A Program Logic for JavaScript

Anonymous Author
Anonymous Institution
anon@ymo.us

Abstract

JavaScript has become the de-facto language for client-side web programming. The inherent dynamic nature of the language makes understanding JavaScript 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, correct 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.

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 [13]. Adobe Flash, used in e.g. YouTube, also features a programming language based on ECMAScript, called 'ActionScript'. 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 used 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 notoriously difficult to understand [11, 15, 23]. For example, the lack of abstraction mechanisms for libraries leads to many buggy programs on the Web. While there are promising approaches to problem-specific static analyses of JavaScript [1, 6, 14, 16, 19, 26, 27, 32], there is a growing need for general-purpose, more expressive analysis tools, able to provide simple, correct descriptions of program behaviour, a clear understanding of module boundaries, and the ability to verify security contracts.

We believe that formal methods will have a significant role to play in the development of static analysis tools for Javascript, especially IDE support. In this paper, we introduce the first program logic for reasoning about JavaScript. 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. This code can be badly written, untrusted and potentially malicious. We are particularly concerned with library code, which must be well-behaved when called by arbitrary code. Our reasoning is therefore based on 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 by establishing several natural layers of abstraction on top of our basic rules. 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 [3, 4, 8, 17, 31]. A key characteristic of JavaScript is that the entire state of the language resides in the object heap. 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 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 [17]. 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 from arbitrary objects in the heap. Therefore, we prove soundness using a new concept of 'weak locality', recently introduced by Smith [25].

In this paper, we reason about a substantial subset of JavaScript, including prototype inheritance, `with` and dynamic functions. We do not provide higher-order reasoning about functions, and only provide conservative reasoning about `eval`. We prove soundness of our reasoning with respect a faithful subset of the formal operational semantics of Maffeis *et al.* [15]. Our soundness result is strong. Libraries written in our subset and proved correct with respect to their specifications will be well behaved, even when called by arbitrary JavaScript code. Our soundness result is constructed in such a way that it will be simple to extend to higher-order reasoning and reasoning about `eval` in due course.

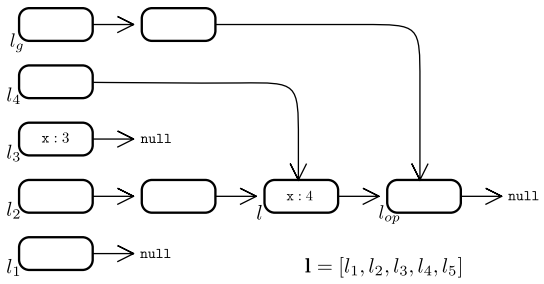

Figure 1. A JavaScript Emulated Variable Store

2. Motivating Examples

As convincingly argued in [11, 15, 22, 23], many different factors drive the complexity of JavaScript behaviour. For a start, JavaScript is a dynamically typed, prototype-oriented language, which has no variable store. Instead, JavaScript variables are stored in the heap, in a structure which imperfectly emulates the variable store of many other programming languages. This structure consists of an abstract list of scope objects, 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 counterintuitive. 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 value 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:

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

The current list of scope objects is identified by a global logical expression $\mathbf{l}$. The store predicate $\mathsf{store}_{\mathbf{l}}(\mathtt{x},\mathtt{y},\mathtt{z},\mathtt{f},\mathtt{v}|)$ states that the store-like structure referred to by $\mathbf{l}$ contains *none* of the variables mentioned in the program; the variables occur to the left of the vertical bar. The store predicate $\mathsf{store}_{\mathbf{l}}(|\mathtt{x}:\mathtt{undefined},\mathtt{y}:4,\mathtt{z}:5,\mathtt{f}:\mathrm{L},\mathtt{v}:5)$ denotes the final values for all the variables; the variables are now on 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. Each such object has a pointer to a list of prototypes, with the arrows representing prototype relationships. These prototype lists can be shared, as illustrated. They can be complete, in the sense that they end with the distinguished object l_{op} which points to `null`. They can be empty, since the prototype of a scope object may be `null`. Finally, if the browser running the program uses SpiderMonkey, V8 or WebKit, the lists can be partial in the sense that they have a `null` prototype pointer at any point in the prototype list. This last case is not illustrated in Figure 1, because it is not allowed by the ECMAScript specification. It is however sufficiently common that it is worth ensuring that it does not affect the soundness of our reasoning. Our scope predicate therefore allows such partial lists.

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. This predicate is subtle and requires some adaptation of separation logic. As well as the separating conjunction $*$ for reasoning about disjoint resource, we introduce the sepish connective $\bowtie$ for reasoning about paratially-separated resource. It is used, for example, to account for the sharing the prototype lists, illustrated in Figure 1. We also use the assertion $(l, \mathtt{x}) \mapsto \varnothing$, which states that the field variable `x` is *not* 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 protype 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:

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

where the assertion true hides possible garbaged prototype lists.

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

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

As well as the store predicate, we assert that the state also contains object cells such as $(\mathrm{L}, @scope) \mapsto \mathbf{l}$. 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 $\mathbf{l}$. 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 complicated, because the final values of the variables are counterintuitive. 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:

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

To fill-in a suitable postcondition, we must reason about the function body. The precondition of the function-body triple 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 parameters 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 $@proto$ field, which points to `null` since scope objects do not inherit any behaviour, and a $@this$ field, which can only be read. We also have the predicate $\mathsf{newobj}(\mathrm{L}', @proto, \mathtt{w}, \mathtt{v}, @this)$, which asserts the absence of all the fields we have not initialised. 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, which runs the code having assigned all the local variable declarations to `undefined`:

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

```
x = v ; v = 4 ; var v ; y = v;
```

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

The postcondition follows simply, resulting from the three assignments in the new local variable store; the `var v` statement has no effect in the second pass of the function call: 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`. Hence, we obtain the counterintuitive assignments in the store of the postcondition.

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

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

Reasoning about the final assignment is simple, 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 local scope object L:

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

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 the next version of ECMAScript, version 5. However, it is widely used in practice [23] 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:

```
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, and may execute arbitrary code from elsewhere in the variable store. In a sanitised environment such as a Facebook app, this could lead to a security violation.

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

$$\begin{array}{c} \{\mathsf{store}_{\mathbf{l}}(\mathtt{a}, \mathtt{f}|) \uplus (l_{op}, \mathtt{f}) \mapsto \varnothing \uplus (l_{op}, @proto) \mapsto \mathtt{null}\} \\ \ldots \\ \{\mathbf{r} \doteq 1 * true\} \end{array}$$

in which the precondition ensures the program returns the value 1 as expected. The obvious first try was to have just $\mathsf{store}_{\mathbf{l}}(\mathtt{a}, \mathtt{f}|)$ as the precondition. This does not work as, when reasoning about the assigment to the variable `f`, we cannot assert that the variable `f` is not in the local scope. The reason for this is that the statement `a = {b:1}` results in the creation of a new object L, with field `b`, no field `f` as expected, but with field $@proto$ pointing to the distinguished object l_{op}:

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

The `with` statement puts this new object at the beginning of the local scope object. This means that, when we attempt to assign to `f`, we must check the whole prototype list of L for field `f`. Thus, we need assertions in the precondition stipulating that the prototype chain of l_{op} does not contain any instance of `f`. Otherwise, the assignment to `f` in our program might be a local assignment. In this case, when the program returns from the `with` statement, the function call to `f(null)` will call whatever global `f()` might previously have existed in the store including, for example, an `f` at l_{op}. This can result in a fault, if the variable `f` does not point to a function object or a security breech if `f` points to bad code. This example also shows that the `with` construct cannot be soundly compiled away form JavaScript code.

3. Operational Semantics

We define a big-step operational semantics for JavaScript that represents faithfully the inheritance, prototyping and scoping mechanisms described in the ECMAScript 3 standard. Our semantics is based on the reference formal semantics [15]. Any derivation in our semantics corresponds to a valid JavaScript computation. See Section 3.6 for a discussion of our simplifying assumptions.

3.1 Heaps

The JavaScript heap is a partial function $H\colon (\mathcal{L} \times \mathcal{X}) \rightharpoonup \mathcal{V}$ that maps memory locations and variable names to values. This structure emphasises the important role that references (pairs of locations and variables) play in the semantics of the language.

Values $v \in \mathcal{V}$ can be basic values `u`, locations l and lambda abstractions λ`x.e`. The set of locations $\mathcal{L}$ is lifted to a set $\mathcal{L}_\perp$ 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 heapsby $H_1 * H_2$, and a read operation by $H(l, x)$.

An object is represented by a set of heap cells addressed by the same location but different variables. In this context, variables

stand for object property 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 $(l, x_1) \mapsto v_1 * \ldots * (l, x_n) \mapsto v_n$.

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 a runtime list called the *scope chain* (we use a standard notation $[\,], e{:}L, L +\!\!+ L$ for lists). A variable x is resolved as the property x of the first object in the scope chain that defines it. 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. Scoping constructs such as function calls, cause sub-expressions to be evaluated with respect to a local scope object, which for example defines the local variables of a function, and then defers to its enclosing scope, where the resolution of non-local variables continues. The auxiliary function σ defined below returns the location of the first object in the scope chain to define a given variable.

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

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

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

A similar mechanism is used to model prototype-based inheritance: JavaScript objects are linked in prototype chains, so that an object property is resolved to the first property of an object in the relevant prototype chain that defines it. Function π below returns the location of the first object in the prototype chain to define a given property.

Prototype resolution: $\pi(H, l, x)$**.**

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

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

We use the notation $\mathsf{obj}(l, l')$ to denote a fresh, empty object at location l with prototype l':

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

The set of variables $\mathcal{X}$ is partitioned in two disjoint sets of internal variables $\mathcal{X}^{\mathrm{I}}$ and user variables $\mathcal{X}^{\mathrm{U}}$. The internal variables $\mathcal{X}^{\mathrm{I}} \triangleq \{@scope, @body, @proto, @this\}$ are not directly accessible by user code, but are used by the semantics. User variables are denoted by $\mathtt{x}, \mathtt{y}, \mathtt{z} \in \mathcal{X}^{\mathrm{U}}$ and are a subset of strings. In particular, 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 γ function defined below, which returns the value denoted by the reference.

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

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

A JavaScript expression can only be evaluated in a well-formed heap, with respect to a valid scope chain. A heap H is *well-formed* (denoted by $H \vdash \diamond$) if its objects and prototype chains are well-formed, and if it contains the global scope object l_g and the default prototypes for objects l_{op} and functions l_{fp}. 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 contains the global object l_g. Formal definitions are given in [2].

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

$$H^{\emptyset} \triangleq \left(\begin{array}{l} l_g \mapsto \{@proto : l_{op}, @this : l_g\} * \mathsf{obj}(l_{op}, \texttt{null}) \\ * \mathsf{obj}(l_{fp}, l_{op}) * \mathsf{obj}(l_e, l_{ep}) * \mathsf{obj}(l_{ep}, l_{op}) \end{array} \right)$$

We conclude this section by defining the heap update $- \uplus -$ operation which will be used by the semantics.

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

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

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

$$H \uplus (\texttt{null}, \mathtt{x}) \mapsto v \triangleq H \uplus (l_g, \mathtt{x}) \mapsto v$$

3.2 Terms

The syntax for terms of our JavaScript subset is reported below.

Syntax of Terms: `v`, `e`.

