Resugaring:
Lifting Languages through Syntactic Sugar

by

Justin Pombrio

A thesis proposal submitted to Brown University

Abstract

Syntactic sugar is pervasive in language technology. Programmers use it to shrink the size of a core language; to define domain-specific languages; and even to extend their language. Unfortunately, when syntactic sugar is eliminated by transformation, it obscures the relationship between the user's source program and the program being evaluated. First, it obscures the evaluation steps the program takes when it runs, since these evaluation steps happen in the core (desugared) language rather than the surface (pre-desugaring) language the program was written in. Second, it obscures the semantics of the surface language, since typically a semantics is given only for the core language, with the surface semantics defined implicitly by transformation to the core. And finally, it obscures the scoping rules for the surface language for the same reason. I will address these problems by showing how formal semantics, scoping rules, and evaluation steps can all be *resugared* from core to surface languages, thus restoring the abstraction provided by syntactic sugar. While doing so, I will use the Pyret language as a testbed to ensure that these techniques can be practically applied.

Advisor: Shriram Krishnamurthi (Brown University)
External Committee Members: Eelco Visser (Delft University of Technology) and Mitchell Wand (Northeastern University)

1 Introduction

Syntactic sugar is an essential component of programming languages and systems. It is central to the venerable linguistic tradition of defining programming languages in two parts: a (small) core language and a rich set of usable syntax atop that core. (In this proposal I use the term *surface* to refer to the language the programmer sees, and *core* for the target of desugaring.) It is now actively used in many practical settings:

- In the definition of language constructs in many languages ranging from Python to Haskell.
- To extend the language, in languages ranging from the Lisp family to C++ to Julia to Rust.
- To shrink the semantics of large scripting languages with many special-case behaviors, such as JavaScript and Python, to small core languages that tools can more easily process [17, 31, 32].

In essence, desugaring is a compilation process. It begins with a rich language with its own abstract syntax, and compiles it by applying transformations to get to a smaller language with a correspondingly smaller abstract syntax. Having a smaller core reduces the cognitive burden of learning the essence of the language. It also reduces the effort needed to write tools for the language or do proofs decomposed by program structure (such as type soundness proofs). Thus, heaping sugar atop a core is a smart engineering trade-off that ought to satisfy both creators and users of a language.

Observe that this trade-off does not depend in any way on desugaring being offered as a surface linguistic feature (such as macros).

Unfortunately, once a program has been desugared, it is much harder for its programmer to recognize and obscures the original program. When desugaring is followed by any phase that rewrites terms, such as evaluation, optimization, or theorem proving, there is typically no easy way to view the rewritten terms using their original pre-transformation syntax. This penalizes either the programmer who uses the sugar (who must contend with the details of desugaring) or the language designer (who must decide whether to forgo sugar and deal with a larger, more complex language). In short, it violates the abstraction that syntactic sugar ought to provide.

We call attention to three such ways that syntactic sugar breaks abstractions:

Evaluation Steps Syntactic sugar obscures the evaluation steps the program takes when it runs, since these evaluation steps happen in the core language rather than the surface language the program was written in.

Formal Semantics Typically, terms in a surface language are meant to be defined by their desugaring; hence a language with syntactic sugar may have a formal semantics for its core language but ought not have one for its surface. However, many of the benefits of having a formal semantics are lost when it exists only for the core language.[1]

Scoping Rules In a similar manner, syntactic sugar obscures the (likewise implicitly defined) scoping rules for a surface language.

I propose to address these problems with a general approach called *resugaring*, whereby the items of concern (terms in the evaluation sequence, formal semantic rules, or scoping rules) are *lifted* through syntactic sugar from the core language to the surface language. I have begun to address the first of these three problems with this approach. This preliminary effort has already yielded promising results, which we discuss next, in section 2. With this proposal, I want to continue this successful line of research. I believe this work will result in giving desugaring its rightful place in the programming language space, making future implementers and researchers free to take full advantage of it, if they so choose, in their semantics and systems work.

