Final version to appear in Onward 2015

Slimming Languages by Reducing Sugar: A Case for Semantics-Altering Transformations

Junsong Li, Justin Pombrio, Joe Gibbs Politz, Shriram Krishnamurthi
Brown University
{lijunsong, justinpombrio, joe, sk}@cs.brown.edu

Abstract

Splitting a language into a core language and a desugaring function makes it possible to produce tractable semantics for real-world languages. It does so by pushing much of the language's complexity into desugaring. This, however, produces large and unwieldy core programs, which has proven to be a significant obstacle to actual use of these semantics.

In this paper we analyze this problem for a semantics of JavaScript. We show that much of the bloat is *semantic bloat*: a consequence of the language's rich semantics. We demonstrate how assumptions about language use can confine this bloat, and codify these through several transformations that, in general, do not preserve the language's semantics. We experimentally demonstrate the effectiveness of these transformations. Finally, we discuss the implications of this work on language design and structure.

Categories and Subject Descriptors CR-number [*subcategory*]: third-level

1. Desugared Semantics in Practice

Programming language semantics research has a venerable tradition of describing a language by defining a small core and a desugaring function that maps surface programs into the core [17]. This approach has been used to define numerous languages such as ML [21]. More recently, it has been used in conjunction with tested semantics (appendix B) to describe the semantics of many scripting languages [3, 11, 13, 22, 23]. The resulting semantics, such as λ_{JS}, have been used to build a variety of tools (such as type checkers and security analyses), and have seen attempted adoption by many research groups in universities and industrial labs.

It is easy to see the benefits of using a desugared semantics for scripting languages. These languages have a large quantity of implicit and overloaded behavior, each of which would have to be handled directly by tools. By using desugaring, all these behaviors are made explicit, and the resulting operations can have much simpler behavior. The use of testing helps confirm that the semantics has not left out any important behavior. The tool-builder can then write programs over a much simpler language, resting assured that all the behaviors of the surface language have been handled.

This is the appeal—in principle. It resulted in several potential users, from universities and industrial labs, examining these semantics and trying to build on them. This would appear to represent a success story. In practice, however, many users have found the semantics difficult to adopt, and have even abandoned them. This paper explains the problem they have encountered, and presents a significant step towards addressing their problem. At its heart is an idea that many semanticists would find repugnant: program transformations that are intentionally not semantics-preserving.

2. Semantic Bloat Causes Code Bloat

The central problem is a very simple one: code bloat. To make this visceral, we present an example in λ_{S5} [22], a semantics of the strict-mode of ECMAScript 5 language. (Strict mode is a restriction of the language, implemented by browsers. It disallows some of JavaScript's more unsavory behavior, such as using undeclared variables, assigning to a read-only property, or using the dynamically scoped `with` statement.) Consider `function iden(x){return x;}`. It desugars to the 41 lines of code in fig. 1. We explain this desugared program in detail in section 2.2.

This explosion of code presents a tremendous challenge to the user of the semantics. The problem is *not* what the reader might imagine, which is that tools become slower because programs get large: because this represents the true meaning of the given JavaScript fragments, this state space must somehow be explored, whether in the source of the core

program or within the tool. In fact, the traversal time has not been a problem in practice in the tools we have built.

Rather, the real problems are the interlinked ones of comprehension and debugging. Building sophisticated tools is a complex activity; sometimes the tool itself may be doing something novel, and is hence difficult in its own right. Therefore, when things go wrong, it is vital to be able to produce small inputs and carefully trace their treatment. If even tiny JavaScript fragments produce output this large, the desugared semantics switches from being an aid to an active obstacle for the tool-builder.

Responding to this user feedback, we have performed an extensive study of the causes of code bloat in desugaring output. We found the primary cause to be *semantic bloat*: the inflation of code necessary to handle complicated language features. The solution is therefore simple: *instead of considering the full semantics of the language, consider restrictions on the semantics*. These restrictions would result in programs having fewer behaviors, which ought to result in much smaller desugared output.[1]

We could make this argument in the abstract, but we feel it better to make it concrete. By applying it to λ_{S5}, a real semantics, we can measure the exact effect of our claims. Besides, in practice many of these semantic restrictions are bound to be language-specific. The result also forms a useful catalog of JavaScript restrictions that can be evaluated in their own right.

This code bloat is, however, by no means limited to JavaScript: we have also noticed it with Python [23] and (to a lesser extent) Racket's Beginning Student Language [10], and appendices C and D show worked examples of this. We conjecture that the heavily overloaded and implicit behavior of languages like Matlab, R, etc. would also exhibit similar explosion after desugaring. Thus, we believe this is a general problem faced by many modern languages that value concision, in-language reflection, the ascription of object properties to many kinds of values, and so on, which is especially a trend in so-called "scripting" languages [15].

2.1 A Short Introduction to λ_{S5}

To aid in reading our code examples, we provide here an overview of some of the more surprising aspects of λ_{S5} syntax. Appendix A offers a quick-reference of syntactic forms, while the original paper [22] is a complete reference.

- Both `%` and `#` can legally occur in λ_{S5} identifiers. In fact, `%` serves as a common prefix for λ_{S5} internals.
- `label` and `break` are used in functions to represent non-linear control flow. `break` lbl val jumps to `label` lbl with value val.

[1] Our interest is not only in size but in code complexity in general. However, size is both easy to measure and a fairly good proxy for complexity. Furthermore, it is a dominant variable to a tool-writer trying to fit output on a screen.

```
1  let (%context = %strictContext) {
2   %defineGlobalVar(%context, "iden");
3   let (#strict = true) {
4    let (%fobj4 = {
5     let (%prototype2 =
6           {[...]
7            "constructor" : {#value (undefined),...}}) {
8      let (%parent = %context) {
9       let (%thisfunc3 =
10      {[#proto: %FunctionProto,
11        #code:
12         func(%this, %args) {
13          %args[delete "%new"];
14          label %ret: {
15           let (%this = %resolveThis(#strict, %this)) {
16            let (%context = {let (%x1 = %args["0",null])
17              {[#proto: %parent,
18                #class: "Object",
19                #extensible: true,]
20               "arguments": {#value (%args), ...},
21               "x": {#getter func (this, args) {
22                           label %ret: {
23                           break %ret %x1}},
24                     #setter func (this, args) {
25                           label %ret: {
26                           break %ret %x1 := args["0",...]}}}}}){
27             break %ret %context["x",...];
28             undefined}}}},
29        #class: "Function",
30        #extensible: true,]
31       "prototype" : {#value (%prototype2), ...},
32       "length" : {#value (1.), ...},
33       "caller" : {#getter %ThrowTypeError,
34                   #setter %ThrowTypeError},
35       "arguments" : {#getter %ThrowTypeError ,
36                      #setter %ThrowTypeError}}) {
37        %prototype2["constructor" = %thisfunc3, null];
38        %thisfunc3}}}})
39    %context["iden" = %fobj4 ,
40             {[#proto: null, #class: "Object", #extensible: true,]
41              "0" : {#value (%fobj4), #writable true, #configurable true}}]}}
```

function object — *function body* — *context object*

Figure 1. The desugaring of function `iden(x){return x;}`

- Like JavaScript, λ_{S5} uses an explicit *context object*, represented by the identifier `%context`, to map identifier names to their values. For instance, a reference to a JavaScript variable `x` desugars to a field access on the context object `%context["x", ...]`.
- In JavaScript, object fields are called *properties*, and both objects and properties have *attributes*. This is reflected in λ_{S5} objects. Take this sample λ_{S5} object literal:

```
{[#proto : %ObjectProto,
  #class : "Function",
  #code : func(this, args){ undefined },]
 "length" : {#value 0, #writable true} }
```

The object's attributes are listed between square brackets: `#proto` gives the object's internal prototype, `#class` gives its class name as a string, and `#code` gives the function body for objects derived from function definitions. This object also has a `"length"` property, whose attributes are written between braces: `#value` gives its current value, and `#writable` says whether the property can be mutated. There are other attributes as well: of note, rather than a `#value`, a property may have `#getter` and `#setter` functions that are called when it is accessed or mutated.

- Accessing a property is written as `object["prop"]`, while setting one is written `object["prop"= expr]`.[2]

Terminology The term "de*sugar*ing" with respect to λ_{S5} is somewhat misleading, since its core language is not a strict subset of JavaScript.[3] However, the translation from JavaScript to λ_{S5} is a context-insensitive recursive-descent process, and performs no aggressive transformations like many compilers do; it is thus sufficiently suggestive of traditional desugaring that we choose to call it "desugaring".

