The Essence of JavaScript

Arjun Guha, Claudiu Saftoiu, and Shriram Krishnamurthi

Brown University

Abstract. We reduce JavaScript to a core calculus structured as a small-step operational semantics. We present several peculiarities of the language and show that our calculus models them. We explicate the desugaring process that turns JavaScript programs into ones in the core. We demonstrate faithfulness to JavaScript using real-world test suites. Finally, we illustrate utility by defining a security property, implementing it as a type system on the core, and extending it to the full language.

1 The Need for Another JavaScript Semantics

The growing use of JavaScript has created whole new technical and business models of program construction and deployment. JavaScript is a feature-rich language with many quirks, and these quirks are often exploited by security and privacy attacks. This is especially true in cases where JavaScript has a familiar syntax but an unconventional semantics.

Due to its popularity and shortcomings, companies and researchers have tried to tame JavaScript via program analyses [4,9,10,13], sub-language [5,7,17], and more. These works claim but do not demonstrate soundness, partly because we lack a tractable account of the language. The JavaScript standard [6] is capacious and informal, while one major formal semantics [15] is large, not amenable to conventional proof techniques, and inherits the standard's complexities, as we discuss in section 5. In contrast:

– We present a core language, λ_{JS}, that embodies JavaScript's essential features (sans `eval`). λ_{JS} fits on three pages and lends itself well to proof techniques such as subject reduction.
– We show that we can desugar JavaScript into λ_{JS}. In particular, desugaring handles notorious JavaScript features such as `this` and `with`, so λ_{JS} itself remains simple (and thus simplifies proofs that utilize it).
– We mechanize both λ_{JS} and desugaring.
– To show compliance with reality, we successfully test λ_{JS} and desugaring against the actual Mozilla JavaScript test suite.
– Finally, we demonstrate the use of our semantics by building a safe subset of JavaScript. This application highlights how our partitioning of JavaScript into core and syntactic sugar lends structure to proofs.

Our supplemental materials (full desugaring, tools, etc.) are available at

`http://www.cs.brown.edu/research/plt/dl/jssem/v1/`

2 λ_{JS}: A Tractable Semantics for JavaScript

JavaScript is full of surprises. Syntax that may have a conventional interpretation for many readers often has a subtly different semantics in JavaScript. To aid the reader, we introduce λ_{JS} incrementally. We include examples of JavaScript's quirks and show how λ_{JS} faithfully models them.

Figures 1, 2, 4, 8, and 9 specify the syntax and semantics of λ_{JS}. We use a Felleisen-Hieb small-step operational semantics with evaluation contexts [8]. We typeset λ_{JS} code in a sans-serif typeface, and JavaScript in a `fixed-width typeface`.

$$
\begin{aligned}
c &= num \mid str \mid bool \mid \texttt{undefined} \mid \texttt{null} \\
v &= c \mid \texttt{func(}x\cdots\texttt{) \{ return } e \texttt{ \}} \mid \texttt{\{ } str\texttt{:}v\cdots \texttt{ \}} \\
e &= x \mid v \mid \texttt{let (}x \texttt{ = } e\texttt{) } e \mid e\texttt{(}e\cdots\texttt{)} \mid e\texttt{[}e\texttt{]} \mid e\texttt{[}e\texttt{] = } e \mid \texttt{delete } e\texttt{[}e\texttt{]} \\
E &= \bullet \mid \texttt{let (}x \texttt{ = } E\texttt{) } e \mid E\texttt{(}e\cdots\texttt{)} \mid v\texttt{(}v\cdots \; E\texttt{, } e\cdots\texttt{)} \\
&\mid \texttt{\{}str\texttt{: } v\cdots \; str\texttt{:}E\texttt{, } str\texttt{:}e\cdots \texttt{ \}} \mid E\texttt{[}e\texttt{]} \mid v\texttt{[}E\texttt{]} \mid E\texttt{[}e\texttt{] = } e \mid v\texttt{[}E\texttt{] = } e \\
&\mid v\texttt{[}v\texttt{] = } E \mid \texttt{delete } E\texttt{[}e\texttt{]} \mid \texttt{delete } v\texttt{[}E\texttt{]}
\end{aligned}
$$

$$\texttt{let (}x\texttt{ = }v\texttt{) } e \hookrightarrow e[x/v]\cdots \qquad \text{(E-Let)}$$

$$\texttt{(func(}x_1\cdots x_n\texttt{) \{ return } e \texttt{ \})(}v_1\cdots v_n\texttt{)} \hookrightarrow e[x_1/v_1\cdots x_n/v_n] \qquad \text{(E-App)}$$

$$\texttt{\{ }\cdots str\texttt{: } v\cdots \texttt{ \}[}str\texttt{]} \hookrightarrow \mathrm{v} \qquad \text{(E-GetField)}$$

$$\frac{str_x \notin (str_1\cdots str_n)}{\texttt{\{ } str_1\texttt{: } v_1 \;\cdots\; str_n\texttt{: } v_n \texttt{ \} [}str_x\texttt{]} \hookrightarrow \texttt{undefined}} \qquad \text{(E-GetField-NotFound)}$$

$$\begin{array}{c}\texttt{\{ } str_1\texttt{: } v_1\cdots \; str_i\texttt{: } v_i \;\cdots str_n\texttt{: } v_n \texttt{ \} [}str_i\texttt{] = v} \\ \hookrightarrow \texttt{\{ } str_1\texttt{: } v_1\cdots \; str_i\texttt{: } v \;\cdots str_n\texttt{: } v_n \texttt{ \}}\end{array} \qquad \text{(E-UpdateField)}$$

$$\frac{str_x \notin (str_1\cdots)}{\texttt{\{ } str_1\texttt{: } v_1\cdots \texttt{ \} [}str_x\texttt{] = } v_x \hookrightarrow \texttt{\{ } str_x\texttt{: } v_x\texttt{, } str_1\texttt{: } v_1\cdots \texttt{ \}}} \qquad \text{(E-CreateField)}$$

$$\begin{array}{c}\texttt{delete \{ } str_1\texttt{: } v_1\cdots \; str_i\texttt{: } v_x \;\cdots str_x\texttt{: } v_n \texttt{ \} [}str_x\texttt{]} \\ \hookrightarrow \texttt{\{ } str_1\texttt{: } v_1\cdots \; str_i\texttt{: } v \;\cdots str_n\texttt{: } v_n \texttt{ \}}\end{array} \qquad \text{(E-DeleteField)}$$

$$\frac{str_x \notin (str_1\cdots)}{\texttt{delete \{ } str_1\texttt{: } v_1\cdots \texttt{ \} [}str_x\texttt{]} \hookrightarrow \texttt{\{ } str_1\texttt{: } v_1\cdots \texttt{ \}}} \qquad \text{(E-DeleteField-NotFound)}$$

Fig. 1. Functions and Objects

$l = \cdots$	Locations
$v = \cdots \mid l$	Values
$\sigma = (l, v) \cdots$	Stores
$e = \cdots \mid e\ \texttt{=}\ e \mid \textbf{ref}\ e \mid \textbf{deref}\ e$	Expressions
$E = \cdots \mid E\ \texttt{=}\ e \mid v\ \texttt{=}\ E \mid \textbf{ref}\ E \mid \textbf{deref}\ E$	Evaluation Contexts

$$\frac{e_1 \hookrightarrow e_2}{\sigma E\langle e_1\rangle \rightarrow \sigma E\langle e_2\rangle}$$

$$\frac{l \notin dom(\sigma) \qquad \sigma' = \sigma, (l, v)}{\sigma E\langle \textbf{ref}\ v\rangle \rightarrow \sigma' E\langle \mathsf{l}\rangle} \qquad \text{(E-Ref)}$$

$$\sigma E\langle \textbf{deref}\ l\rangle \rightarrow \sigma E\langle \sigma(l)\rangle \qquad \text{(E-Deref)}$$

$$\sigma E\langle l\ \texttt{=}\ v\rangle \rightarrow \sigma[l/v] E\langle l\rangle \qquad \text{(E-SetRef)}$$

We use $\twoheadrightarrow$ to denote the reflexive-transitive closure of $\rightarrow$.

