A Tested Semantics for Getters, Setters, and Eval in JavaScript

Joe Gibbs Politz   Matthew J. Carroll   Benjamin S. Lerner   Justin Pombrio   Shriram Krishnamurthi

Brown University

`www.jswebtools.org`

Abstract

We present S5, a semantics for the strict mode of the ECMAScript 5.1 (JavaScript) programming language. S5 shrinks the large source language into a manageable core through an implemented transformation. The resulting specification has been tested against real-world conformance suites for the language.

This paper focuses on two aspects of S5: accessors (getters and setters) and `eval`. Since these features are complex and subtle in JavaScript, they warrant special study. Variations on both features are found in several other programming languages, so their study is likely to have broad applicability.

Categories and Subject Descriptors D.3.1 [*Formal Definitions and Theory*]: Semantics; D.3.3 [*Language Constructs and Features*]: Classes and Objects

Keywords JavaScript, LambdaJS, getters, setters, accessors, eval, desugar, wat, wtfjs

1. Introduction

"JavaScript" is the name given to a collection of implementations of the ECMAScript specification [4]. Defining the semantics of this language crisply makes it possible to build tools that can reason about its programs, and making this semantics small and tractable helps implementers keep the number of cases in their tools manageable. As the language evolves, adding new features and modifying existing ones, any semantics effort must evolve, too.

A new major version of the standard, ECMAScript 5.1 (henceforth *ES5*), introduces two new features that demand particular attention. One is an extended object model with richer properties, including *getters* and *setters*; these features interact subtly with object inheritance. The other is an entirely new *strict mode* of the language, which impacts many parts of its behavior, especially `eval`. Since similar features are present or can be simulated in many "scripting" languages, they deserve careful semantic treatment.

Contributions

This paper describes S5, a core semantics for the strict mode of ES5, the subset that future specifications are expected to use (`mail.mozilla.org/pipermail/es-discuss/2011-February/012895.html`). S5 follows in the tradition of λ_{JS} [7], a widely-used semantics for JavaScript. Concretely, this means:

1. S5 offers a *core* semantics of the language, shrinking a fairly large language to under 35 syntactic forms.
2. All source-language features not covered by the S5 core semantics are handled by an implemented *desugaring* function,[1] which translates all source programs to the core.
3. S5 is a *tested* semantics, meaning it has been checked for conformance with real-world test suites. This ensures coverage of the language and hence utility of the semantics.

In addition, S5 makes two contributions to the semantics literature, which form the core of this paper:

4. A semantic account of getters and setters in a prototype-based language, in section 4.
5. An account of JavaScript's `eval` operator*s*, in section 5.3.

As we describe S5, we highlight some of the tradeoffs that induced the design of the core. For readers familiar with λ_{JS}, we also highlight places where S5 differs from that semantics using $\boxed{\lambda_{JS}\ \textbf{Contrast}}$ callout boxes.

Paper Outline We begin by giving the reader a feel for the complexity that lurks beneath JavaScript's familiar-looking surface syntax (section 2), and discuss how we engineer S5 to handle JavaScript's strict mode and more (section 3). We then focus on objects in S5, with an account of getters and setters and their interaction with inheritance (section 4). We then turn to `eval`, which requires an in-depth discussion of environments (section 5). This paper focuses on the key semantics contributions; it does *not* present the full S5 semantics, which is quite large and possibly only of interest to its actual users. The full semantics is available in the appendix, or at `www.cs.brown.edu/research/plt/dl/s5/`.

2. The Complexity of JavaScript

JavaScript is sometimes thought of as a small language with a Scheme-like core enhanced with Self-like objects.[2] However, the full language contains numerous instances of overloading and other sources of complexity. Consider the following JavaScript program from a popular talk:[3]

```
[] + {}
```

[1] Technically, the term "desugaring" abuses language, because the generated language is not a strict subset of the source. We continue to use the term "desugar" because the term is more evocative of what this transformer does than the generic term "compiler".

[2] `www.crockford.com/javascript/little.html`

[3] `www.destroyallsoftware.com/talks/wat/`

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

DLS'12, October 22, 2012, Tucson, Arizona, USA.
Copyright © 2012 ACM 978-1-4503-1564-7/12/10. . . $10.00

```
 1 > ./desugar "[] + {}"
 2 {let (%context = %nonstrictCxt)
 3     {let (#strict = false)
 4          %PrimAdd({[#proto: %ArrayProto,
 5                     #class: "Array",
 6                     #extensible: true,]
 7                   'length' : {#value 0. ,
 8                               #writable true ,
 9                               #configurable false}},
10                  {[#proto: %ObjectProto,
11                    #class: "Object",
12                    #extensible: true,]})}}
```

Figure 1. Desugaring `[] + {}`

This program evaluates to the string `"[object Object]"`. Why? Consider also this program from a popular Web site:[4]

```
(![]+[])[+[]]+(![]+[])[+!+[]]+([![]]+[][[]])
[+!+[]+[+[]]]+(![]+[])[!+[]+!+[]]
```

This program evaluates to the string `"fail"`. Why?

To tackle the first example, we must inspect the definition of the + operation. The specification is an 8-step algorithm that roughly instructs the evaluator to proceed as follows:

1. Invoke the ToPrimitive internal function on both arguments.
2. If the argument to ToPrimitive is an object (as in the case of arrays), invoke the valueOf method on the argument.
3. If the call to valueOf returns a primitive value, return that value. Otherwise, call the toString method on the argument, and return the result of that call (if it's primitive).
4. If the result of ToPrimitive is a string for either argument, concatenate the string representations of both arguments.

In the first case, the valueOf method returns a non-primitive object for both the empty array and for the object with no fields. This invokes toString, which produces the empty string for the empty array, and `"[object Object]"` for all objects with no fields. Concatenating the appropriate results together gives the answer `"[object Object]"`.

The second example is more complicated, but builds up a string by (ab)using built-in string and number conversions, along with the ability to index the characters of a string. As a hint, the initial code fragment `(![]+[])` evaluates to `"false"`; `+[]` evaluates to `0`, which is used as an index; the `0`th index of `"false"` is `"f"`; and so on.

Whether the reader finds these examples entertaining or horrifying, these represent the reality of JavaScript, and such obfuscations can thwart simplistic filters. The evaluation of + depends on several pages of metafunctions including multiple primitive operators, two instances of prototype lookup (one that goes two levels deep, since the Array prototype delegates to the Object prototype for `valueOf`), and several possible exceptions along the way. What should a tool designer do when confronted with this kind of complexity? Desugaring provides the answer.

The First Example, Desugared S5 makes the hidden complexity of JavaScript programs manifest in the *desugaring* process. All the algorithms of the spec are themselves implemented in the core language of S5, so that a tool only needs to comprehend this core language to be able to process any JavaScript program.

As an illustration, figure 1 shows the desugared S5 program corresponding to the JavaScript program `[] + {}`. The resulting program relies upon several identifiers that we define separately in `es5.env` (figure 2), an S5 program that defines the environment for desugared programs. The desugared code sets up some variables (lines 2 and 3), then applies a function called %PrimAdd defined in the environment. The two arguments to this application of %PrimAdd are both object literals in S5. The first (lines 4–9), the array literal `[]` in the original program, has the prototype %ArrayProto, and the second (lines 10–12) has the prototype %ObjectProto.[5] %PrimAdd explicitly performs the type conversions, type-tests, and primitive operations defined in the spec. `%ToPrimitive` (not shown) invokes the toString method of both arguments; the environment specifies this behavior on the array and object prototypes, which provide the strings that give this example's result.

es5.env

```
let (%PrimAdd = func(left, right) {
  let (leftPrim = %ToPrimitive(right))
  let (rightPrim = %ToPrimitive(right))
  if(typeof leftPrim === 'string') {
    prim("string+", prim("prim->str", leftPrim),
                    prim("prim->str", rightPrim))
  }
  ...
}
...
let (%ObjectToString = func(this, args) {
  prim("string+", "[object ",
    prim("string+", this<#class>, "]"))
})
%ObjectProto["toString" = {[#code: %ObjectToString]}]
...
let (%ArrayToString = func(this, args)) {
  /* join array elements with commas */
}
%ArrayProto["toString"] = {[#code: %ArrayToString]}]
```

Figure 2. A portion of the environment used by `[] + {}`

3. The Engineering of S5

When combined, the environment and the desugared program provide a precise description of a program's behavior. In fact, S5 is Gallic: it is properly divided into three parts.

1. A core language with a small-step operational *semantics*, which provides a formal description of key features.
2. An implementation *in the S5 core language* of a portion of ES5's built-in libraries. This is the *environment*.
3. A *desugaring* function from ES5 programs to programs in the core language.

It is easy to build an interpreter for the S5 core language. We have implemented one in 500 lines of Ocaml, so we can read and execute both S5 environments and S5 programs. The environment we report results for here is about 5000 LOC, implemented in the S5 core language. We use SpiderMonkey (`developer.mozilla.org/en/SpiderMonkey`) as a front-end to parse ES5 source, rather than implementing a parser ourselves. The desugaring function, which is also implemented in (1000 lines of) Ocaml, converts programs in SpiderMonkey's ES5 AST into S5 programs.

Composing a parser, desugaring function, and interpreter gives us another implementation of the ES5 language. We can therefore test it against the ES5 conformance suite (`test262.ecmascript.org`). In addition, whenever we find interesting examples on the Web—such as those of section 2—we add those to our own test suite. We

[4] `wtfjs.com/2010/07/12/fail`

[5] There are other details of the objects shown that are not germane to this example; we will discuss objects more fully in section 4.

Chapter	Passed	Total	Percent passed
07 Lexical Conventions	648	715	90%
08 Types	163	184	89%
09 Type Conversions	102	136	75%
10 Execution Contexts	349	377	93%
11 Expressions	1178	1326	89%
12 Statements	448	532	84%
13 Function Definitions	208	236	88%
14 Programs	22	24	92%
15 Built-in Objects	5039	8076	62%
Totals	8157	11606	70%

Figure 3. The full test suite

Chapter	Passed	Total	Percent passed
07 Lexical Conventions	78	78	100%
08 Types	12	12	100%
09 Type Conversions	0	0	N/A
10 Execution Contexts	181	181	100%
11 Expressions	156	156	100%
12 Statements	60	60	100%
13 Function Definitions	104	104	100%
14 Programs	2	2	100%
15 Built-in Objects	44	53	83%
Totals	637	646	98%

Figure 4. Tests designed specifically for strict mode

therefore constantly improve S5 to attain conformance. In practice, most of our effort is concentrated in implementing desugaring and the environment: once we attained a reasonable coverage of the language's primary features, the definition of the core language stabilized and we have seldom needed to change it. Indeed, the core language has changed only twice in the past year (starting when conformance was below 50%).

Our current coverage of ES5 conformance tests is shown, by chapter, in figure 3. (The first six chapters only provide context for the rest of the spec, and are thus not testable.) The test suite is dominated by tests of built-in objects (like `Regexp` and `Date`), of which we have implemented only 60%. This explains our failure rate in chapter 15, which tests built-in objects. A number of failures in the other chapters are entirely due to our adherence to strict mode. Tests for the semantics of getters, setters, and `eval` are present throughout the test suite, but are mostly focused in section 8.2 (which tests built-in operations on objects) and chapter 10 (which tests the use of the correct scope when using one of the `eval` operations).

Some parts of the test suite are explicitly marked as being for strict mode. This matters because many features behave differently in strict and non-strict mode. On the strict mode tests, we achieve perfect results excepting built-in objects (figure 4). As the reader will notice, however, we pass more than just the strict-mode tests. In part, this is because some tests produce the same results in both modes. In addition, though S5 is technically only for strict mode, we have built some extensions for non-strict features that have restricted or nonexistent strict semantics: e.g., the various modes of `eval` (section 5.3 of this paper), and the **with** operator. Indeed, the non-strict features included in S5 are some of the harder and more interesting ones, such as those that interact with scoping.

4. Objects with Getters and Setters

Designing the semantics of S5 and the corresponding desugaring function involves a number of design decisions. The semantics is supposed to represent the "core" of JavaScript, the definition of which is ultimately a combination of aesthetic and engineering tradeoffs. We illustrate the tradeoffs by showing the decisions we made in modelling and implementing the full range of object features in JavaScript, split across the semantics of S5 and desugaring.