We will use one other piece of terminology. λ_{S5} is shipped with an *environment* that implements built-in JavaScript functions in λ_{S5}. Fully desugaring a JavaScript program, then, consists of performing recursive-decent desugaring over the program, and then prefixing the desugared program with the environment definitions. We call the recursively desugared program user code, thus distinguishing between *user code* and *environment code*.

2.2 Incidental Complexity

Consider the code in fig. 1, which shows a desugared identity function. The desugaring follows the algorithm defined in the specification: it first creates an object literal as the prototype (line 5), and then creates a *function object* literal which has many built-in properties. Finally, the code assigns the function object to the `"iden"` property of the *context object* %context (line 39).

It is not difficult to see that the main component of the function body is a new context object. This context object is created specially for dealing with scopes inside the function. It has the previous context object as its prototype; this is primarily used for looking up non-local variables through the prototype chain.

Much of this code is unnecessary, however:

- The let-binding introduces an unnecessary %parent identifier.
- The getter and setter of `"x"` in the context both have useless `label...break` expressions.
- The expression after `break %ret` is not reachable.
- The desugared function object always has four properties that might never be used.

The cause of this redundant and dead code is simple: the desugaring is a context-insensitive recursive-descent process. This has the advantages of simplicity and predictability, and the disadvantage of generating unnecessary code.

2.3 Essential Complexity

The main source of complexity in fig. 1, however, is not the little bits of redundant code generated by desugaring, but rather the underlying semantics of JavaScript. Pieces of the JavaScript code cannot be represented in straightforward ways in λ_{S5}, because λ_{S5} has to guard against all the strange features of JavaScript. For instance:

1. The `iden` JavaScript function cannot be represented as a λ_{S5} function, because JavaScript functions double as objects and object constructors, and it is possible that `iden` will be used as a constructor: `new iden()`. If this were not the case—if we assume that `iden` will only be used as a function—then the code in fig. 1 can be simplified to:

```
let (%context = %strictContext) {
 %defineGlobalVar(%context, "iden");
  let (#strict = true) {
    let (%fobj4 = {
      let (%parent = %context)
        func(%this, %args) {
        %args[delete "%new"];
        label %ret: {
         let (%this = %resolveThis(#strict, %this)) {
          let (%context = {let (%x1 = %args["0", null])
              {[#proto: %parent,
                #class: "Object",
                #extensible: true,]
               "arguments" : {#value (%args) ,
                              #writable true ,
                              #configurable false},
               "x" : {#getter func(this, args) {
                               label %ret: {
                               break %ret %x1}},
                      #setter func(this, args) {
                               label %ret: {
                               break %ret %x1 := args["0",...]}}}}})
               {break %ret %context["x", {[#proto: null,
                                           #class: "Object",
                                           #extensible: true,]}];
                undefined}}}}})
  %context["iden" = %fobj4 ,
       {[#proto: null, #class: "Object", #extensible: true,]
        "0" : {#value (%fobj4), #writable true, #configurable true}}]}}
```

2. Furthermore, the JavaScript *identifier* `iden` cannot be represented as a λ_{S5} identifier because of JavaScript's complicated scoping. For instance, it is possible to dynamically look up the identifier on the global object: `this["iden"]`. However, if we further assume that no scope shenanigans are used, then `iden` can be represented as a λ_{S5} identifier, and the code can be further simplified to:

```
let (%context = %strictContext) {
 let(iden = undefined) {
  let (#strict = true) {
   let(%fobj4 = {
     let (%parent = %context)
        func(%this, %args) {
          %args[delete "%new"];
          label %ret : {
            let (%this = %resolveThis(#strict, %this)) {
             let (%x1 = %args["0", null]) {
              break %ret %x1;
              undefined}}}}})
    iden := %fobj4}}}
```

3. Finally, JavaScript allows functions to be passed either more or fewer arguments then they were declared to take. Missing arguments are bound to undefined, and

[2] When these operations are given an extra argument—`object["prop", expr]` or `object["prop"= expr, expr2]`—this argument is passed to the properties' getter or setter function, if it has one.

[3] We credit Arjun Guha for creating this confusion.

extra arguments can be accessed through an array called `arguments`. As a result, JavaScript function applications cannot be converted directly to λ_{S5} applications. Rather, the λ_{S5} code must explicitly construct the `arguments` object. If we instead assume that functions will be called with however many arguments they were declared to take, the code can be shrunk further, to produce:

```
let (%context = %strictContext) {
 let (iden = undefined) {
  let (#strict = true) {
   let (%fobj4 = {
      let (%parent = %context)
         func(%this, %x1) {
          label %ret: {
           let (%this = %resolveThis(#strict, %this)) {
             break %ret %x1;
             undefined}}}})
    iden := %fobj4}}}
```

This example demonstrates how a great deal of complexity arises from the semantics of JavaScript, and how *making simplifying assumptions about how the language is used can greatly shrink desugared output*.

In section 5, we will further shrink the definition of iden to the following simple expression:

```
let (iden = func(%this, %x1) {%x1})
```

2.4 Inevitable Complexity

There is actually a third source of complexity: our choice of using a small core language. Readers can conceivably imagine the complexity added to the output by desugaring a `while` loop written in a highly expressive surface language to a recursive function definition in a small core language.

Choosing different core languages will *inevitably* add different levels of complexities to the desugaring output. This paper is not going to compare the impact of choosing different core languages.

3. Roadmap

Now we are ready to tackle the problem of shrinking the output of desugaring. The preceding examples make clear there are potentially two kinds of transformations we can apply. The first are semantics-*preserving*, such as dead-code elimination. Most of these are generic, but some are specific to patterns that arise in this particular language. More intriguing, however, are the ones that cannot be applied in all settings, because doing so would alter the meaning of the program.

The heart of this paper is a case study of these semantics-*altering* transformations. These are described in section 4. However, to perform any experimental analysis, we cannot rely simply on these. Even if we found savings, it could be that the same effect could have been achieved just by generic—or at least language-specific—semantics-preserving transformations. Therefore, we have also implemented a suite of those (section 5). In our experimental evaluation (section 6), we then contrast the effect each of these has, and also the impact of combining them.

There is one decision we have to make to implement the above transformations. We can either modify the existing desugarer directly, or we can introduce separate transformation phases that process the output of desugaring. Take the previous unnecessary `undefined`: we can choose to make the desugarer smarter—to not produce the `undefined` if the function has a return statement—or to leave the desugarer alone and purge it after desugaring.

We haven chosen the latter path, leaving the desugarer alone. First, this simplifies development: some of the transformations are not trivial, and encoding them into the desugarer would make that step much more complex, making it harder to debug when we make mistakes. Second, the desugarer might change as we discover more corners of the language; these changes are harder as the desugaring becomes more complex. Third, it more easily enables experimentation, because we can easily mix-and-match different rules (as we do in section 6). However, we do not consider any of these reasons canonical; there would be advantages to modifying the desugarer too (for instance, some information is currently lost, such as which functions are used as object constructors: this information is present at every new). Similarly, one could sidestep the problem entirely by employing more declarative rewriting systems designed for program transformation. Most of all, we do not believe it has a noticeable impact on the findings in this paper.

Workflow Each semantics-altering transformation has a crisp definition. We leave open how a user selects which transformations they want. In our current toolchain these are specified as command-line options. For example, users can use `-ir -fr` to apply *Identifier Restoration* and then *Function Restoration* to the code. Having a concise summary of which transformations were applied is important, as we discuss in section 7.

Working Assumption All the transformations in this paper make two assumptions. First, that the code is in strict mode; that is all λ_{S5} claims to model faithfully anyway. Second, that the code does not use `eval` or its equivalents; its presence would invalidate most of these transformations. In that regard, then, all the transformations here could be considered semantics-altering with respect to general JavaScript, but we find this an uninteresting classification. Most static analyses already assume the absence of `eval`, and richer analyses expect strict mode or its rough equivalent to ensure reasonable programs. Therefore, we consider this a reasonable precondition to impose.

4. Semantics-Altering Transformations

In this section we describe *semantics-altering* transformations. They purposefully simplify the semantics of JavaScript by weakening particular language features. They are useful

both for the practical purpose of making λ_{S5} code more manageable, and for the expository purpose of showcasing the costs of various JavaScript features (as explored in section 6.4).

We present these transformations in a catalog. Each entry has four parts:

Example of code before and after the transformation,

Justification of why this is a useful and reasonable simplification to make,

Pitfalls showcasing situations in which the transformation can change the meaning of the program, and

Preconditions under which these pitfalls are avoided and the transformation becomes semantics preserving.