2 Preparatory Work

In this section, I address the means of tackling the first of the three problems: how to restore the abstraction provided by syntactic sugar when presenting terms to the end-user of the language.

A solution to this problem requires computing a sensible evaluation sequence in a surface language while remaining faithful to the core language's semantics. My advisor and I took an approach that makes minimal assumptions about the evaluator, treating it as a black-box (since it is often a complex program that we may not be able to modify).[2] We only assumed access to a *stepper* [7] that provides a sequence of evaluation steps (augmented with some meta information) in the *core* language. We have shown how to obtain such a stepper from a generic, black-box evaluator by pre-processing the program before evaluation [33, section 7].

[1] Resugaring typing rules would be similarly useful, though it is not part of this proposal.

[2] An alternative approach would be to construct a lifted (to the surface language) reduction-relation directly. I propose to do exactly this, and we discuss the trade-offs, in section 3.

The main technique we proposed is *resugaring*: lifting the core evaluation sequence into one for the surface. The high-level approach is to follow the evaluation steps in the core language, find surface-level representations of some of the core terms, and emit them. Not every core-level term will have a surface-level representation; these steps will be skipped in the output. The evaluation sequence shown, then, is the sequence of surface-level representations of the core terms that were not skipped.

For example, take the sugar `or` defined by the rule (which let-binds `t` to `x` to ensure that `x` is only evaluated once):

$$x \ \texttt{or} \ y \quad \Rightarrow_{desugar} \quad \texttt{let} \ t = x \ \texttt{in if} \ t \ \texttt{then} \ t \ \texttt{else} \ y$$

Using this sugar, the program `not(true) or true` can be resugared as:

$$\begin{tikzcd}
\textbf{Surface} & & \textbf{Core} \\
\texttt{not(true) or true} \arrow[d] \arrow[rr, Rightarrow, "_{desugar}"] & & \texttt{let t = not(true) in if t then t else true} \arrow[d] \\
\texttt{false or true} \arrow[dd] & & \texttt{let t = false in if t then t else true} \arrow[ll, Rightarrow, "_{resugar}"'] \arrow[d] \\
& & \texttt{if false then false else true} \arrow[d] \\
\texttt{true} & & \texttt{true} \arrow[ll, Rightarrow, "_{resugar}"']
\end{tikzcd}$$

The actual core evaluation sequence is shown on the right, and a resugared surface evaluation sequence shown on the left. The third core term is skipped because it does not correspond to any surface program.

Of course, there are many possible resugarings, and we need some properties to guide our approach. The properties would also enable us to meaningfully conclude whether a given term has a surface representation or not. Central to our approach are these four properties:

Emulation Each term in the generated surface evaluation sequence desugars into the core term which it is meant to represent.

Abstraction Code introduced by desugaring is never revealed in the surface evaluation sequence, and code originating from the original input program is never hidden by resugaring.

Coverage Resugaring is attempted on every core step, and as few core steps are skipped as possible.

Hygiene Both desugaring and resugaring preserve α-equivalence (i.e., both transformations are hygienic in the strong sense advocated by Herman and Wand [20]).

We have written two papers that show how to perform resugaring while achieving these properties: the first paper [33] implements resugaring in a tool called Confection and shows that it achieves the first three properties, and the second paper [34] extends the technique to also achieve the fourth (hygiene).

3 Proposed Work

I will also tackle the following problems related to resugaring. Section 3.1 and section 3.2 are the last two problems mentioned in the introduction; section 3.3 is a useful extension to resugaring; and section 3.4 describes my primary planned application of this work.

3.1 Resugaring Formal Semantics

We have shown how to resugar evaluation sequences. Can formal semantics also be resugared?