4.1 Mutable Records

We proceed by presenting small examples of JavaScript, followed by a continually growing semantics that models the features from the JavaScript program. We start simply, with mutable records mapping string keys to values:

```
var o = {foo : "init"};
o.foo; // evaluates to "init"
o.bar; // evaluates to undefined
o.bar = "defined";
o.bar; // evaluates to "defined"
o.["f" + "oo"] = "changed";
o.["fo" + "o"]; // evaluates to "changed"
```

Object literals (enclosed in `{}`) create mutable objects. Assignment and lookup can both take arbitrary strings (`o.foo` is sugar for `o["foo"]`). When looking up a property that isn't present, the special **undefined** value results. When assigning a property that isn't present, it is added to the object. A simple store-based semantics can capture this functionality:

$$\begin{array}{rcl}
v & := & r \mid \mathsf{str} \mid \textbf{undefined} \\
e & := & v \mid \texttt{\{}\mathsf{str}_1\texttt{:}e_1,\cdots\texttt{\}} \mid e\texttt{[}e\texttt{]} \mid e\texttt{[}e\ \texttt{=}\ e\texttt{]} \\
\theta & := & \texttt{\{}\mathsf{str}_1\texttt{:}v_1,\cdots\texttt{\}} \\
\Theta & := & \cdot \mid r : \theta, \Theta \\
E & := & \bullet \mid \texttt{\{}\mathsf{str}_1\texttt{:}v_1,\cdots\texttt{,}\ \mathsf{str}\texttt{:}E,\cdots\mathsf{str}_n\texttt{:}e_n\texttt{\}} \mid E\texttt{[}e\texttt{]} \mid v\texttt{[}E\texttt{]} \\
 & \mid & E\texttt{[}e\ \texttt{=}\ e\texttt{]} \mid v\texttt{[}E\ \texttt{=}\ e\texttt{]} \mid v\texttt{[}v\ \texttt{=}\ E\texttt{]}
\end{array}$$

$$\boxed{\Theta; e \rightarrow \Theta; e}$$

$$\begin{array}{rl}
\text{(E-Compat)} & \Theta; E[e] \rightarrow \Theta'; E[e'] \\
 & \text{when } \Theta; e \Rightarrow \Theta'; e'
\end{array}$$

$$\boxed{\Theta; e \Rightarrow \Theta; e}$$

$$\begin{array}{rl}
\text{(E-Object)} & \Theta; \theta \Rightarrow r : \theta, \Theta; r \quad r \text{ fresh} \\
\text{(E-GetField)} & \Theta; r\texttt{[}\mathsf{str}\texttt{]} \Rightarrow \Theta; v \\
 & \text{when } \Theta(r) = \texttt{\{}\cdots\ \mathsf{str}\texttt{:}v,\cdots\texttt{\}} \\
\text{(E-NoField)} & \Theta; r\texttt{[}\mathsf{str}\texttt{]} \Rightarrow \Theta; \textbf{undefined} \\
 & \text{when } \Theta(r) = \texttt{\{}\mathsf{str'}\texttt{:}v,\cdots\texttt{\}} \\
 & \text{and } \mathsf{str} \notin \{\mathsf{str'},\cdots\} \\
\text{(E-SetField)} & \Theta; r\texttt{[}\mathsf{str}\ \texttt{=}\ v'\texttt{]} \Rightarrow \Theta'; v' \\
 & \text{when } \Theta(r) = \texttt{\{}\cdots\ \mathsf{str}\texttt{:}v,\cdots\texttt{\}} \\
 & \text{and } \Theta' = \Theta[r/\texttt{\{}\cdots\ \mathsf{str}\texttt{:}v',\cdots\texttt{\}}] \\
\text{(E-AddField)} & \Theta; r\texttt{[}\mathsf{str}\ \texttt{=}\ v\texttt{]} \Rightarrow \Theta'; v \\
 & \text{when } \Theta(r) = \texttt{\{}\mathsf{str'}\texttt{:}v',\cdots\texttt{\}} \\
 & \text{and } \mathsf{str} \notin \{\mathsf{str'},\cdots\} \\
 & \text{and } \Theta' = \Theta[r/\texttt{\{}\mathsf{str}\texttt{:}v\texttt{,}\mathsf{str'}\texttt{:}v',\cdots\texttt{\}}]
\end{array}$$

We use this small calculus to introduce the notation of this paper. Expressions e can be object literals, with string literals as names and expressions as properties; lookup expressions, where both object and name position are expressions; and assignment expressions, where again all pieces are expressions. Values v can be object references r, strings `str`, or the special value **undefined**. Objects whose properties are all values, rather than expressions, are written θ. Object stores Θ map references to these value-objects. Finally, evaluation contexts E enforce left-to-right evaluation of expressions.

To evaluate expressions, we use the $\Rightarrow$ relation over active expressions. E-Object takes fully-evaluated object literals θ and adds them to the object store Θ, performing an allocation. E-GetField checks that a property with the provided string name is present, and yields the corresponding value, with lookup in the object store written $\Theta(r)$. E-NoField evaluates to **undefined** if the property isn't present. E-SetField replaces a value with a new one if it is present, changing Θ to Θ', with replacement of an object at a particular location written $\Theta[r/\theta]$. E-AddField adds a new property entirely if no such property is present in assignment, also yielding a new Θ'.

$\boxed{\lambda_{JS}\ \textit{Contrast}}$ λ_{JS} has functional records along with **ref** and **deref** expressions, in the style of ML; S5 has only mutable records. One lesson from the desugaring function implemented for λ_{JS} was that separating these features merely led to more verbose compiled code, not any increase in understanding. Because mutable records have a well-understood semantics and more closely match JavaScript's behavior, we use them instead in S5. □

4.2 Accessor Properties

This tiny calculus models mutable records, which are only a small part of the behavior of JavaScript's objects. In particular, *getters* and *setters* (known as *accessors*) are a significant addition in ES5. They extend the behavior of assignment and lookup expressions on JavaScript objects. If a property has a getter defined on it, rather than simply returning the value in property lookup, a getter function is invoked, and its return value is the result of the lookup:

```
var timesGotten = 0;
var o = {get x() { timesGotten++; return 22; }};
o.x; // calls the function above, evaluates to 22
timesGotten; // is now 1, due to the increment
o.x; // calls the function again, still evaluates to 22
timesGotten; // is now 2, due to another increment
```

Similarly, if a property has a setter defined on it, the setter function is called on property update. The setter function gets the assigned value as its only argument, and its return value is ignored:

```
var foo = 0;
var o = {set x(v) { foo = v; }, get x() { return foo; }};
o.x = 37; // calls the function above (with v=37)
foo; // evaluates to 37
o.x; // evaluates to 37
```

The first thing these examples tell us is that there are two kinds of properties we need to model: simple values, and accessors that have one or both of a getter and a setter. Property lookup and assignment expressions perform the appropriate action based on what kind of property is present. In our semantics, we introduce two kinds of property, and change objects to refer to contain one or the other (this corresponds to *property descriptors* in section 8.10 of the spec). We introduce functions and applications as well, using ... to indicate elided, previously-defined definitions, and shading to indicate altered parts of the production.

$$
\begin{array}{rcl}
v & := & \ldots \mid \textbf{func}\ (x, \cdots)\ \{\ e\ \} \\
pe & := & [\textbf{value}\!:\!e] \mid [\textbf{get}\!:\!e,\ \textbf{set}\!:\!e] \\
pv & := & [\textbf{value}\!:\!v] \mid [\textbf{get}\!:\!v,\ \textbf{set}\!:\!v] \\
p & := & pv \mid \texttt{[]} \\
e & := & \ldots \mid \{\mathsf{str}_1 : pe_1, \cdots\} \mid e(e,\ \cdots) \mid x \\
\theta & := & \{\mathsf{str}_1 : pv_1, \cdots\} \\
\Theta & := & \ldots \\
E_p & := & [\textbf{value}\!:\!E] \mid [\textbf{get}\!:\!E,\ \textbf{set}\!:\!e] \mid [\textbf{get}\!:\!v,\ \textbf{set}\!:\!E] \\
E & := & \ldots \mid \{\mathsf{str}_1 : pv_1, \cdots,\ \mathsf{str}\!: E_p, \cdots \mathsf{str}_n : pe_n\} \\
 & \mid & E(e,\ \cdots) \mid v(v_1, \cdots E, e_1, \cdots)
\end{array}
$$

Since the kind of property descriptor determines what the lookup or assignment expression does, we split apart these features in our semantics. We define a relation, $\Theta; r\,\texttt{[}\mathsf{str}\texttt{]} \Downarrow p$, that yields the appropriate property descriptor, or the empty descriptor `[]`, for a given reference and string:

$$\boxed{\Theta; r\,\texttt{[}\mathsf{str}\texttt{]} \Downarrow p}$$

$$
\frac{\Theta(r) = \{\cdots\ \mathsf{str}\!:\!pv, \cdots\}}{\Theta; r\,\texttt{[}\mathsf{str}\texttt{]} \Downarrow pv}
\qquad
\frac{\Theta(r) = \{\mathsf{str'}\!:\!pv, \cdots\} \quad \mathsf{str} \not\in \{\mathsf{str'}, \cdots\}}{\Theta; r\,\texttt{[}\mathsf{str}\texttt{]} \Downarrow \texttt{[]}}
$$

We can then define lookup and assignment relative to this:

$$\boxed{\Theta; e \Rightarrow \Theta; e}$$

(E-Object) as before

(E-GetField)

$$\frac{\Theta; r\,\texttt{[}\mathsf{str}\texttt{]} \Downarrow [\textbf{value}\!:\!v]}{\Theta; r\,\texttt{[}\mathsf{str}\texttt{]} \Rightarrow \Theta; v}$$

(E-Getter)

$$\frac{\Theta; r\,\texttt{[}\mathsf{str}\texttt{]} \Downarrow [\textbf{get}\!:\!v_g,\ \textbf{set}\!:\!v]}{\Theta; r\,\texttt{[}\mathsf{str}\texttt{]} \Rightarrow \Theta; v_g()}$$

(E-NoField)

$$\frac{\Theta; r\,\texttt{[}\mathsf{str}\texttt{]} \Downarrow \texttt{[]}}{\Theta; r\,\texttt{[}\mathsf{str}\texttt{]} \Rightarrow \Theta; \textbf{undefined}}$$

(E-SetField)[6]

$$\frac{\begin{array}{c}\Theta; r\,\texttt{[}\mathsf{str}\texttt{]} \Downarrow [\textbf{value}\!:\!v] \\ \Theta(r) = \{\cdots\ \mathsf{str}\!:[\textbf{value}\!:\!v], \cdots\} \\ \Theta' = \Theta[r/\{\cdots\ \mathsf{str}\!:[\textbf{value}\!:\!v\texttt{'}], \cdots\}]\end{array}}{\Theta; r\,\texttt{[}\mathsf{str}\ \texttt{=}\ v\texttt{']} \Rightarrow \Theta'; v'}$$

(E-AddField)

$$\frac{\begin{array}{c}\Theta; r\,\texttt{[}\mathsf{str}\texttt{]} \Downarrow \texttt{[]} \\ \Theta(r) = \{\mathsf{str'}\!:\!pv, \cdots\} \\ \Theta' = \Theta[r/\{\mathsf{str}\!:[\textbf{value}\!:\!v],\ \mathsf{str'}\!:\!pv, \cdots\}]\end{array}}{\Theta; r\,\texttt{[}\mathsf{str}\ \texttt{=}\ v\texttt{']} \Rightarrow \Theta'; v'}$$

(E-Setter)

$$\frac{\Theta; r\,\texttt{[}\mathsf{str}\texttt{]} \Downarrow [\textbf{get}\!:\!v_g,\ \textbf{set}\!:\!v_s]}{\Theta; r\,\texttt{[}\mathsf{str}\ \texttt{=}\ v\texttt{']} \Rightarrow \Theta; v_s\texttt{(}v\texttt{')}}$$

(E-App)

$$\begin{array}{c}\Theta; \textbf{func}\texttt{(}x_1, \cdots\texttt{)}\ \{\ e\ \}\texttt{(}v_1, \cdots\texttt{)} \\ \Rightarrow \Theta; e[x_1/v_1, \cdots]\end{array}$$

We can thus summarize the behavior of getters and setters. If the property is an accessor in lookup, the getter function is called (E-Getter). If it is an accessor in assignment, the setter is called with the assigned value passed as an argument (E-Setter). In other cases, behavior is the same as before, just with raw values changed to property descriptors with a **value** attribute, and the check for missing properties is now implicit in $\Downarrow$ `[]`. The rewriting to use $\Downarrow$ makes the setting rules a touch more verbose, but will be invaluable in the next section, so we introduce it here. Finally, to fully evaluate these terms, we include application, which performs capture-avoiding substitution of values for identifiers. This small semantics is adequate to capture the essence of the examples presented so far.