```
v  ::=  n | m | undefined | null
e  ::=  e;e | x | v | if(e){e}{e} | while(e){e} | var x
        | this | delete e | e ⊕ e | e.x | e(e) | e = e
        | function (x){e} | function x(x){e} | new e(e)
        | {x1 : e1 ... xn : en} | e[e] | with(e){e}
```

A basic value `v` can be a number `n`, a string `m`, the special constant `undefined` or the `null` location. $\oplus \in \{+, -, *, /, \&\&, ||, ==, .\}$ are the standard number and boolean operators plus string concatenation. Expressions `e` include sequential composition, variable lookup, literal values, conditional expressions, loops, arithmetic and string concatenation, object property lookup, function call, assignment, literal objects, functions and recursive functions.

3.3 Evaluation rules

An expression `e` is evaluated in a heap H, with a scope chain L. If it successfully terminates, it returns a modified heap H' and a final value r. Our big-step operational semantics for expressions uses an evaluation relation $\longrightarrow$, defined on configuration triples $H, L, \mathtt{e}$, and terminal states H', r or fault. Selected evaluation rules are given below, see [2] for the full table. Recall that a heap value v can be a user value `v`, a memory location l or a function closure $\lambda \mathtt{x}.\mathtt{e}$, and a return value r can also be a reference $l \cdot \mathtt{x}$ (see [2]).

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)
$$H, L, \texttt{var x} \longrightarrow H, \texttt{undefined}$$

(Value)
$$H, L, \mathtt{v} \longrightarrow H, \mathtt{v}$$

(Member Access)
$$\frac{H, L, \mathtt{e} \xrightarrow{\gamma} H', l' \quad l' \neq \texttt{null}}{H, L, \mathtt{e.x} \longrightarrow H', l' \cdot \mathtt{x}}$$

(Computed Access)
$$\frac{H, L, \mathtt{e1} \xrightarrow{\gamma} H_1, l' \quad l' \neq \texttt{null} \quad H_1, L, \mathtt{e2} \xrightarrow{\gamma} H', \mathtt{x}}{H, L, \mathtt{e1[e2]} \longrightarrow H', l' \cdot \mathtt{x}}$$

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

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

(Binary Operators)
$$\frac{H, L, \mathtt{e1} \xrightarrow{\gamma} H'', \mathtt{v1} \quad H'', L, \mathtt{e2} \xrightarrow{\gamma} H', \mathtt{v2} \quad \mathtt{v1} \oplus \mathtt{v2} = \mathtt{v}}{H, L, \mathtt{e1} \oplus \mathtt{e2} \longrightarrow H', \mathtt{v}}$$

(Assignment)
$$\frac{H, L, \mathtt{e1} \longrightarrow H_1, l' \cdot \mathtt{x} \quad H_1, L, \mathtt{e2} \xrightarrow{\gamma} H_2, v \quad H' = H_2 \uplus (l', \mathtt{x}) \mapsto v}{H, L, \mathtt{e1=e2} \longrightarrow H', v}$$

(This)
$$\frac{\sigma(H, L, @this) = l_1 \quad \pi(H, l_1, @this) = l_2 \quad H(l_2, @this) = l'}{H, L, \texttt{this} \longrightarrow H, l'}$$

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

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

(Eval)
$$\frac{H, L, \mathtt{e1} \xrightarrow{\gamma} H_1, l_e \quad H_1, L, \mathtt{e2} \xrightarrow{\gamma} H_2, \mathtt{s} \quad \mathsf{parse}(\mathtt{s}) = \mathtt{e} \quad H_2, L, \mathtt{e} \xrightarrow{\gamma} H', v'}{H, L, \texttt{e1(e2)} \longrightarrow H', v'}$$

(With)
$$\frac{H, L, \mathtt{e1} \xrightarrow{\gamma} H_1, l \quad H_1, l{:}L, \mathtt{e2} \longrightarrow H', r}{H, L, \texttt{with(e1)\{e2\}} \longrightarrow H', r}$$

(Fault)
$$\frac{\text{otherwise}}{H, L, \mathtt{e} \longrightarrow \mathsf{fault}}$$

We now briefly discuss some of the evaluation rules that show non-standard features typical of JavaScript. Rule (Definition) has no effect: the `var` declaration is only used by defs (defined below) to identify function local variables. Rule (Variable) determines the scope object where a given variable can be found, without dereferencing the variable. Rules (Member/Computed Access) return a reference to the object field denoted by the corresponding expressions. Rule (Object) introduces a fresh, empty object at location l, and then initializes its fields accordingly. Freshness is ensured by well-formedness of H and disjointness of $*$. Rule (Binary Operations) 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`. If `x` is defined as a property of an object in the scope chain (for example, if we are executing the body of a function where `x` is a local variable) then `x=4` is the usual *overwriting* assignment. If `x` cannot be found anywhere, then it is created as a global variable. Finally, `x` could be found in a prototype of an object in the scope chain. In that case, `x=4` is an *overriding* assignment, with the effect of adding `x` as a local variable in the scope whose prototype defined `x`. Rule (This) resolves the `this` identifier, always returning a valid (non-`null`) object in a well-formed state. Rule (Function) introduces the notation $\mathsf{fun}(l', L, \mathtt{x}, \mathtt{e}, l) \triangleq$

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

to allocate a fresh function object at location l', and creates a new empty prototype object at l. Rule (Function Call) uses two auxiliary functions This and act. Function This determines the object that should be bound to the `this` in the function body:

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

The rationale is that in the expression `o.f(null)`, the `this` of `f` will be `o`, whereas in the expression `f(null)` the `this` of `f` will be the global object. The auxiliary function act describes the allocation of a function activation record (which is a special kind of scope object): $\mathsf{act}(l, \mathtt{x}, v, \mathtt{e}, l'') \triangleq$

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

The auxiliary function defs, defined in [2],returns the fresh memory needed to allocate the local variables defined by a function body. $\mathsf{defs}(\mathtt{y}, l, \texttt{var x})$, where `y` is the formal parameter of the function being defined, returns the cell $(l, \mathtt{x}) \mapsto \texttt{undefined}$ if $\mathtt{x} \neq \mathtt{y}$. All other rules for defs propagate this information homomorphically. Rule (Eval) assumes a partial function parse that parses a string into a JavaScript expression, only if there are no syntax errors.

The imperative statements are more standard than the expressions, except for the unusual (With) rule that uses the (possibly user-defined) object obtained by evaluating `e1` as a local scope to evaluate `e2`. The (Fault) rule applies when no other rules apply. Because of potential divergences in the evaluation of subexpressions, this rule is undecidable, and should be considered as a specification device to define illegal states rather, than an operational rule.

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 $H, L, \mathtt{e} \longrightarrow H', r$ then $H' \vdash \diamond$.*

Although the theorem is about end-to-end well-formedness, its proof (reported in Section 2 of [2]) 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 Scope Example

We now revisit the scope example to illustrate some actual evaluation steps of the semantics. Let `e1` be the code:

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

We evaluate the state $(H^{\emptyset}, [l_g], \mathtt{e})$ using rule (Sequence). We first compute $(H^{\emptyset}, [l_g], \texttt{x = null}) \longrightarrow H_1, \texttt{null}$, where $H_1 = H^{\emptyset} * (l_g, \mathtt{x}) \mapsto \texttt{null}$. Then, if we find some H, r such that $H_1, [l_g], \mathtt{e2} \longrightarrow H, r$ we can conclude that $H^{\emptyset}, [l_g], \mathtt{e1} \longrightarrow H, r$, where H and r are the final heap and return value of the computation, and `e2` is the continuation of `e1`. Applying a similar reasoning to `e2` a few times, we can isolate the last two sub-derivations $H_5, [l_g], \texttt{f(null)} \longrightarrow H_6, r_1$ and $H_6, [l_g], \texttt{z = v} \longrightarrow H, v$. The first sub-derivation is the most interesting. It must follow by rule (Function Call), where the last premise is the sub-derivation

$$H', [l_\lambda, l_g], (\texttt{x=v; v=4; var v; y=v;}) \xrightarrow{\gamma} H_6, v_1$$

where l_λ is the scope object for executing the function call. It is easy to see that $H_5 = I * l_g \mapsto \{G\} * l_{\mathtt{f}} \mapsto \{F\} * \mathsf{obj}(l_{\mathtt{f}_p})$, where

$$I = l_{op} \mapsto \{@proto : \texttt{null}\} * \mathsf{obj}(l_{fp}) * \\ l_e \mapsto \{@proto : l_{ep}\} * \mathsf{obj}(l_{ep})$$

$$G = @proto{:}\, l_{op}, @this{:}\, l_g, \mathtt{x}{:}\, \texttt{null}, \mathtt{y}{:}\, \texttt{null}, \mathtt{z}{:}\, \texttt{null}, \mathtt{f}{:}\, l_{\mathtt{f}}, \mathtt{v}{:}\, 5$$

$$F = @proto{:}\, l_{fp}, \texttt{prototype}{:}\, l_{\mathtt{f}_p}, @scope{:}\, [l_g], @body{:}\, \texttt{[...]}$$