◊ Fixing Function Arity (FA)

JavaScript supports variable arity functions by constructing an array of arguments at the call site called the *argument object*. Functions can index into this array to refer to each argument. Because desugaring explicates this, even calls that do not exploit this feature become much more complex in desugared output. In particular, this complicates any analysis that needs to distinguish parameters—such as a type system or flow analysis—because the parameters get lumped into a single object, and must be tracked through that object by the analysis. Therefore, this transformation simply disables variable-arity calls.

Example Take a simplified version of the iden function in fig. 1:

```
func(%this, %args) {
  let (%x1 = %args["0",null])
  ...
}
```

This transformation fixes the arity of iden by giving it a new function signature to match the declared arity in JavaScript:

```
func(%this, %x1) {
  ...
}
```

The transformation must transform call sites as well. Instead of constructing an argument object:

```
iden(undefined,
     %mkArgsObj({[...]
                 "0" : {#value (%context["a",...])
                         #writable true,
                         #configurable true}}))
```

the argument is passed in directly:

```
iden(undefined, %context["a",...])
```

Justification Variable-arity functions in JavaScript are used both for functions that can take any number of arguments, and to simulate optional arguments (whose default becomes undefined). They can also be error-prone, since the programmer is given no warning if they pass the wrong number of arguments to a function. It makes sense to use this transformation for code that does not use variable-arity functions for either of these use-cases.

Pitfalls Naturally, this transformation is not sound if the number of arguments in the call does not match the parameters of the callee. Concretely, when run under λ_{S5}, the resulting program will issue an arity mismatch error if the number of arguments differ, though this is an artifact of the λ_{S5} implementation (which could have chosen to simply ignore the mismatch and enable arbitrary behavior). However, this behavior is not disconcerting: a static analysis would presumably have caught and reported this mismatch in the first place, so the behavior of the λ_{S5} interpreter is uninteresting. Nevertheless, a program that would have run before with a mismatched number of arguments will now exhibit some other behavior.

Preconditions To preserve semantics, this transformation must be applied only when

- All functions are called with exactly as many parameters as their headers declare. (Since JavaScript is a higher-order language, this property is of course not soundly and completely decidable.)
- The `argument` keyword is not used.
- Due to the implementation of λ_{S5}, this transformation cannot be applied to getter and setter attributes. Fixing this would require modifying the λ_{S5} interpreter, which we purposefully avoid in this work.

◊ Function Restoration (FR)

In JavaScript, functions are objects and can also be used to create objects. In the latter case, the function is called a *constructor*. For example, the expression `new Date()` uses the function `Date` as a constructor to create a new object. This transformation conservatively restores some λ_{S5} function objects to λ_{S5} functions. It uses a (good) heuristic: function objects whose body does not use this can be restored to functions.

Example The code in fig. 1 presents a *function object*. After this transformation, the code contains only the function body; its object properties are trimmed away:

```
let (%fobj =
   func(%this, %args) {
     %args[delete "%new"];
     label %ret : {
       ...context object...
       break %ret %context["x", ...];
       undefined
     }})
  ...
```

Justification If a function is never used as a constructor or an object, it is unnecessary to keep its compound object form.

Pitfalls This transformation hobbles code that tries to access properties from the function object (e.g., obtaining the length of formal parameters). For instance, this code gets the length of function `iden` defined in fig. 1:

```
%context["iden",...]["length", ...]
```

This code will fail because the property will not be present. In the λ_{S5} interpreter this issues an error, but again this is relatively unimportant.

Preconditions The following preconditions should hold for restored functions (those whose body does not use this):

- There is no property inspection of restored functions in the code.
- Restored functions are not used as constructors.
- instanceof and typeof are not used on the restored functions.

◊ Simplifying Arithmetic Operations (SAO)

JavaScript's arithmetic operators have complicated semantics. The addition operator, $+$, for example, produces surprising results when applied to objects and arrays:

```
[] + {} // => "[object Object]"
```

This transformation simplifies the semantics by making strong assumptions about how operators are used: *Additive* and *Relational* operators must be applied to arguments of the same type (either `Number` or `String`)), and *Bitwise Shift* and *Multiplicative* must be applied to `Numbers` only.

Example This transformation does not change the user code. The new semantics of arithmetic operations are implemented as a replacement of the old one. For example, the JavaScript program `a + b` desugars to:

```
%PrimAdd(%context["a",...], %context["b",...])
```

where %PrimAdd is defined in the environment. This transformation replaces the definition of %PrimAdd and other arithmetic operators.

Justification The variety of behaviors exhibited by operators like $+$ can be baffling to newcomers. JavaScript famously provides an eight-step algorithm [9] to describe the behavior of $+$, and by design, this ends up being reflected explicitly in the semantics. Whether this behavior is presented directly in desugared output or hidden behind a primitive operator (like %PrimAdd), it must still be handled by tools that process the language. Indeed, it makes the first introduction to λ_{S5} particularly daunting, because $1 + 2$ is the kind of example a user thinks to try first.

Pitfalls Some programs really do exploit the full range of behaviors of $+$. For instance, programmers sometimes use it to convert integers to strings, since adding a number to an empty string converts the number to a string.

To the author of a tool, this presents a choice. If they are unwilling to make assumptions about their source programs, they can still use this transformation to initially simplify the semantics that they work with. This provides a more tractable debugging target. Having developed the core of their system, they can enlarge the semantics by removing this assumption, admitting the full range of JavaScript behaviors.

Preconditions To preserve semantics, this transformation must be applied on the condition that

- *Additive* and *Relational* operators are applied to operands of the same type, specifically either `Number` or `String`; and
- *Bitwise Shift* and *Multiplication* are only applied to `Numbers`.

◊ Identifier Restoration (IR)

JavaScript has a complicated scoping semantics that prevents JavaScript identifiers from being represented as λ_{S5} identifiers. (λ_{S5} is lexically scoped.) For instance, identifiers can be dynamically accessed through the context object this:

```
var n0 = 2;
print(this["n" + 0]); // 2
```

To handle complications like these, a JavaScript identifier `n0` desugars into a property on a *context object* %context["n0"]. Variable declarations are performed using the helper function %defineGlobalVar(%context, "n0"), and assignments use %EnvCheckAssign(%context, "n0", 2, true). However, if the JavaScript context object is not used to interfere with variables' scoping, then they *can* be safely represented as λ_{S5} identifiers.

Example See fig. 2.

Justification The benefits of lexical scope are well-known: in short, it allows you to tell where an identifier is bound only by looking at the code surrounding it (above it in the AST, more or less). JavaScript scope is nearly lexical. Two things that stand in its way are the with statement (that introduces dynamic scope), and manipulation of the context object. JavaScript's strict mode removes the with statement (making it an error to use). Thus simply assuming the scope object is not used in certain ways makes JavaScript scope lexical, and allows its identifiers to be represented as λ_{S5} identifiers.

This transformation is particularly useful because it opens the door for semantics-preserving transformations. It enables constant and non-constant propagation, which in turn allow other optimizations to take effect. It also increases readability: identifiers are the parts of our code we choose to name, and most code makes heavy use of identifiers. Thus, it tends to be easier to tell what desugaring has done to your code when your identifiers have been left intact.

Pitfalls

JavaScript	λ_{S5}	Restored λ_{S5}

`var x;`
`x = 1;`
`this.x = 1;`
 	
`x;`
	`%defineGlobalVar(%context, "x");`
	`%EnvCheckAssign(%context, "x", 1, true);`
	`try {`
	`  %set-property(%ToObject(%this), "x", 1.)`
	`} catch {`
	`  %ErrorDispatch`
	`};`
	`%context["x"]`
		`let (x = undefined) {`
		`  x := 1;`
		`  try {`
		`  x := 1`
		`  } catch {`
		`  %ErrorDispatch`
		`  };`
		`  x}`

Figure 2. *Identifier Restoration* (IR) example

1. After this transformation, variables will no longer magically be accessible on the context object, so looking them up will instead return undefined. For example, the following code:

   ```
   var k = 1;
   access(this);
   ```

   is desugared and then transformed to:

   ```
   let (k = undefined) {
     k := 1;
     ...
     access(undefined, %this)
     ...
   }
   ```

   If calling `access(obj)` prints the property `"k"` of `obj`, this will cause `access(this)` to print undefined, instead of 1, as `this["k"]` is unbound and defaults to undefined.