Fig. 2. Mutable References in λ_{JS}

2.1 Functions, Objects and State

We begin with the small subset of λ_{JS} specified in figure 1 that includes just functions and objects. We model operations on objects via functional update. This seemingly trivial fragment already exhibits some of JavaScript's quirks:

– In field lookup, the name of the field need not be specified statically; instead, field names may be computed at runtime (E-GetField):

```
let (obj = { "x" : 500, "y" : 100 })
  let (select = func(name) { return obj[name] })
    select("x") + select("y")
↪* 600
```

– A program that looks up a non-existent field does not result in an error; instead, JavaScript returns the value **undefined** (E-GetField-NotFound):

```
{ "x" : 7 }["y"] ↪ undefined
```

– Field update in JavaScript is conventional (E-UpdateField)—

```
{ "x" : 0 }["x"] = 10 ↪ { "x" : 10 }
```

—but the same syntax also creates new fields (E-CreateField):

```
{ "x" : 0 }["z"] = 20 ↪ {"z" : 20, "x" : 10 }
```

– Finally, JavaScript lets us delete fields from objects:

```
delete { "x": 7, "y": 13}["x"] ↪ { "y": 13 }
```

```
function sum(arr) {
  var r = 0;
  for (var i = 0; i < arr["length"]; i = i + 1) {
    r = r + arr[i] };
  return r };

sum([1,2,3]) ↠ 6
var a = [1,2,3,4];
delete a["3"];
sum(a) ↠ NaN
```

Fig. 3. Array Processing in JavaScript

JavaScript also supports a more conventional dotted-field notation: `obj.x` is valid JavaScript, and is equivalent to `obj["x"]`. To keep λ_{JS} small, we omit the dotted-field notation in favor of the more general computed lookup, and instead explicitly treat dotted fields as syntactic sugar.

Assignment and Imperative Objects. JavaScript has two forms of state: objects are mutable, and variables are assignable. We model both variables and imperative objects with first-class mutable references (figure 2).[1] We desugar JavaScript to explicitly allocate and dereference heap-allocated values in λ_{JS}.

Example: JavaScript Arrays. JavaScript has arrays that developers tend to use in a traditional imperative style. However, JavaScript arrays are really objects, and this can lead to unexpected behavior. Figure 3 shows a small example of a seemingly conventional use of arrays. Deleting the field `a["3"]` (E-DeleteField) does not affect `a["length"]` or shift the array elements. Therefore, in the loop body, `arr["3"]` evaluates to `undefined`, via E-GetField-NotFound. Finally, adding `undefined` to a number yields `NaN`; we discuss other quirks of addition in section 2.6.

2.2 Prototype-Based Objects

JavaScript supports *prototype inheritance* [3]. For example, in the following code, `animal` is the prototype of `dog`:

```
var animal = { "length": 13, "width": 7 };
var dog = { "__proto__": animal, "barks": true };
```

Prototypes affect field lookup:

```
dog["length"] ↠ 13
dog["width"] ↠ 7
```

[1] In the semantics, we use $E\langle e\rangle$ instead of the conventional $E[e]$ to denote a filled evaluation context, to avoid confusion with JavaScript's objects.

$$\frac{str_x \notin (str_1 \cdots str_n) \qquad \texttt{"\_\_proto\_\_"} \notin (str_1 \cdots str_n)}{\texttt{\{}\ str_1 : v_1 , \cdots , str_n : v_n \texttt{\}}\ \texttt{[}str_x\texttt{]} \hookrightarrow \textsf{\textbf{undefined}}} \text{(E-GetField-NotFound)}$$

$$\frac{str_x \notin (str_1 \cdots str_n)}{\texttt{\{}\ str_1 : v_1 \cdots \texttt{"\_\_proto\_\_"}: \textsf{\textbf{null}} \cdots str_n : v_n \texttt{\}}\ \texttt{[}str_x\texttt{]} \hookrightarrow \textsf{\textbf{undefined}}} \text{(E-GetField-Proto-Null)}$$

$$\frac{str_x \notin (str_1 \cdots str_n) \qquad p = \textsf{\textbf{ref}}\ l}{\texttt{\{}\ str_1 : v_1 \cdots \texttt{"\_\_proto\_\_"}: p \cdots str_n : v_n \texttt{\}}\ \texttt{[}str_x\texttt{]} \hookrightarrow (\textsf{\textbf{deref}}\ p)\texttt{[}str_x\texttt{]}} \text{(E-GetField-Proto)}$$

Fig. 4. Prototype-Based Objects

```
var lab = { "__proto__": dog, "length": 2 }
lab["length"] ↠ 2
lab["width"] ↠ 7
lab["barks"] ↠ true
```

Prototype inheritance does not affect field update. The code below creates the field `dog["width"]`, but it does not affect `animal["width"]`, which `dog` had previously inherited:

```
dog["width"] = 19
dog["width"] ↠ 19
animal["width"] ↠ 7
```

However, `lab` now inherits `dog["width"]`:

```
lab["width"] ↠ 19
```

Figure 4 specifies prototype inheritance. The figure modifies E-GetField-NotFound to only apply when the "__proto__" field is missing.

Prototype inheritance is simple, but it is obfuscated by JavaScript's syntax. The examples in this section are not standard JavaScript because the "__proto__" field is not directly accessible by JavaScript programs.[2] In the next section, we unravel and desugar JavaScript's syntax for prototypes.

2.3 Prototypes

JavaScript programmers can indirectly manipulate prototypes using syntax that is reminiscent of class-based languages like Java. In this section, we explain this syntax and its actual semantics. We account for this class-like syntax by desugaring it to manipulate prototypes directly (section 2.2). Therefore, this section does not grow λ_{JS} and only describes desugaring. Figure 5 specifies the portion of desugaring that is relevant for the rest of this section.

[2] Some browsers, such as Firefox, can run these examples.

```
desugar⟦{prop: e···} ⟧ =

ref {
  prop : desugar⟦e⟧ ···,
  "__proto__": (deref Object)["prototype"]
}

desugar⟦function(x···) { stmt··· } ⟧ =

 ref {
  "code": func(this, x···) { return desugar⟦stmt···⟧ },
  "prototype": ref { "__proto___": (deref Object)["prototype"] } }

desugar⟦new e_f(e···)}⟧ =

  let (constr = deref desugar⟦e_f⟧)
    let (obj = ref { "__proto__" : constr["prototype"]})
      constr["code"](obj, desugar⟦e⟧ ···);
      obj

desugar⟦obj[field](e···)⟧ =

  let (obj = desugar⟦obj⟧)
    let (f = (deref obj)[field])
      f["code"](obj, desugar⟦e⟧ ···)

desugar⟦e_f(e···)⟧ =

  let (obj = desugar⟦e_f⟧)
    let (f = deref obj)
      f["code"](window, desugar⟦e⟧ ···)

desugar⟦obj instanceof constr⟧ =

  let (obj = ref (deref desugar⟦obj⟧),
       constr = deref desugar⟦constr⟧)
    done: {
      while (deref obj !== null) {
        if ((deref obj)["__proto__"] === constr["prototype"]) {
          break done true }
        else { obj = (deref obj)["__proto__"] } };
      false }

desugar⟦this⟧ = this (an ordinary identifier, bound by functions)
desugar⟦e.x⟧ = desugar⟦e⟧["x"]
```

Fig. 5. Desugaring JavaScript's Object Syntax

The `this` Keyword. JavaScript does not have conventional methods. Function-valued fields are informally called "methods", and provide an interpretation for a `this` keyword, but both are quite different from those of, say, Java.

For example, in figure 6, when `obj.setX(10)` is applied, `this` is bound to `obj` in the body of the function. In the same figure however, although `f` is bound

```
var obj = {
  "x" : 0,
  "setX": function(val) { this.x = val } };

// window is the name of the global object in Web browsers
window.x ↠ undefined
obj.setX(10);
obj.x ↠ 10
var f = obj.setX;
f(90);
obj.x ↠ 10 // obj.x was not updated
window.x ↠ 90 // window.x was created
```

Fig. 6. Implicit `this` Parameter

to `obj.setX`, `f(90)` does not behave like a traditional method call. In fact, the function is applied with `this` bound to the *global object* [6, Section 10.1.5].

In general, `this` is an implicit parameter to all JavaScript functions. Its value is determined by the syntactic shape of function applications. Thus, when we desugar functions to λ_{JS}, we make `this` an explicit argument. Moreover, we desugar function calls to explicitly supply a value for `this`.

Functions as Objects. In JavaScript, functions are objects with fields:

```
f = function(x) { return x + 1 }
f.y = 90
f(f.y) ↠ 91
```

We desugar JavaScript's `function` to objects in λ_{JS} with a distinguished `code` field that refers to the actual function. Therefore, we also desugar application to lookup the `code` field.