[3] My approach to resugaring was to lift *evaluation sequences* through syntactic sugar. This has the advantage of allowing resugaring to operate even for core languages with extremely complicated semantics, or whose semantics isn't formalized. However, in the special case of a language whose core semantics has been formalized, it should be possible to lift the reduction-relation itself to the surface language. This gives a formal semantics to the *surface* language, which previously lacked one. This confers many benefits: the semantics can be read by humans, processed by tools, and targeted by proofs, making the language an easier object of study. (Some authors [27, 12] argue that it is important for a semantics to be defined over the surface language rather than the core.) The resulting semantics will be useful both to language authors, who can read the derived reduction rules for their language to ensure that they behave as expected, and for end-users of a language, who can obtain tools that operate directly on the surface language.

As an example, take the `or` sugar defined above. Suppose we know the following typical reduction rules for the core language (in addition to others):

$$\frac{a \rightsquigarrow a'}{\texttt{let } x = a \texttt{ in } b \rightsquigarrow \texttt{let } x = a' \texttt{ in } b}\,\textit{LetSimpl} \qquad \frac{a \text{ is value}}{\texttt{let } x = a \texttt{ in } b \rightsquigarrow b[x/a]}\,\textit{LetSubs}$$

$$\frac{}{\texttt{if } \textit{true} \texttt{ then } a \texttt{ else } b \rightsquigarrow a}\,\textit{IfTrue} \qquad \frac{}{\texttt{if } \textit{false} \texttt{ then } a \texttt{ else } b \rightsquigarrow b}\,\textit{IfFalse}$$

Using these rules, three reduction rules for the `or` sugar can be syntatically derived:

$$\frac{a \rightsquigarrow a'}{a \texttt{ or } b \rightsquigarrow a' \texttt{ or } b}\,\textit{OrSimpl} \qquad \frac{}{\textit{true} \texttt{ or } b \rightsquigarrow \textit{true}}\,\textit{OrTrue} \qquad \frac{}{\textit{false} \texttt{ or } b \rightsquigarrow b}\,\textit{OrFalse}$$

We show here how the *OrTrue* reduction rule can be derived; the others are no harder:

$$\begin{array}{lll}
 & \textit{true} \texttt{ or } \mathrm{b} & \\
\Rightarrow_{\textit{desugar}} & \texttt{let } t = \textit{true} \texttt{ in if } t \texttt{ then } t \texttt{ else } b & \\
\rightsquigarrow & \texttt{if } \textit{true} \texttt{ then } \textit{true} \texttt{ else } b & \text{by } \textit{LetSubs} \\
\rightsquigarrow & \textit{true} & \text{by } \textit{IfTrue}
\end{array}$$

I propose to implement a tool that performs derivations like this, taking a set of syntactic sugar rules and a set of reduction rules for the core language, and automatically deriving reduction rules for the surface language. To make this work as practically useful as possible, I will choose pre-existing standards for expressing both the syntactic sugar and the reduction rules. Many options exist. For instance, both Racket [15] and Spoofax [22] have suitable options for both.

[3] A similar question is weather typing rules can be resugared. This question is just as interesting, but I do not propose to address it here.

3.2 Resugaring Scoping Rules

Similarly, if scoping rules are given for a core language, is it possible to resugar them to obtain scoping rules for the surface language?

In my resugaring work, I have shown that there is a strong advantage to having scoping rules for the surface language. When the binding structure of the surface language is known, it is very easy to make desugaring *hygienic*. There are two competing definitions of this word, producing a divide in the literature. Hygiene has traditionally been defined *negatively* [23], as merely avoiding variable capture and other "unhygienic" problems; but this also allows the α-equivalence of surface programs to be defined in terms of their desugaring ($t =_\alpha t'$ iff $\text{desugar}(t) =_\alpha \text{desugar}(t')$). Other work [20] defines hygiene *positively* as the preservation of α-equivalence during desugaring; but this requires there to be an a-priori notion of α-equivalence over the surface language.

