Towards a Program Logic for JavaScript

Philippa Gardner
Imperial College London
pg@doc.ic.ac.uk

Sergio Maffeis
Imperial College London
maffeis@doc.ic.ac.uk

Gareth Smith
Imperial College London
gds@doc.ic.ac.uk

Abstract

JavaScript has become the most widely used language for client-side web programming. The dynamic nature of JavaScript makes understanding its code notoriously difficult, leading to buggy programs and a lack of adequate static-analysis tools. We believe that logical reasoning has much to offer JavaScript: a simple description of program behaviour, a clear understanding of module boundaries, and the ability to verify security contracts.

We introduce a program logic for reasoning about a broad subset of JavaScript, including challenging features such as prototype inheritance and `with`. We adapt ideas from separation logic to provide tractable reasoning about JavaScript code: reasoning about easy programs is easy; reasoning about hard programs is possible. We prove a strong soundness result. All libraries written in our subset and proved correct with respect to their specifications will be well-behaved, even when called by arbitrary JavaScript code.

Categories and Subject Descriptors F.3.1 [*Theory of Computation*]: Logics and Meanings of Programs—Specifying and Verifying and Reasoning about Programs

General Terms Reliability, Theory, Verification

Keywords JavaScript, Separation Logic, Web

1. Introduction

JavaScript has become the de-facto language for client-side web programming. Ajax web applications, used in e.g. Google Docs, are based on a combination of JavaScript and server-side programming. JavaScript has become an international standard called ECMAScript [14]. Adobe Flash, used in e.g. YouTube, features ActionScript, a programming language based on ECMAScript. Even web applications written in e.g. Java, F♯ or purpose-designed languages such as Flapjax or Milescript are either compiled to JavaScript, or they lack browser integration or cross-platform compatibility. JavaScript is currently the assembly language of the Web, and this seems unlikely to change.

JavaScript was initially created for small web-programming tasks, which benefited from the flexibility of the language and tight browser integration. Nowadays, the modern demands placed on JavaScript are huge. Although this flexibility and browser integration are still key advantages, the inherent dynamic nature of the language makes current web code based on ECMAScript 3 notoriously difficult to write and use [12, 18, 28]. The expectation is that ECMAScript 5 and future standards will improve the situation. However, although the main browsers now support ECMAScript 5, the majority of code being written today is in ECMAScript 3. Even if there is a wide acceptance of ECMAScript 5, which is certainly not clear from the current blogs, it is inevitable that ECMAScript 5 libraries will have to interface properly with ECMAScript 3 code. We therefore believe that there is a growing need for general-purpose, expressive analysis tools for both ECMAScript 3 and 5, which provide simple, correct descriptions of program behaviour and a clear understanding of module boundaries.

We introduce a program logic for reasoning about ECMAScript 3. While it is tempting to ignore the ‘ugly’ parts of the language, and reason only about ‘well-written’ code, in practice JavaScript programs have to interface with arbitrary web code. We therefore use a model of the language that does not shun the most challenging JavaScript features. For example, the behaviour of prototype inheritance, and the interplay between scoping rules and the `with` statement, is complex. This means that our basic reasoning rules must also be complex. We overcome this complexity by establishing natural layers of abstraction on top of our basic reasoning. With principled code, we can stay within these layers of abstraction and the reasoning is straightforward. With arbitrary code, we must break open the appropriate abstraction layers until we can re-establish the invariants of the abstraction. In this way, we are able to provide clean specifications of a wide variety of JavaScript programs.

Our reasoning is based on separation logic. Separation logic has proven to be invaluable for reasoning about programs which directly manipulate the heap, such as C and Java programs [2, 3, 8, 20, 36]. A key characteristic of JavaScript is that the entire state of the language resides in the object heap, in a structure that imperfectly emulates the variable store of many other programming languages. It is therefore natural to investigate the use of separation logic to verify JavaScript programs. In fact, we had to fundamentally adapt separation logic, both to present an accurate account of JavaScript’s emulated variable store (see Section 2: Motivating Examples) and also to establish soundness. For soundness, it is usual to require that all the program commands are ‘local’, according to a definition first given in [20]. Many JavaScript statements are not local by this definition: for example, even a simple variable read is non-local because its result may depend on the *absence* of certain fields in the emulated variable store. We instead prove soundness using a concept of *weak locality*, recently introduced by Smith [30].

In this paper, we reason about a substantial subset of JavaScript, including prototype inheritance, `with`, simple functions (no recursive or higher-order functions) and simple `eval`. Our treatment of functions and `eval` is precisely enough to expose fully the complexity of the emulated variable store. Building on the work of Charlton and Reus [6, 29], we will extend our reasoning to higher-order functions and complex `eval` in future. We prove soundness of our reasoning with respect to a faithful subset of the formal

POPL’12, January 25–27, 2012, Philadelphia, PA, USA.

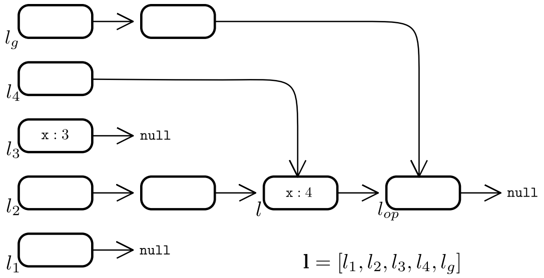

Figure 1. A JavaScript emulated variable store

operational semantics of Maffeis *et al.* [18]. Our soundness result has powerful implications. Library code written in our subset and proved correct with respect to their specifications will be well-behaved, even when called by arbitrary JavaScript code. Our result is constructed in such a way that it will be simple to extend to reasoning about higher-order functions and complex `eval` in due course.

2. Motivating Examples

As convincingly argued in [12, 18, 27, 28], there are many reasons why the behaviour of JavaScript programs is complex. For a start, JavaScript is a dynamically typed, prototype-oriented language, which does not have a standard notion of variable store. Instead, JavaScript variables are best regarded as being stored in the heap, in a structure which imperfectly emulates the variable store. This structure consists of an abstract list of scope objects, called the *scope chain*, analogous to stack frames in other languages. Every scope object has a pointer to a linked list of prototypes, providing prototype-based inheritance. Since scope objects inherit data from their prototypes, the value of a variable cannot be resolved by a simple list traversal. Variable resolution is further complicated by the fact that JavaScript objects may share a common prototype.

JavaScript's behaviour can make apparently simple programs deceptively counter-intuitive. Consider the code `C` defined below:

```
x = null; y = null; z = null;
f = function(w){x = v; v = 4; var v; y = v;};
v = 5; f(null); z = v;
```

What values should the variables `x`, `y` and `z` store at the end of the program? The correct answer is `undefined`, 4 and 5 respectively. We explain how this occurs as we walk through our reasoning.

In Section 6.2 we prove the following triple of this code:

$$\left\{\ \mathsf{store}_{\mathrm{LS}}(\mathtt{x},\mathtt{y},\mathtt{z},\mathtt{f},\mathtt{v}|) * \mathbf{l} \doteq \mathrm{LS}\ \right\}$$
$$\mathtt{C}$$
$$\left\{\begin{array}{l} \exists \mathrm{L}.\,\mathsf{store}_{\mathrm{LS}}(|\mathtt{x}:\mathtt{undefined},\mathtt{y}:4,\mathtt{z}:5,\mathtt{f}:\mathrm{L},\mathtt{v}:5) \\ * \mathbf{l} \doteq \mathrm{LS} * \mathrm{true} \end{array}\right\}$$

We distinguish a global logical expression $\mathbf{l}$ with value LS denoting the scope chain. The store predicate $\mathsf{store}_{\mathrm{LS}}(\mathtt{x},\mathtt{y},\mathtt{z},\mathtt{f},\mathtt{v}|)$ states that the store-like structure referred to by LS contains *none* of the program variables mentioned; the variables occur to the left of the bar. The store predicate $\mathsf{store}_{\mathrm{LS}}(|\mathtt{x}:\mathtt{undefined},\mathtt{y}:4,\mathtt{z}:5,\mathtt{f}:\mathrm{L},\mathtt{v}:5)$ denotes the final values for all the variables; the variables occur to the right of the bar with assigned values.

To understand the complexity of the heap structures described by store predicates, consider the example heap given in Figure 1. This diagram illustrates a typical shape of a JavaScript variable store. Each object is denoted by a box. In this example, the current list of scope objects is given by $\mathbf{l} = [l_1, l_2, l_3, l_4, l_g]$, where the l_i are object addresses and l_g is a distinguished object containing the global variables which must occur at the end of the current list of scope objects. Each scope object has a pointer to a list of prototypes, with the arrows representing prototype relationships. These prototype lists can be shared, but cannot form cycles. In ECMAScript 3, prototype lists must either end with the distinguished object l_{op} or be empty. However, many implementations (SpiderMonkey, V8 and WebKit) allow the programmer to directly access and change the prototype pointers, allowing incomplete prototype chains ending in `null` but not allowing the creation of cycles. We work with incomplete prototype chains, since we want ECMAScript 3 library code to work well with such implementations.

To look up the value of a variable `x` in our example heap, we check each object for a field with name `x`, starting with l_1, checking the prototype list from l_1 then moving along the list of scope objects. In our example, the `x` in object l will be found first, since the whole prototype chain of l_2 will be visited before l_3. When reading the value stored in `x`, this is all we need to know. If we write to the same variable `x`, the effect will be to create a new field `x` at l_2. This new field will override the `x` field in object l in the usual prototype-oriented way.

All of this messy detail is abstracted away by the store predicate. The formation of this predicate is subtle and requires some adaptation of separation logic. As well as the separating conjunction $*$ for reasoning about disjoint heaps, we introduce the *sepish* connective $\sqcup\!\!\!\!*$ for reasoning about partially separated heaps. It is used, for example, to account for the sharing of prototype lists illustrated in Figure 1. We also use the assertion $(l,\mathtt{x}) \mapsto \varnothing$, which states that the field `x` is *not* present at object address l. This predicate is reminiscent of the 'out' predicate in [7] stating that values are not present in a concurrent list. It is necessary to identify the first `x` in the structure: in our example, the `x` at l is the first `x`, since it does not occur in the prototype list of l_1 nor in the prototype list of l_2 until l.

Our store predicate allows us to make simple inferences about variable assignments, without breaking our store abstraction:

$$\left\{\ \mathsf{store}_{\mathrm{LS}}(\mathtt{x},\mathtt{y},\mathtt{z},\mathtt{f},\mathtt{v}|) * \mathbf{l} \doteq \mathrm{LS}\ \right\}$$
$$\mathtt{x = null;}$$
$$\left\{\begin{array}{l} \mathsf{store}_{\mathrm{LS}}(\mathtt{y},\mathtt{z},\mathtt{f},\mathtt{v}|\mathtt{x}:\mathtt{null}) \\ * \mathbf{l} \doteq \mathrm{LS} * \mathrm{true} \end{array}\right\}$$

where the assertion true hides potentially garbage-collected prototype lists.

The evaluation of the function expression `function(w)` $\{\ldots\}$ has the effect of creating a new function object and returning the address L of that object. The object contains a number of internal housekeeping fields, including $@body$ which contains the body of the function and $@scope$ which stores the function closure LS. Our inference for the function definition is approximately:

$$\left\{\ \mathsf{store}_{\mathrm{LS}}(\mathtt{f},\mathtt{v}|\mathtt{x}:\mathtt{null},\mathtt{y}:\mathtt{null},\mathtt{z}:\mathtt{null}) * \mathbf{l} \doteq \mathrm{LS}\ \right\}$$
$$\mathtt{f = function(w)}\ \{\ldots\}$$
$$\left\{\begin{array}{l} \exists \mathrm{L}.\,\mathsf{store}_{\mathrm{LS}}(\mathtt{v}|\mathtt{x}:\mathtt{null},\mathtt{y}:\mathtt{null},\mathtt{z}:\mathtt{null},\mathtt{f}:\mathrm{L}) * \\ (\mathrm{L}, @body) \mapsto \lambda w.\{\ldots\} * (\mathrm{L}, @scope) \mapsto \mathrm{LS} * \\ \mathbf{l} \doteq \mathrm{LS} * \mathrm{true} \end{array}\right\}$$

As well as the store predicate, we assert that the state also contains object cells such as $(\mathrm{L}, @scope) \mapsto \mathrm{LS}$. This assertion means that there is an object with address L in the heap, and it definitely contains at least the field $@scope$ which has value LS. The assertion says nothing about any other field of L. We assert that our function object has fields $@body$ and $@scope$. The full specification, given in Section 6.2, is actually a little more complicated than this. For now, we hide additional housekeeping fields in the assertion true.

We know that this program example is challenging, because the final values of the variables are counter-intuitive. All the complexity of the example occurs within the function call. When JavaScript calls a function, it performs two passes on the body: in the first pass, it creates a new scope object and initialises local variables to `undefined`; in the second pass, it runs the code in the newly constructed local scope. Our reasoning reflects this complexity. The Hoare triple for the function call has the following shape:

$$\left\{\begin{array}{l}\mathsf{store}_{\mathrm{LS}}(|\mathtt{x}:\mathtt{null},\mathtt{y}:\mathtt{null},\mathtt{z}:\mathtt{null},\mathtt{f}:\mathrm{L},\mathtt{v}:5)*\\(\mathrm{L},@\mathit{body})\mapsto\lambda w.\{\ldots\}*(\mathrm{L},@\mathit{scope})\mapsto\mathrm{LS}*\\\mathbf{l}\doteq\mathrm{LS}*\mathrm{true}\end{array}\right\}$$
$$\mathtt{f(null);}$$
$$\{\ \mathbf{???}\ \}$$

To find a suitable postcondition, we must reason about the function body. The precondition of the function-body triple given below is constructed from the first pass of the function call. As well as containing the precondition of the function call, it contains a new scope object L' with fields given by the parameter of the function and the local variables discovered by the first pass. For our example, it contains the assertions $(\mathrm{L}',\mathtt{w})\mapsto\mathtt{null}$ for the parameter declaration and $(\mathrm{L}',\mathtt{v})\mapsto\mathtt{undefined}$ for the local variable declaration. The object L' also has a $@\mathit{proto}$ field, which points to `null` since scope objects do not inherit any behaviour, and a $@\mathit{this}$ field, which can only be read. We also have the predicate $\mathsf{newobj}(\mathrm{L}',@\mathit{proto},@\mathit{this},\mathtt{w},\mathtt{v})$, which asserts the absence from L' of all the fields we have not mentioned as parameters. Knowing this absence of fields is essential if, in the function body, we wish to write to variables, such as the `x` and `y`, which do not appear in the local scope object. Finally, the new scope object L' is prepended to the scope list $\mathbf{l}$.

Using this precondition, we are now able to give the triple obtained by the second pass of the function call:

$$\left\{\begin{array}{l}\exists\mathrm{L}'.\,\mathbf{l}\doteq\mathrm{L}':\mathrm{LS}*\\\mathsf{store}_{\mathrm{LS}}(|\mathtt{x}:\mathtt{null},\mathtt{y}:\mathtt{null},\mathtt{z}:\mathtt{null},\mathtt{f}:\mathrm{L},\mathtt{v}:5)*\\(\mathrm{L},@\mathit{body})\mapsto\lambda w.\{\ldots\}*(\mathrm{L},@\mathit{scope})\mapsto\mathrm{LS}*\\\mathsf{newobj}(\mathrm{L}',@\mathit{proto},@\mathit{this},\mathtt{w},\mathtt{v})*\\(\mathrm{L}',\mathtt{w})\mapsto\mathtt{null}*(\mathrm{L}',\mathtt{v})\mapsto\mathtt{undefined}*\\(\mathrm{L}',@\mathit{proto})\mapsto\mathtt{null}*(\mathrm{L}',@\mathit{this})\mapsto\mathrm{L}''*\mathrm{true}\end{array}\right\}$$
$$\mathtt{x\ =\ v\ ;\ v\ =\ 4\ ;\ var\ v\ ;\ y\ =\ v;}$$
$$\left\{\begin{array}{l}\exists\mathrm{L}'.\,\mathbf{l}\doteq\mathrm{L}':\mathrm{LS}*\\\mathsf{store}_{\mathrm{LS}}(|\mathtt{x}:\mathtt{undefined},\mathtt{y}:4,\mathtt{z}:\mathtt{null},\mathtt{f}:\mathrm{L},\mathtt{v}:5)*\\(\mathrm{L},@\mathit{body})\mapsto\lambda w.\{\ldots\}*(\mathrm{L},@\mathit{scope})\mapsto\mathrm{LS}*\\\mathsf{newobj}(\mathrm{L}',@\mathit{proto},@\mathit{this},\mathtt{w},\mathtt{v})*\\(\mathrm{L}',\mathtt{w})\mapsto\mathtt{null}*(\mathrm{L}',\mathtt{v})\mapsto 4*\\(\mathrm{L}',@\mathit{proto})\mapsto\mathtt{null}*(\mathrm{L}',@\mathit{this})\mapsto\mathrm{L}''*\mathrm{true}\end{array}\right\}$$

The postcondition follows simply from the three assignments in the function body: first, variable `x` gets the value `undefined`, since this is the current value of the local `v`; then the local `v` is assigned 4; and, finally, the global variable `y` is assigned the value of the local variable `v`. The `var v` statement has no effect in the second pass of the function call.

