Why Formal Semantics?

Harry R. Schwartz

January 7, 2017

A *formal semantics* is a mathematical model used by programming language designers and researchers to describe how a language works.

Terms in a formal semantics might look like:

$$e ::= x \mid \lambda x : \tau . e \mid e\, e \mid c$$

Or like:

$$\frac{\Gamma \vdash e_1 : \sigma \to \tau \quad \Gamma \vdash e_2 : \sigma}{\Gamma \vdash e_1 e_2 : \tau}$$

These terms define aspects of the structure and evaluation of a language.[1] We'll discuss how to actually read a formal semantics in a future post.

A formal semantics is distinct from an *informal semantics.* Some examples of those are:

- An ANSI specification written in prose,
- An RFC,
- A canonical book describing the language, or
- A test suite that all implementations of a language must satisfy.

These can all describe the behavior of a language, often extremely well, but they don't give us the machinery to *prove* things about a language the way a more mathematically modeled formal semantics does.

What might we want to prove? Well, all kinds of things:

- Certain statements will always terminate.
- It's impossible to dereference a null pointer.
- Type safety.

[1] These particular terms are from the formal semantics of simply typed lambda calculus. The first one defines the possibles values of an expression, and the second describes function application. Technically the first expression describes *syntax*, and only the second is really *semantics*, but I'm being intentionally careless with terminology in this article to get the basic concepts across.

- Race conditions can't occur.

It's reasonable to wonder whether a formal semantics really makes it easier to make these guarantees. Is it easier to write bug-free mathematical logic than it is to write bug-free code?

For a working programmer it's definitely advantageous, since you don't have to do anything yourself to gain these benefits. Rather than proving (or, more often, hoping) that these kinds of bugs don't infest your program, you can be confident that the designers of your language have proved that those bugs can't exist in *any* program in the language.

A formal semantics is useful for a language designer, too, but lots of mainstream languages don't have one. I suspect that's because programming language theory is still a niche topic and most language designers haven't been deeply steeped in it. It's much easier to hack together an interpreter for a language than it is to prove theorems about it, at least at first, so more people do that.[2] It sort of reminds me of the "test-driven development slows you down" fallacy, but that's a separate rant.

Using a language with a formal semantics also makes it practical to perform complex static analysis on your code. It's awfully hard to leverage tools like SMT solvers or theorem provers without a mathematical definition of the language. We can sometimes even statically analyze the resource usage and performance characteristics of our code,[3] which feels totally magical to me.

So, formal semantics: pretty handy!

[2]Heck, I've done it, too. Blueprint is a cute little language with no formally defined semantics (or guarantees of any kind, for that matter). On the bright side, though, it has no users and is therefore unlikely to harm anyone. :-)

[3]Jan Hoffman gave some terrific lectures on this at OPLSS 2016, and his research group at CMU is actively working in this area.