The downside to this assumption is that it requires sugar authors to write down scoping rules for their sugars. I propose to remove this burden (while *keeping* the stronger notion of hygiene) by automatically *lifting* scoping rules through syntactic sugar. For instance, take the sugar for a single-variable `let`:

$$\texttt{let } x = a \texttt{ in } b \quad \Rightarrow_{desugar} \quad (\lambda x.\ b)(a)$$

The scoping rules for the core language say that x is in scope within b in the desugared λ expression; hence x should be in scope within b in the original `let`. While this `let` example is trivial, lifting scope becomes a research question when scoping constructs such as pattern matching, recursive bindings, and modules are involved.

The immediate benefit of lifting scope is to remove the burden of writing scope rules, while simultaneously making the (resugared) scoping rules available to surface-level tools. This has clear practical value, because many programming tools benefit from this "lightweight semantics" of a language. For instance, some interactive development environments overlay arrows over programs to show bindings, or offer renaming transformations and other such refactorings [16]; however, these environments operate over the surface language, and hence require surface-level scoping information. We can generate this information automatically and supply it to the environment even when the actual scope is defined using the core.

I believe that lifting scope will also advance the state of hygiene systems. Lifting scope would enable bridging the gap between the "negative" and "positive" definitions, allowing the former category of work to justify itself under the more stringent requirements of the latter. In addition, it may also help reduce overall system complexity: many current real-world hygiene systems, such as that of Racket [15], tend to contain complex algorithms that encode scope rules internally.[4] In contrast, my preliminary studies suggest that transformations between two languages with explicit scope rules—an area that has not really been explored in the literature—are much simpler, and do not need convoluted renaming algorithms (a critique that has long been leveled against hygiene [2]). It would also remove scope from the innards of the algorithms, making them simpler and thus more reusable, more likely to be correct, and less susceptible to attacks. This is a particular concern since scoping is often used for protection in security settings [28, 35].

3.3 Partial Resugaring

Can partial resugaring be performed, and what are its formal properties?

[4]Recent work by Flatt somewhat improves the state of affairs for Racket [14].

Resugaring is predicated on the assumption that the implementation details of syntactic sugar should be hidden from the end user. This is often true, especially for, e.g., a student learning a language, who ideally isn't even *aware* which syntactic constructs are built into the language and which are implemented as sugar. Sometimes, however, a user does want to see these implementation details. For instance, a student may have just implemented a piece of sugar *themselves*, or a programmer may be trying to understand the reason for a program slowdown.

In addition, resugaring sometimes skips steps entirely. For instance, the recursive bindings defined by a `letrec` are skipped. They *cannot* be shown, on principle, because doing so would violate Emulation due to the recursive nature of the desugaring. This gives little solace, however, to a user who would much rather see a partially resugared term than nothing at all.

For both use cases, most syntax should be resugared, but a few syntaxes (e.g., the one the student implemented, or the `letrec`) should be left in desugared form, exposing their gory details. This breaks the Abstraction property; thus I will find a weaker version of the property that allows certain specified sugars to remain un-resugared. In return for weakening this property, I can further resugaring's central aim of increasing the comprehensibility of desugared programs, while doing so more flexibly.

3.4 Syntactic Extension for Pyret

Can these techniques be combined to produce user-friendly syntactic extensions for the Pyret language?

Pyret is an educational language developed at Brown. It is designed to be approachable to novices, suitable for education, and to have first-class support for testing. In good linguistic style, it makes heavy use of syntactic sugar. Take, for instance, the desugaring of this Pyret expression:

```
for map(x from [list: 1, 2, 3]):
  x * x                                   map(lam(x): _times(x, x) end,
end                             ⇒desugar      list.make([1, 2, 3]))
# produces [list: 1, 4, 9]
```

In this example, Pyret's `for` loop desugars into a function call, its binary operator desugars into a function call, and its list desugars into a more primitive constructor[5]. This desugaring is currently implemented as free-form Pyret code that manipulates the Pyret AST. Hygiene is ensured by giving identifiers both a name and a unique id, and by careful use of generating fresh ids.

