Modeling Programming Languages Formally

CPSC 509: Programming Language Principles

Ronald Garcia*

8 January 2014
(**Time Stamp**: 20:35, Sunday 16th January, 2022)

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

These variations on the Vapid language provide an opportunity 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* (i.e. the function that relates each program to its *VALU*e).

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 and nothing more. To express that, we give an *extensional* definition of the set, where we just state the *extension*—fancy word for "members"—of the set. In particular, the *expression* $\{\,1, 2\,\}$ *describes* our set of interest as "*the unique set* S that has the property that the set x is an element of S if and only if x is 1 or x is 2." Though you've likely seen this notation before, we'll have occasion to discuss this further when we study the foundations of set theory.

Vapidly enough, the set of observables OBS is *identical* to the set of programs PGM, are specified using the same expression $\{\,1, 2\,\}$. For the sake of the humans who will read this model, we introduce the names OBS and PGM as synonyms for $\{\,1, 2\,\}$.[2] The two "equations" $\textsc{Pgm} = \{\,1, 2\,\}$, $\textsc{Obs} = \{\,1, 2\,\}$ provide no new mathematical meaning: the simply introduce two human-friendly synonyms for the same set. I put equations in quotes above because these are not equations, but rather directives. To be more accurate about intent we should write $\textsc{Pgm} \equiv \{\,1, 2\,\}$ to make clear that *this is a "macro definition" not an equation*: from now on, whenever you see the name PGM, just replace it with the expression $\{\,1, 2\,\}$. Given this, it naturally follows that $\textsc{Pgm} = \{\,1, 2\,\}$ is true because after macro expansion you get $\{\,1, 2\,\} = \{\,1, 2\,\}$, which is a true equation (we'll discuss reasoning about equality in detail later). So you could call it an *abuse of notation* that we wrote $\textsc{Pgm} = \{\,1, 2\,\}$ instead of $\textsc{Pgm} \equiv \{\,1, 2\,\}$. Such abuses of notation arise *a lot* in formal mathematics, for better or worse, so take this as an early warning to be on the lookout. Ideally I will identify and explain when and where such abuses show up and we'll unpack them together.

To define the evaluator function for Vapid, we choose to take a somewhat direct *equational* approach, which proceeds in three steps. First, we observe that once we know what the set PGM is, and the set OBS, we can immediately describe the set $\textsc{Pgm} \to \textsc{Obs}$ of all *total functions* from programs to observables. This set of function inputs PGM is called the *domain*, and the set of possible outputs OBS is called the *codomain*.[3] Second, the expression $\mathit{eval} : \textsc{Pgm} \to \textsc{Obs}$ declares that eval is a total function from programs to observables, that is, that it is an element of the set $\textsc{Pgm} \to \textsc{Obs}$. The colon notation is common, and looks like a type declaration from some programming languages, but could just as well be written $\mathit{eval} \in \textsc{Pgm} \to \textsc{Obs}$, where $x \in X$ is the standard notation for saying that set x is an element of set X. Third, we state two equations that we require our function to satisfy. These two equations suffice to describe a single (unique) function that is a member of the set $\textsc{Pgm} \to \textsc{Obs}$.

Since Vapid has only a finite number of programs, we can also define the evaluator equivalently using an extensional definition, literally writing down the input-output pairs of the function. This kind of definition makes evident that the eval function, in fact *any* set-theoretic function, is just a set of ordered 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.[4]

However, we cannot typically define them extensionally, especially if the domain of the function has an infinite number of elements. We don't have enough ink, paper, or time!

[2] Often, that human is ourselves several days, or hours, later.

[3] Codomain is short for "counter-domain." The prefix"co-" appears a lot in mathematics in this way.

[4] As I often say: *math functions don't run*: they're more like database tables than computational procedures.

A model of 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 ∞. This addition is interesting because we are using a set-theoretic value to represent a computational phenomenon: the failure of an expression to produce a *value*). We will have to use similar techniques to represent other computational *effects* like input-output, mutable state, and concurrent threads.

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.

For a slight contrast in styles, here is a different model of Vapid 1.0 that makes one different modeling choice:

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

Like our earlier definition, this one presents an equational definition of its evaluator. The key difference is that the evaluator is represented using a *partial* function rather than a total function. In particular, 3 is a program now, but the evaluator is *undefined* for it—rather than mapping it to ∞, the partial function has no mapping whatsoever for 3. In fact, given the definition of $\textsc{Pgm}$ here, the three eval properties (two equations and one more general proposition) are again equivalent to the extensional definition $\mathit{eval} = \{\, \langle 2, 1\rangle, \langle 1, 2\rangle \,\}$. 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 $\rightharpoonup$. This eval is identical to the set we defined for Vapid 0.0, but now we *interpret* it as meaning a partial function from $\{\, 1, 2, 3 \,\}$ rather than a total function from $\{\, 1, 2 \,\}$. Set theory has a very "untyped" feel to it!!! This is both a blessing and a curse. It provides great flexibility, but demands similarly great discipline.