Heap $H' = H_5 * S$ is obtained by allocating the scope S at l_λ:

$$S = l_\lambda \mapsto \{\mathtt{w} : \texttt{null}, @this : l_g, @proto : \texttt{null}, \mathtt{v} : \texttt{undefined}\}$$

Note that `v` has been added as an undefined local variable by defs. At this point we can easily deduce H_6 and H. In particular, for some appropriate H'',

$$H = H'' * l_g \mapsto \{\mathtt{x} : \texttt{undefined}, \mathtt{y} : 4, \mathtt{z} : 5, \mathtt{v} : 5\}$$

3.6 Simplifying Assumptions

We discuss some non-core JavaScript features that we have omitted from our semantics, in order to limit size and complexity of our semantics for this paper. Unless explicitly stated, adding such features would not incure significant conceptual difficulties. Despite these omissions, we work with a subset of JavaScript which is very faithful to the ECMAScript 3 standard, modulo the corner cases discussed below. Significantly, 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. Hence, our reasoning of Section 5 will interface well with real-world JavaScript programs.

We do not model implicit type-coercion functions, and as a consequence of this, we have no boolean type. Instead, where control structures (`if` and `while`) require a boolean, we use other types with semantics equivalent to the type conversion that occurs in JavaScript. 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. Although JavaScript separates programs, statements and expressions, we merge these three categories in a single sort of expression. This choice simplifies the formal machinery but also allows our model to have valid expressions that are not JavaScript terms (for example, the sum of two `if`s returning numbers). We also chose to omit several JavaScript constructs such as labels, `switch` and `for`, as they do not contribute significantly to the problem of program reasoning. At this stage, we consider a native heap inhabited only by the global object, `Object.prototype`, the `eval` function, its prototype, and `Function.prototype`, and we only model the properties of these objects that we find useful to illustrate the semantics or the reasoning rules. Instead of exceptions, we use a general semantic rule evaluating to `fault`. Our reasoning conservatively avoids faults, and our operational semantics induces faults in many cases where full JavaScript is more subtle. As a result, programs which are proved using our fault-avoiding local Hoare reasoning will run without throwing exceptions in JavaScript interpretors.

4. Assertion Language

When reasoning about programs using separation logic, it is usual to define an assertion language which describes sets of partial heaps, which may be combined using disjoint union. By describing *partial* heaps, separation logic allows the programmer to focus only on the portion of the heap which is actually affected by a given program. This portion of the heap is called the program's "footprint". Through the use of the frame rule, it is possible to infer the behaviour of a program in a larger context, so long as we know how it behaves on its footprint. In most programming languages, the footprint of a program will be a collection of allocated heap cells. If these cells are present, the program runs predictably, and if they are not, it will fault.

JavaScript is different. The reader will notice that many of the operational rules given in Section 3.3 make use of auxiliary functions such as π and $\uplus$, which are defined using explicit checks of the form $(l, \mathtt{x}) \notin \mathsf{dom}(H)$ and σ, which involves partial sharing of sub-heaps described by the π function. Programs which evaluate using these rules have complex footprints consisting of both allocated cells, and known de-allocated cells. In order to be sure that the program behaves predictably, we must know that certain cells are *not* in the heap. In order to manage sharing, we introduce a new logical operator $\bowtie$, described below. In order to make positive statements about the non-existence of certain cells in a JavaScript heap, we define an abstract value $\varnothing$ so that the notation $(l, \mathtt{x}) \mapsto \varnothing$ indicates that the cell $(l, \mathtt{x})$ is not allocated in the heap. Accordingly, we introduce the concept of an *abstract heap*, which is defined like the JavaScript heap of Section 3.3 but where each cell can store either a JavaScript value v or the symbol $\varnothing$. We also define an evaluation $\lfloor\ \rfloor$ which takes an abstract heap to a JavaScript heap.

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

Note that $(l, x) \mapsto \varnothing * (l, x) \mapsto 4$ is undefined. If the footprint of a program contains $(l, x) \mapsto \varnothing$, then it will be impossible to frame on any other heap containing (l, x). In this way, we can make positive statements about the absence of concrete heap cells.

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 any JavaScript term, return value, $\varnothing$ or list. Lists L may be nested. We also define logical expressions E, which are different from program expressions in that they can't 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: $[\![E]\!]^L_\epsilon$.

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

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

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

$$\frac{[\![E_2]\!]^L_\epsilon = Ls}{[\![E_1{:}E_2]\!]^L_\epsilon \triangleq [\![E_1]\!]^L_\epsilon{:}Ls} \qquad \frac{[\![E_1]\!]^L_\epsilon = l' \wedge [\![E_2]\!]^L_\epsilon = \mathtt{x}}{[\![E_1{\cdot}E_2]\!]^L_\epsilon \triangleq [\![E_1]\!]^L_\epsilon{\cdot}[\![E_2]\!]^L_\epsilon}$$

$$\frac{[\![E_1]\!]^L_\epsilon = n \wedge [\![E_2]\!]^L_\epsilon = n'}{[\![E_1 \oplus E_2]\!]^L_\epsilon \triangleq n \mathbin{\overline{\oplus}} n'} \qquad \frac{[\![E_1]\!]^L_\epsilon = \mathtt{x}}{[\![\lambda E_1.E_2]\!]^L_\epsilon \triangleq \lambda[\![E_1]\!]^L_\epsilon{\cdot}[\![E_2]\!]^L_\epsilon}$$

The assertions of our assertion language include standard boolean assertions, expression equality, set assertions, and quantifiers.

Assertions.

$P ::=$	$P \wedge P \mid P \vee P \mid \neg P \mid \mathrm{true} \mid \mathrm{false}$	Boolean assertions
	$\mid P * P \mid P \mathrel{-\!\!*} P \mid P \bowtie P$	Structural assertions
	$\mid (E, E) \mapsto E \mid \oslash$	JavaScript assertions
	$\mid E = E$	Expression equality
	$\mid E \in \mathrm{SET}$	Set inclusion
	$\mid E \in E$	List element
	$\mid \exists \mathrm{X}.\, P \mid \forall \mathrm{X}.\, P$	Quantification

Notation: $E\ \not\circ E \triangleq \neg(E \circ E)$ for $\circ \in \{=, \in\}$
$E_1 \mathrel{\dot\circ} E_2 \triangleq E_1 \circ E_2 \wedge \oslash$ for $\circ \in \{=, \neq, \in, \notin\}$.

A SET may be a literal set, or a named set such as $\mathcal{X}$, the set of JavaScript field names. In this way we can check the types of JavaScript values when it is feasible. The JavaScript assertions lift abstract heaps into the logic. The structural assertions $*$ and $\mathrel{-\!\!*}$ are standard separation logic assertions, but $\bowtie$ is novel, and deserves further comment. The intuition of $P \bowtie Q$ is that its footprint is the union of the footprints of P and Q, but that sharing of resource is permitted between P and Q. This allows us to reason naturally about complex structures that permit sharing, such as the JavaScript emulated variable store. Note that $P \wedge Q \implies P \bowtie Q$ holds, as does $P * Q \implies P \bowtie Q$. Neither of the reverse implications hold. By convention, the operator $\bowtie$ binds more tightly than $*$.

An assertion P may be satisfied by a triple h, L, ϵ of a heap, stack list, and logical environment. The satisfaction of boolean assertions is straightforward, the other cases are reported below.

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

$$
\begin{array}{ll}
h, L, \epsilon \models P * Q & \iff \exists h_1, h_2.\, h \equiv h_1 * h_2 \wedge \\
& \quad (h_1, L, \epsilon \models P) \wedge (h_2, L, \epsilon \models Q) \\
h, L, \epsilon \models P \mathrel{-\!\!*} Q & \iff \forall h_1.\, (h_1, L, \epsilon \models P) \wedge h \# h_1 \\
& \quad \implies ((h * h_1), L, \epsilon \models Q) \\
h, L, \epsilon \models P \between Q & \iff \exists h_1, h_2, h_3. \\
\multicolumn{2}{l}{\quad h \equiv h_1 * h_2 * h_3 \wedge (h_1 * h_3, L, \epsilon \models P) \wedge (h_2 * h_3, L, \epsilon \models Q)} \\
h, L, \epsilon \models (E_1, E_2) \mapsto E_3 & \iff h \equiv ([\![E_1]\!]^L_\epsilon, [\![E_2]\!]^L_\epsilon) \mapsto [\![E_3]\!]^L_\epsilon \\
h, L, \epsilon \models \oslash & \iff h = \mathtt{emp} \\
