Formal Semantics as a Language Designer's Toolbox

A case for semantics-inspired language design

Paolo G. Giarrusso   Klaus Ostermann   Tillmann Rendel   Eric Walkingshaw

University of Marburg

With the rise of domain-specific languages, more people are designing languages than ever, but the art of language design is hard to master. In this talk, we make a case for adapting insights from formal semantics into design patterns and conceptual tools for language design. Semanticists have invested significant effort in developing these insights in the pursuit of mathematical elegance, indirectly providing us with a toolbox full of design tools that are simple and powerful. We will introduce a few of these design tools, illustrating how language designers can start applying insights from formal semantics right now, without being experts in the subject or incurring the overhead of formalizing their entire language. The design tools we present are just a starting point and we hope to inspire other language designers to adapt this utilitarian view of formal semantics, to help us reuse the effort and expertise of semanticists by adapting their insights into easily applied design tools.

Designing a program well is hard. Designing a reusable library well is harder. Designing a programming language well is almost impossible—whether it's general-purpose or domain-specific. Many programmers know this and the programmer's corner of the Internet is full of discussions about the merits of language design, ranging from frustrated rants to insightful analyses. Programmers worry a lot about language design because they have seen plenty of not-so-well designed languages from a user's perspective, complained about their design problems, discussed how to fix them, or even started a side project to finally design a language well, once and for all. Some of these side projects lead to implemented and released languages, whose design is then worried about by other programmers. It turns out that it is easier to dislike a language than to design a better one.

With general-purpose languages, only a few experts are involved in language design. But in a software project using *language-oriented programming* (Ward 1994), programmers design and implement domain-specific languages (DSLs) for the domain of the problem at hand before implementing the actual program in that language. In a project that follows this methodology, *programmers* have to suddenly become *language designers*, too. Where can these programmer–language designers look for guidance, if they want to achieve the same elegance, clarity, and coherence in their language design as they strive for in their program and library design?

One underused source of guidance is the work of philosophers, mathematicians, and theoretical computer scientists that study the formal semantics of programming languages. These semanticists are experts in the elegance, clarity, and coherence of language design: They are trained to find and exploit structure in programming languages. They understand how language features relate to each other. They know different perspectives on language semantics. They can specify a language semantics concisely and at the same time precisely enough to formally prove theorems about it. Their formalisms are fine-tuned for looking at specific aspects of language design at a time.

Semanticists take for granted the importance of elegance, but the pragmatic language designer may be skeptical since it is too subjective to make its importance precise. However, elegant theories are preferred—in mathematics, science, and engineering—not only because of some subjective aesthetic value, but because elegance in these contexts implies simplicity and effectiveness of reasoning (Hardy 1940). Because elegant theories allow simple reasoning to be effective, they enable tackling complex problems in a manageable way: in other words, mathematicians, scientists, and engineers use *elegant* theories for *pragmatic* reasons. In language design, elegance is especially important: language designers need to reason about their language designs, and programmers need to reason about the programs they write in the language. Semanticists are experts when it comes to elegance, so it makes sense to reuse their expertise.

But most programmer–language designers never studied philosophy, mathematics, or theoretical computer science in depth. In fact, there are so many approaches to programming language semantics, focusing on different qualities of a language, that they wouldn't even know what to study or where to start. Moreover, the perceived role of formal semantics is to formalize a language, but many programmer–language designers do not have the resources or desire for such an effort, and so may not see value in learning about formal semantics in the first place.

In this talk, we make a case for a *utilitarian view of formal semantics*, where the guidance and insights of semanticists are made available to language designers, even if they are neither experts in formal semantics nor interested in formalizing their language.

To achieve this, our long-term goal is to collect principles, heuristics, design patterns, and useful tricks developed in work on formal semantics, and to present them in a structured way that is self-contained and accessible to programmer–language designers. As a starting point, we will present just a couple of initial examples of design tools adapted from semantic insights. This will illustrate how language designers can immediately apply insights from formal semantics, reusing the effort and expertise that semanticists have invested in refining their approaches.

Our story is preliminary since we cannot yet present a comprehensive and structured set of design tools. We are also not experts in all of the many different semantic approaches. However, we believe that our view suggests a fruitful new area for research and inspires other researchers who are interested in language design to contribute to this effort.

Acknowledgments. We thank Yi Dai for his extensive comments and discussions on the topic and previous versions of this text.

References

G. Hardy. *A Mathematician's Apology*. Cambridge University Press, 1940.

M. P. Ward. Language-oriented programming. *Software - Concepts and Tools*, 15(4):147–161, 1994.