The postcondition of the function call is the postcondition of the function body, with local scope object L' popped off the current scope list $\mathbf{l}$:

$$\left\{\begin{array}{l}\exists\mathrm{L}'.\,\mathsf{store}_{\mathrm{LS}}(|\mathtt{x}:\mathtt{undefined},\mathtt{y}:4,\mathtt{z}:\mathtt{null},\mathtt{f}:\mathrm{L},\mathtt{v}:5)*\\(\mathrm{L},@\mathit{body})\mapsto\lambda w.\{\ldots\}*(\mathrm{L},@\mathit{scope})\mapsto\mathrm{LS}*\\\mathsf{newobj}(\mathrm{L}',@\mathit{proto},@\mathit{this},\mathtt{w},\mathtt{v})*\\(\mathrm{L}',\mathtt{w})\mapsto\mathtt{null}*(\mathrm{L}',\mathtt{v})\mapsto 4*(\mathrm{L}',@\mathit{proto})\mapsto\mathtt{null}*\\(\mathrm{L}',@\mathit{this})\mapsto L''*\mathbf{l}\doteq\mathrm{LS}*\mathrm{true}\end{array}\right\}$$

Reasoning about the final assignment is straightforward, with `z` assigned the value of the global variable `v`. The final postcondition is obtained using the consequence rule to hide the function object and local scope object behind the assertion true, since they are surplus to requirements, and existentially quantifying function object L:

$$\left\{\begin{array}{l}\exists\mathrm{L}.\,\mathsf{store}_{\mathrm{LS}}(|\mathtt{x}:\mathtt{undefined},\mathtt{y}:4,\mathtt{z}:5,\mathtt{f}:\mathrm{L},\mathtt{v}:5)*\\\mathbf{l}\doteq\mathrm{LS}*\mathrm{true}\end{array}\right\}$$

Part of the challenge of understanding this example is knowing the scope of local variable `v`. In JavaScript, variables can only be declared local to functions, not other blocks such as `if` and `while`. This can lead to undesirable behaviour, especially when a local variable overloads the name of a global variable. One controversial technique for solving this problem is to use the `with` statement and a literal object to declare local variable blocks precisely where they are needed. Using `with` is often considered bad practice, and it is deprecated in ECMAScript 5. However, it is widely used in practice [22, 28] and can certainly be used to improve the program readability. We are able to reason about even extremely confusing uses of `with`. Consider the program $\mathtt{C}'$:

```
a = {b:1}; with (a){f=function(c){return b}};
a = {b:2}; f(null)
```

Armed with an operational understanding of JavaScript's emulated variable store, it is not so difficult to understand that this program returns the value 1, even though the value of `a.b` at the end of the program is 2. It may not be quite so clear that this program can fault. It may also execute arbitrary code from elsewhere in the emulated variable store, leading to a possible security violation.

We only understood this example properly by doing the verification. In Section 6.2, we prove the triple:

$$\{\mathsf{store}_{\mathbf{l}}(\mathsf{a},\mathtt{f}|)\boxast(l_{op},\mathtt{f})\mapsto\varnothing\boxast(l_{op},@\mathit{proto})\mapsto\mathtt{null}\}$$
$$\mathtt{C}'$$
$$\{\mathbf{r}\doteq 1*\mathrm{true}\}$$

A similar proof is possible for a precondition where `a` and `f` are in the store with arbitrary values. Either precondition ensures the program returns the value 1 as expected. The obvious first try was to have e.g. just $\mathsf{store}_{\mathbf{l}}(\mathsf{a},\mathtt{f}|)$ as the precondition. This does not work as, when reasoning about the assignment to the variable `f`, we cannot assert that the variable `f` is not in the local scope. As discussed earlier, we work with incomplete prototype chains in the emulated variable store. With some web code, it is possible for programmers to directly access and change the prototype pointers, resulting in the distinguished object l_{op} not being a part of the emulated variable store. This means that l_{op} may contain the field `f` without violating our proposed precondition. The statement `a = {b:1}` then results in the creation of a new object L, with no field `f` as expected, but with field $@\mathit{proto}$ pointing to l_{op} which may contain f:

$$\exists\mathrm{L}.\,(\mathrm{L},\mathtt{b})\mapsto 1*(\mathrm{L},\mathtt{f})\mapsto\varnothing*(\mathrm{L},@\mathit{proto})\mapsto l_{op}$$

The `with` statement makes this new object L the most local cell in our emulated variable store. But because `f` is an inherited property of L, the meaning of the assignment to `f` has now changed – it has become an overriding assignment of a local variable. We write the new function `f` into the most local object L. When the program returns from the `with` statement, the function call to `f(null)` will fault, since there is no `f` in scope to call. In many web browsers (including Chrome, Firefox, and Safari), this behaviour can be induced by running the program `Object.prototype.f = 4 ; window.__proto__ = null ; C'` in the default starting state[1]. The `__proto__` notation allows the programmer to directly access and change the internal $@\mathit{proto}$

[1] In web browsers, `Object.prototype` usually contains a pointer to the object l_{op} and `window` usually contains a pointer to the object l_g.

fields, resulting in a program state which causes the program `C'` to fault. Non-standard features such as this are widely used in practice, and so it is important that our reasoning be robust to the states they produce. In this example, it is possible to induce a similarly tricky starting state without the use of any non-standard features, which results in the call to `f()` executing arbitrary code not mentioned in our program. Consider running the program `Object.prototype.f = function(c) {C''};C'` where `C''` is suspicious code. This could result in a security breach if `f` is passed sensitive data.

3. Operational Semantics

We define a big-step operational semantics for a large subset of JavaScript that represents faithfully the inheritance, prototyping and scoping mechanisms described in the ECMAScript 3 standard. Our semantics follows closely the full small-step JavaScript semantics of Maffeis, Mitchell and Taly [18], except that we make some simplifications as discussed in Section 3.5. We work with a big-step semantics because it connects better to our reasoning.

3.1 Heaps

The JavaScript heap is a partial function $H: \mathcal{R} \rightharpoonup \mathcal{V}$ that maps *references*, $r \in \mathcal{R} = \mathcal{L} \times \mathcal{X}$, which are pairs of memory locations and field names, to values. This structure emphasises the important role that references play in the semantics of the language. Values $v \in \mathcal{V}$ can be basic values $\mathtt{v}$, locations l and lambda abstractions $\lambda\mathtt{x}.\mathtt{e}$. The set of locations $\mathcal{L}$ is lifted to a set $\mathcal{L}_{\mathtt{null}}$ containing the special location `null`, analogous to a null-pointer in C, which cannot be in the domain of any heap. We denote the empty heap by `emp`, a heap cell by $(l, x) \mapsto v$, the union of two disjoint heaps by $H_1 * H_2$, and a read operation by $H(l, x)$.

An object is represented by a set of heap cells addressed by the same location but with different field names. For ease of notation, we use $l \mapsto \{x_1 \colon v_1, \ldots, x_n \colon v_n\}$ as a shorthand for the object $(l, x_1) \mapsto v_1 * \ldots * (l, x_n) \mapsto v_n$.

As discussed, JavaScript has no variable store. Instead, variables are resolved with respect to a scope object implicitly known at run time. Scope objects are just objects whose locations are recorded in the *scope chain* (we use a standard notation $[\,], e{:}L, L +\!\!+\, L$ for lists). Each scope object has a pointer to a *prototype list* (which need not point to l_{op}). A variable x is resolved as the property named "x" of the first object in the scope chain whose prototype list defines "x". Scoping constructs, such as function calls and `with`, cause sub-expressions to be evaluated with respect to a local scope object, by putting the local scope object at the beginning of the scope chain and then removing it after the sub-expressions have been evaluated. All user programs are evaluated starting from the default scope chain $[l_g]$, where l_g is the location of the global JavaScript object, described below. The final object in any scope chain is always l_g, but duplicates in a scope chain *are* allowed. For example, the JavaScript program `with(window){C}` is perfectly valid, and (usually) results in the subprogram `C` being evaluated in a state in which l_g is both the most global *and* the most local object.

The auxiliary scope function σ, defined below, returns the location of the first object in the scope chain to define a given variable. It depends on the prototype function π, which returns the location of the first object in the prototype chain to define the variable.

Scope and prototype resolution: $\sigma(H, l, x)$ **and** $\pi(H, l, x)$**.**

$$\sigma(H, [\,], x) \triangleq \mathtt{null}$$

$$\frac{\pi(H, l, x) \neq \mathtt{null}}{\sigma(H, l{:}L, x) \triangleq l} \qquad \frac{\pi(H, l, x) = \mathtt{null}}{\sigma(H, l{:}L, x) \triangleq \sigma(H, L, x)}$$

$$\pi(H, \mathtt{null}, x) \triangleq \mathtt{null}$$

$$\frac{(l, x) \in \mathsf{dom}(H)}{\pi(H, l, x) \triangleq l} \qquad \frac{(l, x) \notin \mathsf{dom}(H) \qquad H(l, @proto) = l'}{\pi(H, l, x) \triangleq \pi(H, l', x)}$$

The set of variable names $\mathcal{X}$ is partitioned in two disjoint sets: the *internal* variable names$\mathcal{X}^{\mathrm{I}}$ and the *user* variable names $\mathcal{X}^{\mathrm{U}}$. The internal names $\mathcal{X}^{\mathrm{I}} \triangleq \{@scope, @body, @proto, @this\}$ are not directly accessible by user-defined ECMAScript 3 code, but are used by the semantics. As noted in Section 2, some implementations do give programmers direct access to some of these internal variables, but we do not model those implementation-dependant programs. We do model all the program states that such programs might create, and take care not to make strong assumptions about the types or values of those suspect internal variables. In this way, libraries verified using our techniques are robust in the presence of unverified real-world code. User names are denoted by $\mathtt{x}, \mathtt{y}, \mathtt{z} \in \mathcal{X}^{\mathrm{U}}$ and are considered a subset of strings; keywords such as `var` are not valid variable names. It is worth anticipating at this point a subtlety of the JavaScript semantics. The evaluation of a user variable `x` does not return its value, but rather the reference $l' \cdot \mathtt{x}$ where such value can be found (l' is obtained using the σ predicate). In general, the values $r \in \mathcal{V}^{\mathrm{R}}$ returned by JavaScript expressions can be normal values $\mathcal{V}$ or references $\mathcal{R}$. When a user variable `x` needs to be dereferenced in an expression, the semantics implicitly calls the dereferencing function γ, defined below, which returns the value denoted by the reference.

Dereferencing values: $\gamma(H, r)$**.**

$$\frac{r \neq l \cdot \mathtt{x}}{\gamma(H, r) \triangleq r} \qquad \frac{\begin{array}{l}\pi(H, l, \mathtt{x}) = \mathtt{null} \\ l \neq \mathtt{null}\end{array}}{\gamma(H, l \cdot \mathtt{x}) \triangleq \mathtt{undefined}} \qquad \frac{\begin{array}{l}\pi(H, l, \mathtt{x}) = l' \\ l' \neq \mathtt{null}\end{array}}{\gamma(H, l \cdot \mathtt{x}) \triangleq H(l', \mathtt{x})}$$

We introduce the definition of a well-formed JavaScript heap. A JavaScript expression can only be evaluated in a sub-heap of a well-formed heap, with respect to a valid scope chain. All JavaScript expressions maintain the well-formedness of heaps. A heap H is *well-formed* (denoted by $H \vdash \diamond$) if its objects and prototype chains are well-formed (prototype chains must be acyclic, they can end with `null` rather than l_{op}), and if it contains the global scope object l_g and the distinguished objects l_{op} and l_{fp}, which we will see later serve as the default prototypes for new objects and functions. A scope chain L is *valid* with respect to heap H, denoted by $\mathsf{schain}(H, L)$, if all the locations in the chain correspond to objects allocated in H, and if it ends with the global object l_g. Formal definitions are given in [11].

The default initial state $H^{\emptyset}$ is the smallest well-formed heap that also contains the special function object l_e (the `eval` function) and its prototype l_{ep}:

$$H^{\emptyset} \;\triangleq\; \left(\begin{array}{c} \mathsf{obj}(l_g, l_{op}) * \; \mathsf{obj}(l_{op}, \mathtt{null}) \\ * \, \mathsf{obj}(l_{fp}, l_{op}) * \; \mathsf{obj}(l_e, l_{ep}) * \mathsf{obj}(l_{ep}, l_{op}) \end{array} \right)$$

where $\mathsf{obj}(l, l')$ denotes an object at location l which is empty except for the prototype l':

$$\mathsf{obj}(l, l') \;\triangleq\; (l, @proto) \mapsto l'$$

We conclude this section by defining the heap update $-[-]$ operation which will be used by the semantics. Many JavaScript expressions update a heap cell if it is present, or create it if it is not present. We reflect this form of update in our heap update operation.

Update $H[(l, \mathtt{x}) \mapsto v]$**.**

$$\frac{(l, \mathtt{x}) \notin dom(H) \qquad l \neq \mathtt{null}}{H[(l, \mathtt{x}) \mapsto v] \triangleq H * (l, x) \mapsto v}$$

$$(H * (l, \mathtt{x}) \mapsto v)[(l, \mathtt{x}) \mapsto v'] \triangleq H * (l, \mathtt{x}) \mapsto v'$$

$$H[(\mathtt{null}, \mathtt{x}) \mapsto v] \triangleq H[(l_g, \mathtt{x}) \mapsto v]$$

The last rule says that an update to a non-allocated reference (whose location is `null`) has the effect of allocating and updating a new global variable, which mimics JavaScript's behaviour of implicitly creating global variables when a variable name is first used.

3.2 Values and Expressions

To aid our presentation, we introduce the syntax of all JavaScript programs, statements and expressions simply as expressions. Our operational semantics will only describe the behaviour of well-formed programs, as defined by [18].

Syntax of terms: values $\mathtt{v}$ and expressions $\mathtt{e}$.

$$\begin{array}{lll}
\mathtt{v} & ::= & \mathtt{n} \mid \mathtt{m} \mid \mathtt{undefined} \mid \mathtt{null} \\
\mathtt{e} & ::= & \mathtt{e;e} \mid \mathtt{x} \mid \mathtt{v} \mid \mathtt{if(e)\{e\}\{e\}} \mid \mathtt{while(e)\{e\}} \mid \mathtt{var\ x} \\
 & & \mid \mathtt{this} \mid \mathtt{delete\ e} \mid \mathtt{e} \oplus \mathtt{e} \mid \mathtt{e.x} \mid \mathtt{e(e)} \mid \mathtt{e\ =\ e} \\
 & & \mid \mathtt{function\,(x)\{e\}} \mid \mathtt{function\,x(x)\{e\}} \mid \mathtt{new\ e(e)} \\
 & & \mid \{\mathtt{x}_1 : \mathtt{e}_1 \ldots \mathtt{x}_n : \mathtt{e}_n\} \mid \mathtt{e[e]} \mid \mathtt{with(e)\{e\}}
\end{array}$$

where $\oplus \in \{+, -, *, /, \&\&, ||, ==, .\}$

A basic value $\mathtt{v}$ can be a number $\mathtt{n}$, a string $\mathtt{m}$ (including the user variable names), the special constant `undefined` or the `null` location. The operator $\oplus$ denotes a standard number and boolean operator, or string concatenation. Expressions $\mathtt{e}$ include sequential composition, variable lookup, literal values, conditional expressions, loops, local variable declaration, `this`, deletion, arithmetic and string concatenation, object property lookup, function call and eval, assignment, function declaration, recursive functions, constructors, literal objects, computed access and the `with` statement.

3.3 Evaluation Rules

An expression $\mathtt{e}$ is evaluated in a heap H, with respect to a scope chain L. If it successfully terminates, it returns a modified heap H' and a final value r. Selected evaluation rules are given below and then discussed. See [11] for the full definition. Recall that the set of variables is a subset of the set of strings, that a heap value v can be a basic value $\mathtt{v}$, a memory location l or a function closure $\lambda\mathtt{x.e}$, and that a return value r can also be a reference $l{\cdot}\mathtt{x}$.

Operational semantics: $H, L, \mathtt{e} \longrightarrow H', r$**.**

Notation: $H, L, \mathtt{e} \xrightarrow{\gamma} H', v \triangleq \exists r.(H, L, \mathtt{e} \longrightarrow H', r \wedge \gamma(H', r) = v)$.