4.3 Object-Oriented JavaScript

Consider this program:

```
var o = {
  // The property _x stores data shared between
  // these two functions
  get x() { return this._x + 1; },
  set x(v) { this._x = v * 2; }
```

[6] The first two antecedents here are redundant. In the next section, for a rule similar to this one, they are not, so we present the rule this way for contrast.

```
};
o.x = 5; // calls the set function above (with v=5)
o._x; // evaluates to 10, because of assignment in setter
o.x; // evaluates to 11, because of addition in getter
```

We see that the functions have access to the target object of the assignment or lookup, via **this**. But JavaScript also has prototype inheritance. Let's start with an object called `parent`, and make it the prototype of another object using the `Object.create()` method:

```
var parent = {
  get x() { return this._x + " was gotten"; },
  set x(v) { this._x = v; }
};
var child = Object.create(parent);
child.x = "set value"; // Sets... what exactly?
parent._x; // evaluates to undefined
child._x; // evaluates to "set value"
parent.x; // evaluates to "undefined was gotten"
child.x; // evaluates to "set value was gotten"
```

Here, JavaScript is passing the object *in the lookup expression* into the function, for both property access and update. This is in contrast to if it passed in the prototype object itself (`parent` in the example above). Something else subtle is going on, as well. Recall that before, when an update occurred on a property that wasn't present, JavaScript simply added it to the object. Now, on property update, we see that the assignment traverses the prototype chain to check for setters before adding it to the top level object. This is fundamentally different than JavaScript without accessors, where assignment never considered prototypes. Our semantics thus needs to do three new things:

- define prototypes on objects;
- traverse the prototype chain to look for properties; and,
- pass the correct **this** argument to getters and setters.

Prototypes Prototypes aren't properties[7]—they can't be accessed with property lookup—but are instead *object attributes*. We add a special attribute space to objects for the prototype:

$$\begin{array}{rcl} e & ::= & \ldots \mid \{\,[\mathbf{proto}\!:\!e]\ \ \mathsf{str}\!:\!pe,\cdots\} \\ \theta & ::= & \ldots \mid \{\,[\mathbf{proto}\!:\!v]\ \ \mathsf{str}\!:\!pv,\cdots\} \\ E & ::= & \ldots \mid \{\,[\mathbf{proto}\!:\!E]\ \ \mathsf{str}\!:\!pe,\cdots\} \end{array}$$

Note that now, providing an implementation of `Object.create` from section 15.2.3.5 of the specification is straightforward. Its core is just this function (using a few obvious, but yet-to-be introduced features):

```
func(prototype) {
  if(typeof prototype !== "object") { throw TypeError() }
  else { {[proto: prototype]} }
}
```

This is our first example of translating a spec feature into an S5 program, rather than modelling it directly in the semantics. We choose to add the object attribute **proto** as a minimal feature. Then, we use it to implement `Object.create` to avoid polluting our semantics with type tests and non-essential features. The essence of the semantics is the prototype attribute; with that, we can implement this and several other features of the spec.

$\boxed{\lambda_{JS}\ \textbf{\textit{Contrast}}}$ λ_{JS} did not have attributes on objects, and instead encoded the prototype as a property. Compare these two (abbreviated) desugarings of `{"a-prop": 42}`:

```
{[proto: %ObjectProto]},
  "a-prop":
    [ value:42 ] }
```

S5

```
{"__proto__": %ObjectProto,
 "a-prop": 42 }
```

λ_{JS}

The λ_{JS} desugaring admits JavaScript programs that look up the string `"__proto__"`, while the semantics doesn't allow this. This mismatch doesn't come up in most test suites, but it is still not a faithful representation of the underlying semantics. In S5, properties and attributes are separate and each fill a specific role, removing this inelegance. □

To model prototype inheritance, we need to augment $\Downarrow$. Finding a property is unchanged aside from the addition of **proto**. To yield the empty descriptor `[]`, the property must not be present and the **proto** must be **undefined**. Finally, if the property isn't present, but a prototype object is, the prototype is consulted:

$$\boxed{\Theta; r\,[\mathsf{str}] \Downarrow p}$$

$$\frac{\Theta(r) = \{\,[\mathbf{proto}\!:\!v]\cdots\ \ \mathsf{str}\!:\!pv,\cdots\}}{\Theta; r\,[\mathsf{str}] \Downarrow pv}$$

$$\frac{\begin{array}{c}\Theta(r) = \{\,[\mathbf{proto}\!:\!\mathbf{undefined}]\ \ \mathsf{str'}\!:\!pv,\cdots\} \\ \mathsf{str} \notin \mathsf{str'}\cdots\end{array}}{\Theta; r\,[\mathsf{str}] \Downarrow \texttt{[]}}$$

$$\frac{\begin{array}{c}\Theta(r) = \{\,[\mathbf{proto}\!:\!r']\ \ \mathsf{str'}\!:\!pv,\cdots\} \\ \mathsf{str} \notin \mathsf{str'}\cdots \qquad \Theta; r'\,[\mathsf{str}] \Downarrow p\end{array}}{\Theta; r\,[\mathsf{str}] \Downarrow p}$$

$\boxed{\lambda_{JS}\ \textbf{\textit{Contrast}}}$ λ_{JS} uses an explicit small-step reduction for prototype lookup:

$$\{\cdots\ \mathsf{\_\_proto\_\_}\!:\!v_p,\cdots\}\,[\mathsf{str}] \rightarrow (\mathbf{deref}\ \ v_p)\,[\mathsf{str}]$$

In contrast, S5 abstracts prototype lookup into $\Downarrow$. Since the semantics of S5 needs prototype lookup for both property assignment and property access, this abstraction allows the definition of prototype lookup to be identical and shared between these two forms in the $\Downarrow$ relation. □

***Accessors and* this** If we leave the rules as they are (with the minor addition of prototypes), they will look up the correct properties, but won't pass the this argument to the getter and setter functions. This is easily remedied: the reference r in the lookup or assignment expression is precisely the correct this argument to pass to the function. This minor change is all that's required:

$$\boxed{\Theta; e \Rightarrow \Theta; e}$$

$$\ldots$$

$$\text{(E-Getter)}\quad \frac{\Theta; r\,[\mathsf{str}] \Downarrow [\mathbf{get}\!:\!v_g\,,\ \ \mathbf{set}\!:\!v]}{\Theta; r\,[\mathsf{str}] \Rightarrow \Theta; v_g\,(\,r\,)}$$

$$\text{(E-Setter)}\quad \frac{\Theta; r\,[\mathsf{str}] \Downarrow [\mathbf{get}\!:\!v_g\,,\ \ \mathbf{set}\!:\!v_s]}{\Theta; r\,[\mathsf{str}\ =\ v'] \Rightarrow \Theta; v_s\,(\,r\,,v'\,)}$$

Accessors and the* arguments *Object Thus far, accessors have had a distinguished position as functions within special properties on objects. In reality, any ES5 function can be used as a getter or a setter. To ensure that this is allowed in S5, we need to handle one

[7] Though some browsers providing access to the prototype through a field named `__proto__`, this is not the intent of the specification.

final wrinkle. The specification mandates that each function bind a special variable called `arguments` to an object which is populated with all of the parameter values at the time of application. When a getter is invoked during property lookup, it is handed an argument object with no parameters. Setters have the value stored in the `0`th argument. For example:

```
var x = "init";
var f = function() { x = arguments["0"]; };
var o = Object.create({}, { "fld": {get: f, set: f} });
o.fld; // evaluates to undefined
x; // evaluates to undefined
o.fld = "setter";
x; // evaluates to "setter"
f("function call");
x; // evaluates to "function call"
```

To accomplish this, we add a distinguished position holding an arguments object to property access and assignment statements. We write the modified expressions as $e[e]^e$ and $e[e{=}e]^e$. When desugaring, we add an expression that constructs the appropriate arguments object for these expressions. Figure 6 shows how the arguments object position is propagated to the invocation for both getters and setters.

4.4 Configuring, Enumerating, and Extending

Having discussed the interesting parts of object lookup, we now briefly dwell on the details that are needed to create a semantics specifically for ES5. ES5 has a few more attributes that allow for greater control over access to properties and affect objects' behavior. A simple example:

```
var o = {};
o.x = "add x";
Object.preventExtensions(o);
o.y = "add y";
o.y; // evaluates to undefined
o.x; // evaluates to "add x"
o.x = "change x";
o.x; // evaluates to "change x"
```

`Object.preventExtensions` makes subsequent property updates to `o` unable to add new properties. Existing properties can be changed, however. There is another object attribute, **extensible**, that is a flag for whether new properties can be added to an object or not. Only addition is prevented; existing properties can still be changed, as evidenced by the assignment to `"change x"` above.

As another example, the `Object.freeze()` method can also affect access to properties by both making new additions impossible and preventing all further changes to properties:

```
var o = {};
o.x = "add x";
Object.freeze(o);
o.y = "add y";
o.y; // evaluates to undefined
o.x; // evaluates to "add x"
o.x = "change x";
o.x; // evaluates to "add x"
```

Here, the assignment to `"change x"` doesn't affect the `x` property at all; it retains the value it had before the `freeze`. For the most fine-tuned control, ES5 provides `Object.defineProperty`. This built-in can change data properties to accessors and vice versa:

```
var tmp1, tmp2 = "tmp2 init";
var o = {
  x: "x init",
  get y() { return "y getter";},
  set y(v) { tmp2 = v; }
};
// defineProperty can change a data property to accessor...
Object.defineProperty(o, "x", {
  get: function() { return "in getter"; },
  set: function(v) { tmp1 = v; }
});
o.x; // evaluates to "in getter", rather than "x init"
o.x = "change x";
o.x; // evaluates to "in getter", unchanged by assignment
tmp1; // evaluates to "change x", set in setter
// and vice versa...
Object.defineProperty("y", {value: "y data", writable: true});
o.y; // evaluates to "y data", not "y getter"
o.y = "set y";
o.y; // evaluates to "set y"
tmp2; // evaluates to "tmp2 init", never set in setter
```

It can also have interesting effects on iteration:

```
var o = {x: "foo", y: "bar", z: "baz"};
for(var i in o) { print(i); }
// prints "foo", "bar", "baz"
Object.defineProperty(o, "y", {enumerable: false});
for(var i in o) { print(i); }
// prints "foo", "baz"
```

It can also alter properties to prevent any future changes of these kinds:

```
var o = {x: "foo"};
Object.defineProperty(o, "x", {configurable: false});
Object.defineProperty(o, "x", {
  set: function(v) { o._x = v; }
});
// Error, can't change non-configurable property
```

In summary, there are six different attributes a property can have. We have already addressed **value**, **get**, and **set**, but they can also have **writable**, **enum**, and **config**. There's also another attribute needed for objects, **extensible**. In the full specification, objects' attribute lists have been augmented, and properties now always have one of two forms: a *data* property with **writable** and **value** attributes, or an *accessor* property with **set** and **get** attributes. Both kinds of properties have **enum** and **config** attributes.

The attributes **writable** and **extensible** change the way the expressions we've shown so far evaluate. In particular, **writable** must be true to either update a property or add a new one (if the property exists on the prototype somewhere), and **extensible** must be true to extend the object with the property. These new constraints are shown in the final definition of property access figure 6. This shows another way—in addition to setters—in which property assignment must consider prototypes.

To accompany all these attributes, we've added five new kinds of expressions. We can access object attributes with e[<o>], and update them with e[<o> = e]. This lets us set, for example, **extensible** to **false**. Similarly, property attributes can be accessed with e[e<a>] and updated with e[e<a> = e], in order to perform the operations `defineOwnProperty` requires. The semantics of these new operations is in figure 5.