I propose to implement a syntactic extension[6] system for Pyret with resugaring support. Doing so faces the several challenges:

- The extension system must support all of Pyret's current syntactic sugar. It may also support what Pyret calls its "well-formedness checks", that validate extra constraints beyond those of the grammar. For instance, one such check is that no block ends in a `let` binding, since it is not clear what a `let` binding should return. (There is a design question, though, of whether well-formedness is best left as a separate phase.)

[5]In the desugared output we write `[1, 2, 3]` for a "raw array", though Pyret has no surface syntax for it.

[6]I avoid the phrase "macro system" here because it typically implies allowing in-line syntactic extensions, which may or may not be in line with Pyret's design goals.

- Its error messages must remain at least as good as Pyret's current messages. Thus it must support custom error messages during desugaring (including well-formedness checking), and good messages during scope resolution.
- It should have the ability to lift scoping rules from the core language to the extended surface language. Ideally, these should be available to Pyret's IDE (code.pyret.org) for features like tab completion and viewing binding sites. It will *not* resugar semantic rules, however: Pyret does not have a formal semantics.
- I will build a "macro-aware" (i.e., syntactic-sugar-aware) stepper for Pyret. While the original resugaring paper[33] provided a prototype for such a stepper for an older version of Pyret, Pyret has changed implementations and grown in the meantime, and additionally, the stepper deserves a UI more suitable for student use.

This will show that the resugaring techniques outlined in this proposal can be practically applied to real-world languages.

4 Related Work

I first discuss work related to resugaring, and then mention work further afield: hygienic transformations, specifying binding structure, and syntactic sugar in general.

Resugaring Semantics There is a long history of trying to relate compiled code back to its source. This problem is especially pronounced in debuggers for optimizing compilers, where the structure of the source can be altered significantly [19]. Most of this literature is based on black-box transformations, unlike ours, which we assume we have full control over. As a result, this work tends to be at a lower level of abstraction than ours: some of it is focused on providing high-level representations of data on the heap, which is a strict subproblem of resugaring, or of correlating back to source expression *locations*, which again is weaker than *reconstructing a source term* as resugaring must do. In addition, this work is usually not accompanied by strong semantic guarantees or proofs of them.

One line of work in this direction is SELF's debugging system [21]. Its compiler provides its debugger with debugging information at selected breakpoints by (in part) limiting the optimizations that are performed around them. This is a sensible approach when the code transformation in question is optimization and can be turned off, but does not make sense when the transformation is a desugaring which is necessary to give the program meaning.

Deursen, et al. [38] formalize the concept of tracking the origins of terms within term rewriting systems, and show a variety of applications. Their work does not involve the use of syntactic sugar, however, while resugaring hinges on the interplay between syntactic sugar and evaluation.

Krishnamurthi, et al. [25] develop a macro system meant to support a variety of tools, such as type-checkers and debuggers. Tools can provide feedback to users in terms of the programmer's source using source locations recorded during transformation. The system does not, however, reconstruct source terms; it merely points out relevant parts of the original source. The source tracking mechanisms are based on Dybvig, et al.'s macro system [10].

Racket also has a built-in *macro stepper* [8]. This is *not* an algebraic stepper for programs that contain macros; rather it is a tool for viewing the expansion of macros at compile time. In other words, it shows steps of desugaring, not of evaluation. Thus it is orthogonal (and complementary) to the resugaring work we propose here.

Also within the Racket ecosystem, Culpepper, et al. improve compile-time macro error reporting [9]. Constructing useful error messages is a difficult task that we have not yet addressed. Akin to previous work in debugging, however, any source terms mentioned in an error appear directly in the source, rather than having to be reconstructed (as resugaring would).