(Definition)

$$\frac{H, L, \mathtt{e} \longrightarrow H', v}{H, L, \mathtt{var\ \ e} \longrightarrow H', \mathtt{undefined}}$$

(Value)

$$H, L, \mathtt{v} \longrightarrow H, \mathtt{v}$$

(Member Access)

$$\frac{\begin{array}{l} H, L, \mathtt{e} \xrightarrow{\gamma} H', l \\ l \neq \mathtt{null} \end{array}}{H, L, \mathtt{e.x} \longrightarrow H', l{\cdot}\mathtt{x}}$$

(Computed Access)

$$\frac{\begin{array}{l} H, L, \mathtt{e1} \xrightarrow{\gamma} H_1, l \\ l \neq \mathtt{null} \\ H_1, L, \mathtt{e2} \xrightarrow{\gamma} H', \mathtt{x} \end{array}}{H, L, \mathtt{e1[e2]} \longrightarrow H', l{\cdot}\mathtt{x}}$$

(Variable)

$$\frac{\sigma(H, L, \mathtt{x}) = l}{H, L, \mathtt{x} \longrightarrow H, l{\cdot}\mathtt{x}}$$

(Object)

$$\frac{\begin{array}{l} H_0 = H * \mathsf{obj}(l, l_{op}) \\ \forall i \in 1..n. \left(\begin{array}{l} H_{i-1}, L, \mathtt{ei} \xrightarrow{\gamma} H'_i, v_i \\ H_i = H'_i[(l, \mathtt{xi}) \mapsto v_i] \end{array} \right) \end{array}}{H, L, \{\mathtt{x1:e1}, \ldots, \mathtt{xn:en}\} \longrightarrow H_n, l}$$

(Binary Operators)

$$\frac{\begin{array}{l} H, L, \mathtt{e1} \xrightarrow{\gamma} H'', \mathtt{v1} \\ H'', L, \mathtt{e2} \xrightarrow{\gamma} H', \mathtt{v2} \\ \mathtt{v1} \,\overline{\oplus}\, \mathtt{v2} = \mathtt{v} \end{array}}{H, L, \mathtt{e1} \oplus \mathtt{e2} \longrightarrow H', \mathtt{v}}$$

(Assignment)

$$\frac{\begin{array}{l} H, L, \mathtt{e1} \longrightarrow H_1, l{\cdot}\mathtt{x} \\ H_1, L, \mathtt{e2} \xrightarrow{\gamma} H_2, v \\ H' = H_2[(l, \mathtt{x}) \mapsto v] \end{array}}{H, L, \mathtt{e1=e2} \longrightarrow H', v}$$

(This)

$$\frac{\begin{array}{l} \sigma(H, L, @\mathit{this}) = l_1 \\ \pi(H, l_1, @\mathit{this}) = l_2 \\ H(l_2, @\mathit{this}) = l' \end{array}}{H, L, \mathtt{this} \longrightarrow H, l'}$$

(Function)

$$\frac{H' = H * \mathsf{obj}(l, l_{op}) * \mathsf{fun}(l', L, \mathtt{x}, \mathtt{e}, l)}{H, L, \mathtt{function\,(x)\{e\}} \longrightarrow H', l'}$$

(Function Call)

$$\frac{\begin{array}{lll} H, L, \mathtt{e1} \longrightarrow H_1, r_1 & \mathsf{This}(H_1, r_1) = l_2 & \gamma(H_1, r_1) = l_1 \\ l_1 \neq l_e & H_1(l_1, @\mathit{body}) = \lambda\mathtt{x.e3} & H_1(l_1, @\mathit{scope}) = L' \\ H_1, L, \mathtt{e2} \xrightarrow{\gamma} H_2, v & & \\ H_3 = H_2 * \mathsf{act}(l, \mathtt{x}, v, \mathtt{e3}, l_2) & H_3, l{:}L', \mathtt{e3} \xrightarrow{\gamma} H', v' & \end{array}}{H, L, \mathtt{e1(e2)} \longrightarrow H', v'}$$

(Eval)

$$\frac{\begin{array}{ll} H, L, \mathtt{e1} \xrightarrow{\gamma} H_1, l_e & H_1, L, \mathtt{e2} \xrightarrow{\gamma} H_2, \mathtt{m} \\ \mathsf{parse}(\mathtt{m}) = \mathtt{e} & H_2, L, \mathtt{e} \xrightarrow{\gamma} H', v' \end{array}}{H, L, \mathtt{e1(e2)} \longrightarrow H', v'}$$

(With)

$$\frac{\begin{array}{l} H, L, \mathtt{e1} \xrightarrow{\gamma} H_1, l \\ H_1, l{:}L, \mathtt{e2} \longrightarrow H', r \end{array}}{H, L, \mathtt{with(e1)\{e2\}} \longrightarrow H', r}$$

$\sigma, \pi, \gamma, \mathsf{obj}$ defined earlier
$\mathsf{fun}, \mathtt{this}, \mathsf{act}$ defined below

We briefly discuss some of the evaluation rules that show non-standard features typical of JavaScript. Rule (Definition) for `var  e` simply executes `e` and throws away the return value. The `var` declaration is only used by defs (defined below) to identify function local variables. Rule (Variable) uses σ to determine the scope object where a given variable can be found, without de-referencing the variable. Rules (Member/Computed Access) return a reference to the object field denoted by the corresponding expressions. Rule (Object) uses the obj notation introduced in Section 3.1 to introduce a fresh, empty object at location l, and then initialises its fields accordingly. Freshness is ensured by well-formedness of H and disjointness of $*$. Rule (Binary Operators) assumes the existence of a semantic version $\overline{\oplus}$ for each syntactic operator $\oplus$. Each $\overline{\oplus}$ is a partial function, defined only on arguments of a basic type (in this case numbers or strings) and returning results of some other (possibly the same) basic type, corresponding to the intended meaning of the operation. Rule (Assignment) is quite subtle. Suppose we have the expression `x=4`. Now consider Figure 1 of Section 2. If `x` were defined as a field of the object l_1 then `x=4` would be an *overwriting* assignment. The value of the field `x` in the object l_1 would be overwritten with the value 4. If `x` were not found anywhere, then it would be created as a global variable – a field of the object l_g. Finally, consider the actual case in this figure: `x` is found to be a field of object l, which is a prototype of l_2, which is in the scope chain. In this case, `x=4` is an *overriding* assignment, with the effect of creating a new field `x` in the object l_2 to override the existing `x` in l. This complexity is handled in two places. Firstly, the variable rule uses the σ function (defined in Section 3.1) to return a reference $l_2{\cdot}\mathtt{x}$. Note that the σ function does not return a reference to $l{\cdot}\mathtt{x}$ precisely because we wish to model this behaviour. Secondly, the heap update operation $H_2[\ldots]$ manages the business of either overwriting an existing field or, in this case, creating a new field. Rule (This) resolves the `this` identifier. As we will see, when executing a method of an object, we use the internal variable $@\mathit{this}$ to store the location of that object. When executing a function call, $@\mathit{this}$ points to the global object l_g. The (This) rule uses σ and π to retrieve the value of $@\mathit{this}$ which is then returned by the `this` statement. Rule (Function) introduces the notation $\mathsf{fun}(l', L, \mathtt{x}, \mathtt{e}, l) \triangleq$

$$l' \mapsto \{@\mathit{proto}{:}\, l_{fp}, \mathtt{prototype}{:}\, l, @\mathit{scope}{:}\, L, @\mathit{body}{:}\, \lambda\mathtt{x.e}\}$$

to allocate a fresh function object at location l'. The internal prototype of the new function object points to the standard function prototype object l_{fp}. The rule also creates a new empty object at l and stores a pointer to it in the `prototype` field of the new function. If the function is ever used as a constructor (using the `new` keyword) then the object pointed to by this `prototype` field will serve as the prototype of the newly constructed object. Note that the field is mutable, and so may be used by a programmer to emulate a class-like inheritance structure.

Recall from Section 2 that JavaScript function calls can be surprisingly complex. We now describe rule (Function Call), which uses two auxiliary functions This and act. Recall also that the rule (This) uses internal $@\mathit{this}$ fields to determine the semantics of the `this` keyword. The values of the $@\mathit{this}$ fields are determined by the (Function Call) rule using the auxiliary function This:

$$\begin{array}{rcll} \mathsf{This}(H, l{\cdot}\mathtt{x}) & \triangleq & l & [(l, @\mathit{this}) \notin \mathit{dom}(H)] \\ \mathsf{This}(H, r) & \triangleq & l_g & [\text{otherwise}] \end{array}$$

To understand This, first notice that every newly created local scope object has a $@\mathit{this}$ field, while no other objects ever will[2]. The (Function Call) rule finds a pointer to a function in the location $l{\cdot}\mathtt{x}$. If l is a regular object (which has no $@\mathit{this}$ field), then the function must be a method of that object, and so the $@\mathit{this}$ field of our new local scope object should point to l. On the other hand, if l is a special local scope object (which has a $@\mathit{this}$ field), then the function must be a regular function (and not a method), and so the $@\mathit{this}$ of our new local scope object should point to l_g. This unique behaviour precisely captures the behaviour of the ECMAScript 3 `this` keyword. The auxiliary function act describes the allocation of a new local scope object: $\mathsf{act}(l, \mathtt{x}, v, \mathsf{e}, l'') \triangleq$

$$l \mapsto \{\mathtt{x}\colon v, @\mathit{this}\colon l'', @\mathit{proto}\colon \mathtt{null}\} * \mathsf{defs}(\mathtt{x}, l, \mathsf{e})$$

The object is allocated at address l, and contains a function parameter $\mathtt{x}$ with value v, the internal fields $@\mathit{this}$, $@\mathit{proto}$ and the local variables declared in an expression e. The auxiliary function defs, defined in [11], searches the function body e for instances of the `var` keyword, and sets all the appropriate fields of our new local scope object to the value `undefined`, as discussed in Section 2. Rule (Eval) looks like an ordinary function call, but the function being called is not an ordinary function object. It is the special built-in object l_e. It assumes a partial function parse that parses a string `m` into a JavaScript expression e, only if there are no syntax errors.

The control expressions are mostly standard (see [11]), except for the unusual (With) rule that evaluates `e2` in a scope-chain starting with the object obtained by evaluating `e1`.

3.4 Safety

An important sanity property of the evaluation relation is that it preserves well-formedness of the heap, for any valid scope chain.

Theorem 1 (Well-Formedness)**.** *Let H, L be such that $H \vdash \diamond$ and $\mathsf{schain}(H, L)$. If $H, L, \mathsf{e} \longrightarrow H', r$ then $H' \vdash \diamond$.*

Although the theorem is about end-to-end well-formedness, its proof (reported in [11]) shows that starting from a well-formed state and scope chain, all the intermediate states and scope chains visited during the computation are also well-formed, and all the locations occurring in intermediate return values correspond to objects effectively allocated on the heap.

3.5 JavaScript Subset

We work with a subset of JavaScript, in order to limit the size and complexity of our semantics for this paper. Our subset is substantial and, despite some minor omissions and simplifying assumptions discussed below, faithful to the ECMAScript 3 standard. A significant property of our semantics is that our programs will run reliably in states generated by any valid JavaScript program (including those reached by programs using non-standard features that we do not model, such as `__proto__`), or getters and setters. Our reasoning of Section 5 will therefore interface well with real-world JavaScript programs.

We do not model implicit type-coercion functions. Adding them is straightforward but would add significantly to the length of our presentation. We have no boolean type. Instead, where control structures (`if` and `while`) require a boolean, we use other types whose behaviour is equivalent to the type conversion that occurs in JavaScript. Values such as 0 and `null` behave like `false` and values such as 1 and `"string"` behave like `true`. For simplicity, we use an implicit return statement for functions. Moreover, our functions take only one parameter, rather than the arbitrary list of parameters usual in JavaScript, and do not have the `arguments` object or the `constructor` property. As mentioned in Section 3.2, we simplify our presentation of JavaScript programs, statements and expressions, into a single class of expressions. We also omit several JavaScript constructs such as labels, `switch` and `for`, as they do not contribute significantly to the problem of program reasoning. In this presentation we only consider the core language ECMAScript 3, and do not model the many standard libraries which browsers make available to programmers. Instead of exceptions, we have a single error condition denoted fault. Our reasoning conservatively avoids faults. This means that programs which are proved using our fault-avoiding local Hoare reasoning will run without throwing exceptions in JavaScript interpreters.

4. Assertion Language

Our assertion language follows that of Parkinson and Bierman [4, 23, 24], in their work on reasoning about Java. They use assertions of the form $(\mathrm{l}, \mathrm{x}) \mapsto \mathrm{v}$ to denote that a partial heap contains the object l with field x which has value v. Using the separating conjunction $*$ [20], the assertion $((\mathrm{l}, \mathrm{x}) \mapsto \mathrm{v}) * ((\mathrm{l}, \mathrm{y}) \mapsto \mathrm{w})$ declares that a heap contains an object l which must have two separate fields x, y with the appropriate values. The assertion $(\mathrm{l}, \mathrm{x}) \mapsto \mathrm{v} * (\mathrm{l}, \mathrm{x}) \mapsto \mathrm{w}$ is unsatisfiable since it declares two fields x for l.

This style of reasoning is not enough for JavaScript. We must also assert negative information about fields not being in the heap, and extend the structural assertions of separation logic to account for partial separation due to shared prototype chains. Recall the example of a JavaScript emulated variable store in Figure 1. To find the value of x in the store, we must not only determine that the object l has a field x but also determine that no fields named x occur earlier in the emulated store. We use assertions $(\mathrm{l}, \mathrm{x}) \mapsto \varnothing$ to declare that a heap contains an object l which *does not* have a field x. The assertion $(\mathrm{l}, \mathrm{x}) \mapsto \varnothing * (\mathrm{l}, \mathrm{y}) \mapsto \mathrm{w}$ declares that the heap contains an object l which does not have field x but does have field y. The assertions $(\mathrm{l}, \mathrm{x}) \mapsto \mathrm{v} * (\mathrm{l}, \mathrm{x}) \mapsto \varnothing$ and $(\mathrm{l}, \mathrm{x}) \mapsto \varnothing * (\mathrm{l}, \mathrm{x}) \mapsto \varnothing$ are unsatisfiable. Thus, the assertion $(\mathrm{l}, \mathrm{x}) \mapsto \varnothing$ states the full knowledge that field x is not in object l.

Now consider what happens when we want to describe the state of more than one variable at a time. In Section 5 we introduce a predicate σ which allows us to assert, for example, "The variable `x` is found in the store in object l_2": $\sigma(\_, \mathbf{l}, \mathtt{x}, l_2)$ or "The variable `y` is not in the store": $\sigma(\_, \mathbf{l}, \mathtt{y}, \mathtt{null})$. Both of these assertions must make use of the $@\mathit{proto}$ fields in order to navigate the variable store, so we cannot separate them using $*$. But the first assertion does not mention any `y` field, and the second assertion does not mention any `x` field, so we cannot join them with $\wedge$. In order to make both assertions about the same variable store, we need a way for them to share their common parts. To do this, we introduce the *sepish connective* $P \mathbin{\boxast} Q$ which allows *partial* separation between heaps. We can use $\boxast$ to describe the state of more than one variable in a given store: $\sigma(\_, \mathbf{l}, \mathtt{x}, l_2) \mathbin{\boxast} \sigma(\_, \mathbf{l}, \mathtt{y}, \mathtt{null})$, and we shall see in Section 5 that it is also invaluable when defining the σ predicate.

An assertion may be satisfied by a triple (h, L, ϵ) consisting of an abstract heap h, a scope chain L, and a logical environment ϵ. An *abstract JavaScript heap* is a partial function $H\colon \mathcal{R} \rightharpoonup \mathcal{V} \cup \{\varnothing\}$ that maps references, $r \in \mathcal{R} = \mathcal{L} \times \mathcal{X}$ to values $v \in \mathcal{V}$ or $\varnothing$. Abstract heaps thus declare information about fields not being present in an object, as well as the fields that are present. We also define an

[2] l_g has $@\mathit{this}$ in some, not all, implementations; [14] is silent on the issue.

evaluation function $\lfloor\_\rfloor$ which takes an abstract heap to a concrete heap:

$$\lfloor h\rfloor(l,x) \triangleq h(l,x) \text{ iff } (l,x)\in\mathsf{dom}(h)\wedge h(l,x)\neq\varnothing$$

We use this function in Section 5 to define the relationship between our reasoning using Hoare triples and our operational semantics.

We define a logical environment ϵ, which is a partial function from logical variables $\mathrm{X}\in\mathcal{X}^{\mathrm{L}}$ to logical values $\mathcal{V}^{\mathrm{L}}$, which may be a return value r, any expression $\mathtt{e}$, $\varnothing$ or a list Ls of logical values. We also define logical expressions E, which are different from program expressions in that they can not read or alter the heap. Expressions E are evaluated in a logical environment ϵ with respect to a current scope chain L.