These operators manipulate attributes at a low level, while ES5 specifies a number of high-level property-altering methods (such as `Object.seal`, `Object.freeze`, and `Object.create`). Rather than express each of these methods as its own special-purpose reduction step, we define them in the environment as functions that apply combinations of the low-level operators.

Maintaining Invariants Operations that prevent future changes, like `freeze` and `seal`, are designed for security. The specification of these operations was carefully defined to be monotonic; an application that restricts the object's behavior cannot be undone. For example, once `freeze` sets the configurable attribute to **false**, it cannot be reset to **true**. We have a choice of implementation for

$$\boxed{\Theta; e \rightarrow^{\Theta} \Theta; e}$$

$$\text{E-GetObjAttr}\ \frac{\Theta(r) = \{[\cdots o : v \cdots]\ \ \texttt{str}:pv, \cdots\}}{\Theta; r[\texttt{<}o\texttt{>}] \rightarrow^{\Theta} \Theta; v}
\qquad
\text{E-SetObjAttr}\ \frac{\begin{array}{c}\Theta(r) = \{[\cdots o : v_o \cdots]\ \ \texttt{str}:pv, \cdots\} \\ \Theta' = \Theta[r/\{[\cdots o : v \cdots]\ \ \texttt{str}:pv, \cdots\}]\end{array}}{\Theta; r[\texttt{<}o\texttt{>}\ \texttt{=}\ v]] \rightarrow^{\Theta} \Theta'; v}$$

$$\text{E-GetPropAttr}\ \frac{\Theta(r) = \{av\ \cdots \texttt{str}:[\cdots a : v \cdots], \cdots\}}{\Theta; r[\texttt{str<}a\texttt{>}] \rightarrow^{\Theta} \Theta; v}
\qquad
\text{E-SetPropAttr}\ \frac{\begin{array}{c}\Theta(r) = \{av\ \cdots \texttt{str}:[\cdots a : v_a \cdots], \cdots\} \\ \Theta' = \Theta[r/\{av\ \cdots \texttt{str}:[\cdots a : v \cdots], \cdots\}] \\ \mathit{okupdate}([\cdots a : v_a \cdots], a, v)\end{array}}{\Theta; r[\texttt{str<}a\texttt{>}\ \texttt{=}\ v] \rightarrow^{\Theta} \Theta'; v}$$

Figure 5. Manipulating object attributes

$$\boxed{\Theta; e \rightarrow^{\Theta} \Theta; e}$$

$$\text{E-GetField}\ \frac{\Theta; r[\texttt{str}] \Downarrow [\cdots\ \textbf{value}:v\ \ \cdots]}{\Theta; r[\texttt{str}]^{v_a} \rightarrow^{\Theta} \Theta; v}$$

$$\text{E-Getter}\ \frac{\Theta; r[\texttt{str}] \Downarrow [\cdots\ \textbf{get}:v_g\ \ \cdots]}{\Theta; r[\texttt{str}]^{v_a} \rightarrow^{\Theta} \Theta; v_g(r, v_a)}
\qquad
\text{E-NoField}\ \frac{\Theta; r[\texttt{str}] \Downarrow []}{\Theta; r[\texttt{str}]^{v_a} \rightarrow^{\Theta} \Theta; \textbf{undefined}}$$

$$\text{E-SetField}\ \frac{\begin{array}{c}\Theta(r) = \{pv \cdots\ \texttt{str}:[\cdots\ \textbf{value}:v\texttt{'},\ \textbf{writable}:\textbf{true}], \cdots\} \\ \Theta' = \Theta[r/\{pv \cdots\ \texttt{str}:[\cdots\ \textbf{value}:v,\ \textbf{writable}:\textbf{true}], \cdots\}]\end{array}}{\Theta; r[\texttt{str=}v]^{v_a} \rightarrow^{\Theta} \Theta'; v}$$

$$\text{E-Setter}\ \frac{\Theta; r[\texttt{str}] \Downarrow [\cdots\ \textbf{set}:v_s\ \ \cdots]}{\Theta; r[\texttt{str}]^{v_a} \rightarrow^{\Theta} \Theta; v_s(r, v_a)}$$

$$\text{E-AddField}\ \frac{\begin{array}{c}\Theta(r) = \{[\textbf{extensible}:\textbf{true}\ \cdots]\ \ \texttt{str'}:av, \cdots\} \qquad \Theta; r[\texttt{str}] \Downarrow [] \\ \Theta' = \Theta[r/\{[\textbf{extensible}:\textbf{true}\ \cdots]\ \ \texttt{str}:[\textbf{config}:\textbf{true},\ \textbf{enum}:\textbf{true},\ \textbf{value}:v,\ \textbf{writable}:\textbf{true}],\ \texttt{str'}:av, \cdots\}]\end{array}}{\Theta; r[\texttt{str=}v]^{v_a} \rightarrow^{\Theta} \Theta'; v}$$

$$\text{E-ShadowField}\ \frac{\begin{array}{c}\Theta(r) = \{[\textbf{extensible}:\textbf{true}\ \cdots]\ \ \texttt{str'}:av, \cdots\} \qquad \Theta; r[\texttt{str}] \Downarrow [\cdots\ \textbf{writable}:\textbf{true}\ \cdots] \\ \Theta' = \Theta[r/\{[\textbf{extensible}:\textbf{true}\ \cdots]\ \ \texttt{str}:[\textbf{config}:\textbf{true},\ \textbf{enum}:\textbf{true},\ \textbf{value}:v,\ \textbf{writable}:\textbf{true}],\ \texttt{str'}:av, \cdots\}] \\ \texttt{str} \not\in \texttt{str'} \cdots\end{array}}{\Theta; r[\texttt{str=}v]^{v_a} \rightarrow^{\Theta} \Theta'; v}$$

Figure 6. Property access and assignment

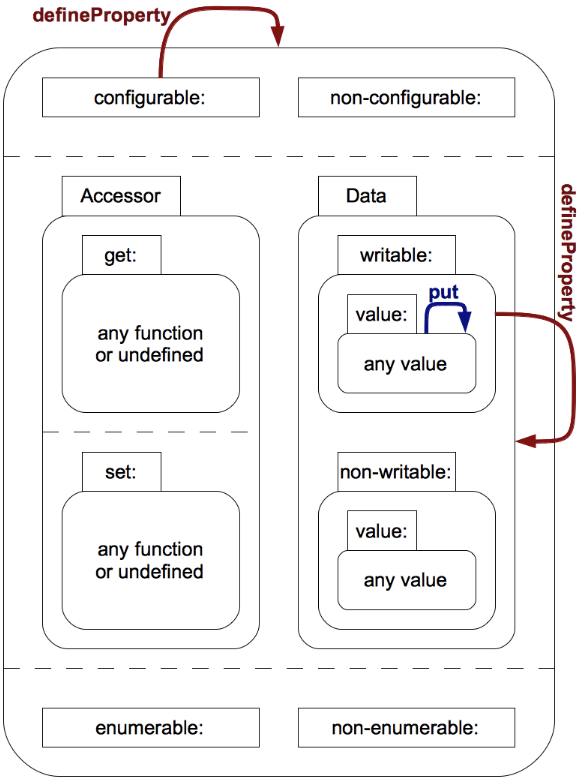

Figure 7. Statecharts diagram of *okupdate*. If configurable is **true**, then any transition is allowed. If **writable** is **true**, then either the **value** or the **writable** property can be changed, but **writable** can only be changed to **false**. No other changes are allowed. The diagram is presented relative to the specification's algorithms; in S5, all of the edges correspond to uses of E-SetPropAttr.

these restrictions. We could simply have an attribute assignment with no checking in the semantics, and desugar all of the checks. Alternatively, we could build the restrictions into the semantics.

One of the goals of S5 is to be a useful tool for analyses of security properties. Since these are security-relevant properties, building them into the semantics gives a richer set of invariants to S5 programs than just having them in desugared expressions alone. Thus, we choose to encapsulate these restrictions in the *okupdate* function of E-SetPropAttr, which checks that for a given property descriptor, the potential change is allowed. Miller describes the invariants in a state diagram, shown in figure 7.[8] The diagram is a more concise description of a long listing of the many combinations of six attributes that need to be considered.

5. Environments and Modelling Eval

The full specification of desugaring is quite long, so we don't attempt to describe it in its entirety. Instead, we focus on how it maps JavaScript identifiers and scopes to S5 environments, and a particularly interesting use case: `eval`.

5.1 Desugaring JavaScript

The end goal of desugaring is to produce an S5 program from JavaScript source; in theory, this is simply a function:

$$\textit{desugar-complete} : e_{JS} \rightarrow e_{S5}$$

For a number of reasons, it is useful to break desugaring up into two pieces:

1. A function from JavaScript programs to open S5 terms containing free variables;
2. an S5 term that contains a hole (a context C), which contains all the bindings needed by the open term.

Then, the composition of the context with the open term produces a (closed) S5 program that corresponds to the initial JavaScript program. So we have:

$$\textit{desugar-complete}(e_{JS}) = C_{env}\,\langle \textit{desugar}[\![e_{JS}]\!]\rangle$$

This strategy has a number of benefits. First, a practical concern: we can change the implementation of the environment term without changing the definition of desugaring. Since different implementations of JavaScript can have different provided built-in functionality, like the implementation on a server versus the implementation in a browser, this strategy allows us to change the environment for these differing scenarios while keeping the core desugaring the same. Second, code that is evaluated with JavaScript's `eval` operator needs to run in slightly altered environments from the environment calling `eval`. This separation allows us to make *desugar* a metafunction that is usable in the semantics because it has no baked-in notion of which environment should be used.

5.2 Compiling JavaScript Identifiers

JavaScript identifiers get their meaning in the specification from *scope objects* [4, Section 8.12], which (roughly) map names to values. At any given point of execution of a JavaScript program, there is a current scope object which can be consulted on each variable dereference to yield the appropriate value, and which can be updated on each variable assignment. Further, the specification allows scope objects to be combined into *scope chains*, so that one scope object can defer lookups to another if it doesn't have a binding for a variable itself.

For the most part, scope objects behave just as lexical scope would. Indeed, their arduous and detailed specification seems unnecessary, given that they seem like a mere implementation decision for lexical scope. However, in a few places, scope objects truly differ from how traditional lexical scope behaves. To accommodate these cases, we desugar JavaScript to S5 programs with explicit scope objects in order to capture all of JavaScript's details.

Global Scope Most of the time, it is impossible for a JavaScript program to obtain a reference to an actual scope object, allowing for a wealth of possible compiling strategies that would emulate scope objects' behavior without implementing them as bona fide JavaScript objects. However, in the case of global scope, JavaScript code can access and alter a scope object with both variable dereferences and with object operations. This means that the identifiers that appear in the source of the JavaScript program might "look" like free variables to a naïve recursive descent, but in fact be defined once they are used:

```
function foo() { xyz = 52; }
this["x" + "yz"] = "assigned via object assignment";
foo();
xyz === 52 && this["xyz"] === 52 // evaluates to true
```

The converse is also true: a variable may appear to be bound, but in fact throw an error when it is accessed at runtime:

[8] Figure taken from `wiki.ecmascript.org/doku.php?id=es3.1:attribute_states`, reproduced from wiki.ecmascript.org under provisions of the Creative Commons License.

```
this.xyz = 52;
xyz; // evaluates to 52
delete this["xyz"];
xyz; // throws ReferenceError exception
```

In order to accommodate these cases, *desugar* converts all JavaScript identifiers into object lookups on the S5 identifier `%context`:

$$\mathit{desugar}[\![\texttt{x}]\!] = \%\texttt{context}[\texttt{"x"}]$$

Then, our desugaring needs to ensure that the correct binding for `%context` is always present.

