Desugaring in Practice: Opportunities and Challenges

Shriram Krishnamurthi

Brown University

sk@cs.brown.edu

Abstract

Desugaring, a key form of program manipulation, is a vital tool in the practical study of programming languages. Its use enables pragmatic solutions to the messy problems of dealing with real languages, but it also introduces problems that need addressing. By listing some of these challenges, this paper and talk aim to serve as a call to arms to the community to give the topic more attention.

Categories and Subject Descriptors D.3.4 [*Programming Languages*]: Processors

Keywords Desugaring; Resugaring; Semantics; Optimization

The Need for Desugaring

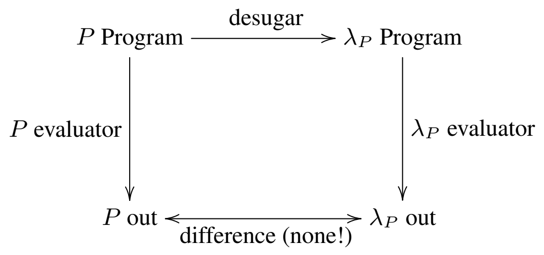
Concise core languages have a venerable place in the study of programming languages. By providing a means to reduce languages to an essence, they help us focus on important details and eschew ones irrelevant to the purpose of study. Furthermore, the difficulty of reducing some features may point to places where the language suffers from design flaws.

The use of core languages is not only of theoretical value. Practical systems also benefit from having a smaller number of features to work with: interpreters, compilers, type-checkers, program analyses, model checkers, and so on. Indeed, lurking inside every one of these systems is usually a smaller core fit for that purpose.

Unfortunately, the *process* of reducing a language to a core—which we loosely dub *desugaring*—has not received the attention it deserves, perhaps because it is not usually considered of theoretical interest.[1] As programming language research is increasingly applied to large languages (usually of industrial importance), the need to shrink languages—and hence for desugaring—has increased significantly, and with it the challenges faced.

[1] I use the term "loosely" for the following reason. In principle, a close reading of the term "desugaring" implies that the core language is a *strict subset* of the source. In practice, it is often useful for the "core" to be a slightly different language, better suited to the task at hand: for instance, for semantic analysis, we might map some object languages to a λ-calculus. In such cases, the desugarer is technically a compiler. However, it is rarely a general-purpose compiler, and the term "desugar" better evokes its intended purpose. This justifies our abuse of language.

PEPM '15, Jan 13–14 2015, Mumbai, India.
Copyright is held by the owner/author(s).
ACM 978-1-4503-3297-2/15/01.
http://dx.doi.org/10.1145/2678015.2678016



$$\begin{tikzcd}
P \text{ Program} \arrow[r, "\text{desugar}"] \arrow[d, "P \text{ evaluator}"'] & \lambda_P \text{ Program} \arrow[d, "\lambda_P \text{ evaluator}"] \\
P \text{ out} \arrow[r, leftrightarrow, "\text{difference (none!)}"'] & \lambda_P \text{ out}
\end{tikzcd}$$

Figure 1. Testing Strategy for λ_P

Desugaring in Semantic Specifications

Figure 1 illustrates a process that is now used for many real-world languages such as JavaScript and Python. For the language P, researchers define a core language, λ_P. Does λ_P really captures the essence of P: i.e., cover all of P and when translated, with the same behavior? It is usually easy to produce an evaluator for λ_P, while P evaluators are already available. Using test suites, programs found in the wild, etc., we can check for this conformance.

Once a suitable λ_P and desugaring pair have been found, the result is of practical value. Writing tools for λ_P is much easier than dealing with the full details of P itself. Therefore, even research groups with limited resources can—with the help of desugaring—try to apply their tools to real programs, thereby improving the utility of their research and the quality of their evaluation.

Challenges

While this picture is attractive in theory, there are many practical problems that confront the widespread use of desugaring. While some research has made progress on many of these fronts, general solutions that apply widely remain out of reach and are therefore ripe areas for future investigation. Some of these are described below, arranged in the order I conjecture to be the simplest to the most sophisticated.

Shrinking output in a semantics-preserving way The first and most persistent problem a user of any desugaring output confronts is the often large size of the output. Even simple examples can produce very large output: for instance, in the JavaScript semantics S5 [2], `console.log("Hello world")` produces the output shown in Figure 2. This code blow-up can frustate attempts to understand the output and to subsequently debug programs that process it.

Two main factors result in this blow-up: the inherent complexity of the source language, and the consequence of using a context-insensitive recursive-descent code generation process. The former is essential complexity from the perspective of preserving the language's behavior. We believe the latter can be addressed

```
{let
 (%context = %nonstrictContext)
 {%defineGlobalAccessors(%context,
                         "console");
  {let
   (#strict = false)
   {let
    (%obj1 = %context["console" , {[#proto: null,
                                   #class: "Object",
                                   #extensible: true,]}])
     {let
      (%fun2 = %ToObject(%obj1)["log" , {[#proto: null,
                                         #class: "Object",
                                         #extensible: true,]}])
       {let
        (%ftype3 = prim("typeof", %fun2))
        if (prim("!", prim("stx=", %ftype3 , "function")))
           {%TypeError("Not a function")}
           else
           {%fun2(%ToObject(%obj1),
                  %mkArgsObj({[#proto: null,
                              #class: "Object",
                              #extensible: true,]
                             '0' : {#value ("hello world") ,
                                    #writable true ,
                                    #configurable true}}))}}}}}}}
```

Figure 2. Desugaring of `console.log("hello world")`

by various program optimization techniques that can usefully shrink this output without altering its semantics. However, techniques designed for source programs can often perform weakly on generated code and vice versa. We are experimenting with the viability of different techniques for this purpose.

