function whatsoever!) will literally be such a lookup table. It just so happens that we cannot always define them (i.e., write them down) that way, especially if there are an infinite number of entries. More on that later.

The Vapid 1.0 language is not much different:

\[
\begin{align*}
\text{PGM} &= \{ 1, 2, 3 \} \\
\text{OBS} &= \{ 1, 2, \infty \} \\
\mathit{eval} : & \text{PGM} \rightarrow \text{OBS} \\
\mathit{eval}(1) &= 2 \\
\mathit{eval}(2) &= 1 \\
\mathit{eval}(3) &= \infty
\end{align*}
\]

Here we are modeling the evaluator as a function again, with the added program 3 and representing the possibility of nontermination with the symbol \( \infty \). Compare the status of 3 to that of 4, which is not in the set PGM so not a legal program.

It’s worth noting that in many works on PL theory, you would see the same language modeled as follows:

\[
\begin{align*}
\text{PGM} &= \{ 1, 2, 3 \} \\
\text{OBS} &= \{ 1, 2 \} \\
\mathit{eval} : & \text{PGM} \rightarrow \text{OBS} \\
\mathit{eval}(1) &= 2 \\
\mathit{eval}(2) &= 1 \\
\mathit{eval}(3) &\uparrow
\end{align*}
\]

In this version, the definition of the \( \mathit{eval} \) function is exactly as in Vapid 0.0: the difference is that 3 is a program now, but the evaluator is undefined for it (denoted by an upward arrow). In fact, given the definition of PGM here, the three \( \mathit{eval} \) equations are again equivalent to \( \mathit{eval} = \{ \langle 2, 1 \rangle, \langle 1, 2 \rangle \} \). We’ll discuss later how to show that this is true: for now just trust me. In short, 3 is now a program but it’s observable result is “undefined” so \( \mathit{eval} \) is a partial function, since it is undefined for 3. This is indicated using a harpoon \( \rightarrow \). In our preferred definition above, \( \mathit{eval} \) is a total function, meaning that it is defined for every possible program.

Comparing these two plausible models of Vapid 1.0, you could say that each has its benefits and shortcomings. The total-function model forces us to provide a definition for every possible program; in contrast, the partial function model lets us simply leave out programs that don’t terminate. On the other hand, the total-function model forces us to account for all programs: we need not worry that we accidentally defined some program to diverge (fancy word for “not terminate”) by simply forgetting to define it. As language semantics get more complex, this becomes a real problem (we’ll see this later). As such, we’ll prefer the total-function models in this class: they force us to be precise and clear (but more long-winded) about our intent.

\[1\] and in fact in previous versions of these course notes
1.1.1 Terminology: Syntax and Semantics

In the general case, where there are more than 3 programs, we resort to more sophisticated tools to describe the set of programs. Especially when there are an infinite number of programs, we describe them in terms of repeating structure. We call this structure the *syntax* of programs.

Similarly, given more complex programs, we need to resort to more complicated means of defining the evaluator, usually in terms of the syntax of programs (plus some extra bits as the language gets more sophisticated). We call this general structured description of behaviour the *semantics* of the language. Semantics is just a fancy word for “meaning”.

Sometimes you will see papers, books, etc. refer to static semantics and dynamic semantics of a language. In general dynamic semantics refers to the behaviour of programs (what I call “semantics” above). Static semantics typically refers to some aspects of what I call “syntax” above: those things that determine what counts as a legal program. However there are some subtleties involved which make the term “static semantics” make some sense. We’ll get into that later in the course. For these notes I will stick with “syntax” and “semantics” as described above.

**Exercise 1.** Write a formal model of Vapid 2.0. Even if you think you get the idea in the abstract, I recommend writing it down so that you have some practice writing down all of the details from scratch by yourself.

## 2 How do we Reason about our Models?

Now we have precise formal models of some programming languages: whoop-dee-doo! Or rather, what do we do with them? Well, one of the key things we can do is reason (formally) about the properties of programs in our language, and the language itself!

