Modeling Programming Languages Formally

CPSC 509: Programming Language Principles

Ronald Garcia[*]

8 January 2014
(**Time Stamp**: 16:41, Wednesday 16th September, 2020)

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: we appeal to *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. In fact, if you *have* been exposed to logic and set theory before, I recommend that you pay close attention, because our approach is likely be quite different than you were previously taught.

We 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, though basic, gives us an opportunity to introduce a substantial number of important concepts. We'll build on these concepts for the remainder of the course.

A side-note: below, a variety of mathematical machinery (e.g., sets, propositions, and proofs) is introduced without much explanation. Don't be frightened: we cover these in more depth as we proceed. I expect that students come from a variety of backgrounds, so we take the time to explain concepts in detail and get you up to speed. My goal 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 behaviour*—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 name 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 *observable 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 like this:

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

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 purported program `4` stored in `four.vpd` does not compile because it is not meaningful: that is to say, it's not a well-formed Vapid program. The program `5`, on the other hand compiles fine, but it throws an exception. That's it's meaning!

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

The two languages Vapid 0.0 and Vapid 1.0 give us a chance to introduce the basic mathematical framework for specifying languages.[1] We specify these languages in three parts:

[1]Make no mistake about it: two "different versions of the same language" are not *technically* the same, though they are likely related. In general, precisely characterizing the relationships between them can be a challenge.

1. A *set* of legal (i.e. well-formed) *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:

$$\begin{aligned}&\textsc{Pgm} = \{\,1, 2\,\}\\&\textsc{Obs} = \{\,1, 2\,\}\\&\mathit{eval} : \textsc{Pgm} \to \textsc{Obs}\\&\mathit{eval}(1) = 2\\&\mathit{eval}(2) = 1\end{aligned}$$

The entire collection, or *set*, of programs includes exactly 1 and 2. To express that, we give an *extensional* definition of the set, where we just state the *extent*—fancy word for "members"—of the set. Vapidly enough, the set of observables is identical to the set of programs, since both are specified using the same mathematical *description* $\{\,1, 2\,\}$. However, for the sake of the humans who will read this spec, we use the names $\textsc{Obs}$ and $\textsc{Pgm}$ as synonyms for $\{\,1, 2\,\}$. Those synonyms have no new mathematical meaning: they refer to the same set. But we use both names to communicate our intent to humans, who need to understand what concepts our set-theoretic model is trying to communicate.[2]

Since there are a finite number of programs, we can immediately define its evaluator *equationally*: we first specify that it is a function, stating the *domain* $\textsc{Pgm}$ and *codomain*[3] $\textsc{Obs}$ of the desired function. Then we state two equations that we would like our function to satisfy. Conveniently enough, these two equations are sufficient to describe a single unique function that has the stated domain and codomain. In this case, we could have given an equivalent extensional definition, literally writing down the input-output pairs of the function like a table. This kind of definition makes it quite evident that the eval function, in fact *any* set-theoretic function, is just a set of pairs:

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

Every eval function you see in this course (in fact every set-theoretic function whatsoever!) will literally *be* such a lookup table of pairs. However, we cannot always define them extensionally, especially if the domain of the function has an infinite number of elements. More on that later.

The Vapid 1.0 language is not much different:

$$\begin{aligned}&\textsc{Pgm} = \{\,1, 2, 3\,\}\\&\textsc{Obs} = \{\,1, 2, \infty\,\}\\&\mathit{eval} : \textsc{Pgm} \to \textsc{Obs}\\&\mathit{eval}(1) = 2\\&\mathit{eval}(2) = 1\\&\mathit{eval}(3) = \infty\end{aligned}$$

Once again we model the evaluator as a function again, adding a new program 3, and representing the possibility of nontermination with the observable ∞. With these new definitions, $\textsc{Pgm} \neq \textsc{Obs}$ since the sets no longer comprise the same elements (which, don't forget, are also sets). Think back to our earlier interactions with Vapid 1.0, and compare the status of 3, which is a program, to that of 4, which is not in the set $\textsc{Pgm}$ so not a well-formed progam in this language.