h, L, \epsilon \models E_1 = E_2 & \iff [\![E_1]\!]^L_\epsilon = [\![E_2]\!]^L_\epsilon \\
h, L, \epsilon \models E \in \text{SET} & \iff [\![E]\!]^L_\epsilon \in \text{SET} \\
h, L, \epsilon \models E_1 \in E_2 & \iff [\![E_1]\!]^L_\epsilon \text{ is in the list } [\![E_2]\!]^L_\epsilon \\
h, L, \epsilon \models \exists \mathrm{X}.\, P & \iff \exists \mathrm{v}.\, h, L, [\epsilon|\mathrm{X} \leftarrow \mathrm{v}] \models P \\
h, L, \epsilon \models \forall \mathrm{X}.\, P & \iff \forall \mathrm{v}.\, h, L, [\epsilon|\mathrm{X} \leftarrow \mathrm{v}] \models P
\end{array}
$$

Although we have given a direct definition of $\between$ to favour the intuition, when logical variables can range over heaps $\between$ can be derived: $P \between Q \Leftrightarrow \exists \mathrm{R}.\, (\mathrm{R} \mathrel{-\!\!*} P) * (\mathrm{R} \mathrel{-\!\!*} Q) * \mathrm{R}$.

5. Program Reasoning

We give a fault-avoiding program reasoning which is sound with respect to the operational semantics in Section 3.3. Since many JavaScript statements are not local in the traditional sense, we prove soundness using 'weak locality' as introduced in [25]. Our reasoning closely mirrors the operational semantics in all cases except for the usual approximation for `while` and conservative approximations of function call and `eval`. These last two constructs are interesting, and will be the focus of future work as outlined in Section 8. Our fault-avoiding Hoare triples take the form: $\{P\}\mathtt{e}\{Q\}$, which means "if `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 **r**, which is equal to the return value of `e`.

5.1 Auxiliary Predicates

We define predicates to correspond to the functions used by the operational semantics in Section 3.3. The $\sigma(Ls, Sc, Var, L)$ predicate uses $\between$ to closely follow the form of the σ function. It holds precisely when searching for the variable Var in the scope list Sc will traverse the addresses in the list Ls and return the value L (which may be `null`). The σ predicate makes use of the π predicate, just like in the operational semantics. $\pi(Ls, St, Var, L)$ holds precisely when searching for the variable Var in the prototype chain pointed to by St will traverse the addresses in the list Ls and find the variable Var in the object pointed to by L. If L is `null`, then no variable Var can be found. The predicate $\gamma(Ls, Ref, Val)$ holds when the semantic function γ would return a value equivalent to Val if called with the current heap and a value equivalent to Ref after traversing the list of cells in Ls.

Logical Predicates: σ, π, γ.

$$
\begin{array}{l}
\sigma([\,], [\,], \_, \mathtt{null}) \triangleq \oslash \\
\sigma([Ls], St : Sc, Var, St) \triangleq \exists \mathrm{L}.\, \pi(Ls, St, Var, \mathrm{L}) * \mathrm{L} \neq \mathtt{null} \\
\sigma((Ls_1 : Ls_2), St : Sc, Var, L) \triangleq \\
\qquad \pi(Ls_1, St, Var, \mathtt{null}) \between \sigma(Ls_2, Sc, Var, L) \\
\\
\pi([\,], \mathtt{null}, \_, \mathtt{null}) \triangleq \oslash \\
\pi([St], St, Var, St) \triangleq \exists \mathrm{V}.\, (St, Var) \mapsto \mathrm{V} * \mathrm{V} \neq \varnothing \\
\pi((St : Ls), St, Var, L) \triangleq \\
\qquad \exists \mathrm{N}.\, (St, Var) \mapsto \varnothing * (St, @proto) \mapsto \mathrm{N} * \pi(Ls, \mathrm{N}, Var, L) \\
\\
\gamma([\,], Val, Val) \triangleq Val \notin \mathcal{R} \\
\gamma(Ls, L{\cdot}X, \mathtt{undefined}) \triangleq \pi(Ls, L, X, \mathtt{null}) * L \neq \mathtt{null} \\
\gamma(Ls, L_1{\cdot}X, Val) \triangleq \\
\qquad \exists \mathrm{L}_2.\, \pi(Ls, L_1, X, \mathrm{L}_2) \between (\mathrm{L}_2, X) \mapsto Val * Val \neq \varnothing
\end{array}
$$

Notice that while we have taken care with the parameter Ls to precisely determine the footprint of these predicates, the π predicate (and hence the σ predicate which uses it) is inexact in the value of the variable being searched for. Normally, the scope predicate appears σ in conjunction with the de-referencing predicate γ, which compensates for this inexactness.

The inference rules also require logical predicates corresponding to a number of other auxiliary semantic functions. 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 define the defs predicate in [2].The other predicates are reported below.

Auxiliary Predicates

$$
\begin{array}{l}
\mathsf{This}(L{\cdot}\_, L) \triangleq (L, @this) \mapsto \varnothing \quad \text{where } L \neq l_g \\
\mathsf{This}(L{\cdot}\_, l_g) \triangleq \exists \mathrm{V}.\, (L, @this) \mapsto \mathrm{V} * \mathrm{V} \neq \varnothing \\
\\
\mathsf{True}(E) \triangleq E \notin \{0, \text{“”}, \mathtt{null}, \mathtt{undefined}\} \\
\mathsf{newobj}(L, V_1, \ldots, V_n) \triangleq \mathop{*}_{V \in \mathcal{X} \setminus \{V_1 \ldots V_n\}} (L, V) \mapsto \varnothing \\
\\
\mathsf{fun}(F, Closure, Var, Body, Proto) \triangleq \\
\qquad (F, @scope) \mapsto Closure * (F, @body) \mapsto \lambda\, Var.Body * \\
\qquad (F, \mathtt{prototype}) \mapsto Proto * (F, @proto) \mapsto l_{fp} \\
\\
\mathsf{decls}(X, L, \mathtt{e}) \triangleq \mathtt{x1}, \ldots, \mathtt{xn} \quad \text{where } (L, \mathtt{x}i) \in \mathsf{dom}(\mathsf{defs}(X, L, \mathtt{e}))
\end{array}
$$

5.2 Inference Rules

We define below some inference rules $\{P\}\mathtt{e}\{Q\}$ for reasoning about of JavaScript expressions. The full list can be found in [2].

Inference Rules: $\{P\}\mathtt{e}\{Q\}$.

(Definition)
$$\{\mathtt{emp}\}\mathtt{var\ x}\{\mathbf{r} \doteq \mathtt{undefined}\}$$

(Value)
$$\{\oslash\}\mathtt{v}\{\mathbf{r} \doteq \mathtt{v}\}$$

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

(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)
$$\frac{\{P\}\mathtt{e}\{Q * \mathbf{r} \doteq V\} \qquad Q = R * \gamma(Ls, V, L) * L \neq \mathtt{null}}{\{P\}\mathtt{e.x}\{Q * \mathbf{r} \doteq L{\cdot}\mathtt{x}\}}$$

(Computed Access)
$$\frac{\{P\}\mathtt{e1}\{R * \mathbf{r} \doteq V_1\} \qquad R = S_1 * \gamma(Ls_1, V_1, L) * L \neq \mathtt{null} \qquad \{R\}\mathtt{e2}\{Q * X \doteq \mathcal{X}^{\mathrm{U}} * \mathbf{r} \doteq V_2\} \qquad Q = S_2 * \gamma(Ls_2, V_2, X)}{\{P\}\mathtt{e1[e2]}\{Q * \mathbf{r} \doteq L{\cdot}X\}}$$

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

(Binary Operators)
$$\frac{\{P\}\mathtt{e1}\{R * \mathbf{r} \doteq V_1\} \qquad R = S_1 * \gamma(Ls_1, V_1, V_3) \qquad \{R\}\mathtt{e2}\{Q * \mathbf{r} \doteq V_2\} \qquad Q = S_2 * \gamma(Ls_2, V_2, V_4) \qquad V = V_3 \overline{\oplus} V_4}{\{P\}\mathtt{e1} \oplus \mathtt{e2}\{Q * \mathbf{r} \doteq V\}}$$

(Assign Global)
$$\frac{\{P\}\mathtt{e1}\{R * \mathbf{r} \doteq \mathtt{null}{\cdot}X\} \qquad \{R\}\mathtt{e2}\{Q * (l_g, X) \mapsto \varnothing * \mathbf{r} \doteq V_1\} \qquad Q = S * \gamma(Ls, V_1, V_2)}{\{P\}\mathtt{e1 = e2}\{Q * (l_g, X) \mapsto V_2 * \mathbf{r} \doteq V_2\}}$$

(Assign Local)
$$\frac{\{P\}\mathtt{e1}\{R * \mathbf{r} \doteq L{\cdot}X\} \qquad \{R\}\mathtt{e2}\{Q * (L, X) \mapsto V_3 * \mathbf{r} \doteq V_1\} \qquad Q = S * \gamma(Ls, V_1, V_2)}{\{P\}\mathtt{e1 = e2}\{Q * (L, X) \mapsto V_2 * \mathbf{r} \doteq V_2\}}$$

(Function)

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

$$\{\oslash\}\mathtt{function\ (x)\{e\}}\{Q\}$$

(Function Call)

$$\frac{\begin{array}{l}\{P\}\mathtt{e1}\{R_1 * \mathbf{r} \doteq F_1\} \\ R_1 = \left(\begin{array}{l} S_1 \uplus \mathsf{This}(F_1, T) \uplus \gamma(Ls_1, F_1, F_2) * F_2 \not\doteq l_e * \\ (F_2, @body) \mapsto \lambda X.\mathtt{e3} * (F_2, @scope) \mapsto Ls_2 \end{array}\right) \\ \{R_1\}\mathtt{e2}\{R_2 * \mathbf{l} \doteq Ls_3 * \mathbf{r} \doteq V_1\} \quad R_2 = S_2 * \gamma(Ls_4, V_1, V_2) \\ R_3 = \left(\begin{array}{l} R_2 * \exists \mathrm{L}.\, \mathbf{l} \doteq \mathrm{L}{:}Ls_2 * (\mathrm{L}, X) \mapsto V_2 * \\ (\mathrm{L}, @this) \mapsto T * \\ (\mathrm{L}, @proto) \mapsto \mathtt{null} * \mathsf{defs}(X, \mathrm{L}, \mathtt{e3}) * \\ \mathsf{newobj}(\mathrm{L}, @proto, @this, X, \mathsf{decls}(X, \mathrm{L}, \mathtt{e3})) \end{array}\right) \\ \{R_3\}\mathtt{e3}\{\exists \mathrm{L}.\, Q * \mathbf{l} \doteq \mathrm{L}{:}Ls_2\} \qquad \mathbf{l} \notin \mathsf{fv}(Q) \cup \mathsf{fv}(R_2) \end{array}}{\{P\}\mathtt{e1(e2)}\{\exists \mathrm{L}.\, Q * \mathbf{l} \doteq Ls_3\}}$$

(With)

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

(While)

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

(Frame)

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

(Consequence)

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

(Elimination)

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

(Disjunction)

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