2. In JavaScript, deleting a variable that was previously declared will throw an `TypeError`. For example,

   ```
   var k = 1;
   delete this.k; // throw JavaScript TypeError
   ```

   The above code desugars to:

   ```
   %defineGlobalVar(%context, "k");
   %EnvCheckAssign(%context, "k", 1., true);
   ...
   %this[delete "k"]; // TypeError. "k" is nonconfigurable.
   ...
   ```

   After this transformation is applied, delete no longer throws `TypeError`, and instead returns a boolean value for an unbound property:

   ```
   let (k = undefined) {
    k := 1;
    %this[delete "k"]; // No TypeError
   }
   ```

Preconditions To preserve the semantics, the code must not:

- Delete properties of the context object,
- Enumerate the context object's properties, or
- Access properties of the context object by computing names.[4]

[4] A *static* access `this.k` will, however, be restored to `k` when `k` is in scope.

♢ Unsafe Assertion Elimination (UAE)

The JavaScript specification is littered with implicit type conversions and type checks. These are reified in λ_{S5}, and result in a lot of generated code. This transformation removes them under the assumption that the code being run is already correct and never fails these checks.

Example For example, a simple function application:

```
iden(x);
```

desugars to

```
let (%fun6 = %context["iden", ...]) {
  let (%ftype7 = prim("typeof", %fun6)) {
    if (prim("!", prim("stx=", %ftype7, "function"))) {
      %TypeError("Not a function")
    } else {
      %fun6(undefined,
            %mkArgsObj({[..]
                        "0" : {#value %context["x", ...],
                                ...}}))}}}
```

By removing the assertion that iden refers to a function, the code becomes:

```
let (%fun6 = %context["iden", ...]) {
  let (%ftype7 = prim("typeof", %fun6)) {
     %fun6(undefined,
           %mkArgsObj({[..]
                       "0" : {#value %context["x", ...],
                               ...}}))}}
```

(The binding to %ftype7 then becomes unused and can be removed by *Dead Code Elimination*.)

Justification This transformation eliminates clutter that generally goes unused in correct programs.

Pitfalls This transformation obviously violates the language's semantics. If a program contains an error, it will now raise an internal λ_{S5} exception rather than the correct JavaScript exception. More subtly, code that might have relied on these type tests—such as a flow-sensitive type checker [14]—will clearly behave differently.

Well-behaved code should, however, generally not trigger built-in JavaScript exceptions. For instance, if a value ought to be a function but is not, the correct time to notice is some time *before* applying it. Correct JavaScript code that obeys this principle will be unaffected by this change; code that does not obey it will still throw an exception, though it will

be an internal λ_{S5} exception rather than the more correct JavaScript exception.

Preconditions The code must not fail any runtime type checks.

5. Semantics-Preserving Transformations

In this section we list our *semantics-preserving* transformations. These are (primarily) traditional optimizations that could be applied to most languages, and in this case are used to optimize λ_{S5} code. Most readers could get the gist of these transformations by simply skimming over their names. We have not performed flow analysis over the whole program: this would generate more opportunities for code compression, but would likely have little impact on the semantics-altering optimizations that are the focus of this paper.

We demonstrate the use of the semantics-preserving transformations by showing in steps how they can greatly simplify a piece of desugared code. This section will shrink it bit-by-bit, eventually reaching the denouement promised in section 2.3 of restoring a desugared identity function to a comparable form. We start with the following code:

```
"use strict";
function iden(x) {return x;}
var a = 1, b = 2, sum;
sum = a + b;
iden(sum);
```

After desugaring, as well as a few simplifications from semantics-altering transformations,[5] this code becomes:

```
let (%context = %strictContext) {
 let (sum = undefined)
  let (b = undefined) {
   let (a = undefined) {
    let (#strict = true) {
     "use strict";
     {let (%fobj9 ={let (%parent = %context)
                    func(%this, %x6) {
                     label %ret: {
                     {{let (%this = %resolveThis(#strict, %this)){
                       break %ret %x6;
                       undefined}}}}}})
        iden := %fobj9};
     a := 1.;
     b := 2.;
     undefined;
     sum := %PrimAdd(a, b);
     let (%fun2 = iden) {
      let (%ftype3 = prim("typeof", %fun2))
       if (prim("!", prim("stx=", %ftype3, "function"))) {
        %TypeError("Not a function")
       } else {
        %fun2(undefined, sum)}}}}}}
```

We will show how the semantics-preserving transformations can simplify it.

[5] Specifically, we apply IR, FR, and FA: these were chosen so the remaining code best illustrates our semantics-preserving transformations

◊ Assignment Conversion

The *Identifier Restoration* transformation produces identifiers and assignments, but other transformations are still unable to directly benefit from assignments. This transformation replaces many λ_{S5} assignments with `let` bindings, which have a simpler semantics that more transformations can exploit. Take our example: this phase can transform all the assignments while preserving the semantics.

```
let (%context = %strictContext)
...omitted...
 let (#strict = true){
 "use strict";
 let(iden ={let (%parent = %context)
            func(%this, %x6) {
             label %ret: {
              {let (%this = %resolveThis(#strict, %this)){
                break %ret %x6;
                undefined}}}}}) {
    let (a = 1.) {
     let (b = 2.) {
      undefined;
      let (sum = %PrimAdd(a, b)){
       let (%fun2 = iden) {
        let (%ftype3 = prim("typeof", %fun2))
        if (prim("!", prim("stx=", %ftype3, "function"))) {
         %TypeError("Not a function")
        } else {
         %fun2(undefined, sum)}}}}}}
```

◊ Constant Propagation

If an identifier is let-bound to a constant, it can be propagated to its use sites. The previous transformation exposes such let bindings. By propagating them, the code becomes:

```
let (%context = %strictContext)
...omitted...
 let (#strict = true){
 "use strict";
  let(iden ={let (%parent = %context)
             func(%this, %x6) {
              label %ret: {
               {let (%this = %resolveThis(true, %this)){
                 break %ret %x6;
                 undefined}}}}}) {
    let (a = 1.) {
     let (b = 2.) {
      undefined;
      let (sum = %PrimAdd(1, 2)){
       let (%fun2 = iden) {
        let (%ftype3 = prim("typeof", %fun2))
        if (prim("!", prim("stx=", %ftype3, "function"))) {
         %TypeError("Not a function")
        } else {
         %fun2(undefined, sum)}}}}}}
```

A binding is considered constant if its identifier does not undergo variable mutation, its evaluation has no side-effects, and its value is not mutable. JavaScript object literals are rarely constant, because by default they allow both old properties to be changed and new properties to be added. They can be made constant by calling the built-in `freeze` method after initialization:

```
var o = {"x" : 1, "y" : 2};
```

```
o.freeze();
foo(o);
```

However, since constant objects are rare to begin with, and propagating an object that is used more than once has the potential to increase code size, we only propagate objects that are used exactly once. Similarly, since propagating a function that is used more than once would almost certainly increase code size, we only propagate functions that are used exactly once. In both of these cases, propagation is actually done in the more general *Non-constant Propagation* transformation.

◊ Constant Folding

This transformation simplifies constant expressions: when it sees a side-effect-free primitive operation or a known built-in function whose arguments are constants, such as the %PrimAdd function call in the this example, it applies the primitive or function. The previous program thus transforms to:

```
let (%context = %strictContext)
...omitted...
 let (#strict = true){
 "use strict";
  let(iden ={let (%parent = %context)
             func(%this, %x6) {
              label %ret: {
               {let (%this = %resolveThis(true, %this)){
                 break %ret %x6;
                 undefined}}}}}) {
    let (a = 1.) {
     let (b = 2.) {
      undefined;
      let (sum = 3.){
       let (%fun2 = iden) {
         ...omitted...
         %fun2(undefined, sum)}}}}}}
```

Constant folding potentially produces more constants (e.g sum in this example) for propagating. Applying *Constant Propagation* again replaces the use of sum with 3.

◊ Dead Code Elimination

This transformation removes unreachable code, such as the `undefined`, and the needless `label...break` patterns. It also removes other useless code, such as the `let`s and expressions that have no side effect in our running example. Thus, we obtain:

```
let (iden = func(%this, %x6) {%x6}) {
 let (%fun2 = iden) {
  let (%ftype3 = prim("typeof", %fun2))
   if (prim("!", prim("stx=", %ftype3, "function"))) {
    %TypeError("Not a function")
   } else {
    %fun2(undefined, 3.)}}}
```

◊ Non-constant Propagation

Desugaring often introduces let-bindings to avoid duplicate evaluation, though they're not always strictly necessary. For example, in the previous transformed code, %fun2 is just used as an alias of iden and %ftype3 is used only once as a temporary variable. This transformation propagates and eliminates them, producing:

```
let (iden = func(%this, %x6) {%x6})
 if (prim("!", prim("stx=", prim("typeof", iden),
                            "function"))) {
  %TypeError("Not a function")
 } else {
  iden(undefined, 3.)}
```

In general, this transformation must take care to avoid possibly re-arranging the order of side-effects. It propagates non-constant functions and objects that are used exactly once, alpha-renaming captured variables if necessary.