We could design λ_{JS} so that functions truly are objects, making this bit of desugaring unnecessary. In our experience, JavaScript functions are rarely used as objects. Therefore, our design lets us reason about simple functions when possible, and functions as objects only when necessary.

In addition to the `code` field, which we add by desugaring, and any other fields that may have been created by the programmer, all functions also have a distinguished field called `prototype`. As figure 5 shows, the `prototype` field is a reference to an object that eventually leads to the prototype of `Object`. Unlike the `__proto__` field, `prototype` is accessible and can be updated by programmers. The combination of its mutability and its use in `instanceof` leads to unpredictable behavior, as we show below.

Constructors and Prototypes. JavaScript does not have explicit constructors, but it does have a `new` keyword that invokes a function with `this` bound to a new object. For example, the following code—

```
function Point(x, y) {
  this.x = x;
  this.y = y }

pt = new Point(50, 100)
```

—applies the function `Point` and returns the value of `this`. `Point` explicitly sets `this.x` and `this.y`. Moreover, `new Point` implicitly sets `this.__proto__` to `Point.prototype`. We can now observe prototype inheritance:

```
Point.prototype.getX = function() { return this.x }
pt.getX() ↠ pt.__proto__.getX() ↠ 50
```

In standard JavaScript, because the `__proto__` field is not exposed, the only way to set up a prototype hierarchy is to update the `prototype` field of functions that are used as constructors.

The `instanceof` Operator. JavaScript's `instanceof` operator has an unconventional semantics that reflects the peculiar notion of constructors that we have already discussed. In most languages, a programmer might expect that if `x` is bound to the value created by `new Constr(···)`, then `x instanceof Constr` is true. In JavaScript, however, this invariant does not apply.

For example, in figure 7, `animalThing` dispatches on the type of its argument using `instanceof`. However, after we set `Cat.prototype = Dog.prototype`, the type structure seems to break down. The resulting behavior might appear unintuitive in JavaScript, but it is straightforward when we desugar `instanceof`

```
function Dog() { this.barks = "woof" };
function Cat() { this.purrs = "meow" };
dog = new Dog();
cat = new Cat();
dog.barks; ↠ "woof"
cat.purrs; ↠ "meow"

function animalThing(obj) {
  if (obj instanceof Cat) { return obj.purrs }
  else if (obj instanceof Dog) { return obj.barks }
  else { return "unknown animal" } };

animalThing(dog); ↠ "woof"
animalThing(cat); ↠ "meow"
animalThing(4234); ↠ "unknown animal"

Cat.prototype = Dog.prototype;
animalThing(cat); ↠ "unknown animal"
animalThing(dog) ↠ undefined // dog.purrs (E-GetField-NotFound)
```

Fig. 7. Using `instanceof`

into λ_{JS}. In essence, `cat instanceof Cat` is `cat.__proto__ === Cat.prototype`.[3] In the figure, before `Cat.prototype = Dog.prototype` is evaluated, the following are true:

```
cat.__proto__ === Cat.prototype
dog.__proto__ === Dog.prototype
Cat.prototype !== Dog.prototype
```

However, after we update `Cat.prototype`, we have:

```
cat.__proto__ === the previous value of Cat.prototype
dog.__proto__ === Dog.prototype
Cat.prototype === Dog.prototype
```

Hence, `cat instanceof Cat` becomes `false`. Furthermore, since `animalThing` first tests for `Cat`, the test `dog instanceof Cat` succeeds.

2.4 Statements and Control Operators

JavaScript has a plethora of control statements. Many map directly to λ_{JS}'s control operators (figure 8), while the rest are easily desugared.

For example, consider JavaScript's `return` and `break` statements. A `break` l statement transfers control to the local label l. A `return` e statement transfers control to the end of the local function and produces the value of e as the result. Instead of two control operators that are almost identical, λ_{JS} has a single **break** expression that produces a value.

Concretely, we elaborate JavaScript's functions to begin with a label ret:

$$\begin{gathered} \mathit{desugar}[\![\texttt{function(}x\cdots\texttt{) \{ } \mathit{stmt}\cdots \texttt{ \}}]\!] = \\ \textsf{\textbf{func}}(\mathit{this}\ x\cdots)\ \texttt{\{ } \textsf{\textbf{return}}\ \mathit{ret}\texttt{: \{ } \mathit{desugar}[\![\mathit{stmt}\cdots]\!] \texttt{ \} \}} \end{gathered}$$

Thus, `return` statements are desugared to **break** ret:

$$\mathit{desugar}[\![\texttt{return}\ e]\!] = \textsf{\textbf{break}}\ \textsf{ret}\ \mathit{desugar}[\![e]\!]$$

while `break` statements are desugared to produce **undefined**:

$$\mathit{desugar}[\![\texttt{break}\ \mathit{label}]\!] = \textsf{\textbf{break}}\ \mathit{label}\ \textsf{\textbf{undefined}}$$

2.5 Static Scope in JavaScript

The JavaScript standard specifies identifier lookup in an unconventional manner. It uses neither substitution nor environments, but *scope objects* [6, Section 10.1.4]. A scope object is akin to an activation record, but is a conventional JavaScript object. The fields of this object are interpreted as variable bindings.

In addition, a scope object has a distinguished parent-field that references another scope object. (The global scope object's parent-field is `null`.) This linked list of scope objects is called a *scope chain*. The value of an identifier `x` is the

[3] The `===` operator is the physical equality operator, akin to `eq?` in Scheme.

$$
\begin{array}{rl}
\mathit{label} = & \text{(Labels)} \\
e = & \cdots \mid \textsf{\textbf{if}}\ \texttt{(}e\texttt{)}\ \texttt{\{}\ e\ \texttt{\}}\ \textsf{\textbf{else}}\ \texttt{\{}\ e\ \texttt{\}} \mid e\texttt{;}e \mid \textsf{\textbf{while}}\texttt{(}e\texttt{)}\ \texttt{\{}\ e\ \texttt{\}} \mid \mathit{label}\texttt{:\{}\ e\ \texttt{\}} \\
& \mid \textsf{\textbf{break}}\ \mathit{label}\ e \mid \textsf{\textbf{try}}\ \texttt{\{}\ e\ \texttt{\}}\ \textsf{\textbf{catch}}\ \texttt{(}x\texttt{)}\ \texttt{\{}\ e\ \texttt{\}} \mid \textsf{\textbf{try}}\ \texttt{\{}\ e\ \texttt{\}}\ \textsf{\textbf{finally}}\ \texttt{\{}\ e\ \texttt{\}} \\
& \mid \textsf{\textbf{err}}\ v \mid \textsf{\textbf{throw}}\ e \\
E = & \cdots \mid \textsf{\textbf{if}}\ \texttt{(}E\texttt{)}\ \texttt{\{}\ e\ \texttt{\}}\ \textsf{\textbf{else}}\ \texttt{\{}\ e\ \texttt{\}} \mid E\texttt{;}e \mid \mathit{label}\texttt{:\{}\ E\ \texttt{\}} \\
& \mid \textsf{\textbf{try}}\ \texttt{\{}\ E\ \texttt{\}}\ \textsf{\textbf{catch}}\ \texttt{(}x\texttt{)}\ \texttt{\{}\ e\ \texttt{\}} \mid \textsf{\textbf{try}}\ \texttt{\{}\ E\ \texttt{\}}\ \textsf{\textbf{finally}}\ \texttt{\{}\ e\ \texttt{\}} \mid \textsf{\textbf{throw}}\ E \\
E' = & \bullet \mid \textsf{\textbf{let}}\ \texttt{(}x\ \texttt{=}\ v \cdots\ x\ \texttt{=}\ E'\texttt{,}\ x\ \texttt{=}\ e \cdots\texttt{)}\ e \mid E'\texttt{(}e\cdots\texttt{)} \mid v\texttt{(}v\cdots\ E'\texttt{,}\ e\cdots\texttt{)} \\
& \mid \textsf{\textbf{if}}\ \texttt{(}E'\texttt{)}\ \texttt{\{}\ e\ \texttt{\}}\ \textsf{\textbf{else}}\ \texttt{\{}\ e\ \texttt{\}} \mid \texttt{\{}\ \mathit{str}\texttt{:}\ v\cdots\ \mathit{str}\texttt{:}\ E'\texttt{,}\ \mathit{str}\texttt{:}\ e\cdots\ \texttt{\}} \\
& \mid E'\texttt{[}e\texttt{]} \mid v\texttt{[}E'\texttt{]} \mid E'\texttt{[}e\texttt{]}\ \texttt{=}\ e \mid v\texttt{[}E'\texttt{]}\ \texttt{=}\ e \mid v\texttt{[}v\texttt{]}\ \texttt{=}\ E' \mid E'\ \texttt{=}\ e \mid v\ \texttt{=}\ E' \\
& \mid \textsf{\textbf{delete}}\ E'\texttt{[}e\texttt{]} \mid \textsf{\textbf{delete}}\ v\texttt{[}E'\texttt{]} \mid \textsf{\textbf{ref}}\ E' \mid \textsf{\textbf{deref}}\ E' \mid E'\texttt{;}\ e \mid \textsf{\textbf{throw}}\ E' \\
F = & E' \mid \mathit{label}\texttt{:\{}\ F\ \texttt{\}} \quad \text{(Exception Contexts)} \\
G = & E' \mid \textsf{\textbf{try}}\ \texttt{\{}\ G\ \texttt{\}}\ \textsf{\textbf{catch}}\ \texttt{(}x\texttt{)}\ \texttt{\{}\ e\ \texttt{\}} \quad \text{(Local Jump Contexts)}
\end{array}
$$