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 techniques 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*. You will need to choose a representation for uncaught runtime exceptions.

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 its equational definition, $eval$ 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.[5]

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.*[6].

Proof. Consider the program 3. Then according to its definition, $eval(3) = \infty$, 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 Prop. 1, 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 might satisfy 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. Insight is often extra-logical.

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:

[5] 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*. However, in practice formal specs and implementations are developed together, with observations about each informing the other, and when reverse-engineering a formal semantics from a language without one, the role of gold standard is often taken by the implementation.

[6] In formal notation, $\exists p \in \text{PGM}.eval(p) = \infty$.

Proposition 3. *In Vapid 1.0, $eval$ is a total function from* PGM *to* OBS*.*

We'll prove this two different ways: first an easy way, which demonstrates how sometimes the structure of a definition immediately implies the property of interest, much like Prop. 1. Second we'll demonstrate a more involved proof that exploits the structure of the proposition being proved *and* the structure of the definitions of sets that are referenced in its statement.[7]

Here's the easy proof:

Proof. Since $eval : \text{PGM} \to \text{OBS}$, then $eval$ must be total. □

Our definition of $eval$ begins with $eval : \text{PGM} \to \text{OBS}$, which means that we are choosing the unique element from the set of total functions from PGM to 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 $eval : \text{PGM} \rightharpoonup \text{OBS}$. The definition would still describe the exact same set.

But the definition as stated no longer *immediately* implies that $eval$ is a total function, only that it is a partial function. Such a definition chooses from among the partial functions that satisfy our equations, and not all partial functions are total (though all total functions are partial functions, strange as it sounds). We must work harder to show that it is *total*. We can intuitively see that $eval$ is total simply by visually inspecting the definition, and 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. We state the totality proposition more precisely and then prove it:

Proposition 4. *For every $p \in \text{PGM}$, there is some $o \in \text{OBS}$ such that $eval(p) = o$.*[8]

Proof. Suppose $p \in \text{PGM}$. Then we proceed by cases on $p \in \text{PGM}$.

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

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

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

□

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.

This is quite common: in many cases, the structure of a proposition alludes to the structure of its proof. This is especially true for proofs of small propositions (often called lemmas) which we combine to prove more complex propositions.

Now 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). In particular, recall that we should interpret the expression $\{ 1, 2, 3 \}$ to mean "*the unique set S* that has the property that the set x is an element of S if and only if x is 1 or x is 2 or x is 3." As such, it suffices to prove a property individually for 1, 2, and 3, to prove that it holds for any x that claims to be an element of the set. We can state this reasoning principle more precisely as a *general reasoning principle* for proving properties that hold of all Vapid 1.0 programs:

[7] You'll often hear me say: "the structure of your definitions determines the structure of your reasoning properties!"

[8] Formally, $\forall p \in \text{PGM}.\ \exists o \in \text{OBS}.\ eval(p) = o$.

Proposition 5 (Principle of Cases on $p \in \text{PGM}$)**.** *Let Φ be a property of Vapid programs $p \in \text{PGM}$. Then if $\Phi(1)$, $\Phi(2)$, and $\Phi(3)$, then $\Phi(p)$ holds for all $p \in \text{PGM}$.*[9]

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

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

Where $\equiv$ basically means "is a macro that expands to". Here Φ 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 "macro" for a logical proposition that takes an argument in place of p. 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 principle will not work (we'll get hungry well before we finish checking each one). But that's ok, because our approaches to defining infinite sets provide their own effective reasoning principles.

In the above proof, each case requires only a single step of reasoning to find the observable result of each program. 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. Later, more complex languages will satisfy analogous propositions, and the proofs of those propositions will similarly encode some *method* of deduction that starts from any program p and arrives at some o, particularly the o that the evaluator associates with p, just like the theorem insists.

In general, we will encounter deep connections between *deduction* (i.e., systematic formal[10] proof via symbol-pushing by a human computer) and *computation* (i.e., systematic 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 $eval$. We could prove other properties of all programs in roughly the same way, but substituting a different property into our reasoning. As we proceed, we will introduce and exploit increasingly sophisticated reasoning principles, starting from a rudimentary set of initial principles. We will use our tools to build more tools!

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 arise from those those definitions to deduce properties of those defined 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 can be more comprehensive: 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.[11]

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

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

[10] 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!

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

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.