As another example, unnecessary let bindings similarly occurs in fig. 1. The %context and %parent bindings are both unnecessary. These values can be propagated without altering the semantics of the program.

◊ Assertion Elimination

This is a variant of *Unsafe Assertion Elimination* (section 4) that only removes the type conversions and type checks that are demonstrably redundant. For example, the previous code checks iden's type before applying the function. This is unnecessary as iden's type is already determinate from the source. This code can be simplified to:

```
let (iden = func(%this, %x6) {%x6})
 if (prim("!", prim("stx=", "function", "function"))) {
  %TypeError("Not a function")
 } else {
  iden(undefined, 3.)}
```

Constant Folding can then clean the `if` branch and shrink the code to:

```
let (iden = func(%this, %x6) {%x6})
 iden(undefined, 3.)
```

◊ Function Inlining

Function inlining replaces a λ_{S5} function application with the function's body, substituting the actual parameters for the formal ones. Given the ability of *Non-constant/Constant Propagation* to propagate functions that are used exactly once, this transformation only inlines functions that have been propagated to a call site. Therefore, an inlinable function is allowed to contain free variables but must not contain assignments to its formal parameters. To prevent potential bloat, the actual arguments at the call site must be either constant primitive values or identifiers. The running example can thus be shrunk in the following steps:

```
let (iden = func(%this, %x6) {%x6})
 iden(undefined, 3.)
```

iden must first be propagated to the call site, yielding

```
let (iden = func(%this, %x6) {%x6})
 (func(%this, %x6) {%x6})(undefined, 3.)
```

Then this transformation inlines the function, producing

```
let (iden = func(%this, %x6) {%x6})
  3
```

A pass of *Dead Code Elimination* gives the expression `3`.

Function Inlining is less useful than it would otherwise appear because JavaScript functions double as objects and thus cannot be represented as λ_{S5} functions and cannot be inlined. `iden` in the running example is inlinable mainly due to the *Function Restoration* transformation.

♢ **Environment Cleaning**

This transformation eliminates the definitions of unused built-in functions. In this way it is similar to *Dead Code Elimination*. It has the benefit, however, of some extra information. In the environment, most of the global, built-in functions are injected into the global *context object* by direct assignment to the `%global` object. For example:

```
%global["print" = %print];
%global["console" = %console];
```

Dead Code Elimination is unable to remove assignments like these, because objects in JavaScript may have "setter" functions that are triggered when an object's properties are modified, and these setters may have side-effects. However, the environment never puts setters or getters on these global variables, and user code lacks direct access to them. Thus if a built-in object like `console` is unused or is redefined in user code, the previous injection of `%console` and all of `console`'s properties are safely eliminated by this phase.

This transformation is technically semantics-altering: it might eliminate references to built-in objects if all references to them are computed. Unlike the other semantic-altering transformations, however, it does not correspond to a simplification of JavaScript semantics; therefore we choose to put it in this category.

6. Evaluation

Having described our transformations in detail, we now examine their effectiveness.

6.1 Semantics-Preserving Transformations

Our semantics-preserving transformations are based on well-understood techniques, and we have carefully checked them by hand. Because we are beginning with a tested semantics, however, we can use the same ECMAScript test suite [8] to experimentally ensure we haven't altered correctness. Because we assume that programs do not use `eval` and related dynamic code-generation operations, however, we filter out the tests that use these features. The remaining tests constitute our *baseline*.

The test suite is grouped into the chapters of the ECMAScript spec. The first six chapters set the context for the spec, so they are not testable. Each chapter contains tests for both strict-mode and non-strict-mode semantics except for chapter 9, which only contains tests for non-strict mode. Figure 3 shows the testing results. These transformations are applied in an order that, our experiments find, maximizes shrinkage. Figure 3 shows that even after applying all these transformations, the semantics continues to conform to the tests. Because the number of strict mode tests is small, we also consider non-strict tests. As the figure shows this results in substantially more tests, which gives us much more confidence in the implementation of our transformations.

Chapter	Passed	Baseline	Percent Passed
ch07-strict	19	19	100%
ch08-strict	11	11	100%
ch09-strict	0	0	N/A
ch10-strict	130	130	100%
ch11-strict	82	82	100%
ch12-strict	7	7	100%
ch13-strict	52	52	100%
ch14-strict	5	5	100%
ch15-strict	16	16	100%
ch07-nonstrict	471	471	100%
ch08-nonstrict	140	140	100%
ch09-nonstrict	108	108	100%
ch10-nonstrict	87	87	100%
ch11-nonstrict	1024	1024	100%
ch12-nonstrict	377	377	100%
ch13-nonstrict	98	98	100%
ch14-nonstrict	20	20	100%
ch15-nonstrict	4677	4677	100%

Figure 3. Test results for the semantics-preserving transformations

6.2 Semantics-Altering Transformations

For semantics-altering transformations, the number of passed tests drops, which is not surprising. We focus instead on the trade-off between code size and correctness (as measured by the percentage of baseline tests passed): one would expect that the more code we excise, the less correct the resulting program becomes.

Figure 4 presents this information for each individual semantics-altering transformation and for ordered combinations of them. It shows only some combinations because these transformations are largely independent of each other and other orderings do not have substantially different impact. As expected, the more transformations we apply, the more we shrink code while trading correctness. Of course, the expectation is that these are applied either for debugging purposes or because we can make an environmental assumption about code that enables these optimizations.

Figure 4 also shows the correctness of our semantics-altering transformations. This confirms that our transformations are not *too* aggressive: even applying *all* of them results in code that passes over 70% of tests.

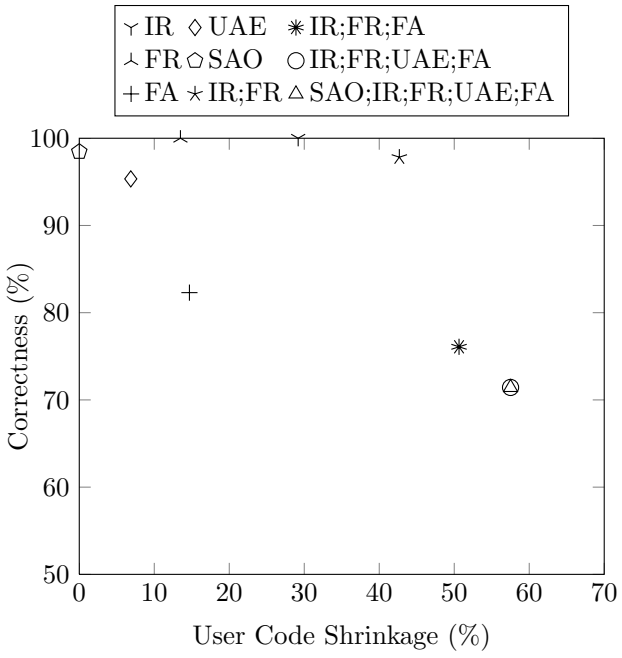

Figure 4. User code shrinkage vs. correctness of semantics-altering transformations

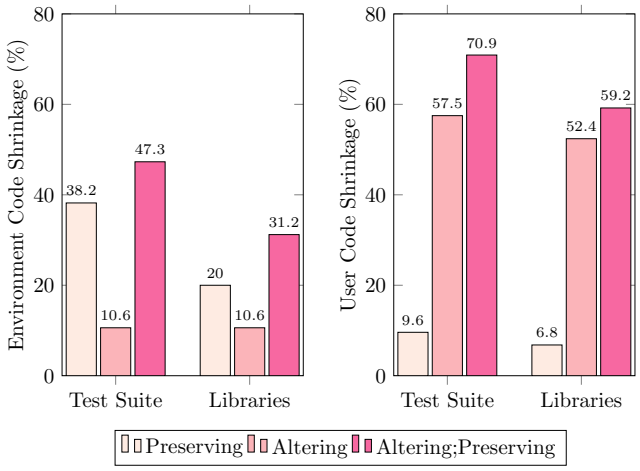

Figure 5. Environment code and user code shrinkage, per strategy

6.3 True Code Shrinkage

We have just examined the code shrinkage obtained by semantics-altering transformations, and seen that significant shrinkage can be obtained in exchange for reasonable amounts of "correctness". But perhaps the code bloat that was removed was actually *incidental*, rather than *semantic*, in nature and could have been removed by semantics-preserving transformations alone. We will now show that this is not the case by comparing the shrinkages of the semantics-preserving and semantics-altering transformations on the JavaScript test suite.

