Minding the (Semantic) Gap

Engineering Programming Language Theory

Arjun Guha
Computer Science Department
Brown University
arjun@cs.brown.edu

Shriram Krishnamurthi
Computer Science Department
Brown University
sk@cs.brown.edu

ABSTRACT

Like programs, programming languages are not only mathematical objects but also software engineering artifacts. Describing the semantics of real-world languages can help bring language theory to bear on both exciting and important real-world problems. Achieving this is not purely a mathematical task, but equally one of (semantic) engineering.

Categories and Subject Descriptors

D.3.1 [**Prog. Lang.**]: Formal Definitions and Theory

General Terms

Languages

1. SOME SOBERING MOTIVATION

Can an ad steal personal data? Can one ad alter the content of another? Can a mashup hurt your privacy settings? The default answer to all these questions is yes, quite easily, without a great deal of careful effort. Major Internet companies like Facebook, Google, and Yahoo! have devised libraries to attempt to provide such protection. But though these libraries come with instructions on use, they—like just virtually all other software—come with no guarantees.

> Replac[e] disclaimers by warranties.
>
> —David Parnas [6]

2. ON PROGRAMMING LANGUAGES

A careful examination of the "libraries" that these Internet companies provide shows that they are really trying to define secure sub-*languages* of JavaScript, the lingua franca of the contemporary Web. Indeed, programming languages are how programmers ultimately communicate with computers—that is, they are a human-computer interface. Languages can be designed to provide guarantees: type-safety, information flow security, termination, and more. These guarantees lift the level of abstraction: both saving programmers from having to implement these, and offering them building-blocks to attain higher goals. They are, thus, perhaps the ultimate software engineering abstraction.

These innovations continue to grow apace; we routinely see thoughtful programmers engaged in myriad design experiments. The Web, in particular, has unleashed innovation in this space by enabling dissemination and discussion. As a result, languages now toy with novelty in iteration constructs, object systems, modularity mechanisms, persistence protocols, concurrency techniques, deployment platforms, and much more. Many of these are created not by certified academics but by thoughtful and well-intentioned designers trying to address a felt or perceived need. The languages are defined not by formal specifications but by prose and implementations. When the designers can articulate the need and explain the applicability of their solution, they gather critical masses of developers who move the language forward. Canonical examples that have grown well beyond their initial niches include scripting languages such as Perl, PHP, Python, Ruby, and the aforementioned JavaScript.

Reading popular books and blogs on programming reveals these ongoing experiments. Unfortunately, software engineering research—which appears obsessed with "mainstream" developers—seems blind to these trends, while programming languages papers are no more representative of this great cacophony of ideas and experimentation. In fact, these languages represent a terrific opportunity in the intersection between software engineering and programming languages.

3. DEFINING THE GAP

To provide any kind of guarantee about languages, sublanguages, frameworks, or tools, we need semantic foundations. Unfortunately, the majority of languages in regular use have little by way of these; indeed, even simple but key questions (such as: Does the language obey lexical scope?) remain unanswered. This semantic gap stands in the way of any rigorous, higher-level analysis (especially by machine).

A central tool in the language researcher's toolkit is the core calculus. When approaching a new language, we are experienced in extracting and formalizing its essence, reducing it to a mere handful of pages. The resulting semantics can easily be held in one's head, and understanding it offers a sense of satisfaction.

FoSER 2010, November 7–8, 2010, Santa Fe, New Mexico, USA.

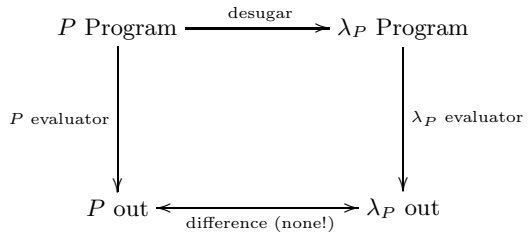

Figure 1: Testing Strategy for λ_P

This core is valuable, but falls short in two important ways. First, there is usually much more to the language (even just in terms of programming constructs) than what fits on these two pages. Second, for reasoning performed over the semantics to have validity (i.e., to provide programmers those elusive warranties), the actual language implementations have to conform to these semantics.