$$\textsf{\textbf{if}}\ \texttt{(}\textsf{\textbf{true}}\texttt{)}\ \texttt{\{}\ e_1\ \texttt{\}}\ \textsf{\textbf{else}}\ \texttt{\{}\ e_2\ \texttt{\}} \hookrightarrow e_1 \qquad \text{(E-IfTrue)}$$

$$\textsf{\textbf{if}}\ \texttt{(}\textsf{\textbf{false}}\texttt{)}\ \texttt{\{}\ e_1\ \texttt{\}}\ \textsf{\textbf{else}}\ \texttt{\{}\ e_2\ \texttt{\}} \hookrightarrow e_2 \qquad \text{(E-IfFalse)}$$

$$v\texttt{;}e \hookrightarrow e \qquad \text{(E-Begin-Discard)}$$

$$\textsf{\textbf{while}}\texttt{(}e_1\texttt{)}\ \texttt{\{}\ e_2\ \texttt{\}} \hookrightarrow \textsf{\textbf{if}}\ \texttt{(}e_1\texttt{)}\ \texttt{\{}\ e_2\texttt{;}\ \textsf{\textbf{while}}\texttt{(}e_1\texttt{)}\ \texttt{\{}\ e_2\ \texttt{\}}\ \texttt{\}}\ \textsf{\textbf{else}}\ \texttt{\{}\ \textsf{\textbf{undefined}}\ \texttt{\}} \qquad \text{(E-While)}$$

$$\textsf{\textbf{throw}}\ v \hookrightarrow \textsf{\textbf{err}}\ v \qquad \text{(E-Throw)}$$

$$\textsf{\textbf{try}}\ \texttt{\{}\ F\langle\textsf{\textbf{err}}\ v\rangle\ \texttt{\}}\ \textsf{\textbf{catch}}\ \texttt{(}x\texttt{)}\ \texttt{\{}\ e\ \texttt{\}} \hookrightarrow e[x/v] \qquad \text{(E-Catch)}$$

$$\sigma F\langle\textsf{\textbf{err}}\ v\rangle \to \sigma\textsf{\textbf{err}}\ v \qquad \text{(E-Uncaught-Exception)}$$

$$\textsf{\textbf{try}}\ \texttt{\{}\ F\langle\textsf{\textbf{err}}\ v\rangle\ \texttt{\}}\ \textsf{\textbf{finally}}\ \texttt{\{}\ e\ \texttt{\}} \hookrightarrow e\texttt{;}\ \textsf{\textbf{err}}\ v \qquad \text{(E-Finally-Error)}$$

$$\textsf{\textbf{try}}\ \texttt{\{}\ G\langle\textsf{\textbf{break}}\ \mathit{label}\ v\rangle\ \texttt{\}}\ \textsf{\textbf{finally}}\ \texttt{\{}\ e\ \texttt{\}} \hookrightarrow e\texttt{;}\ \textsf{\textbf{break}}\ \mathit{label}\ v \qquad \text{(E-Finally-Break)}$$

$$\textsf{\textbf{try}}\ \texttt{\{}\ v\ \texttt{\}}\ \textsf{\textbf{finally}}\ \texttt{\{}e\ \texttt{\}} \hookrightarrow e\texttt{;}\ v \qquad \text{(E-Finally-Pop)}$$

$$\textsf{label}\texttt{:\{}\ G\langle\textsf{\textbf{break}}\ \textsf{label}\ v\rangle\ \texttt{\}} \hookrightarrow v \qquad \text{(E-Break)}$$

$$\frac{\mathit{label}_1 \neq \mathit{label}_2}{\mathit{label}_1\texttt{:\{}\ G\langle\textsf{\textbf{break}}\ \mathit{label}_2\ v\rangle\ \texttt{\}} \hookrightarrow \textsf{\textbf{break}}\ v} \qquad \text{(E-Break-Pop)}$$

$$\mathit{label}\texttt{:}\ \texttt{\{}v\texttt{\}} \hookrightarrow v \qquad \text{(E-Label-Pop)}$$

Fig. 8. Control operators for λ_{JS}

value of the first `x`-field in the *current scope chain*. When a new variable `y` is defined, the field `y` is added to the scope object at the head of the scope chain.

Since scope objects are ordinary JavaScript objects, JavaScript's `with` statement lets us add arbitrary objects to the scope chain. Given the features discussed below, which include `with`, it is not clear whether JavaScript is lexically scoped. In this section, we describe how JavaScript's scope-manipulation statements are desugared into λ_{JS}, which is obviously lexically scoped.

Local Variables. In JavaScript, functions close over their current scope chain (intuitively, their static environment). Applying a closure sets the current scope chain to be that in the closure. In addition, an empty scope object is added to the head of the scope chain. The function's arguments and local variables (introduced using `var`) are properties of this scope object.

Local variables are automatically *lifted* to the top of the function. As a result, in a fragment such as this—

```
function foo() {
  if (true) { var x = 10 }
  return x }

foo() ↠ 10
```

—the `return` statement has access to the variable that appears to be defined inside a branch of the `if`. This can result in somewhat unintuitive answers:

```
function bar(x) {
  return function() {
    var x = x;
    return x }}

bar(200)() ↠ undefined
```

Above, the programmer might expect the `x` on the right-hand side of `var x = x` to reference the argument `x`. However, due to lifting, all bound occurrences of `x` in the nested function reference the local variable `x`. Hence, `var x = x` reads and writes back the initial value of `x`. The initial value of local variables is `undefined`.

We can easily give a lexical account of this behavior. A local variable declaration, `var x = e`, is desugared to an assignment, `x = e`. Furthermore, we add a let-binding at the top of the enclosing function:

let (x = **ref** **undefined**) ⋯

Global Variables. Global variables are subtle. Global variables are properties of the global scope object (`window`), which has a field that references itself:

```
window.window === window ↠ true
```

Therefore, a program can obtain a reference to the global scope object by simply referencing `window`.[4]

[4] In addition, `this` is bound to `window` in function applications (figure 5).

As a consequence, globals seem to break lexical scope, since we can observe that they are properties of `window`:

```
var x = 0;
window.x = 50;
x ↠ 50
x = 100;
window.x ↠ 100
```

However, `window` is the only scope object that is directly accessible to JavaScript programs [6, Section 10.1.6]. We maintain lexical scope by abandoning global variables. That is, we simply desugar the obtuse code above to the following:

```
window.x = 0;
window.x = 50;
window.x ↠ 50
window.x = 100;
window.x ↠ 100
```

Although global variables observably manipulate `window`, local variables are still lexically scoped. We can thus reason about local variables using substitution, α-renaming, and other standard techniques.

With Statements. The `with` statement is a widely-acknowledged JavaScript wart. A `with` statement adds an arbitrary object to the front of the scope chain:

```
function(x, obj) {
  with(obj) {
    x = 50; // if obj.x exists, then obj.x = 50, else x = 50
    return y } } // similarly, return either obj.y, or window.y
```

We can desugar `with` by turning the comments above into code:

```
function(x, obj) {
  if (obj.hasOwnProperty("x")) { obj.x = 50 }
  else { x = 50 }
  if ("y" in obj) { return obj.y }
  else { return window.y } }
```

Nested `with`s require a little more care, but can be dealt with in the same manner. However, desugaring `with` is non-compositional. We will return to this point in section 4.3.