Recall the distinction between the *environment* and the *user* code. The environment is the λ_{S5} "standard library", which defines the meaning of primitives such as the addition operator; it is itself written in λ_{S5}, and hence amenable to transformation. User code is the program of interest. User code shrinkage is more relevant than environment shrinkage since it is the part that reflects the original JavaScript program, scales with its size, and is also usually the part that tools will wrestle with directly.

Measuring shrinkage of just the tests in the test suite may not be very meaningful, because it may reflect artifacts of the test suite. (The tests range in size between 6 and 78,122 AST nodes.) We have therefore assembled a collection of 8 widely-used JavaScript libraries (such as Three.js, D3.js, and Underscore.js), ranging in size from 23,313 to 802,459 nodes. In addition to being much larger, these are also reflective of real-world JavaScript.[6]

When both semantics-altering and semantics-preserving transformations are run together, the semantics-altering ones are run first. This is because they are essentially modifications of desugaring (as discussed in section 3), and must thus logically follow desugaring and precede the semantics-preserving transformations. We also expect the semantics-altering transformations to create new opportunities for semantics-preserving transformations to apply (as witnessed in section 5).

Figure 5 shows the shrinkage of applying both transformations to the test suite and the set of libraries. For each of the tests and libraries, the shrinkage is measured per test or per library; we present the average of these. We will briefly explain some of the causes of the patterns, and then interpret the data.

Preserving The semantics-preserving transformations have a large impact on environment code: this is mostly due to *Environment Cleaning*, which removes unused functions from the environment. (Which environment functions can be removed depends on user code, which is why the test suite and library environment shrinkage differ.) The more important user-code shrinkage is more moderate, however.

Altering The semantics-altering transformations together dramatically reduce user code size, on the order of 50%. The effect on the environment is minimal; they do not aim to shrink it.

[6] We cannot, however, use the libraries to check transformation correctness: the lack of comprehensive non-strict semantics, their use of `eval`, and the inability of λ_{S5} to interact with the DOM all make this a non-starter.

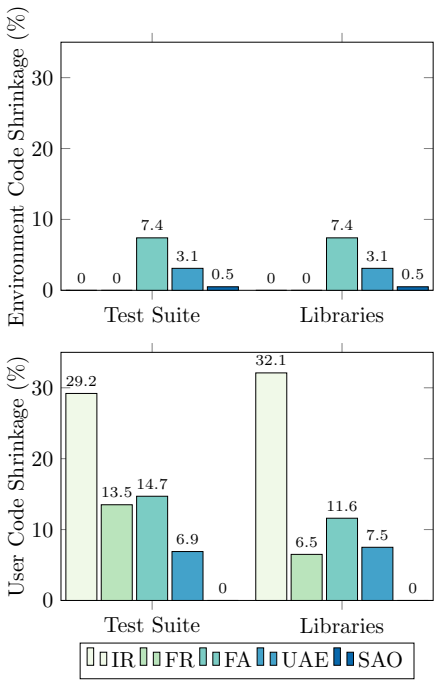

Figure 6. Environment code and user code shrinkage, per transformation, for semantics-altering transformations

Altering; Preserving The combined effect of semantics preserving and altering transformations is very close to the sum of them individually.

These data can be interpreted to indirectly compare the *incidental complexity* of desugared JavaScript code with the more inherent *essential complexity*. The incidental complexity is the "preserving" shrinkage: it is how much extra λ_{S5} code is generated, primarily as a byproduct of straightforward compositional desugaring. The essential complexity, on the other hand, is best taken as the *difference* between "preserving" shrinkage and "altering then preserving" shrinkage: the former is the essential complexity before semantics-altering transformation, while the latter is the essential complexity after transformation. The figure shows that the "preserving" shrinkage for user code is on the order of 10%, while the "altering then preserving" shrinkage is much higher, on the order of 50%. This supports the following observation:

Observation 1. *Most code bloat in λ_{S5} is semantic in nature.*

6.4 Comparing JavaScript Features

In fig. 6, we compare the code shrinkage of the various semantics-altering transformations. Since these transformations largely consist of weakening JavaScript features, this gives us a way to (indirectly) compare the complexity of the features. The reader can draw their own conclusions, but two things stand out:[7]

Observation 2. *By far the largest source of complexity is JavaScript's (unique brand of) dynamic scope.*

Observation 3. *The smallest source of complexity among those we tested is (perhaps surprisingly) the automatic type checks that JavaScript does.*

Of course, these observations are dependent on λ_{S5}'s implementation of desugaring: in particular, a feature that desugars into much more code than necessary would appear more complex than it is.

7. Discussion

Semantic Bloat As our examples show, the essential complexity of JavaScript is considerable. Naturally, not every program and programmer does, or should, exploit all of its complex behavior. A book like *JavaScript: The Good Parts* [7] effectively recommends programming in a temperate sub-language of JavaScript, and it is not alone: various type systems and static analysis tools effectively enforce linguistic shrinkage, as do myriad blog posts and other programmer advice. In the most extreme case, so does the ECMAScript standardization committee, through the creation of "strict mode" [9, section 4.2.2] and the consideration of new proposals such as "strong mode" [12].

Our experiments aid in this process. For one thing, they effectively (crudely) *quantify* the overhead imposed by the unfettered version of a language feature. Similarly, they help point to features that may be doing too much, and suggest ways in which these features might be restricted. Of course, these numbers are relative to a particular choice of core language, and a different core and choice of desugaring decisions would yield different results. Therefore, there must always be a strong element of human judgment involved in these considerations.

Language Modalities It is tempting to ignore semantic bloat in the following sense: why does it matter if a feature has more behavior than a program uses? For instance, consider property lookup in JavaScript: the "property" is usually a number or string, but it can be any arbitrary object; if it is an object, JavaScript invokes its `toString` method and uses that as the index. Most programmers simply never notice this (indeed, informal polls suggest many are not even aware of it), but what harm does it do?

For a well-behaved program, this is indeed unproblematic. In a buggy program, it can at least lead to results that are difficult to understand. But in a security context this is especially dangerous, because a malicious user can construct a payload that exploits this behavior that the program-

[7] It may be surprising that SAO does not shrink user code: this is because arithmetic operations are defined in the λ_{S5} environment, rather than being inlined. Ironically, this paper was originally motivated by the need for this optimization in a tested semantics where they *were* inlined.

mer did not anticipate. This has led to real-world security exploits [19] (for instance, if a program is trying to prevent access to a network communication operator—to avoid data exfiltration—the attacker can construct an object whose `toString` method evaluates to the name of that operator, thereby circumventing checks). For this reason, program analyses—especially those that apply to secure settings—must explore the entire space of language behavior; in turn, combined with the cost of precision, this can also lead to many false alarms.

In short, this is a setting where the two sides in the security-versus-convenience tradeoff take on very different weights. It also suggests that context is critical, and the expectations while "scripting" may be very different than those when "programming". Ideally, therefore, the programmer needs a means to express which form of language they want. The idea of providing a family of related languages has been explored before, first for PL/C [5] and more recently by the teaching languages in Racket [4]. The Racket mechanism (`#lang`) is, however, more general than teaching, and should be applicable here.

Indeed, one can view the JavaScript directive that invokes strict mode—`"use strict"`; at the top of a function—as a very limited form of `#lang`. However, perhaps programming languages should have the opposite defaults: not opt into the sub-language but rather opt-out (i.e., operate in a strict mode unless programmers write `"use script"`; when they want the full set of implicit and overloaded behaviors). This raises several interesting questions: interoperation between multiple, related languages, which is a form of multi-lingual programming [20]; the semantics of having individual functions (or even smaller units of code), rather than whole modules (as in `#lang`) be in different languages; and ways to ameliorate the verbosity of having to explicitly ask for the mode that was supposed to make programs more concise. We believe these are all interesting questions for future work.