Logical expressions and evaluation: $[\![E]\!]^L_\epsilon$**.**

$$\mathrm{v}\in\mathcal{V}^{\mathrm{L}} ::= e \mid v \mid \varnothing \mid Ls \qquad\qquad \epsilon:\mathcal{X}^{\mathrm{L}}\rightharpoonup\mathcal{V}^{\mathrm{L}}$$

$$\begin{array}{lll}
E ::= & \mathrm{X} & \text{Logical variables}\\
 & \mid \mathbf{l} & \text{Scope list}\\
 & \mid \mathrm{v} & \text{Logical values}\\
 & \mid E\oplus E & \text{Binary Operators}\\
 & \mid E{:}E & \text{List cons}\\
 & \mid E\cdot E & \text{Reference construction}\\
 & \mid \lambda E.E & \text{Lambda values}
\end{array}$$

$$[\![\mathrm{v}]\!]^L_\epsilon \triangleq \mathrm{v} \qquad [\![\mathbf{l}]\!]^L_\epsilon \triangleq L \qquad [\![\mathrm{X}]\!]^L_\epsilon \triangleq \epsilon(\mathrm{X})$$

$$[\![E_1{:}E_2]\!]^L_\epsilon \triangleq [\![E_1]\!]^L_\epsilon{:}Ls \text{ if } [\![E_2]\!]^L_\epsilon = Ls$$

$$[\![E_1\cdot E_2]\!]^L_\epsilon \triangleq l\cdot\mathtt{x} \text{ if } [\![E_1]\!]^L_\epsilon = l\wedge[\![E_2]\!]^L_\epsilon=\mathtt{x}$$

$$[\![E_1\oplus E_2]\!]^L_\epsilon \triangleq v\,\overline{\oplus}\,v' \text{ if } [\![E_1]\!]^L_\epsilon = v\wedge[\![E_2]\!]^L_\epsilon=v'$$

$$[\![\lambda E_1.E_2]\!]^L_\epsilon \triangleq \lambda\mathtt{x}.[\![E_2]\!]^L_\epsilon \text{ if } [\![E_1]\!]^L_\epsilon = \mathtt{x}$$

Assertions include the boolean assertions, structural assertions of separation logic and our new sepish connective, basic assertions for describing cells in a JavaScript heap, expression equality, set and list assertions, and quantification over logical variables.

Assertions.

$$\begin{array}{lll}
P ::= & P\wedge P \mid P\vee P \mid \neg P \mid \mathsf{true} \mid \mathsf{false} & \text{Boolean assertions}\\
 & \mid P*P \mid P\mathrel{-\!\!*}P \mid P\boxast P & \text{Structural assertions}\\
 & \mid (E,E)\mapsto E \mid \oslash & \text{JavaScript heap assertions}\\
 & \mid E=E & \text{Expression equality}\\
 & \mid E\in\textsc{set} & \text{Set membership}\\
 & \mid E\in E & \text{List element}\\
 & \mid \exists\mathrm{X}.\,P \mid \forall\mathrm{X}.\,P & \text{Quantification}
\end{array}$$

Notation: $E \not\circ E \triangleq \neg(E\circ E)$ for $\circ\in\{=,\in\}$

$E_1 \dot{\circ} E_2 \triangleq E_1\circ E_2\wedge\oslash$ for $\circ\in\{=,\neq,\in,\notin\}$.

The logical operators bind in order: $\neg, \boxast, *, \wedge, \vee, \mathrel{-\!\!*}$.

The structural assertions $*$ and $\mathrel{-\!\!*}$ are standard separation logic assertions. The separating conjunction $P*Q$ says that the heap may be split into two disjoint heaps, one satisfying P and the other Q. The right adjoint $P\mathrel{-\!\!*}Q$ says that, whenever the heap is extended by a heap satisfying P, then the resulting heap satisfies Q. It is useful in proving the soundness of some of our layers of abstraction in Section 6, as given in [11]. The sepish connective $P\boxast Q$ is novel. It says that the heap may be split into two heaps, one satisfying P and the other Q, but these two heaps need not be disjoint. They may share zero or more common cells. We shall see in Section 5 that this is particularly useful when reasoning about the emulated variable store. It is possible to define $\mathrel{-\!\!\boxast}$ analogously with $\mathrel{-\!\!*}$, but since this is not useful for JavaScript reasoning we omit it here. Note that $P\wedge Q \Rightarrow P\boxast Q$ and $P*Q\Rightarrow P\boxast Q$, but neither of the reverse implications hold. The assertion $(E_1,E_2)\mapsto E_3$ declares information about a cell, including the information that field E_2 does not occur in object E_1. Assertion $\oslash$ says that the heap is empty. The notation $\textsc{set}$ denotes a literal set, or a named set such as $\mathcal{X}$, the set of JavaScript field names. Note that since there are sets for numbers, strings and locations, we can use set inclusion to assert the type of a particular JavaScript value.

The satisfaction relation $h,L,\epsilon\models P$ is defined below. The cases for the boolean assertions are not given as they are standard.

Satisfaction of assertions: $h,L,\epsilon\models P$**.**

$$\begin{array}{lll}
h,L,\epsilon\models P*Q & \iff & \exists h_1,h_2.\,h\equiv h_1*h_2\wedge\\
 & & (h_1,L,\epsilon\models P)\wedge(h_2,L,\epsilon\models Q)\\
h,L,\epsilon\models P\mathrel{-\!\!*}Q & \iff & \forall h_1.\,(h_1,L,\epsilon\models P)\wedge h\,\#\,h_1\\
 & & \quad\Longrightarrow ((h*h_1),L,\epsilon\models Q)\\
h,L,\epsilon\models P\boxast Q & \iff & \exists h_1,h_2,h_3.\\
\multicolumn{3}{l}{\quad h\equiv h_1*h_2*h_3\wedge(h_1*h_3,L,\epsilon\models P)\wedge(h_2*h_3,L,\epsilon\models Q)}\\
h,L,\epsilon\models(E_1,E_2)\mapsto E_3 & \iff & h\equiv([\![E_1]\!]^L_\epsilon,[\![E_2]\!]^L_\epsilon)\mapsto[\![E_3]\!]^L_\epsilon\\
h,L,\epsilon\models\oslash & \iff & h=\mathtt{emp}\\
h,L,\epsilon\models E_1=E_2 & \iff & [\![E_1]\!]^L_\epsilon=[\![E_2]\!]^L_\epsilon\\
h,L,\epsilon\models E\in\textsc{set} & \iff & [\![E]\!]^L_\epsilon\in\textsc{set}\\
h,L,\epsilon\models E_1\in E_2 & \iff & [\![E_1]\!]^L_\epsilon \text{ is in the list } [\![E_2]\!]^L_\epsilon\\
h,L,\epsilon\models\exists\mathrm{X}.\,P & \iff & \exists\mathrm{v}.\,h,L,[\epsilon|\mathrm{X}\leftarrow\mathrm{v}]\models P\\
h,L,\epsilon\models\forall\mathrm{X}.\,P & \iff & \forall\mathrm{v}.\,h,L,[\epsilon|\mathrm{X}\leftarrow\mathrm{v}]\models P
\end{array}$$

We have given a direct definition of the sepish connective. When logical variables range over heaps, it can be derived: $P\boxast Q\Leftrightarrow\exists\mathrm{R}.\,(\mathrm{R}\mathrel{-\!\!*}P)*(\mathrm{R}\mathrel{-\!\!*}Q)*\mathrm{R}$. It remains to be seen what natural logical properties are satisfied by this connective.

5. Program Reasoning

In the spirit of separation logic, we give small axioms and inference rules which precisely capture the behaviour of all JavaScript expressions, except for the usual approximation for `while` and conservative approximations of function call and `eval`. Reasoning about function call and eval is interesting and complex, and will be the focus of future work as outlined in Section 8. Because our reasoning captures the full complexity of JavaScript semantics, particularly with respect to the emulated variable store and the `with` command, it is possible to prove properties about extremely subtle programs. Unfortunately, proving any program at this level of abstraction involves a level of detail which most programmers would understandably rather avoid. We therefore provide several *layers of abstraction* which make it possible to reason at a natural high level about well-behaved JavaScript programs. We discuss these abstractions in Section 6.

Our fault-avoiding Hoare triples take the form: $\{P\}\mathtt{e}\{Q\}$, which means "if $\mathtt{e}$ is executed in a state satisfying P, then it will not fault, and if it terminates it will do so in a state satisfying Q". The postcondition Q may refer to the special variable $\mathbf{r}$, which is equal to the return value of $\mathtt{e}$.

5.1 Auxiliary Predicates

For our reasoning rules, we require predicates that correspond to functions, such as σ, π and γ, used by the operational semantics in Section 3.3. These predicates present us with two distinct challenges: sharing and precision. To consider sharing, recall the scope function σ from Section 3.1 when searching for a variable `y` in the example emulated store given in Figure 1. Since `y` is not present in the emulated store, σ will check the entire structure before returning `null` to indicate that `y` cannot be found. What is of interest to us is the order in which the cells in the store will be checked. Notice that the cell l will be checked twice, and the cell l_{op} will be checked three times. This is because the cells l and l_{op} are shared between the prototype chain of the object l_2 and the object l_4. In addition, l_{op} is shared by these prototype chains and the prototype chain of l_g. As we shall see below, we can describe these partially shared structures using our sepish connective $\boxast$. To consider precision, recall the function σ when this time searching for a variable `x` either in the emulated store of Figure 1, or in the identical store but for the omission of a single object such that the prototype of l_2 would be l. In each of these stores, the σ function will return the same value – the location l_2. In our program reasoning, our σ predicate (which will correspond to the σ function of the operational semantics) must be more precise as we need to distinguish between these two possible cases. With the challenges of sharing and precision in mind, we first give, then explain, the logical predicates σ, π and γ.

Logical predicates: σ, π, γ**.**

$$
\begin{aligned}
&\sigma([\,],[\,],\_,\mathtt{null}) \triangleq \oslash\\
&\sigma([Ls], L' : Ls', X, L') \triangleq \exists \mathrm{L}.\, \pi(Ls, L', X, \mathrm{L}) * \mathrm{L} \not\doteq \mathtt{null}\\
&\sigma((Ls_1 : Ls_2), L' : Ls', X, L) \triangleq\\
&\qquad \pi(Ls_1, L', X, \mathtt{null}) \boxast \sigma(Ls_2, Ls', X, L)
\end{aligned}
$$

$$
\begin{aligned}
&\pi([\,],\mathtt{null},\_,\mathtt{null}) \triangleq \oslash\\
&\pi([L], L, X, L) \triangleq \exists \mathrm{V}.\, (L, X) \mapsto \mathrm{V} * \mathrm{V} \not\doteq \varnothing\\
&\pi((L' : Ls), L', X, L) \triangleq\\
&\quad \exists \mathrm{N}.\, (L', X) \mapsto \varnothing * (L', @\mathit{proto}) \mapsto \mathrm{N} * \pi(Ls, \mathrm{N}, X, L)
\end{aligned}
$$

$$
\begin{aligned}
&\gamma([\,], \mathit{Val}, \mathit{Val}) \triangleq \mathit{Val} \mathrel{\dot{\notin}} \mathcal{R}\\
&\gamma(Ls, L{\cdot}X, \mathtt{undefined}) \triangleq \pi(Ls, L, X, \mathtt{null}) * L \not\doteq \mathtt{null}\\
&\gamma(Ls, L_1{\cdot}X, \mathit{Val}) \triangleq\\
&\quad \exists \mathrm{L}_2.\, \pi(Ls, L_1, X, \mathrm{L}_2) \boxast (\mathrm{L}_2, X) \mapsto \mathit{Val} * \mathit{Val} \not\doteq \varnothing
\end{aligned}
$$

These predicates closely follow the structure of the functions defined in 3.1, using $\boxast$ to manage the challenge of sharing prototype chains mentioned earlier. The predicate $\sigma(\_, L, \mathtt{x}, l)$ holds only for abstract heaps h such that the function from Section 3.1 gives us $\sigma(\lfloor h \rfloor, L, \mathtt{x}) = l$, meaning that the value of the variable $\mathtt{x}$ in the emulated variable store given by the list L can be found in the object (or a prototype of the object) at address l. However, recall the challenge of precision mentioned earlier. In order to distinguish between all the possible heaps which satisfy the predicate $\sigma(\_, L, \mathtt{x}, l)$, we work with an additional first argument. The predicate $\sigma(Ls, L, \mathtt{x}, l)$ is "precise" in the sense that, for any abstract heap h, it holds for at most one subheap of h. The first argument Ls is a list of lists, which specifies the exact cells which must be visited (and the order in which they must be checked) in order to determine which object in the emulated variable store defines the variable $\mathtt{x}$. For example, recall the heap illustrated in Figure 1. If the prototype of the cell l_2 has address l', then the predicate $\sigma([[l_1],[l_2,l',l]], \mathbf{l}, \mathtt{x}, l_2)$ is satisfied by the abstract heap consisting of the $\mathtt{x}$ and $@\mathit{proto}$ fields of the four objects in the lower left corner of that diagram. Notice that we do not need to visit every object in the variable store in order to discover the location of the variable $\mathtt{x}$. In the spirit of the small axioms of separation logic, our predicate holds for the smallest possible heap in which we can be sure of discovering the variable we are interested in.

The predicates π and γ are similar in that they mirror their operational counterparts, with the addition of one extra argument to make them precise. In the case of π and γ, the first argument Ls is simply a list of addresses, rather than a list of lists of addresses, because each predicate only walks down at most one prototype chain.

The inference rules also require logical predicates corresponding to a number of other auxiliary functions given in Section 3. Below, we define newobj and fun predicates, which assert the existence of a fresh object and function object, and decls that returns the local variables of an expression. In order to reason about function call, we also use a defs predicate which we define in [11].

Auxiliary predicates

$$
\begin{aligned}
&\mathsf{This}(L{\cdot}\_, L) \triangleq (L, @\mathit{this}) \mapsto \varnothing \qquad \text{where } L \neq l_g\\
&\mathsf{This}(L{\cdot}\_, l_g) \triangleq \exists \mathrm{V}.\, (L, @\mathit{this}) \mapsto \mathrm{V} * \mathrm{V} \not\doteq \varnothing
\end{aligned}
$$

$$
\begin{aligned}
&\mathsf{True}(E) \triangleq E \mathrel{\dot{\notin}} \{0, \text{“”}, \mathtt{null}, \mathtt{undefined}\}\\
&\mathsf{False}(E) \triangleq E \mathrel{\dot{\in}} \{0, \text{“”}, \mathtt{null}, \mathtt{undefined}\}\\
&\mathsf{newobj}(L, V_1, \dots, V_n) \triangleq \circledast_{V \in \mathcal{X} \setminus \{V_1 \dots V_n\}} (L, V) \mapsto \varnothing
\end{aligned}
$$

$$
\begin{aligned}
&\mathsf{fun}(F, \mathit{Env}, X, \mathit{Body}, \mathit{Proto}) \triangleq\\
&\qquad (F, @\mathit{scope}) \mapsto \mathit{Env} * (F, @\mathit{body}) \mapsto \lambda X.\mathit{Body} *\\
&\qquad (F, \mathtt{prototype}) \mapsto \mathit{Proto} * (F, @\mathit{proto}) \mapsto l_{fp}
\end{aligned}
$$

$$
\mathsf{decls}(X, L, \mathtt{e}) \triangleq \mathtt{x1}, \dots, \mathtt{xn} \quad \text{where } (L, \mathtt{xi}) \in \mathsf{dom}(\mathsf{defs}(X, L, \mathtt{e}))
$$

5.2 Inference Rules

We define below some inference rules $\{P\}\mathtt{e}\{Q\}$ for reasoning about JavaScript expressions. The full list can be found in [11].