What are Scope Objects? Various authors (including ourselves) have developed JavaScript tools that work with a subset of JavaScript that is intuitively lexically scoped (e.g., [2,5,7,10,11,17]). We show how JavaScript can be desugared into lexically scoped λ_{JS}, validating these assumptions. As a result, we no longer need scope objects in the specification; they may instead be viewed as an implementation strategy.[5]

[5] Scope objects are especially well suited for implementing `with`. Our desugaring strategy for `with` increases code-size linearly in the number of nested `with`s, which scope-objects avoid.

$$\begin{aligned} e &= \cdots \mid op_n(e_1 \cdots e_n) \\ E &= \cdots \mid op_n(v \cdots E\ e \cdots) \\ E' &= \cdots \mid op_n(v \cdots E' e \cdots) \\ \delta_n &: op_n \times v_1 \cdots v_n \to c + err \end{aligned}$$

$$op_n(v_1 \cdots v_n) \hookrightarrow \delta_n(op_n, v_1 \cdots v_n) \qquad \text{(E-PRIM)}$$

Fig. 9. Primitive Operators

2.6 Type Conversions and Primitive Operators

JavaScript is not a pure object language. We can observe the difference between primitive numbers and number objects:

```
x = 10;
y = new Number(7)
typeof x ↠ "number"
typeof y ↠ "object"
```

Moreover, JavaScript's operators include implicit type conversions between primitives and corresponding objects:

```
x + y ↠ 17
```

We can redefine these type conversions without changing objects' values:

```
Number.prototype.valueOf = function() { return 0 }
x + y ↠ 10
y.toString() ↠ "7"
```

Both + and * perform implicit coercions, and + also concatenates strings:

```
x + y.toString() ↠ "107" // 10 converted to the string "10"
x * y.toString() ↠ 70 // "7" converted to the number 7
```

This suggests that JavaScript's operators are complicated. Indeed, the standard specifies `x + y` with a 15-step algorithm [6, Section 11.6.1] that refers to three pages of metafunctions. Buried in these details are four primitive operators: primitive addition, string concatenation, and number-to-string and string-to-number type coercions.

These four primitives are essential and intuitive. We therefore model them with a conventional δ function (figure 9). The remaining details of operators are type-tests and method invocations; as the examples above suggest, JavaScript internally performs operations such as `y.valueOf()` and `typeof x`. In λ_{JS} we make these type-tests and method calls explicit.

This paper does not enumerate all the primitives that λ_{JS} needs. Instead, the type of δ constrains their behavior significantly, which often lets us reason without a specific δ function. (For instance, due to the type of δ, we know that primitives cannot manipulate the heap.)

$$\begin{tikzcd}
\text{JavaScript Program} \arrow[r, "desugar"] \arrow[d, "\text{Real Implementations}"'] & \lambda_{JS}\text{ Program} \arrow[d, "\lambda_{JS}\text{ interpreter}"] \\
\texttt{stdout} \arrow[r, leftrightarrow, "\texttt{diff}"'] & \texttt{stdout}
\end{tikzcd}$$

Fig. 10. Testing Strategy for λ_{JS}

3 Soundness and Adequacy of $\boldsymbol{\lambda_{JS}}$

Soundness. We mechanize λ_{JS} with PLT Redex [8]. The process of mechanizing helped us find errors in our semantics, particularly in the interactions of control operators (figure 8). We use our mechanized semantics to test [14] λ_{JS} for safety.

Property 1 (Progress). *If σe is a closed, well-formed configuration, then either:*

- $e \in v$,
- $e =$ **err** v, *for some* v, *or*
- $\sigma e \to \sigma' e'$, *where* $\sigma' e'$ *is a closed, well-formed configuration.*

This property requires additional evaluation rules for runtime type errors, and definitions of well-formedness. We elide them from the paper, as they are conventional. The supplemental material contains these details.

Adequacy. λ_{JS} is a semantics for the core of JavaScript. We have described how it models many aspects of the language's semantics, warts and all. Ultimately, however, a small core language has limited value to those who want to reason about programs written in full JavaScript.

Given our method of handling JavaScript via desugaring, we are obliged to show that desugaring and the semantics enjoy two properties. First, we must show that all JavaScript programs can be desugared to λ_{JS}.

Claim 1 (Desugaring is Total). *For all JavaScript programs* e, $desugar[\![e]\!]$ *is defined.*

Second, we must demonstrate that our semantics corresponds to what JavaScript implementations actually do.

Claim 2 (Desugar Commutes with Eval) *For all JavaScript programs* e, $desugar[\![eval_{JavaScript}(e)]\!] = eval_{\lambda_{JS}}(desugar[\![e]\!])$.

We could try to prove these claims, but that just begs the question: What is $eval_{JavaScript}$? A direct semantics would require evidence of its own adequacy.

Syntactic Form	Occurrences (approx.)
`with` blocks	15
`var` statements	500
`try` blocks	20
`functions`	200
`if` and `switch` statements	90
`typeof` and `instanceof`	35
`new` expressions	50
`Math` library functions	15

Fig. 11. Test Suite Coverage

In practice, JavaScript is truly defined by its major implementations. Open-source Web browsers are accompanied by extensive JavaScript test suites. These test suites help the tacit standardization of JavaScript across major implementations.[6] We use these test suites to *test* our semantics.

Figure 10 outlines our testing strategy. We first define an interpreter for λ_{JS}. This is a straightforward exercise; the interpreter is a mere 100 LOC, and easy to inspect since it is based directly on the semantics.[7] Then, for any JavaScript program, we should be able to run it both directly and in our semantics. For direct execution we employ three JavaScript implementations: SpiderMonkey (used by Firefox), V8 (used by Chrome), and Rhino (an implementation in Java). We desugar the same program into λ_{JS} and run the result through our interpreter. We then check whether our λ_{JS} interpreter produces the same output as each JavaScript implementation.

Our tests cases are a significant portion of the Mozilla JavaScript test suite. We omit the following tests:

– Those that target Firefox-specific JavaScript extensions.
– Those that use `eval`.
– Those that target library details, such as regular expressions.

The remaining tests are about 5,000 LOC unmodified.

Our λ_{JS} interpreter produces exactly the same output as Rhino, V8, and SpiderMonkey on the entire test suite. Figure 11 indicates that these tests employ many interesting syntactic forms, including statements like `with` and `switch` that are considered complicated. We make the following observations:

– No prior semantics for JavaScript accounts for all these forms (e.g., Maffeis et al. [15] do not model `switch`).
– We account for much of JavaScript by desugaring. Therefore, these tests validate both our core semantics and our desugaring strategy.
– These tests give us confidence that our implemented tools are correct.

[6] For example, the Firefox JavaScript test suite is also found in the Safari source.

[7] PLT Redex can evaluate expressions in a mechanized semantics. However, our tests are too large for Redex's evaluator.

4 Example: Language-Based Sandboxing

Web platforms often combine programs from several different sources on the same page. For instance, on a portal page like iGoogle, a user can combine a weather widget with a stock ticker widget; on Facebook, users can run applications. Unfortunately, this means programs from different authors can in principle examine data from one another, which creates the possibility that a malicious application may steal data or create other harm. To prevent both accidents and malice, sites must somehow sandbox widgets.

To this end, platform developers have defined safe sub-languages (often called "safe subsets") of JavaScript like ADsafe [5], Caja [17], and Facebook JavaScript (FBJS) [7]. These are designed as sub-languages—rather than as whole new languages with, perhaps, security types—to target developers who already know how to write JavaScript Web applications. These sub-languages disallow blatantly dangerous features such as `eval`. However, they also try to establish more subtle security properties using syntactic restrictions, as well as runtime checks that they insert into untrusted code. Naturally, this raises the question whether these sub-languages function as advertised.

Let us consider the following property, which is inspired by FBJS and Caja: we wish to prevent code in the sandbox from communicating with a server. For instance, we intend to block the XMLHttpRequest object:

```
var x = new window.XMLHttpRequest()
x.open("GET", "/get_confidential", false)
x.send("");
var result = x.responseText
```

For simplicity, we construct a sub-language that only disallows access to XMLHttpRequest. A complete solution would use our techniques to block other communication mechanisms, such as `document.write` and `Element.innerHTML`.

We begin with short, type-based proofs that exploit the compactness of λ_{JS}. We then use our tools to migrate from λ_{JS} to JavaScript.