Implementing Contexts A binding for `%context` needs to hold values corresponding to each identifier in the current context, and correctly delegate to outer contexts for identifiers that are unbound in the current context. The specification describes this chain of delegation by saying that each environment record (context) has a “possibly null reference to an outer Lexical Environment” [4, Section 10.2]. This sounds similar to prototype inheritance; we’ve found that a prototype-based chain of objects is an acceptable fit for such a chain of delegation.[9]

We’ll proceed by example to show the development of contexts. Imagine a simple strategy:

```
function() {
  var x = NaN;
  return function() {
    var y; y = x; return y;
  }
}
```

desugars to

```
func() {
  let (%context = {[proto: %global]
    "x" : [value: undefined, #writable: true] })
  %context["x" = %context["NaN"]];
  func() {
    let (%context = {[proto: %context]
      "y" : [value: undefined, writable: true] })
    %context["y" = %context["x"]];
    %context["y"]
  }
}
```

That is, in this strategy, the **var** gets a property on the `%context` object, and `x` is desugared into a lookup for that property, while `x =` v is desugared into an assignment to that property. This leads to correct behavior for this example, and we can see how the prototype chain allows for delegated access to the `x` variable declared in the outer function. Indeed, the inner function quite literally “closes over its context,” lexically capturing the binding for `%context` used as the **proto** of the inner `%context` object. We also see that the root of the chain is `%global`, so the reference to the identifier `NaN` will find the `"NaN"` property on `%global`.

This desugaring, however, isn’t quite right. Assignments don’t do quite the right thing across scope, and the error behavior with respect to the global object isn’t quite correct either. This example demonstrates the unbound identifier problem:

```
(function() { return x; })();
// yields ReferenceError: x is not defined
```

But our desugaring strategy would return **undefined** here, since that’s the result of the missed object lookup in the desugared expression, `%context["x"]`.

$\boxed{\lambda_{JS}\ \textbf{Contrast}}$ λ_{JS} desugared all JavaScript variables directly into λ_{JS} variables bound to first-class references. This strategy works for almost all programs, but failed to capture this subtlety of the global scope object. □

A different example manifests the assignment problem:

```
function() {
  var x = 0;
  var g = function() { x = x + 1; };
  g(); return x;
}
```

desugars to

```
func() {
  let (%context = {[proto: %global]
    "x" : [value: undefined, #writable: true],
    "g" : [value: undefined, #writable: true] })
  %context["x" = 0];
  %context["g" = func() {
    let (%context = {[proto: %context]})
    %context["x" = %context["x"] + 1]
  }]
  %context["g"](); %context["x"]
}
```

Here, the result of calling the defined function should be `1`: the increment to the outer variable `x` should be applied during the call to the function bound to `g`. But the desugaring will *add* a binding to the inner `%context` upon the assignment to the `"x"` property, rather than changing the outer context. The return at the end will lookup `x` in a context that still has it bound to `0`, resulting in the wrong answer.

Accessors provide an answer to both dilemmas. We can allow context objects to hold a getter/setter pair that stores and updates a value in a separate map. Then, if a property is absent, the right accessor function will be called higher up on the prototype chain, rather than resulting in a property addition. For the global scope issue, we can add an accessor for each unbound identifier that throws the appropriate exception (if the property hasn’t become present yet). The assignment scope problem is solved with:

```
function() {
  var x = 0;
  var g = function() { x = x + 1; };
  g(); return x;
}
```

desugars to

```
func() {
  let (%store = {[extensible: true, proto: null]})
  let (%context = {[proto: %global]
    "x" : [get: func() { %store["x"] },
           set: func(v) { %store["x" = v] }],
    "g" : [get: func() { %store["g"] },
           set: func(v) { %store["g" = v] }] })
  %context["x" = 0];
  %context["g" = func() {
    let (%context = {[proto: %context]})
    %context["x" = %context["x"] + 1]
  }]
  %context["g"](); %context["x"]
}
```

And the global environment has a special kind of context:

[9] We aren’t entirely happy with this decision, but the mapping of scope objects to prototype inheritance is so close that it provides an obvious choice.

Strict Context	Strict String	Direct	Strict Semantics
No	No	No	No
No	No	Yes	No
No	Yes	No	Yes
No	Yes	Yes	Yes
Yes	No	No	**No***
Yes	No	Yes	**Yes***
Yes	Yes	No	Yes
Yes	Yes	Yes	Yes

Figure 9. Deciding when to eval with strict semantics

```
(function() { return z; })();
```

desugars to

```
let (%globalContext = {[proto: null]
  "z": [get: func() {
          if (hasProperty(%global, "z")) { %global["z"] }
          else { throw "ReferenceError" }
        },
        set: func(v) { %global["z" = v] }]})
let (%context = %globalContext)
func() {
  let (%context = {[proto: %globalContext]})
  %context["z"]
}
```

Since only a syntactically appearing identifier can cause such a reference error, this strategy will work if we can always statically determine all the identifiers that *might* be free, which is a simple check of the apparent free variables in the ES5 source. The identifier might be defined by the time the actual access happens, so the check for the property being present on `%global` at that time is important.

These two techniques—using accessors for mutable variables, and building up predefined accessors for possibly-unbound global identifiers—are the necessary pieces for building up environment desugaring. There are a few other details to handle, like the conversion from the properties of an `arguments` object to the environment itself, and exceptional constructs like **typeof**, but overall, we find these patterns to be sufficient for encoding all of JavaScript's scope behavior, including the slightly awkward global scope and all the behavior of `eval`.

5.3 Eval in ES5

ES5 has four variants of `eval`, and there are eight different calling configurations that determine which of the variants to use. They differ based on two JavaScript features: whether the call to `eval` is "direct" and whether the argument is in "strict mode". Figure 8 shows an example of each of the four different types. First we will explain how a program chooses a mode, then explain what each of the modes means.

Mode Determination A use of eval is *direct* rather than *indirect* when the program uses the identifier `eval` in the application rather than some other reference. Compare the call on line 7 in the top two examples to the call on line 8 in the bottom row (and the corresponding binding of `indirect` on line 2). The "direct call" is referred to in the specification as a use of the eval *operator*, as opposed to the eval *function*. This is viscerally distasteful as it breaks normal notions of substitutability for the identifier `eval`, but it is the way of JavaScript.

In the examples given, *strictness* is enforced on the evaluated string by prepending `'use strict';` to the evaluated string. This directive instructs the JavaScript runtime to use "strict" semantics for the body of the `eval`. Prepending this directive is just one way to ensure that the expression evaluates strictly; the surrounding context can also declare itself to be in strict mode, and cause direct calls to `eval` to be strict. Bizarrely, indirect calls will *not* be strict in a strict context, unless the strict directive is included in the string.

```
'use strict';
eval("var x = y; x");
// has strict semantics for the eval
var indirect = eval;
indirect("var x = y; x");
// non-strict semantics, despite 'use strict' above
```

This leads to the truth table in figure 9, whose rightmost column states whether strict semantics should be used for the eval or not. The leftmost column denotes whether block of code in which the `eval` appears has a `'use strict'` directive; the Strict String column represents whether the `eval`ed string has a `'use strict'` directive; and the Direct column represents whether the call is direct. The two perhaps unexpected cases, corresponding to the above example, are highlighted.

The Modes' Semantics First, let us understand the semantics of each of the four modes. The crucial distinction is that the directness of the call determines *which scope eval reads variables from* and the strictness determines *which scope eval creates variables in*. The test cases in figure 8 distinguish these cases.

In a Direct, Non-Strict eval, the current scope is used for both reading and creating new variables. The **var** `x` in the `eval` code creates a new variable on the inner function's scope object, so it can be read afterwards at `x === "inner"`. The value of `y` comes from the same scope, giving both `t` and `x` the value `"inner"`.

In a Direct, Strict eval, the current scope is used for reading variables, but new variable bindings are not reflected on the calling scope. The variable creation **var** `x` does create a new variable, but it isn't visible after the `eval`, so `x` is still `"outer"`, its previous value. The value of `y` still reads from the local context, giving `t` the value `"inner"`.

In an Indirect, Non-Strict eval, the *global* scope is used for reading variables, and new variable bindings are reflected on the *global* scope. We see that now, `y` gets its `"globaly"` value from the global object, and **var** `x` has added a new binding on the global scope object, effectively assigning `global.x` to `"global"`. In addition, this scope is the only one affected; `x` in the function still resolves to `"outer"`, the value from the outer function.

In an Indirect, Strict eval, the global scope is used for reading, and no new variables escape. We see that `t` gets the value `"outer"` because `y` refers to global scope, but no new binding is reflected on `global` from the **var** `x` statement.

Implementing Mode Detection To correctly determine which mode should be used for eval, we first need to desugar expressions so that they have information about their strictness. To do so, we enhance desugaring to include a binding for the special variable #strict:

$$\mathit{desugar}[\![\texttt{'use strict'; e;}]\!] = \textbf{let}\ (\texttt{\#strict = }\textbf{true})\ \mathit{desugar}[\![e]\!]$$

Second, we need to know if a call is direct or not. To do this, we desugar instances of applications of the token `eval` to a call to a function, %maybeDirectEval, that uses the context to determine if this is indeed a direct use:

```
desugar⟦eval(e, ···)⟧ =
  let (%args = makeArgumentsObject⟦e, ···⟧)
    %maybeDirectEval(%this, %args, %context, #strict)
```

Then, %maybeDirectEval can check if the binding on the current context is the global one, and determine whether the call should be considered direct or not. In addition, it receives the #strict flag to determine whether the call should be in strict mode.

Non-Strict

Direct

```
var global = this;
global.y = "globaly";
(function () {
  var x = "outer";
  return function () {
    var y = "inner";
    var t = eval("var x = y; y;");
    global.x === undefined // true
    x === "inner" // true
    t === "inner" // true
  }
})()();
```

Indirect

```
var global = this;
var indirect = eval;
global.y = "globaly";
function () {
  var x = "outer";
  return function () {
    var y = "inner";
    var t = indirect("var x = y; y;");
    global.x === "global" // true
    x === "outer" // true
    t === "global" // true
  }
})()();
```

Strict

Direct

```
var global = this;
global.y = "global";
(function () {
  var x = "outer";
  return function () {
    var y = "inner";
    var t = eval("'use strict'; var x = y; y;");
    global.x === undefined; // true
    x === "outer"; // true
    t === "inner"; // true
  }
})()();
```

Indirect

```
var global = this;
var indirect = eval;
global.y = "global";
(function () {
  var x = "outer";
  return function () {
    var y = "inner";
    var t = indirect("'use strict'; var x = y; y;");
    global.x === undefined; // true
    x === "outer"; // true
    t === "global"; // true
  }
})()();
```

Figure 8. The four kinds of eval in JavaScript

Implementing the Semantics The crucial observation for our implementation is that we have reified all of the information about JavaScript's environments into scope objects that the desugaring and environment can manipulate. If we take a JavaScript string and only apply *desugar* to it, it is simply an open term. To get the different `eval` semantics right, we need to somehow provide bindings to the new expression we get from applying *desugar*. In the style of Racket and E [12, 13], we convert a value from the programming language into an environment for the new expression; we choose objects. That is, the **eval** operator in our semantics takes an additional object parameter that specifies its environment.

Our implementation of **eval** thus relies on creating the correct object to use for an environment. For convenience, we define a function %makeGlobalEnv that creates a new object with all the bindings in the default environment. Then, we implement a function, %configurableEval, that takes a context to evaluate in, and a string to `eval`. It then augments the standard environment with two contexts, %strictCxt and %nonstrictCxt, and desugars the expression in the environment with those bindings:

```
let %configurableEval = func(context, toEval) {
  let (env = %makeGlobalEnv()) {
    env["%strictCxt" = {[proto: context, extensible: true,]}];
    env["%nonstrictCxt" = context];
    desugar⟦ toEval, env ⟧
  }
}
```

The binding for %strictCxt creates a *new* scope object, so any bindings added there won't affect the context passed in. Since the **proto** of the new context points to the provided one, any existing bindings will still be readable. The binding for %nonstrictCxt uses the provided context explicitly, so any additions performed inside the eval will be reflected on that scope as seen by the calling code.