Inference rules: $\{P\}\mathtt{e}\{Q\}$**.**

(Definition)

$$
\frac{\{P\}\mathtt{e}\{Q\} \qquad \mathbf{r} \notin \mathsf{fv}(Q)}{\{P\}\mathtt{var\ e}\{Q * \mathbf{r} \doteq \mathtt{undefined}\}}
$$

(Value)

$$
\{\oslash\}\mathtt{v}\{\mathbf{r} \doteq \mathtt{v}\}
$$

(Variable)

$$
\frac{P = \sigma(Ls_1, \mathbf{l}, \mathtt{x}, L) \boxast \gamma(Ls_2, L{\cdot}\mathtt{x}, V)}{\{P\}\mathtt{x}\{P * \mathbf{r} \doteq L{\cdot}\mathtt{x}\}}
$$

(Variable Null)

$$
\frac{P = \sigma(Ls, \mathbf{l}, \mathtt{x}, \mathtt{null})}{\{P\}\mathtt{x}\{P * \mathbf{r} \doteq \mathtt{null}{\cdot}\mathtt{x}\}}
$$

(Member Access)

$$
\frac{\{P\}\mathtt{e}\{Q * \mathbf{r} \doteq V\} \qquad Q = R * \gamma(Ls, V, L) * L \not\doteq \mathtt{null}}{\{P\}\mathtt{e.x}\{Q * \mathbf{r} \doteq L{\cdot}\mathtt{x}\}}
$$

(Computed Access)

$$
\frac{\begin{array}{ll}\{P\}\mathtt{e1}\{R * \mathbf{r} \doteq V_1\} & R = S_1 * \gamma(Ls_1, V_1, L) * L \not\doteq \mathtt{null}\\ \{R\}\mathtt{e2}\{Q * X \mathrel{\dot{\in}} \mathcal{X}^{\mathrm{U}} * \mathbf{r} \doteq V_2\} & Q = S_2 * \gamma(Ls_2, V_2, X)\end{array}}{\{P\}\mathtt{e1[e2]}\{Q * \mathbf{r} \doteq L{\cdot}X\}}
$$

(Object)

$$
\frac{\begin{array}{l}\forall i \in 1..n.\ \big(P_i = R_i * \gamma(Ls_i, Y_i, X_i) \quad \{P_{i-1}\}\mathtt{ei}\{P_i * \mathbf{r} \doteq Y_i\}\big)\\ Q = \left(P_n * \exists \mathrm{L}. \left(\begin{array}{l}\mathsf{newobj}(\mathrm{L}, @\mathit{proto}, \mathtt{x1}, \dots, \mathtt{xn}) *\\ (\mathrm{L}, \mathtt{x1}) \mapsto X_1 * \dots * (\mathrm{L}, \mathtt{xn}) \mapsto X_n *\\ (\mathrm{L}, @\mathit{proto}) \mapsto l_{op} * \mathbf{r} \doteq \mathrm{L}\end{array}\right)\right)\\ \mathtt{x1} \neq \dots \neq \mathtt{xn} \qquad \mathbf{r} \notin \mathsf{fv}(P_n)\end{array}}{\{P_0\}\{\mathtt{x1{:}e1}, \dots, \mathtt{xn{:}en}\}\{Q\}}
$$

(Binary Operators)

$$
\frac{\begin{array}{ll}\{P\}\mathtt{e1}\{R * \mathbf{r} \doteq V_1\} & R = S_1 * \gamma(Ls_1, V_1, V_3)\\ \{R\}\mathtt{e2}\{Q * \mathbf{r} \doteq V_2\} & Q = S_2 * \gamma(Ls_2, V_2, V_4)\\ V = V_3 \mathbin{\overline{\oplus}} V_4 & \end{array}}{\{P\}\mathtt{e1} \oplus \mathtt{e2}\{Q * \mathbf{r} \doteq V\}}
$$

(Assign Global)

$$
\frac{\begin{array}{ll}\{P\}\mathtt{e1}\{R * \mathbf{r} \doteq \mathtt{null}{\cdot}X\} & \\ \{R\}\mathtt{e2}\{Q * (l_g, X) \mapsto \varnothing * \mathbf{r} \doteq V_1\} & Q = S * \gamma(Ls, V_1, V_2)\end{array}}{\{P\}\mathtt{e1\ =\ e2}\{Q * (l_g, X) \mapsto V_2 * \mathbf{r} \doteq V_2\}}
$$

(Assign Local)

$$
\frac{\begin{array}{ll}\{P\}\mathtt{e1}\{R * \mathbf{r} \doteq L{\cdot}X\} & \\ \{R\}\mathtt{e2}\{Q * (L, X) \mapsto V_3 * \mathbf{r} \doteq V_1\} & Q = S * \gamma(Ls, V_1, V_2)\end{array}}{\{P\}\mathtt{e1\ =\ e2}\{Q * (L, X) \mapsto V_2 * \mathbf{r} \doteq V_2\}}
$$

(Function)

$$
\frac{Q = \left(\begin{array}{l}\exists \mathrm{L}_1, \mathrm{L}_2.\, \mathsf{newobj}(\mathrm{L}_1, @\mathit{proto}) * (\mathrm{L}_1, @\mathit{proto}) \mapsto l_{op} *\\ \mathsf{newobj}(\mathrm{L}_2, @\mathit{proto}, \mathtt{prototype}, @\mathit{scope}, @\mathit{body}) *\\ \mathsf{fun}(\mathrm{L}_2, \mathbf{l}, \mathtt{x}, \mathtt{e}, \mathrm{L}_1) * \mathbf{r} \doteq \mathrm{L}_2\end{array}\right)}{\{\oslash\}\mathtt{function\ (x)\{e\}}\{Q\}}
$$

(While)

$$
\frac{\begin{array}{ll}\{P\}\mathtt{e1}\{S * \mathbf{r} \doteq V_1\} & S = R * \gamma(Ls, V_1, V_2)\\ \{S * \mathsf{True}(V_2)\}\mathtt{e2}\{P\} & \\ Q = S * \mathsf{False}(V_2) * \mathbf{r} \doteq \mathtt{undefined} & \mathbf{r} \notin \mathsf{fv}(R)\end{array}}{\{P\}\mathtt{while(e1)\{e2\}}\{Q\}}
$$

(With)

$$
\frac{\begin{array}{ll}\{P * \mathbf{l} \doteq L\}\mathtt{e1}\{S * \mathbf{l} \doteq L * \mathbf{r} \doteq V_1\} & S = R * \gamma(Ls, V_1, L_1)\\ \{S * \mathbf{l} \doteq L_1{:}L\}\mathtt{e2}\{Q * \mathbf{l} \doteq L_1{:}L\} & \mathbf{l} \notin \mathsf{fv}(P, Q, R)\end{array}}{\{P * \mathbf{l} \doteq L\}\mathtt{with(e1)\{e2\}}\{Q * \mathbf{l} \doteq L\}}
$$

(Function Call)

$$\frac{\begin{array}{l}\{P\}\texttt{e1}\{R_1 * \mathbf{r} \doteq F_1\} \\ R_1 = \left(\begin{array}{l} S_1 \boxast \mathsf{This}(F_1, T) \boxast \gamma(Ls_1, F_1, F_2) * \\ (F_2, @body) \mapsto \lambda X.\mathsf{e3} * (F_2, @scope) \mapsto Ls_2 \end{array}\right) \\ \{R_1\}\texttt{e2}\{R_2 * \mathbf{l} \doteq Ls_3 * \mathbf{r} \doteq V_1\} \quad R_2 = S_2 * \gamma(Ls_4, V_1, V_2) \\ R_3 = \left(\begin{array}{l} R_2 * \exists \mathrm{L}.\, \mathbf{l} \doteq \mathrm{L}{:}Ls_2 * (\mathrm{L}, X) \mapsto V_2 * \\ (\mathrm{L}, @this) \mapsto T * \\ (\mathrm{L}, @proto) \mapsto \texttt{null} * \mathsf{defs}(X, \mathrm{L}, \mathsf{e3}) * \\ \mathsf{newobj}(\mathrm{L}, @proto, @this, X, \mathsf{decls}(X, \mathrm{L}, \mathsf{e3})) \end{array}\right) \\ \{R_3\}\texttt{e3}\{\exists \mathrm{L}.\, Q * \mathbf{l} \doteq \mathrm{L}{:}Ls_2\} \qquad \mathbf{l} \notin \mathsf{fv}(Q, R_2) \end{array}}{\{P\}\texttt{e1(e2)}\{\exists \mathrm{L}.\, Q * \mathbf{l} \doteq Ls_3\}}$$

(Frame)

$$\frac{\{P\}\mathsf{e}\{Q\}}{\{P * R\}\mathsf{e}\{Q * R\}}$$

(Consequence)

$$\frac{\{P_1\}\mathsf{e}\{Q_1\} \quad P \implies P_1 \quad Q_1 \implies Q}{\{P\}\mathsf{e}\{Q\}}$$

(Elimination)

$$\frac{\{P\}\mathsf{e}\{Q\}}{\{\exists \mathrm{X}.\, P\}\mathsf{e}\{\exists \mathrm{X}.\, Q\}}$$

(Disjunction)

$$\frac{\{P_1\}\mathsf{e}\{Q_1\} \qquad \{P_2\}\mathsf{e}\{Q_2\}}{\{P_1 \vee P_2\}\mathsf{e}\{Q_1 \vee Q_2\}}$$

Although most of the rules correspond closely to their operational counterparts, some rules deserve further comment. Rule (Definition) shows the use of the reserved variable $\mathbf{r}$ to record the result of an expression. Rule (Variable) shows the use of $\boxast$ to express the overlapping footprints of predicates σ and γ. Rule (Assign Global) shows the use of $\varnothing$ to assert that certain known memory cells are available for allocation. Rule (Function Call) describes JavaScript's dynamic functions but does not support higher order reasoning. Rule (Frame) does not have the usual side condition because JavaScript stores all its program variables on the heap, so any variable modified by an expression is necessarily contained entirely within the footprint of the expression. Rules (Consequence), (Elimination) and (Disjunction) are standard.

5.3 Soundness

We show that our inference rules are sound with respect to the operational semantics of Section 3.3. When proving the soundness of any system involving the frame rule, it is usual to first show the locality of the programming language, and use that property to show the soundness of the frame rule [20]. Unfortunately, many JavaScript statements are not local according to this standard definition. We therefore use the recently introduced notion of weak locality from Smith's thesis [30].

Definition 2 (Soundness of a Hoare triple)**.** A Hoare triple $\{P\}\mathsf{e}\{Q\}$ is sound if, for all abstract heaps h, scope chains L and environments ϵ, it satisfies the following two properties:

$$\textit{Fault Avoidance}: h, L, (\epsilon \setminus \mathbf{r}) \models P \implies \lfloor h \rfloor, L, \mathsf{e} \not\longrightarrow \mathsf{fault}$$

$$\begin{array}{l}\textit{Safety}: \forall H, r.\ h, L, (\epsilon \setminus \mathbf{r}) \models P \wedge \lfloor h \rfloor, L, \mathsf{e} \longrightarrow H, r \\ \quad \implies \ \exists h'.\, H = \lfloor h' \rfloor \wedge h', L, [\epsilon | \mathbf{r} \leftarrow r] \models Q.\end{array}$$

Theorem 3 (Soundness)**.** *All derivable Hoare triples* $\{P\}\mathsf{e}\{Q\}$ *are sound according to Definition 2.*

The proof (reported in [11]) involves showing that the predicates used by the Hoare rules correspond to the auxiliary functions used by the semantics, showing that all JavaScript expressions are weakly local with respect to their preconditions, and finally showing that all our inference rules are sound.

6. Layers of Abstraction

As mentioned in Section 5, using the rules given so far, reasoning about JavaScript program involves detail that most programmers need never consider. Most of the time, programmers will work at a higher level of abstraction, for example, treating the emulated variable store as if it were a regular variable store. This is a good practice as long as the abstraction holds. However, if the program happens to come across a corner case that breaks the abstraction, its resulting behaviour can appear almost inexplicable. This may be a particular problem when writing library code, since the programmer has no control over the programming discipline of the client who uses the library. In this section, we introduce several explicit abstraction formalisms. Many alternatives are possible, but those presented here are enough to demonstrate the concept and reason about some interesting programs. We are able to use these formalisms to reason at a comfortable, high-level way about many well-behaved programs. Crucially, we know exactly what the boundaries of these layers of abstraction are, so we can ensure that our programs remain safely within the abstraction. If we wish, we can even choose to temporarily break an abstraction, execute some particularly subtle low-level code, re-establish the abstraction and then continue to work at the high level.

6.1 Layer 1: Exploring the Scope List

Central to reasoning about JavaScript variables are the σ and π predicates. The first abstraction layer consists of alternative versions of these predicates which make reasoning about certain common cases simpler. The $\overline{\sigma}$ predicate unrolls from the end (the most global end) of the scope rather than from the beginning (the local end) which makes modifying a variable easier to specify. It makes use of $\not\sigma$ which says that a variable does not exist in a particular partial scope. The $\not\sigma^{l_g}$ predicate does the same, but excludes l_g from its specification, in order to make reasoning about global variable instantiation simpler. We give several useful triples about variable assignment. In [11], we prove these triples by using $\overline{\sigma}$ and showing the equivalence of σ and $\overline{\sigma}$.

Layer 1 Predicates.

$$\begin{array}{l}
\overline{\sigma}(Ls, Ls', Var, \texttt{null}) \triangleq \not\sigma(Ls, Ls', Var, \texttt{null}) \\
\overline{\sigma}(Ls_1 +\!\!+ (Ls_2 : []), Ls', Var, L) \triangleq \\
\quad \not\sigma(Ls_1, Ls', Var, L) * \exists \mathrm{L2}.\, \pi(Ls_2, L, Var, L2) * \mathrm{L2} \mathrel{\dot{\neq}} \texttt{null} \\[1ex]
\not\sigma([], L' : Ls', \_, L') \triangleq \oslash \\
\not\sigma(Ls_2 : Ls, L' : Ls', Var, End) \triangleq \\
\qquad \pi(Ls_2, L', Var, \texttt{null}) \boxast \not\sigma(Ls, Ls', Var, End) \\[1ex]
\not\sigma^{l_g}([], L' : Ls', \_, L') \triangleq \oslash \\
\not\sigma^{l_g}(Ls_2 : Ls, L' : Ls', Var, End) \triangleq \\
\qquad \pi^{l_g}(Ls_2, L', Var, \texttt{null}) \boxast \not\sigma^{l_g}(Ls, Ls', Var, End) \\[1ex]
\pi^{l_g}([], \texttt{null}, \_, \texttt{null}) \triangleq \oslash \\
\pi^{l_g}([], l_g, \_, \texttt{null}) \triangleq \oslash \\
\pi^{l_g}([L'], L', Var, L') \triangleq \exists \mathrm{V}.\, (L', Var) \mapsto \mathrm{V} * \mathrm{V} \mathrel{\dot{\neq}} \varnothing * L' \mathrel{\dot{\neq}} l_g \\
\pi^{l_g}((L' : Ls), L', Var, L) \triangleq \exists \mathrm{N}.\, (L', Var) \mapsto \varnothing * \\
\qquad (L', @proto) \mapsto \mathrm{N} * \pi^{l_g}(Ls, \mathrm{N}, Var, L) * L' \mathrel{\dot{\neq}} l_g
\end{array}$$

These predicates give us much more flexibility to reason about JavaScript variables found in various places in the emulated variable store. Even at this quite low level, it is possible to prove general specifications about programs with many corner cases. A good example of this sort of reasoning is simple assignment statements. We prove the following general triples about simple assignments. The first three triples deal with the assignment of a constant to a variable, in the cases of variable initialisation, variable override, and variable overwrite respectively. The fourth triple deals with assigning the value of one variable to another. All four are proved sound in [11].

Simple assignments.

$$\frac{\begin{array}{l} P = \sigma(L_1 +\!\!+ ((l_g{:}L_2){:}L_3), \mathbf{l}, \texttt{x}, \texttt{null}) \\ Q = \left(\begin{array}{c} \exists \mathrm{L}_1', \mathrm{L}_3', \mathrm{Ls}', \mathrm{G}.\quad \not\sigma^{l_g}(\mathrm{L}_1', \mathbf{l}, \texttt{x}, l_g) \boxast \\ \pi(L_2, \mathrm{G}, \texttt{x}, \texttt{null}) \boxast \not\sigma^{l_g}(\mathrm{L}_3', \mathrm{Ls}', \texttt{x}, \texttt{null}) * (l_g, \texttt{x}) \mapsto \texttt{v} * \\ (l_g, @proto) \mapsto \mathrm{G} * \mathbf{l} \doteq \_ +\!\!+ (l_g{:}\mathrm{Ls}') * \mathbf{r} \doteq \texttt{v} \end{array}\right) \end{array}}{\{P\}\texttt{x = v}\{Q\}}$$

$$\frac{\begin{array}{l}P = \sigma(L_1 +\!\!+ [L{:}L_2]), \mathbf{l}, \mathtt{x}, L) \boxast (L, \mathtt{x}) \mapsto \varnothing \boxast \gamma(L{:}L_2, L\cdot\mathtt{x}, V) \\ Q = \left(\begin{array}{l}\exists \mathrm{L}'.\ \not\sigma(L_1, \mathbf{l}, \mathtt{x}, L) * (L, \mathtt{x}) \mapsto \mathtt{v} * (L, @\mathit{proto}) \mapsto \mathit{Pr} * \\ \pi(L_2, \mathit{Pr}, \mathtt{x}, \mathrm{L}') \boxast (\mathrm{L}', \mathtt{x}) \mapsto V * \mathbf{r} \doteq \mathtt{v}\end{array}\right)\end{array}}{\{P\}\mathtt{x\ =\ v}\{Q\}}$$

$$\frac{\begin{array}{l}P = \sigma(L_1 +\!\!+ [[L]], \mathbf{l}, \mathtt{x}, L) \boxast (L, \mathtt{x}) \mapsto V * V \dot{\neq} \varnothing \\ Q = \not\sigma(L_1, \mathbf{l}, \mathtt{x}, L) * (L, \mathtt{x}) \mapsto \mathtt{v} * \mathbf{r} \doteq \mathtt{v}\end{array}}{\{P\}\mathtt{x\ =\ v}\{Q\}}$$

$$\frac{\begin{array}{l}P = \left(\begin{array}{l}\sigma(Ls_1, \mathbf{l}, \mathtt{y}, L_y) \boxast \gamma(Ls_2, L_y\cdot\mathtt{y}, V_y) \boxast \\ \sigma(L_1 +\!\!+ [[L]], \mathbf{l}, \mathtt{x}, L) \boxast (L, \mathtt{x}) \mapsto V * V \dot{\neq} \varnothing\end{array}\right) \\ Q = \sigma(Ls_1, \mathbf{l}, \mathtt{y}, L_y) \boxast \not\sigma(L_1, \mathbf{l}, \mathtt{x}, L) * (L, \mathtt{x}) \mapsto V_y * \mathbf{r} \doteq V_y\end{array}}{\{P\}\mathtt{x\ =\ y}\{Q\}}$$

Compared to the (Assign Global) and (Assign Local) inference rules, these triples provide an explicit account of the footprint of the assignment, and more clearly describe the destructive effects of assignment. Yet, they are still quite complex and are difficult to compose. It would be useful to ignore some information about the exact structure of the variable store, while retaining the information about the mappings of variable names to values. To do this, we introduce a new store predicate.