First, let’s prove a property of a single program. Brace yourself:

**Proposition 1.** \(\text{eval}(2) = 1\).

*Proof.* According to the definition of \(\text{eval}\), \(\text{eval}(2) = 1\).

First proof of the course, let’s celebrate! Now, that may not have involved much work, but in the general case, determining the result of a program is quite important. This is one way that we can help validate that our implementation of the language is correct. There are plenty of arguments on the internet about what some program in some language would do, based on “well my implementation does this, so that’s what it should do,” rather than appealing to the formal definition of the language to figure out whether there might be an inconsistency between the spec and the implementation.\(^2\)

Now that we’ve proven a property of a single program in a language, let’s consider a property of the language itself.

**Proposition 2.** There is (i.e., there exists) a Vapid 1.0 program that diverges.\(^3\)

*Proof.* Consider the program 3. Then \(\text{eval}(3) = \infty\), which represents divergence.

This proof is a little different from the previous one. The statement of the theorem says essentially that “somewhere out there, in the great big world of programs, there’s a program that runs forever.” To prove this statement, we offer up a program and then show that, yes indeed, it diverges! In the proof of the last theorem, we could just follow our noses (consult \(\text{eval}\)) and be done. In this case, we needed a little human insight to find a candidate program that satisfies the property. This is the nature of existence proofs...we pull the witness for the proof essentially out of thin air. Now to find that witness, we may have done a bunch of work off on the side (i.e. check out all the programs, make hunches and throw them at the Vapid interpreter, etc.), but that empirical work doesn’t show up in the proof.

Now let’s prove a property of all programs in a language. Technically for each of our semantics, we are on the hook to prove that it fully defines the language. Let’s do so for one of them:

\(^2\) I say inconsistency because in the real world, which one is right/wrong is a social problem, not a technical problem. Throughout the course, we’ll conveniently assume that our formal semantics is the gold standard, but when reverse-engineering a formal semantics from a language without one, e.g., that position is likely the wrong one to take.

\(^3\) In formal notation, \(\exists p \in P:\text{eval}(p) = \infty\).
Proposition 3. In Vapid 1.0, \( \text{eval} \) is a total function from \( P_{GM} \) to \( O_{BS} \).

Earlier we just “said” that this was true, but we didn’t prove it. In describing the semantics of Vapid 1.0 we discussed the downsides of modeling the evaluator as a partial function, and not being able to prove this proposition formally is part of it. Even though it may seem clear “by inspection”, let’s prove the “total” part, that the evaluator is defined for every input program. The “function” part establishes that the language is deterministic, but we’ll ignore that for now because we can (1). First let’s state the proposition more precisely then prove it:

Proposition 4. For every \( p \in P_{GM} \), there is some \( o \in O_{BS} \) such that \( \text{eval}(p) = o \).\(^4\)

Proof. Suppose \( p \in P_{gm} \). Then we proceed by cases on \( p \in P_{gm} \).

Case \((p = 1)\). Then \( \text{eval}(1) = 2 \) and \( 2 \in O_{BS} \),
Case \((p = 2)\). Then \( \text{eval}(2) = 1 \) and \( 1 \in O_{BS} \),
Case \((p = 3)\). Then \( \text{eval}(3) = \infty \) and \( \infty \in O_{BS} \),

Phew! Here we proved the theorem by exhaustively considering each and every program and then showed that we could evaluate each one. We could do this specifically because of the structure of our definition of \( P_{GM} \).

One of the main themes of this course is that the way you structure your definitions affects the structure of your reasoning about them.

In this case, a definition of programs “by cases” (listing the elements of the set \( P_{GM} \)) gives us the ability to prove things about all programs by cases (checking each one). Later when we have languages with an infinite number of programs, that’s not going to work (we’ll get hungry well before we finish).