The final bit of subtlety is that *desugar* is strictness-aware. When *desugar* works on a strict-mode program, it wraps it in a let-binding for the strict context. It also does the analogous binding for non-strict mode (we don't show the definition of %defineVar here, which adds bindings to the provided context):

```
desugar⟦'use strict'; var x;⟧ =
let (#strict = true)
  let (%context = %strictCxt) %defineVar(%context, "x")
desugar⟦var x;⟧ =
let (#strict = false)
  let (%context = %nonstrictCxt) %defineVar(%context, "x")
```

Thus, the desugaring provides explicit hooks that the implementation of %configurableEval relies on to hand off the correct environment. With these pieces in place, it's easy to define direct and indirect `eval`:

```
let %directEval = func(context, toEval, strict) {
  %configurableEval(context, toEval, strict) }
let %indirectEval = func(toEval, strict) {
  %configurableEval(%globalContext, toEval, strict) }
```

The %directEval function is called from %maybeDirectEval when a direct call is made, and %indirectEval is called from the built-in `eval` property of the global object. The inner function, %configurableEval, can use its strictness parameter to determine whether or not to create a wrapper environment with new bindings. The key difference is that %directEval passes the current scope, while %indirectEval always hands off the special global context, which we saw in section 5.2. The careful construction of contexts and desugaring allows us to give an account of `eval` and its interactions with strict mode.

6. Related Work

JavaScript Semantics Our closest point of comparison is λ_{JS} [7], which too is an engineered semantics. However, neither major contribution of this paper is covered by λ_{JS}. It does not handle getters and setters because they were not a part of the JavaScript language at the time its semantics was defined. As a result, its object model is also not set up to admit them easily. In addition, λ_{JS} chose to elide `eval`.

There are several other existing semantics for various "cores" of JavaScript, and formalizations of fragments of the language used for analyses and proof of the language. Many of these were developed before the ES5 specification was released [1, 2, 9, 11, 17]. None of those semantics are tested against a real-world test suite.

Considerably less work has been done to formalize ES5 specifically. Taly et al. presented a semantics for a subset of strict mode, dubbed SES$_{light}$ (SES stands for Secure ECMAScript) which they used to build a verification tool [15]. They acknowledge that SES$_{light}$ differs in crucial ways from ES5, and implement a strategy for emulating SES$_{light}$ on top of ES5, acknowledging that there does not exist "any rigorous proof of correctness for [the emulation] yet." Their semantics was chosen for its security properties rather than for detailed conformance to the specification; in contrast, our motivation is conformance. We also note that our semantics for ES5 could provide a basis for such a proof of correctness.

Recently, Hedin and Sabelfeld have presented information-flow control results based on a core of ES5 that lacks a few features, including getters and setters, and the control operators **try-finally** and **break** [8]. They state that "the simplifications and omissions have been chosen to not exclude any information flow challenges...Rather, we envision that extension to the full language is possible without further technical development" [8, page 5]. It is unclear whether this statement is true, since leaving out control-flow operations can have a significant impact on the verification of information flow properties. Since our semantics covers the whole (strict mode) language, tools built atop S5 will not need such (risky) disclaimers.

Objects with Properties and Attributes Maffeis et al. [11] allude to the possibility of extending their semantics (which was pre-ES5) with getters and setters, stating "The semantics can be modularly extended to user-defined getters and setters, which are part of JavaScript 1.5 but not in the ECMA-262 standard". They do not show how to do so in their work.

The technique of passing an object as a self reference to a getter comes from Di Gianantonio et al. [3], who used the technique for method invocation. Their semantics uses a small-step relation to carry the self reference to the method invocation through prototype lookup; we use a big-step relation for prototype lookup, followed by a small step to an application. This big-step semantics for member finding is found in other object calculi that model inheritance, like ClassicJava and Featherweight Java [6, 10]. We are not aware of any work that combines these features into a semantic account of getters and setters, which are a feature of objects in many scripting languages, like Python, Ruby, and Lua.

Implementing Eval Queinnec describes a strategy for macro-expanding S-expressions into Lisp programs in a denotational style [14]. The expansion he describes is quite similar to our E-Eval, in that it expands a richer syntax to a simpler one in place, and then proceeds with evaluation. Queinnec defines a pure evaluator over expanded Scheme programs (`pure-meaning`), a function for environment creation `create-standard-env`, and a function for adding new bindings to the environment, `enrich`. His `pure-meaning` function corresponds to our reduction relation $\rightarrow$, but is denotational, rather than operational. We use bindings of `%context` to hand the shared, and possibly extended, environment to evaled code, rather than immutably copying it.

Wand and Friedman discuss a reflective tower of evaluators as well [16]. They describe their evaluator in terms of an *environment*, a *continuation*, and a *store*. Our environment is much the same, and ends up being shared between the outer code and the evaluated code. Our continuation is stored implicitly in the evaluation context, and our store is, like Wand's, unchanged during the internal evaluation. This work reifies the scope used in the target language into the binding of `%context`, while their work has a special operation that reifies the current environment into a first-class value.

Racket and E have a similar strategy for implementing eval. In both languages, the eval operator requires an explicit parameter containing the bindings it should use for variables. In E, this is called a *scope*, with slots for variables [12]. In Racket, eval expects a *namespace* parameter [13]. Both languages have first-class reifications of their environment, so any program can have complete control over the execution of new code. However, neither has formalized their semantics in the fashion shown in this paper.

Acknowledgments We are deeply grateful to Mark S. Miller for numerous useful discussions. We also thank the anonymous reviewers. We are grateful to Arjun Guha for setting a testable semantics as a standard with λ_{JS}. This work was partially supported by multiple US NSF grants.

References

[1] C. Anderson and P. Giannini. Type checking for JavaScript. *Electronic Notes in Theoretical Computer Science*, 138(2):37 – 58, 2005. Proceedings of the Second Workshop on Object Oriented Developments.

[2] R. Chugh, J. A. Meister, R. Jhala, and S. Lerner. Staged Information Flow for JavaScript. In *Programming Languages Design and Implementation*, 2009.

[3] P. Di Gianantonio, F. Honsell, and L. Liquori. A lambda calculus of objects with self-inflicted extension. *SIGPLAN Notices*, 33(10): 166–178, 1998.

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

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

[6] M. Flatt, S. Krishnamurthi, and M. Felleisen. A programmer's reduction semantics for classes and mixins. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 1998.

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

[8] D. Hedin and A. Sabelfeld. Information-Flow Security Control for JavaScript. In *IEEE Computer Security Foundations Symposium*, 2012.

[9] P. Heidegger and P. Thiemann. Recency types for dynamically-typed, object based languages: Strong updates for JavaScript. In *ACM SIGPLAN International Workshop on Foundations of Object-Oriented Languages*, 2009.

[10] A. Igarashi, B. C. Pierce, and P. Wadler. Featherweight Java: A minimal core calculus for Java and GJ. *ACM Transactions on Programming Languages and Systems*, 23(3), 2001.

[11] S. Maffeis, J. Mitchell, and A. Taly. An Operational Semantics for JavaScript. In *ASIAN Symposium on Programming Languages and Systems*, pages 307–325, 2008.

[12] M. S. Miller. The Kernel-E Language Reference Manual. URL `http://www.erights.org/elang/elangmanual.pdf`.

[13] PLT. The Racket Guide. URL `http://docs.racket-lang.org/guide/eval.html`.

[14] C. Queinnec. Macroexpansion Reflective Tower. In *Reflection*, 1996.

[15] A. Taly, Ú. Erlingsson, J. C. Mitchell, M. S. Miller, and J. Nagra. Automated Analysis of Security-Critical JavaScript APIs. In *IEEE Symposium on Security and Privacy*, 2012.

[16] M. Wand and D. P. Friedman. The Mystery of the Tower Revealed: A Non-Reflective Description of the Reflective Tower. In *LISP and Functional Programming*, 1986.

[17] D. Yu, A. Chander, N. Islam, and I. Serikov. JavaScript Instrumentation for Browser Security. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2007.

A. Appendix: The Rest of S5

We've gradually introduced how we model ES5 objects. This section introduces the rest of the operational semantics of S5. Its syntax appears in figure 11, and the full definition of the semantics is presented incrementally throughout this section.

A.1 The Shape of Evaluation

A term in S5's semantics is a 3-tuple. The *store*, σ, maps locations to values v. Object references r map to object literals θ in a separate, dedicated *object store* Θ. Finally, terms contain expressions e, which describe computations on σ, and Θ.

We split the reduction relation $\rightarrow$ into four categories, corresponding to which portions of the term is manipulated. The four categories are syntactically distinguished by the type of term they work over, and the compatible closure of each is a reduction over $\sigma\Theta; e$. E-Eval uses all portions of the term, and is thus a fifth kind of top-level expression. These four compatible closures are shown, along with E-Eval, in figure 12:

1. E-Compat is the traditional compatible closure over the $\Rightarrow$ relation, which works entirely within evaluation contexts, and does not modify or inspect the variable store (σ), or the object store (Θ).
2. E-EnvStore is the compatible closure of the $\rightarrow^{\sigma}$ relation, which may manipulate the variable store, but *not* the object store. This relation also replaces the expression in the current evaluation context E.
3. E-Objects is the compatible closure of $\rightarrow^{\Theta}$, which manipulates the object store, but not variable store. It also manipulates expressions only within the current evaluation context.
4. E-Control is the compatible closure of $\rightarrow^{\mathsf{e}}$, which handles the abnormal control flow operators like **throw** and **break**. E-Control reductions may manipulate the entire expression, rather than just the entire evaluation context, since they need to perform nonlocal operations, like carrying values from a deeply nested **throw** to its corresponding **catch** block.
5. E-Eval uses the *desugar* metafunction on its first argument (which must be a string), to produce an S5 expression for further evaluation. It also creates a new location in the store, σ, for each property on the object referenced by its second argument. These fresh locations are substituted into the body of the expression, giving the invoker of **eval** complete control of the evaluated expression's lexical bindings.[10]

A.2 Types of Reductions

Object Reductions The rules in figures 5, 6, 13, and 19 show all the reductions for objects, written with the $\rightarrow^{\Theta}$ sub-relation. The motivation and explanation of these reductions is in section 4.

We include here the definition for deleting fields (figure 13), which we elided in the main body of the paper. Configurable properties can be deleted, and the expression results in **true**. If a non-existent property is deleted, the object is unchanged and the expression evaluates to **false**.

We also omitted a discussion of three object attributes earlier: **class**, **primval** and **code**, which are included in figure 11. Objects hold an internal **class** attribute that is a string representation of their prototype. Constructors that wrap primitive values, like `Number` and `String`, store the primitive that they wrap in the **primval** attribute.