Clements [6, page 53] implements an algebraic stepper for Racket—a language that has macros—and thus faces precisely the same resugaring problem addressed in this proposal. That work, however, side-steps these issues by handling a certain fixed set of macros specially (those in the "Beginner Student" language) and otherwise showing only expanded code. On the other hand, it proves that its method of instrumenting a program to show evaluation steps is correct (i.e., the instrumented program shows the same evaluation steps that the original program produces), while we only show that the lifted evaluation sequence is correct with respect to the core stepper. Thus its approach could be usefully composed with resugaring evaluation sequences to achieve stronger guarantees.

Fisher and Shivers [13] develop a framework for defining static semantics that connect the surface and core languages. They show how to effectively *lower* a static semantics from a surface language to its core language. In a similar vein, Lorenzen and Erdweg [26] give a method for ensuring the type soundness of syntactic extensions by *lowering* author-provided typing rules for the surface language onto the core language's type system (and automatically verifying that soundness is entailed). Both of these are complementary to resugaring, which aims to *lift* a semantics from core to surface.

Hygienic Transformation Our second paper on resugaring [34] gave a hygienic account of resugaring. A detailed comparison of our approach to hygiene against traditional hygienic algorithms is given in it.

Traditional approaches to hygiene suffered from an inability to formally state a general specification for hygiene. Recent work by Adams improves matters and advances the theory of hygiene by giving a relatively algorithm-independent notion of hygiene, and using it to derive an elegant hygienic transformer. However, a fundamental difficulty remains: the real goal for hygiene is for macros (or syntactic sugar) to preserve α-equivalence, but α-equivalence is typically only *defined* for the core language. Thus Herman and Wand advocate that macros specify the binding structure of the constructs they introduce, and build a system that does so [20]. Stansifer and Wand follow with a more powerful system called Romeo [37]. We used Romeo's binding algebra in our hygienic resugaring work [34] to specify surface language α-equivalence, thus allowing a direct statement of hygiene: desugaring (and resugaring) preserve α-equivalence.

An interesting alternative approach is put forward by Erdweg et al. with the *name-fix* algorithm [11]. *name-fix* also makes use of scope resolution, albeit in a different way. Instead of using scope resolution to *avoid* capture in the first place, *name-fix* uses it to *detect* capture and rename variables as necessary to repair it after the fact. This concession is necessary because *name-fix* assumes that it lacks control over the core language, and cannot place meta-data (such as a unique id based on scope) on variables. We do not assume this limitation, as it makes resugaring impossible.

Specifying Binding Structure There is a plethora of languages for specifying the binding structure for a programming language. In our hygienic resugaring work [34], we choose to use the binding algebra of Romeo [37] for resugaring because it is powerful enough to specify, e.g., `let`,

`let*`, and `letrec`, while still being strongly compositional in a way that allowed our resugaring operations to have a simple inductive definition. There are, however, many other binding specification languages of equal merit. Binding specification in the Ott semantic engineering tool [36] is similar to Romeo's, though it is somewhat more flexible and would, for instance, allow a term to export more than one set of binders. Likewise, Weirich et al. give a set of binding combinators in Haskell of similar power [39].

Neron et al. [29] introduce *scope graphs* as a formal representation for binding structure. Scope graphs leave the crucial requirement of obtaining the binding structure for a term up to other systems, such as the group's previous name binding language NaBL [24]. While NaBL itself lacks expressive power—it cannot describe the binding structure of, e.g., `let*`—we believe our resugaring work could be adapted to work with scope graphs on top of a different binding declaration language. Similarly, the typed HOAS [30] and PHOAS [5] efforts are excellent *representations* of abstract syntax, but do not say how to *construct* that syntax in a language-agnostic way. Thus, as with scope graphs, resugaring could use HOAS so long as these needs were met separately.