4.1 Isolating JavaScript

We must precisely state "disallow access to XMLHttpRequest". In JavaScript, `window.XMLHttpRequest` references the XMLHttpRequest constructor, where `window` names the global object. We make two assumptions:

- In λ_{JS}, we allocate the global object at location `0`. This is a convenient convention that is easily ensured by desugaring.
- The XMLHttpRequest constructor is only accessible as a property of the global object. This assumption is valid as long as we do not use untrusted libraries (or can analyze their code).

Given these two assumptions, we can formally state "disallow access to XMLHttpRequest" as a property of λ_{JS} programs:

```
lookup = func(obj, field) {
  return if (field === "XMLHttpRequest") { undefined }
         else { (deref obj)[field] }
}
```

Fig. 12. Safe Wrapper for λ_{JS}

Definition 1 (Safety). *e is safe if* $e \neq E\langle\langle$ **deref** *(***ref** *0)*$\rangle$ *["XMLHttpRequest"]*$\rangle$.

Note that in the definition above, the active expression is (**deref** (**ref** 0)), and the evaluation context is $E\langle\bullet$["XMLHttpRequest"]$\rangle$.

Intuitively, ensuring safety appears to be easy. Given an untrusted λ_{JS} program, we can elaborate property accesses, e_1[e_2], to *lookup*(e_1,e_2), where *lookup* is defined in Figure 12.

This technique[8] has two problems. First, this elaboration does not allow access to the "XMLHttpRequest" property of *any* object. Second, although *lookup* may appear "obviously correct", the actual wrapping in Caja, FBJS, and other sub-languages occurs in JavaScript, not in a core calculus like λ_{JS}. Hence, *lookup* does not directly correspond to any JavaScript function. We could write a JavaScript function that resembles *lookup*, but it would be wrought with various implicit type conversions and method calls (section 2.6) that could break its intended behavior. Thus, we start with safety for λ_{JS} before tackling JavaScript's details.

4.2 Types for Securing λ_{JS}

Our goal is to determine whether a λ_{JS} program is safe (definition 1). We wish to do this without making unnecessary assumptions. In particular, we do not assume that *lookup* (figure 12) is itself safe.

We begin by statically disallowing *all* field accesses. The trivial type system in Figure 13 achieves this, since it excludes a typing rule for e_1[e_2]. This type system does not catch conventional type errors. Instead, it has a single type, **JS**, of statically safe JavaScript expressions (definition 1). The following theorem is evidently true:

Theorem 1. *For all λ_{JS} expressions e, if $\cdot \vdash e : T$ and $e \twoheadrightarrow e'$ then e' is safe.*

We need to extend our type system to account for *lookup*, taking care not to violate theorem 1. Note that *lookup* is currently untypable, since field access is untypable. However, the conditional in *lookup* seems to ensure safety; our goal is to prove that it does. Our revised type system is shown in figure 14. The new type, **NotXHR**, is for expressions that provably do not evaluate to the string "XMLHttpRequest". Since primitives like string concatenation yield values of type **JS** (T-PRIM in figure 13), programs cannot manufacture unsafe strings

[8] Maffeis et al.'s blacklisting [16], based on techniques used in FBJS, has this form.

$T = \mathbf{JS}$

$$\Gamma \vdash \mathit{string} : \mathbf{JS} \qquad \text{(T-String)}$$

$$\frac{\Gamma(x) = T}{\Gamma \vdash x : T} \qquad \text{(T-Id)}$$

$$\frac{\Gamma, x_1 : \mathbf{JS}, \cdots, x_n : \mathbf{JS} \vdash e : \mathbf{JS}}{\Gamma \vdash \textsf{\textbf{func}}\ (x_1 \cdots x_n)\ \{\ \textsf{\textbf{return}}\ \mathsf{e}\ \} : \mathbf{JS}} \qquad \text{(T-Fun)}$$

$$\frac{\Gamma \vdash e_1 : \mathbf{JS} \qquad \cdots \qquad \Gamma \vdash e_n : \mathbf{JS}}{\Gamma \vdash \delta_n(op_n, e_1 \cdots e_n) : \mathbf{JS}} \qquad \text{(T-Prim)}$$

The type judgments for remaining forms are similar to T-Prim and T-Fun: namely, $\Gamma \vdash e : \mathbf{JS}$ if all subexpressions of e have type **JS**. However, e_1[e_2] is *not typable.*

Fig. 13. Type System that Disallows Field Lookup

with type **NotXHR**. (Of course, trusted primitives could yield values of type **NotXHR**.)

Note this important peculiarity: *These new typing rules are purpose-built for lookup.* There are other ways to establish safe access to fields. However, since we will rewrite all expressions e_1[e_2] to *lookup*(e_1,e_2), our type system need only account for the syntactic structure of *lookup*.

Our revised type system admits *lookup*, but we must prove theorem 1. It is sufficient to prove the following lemmas:[9]

Lemma 1 (Safety). *If* $\cdot \vdash e : \boldsymbol{JS}$*, then* $e \neq E\langle$v["XMLHttpRequest"]$\rangle$*, for any value* v*.*

The proof of this lemma is by induction on typing derivations, given the typing rules in figure 13 and figure 14. This lemma also holds for the typing rules in figure 15, which we introduce below.

Lemma 2 (Subject Reduction). *If* $\cdot \vdash e : \boldsymbol{JS}$*, and* $e \rightarrow e'$*, then* $\cdot \vdash e' : \boldsymbol{JS}$*.*

Proof Technique. The typing rules for *lookup* (figure 14) require a technique introduced in *occurrence typing* for Typed Scheme [18].

Although *lookup* is typable, subject reduction requires all expressions in this reduction sequence to be typable:

```
lookup(window, "XMLHttpRequest")
→ if ("XMLHttpRequest" === "XMLHttpRequest") { undefined }
  else { (deref window)["XMLHttpRequest"] }
→ if (true) { undefined }
  else { (deref window)["XMLHttpRequest"] }
→ undefined
```

[9] Additional proof details are in the supplemental material.

$T = \cdots \mid \mathbf{NotXHR}$

$$\mathbf{NotXHR} <: \mathbf{JS} \qquad \text{(Sub-Safe)}$$

$$\frac{\Gamma \vdash e : S \qquad S <: T}{\Gamma \vdash e : T} \qquad \text{(T-Sub)}$$

$$\frac{v \neq \texttt{"XMLHttpRequest"}}{\Gamma \vdash v : \mathbf{NotXHR}} \qquad \text{(T-SafeValue)}$$

$$\frac{\Gamma \vdash e_1 : \mathbf{JS} \qquad \Gamma \vdash e_2 : \mathbf{NotXHR}}{\Gamma \vdash e_1\texttt{[}e_2\texttt{]} : \mathbf{JS}} \qquad \text{(T-GetField)}$$

$$\frac{x \in dom(\Gamma) \qquad \Gamma \vdash e_2 : \mathbf{JS} \qquad \Gamma[x : \mathbf{NotXHR}] \vdash e_3 : \mathbf{JS}}{\Gamma \vdash \textbf{if}\ \texttt{(}x\ \texttt{==="XMLHttpRequest")}\ \texttt{\{}\ e_2\ \texttt{\}}\ \textbf{else}\ \texttt{\{}\ e_3\ \texttt{\}} : \mathbf{JS}} \qquad \text{(T-IfSafe)}$$

Fig. 14. Type System for Blocking Access to XMLHttpRequest

$$\frac{\Gamma \vdash e_2 : \mathbf{JS}}{\Gamma \vdash \textbf{if}\ \texttt{("XMLHttpRequest"==="XMLHttpRequest")}\ \texttt{\{}\ e_2\ \texttt{\}}\ \textbf{else}\ \texttt{\{}\ e_3\ \texttt{\}} : \mathbf{JS}} \qquad \text{(T-IfTrue-XHR)}$$

$$\frac{\Gamma \vdash e_2 : \mathbf{JS}}{\Gamma \vdash \textbf{if}\ \texttt{(}\textbf{true}\texttt{)}\ \texttt{\{}\ e_2\ \texttt{\}}\ \textbf{else}\ \texttt{\{}\ e_3\texttt{\}} : \mathbf{JS}} \qquad \text{(T-IfTrue)}$$

Fig. 15. Auxiliary Typing Rules for Blocking Access to XMLHttpRequest

The intermediate expressions above are not typable, although they are intuitively safe. We can make them typable by extending our type system with the typing rules in figure 15, which let us prove subject reduction.

However, we have to ensure that our new typing rules do not violate safety (lemma 1). Intuitively, lemma 1 still holds, since our newly-typable expressions are not of the form v`["XMLHttpRequest"]`.