6.2 Layer 2: a Simple Abstract Variable Store

The predicates below provide a convenient abstraction for an emulated variable store.

The store **Predicate.**

$$\begin{array}{l}\mathsf{store}_L(X_1 \ldots X_n | X'_1{:}V_1 \ldots X'_m{:}V_m) \triangleq \\ \quad \exists Ls_1 \ldots, Ls_n, Ls'_1, \ldots, Ls'_m, Ls''_1, \ldots, Ls''_m.\ \ \mathsf{thischain}(L) \boxast \\ \quad \boxast_{i\in 1..n}\ \not\sigma(Ls_i, L, X_i, \mathtt{null}) \quad \boxast (l_g, X_i) \mapsto \varnothing \\ \quad \boxast_{j\in 1..m}\ \sigma(Ls'_j, L, X'_j, L_j) \quad \boxast_{k\in 1..m}\ \gamma(Ls''_k, L_k\cdot X'_k, V_k)\end{array}$$

$$\begin{array}{lcl}\mathsf{thischain}([\,]) & \triangleq & \oslash \\ \mathsf{thischain}(L : Ls') & \triangleq & (L, @\mathit{this}) \mapsto \_ * \mathsf{thischain}(Ls')\end{array}$$

The assertion $\mathsf{store}_{\mathrm{LS}}(\mathsf{a}, \mathsf{b} | \mathtt{x} : 1, \mathtt{y} : 2)$ describes a heap emulating a variable store given by scope chain LS, in which the variables a and b are *not* present and the variables x and y take the values 1 and 2 respectively. The body of the predicate uses the $\not\sigma$ predicate to assert the absence of a and b, and the σ and γ predicates to assert the presence and values of x and y. This information about the exact structure of the emulated store and the locations of x and y is hidden from the programmer, since at this level of abstraction it should be of no concern. The thischain part of the body of the predicate ensures that the store predicate always encompasses enough resource to call any function that is stored in the emulated variable store. The variables a and b can be re-ordered, as can the variables x and y. To facilitate program reasoning at this level of abstraction we provide several inference rules, all of which are derived (using previous levels of abstraction) in [11]. We start by giving rules for initialising, overwriting and overriding a variable with a constant and with another variable.

Writing to a store**.**

Let $Q_1 = \mathsf{store}_{\mathbf{l}}(\mathrm{X}_1 \ldots \mathrm{X}_n | \mathtt{x}{:}v, \mathrm{X}'_1{:}\mathrm{V}_1 \ldots \mathrm{X}'_m{:}\mathrm{V}_m)$.
Let $Q_2 = \mathsf{store}_{\mathbf{l}}(\mathrm{X}_1 \ldots \mathrm{X}_n | \mathtt{x}{:}\mathrm{V}, \mathtt{y}{:}\mathrm{V}, \mathrm{X}'_1{:}\mathrm{V}_1 \ldots \mathrm{X}'_m{:}\mathrm{V}_m)$.

$$\frac{\begin{array}{l}\mathtt{x} \neq \mathrm{X}_1 \neq \cdots \neq \mathrm{X}_n \neq \mathrm{X}'_1 \neq \cdots \neq \mathrm{X}'_m \\ P = \mathsf{store}_{\mathbf{l}}(\mathtt{x}, \mathrm{X}_1 \ldots \mathrm{X}_n | \mathrm{X}'_1{:}\mathrm{V}_1 \ldots \mathrm{X}'_m{:}\mathrm{V}_m)\end{array}}{\{P\}\mathtt{x\ =\ v}\{Q_1 * \mathrm{true} * \mathbf{r} \doteq v\}}$$

$$\frac{\begin{array}{l}\mathtt{x} \neq \mathrm{X}_1 \neq \cdots \neq \mathrm{X}_n \neq \mathrm{X}'_1 \neq \cdots \neq \mathrm{X}'_m \\ P = \mathsf{store}_{\mathbf{l}}(\mathrm{X}_1 \ldots \mathrm{X}_n | \mathtt{x}{:}\mathrm{V}, \mathrm{X}'_1{:}\mathrm{V}_1 \ldots \mathrm{X}'_m{:}\mathrm{V}_m)\end{array}}{\{P\}\mathtt{x\ =\ v}\{Q_1 * \mathrm{true} * \mathbf{r} \doteq v\}}$$

$$\frac{\begin{array}{l}\mathtt{x} \neq \mathtt{y} \neq \mathrm{X}_1 \neq \cdots \neq \mathrm{X}_n \neq \mathrm{X}'_1 \neq \cdots \neq \mathrm{X}'_m \\ P = \mathsf{store}_{\mathbf{l}}(\mathtt{x}, \mathrm{X}_1 \ldots \mathrm{X}_n | \mathtt{y}{:}\mathrm{V}, \mathrm{X}'_1{:}\mathrm{V}_1 \ldots \mathrm{X}'_m{:}\mathrm{V}_m)\end{array}}{\{P\}\mathtt{x\ =\ y}\{Q_2 * \mathrm{true} * \mathbf{r} \doteq \mathrm{V}\}}$$

$$\frac{\begin{array}{l}\mathtt{x} \neq \mathtt{y} \neq \mathrm{X}_1 \neq \cdots \neq \mathrm{X}_n \neq \mathrm{X}'_1 \neq \cdots \neq \mathrm{X}'_m \\ P = \mathsf{store}_{\mathbf{l}}(\mathrm{X}_1 \ldots \mathrm{X}_n | \mathtt{x}{:}\mathrm{V}', \mathtt{y}{:}\mathrm{V}, \mathrm{X}'_1{:}\mathrm{V}_1 \ldots \mathrm{X}'_m{:}\mathrm{V}_m)\end{array}}{\{P\}\mathtt{x\ =\ y}\{Q_2 * \mathrm{true} * \mathbf{r} \doteq \mathrm{V}\}}$$

Notice that each of these rules has a postcondition which includes true. This is because initialising or overriding a variable may render a portion of the emulated variable store superfluous. For example, consider the overriding assignment `x = 1` in the store given in Figure 1. In this case the cell $(l, \mathtt{x})$ has become surplus to requirements. In separation logic it is not sound to simply forget about these cells, so we hide them in the general assertion true.

One limitation of this level of abstraction is that the abstraction only covers a static (and unknown) scope chain. If we call a function which adds a new cell to the scope chain, then the rules above are insufficient to reason about our program. The following rules allow us to reason at this level of abstraction about a program which alters a global variable from within a new local scope frame.

Writing to a store **from a deeper scope.**

Let $Q = \mathsf{store}_{\mathrm{LS}}(\ \mathrm{X}_1, \ldots, \mathrm{X}_n | \mathtt{x} : \mathrm{V}', \mathrm{X}'_1 : \mathrm{V}'_1, \ldots, \mathrm{X}'_m : \mathrm{V}'_m\ )$ and
$S = (\mathrm{L}, @\mathit{proto}) \mapsto \mathtt{null} * (\mathrm{L}, \mathtt{x}) \mapsto \varnothing * (\mathrm{L}, \mathtt{y}) \mapsto \mathrm{V}' * \mathbf{l} \doteq \mathrm{L} : \mathrm{LS}$.

$$\frac{\begin{array}{l}\mathtt{x} \neq \mathtt{y} \neq \mathrm{X}_1 \neq \cdots \neq \mathrm{X}_n \neq \mathrm{X}'_1 \neq \cdots \neq \mathrm{X}'_m \\ P_1 = \mathsf{store}_{\mathrm{LS}}(\ \mathtt{x}, \mathrm{X}_1, \ldots, \mathrm{X}_n | \mathrm{X}'_1 : \mathrm{V}'_1, \ldots, \mathrm{X}'_m : \mathrm{V}'_m\ )\end{array}}{\{P_1 * S\}\mathtt{x=y}\{Q * S * \mathrm{true}\}}$$

$$\frac{\begin{array}{l}\mathtt{x} \neq \mathtt{y} \neq \mathrm{X}_1 \neq \cdots \neq \mathrm{X}_n \neq \mathrm{X}'_1 \neq \cdots \neq \mathrm{X}'_m \\ P_2 = \mathsf{store}_{\mathrm{LS}}(\ \mathrm{X}_1, \ldots, \mathrm{X}_n | \mathtt{x} : \mathrm{V}, \mathrm{X}'_1 : \mathrm{V}'_1, \ldots, \mathrm{X}'_m : \mathrm{V}'_m\ )\end{array}}{\{P_2 * S\}\mathtt{x=y}\{Q * S * \mathrm{true}\}}$$

Finally, we provide two rules for a more general case of store interaction. In these cases, the value which is to be written to the variable is the result of computing some arbitrary expression. These specifications are therefore necessarily more complicated, since they must incorporate some features of sequential composition. We insist that whatever the expression does, it must not alter the variable store in a way that changes the visible values of the variables.

Destructive store **Initialisation.**

$$\frac{\begin{array}{l}\mathtt{x} \neq \mathrm{X}_1 \neq \cdots \neq \mathrm{X}_n \neq \mathrm{X}'_1 \neq \cdots \neq \mathrm{X}'_m \\ R = \mathsf{store}_{\mathbf{l}}(\mathtt{x}, \mathrm{X}_1 \ldots \mathrm{X}_n | \mathrm{X}'_1{:}\mathrm{V}_1 \ldots \mathrm{X}'_m{:}\mathrm{V}_m) \\ \{R * P\}\mathsf{e}\{R \boxast \gamma(LS, \mathrm{V}', \mathrm{V}) * Q * \mathbf{r} \doteq \mathrm{V}'\} \qquad \mathbf{r} \notin \mathsf{fv}(Q) \\ S = \left(\begin{array}{l}\mathsf{store}_{\mathbf{l}}(\ \mathrm{X}_1 \ldots \mathrm{X}_n | \mathtt{x}{:}\mathrm{V}, \mathrm{X}'_1{:}\mathrm{V}_1 \ldots \mathrm{X}'_m{:}\mathrm{V}_m\ ) \\ \boxast \gamma(LS, \mathrm{V}', \mathrm{V}) * Q * \mathrm{true} * \mathbf{r} \doteq \mathrm{V}\end{array}\right)\end{array}}{\{R * P\}\mathtt{x\ =\ }\mathsf{e}\{S\}}$$

$$\frac{\begin{array}{l}\mathtt{x} \neq \mathrm{X}_1 \neq \cdots \neq \mathrm{X}_n \neq \mathrm{X}'_1 \neq \cdots \neq \mathrm{X}'_m \\ R = \mathsf{store}_{L:SLS}(\mathtt{x}, \mathrm{X}_1 \ldots \mathrm{X}_n | \mathrm{X}'_1{:}\mathrm{V}_1 \ldots \mathrm{X}'_m{:}\mathrm{V}_m) \boxast \not\sigma(\mathit{Nsls}, \mathbf{l}, \mathtt{x}, L) \\ \{R\}\mathsf{e}\{R \boxast \gamma(LS, \mathrm{V}', \mathrm{V}) * Q * \mathbf{r} \doteq \mathrm{V}'\} \qquad \mathbf{r} \notin \mathsf{fv}(Q) \\ S = \left(\begin{array}{l}\mathsf{store}_{L:SLS}(\ \mathrm{X}_1 \ldots \mathrm{X}_n | \mathtt{x}{:}\mathrm{V}, \mathrm{X}'_1{:}\mathrm{V}_1 \ldots \mathrm{X}'_m{:}\mathrm{V}_m\ ) \boxast \\ \not\sigma(\mathit{Nsls}, \mathbf{l}, \mathtt{x}, L) \boxast \gamma(LS, \mathrm{V}', \mathrm{V}) * Q * \mathrm{true} * \mathbf{r} \doteq \mathrm{V}\end{array}\right)\end{array}}{\{R * P\}\mathtt{x\ =\ }\mathsf{e}\{S\}}$$

It may seem surprising that we only provide derived rules for destructive variable initialisation, and not for destructive variable *update*. This is because such an update rule would be unsound. The destructive expression might have the side effect of overriding the variable we wish to update. This serves to further demonstrate the need for the low-level reasoning introduced earlier in this paper. We can use high-level abstractions such as the store predicate where they are sound, but if we wish to reason about programs with side-effecting expressions, we will sometimes be forced to reason at a lower level.

The scope of a variable. The store abstraction gives us the tools we need to easily reason about programs with large numbers of variables. For example, consider the program from Section 2:

$$
\begin{array}{l}
\{\ \ \mathsf{store}_{\mathbf{l}}(\texttt{x},\texttt{y},\texttt{z},\texttt{f},\texttt{v}|)\ \ \}\\
\texttt{x = null;y = null;z = null;}\\
\{\ \ \mathsf{store}_{\mathbf{l}}(\texttt{f},\texttt{v}|\texttt{x}:\texttt{null},\texttt{y}:\texttt{null},\texttt{z}:\texttt{null}) * \mathrm{true}\ \ \}\\
\texttt{f = function(w)}\{\texttt{x=v ; v=4 ; var v ; y=v}\}\texttt{;}\\
\left\{\begin{array}{l}
\exists \mathrm{L}.\,\mathsf{store}_{\mathbf{l}}(\texttt{v}|\texttt{x}:\texttt{null},\texttt{y}:\texttt{null},\texttt{z}:\texttt{null},\texttt{f}:\mathrm{L}) *\\
(\mathrm{L},@\mathit{body}) \mapsto \lambda\texttt{w}.\{\dots\} *\\
(\mathrm{L},@\mathit{scope}) \mapsto \mathbf{l} * \mathrm{true}
\end{array}\right\}\\
\texttt{v = 5;}\\
\left\{\begin{array}{l}
\mathsf{store}_{\mathbf{l}}(|\texttt{x}:\texttt{null},\texttt{y}:\texttt{null},\texttt{z}:\texttt{null},\texttt{f}:\mathrm{L},\texttt{v}:5) *\\
(\mathrm{L},@\mathit{body}) \mapsto \lambda\texttt{w}.\{\dots\} *\\
(\mathrm{L},@\mathit{scope}) \mapsto \mathbf{l} * \mathrm{true}
\end{array}\right\}\\
\texttt{f(null);}\\
\left\{\begin{array}{l}
\exists \mathrm{L}'.\,\mathsf{store}_{\mathbf{l}}(|\texttt{x}:\texttt{undefined},\texttt{y}:4,\texttt{z}:\texttt{null},\texttt{f}:\mathrm{L},\texttt{v}:5) *\\
\mathsf{newobj}(\mathrm{L}',@\mathit{proto},@\mathit{this},\texttt{w},\texttt{v}) * (\mathrm{L}',@\mathit{proto}) \mapsto \texttt{null} *\\
(\mathrm{L}',\texttt{w}) \mapsto \texttt{null} * (\mathrm{L}',\texttt{v}) \mapsto 4 * \mathrm{true}
\end{array}\right\}\\
\text{[Frame]}\\
\{\ \ \mathsf{store}_{\mathbf{l}}(|\texttt{x}:\texttt{undefined},\texttt{y}:4,\texttt{z}:\texttt{null},\texttt{f}:\mathrm{L},\texttt{v}:5)\ \ \}\\
\texttt{z = v;}\\
\{\ \ \mathsf{store}_{\mathbf{l}}(|\texttt{x}:\texttt{undefined},\texttt{y}:4,\texttt{z}:5,\texttt{f}:\mathrm{L},\texttt{v}:5) * \mathrm{true}\ \ \}\\
\text{[Frame]}\\
\left\{\begin{array}{l}
\mathsf{store}_{\mathbf{l}}(|\texttt{x}:\texttt{undefined},\texttt{y}:4,\texttt{z}:5,\texttt{f}:\mathrm{L},\texttt{v}:5) *\\
\mathsf{newobj}(\mathrm{L}',@\mathit{proto},@\mathit{this},\texttt{w},\texttt{v}) * (\mathrm{L}',@\mathit{proto}) \mapsto \texttt{null} *\\
(\mathrm{L}',\texttt{w}) \mapsto \texttt{null} * (\mathrm{L}',\texttt{v}) \mapsto 4 * \mathrm{true}
\end{array}\right\}\\
\text{[Cons/Var Elim]}\\
\{\ \ \exists \mathrm{L}.\,\mathsf{store}_{\mathbf{l}}(|\texttt{x}:\texttt{undefined},\texttt{y}:4,\texttt{z}:5,\texttt{f}:\mathrm{L},\texttt{v}:5) * \mathrm{true}\ \ \}
\end{array}
$$

$$
\begin{array}{l}
\left\{\begin{array}{l}
\exists \mathrm{L}',\mathrm{LS}.\,\mathbf{l} \doteq \mathrm{L}' : \mathrm{LS} *\\
\mathsf{store}_{\mathrm{LS}}(|\texttt{x}:\texttt{null},\texttt{y}:\texttt{null},\texttt{z}:\texttt{null},\texttt{f}:\mathrm{L},\texttt{v}:5) *\\
(\mathrm{L},@\mathit{body}) \mapsto \lambda\texttt{w}.\{\dots\} * (\mathrm{L},@\mathit{scope}) \mapsto \mathrm{LS} *\\
\mathsf{newobj}(\mathrm{L}',@\mathit{proto},@\mathit{this},\texttt{w},\texttt{v}) * (\mathrm{L}',@\mathit{proto}) \mapsto \texttt{null} *\\
(\mathrm{L}',\texttt{w}) \mapsto \texttt{null} * (\mathrm{L}',\texttt{v}) \mapsto \texttt{undefined} * (\mathrm{L}',@\mathit{this}) \mapsto \_ * \mathrm{true}
\end{array}\right\}\\
\texttt{x=v;v=4;var v;y=v;}\\
\left\{\begin{array}{l}
\exists \mathrm{L}',\mathrm{LS}.\,\mathbf{l} \doteq \mathrm{L}' : \mathrm{LS} *\\
\mathsf{store}_{\mathrm{LS}}(|\texttt{x}:\texttt{undefined},\texttt{y}:4,\texttt{z}:\texttt{null},\texttt{f}:\mathrm{L},\texttt{v}:5) *\\
\mathsf{newobj}(\mathrm{L}',@\mathit{proto},@\mathit{this},\texttt{w},\texttt{v}) * (\mathrm{L}',@\mathit{proto}) \mapsto \texttt{null} *\\
(\mathrm{L}',\texttt{w}) \mapsto \texttt{null} * (\mathrm{L}',\texttt{v}) \mapsto 4 * \mathrm{true}
\end{array}\right\}
\end{array}
$$

Figure 2. Reasoning about variable scopes.

```
x = null; y = null; z = null;
f = function(w){x=v;v=4;var v;y=v;};
v = 5; f(null); z = v;
```

With the store predicate and the results given above, reasoning about this program is simple. A proof of the main program is shown in Figure 2. It relies on a simple proof of the function body summarised here and given in full in [11].