Well-Defined Sub-Languages Lacking a `#lang`-like mechanism for JavaScript,[8] we have chosen to use command-line flags (section 3). Either way, this leads to a crisp definition of the sub-language in use. Besides its utility to tool developers, this is also vital to research evaluation. In practice, many (research) tools—especially static analyses—process only sub-languages, but the exact sub-language is left ill-specified or not specified at all. This greatly complicates processes like artifact evaluation [16]. Our approach makes clear exactly what language is being processed. This information should be published along with any evaluation results. Then, others can perform a more fair comparison: for instance, perhaps other results are less impressive but this can be attributed to their processing a much bigger sub-language. Right now, authors and reviewers often lack the information to make such comparisons.

[8] Though we note that in the context of the Web, the `script type=` tag and attribute can serve this purpose.

8. Related Work

There is a large class of semi-automatic optimizations that aim for complete correctness, but whose correctness might be compromised by user mistake. For instance, Java classes can be automatically partitioned across network boundaries [27], or `delay` and `force` placement can be automatically inferred when adding laziness points to a language [6], but both of these approaches can cause bugs given bad input from the user. This differs from our work both in that our tool requires no user input to determine how to apply optimizations (beyond *which* to apply), and that these tools aim for correctness whereas we do not.

There is also work in trading *accuracy* away for efficiency. For instance, in EnerJ [25], Java users can declare certain numerical computations to be inaccurate: these computations would then be run on hardware that performs approximate floating point arithmetic that loses some accuracy in exchange for vastly decreasing energy consumption. Recent work by Westbrook et. al. [28] examines optimizations that trade numeric accuracy away for *runtime speed*. Further, they show how to formally reason about the error bounds of these optimizations. Unfortunately, their work on error bounds does not apply to us, since rather than dealing with numeric approximations, our loss of correctness is truly semantic in nature and often involves changing control flow.

Work in test case reduction uses similar techniques to ours for a completely different purpose [24]. The goal here is to take a test case that witnesses a bug and reduce it to a smaller program that also witnesses the bug. This aids developers fixing bug reports, who can now spend their time dealing directly with a minimal test case. Like us, the goal here is to shrink code. In that work, there is no attempt whatsoever to preserve the semantics of the program, beyond that it should still witness the same bug; thus one transformation that might be made is removing an argument both from a function and its call sites.

Another (more specialized) area of work that involves semantics-altering transformations uses them to help perform program analysis [29]. The idea is to use transformations that can arbitrarily change the semantics of the program, but are guaranteed to leave the results of some particular kind of program analysis unchanged. The analysis can then be used on the transformed programs, helping with using analysis tools between different languages.

Based on all these and other papers, we have not found work that directly compares to ours. Specifically, this paper describes a general framework of semantics-altering transformations, presenting it both as an aid to debugging and tool development and as a means of restricting the language to match what program analyses can handle. We also describe its consequences for experiment comparison and reproducibility. Most of all, we couch our transformations as changes to desugaring, and propose this as a particularly good medium for describing these transformations.

Acknowledgments

We are grateful to Ben Livshits, Matt Might, and Mark S. Miller for interesting discussions about this problem, and to Ben Livshits and Alan Schmitt for detailed feedback on a draft. We thank Arjun Guha and our other co-authors for their extraordinary labors to create tested semantics in the first place. Without their work, we would not have platforms to perform these experiments, nor even these problems in the first place. We also thank the users who have tried and complained about using these semantics, for focusing our attention on the issues at the heart of this paper. This work is partially supported by the US National Science Foundation.

References

[1] G. Bernhardt. Wat. A lightning talk from CodeMash, 2012. `https://www.destroyallsoftware.com/talks/wat`.

[2] S. Bishop, M. Fairbairn, M. Norrish, P. Sewell, M. Smith, and K. Wansbrough. Rigorous specification and conformance testing techniques for network protocols, as applied to TCP, UDP, and sockets. In *Special Interest Group on Data Communications*, 2005.

[3] M. Bodin, A. Chargéraud, D. Filaretti, P. Gardner, S. Maffeis, D. Naudžiūnienė, A. Schmitt, and G. Smith. A trusted mechanised JavaScript specification. In *Principles of Programming Languages*, 2014.

[4] R. Bruce Findler, J. Clements, C. Flanagan, M. Flatt, K. Shriram, P. Steckler, and M. Felleisen. DrScheme: A programming environment for scheme. *Journal of Functional Programming*, 12:159–182, 2002.

[5] R. C. Holt and D. B. Wortman. A sequence of structured subsets of PL/I*. In *Special Interest Group on Computer Science Education*, 1974.

[6] S. Chang. Laziness by need. In *European Symposium on Programming Languages and Systems*, 2013.

[7] D. Crockford. *JavaScript: The Good Parts*. O'Reilly Media, 2008.

[8] ECMA International. ECMAScript Language test262. `http://test262.ecmascript.org/`.

[9] ECMA International. ECMAScript Edition 5.1, 2011. `http://www.ecma-international.org/publications/files/ECMA-ST/Ecma-262.pdf`.

[10] M. Felleisen, R. B. Findler, M. Flatt, and S. Krishnamurthi. *How to Design Programs*. MIT Press, 2001.

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

[12] Google Chrome V8 development team. Experiments with strengthening JavaScript, 2015. `https://developers.google.com/v8/experiments`.

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

[14] A. Guha, C. Saftoiu, and S. Krishnamurthi. Typing local control and state using flow analysis. In *European Symposium on Programming Languages and Systems*, 2011.

[15] J. K. Ousterhout. Scripting: Higher level programming for the 21st century. *IEEE Computer Magazine*, 31:23–30, Mar. 1998.

[16] S. Krishnamurthi and J. Vitek. Viewpoint: The real software crisis: Repeatability as a core value. *Communications of the ACM*, 58(3):34–36, Mar. 2015.

[17] P. J. Landin. The mechanical evaluation of expressions. *The Computer Journal*, 6(4):308–320, 1964.

[18] B. S. Lerner, M. J. Carroll, D. P. Kimmel, H. Q. la Vallee, and S. Krishnamurthi. Modeling and reasoning about DOM events. In *USENIX Conference on Web Application Development*, June 2012.

[19] S. Maffeis and A. Taly. Language-based isolation of untrusted JavaScript. In *Computer Security Foundations Symposium*, 2009.

[20] J. Matthews and R. Bruce Findler. Operational semantics for multi-language programs. *Transactions on Programming Languages and Systems*, 31(12), 2009.

[21] R. Milner, R. Harper, D. MacQueen, and M. Tofte. *The Definition of Standard ML, revised edition*. The MIT Press, 1990.

[22] 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.

[23] 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 and Applications*, 2013.

[24] J. Regehr, Y. Chen, P. Cuoq, E. Eide, C. Ellison, and X. Yang. Test-case reduction for C compiler bugs. In *Programming Languages Design and Implementation*, 2012.

[25] A. Sampson, W. Dietl, E. Fortuna, D. Gnanapragasam, L. Ceze, and D. Grossman. EnerJ: Approximate data types for safe and general low-power computation. In *Programming Languages Design and Implementation*, 2011.

[26] P. Sewell and S. Zdancewic. Principles in Practice (PiP), 2014. Co-located with *Principles of Programming Languages*.

[27] E. Tilevich and Y. Smaragdakis. J-Orchestra: Enhancing Java programs with distribution capabilities. *Transactions on Software Engineering and Methodology*, 19(1):1–40, 2009.

[28] E. M. Westbrook and S. Chaudhuri. A semantics for approximate program transformations. *CoRR*, abs/1304.5531, 2013.

[29] X. Zhang, L. Koved, M. Pistoia, S. Weber, T. Jaeger, G. Marceau, and L. Zeng. The case for analysis preserving language transformation. In *International Symposium on Software Testing and Analysis*, 2006.

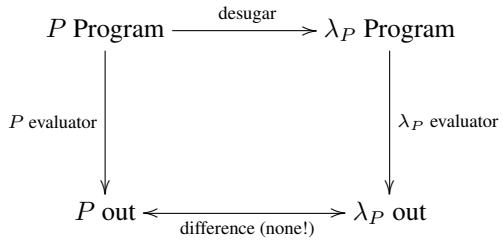

Figure 8. Testing strategy for λ_P

A. λ_{S5} Syntax

For a full description of λ_{S5}, see the paper [22]. Figure 7 gives a quick reference for the salient parts of the language.

B. Background: Tested Semantics

Languages must trade concision for explicitness: by providing more "overloading" (in the broad sense, not just the narrow notion used for function definition), they allow programmers to reduce the number of operations they must remember and use. In return, a given program phrase may have many meanings, some of which may even be decided by its context. This greatly complicates providing a semantic account of such a language.