Our type system may appear ad hoc, but it simply reflects the nature of JavaScript security solutions. Note that our type system is merely a means to an end: the main result is the conclusion of theorem 1, which is a property of the runtime semantics.

4.3 Scaling to JavaScript

Since we can easily implement a checker for our type system,[10] we might claim we have a result for JavaScript as follows: desugar JavaScript into λ_{JS} and

[10] The supplemental material includes a 150-line implementation.

type-check the resultant λ_{JS} code. This strategy is, however, unsatisfying because seemingly harmless changes to a typable JavaScript program may result in a program that fails to type-check, due to the effects of desugaring. This would make the language appear whimsical to the widget developer.

Instead, our goal is to define a safe sub-language (just as, say, Caja and FBJS do). This safe sub-language would provide syntactic safety criteria, such as:

– The JavaScript expression e_1 `+` e_2 is safe when its subexpressions are safe.
– e_1`[`e_2`]`, when rewritten to *lookup*`(`e_1`,` e_2`)`, is safe, but fails if e_2 evaluates to `"XMLHttpRequest"`.

Our plan is as follows. We focus on the *structure* of the desugaring rules and show that a particular kind of compositionality in these rules suffices for showing safety. We illustrate this process by extending the λ_{JS} result to include JavaScript's addition (which, as we explained in section 2.6, is non-trivial). We then generalize this process to the rest of the language.

Safety for Addition. By theorem 1, it is sufficient to determine whether $\Gamma \vdash \mathit{desugar}[\![e_1\texttt{+}e_2]\!] : \mathbf{JS}$. Proving this, however, would benefit from some constraints on e_1 and e_2. Consider the following proposition:

Proposition 1. *If* $\Gamma \vdash \mathit{desugar}[\![e_1]\!] : \boldsymbol{JS}$ *and* $\Gamma \vdash \mathit{desugar}[\![e_2]\!] : \boldsymbol{JS}$, *then* $\Gamma \vdash \mathit{desugar}[\![e_1 \texttt{ + } e_2]\!] : \boldsymbol{JS}$.

By lemma 1, this proposition entails that if e_1 and e_2 are safe, then $e_1 + e_2$ is safe. But is the proposition true? $\mathit{desugar}[\![e_1\texttt{+}e_2]\!]$ produces an unwieldy λ_{JS} expression with explicit type-conversions and method calls. Still, a quick inspection of our implementation shows that:

$$\mathit{desugar}[\![e_1 \texttt{ + } e_2]\!] \texttt{ = } \textbf{let} \texttt{ (x = } \mathit{desugar}[\![e_1]\!]\texttt{) } \textbf{let} \texttt{ (y = } \mathit{desugar}[\![e_2]\!]\texttt{) } \cdots$$

$\mathit{desugar}[\![e_1 \texttt{ + } e_2]\!]$ simply recurs on its subexpressions and does not examine the result of $\mathit{desugar}[\![e_1]\!]$ and $\mathit{desugar}[\![e_2]\!]$. Moreover, the elided body does not contain additional occurrences of $\mathit{desugar}[\![e_1]\!]$ and $\mathit{desugar}[\![e_2]\!]$. Thus, we can write the right-hand side as a two-holed *program context*:[11]

$$\mathit{desugar}[\![e_1 \texttt{ + } e_2]\!] \texttt{ = } C_+\langle \mathit{desugar}[\![e_1]\!], \mathit{desugar}[\![e_2]\!]\rangle$$
$$C_+ \texttt{ = } \textbf{let} \texttt{ (x = } \bullet_1\texttt{) } \textbf{let} \texttt{ (y = } \bullet_2\texttt{) } \cdots$$

Therefore, desugaring e_1 `+` e_2 is *compositional.*

A simple replacement lemma [20] holds for our type system:

Lemma 3 (Replacement). *If:*

i. *$\mathcal{D}$ is a deduction concluding $\Gamma \vdash C[e_1, e_2] : \boldsymbol{JS}$,*
ii. *Subdeductions $\mathcal{D}_1, \mathcal{D}_2$ prove that $\Gamma_1 \vdash e_1 : \boldsymbol{JS}$ and $\Gamma_2 \vdash e_2 : \boldsymbol{JS}$ respectively,*
iii. *$\mathcal{D}_1$ occurs in $\mathcal{D}$, at the position corresponding to $\bullet_1$, and $\mathcal{D}_2$ at the position corresponding to $\bullet_2$, and*

[11] Due to lack of space, we do not formally define program contexts for λ_{JS} in this paper, but evaluation contexts offer a strong hint.

iv. $\Gamma_1 \vdash e_1' : \boldsymbol{JS}$ *and* $\Gamma_2 \vdash e_2' : \boldsymbol{JS}$*,*

then $\Gamma \vdash C\langle e_1', e_2' \rangle : \boldsymbol{JS}$*.*

Replacement, along with weakening of environments, gives us our final lemma:

Lemma 4. *If:*

– $x : \boldsymbol{JS}, y : \boldsymbol{JS} \vdash C_+[x, y] : \boldsymbol{JS}$*, and*
– $\Gamma \vdash desugar[\![e_1]\!] : \boldsymbol{JS}$ *and* $\Gamma \vdash desugar[\![e_2]\!] : \boldsymbol{JS}$*,*

then $\Gamma \vdash C_+\langle desugar[\![e_1]\!], desugar[\![e_2]\!]\rangle : \boldsymbol{JS}$*.*

The conclusion of lemma 4 is the conclusion of proposition 1. The second hypothesis of lemma 4 is the only hypothesis of proposition 1. Therefore, to prove proposition 1, we simply need to prove $x : \mathbf{JS}, y : \mathbf{JS} \vdash C_+\langle x, y\rangle : \mathbf{JS}$.

We establish this using our tools. We assume `x` and `y` are safe (i.e., have type **JS**), and desugar and type-check the expression `x + y`. Because this succeeds, the machinery above—in particular, the replacement lemma—tells us that we may admit `+` into our safe sub-language.

A Safe Sub-Language. The proofs of lemma 3 and 4 do not rely on the definition of C_+. For each construct, we must thus ensure that the desugaring rule can be written as a program context, which we easily verify by inspection. We find this true for all syntactic forms other than `with`, which we omit from our safe sub-language (as do other sub-language such as Caja and FBJS). If `with` were considered important, we could extend our machinery to determine what circumstances, or with what wrapping, it too could be considered safe.

Having checked the structure of the desugaring rules, we must still establish that their expansion does no harm. We mechanically populate a type environment with placeholder variables, create expressions of each kind, and type-check. All forms pass type-checking, except for the following:

– `x[y]` and `x.XMLHttpRequest` do not type—happily, as they are unsafe! This is acceptable because these unsafe forms will be wrapped in *lookup*.
– However, `x[y]++`, `x[y]--`, `++x[y]`, and `--x[y]` also fail to type due to the structure of code they generate on desugaring. Yet, we believe these forms are safe; we could account for them with additional typing rules, as employed below for *lookup*.

Safety for *lookup*. As section 4.2 explained, we designed our type system to account for *lookup* (figure 12). However, *lookup* is in λ_{JS}, whereas we need a wrapper in JavaScript. A direct translation of *lookup* into JavaScript yields:

```
lookupJS = function(obj, field) {
 if (field === "XMLHttpRequest") { return undefined }
 else { return obj[field] } }
```

Since *lookupJS* is a closed expression that is inserted as-is into untrusted scripts, we can desugar and type-check it in isolation. Doing so, however, reveals a surprise: $desugar[\![lookupJS]\!]$ does not type-check.

When we examine the generated λ_{JS} code, we see that `obj[field]` is desugared into an expression that explicitly converts `field` to a string. (Recall that field names are strings.) If, however, `field` is itself an object, this conversion includes the method call `field.toString()`. Working backward, we see that the following exploit would succeed:

```
lookupJS(window, { toString: function() { return "XMLHttpRequest" } })
```

where the second argument to *lookupJS* (i.e., the expression in the field position) is a literal object that has a single method, `toString`, which returns `"XMLHttpRequest"`. Thus, not only does *lookupJS* not type, it truly is unsafe!