Although most of the rules correspond closely to their semantics counterparts, some rules deserve further comment. Rule (Definition) shows the use of the reserved variable **r** to record the result of an expression. Rule (Variable) shows the use of $\uplus$ to express the overlapping footprint of predicates σ and π. Rule (Object), through the predicate newobj, 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 necesarilly 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 semantics of Section 3.3. Since many JavaScript statements are not local according to the standard definition of locality from [17], we use the recently introduced notion of weak locality from [25].

Definition 2 (Soundness of a Hoare triple)**.** A Hoare triple $\{P\}\mathtt{e}\{Q\}$ is sound if, for all h, l, ϵ, it satisfies the following two properties:

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

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

Notice that we are not limited to reasoning about only well-formed heaps. While it is the case that all JavaScript programs maintain the well-formedness of the heap, we are also able to reason about programs that run on partial-heaps, which may not be well-formed.

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

The proof (reported in Section 3 of [2]) involves showing that the predicates used by the language 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

The key to reasoning in an easy and intuitive way about a program is to match the level of abstraction at which it is written. We present a hierarchy of three layers of abstraction, which provide increasingly natural reasoning about well-written programs, while retaining the ability to break the current abstraction and work at a lower level when required.

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 global end of the scope rather than from the local end which makes modifying a variable easier to specify. It makes use of $\neg\sigma$ which says that a variable does not exist in a particular partial scope. The $\neg\sigma^{l_g}$ predicate does the same, but excludes l_g from its footprint, which makes reasoning about global variable instantiation simpler. We prove lemmas such as the equivalence of σ and $\overline{\sigma}$ in Section 4 of [2].

Layer 1 Predicates.

$$\overline{\sigma}(Ls, Sc, Var, \mathtt{null}) \triangleq \neg\sigma(Ls, Sc, Var, \mathtt{null})$$
$$\overline{\sigma}(Ls_1 +\!\!+ (Ls_2 : [\,]), Sc, Var, L) \triangleq \neg\sigma(Ls_1, Sc, Var, L) * \exists \mathrm{L2}.\, \pi(Ls_2, L, Var, L2) * \mathrm{L2} \not\doteq \mathtt{null}$$

$$\neg\sigma([\,], St : Sc, \_, St) \triangleq \oslash$$
$$\neg\sigma(Ls_2 : Ls, St : Sc, Var, End) \triangleq \pi(Ls_2, St, Var, \mathtt{null}) \uplus \neg\sigma(Ls, Sc, Var, End)$$

$$\neg\sigma^{l_g}([\,], St : Sc, \_, St) \triangleq \oslash$$
$$\neg\sigma^{l_g}(Ls_2 : Ls, St : Sc, Var, End) \triangleq \pi^{l_g}(Ls_2, St, Var, \mathtt{null}) \uplus \neg\sigma^{l_g}(Ls, Sc, Var, End)$$

$$\pi^{l_g}([\,], \mathtt{null}, \_, \mathtt{null}) \triangleq \oslash$$
$$\pi^{l_g}([\,], l_g, \_, \mathtt{null}) \triangleq \oslash$$
$$\pi^{l_g}([St], St, Var, St) \triangleq \exists \mathrm{V}.\, (St, Var) \mapsto \mathrm{V} * \mathrm{V} \not\doteq \varnothing$$
$$\pi^{l_g}((St : Ls), St, Var, L) \triangleq \exists \mathrm{N}.\, (St, Var) \mapsto \varnothing * (St, @proto) \mapsto \mathrm{N} * \pi^{l_g}(Ls, \mathrm{N}, Var, L)$$

These predicates give us much more flexibility to reason at a low level about JavaScript variables found in various places in the emulated variable store. At this level, it is possible to prove quite 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 [2].

Simple Assignments.

$$\frac{\begin{array}{l} P = \sigma(L_1 +\!\!+ ((l_g{:}L_2){:}L_3), \mathbf{l}, \mathtt{x}, \mathtt{null}) \\ Q = \left(\begin{array}{c} \exists \mathrm{L}'_1, \mathrm{L}'_3, \mathrm{Sc}, \mathrm{G}.\ \ \neg\sigma^{l_g}(\mathrm{L}'_1, \mathbf{l}, \mathtt{x}, l_g) \uplus \\ \pi(L_2, \mathrm{G}, \mathtt{x}, \mathtt{null}) \uplus \neg\sigma^{l_g}(\mathrm{L}'_3, \mathrm{Sc}, \mathtt{x}, \mathtt{null}) * (l_g, \mathtt{x}) \mapsto \mathtt{v} * \\ (l_g, @proto) \mapsto \mathrm{G} * \mathbf{l} \doteq \_ +\!\!+ (l_g{:}\mathrm{Sc}) * \mathbf{r} \doteq \mathtt{v} \end{array}\right) \end{array}}{\{P\}\mathtt{x = v}\{Q\}}$$

$$\frac{P = \sigma(L_1 \# [L{:}L_2]), \mathbf{l}, \mathtt{x}, L) \bowtie (L, \mathtt{x}) \mapsto \varnothing \bowtie \gamma(L{:}L_2, \mathtt{x}, V) \quad Q = \left(\begin{array}{l} \exists \mathrm{L}'.\, \neg\sigma(L_1, \mathbf{l}, \mathtt{x}, L) * (L, \mathtt{x}) \mapsto \mathtt{v} * (L, @proto) \mapsto Pr * \\ \pi(L_2, Pr, \mathtt{x}, \mathrm{L}') \bowtie (\mathrm{L}', \mathtt{x}) \mapsto V * \mathbf{r} \doteq L{\cdot}\mathtt{x} \end{array}\right)}{\{P\}\mathtt{x = v}\{Q\}}$$

$$\frac{P = \sigma(L_1 \# [[L]], \mathbf{l}, \mathtt{x}, L) \bowtie (L, \mathtt{x}) \mapsto V * V \not\doteq \varnothing \quad Q = \neg\sigma(L_1, \mathbf{l}, \mathtt{x}, L) * (L, \mathtt{x}) \mapsto \mathtt{v} * \mathbf{r} \doteq \mathtt{v}}{\{P\}\mathtt{x = v}\{Q\}}$$

$$\frac{P = \left(\begin{array}{l} \sigma(Ls_1, \mathbf{l}, \mathtt{y}, L_y) \bowtie \gamma(Ls_2, L_y{\cdot}\mathtt{y}, V_y) \bowtie \\ \sigma(L_1 \# ((L{:}[\,]){:}[\,]), \mathbf{l}, \mathtt{x}, L) \bowtie (L, \mathtt{x}) \mapsto V * V \not\doteq \varnothing \end{array}\right) \quad Q = \sigma(Ls_1, \mathbf{l}, \mathtt{y}, L_y) \bowtie \neg\sigma(L_1, \mathbf{l}, \mathtt{x}, L) * (L, \mathtt{x}) \mapsto V_y * \mathbf{r} \doteq V_y}{\{P\}\mathtt{x = y}\{Q\}}$$

Compared to the (Assign -) inference rules, these triples have a clear footprint, and more clearly describe the destructive effects of assignment. Yet, they appear complex and difficult to compose. It would be useful to be also able to ignore some information about the exact structure of the variable store, while retaining the information we care 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.

$$\begin{array}{l} \mathsf{store}_L(X_1 \ldots X_n | X'_1{:}V_1 \ldots X'_m{:}V_m) \triangleq \\ \quad \exists Ls_1 \ldots, Ls_n, Ls'_1, \ldots, Ls'_m, Ls''_1, \ldots, Ls''_m.\ \ \mathsf{thischain}(L) \bowtie \\ \quad \bowtie_{i\in 1..n} \neg\sigma(Ls_i, L, X_i, \mathtt{null}) \quad \bowtie (l_g, X_i) \mapsto \varnothing \\ \quad \bowtie_{j\in 1..m} \sigma(Ls'_j, L, X'_j, L_j) \quad \bowtie_{k\in 1..m}\ \gamma(Ls''_k, L_k{\cdot}X'_k, V_k) \\ \\ \mathsf{thischain}([\,]) \quad \triangleq \quad \varnothing \\ \mathsf{thischain}(L : Sc) \quad \triangleq \quad (L, @this) \mapsto \_ * \mathsf{thischain}(Sc) \end{array}$$

The assertion $\mathsf{store}_{\mathbf{l}}(\mathtt{a}, \mathtt{b} | \mathtt{x} : 1, \mathtt{y} : 2)$ describes a heap emulating a variable store in which the variables a and b are certainly *not* present, and in which the variables x and y take the values 1 and 2 respectively. The subscript **l** says that the variable store being described is the current variable store which the program will access. 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 proved (using previous levels of abstraction) in Section 5 of [2].

We start with rules for variable initialisation and overwrite/override, with a constant and then with the value of a 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 $Q_2 = \mathsf{store}_{\mathbf{l}}(\mathrm{X}_1 \ldots \mathrm{X}_n | \mathtt{x}{:}\mathrm{V}, \mathtt{y}{:}\mathrm{V}, \mathrm{X}'_1{:}\mathrm{V}_1 \ldots \mathrm{X}'_m{:}\mathrm{V}_m)$.

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

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

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

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

One limitation of this level of abstraction is that the abstraction only covers a static (and unknown) list of emulated scope frames. If we call a function which adds a new emulated scope frame to the emulated store, 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 $S = (\mathrm{L}, @proto) \mapsto \mathtt{null} * (\mathrm{L}, \mathtt{x}) \mapsto \varnothing * (\mathrm{L}, \mathtt{y}) \mapsto \mathrm{V}' * \mathbf{l} \doteq \mathrm{L} : \mathrm{LS}$.

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

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

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 lemmas 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

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

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

It may seem surprising that we only provide lemmas 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 higher 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:

```
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 lemmas 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 Section 5.1 of [2].

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

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

$\{\ \mathsf{store_l}(\mathtt{x},\mathtt{y},\mathtt{z},\mathtt{f},\mathtt{v}|)\ \}$
`x = null;y = null;z = null;`
$\{\ \mathsf{store_l}(\mathtt{f},\mathtt{v}|\mathtt{x}:\mathtt{null},\mathtt{y}:\mathtt{null},\mathtt{z}:\mathtt{null}) * \mathrm{true}\ \}$
`f = function(w){x=v ; v=4 ; var v ; y=v};`
$\left\{\begin{array}{l}\exists \mathrm{L}.\ \mathsf{store_l}(\mathtt{v}|\mathtt{x}:\mathtt{null},\mathtt{y}:\mathtt{null},\mathtt{z}:\mathtt{null},\mathtt{f}:\mathrm{L}) * \\ (\mathrm{L},@body) \mapsto \lambda \mathtt{w}.\{\ldots\} * \\ (\mathrm{L},@scope) \mapsto \mathrm{LS} * \mathrm{true}\end{array}\right\}$
`v = 5;`
$\left\{\begin{array}{l}\mathsf{store_l}(|\mathtt{x}:\mathtt{null},\mathtt{y}:\mathtt{null},\mathtt{z}:\mathtt{null},\mathtt{f}:\mathrm{L},\mathtt{v}:5) * \\ (\mathrm{L},@body) \mapsto \lambda w.\{\ldots\} * \\ (\mathrm{L},@scope) \mapsto \mathrm{LS} * \mathrm{true}\end{array}\right\}$
`f(null);`
$\left\{\begin{array}{l}\exists \mathrm{L}'.\ \mathsf{store_l}(|\mathtt{x}:\mathtt{undefined},\mathtt{y}:4,\mathtt{z}:\mathtt{null},\mathtt{f}:\mathrm{L},\mathtt{v}:5) * \\ \mathsf{newobj}(\mathrm{L}',@proto,\mathtt{w},\mathtt{v}) * (\mathrm{L}',@proto) \mapsto \mathtt{null} * \\ (\mathrm{L}',\mathtt{w}) \mapsto \mathtt{null} * (\mathrm{L}',\mathtt{v}) \mapsto 4 * \mathrm{true}\end{array}\right\}$
[Frame]
$\{\ \mathsf{store_l}(|\mathtt{x}:\mathtt{undefined},\mathtt{y}:4,\mathtt{z}:\mathtt{null},\mathtt{f}:\mathrm{L},\mathtt{v}:5)\ \}$
`z = v;`
$\{\ \mathsf{store_l}(|\mathtt{x}:\mathtt{undefined},\mathtt{y}:4,\mathtt{z}:5,\mathtt{f}:\mathrm{L},\mathtt{v}:5) * \mathrm{true}\ \}$
[Frame]
$\left\{\begin{array}{l}\mathsf{store_l}(|\mathtt{x}:\mathtt{undefined},\mathtt{y}:4,\mathtt{z}:5,\mathtt{f}:\mathrm{L},\mathtt{v}:5) * \\ \mathsf{newobj}(\mathrm{L}',@proto,\mathtt{w},\mathtt{v}) * (\mathrm{L}',@proto) \mapsto \mathtt{null} * \\ (\mathrm{L}',\mathtt{w}) \mapsto \mathtt{null} * (\mathrm{L}',\mathtt{v}) \mapsto 4 * \mathrm{true}\end{array}\right\}$
[Cons/Var Elim]
$\{\ \exists \mathrm{L}.\ \mathsf{store_l}(|\mathtt{x}:\mathtt{undefined},\mathtt{y}:4,\mathtt{z}:5,\mathtt{f}:\mathrm{L},\mathtt{v}:5) * \mathrm{true}\ \}$

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

Figure 2. A Proof of the Variable Scopes Program

`a = {b:2}; f(null)`

This program demonstrates the importance of modeling `with` correctly. Notice that when correctly modeled, the closure of the function `f` will refer to the object `{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 `{b:1}`. Thus the program normally returns the value 1, not 2.