It's worth noting that in many works of PL theory, you might see the same language modeled as follows:

$$\begin{aligned}&\textsc{Pgm} = \{\,1, 2, 3\,\}\\&\textsc{Obs} = \{\,1, 2\,\}\\&\mathit{eval} : \textsc{Pgm} \rightharpoonup \textsc{Obs}\\&\mathit{eval}(1) = 2\\&\mathit{eval}(2) = 1\\&\mathit{eval}(3)\ \text{undefined}\end{aligned}$$

[2]Often, that human is ourself several days, or hours, later.
[3]short for "counter-domain." The prefix"co-" appears a lot in mathematics in this way.

In this rendition, the definition of the eval function once again uses the equations from Vapid 0.0. The key difference is that 3 is a program now, but the evaluator is *undefined* for it, rather than mapping it to ∞. In fact, given the definition of PGM here, the three $eval$ "equations" are again equivalent to the extensional definition $eval = \{ \langle 2, 1 \rangle , \langle 1, 2 \rangle \}$. The difference here is that we treat $eval$ as having a larger domain than is reflected in its elements. In short, 3 is now a program but it's observable result is "undefined" so $eval$ is a *partial* function, meaning that it may not be defined for its entire domain. In this case, it is not defined for 3. This is indicated using a harpoon $\rightharpoonup$. We will prefer to model our evaluators as *total* functions, but we will use partial functions for other aspects of our language models. See the corresponding notes on sets and logic to learn more about total and partial functions.

Comparing these two plausible models of Vapid 1.0—one using total functions and one using partial functions—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.

1.1.1 Terminology: *Syntax* and *Semantics*

In real languages, where there are more than 3 programs, we resort to more sophisticated tools to describe the set of programs. This is especially true when there are an infinite number of programs: we surely can't list them all extensionally! Instead, we describe them in terms of a repeating phrase structure. We call this structure the *syntax* of programs.

Similarly, given more complex and possibly infinite sets of programs, we must resort to more complicated means of defining their 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. $eval(2) = 1.$

Proof. According to the definition of $eval$, the function must satisfy the equation $eval(2) = 1$. □

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 a formal definition of the language to figure out whether there might be an inconsistency between the spec and the implementation.[4]

Now that we've proven a property of *a single program in a language*, let's **prove a property of *the entire language***.

Proposition 2. *There is (i.e., there exists) a Vapid 1.0 program that diverges.*[5]*.*

Proof. Consider the program 3. Then according to its definition, $eval(3) = \infty$, which represents divergence. $\square$

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 Proposition , we could just follow our noses (i.e., consult $eval$) and be done. In this case, we needed some human insight to find a candidate program that satisfies the property, and then prove that it does. This is the typical structure of an *existence* proof...we pull the *witness* for the proof—the object that satisfies the property—essentially out of thin air, at least as far as the proof itself is concerned. In practice we may have first done a bunch of work on the side (i.e. check out all the programs, make hunches and throw them at the Vapid interpreter, read tea leaves, 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, that is, it gives meaning to each and every program. Let's do so for one of them:

Proposition 3. *In Vapid 1.0, $eval$ is a total function from* $\textsc{Pgm}$ *to* $\textsc{Obs}$*.*

We'll prove this two different ways: first an easy way, to show how sets inherit properties, then a more structured way, to demonstrate a more structured proof. Here's the easy proof:

Proof. Since $eval : \textsc{Pgm} \to \textsc{Obs}$, then it must be total. $\square$

Our definition of $eval$ begins with $eval : \textsc{Pgm} \to \textsc{Obs}$, which means that we are choosing the unique element from the set of total functions from $\textsc{Pgm}$ to $\textsc{Obs}$ that satisfies the given set of equations. So if our purported definition is in fact a definition at all (more on that later), then we know that eval is a function and it's total (as well as the domain, codomain, and a few equations that it satisfies). To motivate our second proof, suppose that we had instead written $eval : \textsc{Pgm} \rightharpoonup Obs$. In that case, we would be choosing from among the *partial* functions that satisfy our equation, so we would not get totality for free, but would instead *really* have to prove it. Even though we "know" intuitively that it's total simply by inspecting the definition, that gestalt knowledge can be boiled down to a series of precise formal observations from which we can deduce totality. Let's explicitly prove totality, that the evaluator is defined for every input program. The "function" part of "total function" establishes that the language is deterministic, but we'll ignore that for now because we can (☺). Let's state the totality proposition more precisely, and then prove it:

Proposition 4. *For every $p \in \textsc{Pgm}$, there is some $o \in \textsc{Obs}$ such that $eval(p) = o$.*[6]

Proof. Suppose $p \in Pgm$. Then we proceed by cases on $p \in Pgm$.