Our type system successfully caught a bug in our JavaScript implementation of *lookup*. The fix is simple: ensure that `field` is a primitive string:

```
safeLookup = function(obj, field) {
  if (field === "XMLHttpRequest") { return undefined }
  else if (typeof field === "string") { return obj[field] }
  else { return undefined } }
```

This code truly is safe, though to prove it we need to extend our type system. We design the extension by studying the result of desugaring *safeLookup*.[12]

We have noted that desugaring evinces the unsafe method call. However, `toString` is called only if `field` is not a primitive. This conditional is inserted *by desugaring*:

```
if (typeof field === "location") { ... field.toString() ... }
else { field }
```

Thus, the second **if** in *safeLookup* desugars to:

```
if (typeof field === "string") {
  obj[if (typeof field === "location") { ... field.toString() ... }
      else { field }] }
```

To now reach `field.toString()`, both conditions must hold. Since this cannot happen, the unsafe code block is unreachable.

Recall, however, that we designed our type system for λ_{JS} around the syntactic structure of the lookup guard. With this more complex guard, we must extend our type system to employ if-splitting—which we already used in section 4.2—a second time. As long as our extension does not violate safety (lemma 1) and subject reduction (lemma 2), the arguments in this section still hold.

4.4 Perspective

In the preceding sections, we rigorously developed a safe sub-language of Java Script that disallows access to XMLHttpRequest. In addition, we outlined a proof of correctness for the runtime "wrapper". To enhance isolation, we have to disallow access to a few other properties, such as `document.write` and

[12] Desugaring produces 200 LOC of pretty-printed λ_{JS}. We omit this code from the paper, but it is available online.

`Element.innerHTML`. Straightforward variants of the statements and proofs in this section could verify such systems. We believe our approach can scale to tackle more sophisticated security properties as well.

Nevertheless, our primary goal in this section is not to define a safe sublanguage of JavaScript, but rather to showcase our semantics and tools:

– λ_{JS} is small. It is much smaller than other definitions and semantics for JavaScript. Therefore, our proofs are tractable.
– λ_{JS} is adequate and tested. This gives us confidence that our arguments are applicable to real-world JavaScript.
– λ_{JS} is conventional, so we are free to use standard type-soundness techniques [20]. In contrast, working with JavaScript's scope objects would be onerous. This section is littered with statements of the form $\Gamma \vdash e : \mathbf{JS}$. Heap-allocated scope objects would preclude the straightforward use of Γ, thus complicating the proof effort (and perhaps requiring new techniques).
– Finally, *desugar* is compositional. Although we developed a type system for λ_{JS}, we were able to apply our results to most of JavaScript by exploiting the compositionality of *desugar*.

5 Related Work

JavaScript Semantics. JavaScript is specified in 200 pages of prose and pseudocode [6]. This specification is barely amenable to informal study, let alone proofs. Maffeis, Mitchell, and Taly [15] present a 30-page operational semantics, based directly on the JavaScript specification. Their semantics covers most of JavaScript directly, but does omit a few syntactic forms.

Our approach is drastically different. λ_{JS} is a semantics for the core of Java Script, though we desugar the rest of JavaScript into λ_{JS}. In section 3, we present evidence that our strategy is correct. λ_{JS} and desugaring together are much smaller and simpler than the semantics presented by Maffeis, et al. Yet, we cover all of JavaScript (other than `eval`) and account for a substantial portion of the standard libraries as well (available in the supplementary material).

Maffeis, et al. demonstrate adequacy by following the standard, though they discuss various differences between the standard and implementations. In section 3, we demonstrate adequacy by running 3rd-party JavaScript tests in λ_{JS} and comparing results with mainstream JavaScript implementations.

A technical advantage of our semantics is that it is conventional. For example, we use substitution instead of scope objects (section 2.5). Therefore, we can use conventional techniques, such as subject reduction, to reason in λ_{JS}. It is unclear how to build type systems for a semantics that uses scope objects.

David Herman [12] defines a CEKS machine for a small portion of JavaScript. This machine is also based on the standard and inherits some of its complexities, such as implicit type conversions.

CoreScript [21] models an imperative subset of JavaScript, along with portions of the DOM, but omits essentials such as functions and objects. Moreover, their big-step semantics is not easily amenable to typical type safety proofs.

Object Calculi. λ_{JS} is an untyped, object-based language with prototype inheritance. However, λ_{JS} does not have methods as defined in object calculi. Without methods, most object calculi cease to be interesting. However, we do desugar JavaScript's method invocation syntax to self-application in λ_{JS} [1, Chapter 18].

λ_{JS} and JavaScript do not support cloning, which is a crucial element of other prototype-based languages, such as Self [19]. JavaScript does support Self's prototype inheritance, but the surface syntax of JavaScript does not permit direct access to an object's prototype (section 2.3). Without cloning, and without direct access to the prototype, JavaScript programmers cannot use techniques such as dynamic inheritance and mode-switching [1].

Types for JavaScript. There are various proposed type systems for JavaScript that are accompanied by semantics. However, these semantics are only defined for small subsets of JavaScript, not the language in its entirety. For example, Anderson et al. [2] develop a type system and a type inference algorithm for JS_0, a subset that excludes prototypes and first-class functions. Heidegger and Thiemann's recency types [11] admit prototypes and first-class functions, but omit assignment. In contrast, we account for all of JavaScript (excluding `eval`).

Acknowledgments

We thank Matthias Felleisen, Mike Samuel, Peter Thiemann, and the anonymous reviewers for their careful comments on an earlier draft. We thank Sergio Maffeis, Leo Meyerovich, and John Mitchell for enlightening discussions. This work was partially supported by the NSF.

References

1. Abadi, M., Cardelli, L.: A Theory of Objects. Springer, Heidelberg (1996)
2. Anderson, C., Giannini, P., Drossopoulou, S.: Towards type inference for JavaScript. In: Black, A.P. (ed.) ECOOP 2005. LNCS, vol. 3586, pp. 428–452. Springer, Heidelberg (2005)
3. Borning, A.: Classes versus prototypes in object-oriented languages. In: ACM Fall Joint Computer Conference (1986)
4. Chugh, R., Meister, J.A., Jhala, R., Lerner, S.: Staged information flow for JavaScript. In: ACM SIGPLAN Conference on Programming Language Design and Implementation (2009)
5. Crockford, D.: ADSafe, `http://www.adsafe.org`
6. ECMAScript language specification (1999)
7. Facebook. FBJS, `http://wiki.developers.facebook.com/index.php/FBJS`
8. Felleisen, M., Findler, R.B., Flatt, M.: Semantics Engineering with PLT Redex. MIT Press, Cambridge (2009)
9. Guarnieri, S., Livshits, B.: GateKeeper: Mostly static enforcement of security and reliability policies for JavaScript code. In: USENIX Security Symposium (2009)
10. Guha, A., Krishnamurthi, S., Jim, T.: Static analysis for Ajax intrusion detection. In: International World Wide Web Conference (2009)

11. Heidegger, P., Thiemann, P.: Recency types for dynamically-typed, object-based languages: Strong updates for JavaScript. In: ACM SIGPLAN International Workshop on Foundations of Object-Oriented Languages (2009)
12. Herman, D.: ClassicJavaScript, `http://www.ccs.neu.edu/home/dherman/javascript/`
13. Jensen, S.H., Møller, A., Thiemann, P.: Type analysis for JavaScript. In: International Static Analysis Symposium (2009)
14. Klein, C., Finder, R.B.: Randomized testing in PLT Redex. In: ACM SIGPLAN Workshop on Scheme and Functional Programming (2009)
15. Maffeis, S., Mitchell, J.C., Taly, A.: An operational semantics for JavaScript. In: Ramalingam, G. (ed.) APLAS 2008. LNCS, vol. 5356, pp. 307–325. Springer, Heidelberg (2008)
16. Maffeis, S., Mitchell, J.C., Taly, A.: Isolating JavaScript with filters, rewriting, and wrappers. In: Backes, M., Ning, P. (eds.) ESORICS 2009. LNCS, vol. 5789, pp. 505–522. Springer, Heidelberg (2009)
17. Miller, M.S., Samuel, M., Laurie, B., Awad, I., Stay, M.: Caja: Safe active content in sanitized JavaScript. Technical report, Google Inc. (2008), `http://google-caja.googlecode.com/files/caja-spec-2008-06-07.pdf`
18. Tobin-Hochstadt, S., Felleisen, M.: The design and implementation of Typed Scheme. In: ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (2008)
19. Ungar, D., Smith, R.B.: SELF: The power of simplicity. In: ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages & Applications (1987)
20. Wright, A., Felleisen, M.: A syntactic approach to type soundness. Information and Computation 115(1) (1994)
21. Yu, D., Chander, A., Islam, N., Serikov, I.: Javascript instrumentation for browser security. In: ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (2007)