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 $\{P\}\mathtt{b}\{P * \mathbf{r} \doteq 1\}$, where P is

$$\left(\begin{array}{l}\exists \mathrm{LS},\mathrm{L},\mathrm{F},\mathrm{L}',\mathrm{LOC}.\ \mathbf{l} \doteq \mathrm{LOC} : \mathrm{L} : \mathrm{LS} * \\ \mathsf{store_{LS}}(|\mathtt{a}:\mathrm{L}',\mathtt{f}:\mathrm{F}) \bowtie (l_{op},\mathtt{f}) \mapsto \varnothing \bowtie \\ (l_{op},@proto) \mapsto \mathtt{null} * \mathrm{true} * \\ (\mathrm{L},\mathtt{b}) \mapsto 1 * (\mathrm{L},\mathtt{f}) \mapsto \varnothing * (\mathrm{L},@proto) \mapsto l_{op} * \\ (\mathrm{L}',\mathtt{b}) \mapsto 2 * (\mathrm{L}',\mathtt{f}) \mapsto \varnothing * (\mathrm{L}',@proto) \mapsto l_{op} * \\ (\mathrm{F},@body) \mapsto \lambda \mathtt{w}.\{\mathtt{b}\} * (\mathrm{F},@scope) \mapsto \mathrm{L} : \mathrm{LS} * \\ (\mathrm{LOC},\mathtt{b} \mapsto \varnothing * (\mathrm{LOC},@proto) \mapsto \mathtt{null}\end{array}\right)$$

For space reasons we reason here about only 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 more general precondition:

$$\left\{\begin{array}{l}\mathsf{store_l}(\mathtt{a},\mathtt{f}|) \bowtie (l_{op},\mathtt{f}) \mapsto \varnothing \bowtie (l_{op},@proto) \mapsto \mathtt{null} \vee \\ \mathsf{store_l}(\mathtt{f}|\mathtt{a}:\_) \bowtie (l_{op},\mathtt{f}) \mapsto \varnothing \bowtie (l_{op},@proto) \mapsto \mathtt{null} \vee \\ \mathsf{store_l}(\mathtt{a}|\mathtt{f}:\_) \bowtie (l_{op},\mathtt{f}) \mapsto \varnothing \bowtie (l_{op},@proto) \mapsto \mathtt{null} \vee \\ \mathsf{store_l}(|\mathtt{a}:\_,\mathtt{f}:\_) \bowtie (l_{op},\mathtt{f}) \mapsto \varnothing \bowtie (l_{op},@proto) \mapsto \mathtt{null}\end{array}\right\}$$

Let $P = (\mathrm{L},\mathtt{b}) \mapsto 1 * (\mathrm{L},@proto) \mapsto l_{op} * \mathrm{true}$
$\{\ \mathsf{store_l}(\mathtt{a},\mathtt{f}|) \bowtie (l_{op},\mathtt{f}) \mapsto \varnothing \bowtie (l_{op},@proto) \mapsto \mathtt{null}\ \}$
`a = {b:1};`
$\left\{\begin{array}{l}\exists \mathrm{L}.\ \mathsf{store_l}(\mathtt{f}|\mathtt{a}:\mathrm{L}) \bowtie (l_{op},\mathtt{f}) \mapsto \varnothing \bowtie \\ (l_{op},@proto) \mapsto \mathtt{null} * (\mathrm{L},\mathtt{f}) \mapsto \varnothing * P\end{array}\right\}$
`with (a){`
$\left\{\begin{array}{l}\exists \mathrm{LS},\mathrm{L}.\ \mathbf{l} \doteq \mathrm{L} : \mathrm{LS} * \\ \mathsf{store_{LS}}(\mathtt{f}|\mathtt{a}:\mathrm{L}) \bowtie (l_{op},\mathtt{f}) \mapsto \varnothing \bowtie \\ (l_{op},@proto) \mapsto \mathtt{null} * (\mathrm{L},\mathtt{f}) \mapsto \varnothing * P\end{array}\right\}$
`f=function(c){b}`
$\left\{\begin{array}{l}\exists \mathrm{LS},\mathrm{L},\mathrm{F}.\ \mathbf{l} \doteq \mathrm{L} : \mathrm{LS} * \\ \mathsf{store_{LS}}(|\mathtt{a}:\mathrm{L},\mathtt{f}:\mathrm{F}) \bowtie (l_{op},\mathtt{f}) \mapsto \varnothing \bowtie \\ (l_{op},@proto) \mapsto \mathtt{null} * (\mathrm{L},\mathtt{f}) \mapsto \varnothing * \\ (\mathrm{F},@body) \mapsto \lambda \mathtt{w}.\{\mathtt{b}\} * (\mathrm{F},@scope) \mapsto \mathrm{L} : \mathrm{LS} * P\end{array}\right\}$
`};`
$\left\{\begin{array}{l}\exists \mathrm{LS},\mathrm{L},\mathrm{F}.\ \mathbf{l} \doteq \mathrm{LS} * \\ \mathsf{store_{LS}}(|\mathtt{a}:\mathrm{L},\mathtt{f}:\mathrm{F}) \bowtie (l_{op},\mathtt{f}) \mapsto \varnothing \bowtie (l_{op},@proto) \mapsto \mathtt{null} * \\ (\mathrm{L},\mathtt{f}) \mapsto \varnothing * (\mathrm{F},@body) \mapsto \lambda \mathtt{w}.\{\mathtt{b}\} * (\mathrm{F},@scope) \mapsto \mathrm{L} : \mathrm{LS} * P\end{array}\right\}$
`a = b:2;`
$\left\{\begin{array}{l}\exists \mathrm{LS},\mathrm{L},\mathrm{F},\mathrm{L}'.\ \mathbf{l} \doteq \mathrm{LS} * \\ \mathsf{store_{LS}}(|\mathtt{a}:\mathrm{L}',\mathtt{f}:\mathrm{F}) \bowtie (l_{op},\mathtt{f}) \mapsto \varnothing \bowtie (l_{op},@proto) \mapsto \mathtt{null} * \\ (\mathrm{L},\mathtt{f}) \mapsto \varnothing * (\mathrm{L}',\mathtt{b}) \mapsto 2 * (\mathrm{L}',\mathtt{f}) \mapsto \varnothing * (\mathrm{L}',@proto) \mapsto l_{op} * \\ (\mathrm{F},@body) \mapsto \lambda \mathtt{w}.\{\mathtt{b}\} * (\mathrm{F},@scope) \mapsto \mathrm{L} : \mathrm{LS} * P\end{array}\right\}$
`f(null)`
$\left\{\begin{array}{l}\exists \mathrm{LS},\mathrm{L},\mathrm{F},\mathrm{L}',\mathrm{LOC}.\ \mathbf{l} \doteq \mathrm{LS} * \\ \mathsf{store_{LS}}(|\mathtt{a}:\mathrm{L}',\mathtt{f}:\mathrm{F}) \bowtie (l_{op},\mathtt{f}) \mapsto \varnothing \bowtie (l_{op},@proto) \mapsto \mathtt{null} * \\ (\mathrm{L},\mathtt{f}) \mapsto \varnothing * (\mathrm{L}',\mathtt{b}) \mapsto 2 * (\mathrm{L}',\mathtt{f}) \mapsto \varnothing * (\mathrm{L}',@proto) \mapsto l_{op} * \\ (\mathrm{F},@body) \mapsto \lambda \mathtt{w}.\{\mathtt{b}\} * (\mathrm{F},@scope) \mapsto \mathrm{L} : \mathrm{LS} * \\ (\mathrm{LOC},\mathtt{b} \mapsto \varnothing * (\mathrm{LOC},@proto) \mapsto \mathtt{null} * P * \mathbf{r} \doteq 1\end{array}\right\}$
$\{\mathbf{r} \doteq 1 * \mathrm{true}\}$

Figure 3. Reasoning about `with`

Notice that even in the more general case, we constrain our precondition with the assertion $(l_{op},\mathtt{f}) \mapsto \varnothing \bowtie (l_{op},@proto) \mapsto \mathtt{null}$. The requirement for this term may seem surprising. Consider running the above program in a state satisfying $\mathsf{store_l}(\mathtt{a},\mathtt{f}|) \bowtie (l_{op},\mathtt{f}) \mapsto 4$ In this case, when the assignment to `f` is made, the function pointer will be written to the cell $(\mathrm{L},\mathtt{f})$, 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 $(l_{op},\mathtt{f})$ 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 'Apps' from system 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 more function calls, another abstraction is called for. We choose to represent 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 local-most 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 $\mathsf{recstore}_L(\mathit{EmptyVars},\mathit{FullVars})$ which describes an abstract variable store $\mathit{FullVars}$, which does not contain the variables in the list $\mathit{EmptyVars}$.