In this particular example, it’s a single step of reasoning in each case to find the observable result of each program, and that’s great: the broader implications of our language design are right before us. In fact, the structure of \( \text{eval} \)’s definition gives plenty of guidance to language implementors. We’ll see this come up again and again even as our languages become less vapid.

Now consider how we used \( \text{eval} \). We defined \( \text{eval} \) a couple of ways: using equations, and explicitly listing the set of pairs. Within the proof, we took advantage of the equations. Thus the structure of the definition mattered here too. For illustration, let’s consider how the cases might have looked if we used the “set of pairs” definition of \( \text{eval} \):

Case \((p = 1)\). Then \( \text{eval}(1) = n \) only if \( (1, n) \in \text{eval} \) for some \( n \). By definition of \( \text{eval} \), \( (1, 2) \in \text{eval} \) so \( \text{eval}(1) = 2 \).

This proof case is in some ways pedantic, in the sense that we often shift between the view of a function as a list of entries and as a “function” in the sense of a black box that takes inputs and spits out outputs. But it’s useful to get some sense of how definition structure can affect your deductions. We’ll clarify this more later in cases where it seems less pedantic.

So to summarize, for this proposition we proved a property of all programs (that they all evaluate) by reasoning over all programs, and in each case we take advantage of the equations that we used to define \( \text{eval} \). Let’s take a step back and reconsider this idea of proving a property of all programs. We could prove other properties of all programs in roughly the same way, but substituting a different property into our reasoning. In this way we have a general reasoning principle for proving properties that hold of all Vapid 1.0 programs:

Proposition 5 (Principle of Cases on \( p \in P_{GM} \)). Let \( P \) be a property of vapid programs \( p \in P_{GM} \). Then if \( P(1), P(2), \) and \( P(3) \), then \( P(p) \) holds for all \( p \in P_{GM} \).

We’re not going to prove this proposition, because that goes a bit too “low-level” for our purposes (let’s not spend the whole course “programming in mathematical machine language!”). But I hope that you can believe that it’s true.

\(^4\)Formally, \( \forall p \in P_{GM}. \exists o \in O_{BS}. \text{eval}(p) = o \).
In the proof of totality above, we used exactly this general principle, but specialized it to our problem:

\[ P(p) \equiv \exists o \in \text{Obs}.\text{eval}(p) = o. \]

Where \( \equiv \) basically means “is a macro that expands to”. Here \( P \) does not represent a set, like \( \text{Obs} \) does, and it’s not a variable representing an element of a set, like \( o \), but rather it is a placeholder for a statement in logic (about sets) that needs an argument (in this case \( p \)) to complete it. This distinction is a bit subtle, and we’ll get into it more later.

We will spend a lot of time in this course defining models (as sets) using various formalisms, and then exploiting general-purpose reasoning principles that we get (“for free” in a sense) to reason about properties of those sets. We’ll see some more interesting examples of this kind of reasoning very soon.

In informal practice, we take these principles for granted much of the time, but when our reasoning gets more complicated, being aware of these principles and their explicit structure can help you write precise proofs, and catch bugs in your definitions and theorems! One of my old professors used to say “your theorems are the unit tests of your definitions.” This is a good analogy, except that properly stated theorems are complete: If the proof is a correct proof of the theorem, then you know the proposition is true. With test cases, unless you can and do cover all possible cases, then there’s still room for bugs.\(^5\)

To summarize, once we have a formal model of a programming language (or really ANY complex system in computer science), we can exploit the structure of our definition to develop general-purpose reasoning principles and use those to establish once and for all properties of interest. Some properties, like the result of evaluating one program, can serve as a source of test-cases for an existing implementation. Some properties, like the fact that all programs produce results, can guide the development of an implementation in the first place, and other properties, like the existence of a diverging program, can tell us important properties about the general nature of a language.

We will see more examples of these ideas in action, especially in the context of language semantics that are complex enough that the models and propositions are significantly more supportive.

\(^5\)Mind you proving things correct can be quite costly compared to testing, so there is a significant productivity tradeoff here.