Reasoning about `with`. This level of abstraction also leads itself to reasoning about the notorious `with` statement. Consider the `with` example from Section 2 (where `f` implicitly returns `b`):

```
a = {b:1}; with (a){f=function(c){b}};
a = {b:2}; f(null)
```

This program demonstrates the importance of modelling `with` correctly. Recall that when correctly modelled, the closure of the function `f` will refer to the object $\{\texttt{b:1}\}$, which was pointed to by the variable `a` at the time that `f` was defined. However, even though the variable `a` is changed to point to a different object before `f(null)` is called, the closure continues to point to the object $\{\texttt{b:1}\}$. Thus the program normally returns the value 1, not 2. Recall also that we must be careful with the `f` field of the object l_{op}.

We can reason about this program using the store predicate. The proof is in Figure 3. This proof relies on a sub-proof for the invocation of the function `f(null)`, which culminates with the judgement $\{P\}\texttt{b}\{P * \mathbf{r} \doteq 1\}$, where P is

$$
\begin{array}{l}
\text{Let } P = (\mathrm{L},\texttt{b}) \mapsto 1 * (\mathrm{L},@\mathit{proto}) \mapsto l_{op} * \mathrm{true}\\
\{\ \ \mathsf{store}_{\mathbf{l}}(\texttt{a},\texttt{f}|) \mathbin{\sqcup\!\!\!\!\ast} (l_{op},\texttt{f}) \mapsto \varnothing \mathbin{\sqcup\!\!\!\!\ast} (l_{op},@\mathit{proto}) \mapsto \texttt{null}\ \ \}\\
\texttt{a = }\{\texttt{b:1}\}\texttt{;}\\
\left\{\begin{array}{l}
\exists \mathrm{L}.\,\mathsf{store}_{\mathbf{l}}(\texttt{f}|\texttt{a}:\mathrm{L}) \mathbin{\sqcup\!\!\!\!\ast} (l_{op},\texttt{f}) \mapsto \varnothing \mathbin{\sqcup\!\!\!\!\ast}\\
(l_{op},@\mathit{proto}) \mapsto \texttt{null} * (\mathrm{L},\texttt{f}) \mapsto \varnothing * P
\end{array}\right\}\\
\texttt{with (a)}\{\\
\quad\left\{\begin{array}{l}
\exists \mathrm{LS},\mathrm{L}.\,\mathbf{l} \doteq \mathrm{L} : \mathrm{LS} *\\
\mathsf{store}_{\mathrm{LS}}(\texttt{f}|\texttt{a}:\mathrm{L}) \mathbin{\sqcup\!\!\!\!\ast} (l_{op},\texttt{f}) \mapsto \varnothing \mathbin{\sqcup\!\!\!\!\ast}\\
(l_{op},@\mathit{proto}) \mapsto \texttt{null} * (\mathrm{L},\texttt{f}) \mapsto \varnothing * P
\end{array}\right\}\\
\quad\texttt{f=function(c)}\{\texttt{b}\}\\
\quad\left\{\begin{array}{l}
\exists \mathrm{LS},\mathrm{L},\mathrm{F}.\,\mathbf{l} \doteq \mathrm{L} : \mathrm{LS} *\\
\mathsf{store}_{\mathrm{LS}}(|\texttt{a}:\mathrm{L},\texttt{f}:\mathrm{F}) \mathbin{\sqcup\!\!\!\!\ast} (l_{op},\texttt{f}) \mapsto \varnothing \mathbin{\sqcup\!\!\!\!\ast}\\
(l_{op},@\mathit{proto}) \mapsto \texttt{null} * (\mathrm{L},\texttt{f}) \mapsto \varnothing *\\
(\mathrm{F},@\mathit{body}) \mapsto \lambda\texttt{c}.\{\texttt{b}\} * (\mathrm{F},@\mathit{scope}) \mapsto \mathrm{L} : \mathrm{LS} * P
\end{array}\right\}\\
\}\texttt{;}\\
\left\{\begin{array}{l}
\exists \mathrm{LS},\mathrm{L},\mathrm{F}.\,\mathbf{l} \doteq \mathrm{LS} *\\
\mathsf{store}_{\mathrm{LS}}(|\texttt{a}:\mathrm{L},\texttt{f}:\mathrm{F}) \mathbin{\sqcup\!\!\!\!\ast} (l_{op},\texttt{f}) \mapsto \varnothing \mathbin{\sqcup\!\!\!\!\ast} (l_{op},@\mathit{proto}) \mapsto \texttt{null} *\\
(\mathrm{L},\texttt{f}) \mapsto \varnothing * (\mathrm{F},@\mathit{body}) \mapsto \lambda\texttt{c}.\{\texttt{b}\} * (\mathrm{F},@\mathit{scope}) \mapsto \mathrm{L} : \mathrm{LS} * P
\end{array}\right\}\\
\texttt{a = }\{\texttt{b:2}\}\texttt{;}\\
\left\{\begin{array}{l}
\exists \mathrm{LS},\mathrm{L},\mathrm{F},\mathrm{L}'.\,\mathbf{l} \doteq \mathrm{LS} *\\
\mathsf{store}_{\mathrm{LS}}(|\texttt{a}:\mathrm{L}',\texttt{f}:\mathrm{F}) \mathbin{\sqcup\!\!\!\!\ast} (l_{op},\texttt{f}) \mapsto \varnothing \mathbin{\sqcup\!\!\!\!\ast} (l_{op},@\mathit{proto}) \mapsto \texttt{null} *\\
(\mathrm{L},\texttt{f}) \mapsto \varnothing * (\mathrm{L}',\texttt{b}) \mapsto 2 * (\mathrm{L}',\texttt{f}) \mapsto \varnothing * (\mathrm{L}',@\mathit{proto}) \mapsto l_{op} *\\
(\mathrm{F},@\mathit{body}) \mapsto \lambda\texttt{c}.\{\texttt{b}\} * (\mathrm{F},@\mathit{scope}) \mapsto \mathrm{L} : \mathrm{LS} * P
\end{array}\right\}\\
\texttt{f(null)}\\
\left\{\begin{array}{l}
\exists \mathrm{LS},\mathrm{L},\mathrm{F},\mathrm{L}',\mathrm{LOC}.\,\mathbf{l} \doteq \mathrm{LS} *\\
\mathsf{store}_{\mathrm{LS}}(|\texttt{a}:\mathrm{L}',\texttt{f}:\mathrm{F}) \mathbin{\sqcup\!\!\!\!\ast} (l_{op},\texttt{f}) \mapsto \varnothing \mathbin{\sqcup\!\!\!\!\ast} (l_{op},@\mathit{proto}) \mapsto \texttt{null} *\\
(\mathrm{L},\texttt{f}) \mapsto \varnothing * (\mathrm{L}',\texttt{b}) \mapsto 2 * (\mathrm{L}',\texttt{f}) \mapsto \varnothing * (\mathrm{L}',@\mathit{proto}) \mapsto l_{op} *\\
(\mathrm{F},@\mathit{body}) \mapsto \lambda\texttt{c}.\{\texttt{b}\} * (\mathrm{F},@\mathit{scope}) \mapsto \mathrm{L} : \mathrm{LS} *\\
(\mathrm{LOC},\texttt{b}) \mapsto \varnothing * (\mathrm{LOC},@\mathit{proto}) \mapsto \texttt{null} * P * \mathbf{r} \doteq 1
\end{array}\right\}\\
\{\mathbf{r} \doteq 1 * \mathrm{true}\}
\end{array}
$$

Figure 3. Reasoning about `with`.

$$
\left(\begin{array}{l}
\exists \mathrm{LS},\mathrm{L},\mathrm{F},\mathrm{L}',\mathrm{LOC}.\,\mathbf{l} \doteq \mathrm{LOC} : \mathrm{L} : \mathrm{LS} *\\
\mathsf{store}_{\mathrm{LS}}(|\texttt{a}:\mathrm{L}',\texttt{f}:\mathrm{F}) \mathbin{\sqcup\!\!\!\!\ast} (l_{op},\texttt{f}) \mapsto \varnothing \mathbin{\sqcup\!\!\!\!\ast}\\
(l_{op},@\mathit{proto}) \mapsto \texttt{null} * \mathrm{true} *\\
(\mathrm{L},\texttt{b}) \mapsto 1 * (\mathrm{L},\texttt{f}) \mapsto \varnothing * (\mathrm{L},@\mathit{proto}) \mapsto l_{op} *\\
(\mathrm{L}',\texttt{b}) \mapsto 2 * (\mathrm{L}',\texttt{f}) \mapsto \varnothing * (\mathrm{L}',@\mathit{proto}) \mapsto l_{op} *\\
(\mathrm{F},@\mathit{body}) \mapsto \lambda\texttt{c}.\{\texttt{b}\} * (\mathrm{F},@\mathit{scope}) \mapsto \mathrm{L} : \mathrm{LS} *\\
(\mathrm{LOC},\texttt{b}) \mapsto \varnothing * (\mathrm{LOC},@\mathit{proto}) \mapsto \texttt{null}
\end{array}\right)
$$

For space reasons, we only give the reasoning for the case in which neither `a` nor `f` are in the variable store. The same techniques in tandem with the disjunction rule can be used to prove the general precondition.

In both the simple and the general case, we must constrain our precondition with the assertion $(l_{op},\texttt{f}) \mapsto \varnothing \mathbin{\sqcup\!\!\!\!\ast} (l_{op},@\mathit{proto}) \mapsto \texttt{null}$. The requirement for this term may seem surprising. Consider running the above program in a state satisfying $\mathsf{store}_{\mathbf{l}}(\texttt{a},\texttt{f}|) \mathbin{\sqcup\!\!\!\!\ast} (l_{op},\texttt{f}) \mapsto 4$. In this case, when the assignment to `f` is made, the function pointer will be written to the cell $(\mathrm{L},\texttt{f})$, rather than into the global variable store. Since the variable store does not contain a function value for the variable `f`, the call to `f(null)` will cause the program to fault. The problem is potentially even worse if $(l_{op},\texttt{f})$ contains a function pointer. In this case, the call to `f(null)` will not fault, but rather will execute whatever code it finds. This kind of unpredictability could lead to very confusing bugs. In the case of a system like Facebook which attempts to isolate user-application code from host-page code, it could even lead to a security flaw.

6.3 Layer 3: a Recursive Abstract Variable Store

While reasoning using the store predicate, it is possible to handle large numbers of assignments and small numbers of function calls. However, for large numbers of function calls, another abstraction is helpful. Consider an *abstract* variable store as a list of lists of variable-value pairs, with the most local scope frame at the head of the outer list. The list $[[x = 4], [y = 5], [x = 6, z = 7]]$ represents a store in which the global scope contains the variables x and z, an intermediate scope adds the variable y, and the most local scope overrides the variable x. The list elements of variable-value pairs can be represented in our logical expression language as lists containing two elements. For readability, we use the notation $x = v$ above. We define the recursive store predicate $\mathsf{recstore}_L(\mathit{NoVars}, \mathit{Store})$ which describes an abstract variable store Store, which does not contain the variables in the list NoVars.

The Recursive recstore **predicate.**

$$
\begin{array}{l}
\mathsf{recstore}_L([\mathtt{x1}', \cdots \mathtt{xm}'], [[\mathtt{x1} = V_1, \cdots, \mathtt{xn} = V_n]]) \triangleq \\
\quad \mathsf{store}_L(\mathtt{x1}', \cdots \mathtt{xm}' | \mathtt{x1} : V_1, \cdots, \mathtt{xn} : V_n) \\[1em]
\mathsf{recstore}_{L:LS}([\mathtt{x1}', \cdots \mathtt{xm}'], ([\mathtt{x1} = V_1, \cdots, \mathtt{xn} = V_n] : Ls')) \triangleq \\
\quad \mathsf{recstore}_{LS}([\mathtt{x1}', \cdots \mathtt{xm}'], Ls') * (L, @\mathit{proto}) \mapsto \mathtt{null} * L \dot{\neq} l_g * \\
\quad *_{i \in 1..m} \ (L, \mathtt{xi}') \mapsto \varnothing *_{j \in 1..n} \ (L, \mathtt{xj}) \mapsto V_j * \\
\quad \mathsf{nones}_L([\mathtt{x1}, \cdots, \mathtt{xn}], Ls') \\[1em]
\mathsf{nones}_L(\_, [\,]) \triangleq \emptyset \\[1em]
\mathsf{nones}_L(Ls, ([\mathtt{x1} = V_1, \cdots, \mathtt{xn} = V_n] : Ls')) \triangleq \\
\quad\quad *_{i \in 1..n}((\mathtt{xi} \mathrel{\dot{\in}} Ls) \vee (\mathtt{xi} \mathrel{\dot{\notin}} Ls \wedge (L, \mathtt{xi}) \mapsto \varnothing)) * \\
\quad\quad \mathsf{nones}_L((\mathtt{x1} : \cdots : \mathtt{xn} : Ls), Ls')
\end{array}
$$

Notice that recstore uses the store predicate to constrain the most global scope frame in the abstract scope list, while being rather more restrictive about more local scope frames. Local scope frames must be emulated by JavaScript objects which contain fields for each variable defined at that scope level. Furthermore, they must assert the absence of fields which are not in the store (the NoVars), or which would otherwise shadow a variable declared in a more global position in the store. This is handled by the nones predicate. Notice that local scope frames have a `null` prototype, and may not be the l_g object. These criteria are met by the emulated scope frames created by a normal function call, and are not normally met by `with` calls. This makes this abstraction ideal for reasoning about programs with many function calls and no internal uses of the `with` statement. Notice, however, that we do not outlaw `with` calls in the enclosing scope, represented here by a top-level use of the store predicate. This means that this abstraction will facilitate reasoning about libraries which are written in a principled way, and which may be called by unprincipled clients.

We provide several rules for reasoning at this level of abstraction in [11], the most interesting of which are destructive variable initialisation and update.