Case ($p = 1$). Consider the observable $2 \in \textsc{Obs}$. Then $eval(1) = 2$.

Case ($p = 2$). Consider the observable $1 \in \textsc{Obs}$. Then $eval(2) = 1$.

Case ($p = 3$). Consider the observable $\infty \in \textsc{Obs}$. Then $eval(3) = \infty$.

$\square$

[4] 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, for example, that stance is almost surely the wrong one to take.

[5] In formal notation, $\exists p \in \textsc{Pgm}.eval(p) = \infty$.

[6] Formally, $\forall p \in \textsc{Pgm}.\ \exists o \in \textsc{Obs}.\ eval(p) = o$.

Phew! Here we proved the theorem by exhaustively considering each and every program and then showed that we could evaluate each one.

Now notice the structure of the proposition: "for every ... there is some ... such that ... ." And compare that to the structure of the proof: "suppose ... consider ... then ... ."

It turns out that these two structures line up:

1. the "for every ..." part of the proposition matches up with the "suppose ..."part of the proof;
2. the "there is some ..." part of the proposition matches up with the "consider ..." part of the proof.

What about the "proceed by cases" part of the proof? That part can be matched up with *the structure of our definition of* PGM. One of the main themes of this course is that *the structure of your definitions affects the structure of your reasoning (about the defined objects).*

In this case, an extensional definition of programs (i.e., defining PGM by explicitly listing the elements of the set) licenses us with the ability to prove things about *all* programs by explicitly reasoning about each individual program (checking each one). We can state this reasoning principle in a more general form. 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 $p \in \text{PGM}$)**.** *Let P be a property of vapid programs $p \in \text{PGM}$. Then if $P(1)$, $P(2)$, and $P(3)$, then $P(p)$ holds for all $p \in \text{PGM}$.*[7]

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 and that in principle it could be proven by appealing to the rules of logic and the axioms of set theory.

In the proof of totality above, we used exactly this general principle, but specialized it to our problem:

$$P(p) \equiv \exists o \in \text{OBS}. eval(p) = o.$$

Where $\equiv$ basically means "is a macro that expands to". Here P does *not* represent a set, like 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.

Later when we have languages with an infinite number of programs, this kind of reasoning will not work (we'll get hungry well before we finish checking each one).

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 $eval$'s definition gives plenty of guidance to language implementors. If you squint your eyes a bit, you will see hiding in the *proof* of Proposition 4 a recipe for implementing an interpreter for Vapid.

We'll see this deep connection between *deduction* (i.e., systematic formal[8] proof via symbol-pushing by a human computer) and *computation* (i.e., symbol-pushing as performed by a mechanical computer) come up again and again, even as our languages become less vapid.

Now consider how we used $eval$. We defined $eval$ 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 a proof by cases might have looked if we used the "set of pairs" definition of $eval$:

Case ($p = 1$). Then $eval(1) = n$ means that $\langle 1, n \rangle \in eval$ for some n. By definition of $eval$, $\langle 1, 2 \rangle \in eval$ so $eval(1) = 2$.

This proof case is in some ways pedantic, but it's useful to get some sense of how the structure of the two equivalent definitions can lead to slightly different steps of deduction. We'll clarify this more later in cases where the difference is less subtle.

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

[7]Formally, $P(1) \wedge P(2) \wedge P(3) \implies \forall p \in \text{PGM}. P(p)$.

[8]Here the word "formal" refers to "forms", as in symbols written on paper, not tuxedos and ball gowns. So "formal logic" is just another way of saying symbolic logic: deduction via symbol-pushing. Nothing fancy here!

$eval$. 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.

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" from those definitions to deduce properties of those sets. We'll see some more interesting examples of this kind of reasoning *very* soon.

In informal practice, we take the general proof principles that arise from definitions of mathematical objects 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 cover all possible cases—which you often *can't*—then there's still room for bugs.[9]

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. Other properties, like the fact that all programs produce results, can guide the development of an implementation in the first place, and finally properties, like the existence of a diverging program, can tell us important things about the general nature of a language as a whole.

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 interesting and less vapid.

[9]Mind you proving things correct can be quite costly compared to testing, so there is a significant productivity tradeoff here. And sometimes it's just hard to figure out what the write proposition would even be!