$$
\begin{array}{rcl}
r & := & \textit{object references}\\
l & := & \textit{locations}\\
v & := & \textbf{null} \mid \textbf{undefined} \mid \mathsf{str} \mid \mathsf{num} \mid \textbf{true} \mid \textbf{false}\\
 & \mid & r \mid \textbf{func}(x,\cdots)\ \{\ e\ \}\\
 & & \textit{These expressions also appear in } \lambda_{JS}\\
e & := & v \mid x \mid l \mid x\,\texttt{:=}\,e \mid \mathsf{op1}(e) \mid \mathsf{op2}(e,e)\\
 & \mid & e(e,\cdots) \mid e;e \mid \textbf{let}\ (x\ \texttt{=}\ e)\ e\\
 & \mid & \textbf{if}\ (e)\ \{\ e\ \}\ \textbf{else}\ \{\ e\ \}\\
 & \mid & \textbf{label:}\ x\ e \mid \textbf{break}\ x\ e \mid \textbf{err}\ v\\
 & \mid & \textbf{try}\ e\ \textbf{catch}\ x\ e \mid \textbf{try}\ e\ \textbf{finally}\ e \mid \textbf{throw}\ e\\
 & & \\
 & & \textit{From } \textbf{eval} \textit{ on extend or replace constructs from } \lambda_{JS}\textit{:}\\
 & \mid & \textbf{eval}(e,e)\\
 & \mid & \{ae\ \mathsf{str}:pe,\cdots\}\quad \textit{object literals}\\
 & \mid & e[\texttt{<}o\texttt{>}] \mid e[\texttt{<}o\texttt{>}\ \texttt{=}\ e]\quad \textit{object attributes}\\
 & \mid & e[e\texttt{<}a\texttt{>}] \mid e[e\texttt{<}a\texttt{>}\ \texttt{=}e]\quad \textit{property attributes}\\
 & \mid & \textbf{props}(e)\quad \textit{property names}\\
 & \mid & e[e]^{e} \mid e[e\texttt{=}e]^{e} \mid e[\textbf{delete}\ e]\quad \textit{properties}\\
o & := & \textbf{class} \mid \textbf{extensible} \mid \textbf{proto} \mid \textbf{code} \mid \textbf{primval}\\
a & := & \textbf{writable} \mid \textbf{config} \mid \textbf{value} \mid \textbf{enum}\\
ae & := & [\textbf{class}:e,\ \textbf{extensible}:e,\ \textbf{proto}:e,\ \textbf{code}:e,\ \textbf{primval}:e]\\
av & := & [\textbf{class}:v,\ \textbf{extensible}:v,\ \textbf{proto}:v,\ \textbf{code}:v,\ \textbf{primval}:v]\\
pe & := & [\textbf{config}:e,\ \textbf{enum}:e,\ \textbf{value}:e,\ \textbf{writable}:e]\\
 & \mid & [\textbf{config}:e,\ \textbf{enum}:e,\ \textbf{get}:e,\ \textbf{set}:e]\\
pv & := & [\textbf{config}:v,\ \textbf{enum}:v,\ \textbf{value}:v,\ \textbf{writable}:v]\\
 & \mid & [\textbf{config}:v,\ \textbf{enum}:v,\ \textbf{get}:v,\ \textbf{set}:v]\\
p & := & pv \mid [\,]\\
\mathsf{op1} & := & \mathsf{string\text{-}{>}num} \mid \textbf{typeof} \mid \mathsf{log} \mid \mathsf{prim\text{-}{>}bool} \mid \cdots\\
\mathsf{op2} & := & \mathsf{string\text{-}append} \mid + \mid \div \mid > \mid \cdots\\
\theta & := & \{[av]\ \mathsf{str}:pv,\cdots\}\\
\sigma & := & \cdot \mid \sigma, l : v\\
\Theta & := & \cdot \mid \Theta, r : \theta\\
E_{ae} & := & [\textbf{class}:E',\ \textbf{extensible}:e,\ \textbf{proto}:e,\ \textbf{code}:e,\ \textbf{primval}:e]\\
 & \mid & [\textbf{class}:v,\ \textbf{extensible}:E',\ \textbf{proto}:e,\ \textbf{code}:e,\ \textbf{primval}:e]\\
 & \mid & [\textbf{class}:v,\ \textbf{extensible}:v,\ \textbf{proto}:E',\ \textbf{code}:e,\ \textbf{primval}:e]\\
 & \mid & [\textbf{class}:v,\ \textbf{extensible}:v,\ \textbf{proto}:v,\ \textbf{code}:E',\ \textbf{primval}:e]\\
 & \mid & [\textbf{class}:v,\ \textbf{extensible}:v,\ \textbf{proto}:v,\ \textbf{code}:v,\ \textbf{primval}:E']\\
E_{pe} & := & [\textbf{config}:E',\ \textbf{enum}:e,\ \textbf{value}:e,\ \textbf{writable}:e]\\
 & \mid & [\textbf{config}:v,\ \textbf{enum}:E',\ \textbf{value}:e,\ \textbf{writable}:e]\\
 & \mid & [\textbf{config}:v,\ \textbf{enum}:v,\ \textbf{value}:E',\ \textbf{writable}:e]\\
 & \mid & [\textbf{config}:v,\ \textbf{enum}:v,\ \textbf{value}:v,\ \textbf{writable}:E']\\
 & \mid & [\textbf{config}:E',\ \textbf{enum}:e,\ \textbf{get}:e,\ \textbf{set}:e]\\
 & \mid & [\textbf{config}:v,\ \textbf{enum}:E',\ \textbf{get}:e,\ \textbf{set}:e]\\
 & \mid & [\textbf{config}:v,\ \textbf{enum}:v,\ \textbf{get}:E',\ \textbf{set}:e]\\
 & \mid & [\textbf{config}:v,\ \textbf{enum}:v,\ \textbf{get}:v,\ \textbf{set}:E']\\
E' & := & \bullet \mid E'\,\texttt{:=}\,e \mid v\,\texttt{:=}\,E' \mid \mathsf{op1}(E') \mid \mathsf{op2}(E',e) \mid \mathsf{op2}(v,E')\\
 & \mid & E'(e,\cdots) \mid v(v,\cdots E',e,\cdots) \mid E';\ e \mid v;\ E'\\
 & \mid & \textbf{let}\ (x\ =\ E')\ e \mid \textbf{if}\ (E')\ \{\ e\ \}\ \textbf{else}\ \{\ e\ \}\\
 & \mid & \textbf{throw}\ E' \mid \textbf{eval}(E',e) \mid \textbf{eval}(v,E')\\
 & \mid & \{E_{ae}\ \mathsf{str}:pe,\cdots\}\\
 & \mid & \{av\ \mathsf{str}_1 : pv,\cdots \mathsf{str}_x : E_{pe},\ \mathsf{str}_n : pe,\cdots\}\\
 & \mid & E'[\texttt{<}o\texttt{>}] \mid E'[\texttt{<}o\texttt{>}\ \texttt{=}\ e] \mid v[\texttt{<}o\texttt{>}\ \texttt{=}\ E']\\
 & \mid & E'[e\texttt{<}a\texttt{>}] \mid v[E'\texttt{<}a\texttt{>}]\\
 & \mid & E'[e\texttt{<}a\texttt{>}\ \texttt{=}e] \mid v[E'\texttt{<}a\texttt{>}\ \texttt{=}e] \mid v[v\texttt{<}a\texttt{>}\ \texttt{=}E']\\
 & \mid & \textbf{props}(E')\\
 & \mid & E'[e]^{e} \mid v[E']^{e} \mid v[v]^{E'}\\
 & \mid & E'[\textbf{delete}\ e] \mid v[\textbf{delete}\ E']\\
 & \mid & E'[e\texttt{=}e]^{e} \mid v[E'\texttt{=}e]^{e} \mid v[v\texttt{=}E']^{e} \mid v[v\texttt{=}v]^{E'}\\
E & := & E' \mid \textbf{label:}\ x\ E \mid \textbf{break}\ x\ E\\
 & \mid & \textbf{try}\ E\ \textbf{catch}\ e \mid \textbf{try}\ E\ \textbf{finally}\ e\\
F & := & E' \mid \textbf{label:}\ x\ F \mid \textbf{break}\ x\ F\quad \text{(Exception Contexts)}\\
G & := & E' \mid \textbf{try}\ G\ \textbf{catch}\ e\quad \text{(Local Jump Contexts)}
\end{array}
$$

Figure 11. Syntax for S5

[10] Note that we assume the all strings str are valid variables x; this is true in our implementation (though our parser may not recognize some tokens as identifiers). Suitable restrictions can be placed on the allowed strings in eval environment objects (e.g. no spaces) for particular applications; such considerations are routine to build on top of the semantics presented here.

$$\boxed{\Theta; r\,[\mathsf{str}] \Downarrow p}$$

$$\text{P-GetPropFound}\ \frac{\Theta(r) = \{av\ \ \cdots\ \mathsf{str}{:}pv, \cdots\}}{\Theta; r\,[\mathsf{str}] \Downarrow pv} \qquad \text{P-GetPropNotFound}\ \frac{\begin{array}{c}\Theta(r) = \{[\cdots\ \mathbf{proto}{:}\mathbf{null}\ \cdots]\ \mathsf{str'}{:}pv, \cdots\} \\ \mathsf{str} \not\in \mathsf{str'}\ \cdots\end{array}}{\Theta; r\,[\mathsf{str}] \Downarrow [\,]}$$

$$\text{P-GetPropProto}\ \frac{\begin{array}{c}\Theta(r) = \{[\cdots\ \mathbf{proto}{:}r_p\ \ \cdots]\ \mathsf{str'}{:}pv\mathsf{'}, \cdots\} \\ \mathsf{str} \not\in \mathsf{str'}\ \cdots \qquad \Theta; r_p\,[\mathsf{str}] \Downarrow p\end{array}}{\Theta; r\,[\mathsf{str}] \Downarrow p}$$

Figure 10. Attribute lookup through prototypes

$$\boxed{\Theta; e \to^{\Theta} \Theta; e}$$

$$\text{E-DeleteFound}\ \frac{\begin{array}{c}\Theta(r) = \{av\ \ \mathsf{str}_1 : pv_1,\ \ \cdots\ \mathsf{str}{:}\ \ [\cdots\ \mathsf{configurable}{:}\ \mathbf{true}\ \cdots]\,,\ \mathsf{str}_n : pv_n,\ \ \cdots\ \} \\ \Theta' = \Theta[r/\{av\ \ \mathsf{str}_1 : pv_1,\ \ \cdots\ ,\ \mathsf{str}_n : pv_n,\ \ \cdots\ \}]\end{array}}{\Theta; r\,[\mathbf{delete}\ \ \mathsf{str}] \to^{\Theta} \Theta'; \mathbf{true}}$$

$$\text{E-DeleteNotFound}\ \frac{\begin{array}{c}\Theta(r) = \{av\ \ \mathsf{str}_1 : pv_1,\ \ \cdots\ \} \\ \mathsf{str} \not\in \{\mathsf{str}_1, \cdots\}\end{array}}{\Theta; r\,[\mathbf{delete}\ \ \mathsf{str}] \to^{\Theta} \Theta; \mathbf{false}}$$

Figure 13. Deleting fields

$$\boxed{\sigma\Theta; e \to \sigma\Theta; e}$$

$$\begin{array}{rll}
\text{E-Compat} & \sigma\Theta; E\,\langle e\rangle \to \sigma\Theta; E\,\langle e'\rangle & \text{when } e \Rightarrow e' \\
\text{E-EnvStore} & \sigma\Theta; E\,\langle e\rangle \to \sigma'\Theta; E\,\langle e'\rangle & \text{when } \sigma; e \to^{\sigma} \sigma'; e' \\
\text{E-Objects} & \sigma\Theta; E\,\langle e\rangle \to \sigma\Theta'; E\,\langle e'\rangle & \text{when } \Theta; e \to^{\Theta} \Theta'; e' \\
\text{E-Control} & \sigma\Theta; e \to \sigma\Theta; e' & \text{when } e \to^{\mathsf{e}} e' \\
\text{E-Eval} & \sigma\Theta; E\,\langle \mathbf{eval}(\mathsf{str}, r)\rangle \to & \\
 & \quad \sigma'\Theta; E\,\langle \mathit{desugar}[\![\mathsf{str}]\!][\mathsf{str}_1/l_1, \cdots]\rangle & \\
 & \quad \text{where } \Theta(r) = \{av\ \ \mathsf{str}_1{:}\ \ [\mathbf{value}{:}v_1]\cdots\} & \\
 & \quad \text{and } \sigma' = \sigma, l_1 : v_1, \cdots & \\
 & \quad \text{and } l_1, \cdots\ \text{fresh in } \sigma, \mathit{desugar}[\![\mathsf{str}]\!] &
\end{array}$$

Figure 12. Top-level reductions

Functions in ES5 are actually objects that contain an internal **code** attribute. We store S5 **func** values in the **code** attribute of desugared ES5 functions.