Syntactic Sugar in General Macros are user-defined syntactic sugar. While they have long been used in parenthetical languages like Scheme, they were only relatively recently extended to languages with more complex syntax. Notable early examples include an extensible parser by Cardelli et al. [4], and the Dylan programming language [1]. Both of these works present languages designed around syntactic extensibility. They provide Scheme-like macro systems, but also allow defining new syntactic forms to be handled by the parser, such as unary and binary operators as well as more complex forms of syntax. In each system, the focus is *not* on its macro system per se, but rather the algorithms for extending the grammar and parsing with the extended language that precede macro expansion.

Even more recently, macro facilities have become fairly common in non-parenthetical languages: notable examples include C++, Scala, Julia, and Rust. Again, however, none of them deal with the high-level concerns addressed in this proposal.

The Spoofax Language Workbench [22] is a platform for defining and then working with domain-specific languages. Given declarations for a language's syntax, scoping rules, and typing rules, it provides an Eclipse editor with features such as syntax highlighting and code completion. Spoofax combines a wide variety of tools: among others, it uses SDF [18] for syntax definitions, NaBL [24] for scope definitions, and Stratego [3] for declaring rewrite rules. While Spoofax does not have facilities that address any of the three resugaring problems we raise in this proposal, we believe they would fit well into its framework; our proposed work is wholly complementary to Spoofax.

References

[1] J. Bachrach and K. Playford. D-Expressions: Lisp power, Dylan style. `http://people.csail.mit.edu/jrb/Projects/dexprs.pdf`, 1999.

[2] A. Bawden and J. Rees. Syntactic closures. In *ACM Symposium on Lisp and Functional Programming*, 1988.

[3] M. Bravenboer, K. T. Kalleberg, R. Vermaas, and E. Visser. Stratego/XT 0.17. A language and toolset for program transformation. *Science of Computer Programming*, 72(1–2), 2008.

[4] L. Cardelli, F. Matthes, and M. Abadi. Extensible syntax with lexical scoping. Research Report 121, Digital SRC, 1994.

[5] A. Chlipala. Parametric higher-order abstract syntax for mechanized semantics. In *International Conference on Functional Programming*, New York, NY, USA, 2008. ACM.

[6] J. Clements. *Portable and high-level access to the stack with Continuation Marks*. PhD thesis, Northeastern University, 2006.

[7] J. Clements, M. Flatt, and M. Felleisen. Modeling an algebraic stepper. In *European Symposium on Programming Languages and Systems*, London, UK, 2001. Springer-Verlag.

[8] R. Culpepper. Macro debugger: Inspecting macro expansion, 2015. `http://docs.racket-lang.org/macro-debugger/index.html`.

[9] R. Culpepper and M. Felleisen. Fortifying macros. In *International Conference on Functional Programming*, New York, NY, USA, 2010. ACM.

[10] R. K. Dybvig, D. P. Friedman, and C. T. Haynes. Expansion-passing style: A general macro mechanism. In *Lisp and Symbolic Computation*, 1988.

[11] S. Erdweg, T. van der Storm, and Y. Dai. Capture-avoiding and hygienic program transformations. In *European Conference on Object-Oriented Programming*, Berlin, Heidelberg, 2014. Springer-Verlag.

[12] D. Filaretti and S. Maffeis. An executable formal semantics of PHP. In *European Conference on Object-Oriented Programming*, Berlin, Heidelberg, 2014. Springer-Verlag.

[13] D. Fisher and O. Shivers. Static analysis for syntax objects. In *International Conference on Functional Programming*, New York, NY, USA, 2006. ACM.

[14] M. Flatt. Binding as sets of scopes. In *Principles of Programming Languages*. ACM, 2016. To appear.

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

[16] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Professional, 1999.

[17] A. Guha, C. Saftoiu, and S. Krishnamurthi. The essence of JavaScript. In *European Conference on Object-oriented Programming*, Berlin, Heidelberg, 2010. Springer-Verlag.