The first problem is especially well addressed in Scheme, which explicitly defines constructs outside the core through *macros*. Even in languages without such an integrated formalism, extensions are designed to act as syntactic sugar, with a clear sense of the intended rewriting (which is often implemented in actual evaluators).

The second problem is a little more subtle. The language may define `+` to always return numbers, but does the underlying implementation guarantee this? If it fails to do so, an attacker may be able to exploit this to create a security or other vulnerability. If the language gave primacy to the semantics, then at least the semanticist can feel exculpated because the fault is clearly the implementor's (though the user, whose warranty came up void, may be less generous). And where the semantics is primary, the implementor knows what to program in the first place.

The user of a scripting language is not so fortunate. The language may not have been designed around a small core, and non-orthogonal features may make such a reduction complicated. Any attempt to define a semantics is irrelevant in that the language is usually defined by an implementation, which is free to change irrespective of what the semantics says. Finally, the language may have various corner-cases such as heavily overloaded operations, which make an attempt at a semantics useful for reasoning very difficult.

> In practice most development effort goes into the "noise" that researchers abstract away. [...] [M]inimalistic subsets give rise to a nice and simple formalization, whereas language implementers actually need help formalizing the rough edges of the language, not the beautiful and clean subset.
>
> —Erik Meijer [4]

4. A WAY FORWARD

Is there any hope? Yes, there is! The implementations of all major languages—especially scripting languages defined by implementations—come with large and well-structured test suites. These suites embody the intended semantics of the language. We should be able to use such a test suite to retrofit a semantics.

For this to be useful, it is not sufficient to merely create a semantics for the core language. Instead, we must also make explicit the desugaring process that maps the full language into the core. Desugaring a scripting language is similar to desugaring C [1, 5]. Like C, scripting languages are also defined by their implementations.

Figure 1 shows how desugaring relates to testing. A P program can be desugared into the semantics (call it λ_P), which can then be evaluated. (If λ_P is small enough, an interpreter is easy to write and inspect for correctness.) Of course, P programs can also be run on actual implementations. These two outputs can then be checked to be the same. By applying this check to all programs in the test suite, we can argue that the semantics enjoys two properties: (1) that desugaring is total (i.e., that it maps all P programs to λ_P), and (2) that λ_P faithfully represents the language's intent. In short, we can achieve very high confidence that the semantics actually mirrors what happens in reality: it would be *as tested as the implementation itself*. Our work for JavaScript [3] is a first step in this direction.

Such a strategy leaves open many tactical choices. There are many possible λ_P core semantics (with correspondingly different desugaring procedures) but, provided they all obey the above diagram chase, they can be regarded as equivalent and users can choose the one most appropriate for their task. A λ_P evaluator can also take liberties, such as using meta-language features to implement certain functions—perhaps even the P evaluator, if this would not interfere with the intended use of the semantics.

Ultimately, it would be nice for the semantics to transition into the language's specification and guide its subsequent growth. Our strategy, however, does not rely on such high ceremony. If specifiers and implementors will simply bless a comprehensive conformance test suite, researchers can create an adequate semantics, thereby closing the critical gap that lies between practice and the desire for warranties.

Acknowledgments.

The perspective of Matthias Felleisen, Robby Findler, and Matthew Flatt [2] has greatly informed ours. Our work is partially supported by the US NSF and by Google.

5. REFERENCES

[1] S. Blazy and X. Leroy. Mechanized semantics for the Clight subset of the C language. *Journal of Automated Reasoning*, 43(3):263–288, 2009.

[2] M. Felleisen, R. B. Findler, and M. Flatt. *Semantics Engineering with PLT Redex*. MIT Press, 2009.

[3] A. Guha, C. Saftoiu, and S. Krishnamurthi. The essence of JavaScript. In *European Conference on Object-Oriented Programming*, 2010.

[4] E. Meijer. Confessions of a used programming language salesman. In *ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages & Applications*, pages 677–694, 2007.

[5] G. C. Necula, S. McPeak, S. P. Rahul, and W. Weimer. CIL: Intermediate language and tools for analysis and transformation of C programs. In *Compiler Construction*, pages 213–228, 2002.

[6] D. L. Parnas. Software engineering: An unconsummated marriage. *ACM SIGSOFT Software Engineering Notes*, 22(2), Nov. 1997.