Destructive recstore **update.**

$$
\frac{\begin{array}{l}
R = \mathsf{recstore}_{\mathbf{l}}((\mathtt{x} : \mathit{EmpVars}), (\mathit{Locals} +\!\!+ [\mathit{Globals}])) \\
\{R * P\}\mathsf{e}\{R * Q * \mathbf{r} \doteq \mathit{Var}\} \\
\mathbf{r} \notin \mathsf{fv}(Q) \\
S = \mathsf{recstore}_{\mathbf{l}}((\mathit{EmpVars}), (\mathit{Locals} +\!\!+ [\mathtt{x} = \mathit{Var} : \mathit{Globals}]))
\end{array}}{\{R * P\}\mathtt{x = e}\{S * Q * \mathrm{true}\}}
$$

$$
\frac{\begin{array}{l}
R = \mathsf{recstore}_{\mathbf{l}}((\mathit{Emps}), (\mathit{Ls} +\!\!+ ((\mathtt{x} = \mathit{Var}) : \mathit{Curr}) +\!\!+ \mathit{Globs})) \\
\{R * \mathit{Globs} \dot{\neq} [\,] * P\}\mathsf{e}\{R * \mathit{Globs} \dot{\neq} [\,] * Q * \mathbf{r} \doteq \mathit{Var}'\} \\
\mathbf{r} \notin \mathsf{fv}(Q) \\
\forall \mathrm{LS} \in \mathit{Ls}.\, (\mathtt{x} = \_) \mathrel{\dot{\notin}} \mathrm{LS} \\
S = \mathsf{recstore}_{\mathbf{l}}((\mathit{Emps}), (\mathit{Ls} +\!\!+ ((\mathtt{x} = \mathit{Var}') : \mathit{Curr}) +\!\!+ \mathit{Globs}))
\end{array}}{\{R * \mathit{Globs} \dot{\neq} [\,] * P\}\mathtt{x = e}\{S * \mathit{Globs} \dot{\neq} [\,] * Q * \mathbf{r} \doteq \mathit{Var}'\}}
$$

Notice that we may not safely update variables in the global portion of the abstract variable store with the results of potentially destructive expressions. This is for the same reason as the corresponding restriction on the store predicate in Section 6.2; there is a corner case which would lead to very unexpected behaviour. At this level of abstraction however, we have an advantage. We can be sure that the local abstract scope frames were constructed in a more principled way, and so we are able to reason about updating them with destructive expressions using the second rule above.

$$
\left\{\begin{array}{l}
\mathsf{recstore}_{\mathbf{l}}\left([\,], \left[\left[\begin{array}{l}\mathtt{data} = \mathrm{L}, \\ \mathtt{checkField} = \mathtt{\&undefined}, \\ \mathtt{i} = \mathtt{\&undefined}\end{array}\right], [\,]\right]\right) * \\
(\mathrm{L}, \mathtt{numEntries}) \mapsto \mathrm{N} * (\mathrm{L}, \mathtt{buttonDisabled}) \mapsto \_ * \\
(\mathrm{L}, 0) \mapsto \mathrm{TXT}_0 * \ldots * (\mathrm{L}, \mathrm{N}) \mapsto \mathrm{TXT}_{\mathrm{N}}
\end{array}\right\}
$$

$$\ldots\mathtt{checkForm}\ldots$$

$$
\left\{\begin{array}{l}
\exists \mathrm{L}'.\, \mathsf{recstore}_{\mathbf{l}}([\,], [[\mathtt{data} = \mathrm{L}, \mathtt{checkField} = \mathrm{L}', \mathtt{i} = \mathrm{N}], [\,]]) * \\
(\mathrm{L}, \mathtt{numEntries}) \mapsto \mathrm{N} * \\
(\mathrm{L}, 0) \mapsto \mathrm{TXT}_0 * \ldots * (\mathrm{L}, \mathrm{N}) \mapsto \mathrm{TXT}_{\mathrm{N}} * \\
\left(\begin{array}{l}
\left(\begin{array}{l}\mathrm{TXT}_0 \dot{\neq} \text{""} * \ldots * \mathrm{TXT}_{\mathrm{N}} \dot{\neq} \text{""} * \\ (\mathrm{L}, \mathtt{buttonDisabled}) \mapsto 0\end{array}\right) \\
\vee (\mathrm{L}, \mathtt{buttonDisabled}) \mapsto 1
\end{array}\right)
\end{array}\right\}
$$

Figure 4. The specification of `checkForm`.

Form validation. Consider a web form with a number of mandatory text fields and a submit button. If the button is "disabled" when the page loads, then an event handler on the form can be used to regularly check if valid data has been entered in all the fields before enabling the button. Let us assume that the programmer has separated the concerns of parsing the web page and of validating the data. The data validation function will be called with a single parameter: an object with one field for each text value to check, a count of those text values, and boolean toggle corresponding to whether the submit button should be disabled. An example function which might perform the validation check is:

```
checkForm = function(data) {
    data.buttonDisabled = 0;
    var checkField = function(text) {
        if(text == "") {data.buttonDisabled = 1;}}
    var i = 0;
    while(i < data.numEntries) {
        checkField(data[i]); i = i+1;}}
```

Notice that this code deals with variables in a principled way. It makes use of no global variables, preferring instead to use function parameters and local variables. The repeated work of the loop body is factored into a function which could be expanded to provide extra functionality or used elsewhere with little cost in readability. Using the recstore abstraction it is straightforward to show that the function body satisfies the specification given in Figure 4.

7. Related Work

We believe this paper is the first to propose a program logic for reasoning about JavaScript. Our program logic adapts ideas from separation logic, and proves soundness with respect to a big-step operational semantics. In this section, we discuss related work on separation logic and the semantics of JavaScript.

We build on the seminal work of O'Hearn, Reynolds and Yang [20], who introduced separation logic for reasoning about C-programs, and on the work of Parkinson and Bierman [4, 23, 24], who adapted separation logic to reason about Java. We made several adaptations to their work in order to reason about JavaScript. As in [4, 23, 24], we use assertions of the form $(\mathtt{l}, \mathtt{x}) \mapsto 5$ to denote that a field `x` in object `l` has value 5. We extend these assertions by $(\mathtt{l}, \mathtt{x}) \mapsto \varnothing$, which denotes that the field is *not* in `l`. This is inspired by Dinsdale-Young *et al.*'s use of the 'out' predicate to state that values are not present in a concurrent set [7]. We introduce the sepish connective $\sqcup\!\!\!\!\!\ast$ to account for partially-shared data structures. We have not seen this connective before, which is surprising since shared data structures are common for example in Linux. There has been much work on various forms of concurrent separation logic with sharing [9, 21, 35], but they all seem to take a different approach to our $\sqcup\!\!\!\!\!\ast$ connective.

We prove the soundness of our frame rule using a new technique developed in Smith's PhD thesis [30]. There are several approaches to proving soundness of the separation-logic frame rule, and we list the key developments here. The first and most commonly used approach by O'Hearn, Reynolds and Yang involves first proving that the commands of the programming language under consideration are *local*, and then using that property to prove the frame rule [20]. Later, Birkedal and Yang devised a method "baking in" the soundness of the frame rule into their definition of Hoare triples [5]. This made it possible to prove the frame rule without reference to locality, but did not extend well to concurrent programs. Vafeiadis solved the concurrency problem in [34], where he proves soundness of concurrent separation logic without reference to locality. Smith's thesis was written at the same time as [34], and takes a different approach. Rather than removing all reference to locality from the soundness proof, Smith uses a generalised *weak locality* property, which allows us to handle some non-local behaviour.

We prove our soundness result with respect to a big-step operational semantics of JavaScript, faithfully following the small-step semantics of Maffeis *et al.* [18] except where discussed in Section 3.5. Their semantics captures the complete ECMAScript 3 language, at the same level of abstraction to where a JavaScript programmer reasons. In contrast,Herman and Flanagan [13] provide a definitional interpreter of JavaScript written in ML, which has the advantage of being directly executable, but includes implementation details that obscure the semantic meaning. 'Elsewhere, Guha *et al.* [12] compile JavaScript to an intermediate Scheme-like language, in order to provide type-based analyses on the object language. They justify their translation in a novel way, by checking that it satisfies an established test suite of JavaScript programs. However, the translation is complicated and it is not immediately apparent that it is sound. In fact, many approaches translate JavaScript programs using troublesome features such as the `with` construct into programs which do not. For example, Park, Lee and Ryu show that this is often feasible [22], but do not handle interactions between `with` and `eval`. There are a number of more abstract models of JavaScript, which have proven useful to study selected language features [1, 32, 37], but which are not sufficiently concrete for our purpose. Overall, we have chosen the semantics in [18] because it appears to be the most faithful to the actual JavaScript semantics. As Richards *et al.* argue in [28], all the unusual features of JavaScript are well-used in the wild, and cannot be easily abstracted away.

8. Conclusions and Future Work

We have defined a program logic for reasoning about JavaScript, based on an operational semantics which is essentially faithful to ECMAScript 3. We have adapted separation logic to reason about a JavaScript subset, modelling many complex features such as prototype inheritance and `with`. We reason about the full dynamic nature of JavaScript's functions, but do not provide higher-order reasoning. We also provide only conservative reasoning about `eval`. Full reasoning about these features will be technically challenging, although we believe that we can build on the recent work of [6, 10, 29]. We do not currently reason about implicit type coercion. JavaScript allows the programmer to define functions to convert values between different types. Adding this feature to our operational semantics is simple. The complexity in handling it with our program reasoning will be in reasoning about higher-order functions, as discussed above.

Due to our choice of operational semantics, we have been able to prove a strong soundness result. Any library, formally specified using our reasoning, will behave well, even when called by arbitrary JavaScript code. In Section 2, we illustrated this property by demonstrating that our `with` example behaves according to its specification even when embedded in a program which maliciously uses the non-standard `__proto__` feature. Finally, our soundness result can be extended compositionally to include more sophisticated reasoning about higher-order functions and `eval`.

We have given several examples of our reasoning, demonstrating with small code examples that JavaScript is fiendish to understand and our reasoning can help. In particular, our `with` example shows a potential bug that could easily go unnoticed for some time, and perhaps lead to security holes in sanitised mashup environments. Despite the complexity of the language and the subtlety of the bug, reasoning about this and other examples is made surprisingly simple by our example abstraction layers.

We hope that this work will form the core of a larger body of work on client-side web programming. For example, Thiemann [33] has defined a type-safe DOM API, and Smith [30] has developed context-logic reasoning about DOM Core Level 1. It would be valuable to integrate these approaches to DOM modelling with the JavaScript reasoning presented here.

Maffeis et al. [16, 17, 19] have developed techniques for building secure JavaScript mashups out of untrusted code. They prove security properties of their enforcement mechanisms, but do not study their functional correctness. Our reasoning makes such an analysis possible. We also intend to develop reasoning for high-level libraries such as jQuery, Prototype.js and Slidy [15, 25, 26]. We will make each of these libraries the focus of its own layer of abstraction, following the examples in Section 6, to make reasoning about idiomatic uses of those libraries very natural. As with our existing layers of abstraction, it will be possible to safely break these abstractions where necessary in order to code and reason at a lower level.

While this paper deals with ECMAScript 3, the newer ECMAScript 5 and the ongoing ECMAScript Harmony both provide enticing targets. Taly and others have recently given a small-step operational semantics of ECMAScript 5 [31]. It would certainly be interesting to reason about ECMAScript 5 code, and to formalise the connections with our reasoning presented here about ECMAScript 3. We are particularly interested in understanding the behaviour of libraries written in ECMAScript 5 when called by code written in ECMAScript 3.

To make this program reasoning genuinely useful for JavaScript programmers, it is essential that we provide tool support. We intend to produce analysis tools capable of spotting bugs such as the one described in the `with` example in Section 6.2, and integrate our tools with IDEs such as Eclipse or Visual Studio.

Acknowledgements.
Gardner and Smith are supported by EPSRC grant COTFM_P21654.
Maffeis is supported by EPSRC grant EP/I004246/1.
We thank Charlton, Dinsdale-Young, Ntzik, O'Hearn, da Rocha Pinto, Wheelhouse, and Wright for insightful discussions.
We especially thank Daiva Naudziuniene for her impressive eye for detail.

References

[1] C. Anderson, P. Giannini, and S. Drossopoulou. Towards type inference for JavaScript. In *Proc. of ECOOP'05*, 2005. 7

[2] J. Berdine, C. Calcagno, and P. O'Hearn. Smallfoot: Modular automatic assertion checking with separation logic. In *FMCO*, 2005. 1

[3] J. Berdine, B. Cook, and S. Ishtiaq. Slayer: Memory safety for systems-level code. In *CAV*, 2011. 1

[4] G.M. Bierman, M.J. Parkinson, and A. M. Pitts. MJ: An imperative core calculus for java and java with effects. Technical report, Cambridge, 2003. 4, 7

[5] L. Birkedal and H. Yang. Relational parametricity and separation logic. In *FoSSaCS*, pages 93–107, 2007. 7

[6] N. Charlton. Hoare logic for higher order store using simple semantics. In *Proc. of WOLLIC 2011*, 2011. 1, 8

[7] T. Dinsdale-Young, M. Dodds, P. Gardner, M. Parkinson, and V. Vafeiadis. Concurrent abstract predicates. *ECOOP*, 2010. 2, 7

[8] D. Distefano and M. Parkinson. jStar: towards practical verification for Java. In *OOPSLA '08*, pages 213–226. ACM, 2008. 1

[9] M. Dodds, X. Feng, M.J. Parkinson, and V. Vafeiadis. Deny-guarantee reasoning, 2009. 7

[10] D. Dreyer, G. Neis, and L. Birkedal. The impact of higher-order state and control effects on local relational reasoning. In *ICFP*, pages 143–156, 2010. 8

[11] P. Gardner, S. Maffeis, and G. Smith. Towards a program logic for JavaScript. Imperial College London Technical Report number DTR11-11, November 2011. 3.1, 3.3, 3.3, 3.4, 4, 5.1, 5.2, 5.3, 6.1, 6.2, 6.3

[12] A. Guha, C. Saftoiu, and S. Krishnamurthi. The Essence of JavaScript. *ECOOP 2010*, pages 126–150, 2010. 1, 2, 7

[13] D. Herman and C. Flanagan. Status report: specifying JavaScript with ML. In *Proc. of ML'07*, pages 47–52, 2007. 7

[14] ECMA International. ECMAScript language specification. stardard ECMA-262, 3rd Edition, 1999. 1, 2

[15] jQuery: The write less, do more, JavaScript library. `http://jquery.com`. 8

[16] S. Maffeis, J. C. Mitchell, and A. Taly. Isolating javascript with filters, rewriting, and wrappers. In *ESORICS*, pages 505–522, 2009. 8

[17] S. Maffeis, J. C. Mitchell, and A. Taly. Object capabilities and isolation of untrusted web applications. In *IEEE Symposium on Security and Privacy*, pages 125–140, 2010. 8

[18] S. Maffeis, J.C. Mitchell, and A. Taly. An operational semantics for JavaScript. In *Proc. of APLAS'08*, LNCS, 2008. 1, 2, 3, 3.2, 7

[19] S. Maffeis and A. Taly. Language-based isolation of untrusted javascript. In *CSF*, pages 77–91, 2009. 8

[20] P. O'Hearn, J. C. Reynolds, and H. Yang. Local reasoning about programs that alter data structures. In *CSL*, 2001. 1, 4, 5.3, 7

[21] P. W. OHearn. Resources, concurrency, and local reasoning. *Theor. Comput. Sci.*, 375(1-3):271–307, 2007. 7

[22] Changhee Park, Hongki Lee, and Sukyoung Ryu. An empirical study on the rewritability of the with statement in javascript. In *FOOL*, 2011.

[23] M. J. Parkinson and G. M. Bierman. Separation logic, abstraction and inheritance. In *POPL*, 2008. 2, 7

[24] M. J. Parkinson. Local reasoning for Java. Technical Report 654, Univ. of Cambridge Computer Laboratory, 2005. Ph.D. dissertation. 4, 7

[25] Prototype Core Team. Prototype JavaScript framework: Easy Ajax and DOM manipulation for dynamic web applications. `http://www.prototypejs.org`. 4, 7

[26] Dave Raggett. W3C Slidy. `http://www.w3.org/Talks/Tools/Slidy2/`, 2005. 8

[27] G. Richards, C. Hammer, B. Burg, and J. Vitek. The Eval that men do A large-scale study of the use of Eval in JavaScript applications. Accepted for publication at ECOOP 2011. 8

[28] G. Richards, S. Lebresne, B. Burg, and J. Vitek. An analysis of the dynamic behavior of JavaScript programs. In *PLDI*, 2010. 2

[29] J. Schwinghammer, L. Birkedal, B. Reus, and H. Yang. Nested hoare triples and frame rules for higher-order store. In *In Proc. of CSL'09*, 2009. 1, 2, 2, 7

[30] G. D. Smith. Local reasoning about web programs. PhD Thesis, Dep. of Computing, Imperial College London, 2011. 1, 8

[31] A. Taly, U. Erlingsson, M. S. Miller, J. C. Mitchell, and J. Nagra. Automated analysis of security-critical javascript apis. In *Proc. of IEEE Security and Privacy'11*. IEEE, 2011. 1, 5.3, 7, 8

[32] P. Thiemann. Towards a type system for analyzing javascript programs. In *Proc. of ESOP'05*, volume 3444 of *LNCS*, 2005. 8

[33] P. Thiemann. A type safe DOM API. In *Proc. of DBPL*, pages 169–183, 2005. 7

[34] V. Vafeiadis. Concurrent separation logic and operational semantics. In *MFPS11*, 2011. 8

[35] Viktor Vafeiadis and M. Parkinson. A marriage of rely/guarantee and separation logic. In *IN 18TH CONCUR*. Springer, 2007. 7

[36] H. Yang, O. Lee, J. Berdine, C. Calcagno, B. Cook, D. Distefano, and P. O'Hearn. Scalable shape analysis for systems code. In *CAV*, 2008. 7

[37] D. Yu, A. Chander, N. Islam, and I. Serikov. JavaScript instrumentation for browser security. In *Proc. of POPL'07*, 2007. 1

7