[18] J. Heering, P. Hendriks, P. Klint, and J. Rekers. The syntax definition formalism SDF - Reference manual, 2001.
http://www.meta-environment.org/doc/books/syntax/sdf/sdf.pdf.

[19] J. Hennessy. Symbolic debugging of optimized code. *Transactions on Programming Languages and Systems*, 4(3), 1982.

[20] D. Herman and M. Wand. A theory of hygienic macros. In *European Symposium on Programming Languages and Systems*, Berlin, Heidelberg, 2008. Springer-Verlag.

[21] U. Hölzle, C. Chambers, and D. Ungar. Debugging optimized code with dynamic deoptimization. In *Programming Languages Design and Implementation*, New York, NY, USA, 1992. ACM.

[22] L. C. L. Kats and E. Visser. The Spoofax language workbench: Rules for declarative specification of languages and IDEs. In *ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications*. ACM, 2010.

[23] E. Kohlbecker, D. P. Friedman, M. Felleisen, and B. Duba. Hygienic macro expansion. In *ACM Conference on LISP and Functional Programming*, New York, NY, USA, 1986. ACM.

[24] G. Konat, L. Kats, G. Wachsmuth, and E. Visser. Declarative name binding and scope rules. In *Software Language Engineering*, Berlin, Heidelberg, 2012. Springer-Verlag.

[25] S. Krishnamurthi. PLT McMicMac: Elaborator manual. Technical Report 99-334, Rice University, Houston, TX, USA, 1999.

[26] F. Lorenzen and S. Erdweg. Modular and automated type-soundness for language extensions. In *International Conference on Functional Programming*, New York, NY, USA, 2013. ACM.

[27] S. Maffeis, J. C. Mitchell, and A. Taly. An operational semantics for JavaScript. In *Asian Symposium on Programming Languages and Systems*. Springer-Verlag, 2008.

[28] M. S. Miller. *Robust Composition: Towards a Unified Approach to Access Control and Concurrency Control*. PhD thesis, Johns Hopkins University, 2006.

[29] P. Neron, A. Tolmach, E. Visser, and G. Wachsmuth. A theory of name resolution. In *European Symposium on Programming Languages and Systems*, Berlin, Heidelberg, 2015. Springer-Verlag.

[30] F. Pfenning and C. Elliot. Higher-order abstract syntax. In *Programming Languages Design and Implementation*, New York, NY, USA, 1988. ACM.

[31] 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.

[32] J. G. Politz, A. Martinez, M. Milano, S. Warren, D. Patterson, J. Li, A. Chitipothu, and S. Krishnamurthi. Python: The Full Monty: A tested semantics for the Python programming language. In *ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages & Applications*, 2013.

[33] J. Pombrio and S. Krishnamurthi. Resugaring: Lifting evaluation sequences through syntactic sugar. In *Programming Languages Design and Implementation*, New York, NY, USA, 2014. ACM.

[34] J. Pombrio and S. Krishnamurthi. Hygienic resugaring of compositional desugaring. In *ACM SIGPLAN International Conference on Functional Programming*, 2015.

[35] J. A. Rees. *A Security Kernel Based on the Lambda-Calculus.* PhD thesis, MIT, 1995.

[36] P. Sewell, F. Z. Nardelli, S. Owens, G. Peskine, T. Ridge, S. Sarkar, and R. Strniša. Ott: Effective tool support for the working semanticist. *Journal of Functional Programming*, 20(1), 2010.

[37] P. Stansifer and M. Wand. Romeo: a system for more flexible binding-safe programming. In *International Conference on Functional Programming*, New York, NY, USA, 2014. ACM.

[38] A. Van Deursen, P. Klint, and F. Tip. Origin tracking. *Journal of Symbolic Computation*, 15(5–6), 1993.

[39] S. Weirich, B. Yorgey, and T. Sheard. Binders unbound. In *International Conference on Functional Programming*, New York, NY, USA, 2011. ACM.