$\boxed{\lambda_{JS}\ \textit{Contrast}}$ In λ_{JS}, class, primval, and code were like proto: they were properties on objects with special names. Special care needed to be taken to avoid exposing them to the JavaScript program in property iterations like for-in loops. Much like with **proto**, we put these values into distinguished attributes to remove this overloaded use of properties. □

Local Reductions The local reductions of the ⇒ relation in figure 14 are mostly standard. Primitive operations `op1` and `op2` delegate to a δ function that works over pure values. Figure 15 shows a portion of the δ function, and some of the errors it can throw. The **if** statement and sequencing are routine. We note that **eval** refers to the currently undefined metafunction *desugar*; we return to *desugar* and its interaction with **eval** in section 5.3.

$$\boxed{e \Rightarrow e}$$

$$\begin{array}{rl}
\text{E-Op1} & \mathtt{op1}(v) \Rightarrow \delta_1(\mathsf{op1}, v) \\
\text{E-Op2} & \mathtt{op2}(v_1, v_2) \Rightarrow \delta_2(\mathsf{op2}, v_1, v_2) \\
\text{E-IfTrue} & \mathbf{if}\ (\mathbf{true})\ \{\ e_1\ \}\ \mathbf{else}\ \{\ e_2\ \} \Rightarrow e_1 \\
\text{E-IfFalse} & \mathbf{if}\ (\mathbf{false})\ \{\ e_1\ \}\ \mathbf{else}\ \{\ e_2\ \} \Rightarrow e_2 \\
\text{E-SeqPop} & v; e \Rightarrow e
\end{array}$$

Figure 14. Expression-local reductions

$$\boxed{\delta_1 : \mathsf{op} \times v = v + \mathbf{err}\ \ v}$$

$$\begin{array}{rcl}
\delta(\mathbf{typeof}, r) & = & \texttt{"object"} \\
\delta(\mathbf{typeof}, \mathbf{null}) & = & \texttt{"object"} \\
\delta(\mathbf{typeof}, \mathbf{undefined}) & = & \texttt{"undefined"} \\
\delta(\mathbf{typeof}, \mathsf{num}) & = & \texttt{"number"} \\
\delta(\mathbf{typeof}, \mathsf{str}) & = & \texttt{"string"} \\
 & \cdots &
\end{array}$$

$$\boxed{\delta_2 : \mathsf{op} \times v \times v = v + \mathbf{err}\ \ v}$$

$$\begin{array}{rcl}
\delta(\mathsf{string\text{-}append}, \mathsf{str}_1, \mathsf{str}_2) & = & \mathsf{str1}\ \hat{}\ \mathsf{str2} \\
\delta(\mathsf{string\text{-}append}, \mathsf{num}, \mathsf{str}) & = & \mathbf{err}\ \texttt{"num-string-append"} \\
\delta(\mathsf{string\text{-}append}, r, \mathsf{str}) & = & \mathbf{err}\ \texttt{"obj-string-append"} \\
 & \cdots &
\end{array}$$

Figure 15. A portion of the δ function

$$\boxed{\sigma; e \rightarrow^{\sigma} \sigma; e}$$

$$\begin{array}{rl}
\text{E-Assign} & \sigma; l\texttt{:=}v \rightarrow^{\sigma} \sigma[l/v]; v \\
\text{E-Loc} & \sigma; l \rightarrow^{\sigma} \sigma; \sigma(l) \\
\text{E-Let} & \sigma; \textbf{let}\ \texttt{(}x\ \texttt{=}\ v\texttt{)}e \rightarrow^{\sigma} \sigma, l : v; e[x/l] \\
 & \text{where } l \text{ fresh in } \sigma, e, v \\
\text{E-Apply} & \sigma; \textbf{func}(\mathsf{x}_1, \cdots, \mathsf{x}_n)\ \texttt{\{}\ e\ \texttt{\}}(v_1, \cdots, v_n) \\
 & \rightarrow^{\sigma} \sigma, l_1 : v_1, \cdots, l_n : v_n; e[x_1/l_1, \cdots, x_n/l_n] \\
 & \text{where } l_1, \cdots, l_n \text{ fresh in } \sigma, e, v_1, \cdots, v_n
\end{array}$$

Figure 16. Variable store reductions

Variable Store Reductions The reductions that affect the variable store in the $\rightarrow^{\sigma}$ are shown in figure 16. Assignment replaces the value in the store at the location of l with the new value. Replacement is written $\sigma[l/v]$. In E-Let, we allocate a fresh location, bind it to the value of the binding in σ, and use capture-avoiding substitution (written $e[x/v]$), to replace instances of the variable with the new location. On application with E-App, fresh locations are created for each formal argument with the appropriate values in the store, and the formal arguments are replaced with the appropriate locations via substitution.[11]

$\boxed{\lambda_{JS}\ \textit{\textbf{Contrast}}}$ λ_{JS} had explicity **ref** and **deref** operators. S5 creates references implicitly, which experience shows is a default that leads to more understandable desugared code. A simple check for assignable variables can recover guarantees about immutability that are lost by implicitly creating new locations. □

Control Flow Reductions The rules for control flow are detailed and somewhat subtle, but completely unchanged from [7, figure 10], modulo a new definition of evaluation contexts for the new expressions in S5.

A.3 Reflection: A Design Tradeoff

Our semantics up to this point is adequate for modelling JavaScript's objects but for one omission: reflecting on property names. In ES5, we can request all the property names of an object, and also ask about the presence of properties anywhere on the prototype chain:

```
var o = {x: "something", y: "something else"};
var o2 = Object.create(o);
Object.getOwnPropertyNames(o); // evaluates to ["x", "y"]
Object.getOwnPropertyNames(o2); // evaluates to []
"x" in o2; // true
o2.hasOwnProperty("x"); // false
```

Adding support for reflective operators like `.keys` requires some thought, since it needs to provide a way to compute over a structured form of data: the set of property names. We considered two main approaches: keeping the state of the computation in a term, and returning a structured value of some kind.

Folding within a Term Building up the computation in the term is attractive as it yields a clean semantics at first glance. Figure 18 shows a potential semantics that uses this strategy. The names syntactic form takes a function in its second position, and folds it over all of the string names of the object. From this primitive, we could write computations that build up lists, check for property presence, etc.

If all we needed was an interpeter and a semantics, we would be happy with this semantics. However, we have started building several additional tools on top of S5, so names would need to fit into all those frameworks as well. In particular, we use a CPS transformation to simplify control flow. Performing a CPS transformation on names is anything but straightforward; the easiest solution is to make a new, CPS-names syntactic form with two extra positions that carries success and exception continuations along with it:

$$\mathtt{names(}\texttt{[}\mathsf{str}_1, \cdots\texttt{]}, v_f, v_i, v_{success}, v_{failure}\mathtt{)}$$

Along with this, there would need to be new reduction steps for the CPS-names, so that each invocation of v_f could be passed a suitable continuation. This is hardly an elegant solution, as it requires having a separate CPS-interpreter step for names, so we looked for alternatives. Having a form that returns a value with all of the string names inside it would be ideal, but what kind of value should be returned?

Creating a Value The natural choice for concretely representing the set of property names would be a list or a sequence. The semantics of S5 we've presented so far hasn't included sequences or lists; objects and functions are the only structured values. If we were to add sequences, we would need to add operators over sequences. Further, sequences have no meaning in JavaScript, so our compiling process would need to avoid ever exposing sequences to JavaScript programs, creating a greater divide between the semantics and JavaScript.

A less natural choice would be to allocate a new object, and populate its properties in some well-known way with the strings from the object being reflected on. For example, we could choose the strings `"0"`, `"1"`, etc, and rely on the incrementing and number to string operations in the rest of the semantics to perform lookups into the newly allocated object. This roughly lines up with the JavaScript convention of having "array-like" objects with numeric properties.

For now, we've chosen to allocate a new object, using the **props** rule in figure 19. Given a different compiling strategy that leveraged lists and sequences more, this choice could be different. In the current construction, in exchange for one slightly complicated rule, we avoid growing the semantics with a construct that is dissonant with the essence of JavaScript.

$\boxed{\lambda_{JS}\ \textit{\textbf{Contrast}}}$ λ_{JS} has a stateful δ-function that handles property iteration by keeping track of an iterator on subsequent uses. To keep the type of the δ function simple, S5 makes property reflection a part of the full reduction relation over objects. □

[11] We use a style for allocation similar to Chapter 9 of Felleisen et al. [5].

$$\boxed{e \rightarrow^{\mathsf{e}} e}$$

$$\begin{array}{rl}
\text{E-Throw} & E\left\langle \textbf{throw}\ v \right\rangle \rightarrow^{\mathsf{e}} E\left\langle \textbf{err}\ v \right\rangle \\
\text{E-Catch} & E\left\langle \textbf{try}\ F\left\langle \textbf{err}\ v \right\rangle \textbf{catch}\ e \right\rangle \rightarrow^{\mathsf{e}} E\left\langle e(v) \right\rangle \\
\text{E-Uncaught-Exception} & F\left\langle \textbf{err}\ v \right\rangle \rightarrow^{\mathsf{e}} \textbf{err}\ v \\
\text{E-Finally-Error} & E\left\langle \textbf{try}\ F\left\langle \textbf{err}\ v \right\rangle \textbf{finally}\ e \right\rangle \rightarrow^{\mathsf{e}} E\left\langle e;\ \textbf{err}\ v \right\rangle \\
\text{E-Finally-Break} & E\left\langle \textbf{try}\ G\left\langle \textbf{break}\ x\ v \right\rangle\ \textbf{finally}\ e \right\rangle \rightarrow^{\mathsf{e}} E\left\langle e; \textbf{break}\ x\ v \right\rangle \\
\text{E-Catch-Pop} & E\left\langle \textbf{try}\ v\ \textbf{catch}\ e \right\rangle \rightarrow^{\mathsf{e}} E\left\langle v \right\rangle \\
\text{E-Finally-Pop} & E\left\langle \textbf{try}\ v\ \textbf{finally}\ e \right\rangle \rightarrow^{\mathsf{e}} E\left\langle e;\ v \right\rangle \\
\text{E-Break} & E\left\langle \textbf{label:}\ x\ G\left\langle \textbf{break}\ x\ v \right\rangle \right\rangle \rightarrow^{\mathsf{e}} E\left\langle v \right\rangle \\
\text{E-Break-Pop} & E\left\langle \textbf{label:}\ x_1\ G\left\langle \textbf{break}\ x_2\ v \right\rangle \right\rangle \rightarrow^{\mathsf{e}} E\left\langle \textbf{break}\ x_2\ v \right\rangle \\
 & \quad \text{when } x_1 \neq x_2 \\
\text{E-Label-Pop} & E\left\langle \textbf{label:}\ x\ v \right\rangle \rightarrow^{\mathsf{e}} E\left\langle v \right\rangle \\
\text{E-Break-Break} & E\left\langle \textbf{break}\ x_1\ G\left\langle \textbf{break}\ x_2\ v \right\rangle \right\rangle \rightarrow^{\mathsf{e}} E\left\langle \textbf{break}\ x_2\ v \right\rangle
\end{array}$$

Figure 17. Abnormal control flow reductions

$$\text{E-NamesInit}\ \frac{\Theta(r) = \{av\ \ \mathsf{str}_1 : pv_1, \cdots\}}{\Theta; \mathsf{names(}r\mathsf{,}v_f\mathsf{,}v_i\mathsf{)} \rightarrow \Theta; \mathsf{names([str}_1, \cdots\mathsf{]}\mathsf{,}v_f\mathsf{,}v_i\mathsf{)}} \qquad \text{E-NamesFinish}\ \frac{}{\Theta; \mathsf{names([]}\mathsf{,}v_f\mathsf{,}v_i\mathsf{)} \rightarrow \Theta; v_i}$$

$$\text{E-NamesPop}\ \frac{}{\Theta; \mathsf{names([str}_1\mathsf{,}\ \ \mathsf{str}_2 \cdots\mathsf{]}\mathsf{,}v_f\mathsf{,}v_i\mathsf{)} \rightarrow \Theta; \mathsf{names([str}_2 \cdots\mathsf{]}\mathsf{,}v_f\mathsf{,}v_f\mathsf{(}v_i\mathsf{,}\ \ \mathsf{str}_1\mathsf{))}}$$

Figure 18. A false start at iterating over field names

$$\boxed{\Theta; e \rightarrow^{\Theta} \Theta; e}$$

$$\text{E-GetPropNames}\ \frac{\Theta(r) = \{pv\ \ \mathsf{str}_0 : \mathsf{v}_0, \cdots\ \ \mathsf{str}_n : \mathsf{v}_n\}}{\Theta; \textbf{props}\mathsf{[r]} \rightarrow^{\Theta} \Theta; \left\{ \begin{bmatrix} \textbf{extensible}:\textbf{false}, \\ \textbf{proto}:\textbf{null}, \\ \textbf{code}:\textbf{undefined}, \\ \textbf{primval}:\textbf{undefined} \end{bmatrix} \mathsf{0:} \begin{bmatrix} \textbf{config}:\textbf{false}, \\ \textbf{enum}:\textbf{false}, \\ \textbf{value}:\mathsf{str}_1, \\ \textbf{writable}:\textbf{false} \end{bmatrix} \cdots\ n\mathsf{:} \begin{bmatrix} \textbf{config}:\textbf{false}, \\ \textbf{enum}:\textbf{false}, \\ \textbf{value}:\mathsf{str}_n, \\ \textbf{writable}:\textbf{false} \end{bmatrix} \right\}}$$

Figure 19. Reflecting on property names