The trend towards concision has been epitomized by scripting languages [15]. In the name of flexibility they provide extensive overloading, in-language reflection, dynamic error recovery, etc. As a result, seemingly simple program fragments can have a very rich set of behaviors. Gary Bernhardt's "Wat" talk [1] presents these behaviors humorously, but to a semanticist, these behaviors represent a true challenge. The semantics must cover all these possible behaviors to be useful for providing formal guarantees such as soundness.

The central problem, then, is determining whether all the behaviors have been captured in a semantics. The languages under question usually have large natural language descriptions that may not necessarily cover all (corner) cases. In many cases, the true definition is a binary that implements the system. Therefore, over the past few years, many researchers have worked on what are sometimes called *tested* semantics for languages and systems, including for JavaScript 3 [13], JavaScript 5's strict mode [22] and both strict and non-strict mode [3], PHP [11], Python [23], the browser DOM's event model [18], TCP/IP [2], and so on.[9] All these semantics use external validation, such as test suites, to confirm that the semantics do actually (substantially) model the behavior of the real-world systems they mean to represent.

Unfortunately, the resulting semantics can be large and unwieldy: after all, hundreds of pages of prose can be reduced only so much (for instance, the JavaScript semantics of Bodin et. al. [3]—which aims to conform to the specification, rather than to implementations—stretches to 3000 lines). Therefore, some semantics use the principle of a core combined with desugaring. The semantics therefore conform to the schematic shown in fig. 8. Given a language P, we define a corresponding core language, λ_P. Desugaring maps all P terms into ones in λ_P. We construct an evaluator for λ_P (which is often a straightforward exercise). The composition of desugaring and evaluation gives us a fresh implementation for P. We then use existing evaluator(s) for P and a suite of programs (such as implementation test suites, or even programs found in the wild—such as the examples in the Wat talk[10]) to ensure the fidelity of the semantics.

C. Code Bloat in λ_π

Semantic bloat also occurs in λ_π [23], a tested semantics for Python. The identity function in Python (similar to previous λ_{S5} examples) desugars to a λ_π program which has 1198 AST nodes:

```
(CModule
  (CApp (CFunc () (none) (CNone) (none)) () (none))
  (CSeq
    (CApp (CFunc () (none) (CNone) (none)) () (none))
    (CLet iden (GlobalId) (CUndefined)
      (CSeq
        (CSeq
          (CAssign
            (CId %locals (GlobalId))
            (CId %globals (GlobalId)))
          (CAssign (CId iden (GlobalId))
            (CFunc (x) (none)
              (CLet %locals-save (LocalId)
                (CId %locals (GlobalId))
                (CSeq
                  (CSeq
                    (CAssign (CId %locals (GlobalId))
                      (CFunc () (none)
                        ...omitted...
                        (none)))
                    (CLet
                      return-cleanup
                      (LocalId)
                      (CId x (LocalId))
                      (CSeq
                        (CAssign
                          (CId %locals (GlobalId))
                          (CId %locals-save (LocalId)))
                        (CReturn
                          (CId return-cleanup (LocalId))))))
                  (CAssign
                    (CId %locals (GlobalId))
                    (CId %locals-save (LocalId)))))))
          (none))))))))
```

In Python, programs can inspect the global/local scope: e.g., the built-in function `locals` in a function body can be used to access to local variables. This feature is the main source of the bloat in this example (omitted lines, roughly about 1000 AST nodes). If the code never exploits this feature, a semantics-altering transformation can eliminate the elided function, which yields a reasonable code size as follows:

```
1 (CModule
```

[9] Tested semantics like these were a major focus of the 2014 "Principles in Practice" workshop [26].

[10] In fact, using these examples helped us unearth a small but subtle bug in one of our semantics.

Syntax	Semantics
`%context`	Like JavaScript, λ_{S5} uses an explicit *context object*, represented by the identifier %context, to map identifier names to their values. (The % sign can legally occur in λ_{S5} identifiers. In fact, it serves as a common prefix for λ_{S5} internals.)

`{[#proto: null,`
`#class: "Object",`
`#extensible: true]}`
	A minimal **object literal**

`{[ ]}`	An abbreviation of the previous object.

`{[ ]`
`"name" : {#value "Customized",`
`#writable false}}`
	An object with one unchangeable field.

`let (brush = {[ ]})` `...`	Binding brush to an empty object.

`brush["size" = 16.0]`
`brush["radius" = 9.0]`
`brush["name" = "Pencil Brush"]`
	Setting **properties** of the brush object.

`brush["name", {[ ]}]`	Getting properties of the brush object. The additional argument `{[ ]}` is used in case this property has a *getter*.
`brush["name"]`	An abbreviation for the previous example.

`{[]`
`'opacity':{`
`#getter func(this, v) {0.7},`
`#setter func(this, v)`
`{throw "unchangeable"}}}`
	Defining a **getter** and **setter** for the opacity property of an object. These functions are called when the property is accessed or modified.

`let (getBrushName =`
`func(b) { b["name"] })`
`...`
	Binding getBrushName to a **function definition**.

`label %ret :` `...`	Declaring a jump label named %ret.
`break %ret exp`	Jumping to the label %ret with the value exp.

Figure 7. A quick reference of λ_{S5} syntax

```
2   (CApp (CFunc () (none) (CNone) (none)) () (none))
3   (CSeq
4     (CApp (CFunc () (none) (CNone) (none)) () (none))
5     (CLet iden (GlobalId) (CUndefined)
6       (CSeq
7         (CSeq
8           (CAssign
9             (CId %locals (GlobalId))
10            (CId %globals (GlobalId)))
11          (CAssign (CId iden (GlobalId))
12            (CFunc (x) (none)
13              (CLet %locals-save (LocalId)
14                (CId %locals (GlobalId))
15                  (CLet return-cleanup (LocalId)
16                    (CId x (LocalId))
17                    (CSeq
18                      (CAssign
19                        (CId %locals (GlobalId))
20                        (CId %locals-save (LocalId)))
21                      (CReturn (CId return-cleanup (LocalId))))))
22              (none))))))))
```

A smaller code size can be further obtained by semantics-preserving transformations as in section 5. For example, eliminating the unused bindings (line 13 and line 15), inlining functions (line 2 and line 4) and cleaning useless assignments (line 8 and line 18) produces:

```
(CModule (CNone)
  (CLet iden (GlobalId) (CUndefined)
    (CAssign (CId iden (GlobalId))
      (CFunc (x) (none)
        (CReturn (CId x (LocalId))) (none)))))
```

The final version of the code has around 20 AST nodes. The overall shrinkage comes from the two kinds of transformations similar to those in section 4 and section 5. This example demonstrates that making assumptions about the program can be useful in languages other than λ_{S5}.

D. Code Bloat in Racket's BSL

The *Beginning Student Language* (BSL) is a small version of Racket for teaching [10]. Consider this function:

```
#lang htdp/bsl
(define (len lst)
  (cond [(empty? lst) 0]
        [else (+ 1 (len (cdr lst)))]))
```

It desugars to:

```
1  (module x lang/htdp-beginner
2    (#%plain-module-begin
3     (define-syntaxes (len)
4      (#%app
5       make-first-order-function `procedure `1
6      (quote-syntax len)
7      (quote-syntax #%app)))
8     (define-values
9       (len)
10      (lambda (lst)
11        (begin0
12          (if (#%app verify-boolean (#%app empty? lst) `cond)
13              (let-values () `0)
14              (if `#t
```

```
15                (let-values ()
16                  (#%app
17                   beginner-+
18                   `1
19                   (#%app len (#%app beginner-cdr lst))))
20                (let-values ()
21                  (#%app error `cond
22                   `"all question results were false")))))))))
```

The block on lines 3 to 7 performs static arity checks when `len` is used in a different module. If this is assumed to never happen, a semantics-altering transformation could remove it. The code can be further shrunk by semantics-preserving transformations. Several expressions here have no effect or can be specialized, resulting in

```
1  (module x lang/htdp-beginner
2    (#%plain-module-begin
3     (define-values
4       (len)
5       (lambda (lst)
6         (if (#%app empty? lst)
7             `0
8             (if `#t
9                beginner-+
10               `1
11               (#%app len (#%app beginner-cdr lst)))))))))
```

Unlike λ_{S5} and λ_{π}, there is significantly less code bloat, and the bloat that is present seems best dealt with by semantics-preserving rather than semantics-altering transformations. We posit that this reflects the relative simplicity of the semantics of BSL.