The Recursive recstore **Predicate**

$\mathsf{recstore}_L([\mathtt{x1}',\cdots\mathtt{xm}'],[[\mathtt{x1}=V_1,\cdots,\mathtt{xn}=V_n]]) \triangleq$
$\quad \mathsf{store}_L(\mathtt{x1}',\cdots\mathtt{xm}'|\mathtt{x1}:V_1,\cdots,\mathtt{xn}:V_n)$

$\mathsf{recstore}_{L:LS}([\mathtt{x1}',\cdots\mathtt{xm}'],([\mathtt{x1}=V_1,\cdots,\mathtt{xn}=V_n]:Sc)) \triangleq$

$$\mathsf{recstore}_{LS}([\mathtt{x1}', \cdots \mathtt{xm}'], Sc) * (L, @proto) \mapsto \mathtt{null} * L \not\doteq l_g * *_{i \in 1..m} \ (L, \mathtt{xi}') \mapsto \varnothing *_{j \in 1..n} \ (L, \mathtt{xj}) \mapsto V_j * \mathsf{nones}_L([\mathtt{x1}, \cdots, \mathtt{xn}], Sc)$$

$$\mathsf{nones}_L(\_, [\,]) \triangleq \oslash$$

$$\mathsf{nones}_L(Locs, ([\mathtt{x1} = V_1, \cdots, \mathtt{xn} = V_n] : Sc)) \triangleq *_{i \in 1..n}((\mathtt{xi} \in Locs \wedge \oslash) \vee (\mathtt{xi} \notin Locs \wedge (L, \mathtt{xi}) \mapsto \varnothing)) * \mathsf{nones}_L((\mathtt{x1} : \cdots : \mathtt{xn} : Locs), Sc)$$

Notice that recstore uses the store predicate to constrain the global-most 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 have a `null` prototype, and which are not 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 the accompanying document, the most interesting of which are destructive variable initialisation and update.

Destructive recstore **update**

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

$$\frac{\begin{array}{l} R = \mathsf{recstore}_{\mathsf{l}}((Emps), (Locs \# ((\mathtt{x} = Var) : Curr) \# Globs)) \\ \{R * Globs \not\doteq [\,] * P\}\mathtt{e}\{R * Globs \not\doteq [\,] * Q * \mathbf{r} \doteq Var'\} \\ \mathbf{r} \notin \mathsf{fv}(Q) \\ \forall \mathrm{LS} \in Locs.\, (\mathtt{x} = \_) \notin \mathrm{LS} \\ S = \mathsf{recstore}_{\mathsf{l}}((Emps), (Locs \# ((\mathtt{x} = Var') : Curr) \# Globs)) \end{array}}{\{R * Globs \not\doteq [\,] * P\}\mathtt{x = e}\{S * Globs \not\doteq [\,] * Q * \mathbf{r} \doteq V\}}$$

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 more 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.

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) {
```

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

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

$$\left\{ \begin{array}{l} \exists \mathrm{L}'.\, \mathsf{recstore}_{\mathsf{l}}([\,], [[\mathtt{data} = \mathrm{L}, \mathtt{checkField} = \mathrm{L}', \mathtt{i} = \mathrm{N}], [\,]]) * \\ (\mathrm{L}, \mathtt{numEntries}) \mapsto \mathrm{N} * \\ (\mathrm{L}, 0) \mapsto \mathrm{TXT}_0 * \ldots * (\mathrm{L}, \mathrm{N}) \mapsto \mathrm{TXT}_{\mathrm{N}} * \\ \left(\begin{array}{l} \left(\begin{array}{l} \mathrm{TXT}_0 \not\doteq \text{""} * \ldots * \mathrm{TXT}_{\mathrm{N}} \not\doteq \text{""} * \\ (\mathrm{L}, \mathtt{buttonDisabled}) \mapsto 0 \end{array} \right) \\ \vee (\mathrm{L}, \mathtt{buttonDisabled}) \mapsto 1 \end{array} \right) \end{array} \right\}$$

Figure 4. The Specification of `checkForm`

```
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

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 derived from the semantics of Maffeis, Mitchell and Taly [15]. 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 [17], who introduced separation logic for reasoning about C-programs, and on the work of Parkinson and Bierman [21], who adapted separation logic to reason about Java. We made several adaptations to their work in order to reason about JavaScript. As in [20], 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's *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 ⋈ 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, 18, 30], but they all seem to take a different approach to our our ⋈ connective.

Most work on separation logic proves soundness by requiring that commands are *local*. Javascript commands are inherently non-local, since their behaviour changes depending on where the program variables reside in the JavaScript's emulated variable store. We base our soundness result on *weak locality*, recently introduced by Smith in his PhD thesis [25]. At a similar time, Vafeiadis proved soundness of concurrent separation logic [29], using an elegant technique which does not rely on traditional locality. This technique differs from Smith's in that it does not aim to be compatible with existing locality proofs. Smith's technique allows the re-use of existing locality proofs when available.

We prove our soundness result with respect to a big-step operational semantics of JavaScript derived from the one of Maffeis *et al.* [15]. They define a small-step operational semantics of the complete ECMAScript 3 language, at the same level of abstraction where a JavaScript programmer reasons. In contrast, [12] provide a definitional interpreter of JavaScript written in ML, which has the advantage of being directly executable, but includes im-

plementation details that obscure the semantic rules. Elsewhere, Guha *et al.* [11] compile JavaScript to an intermediate Scheme-like language. Their approach helps defining type-based analyses on the object language, but does not enjoy the one-to-one correspondence between semantic-rules and inference-rules exploited by our approach. Moreover, in some cases the compilation-phase introduces a loss of precision (for example in the case of the `with` construct). There are also a number of more abstract models of JavaScript, which have proven useful to study selected language features [1, 27, 32], but that are not sufficiently concrete for our purpose. Overall, we have chosen the semantics in [15] because it appears to the most faithful to the actual JavaScript semantics. As Richards *et al.* argue in [23], 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 faithful to the ECMAScript standard. We have adapted separation logic to reason about JavaScript subset, modelling many complex features, such as for example 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 [5, 10, 24].

Due to our choice of operational semantics, we have been able to prove a strong soundness result. All syntactically correct library code, proved using our reasoning to be correct with respect to their specifications, will be well behaved, even when called by arbitrary JavaScript code possibly containing features not currently included in our semantics. Also, 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 through short snippets of code that JavaScript is fiendish to understand, and our reasoning can help. The `with` example in Section 6.2 shows a potential bug that could easily go unnoticed for some time, whilst leading to security holes in sanitised mashup environments such as Facebook Apps. Despite the complexity of the language and the subtlety of the bug, reasoning about this and other examples is made surprisingly simple by our 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 [28] defines a type-safe DOM API, and Smith [25] develops a context-logics for reasoning about DOM Core Level 1. It would be valuable to integrate these approaches to DOM modelling with the JavaScript reasoning presented here. We intend to develop reasoning for higher level libraries such as jQuery, Prototype.js and Slidy. This high level reasoning about JavaScript libraries will take the idea of our layers of abstraction to the next level 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.

References

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

[2] Anonymous. Accompanying document for POPL 2012 paper #190. Attached. 3.1, 3.3, 3.3, 3.4, 5.1, 5.2, 5.3, 6.1, 6.2

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

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

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

[6] R. Chugh, J. Meister, R. Jhala, and S. Lerner. Staged information flow for javascript. In *Proc. of PLDI 2009*, pages 50–62. ACM, 2009. 1

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

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

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

[14] S. H. Jensen, A. Møller, and P. Thiemann. Type analysis for JavaScript. In *Proc. of SAS '09*, volume 5673 of *LNCS*, 2009. 1

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

[16] S. Maffeis and A. Taly. Language-based isolation of untrusted Javascript. In *Proc. of CSF'09*, IEEE, 2009. 1

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

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

[19] D. Sands P. Phung and A. Chudnov. Lightweight self protecting JavaScript. In *ASIACCS 2009*. ACM Press, 2009. 1

[20] M. Parkinson. When separation logic met java (by example). FTfJP 2006. 7

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

[22] 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. 2

[23] G. Richards, S. Lebresne, B. Burg, and J. Vitek. An analysis of the dynamic behavior of JavaScript programs. In *PLDI*, 2010. 1, 2, 2, 7

[24] 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. 8

[25] G. D. Smith. Local reasoning about web programs. PhD Thesis, Dep. of Computing, Imperial College London, 2011. 1, 5, 5.3, 7, 8

[26] 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

[27] P. Thiemann. Towards a type system for analyzing javascript programs. In *Proc. of ESOP'05*, volume 3444 of *LNCS*, 2005. 1, 7

[28] P. Thiemann. A type safe DOM API. In *Proc. of DBPL*, pages 169–183, 2005. 8

[29] V. Vafeiadis. Concurrent separation logic and operational semantics. In *MFPS11*, 2011. 7

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

[31] 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. 1

[32] D. Yu, A. Chander, N. Islam, and I. Serikov. JavaScript instrumentation for browser security. In *Proc. of POPL'07*, 2007. 1, 7