Shrinking output by altering semantics It is both worthwhile and uncontroversial to apply semantics-preserving optimizations. A more controversial idea is to engage in semantics-*altering* transformations.

A fully semantics-preserving desugaring must contain code that handles every possible corner case; the more complex the language, the more cases there are. This will include cases that are being disregarded in the current application. For instance, most static analyses assume away the presence of dynamic features like `eval` or reflective ones like Python's `locals`. By removing support for some features, we can have a noticeable effect on the size of desugared output. Furthermore, this interacts with semantics-preserving optimizations by exposing new opportunities for reduction.

Resugaring Desugared programs can be hard to work with. By having control over the desugaring process, it should be possible to at least partially "reverse" the desugaring process—in effect, to "resugar" programs. This becomes especially interesting in the presence of black-box transformations such as evaluation, optimization, and so on: to still present the new program in terms of the source that the author wrote. While there have been many ad hoc proposals to do this (e.g., for debugging optimized code), there are relatively few approaches that formally specify the properties expected of the resugaring process.

Relating multiple desugarings Desugaring is always relative to some purpose. In practice, different purposes often rely on different desugarings. For instance, a type-checker will want algebraic datatype definitions preserved; a type-inference system might want a `let`-like construct to also be preserved to perform polymorphic generalization; while an interpreter might want all these eliminated. Therefore, there is rarely *the* core of a language, but rather many cores.

Because all these desugarings correspond to the same surface language, it would be ideal to relate them to one another. While a testing approach (as in Figure 1) offers one way to relate them, it would be helpful to also do so formally, especially by focusing on the differences between them and identifying how these differences are semantics-preserving.

Learning desugaring from examples Finally, it is worth asking how much the process of generating a desugaring can be automated. In our experience, creating a semantics—with most of the time spent on desugaring—can be enormously time-consuming: from 6 to 28 person-months of highly-trained labor. This level of effort makes it infeasible to tackle the semantics of the large number of languages and systems in widespread use.

However, as Figure 1 suggests, we can think of this as a learning problem where, because of the presence of the evaluators acting as ground truth, we can generate new examples to improve the quality of learning. We have tried to apply a variety of machine learning techniques to this task, with distinctly mixed results so far. Nevertheless, it seems necessary to work on this if we are to have the Next 700 Semantics.

Impact on Reproducible Research

Desugaring has a valuable role in presenting research in a reproducible way. When researchers attempt to tackle large, industrial languages, they rarely ever tackle the language as a whole. Rather, it is common to pick some strict subset of the language. Unfortunately, the precise parameters of this subset are usually left loosely specified—or even unspecified—and need to be reconstructed from a paper presentation. This makes accurate comparison of competing work difficult, giving too much credit to some work and not enough to others.

Having a family of related desugarings, and semantics-altering desugarings that formally restrict the language, offers a way out of this dilemma. By picking and publishing a formally restricted desugaring, authors can effectively advertise precisely what sub-language they are actually working with. Because this is a computational artifact, other authors can use it in their comparisons, making it easier for authors, program committees, and third-party researchers to arrive at a much more accurate understanding of how different projects compare.

Desguaring in the Language

Finally, the Lisp family has a venerable tradition of providing desguaring features in the language itself. Recently, languages like Racket have taken this to new heights, enabling the definition of entirely new languages [1]. Though the challenges above have been written in terms of desugaring for semantics, many of them apply equally well to desugaring in the language, such as shrinking the output in semantics-preserving and even semantics-altering ways (for comprehension, debugging, and perhaps even performance), resugaring, and applying different desugarings in different contexts. As this powerful idea ripples (in various guises) through many other language families—from C++ to Scala to Haskell—the need for research to tackle these problems will only increase.

Acknowledgments

The work reported here was primarily done in collaboration with Arjun Guha, Joe Gibbs Politz, Ben Lerner, and Justin Pombrio. I thank them for forming and shaping my perspective. I have also benefited greatly by working with Claudiu Saftoiu, Junsong Li, and my other co-authors. This work is partially supported by the NSF and Google.

References

[1] M. Flatt and PLT. Reference: Racket. Technical Report PLT-TR2010-1, PLT Inc., June 2010. `http://racket-lang.org/tr1/`.

[2] J. G. Politz, M. Carroll, B. S. Lerner, J. Pombrio, and S. Krishnamurthi. A tested semantics for Getters, Setters, and Eval in JavaScript. In *Dynamic Languages Symposium*, 2012.