Towards An Axiomatic Verification System for JavaScript

Shengchao Qin[1] Aziem Chawdhary[1] Wei Xiong[2] Malcolm Munro[2] Zongyan Qiu[3] Huibiao Zhu[4]
[1]Teesside University [2]Durham University [3]Peking University [4]East China Normal University

Abstract

JavaScript as a Web scripting language has been widely used following the fast growth of Internet. Due to the flexible and dynamic features offered by the JavaScript language, it has become a challenging problem to statically reason about code written in JavaScript. As a first step towards building a mechanised verification system for JavaScript, we present, in this paper, an axiomatic verification system for a core subset of JavaScript based on a variant of separation logic. We have also defined a big-step operational semantics with respect to which we have demonstrated the soundness of our verification system.

1. Introduction

The Internet has been growing extremely fast supported by various web applications. JavaScript as a web scripting language has been widely used over the world. As the "programs" written in JavaScript run on the client platform, the correctness and harmlessness of JavaScript applications are extremely concerned both by the developers and the end users. JavaScript has many dynamic features for allowing fascinating activities for the web page design. For example, JavaScript permits the programs to vary their object structures and behaviors in many ways in the run-time of an application, that may make the dynamic behaviors of a JavaScript program extremely hard to understand and reason about.

To better understand the JavaScript semantics and reason about behaviours of its programs, it is crucial to have a formal semantics and a static verification framework. To the best of our knowledge, this has been an open research problem. To address this challenge, many formal attempts have been carried on, such as operational semantics [1], type analyses [2], staged information flow analysis [3], FRW (filter, rewriting and wrapper) isolation [4]. However, there has yet to be a formal logic system to reason about JavaScript. One reason for this is the tricky and complex semantics of JavaScript. Nevertheless, recent progress has shown that an interesting subset of the language can be formalised in an elegant manner [5].

In this work we make a start on defining a program logic for a (core) subset of JavaScript, based on a variant of separation logic. Separation logic [6], as an extension of Hoare logic, has been used to facilitate reasoning about imperative programs manipulating shared mutable data structure [7]. Separation logic has also been extended to support Java-like languages with only single inheritance allowed [8], [9]. The motivation for this paper is to use the compositional ideas from separation logic, and to build a separation logic-based verification framework for JavaScript.

Due to the existence of a lot of complex and dynamic features in JavaScript, it is a challenging and tedious process to define a logic for the entire language. As a first step of the study, it would be better if we focus on a core subset of the language. In this paper, we focus on prototypal inheritance and JavaScript intricate object semantics, because they are essential in making JavaScript a distinct language from the others. We design a small JavaScript-like language, named JS_{sl}, present a concise operational semantics for it, and then develop a separation logic-based inference framework for reasoning about the programs written in this language. To the best of our knowledge, this is the first axiomatic separation logic style verification system for JavaScript-like languages. We have also shown that our separation logic-based verification system is sound with respect to the operational semantics.

The remainder of this paper is organised as follows. Section 2 depicts the core subset of JavaScript that we use for the presentation, with its operational semantics given in Section 3. The verification system is presented in section 4, where we also present the soundness of the verification system and illustrate some rules via an example. Related work and concluding remarks then follow afterwards.

2. The Language JS_{sl}

We first define a simple language with characterized JavaScript features, including prototypal inheritance, functions as objects, automatic object-amplifying with assignment to new field, etc. The abstract syntax of JS_{sl} is given in Fig 1. The main features of the language are as follows.

- Functions are treated as objects.
- Object fields[1] can be created dynamically on the fly via a field mutation command.
- It supports prototypical inheritance.
- It has a constant cproto which refers to a global object.
- It follows the good parts of JavaScript [10].

We go through some of the language constructs:

- The command $x = \{f_1{:}e_1, .., f_n{:}e_n\}$ creates a new object x with an object literal $\{f_1{:}e_1, .., f_n{:}e_n\}$. It sets the object x's fields $f_1, .., f_n$ to $e_1, .., e_n$, respectively. Note that, for simplicity of presentation, we do not include function declarations as expressions, but we allow function declarations to appear in objection literals (as

1. In JavaScript, concept "property" is used instead of "field". We use *field* in conforming to the common terminology in OO area.

in the example given in Sec 4). The command $x = \{\mathit{fld}{:}\mathbf{func}\ f\ (x_1, ..., x_n)\ \{c\}\}$ is an abbreviation of $y = \mathbf{func}\ f\ (x_1, ..., x_n)\ \{c\}; x = \{\mathit{fld}{:}y\}$.

- Functions here are objects, therefore can be assigned to a variable. In a function declaration, the function name f is optional in JavaScript, as well as in JS_{sl}.
- Prototypical inheritance is achieved by **new** command, where $x = \mathbf{new}\ x'$ creates a new object x with its prototype field set as its parent object x'. Similarly, $x = \mathbf{new}\ x'(e_1, .., e_n)$ creates a new object x by calling the function x'. Note that the argument list $e_1, .., e_n$ can be empty, e.g. in the case when the function x' does not need any parameters.
- There are two forms of function calls: directly calling a function or calling a function through an object. There is a subtle difference between the two forms. Calling through an object ($x = x'.f(e_1, .., e_n)$) implies that **this** in the function refers to the receiver object (x'); whereas when calling directly a function ($x = f(e_1, .., e_n)$), **this** refers to the global object (OProto).
- The field lookup command $x = x'.f$ works by searching the current object for the field; if it cannot be found, it searches along the prototypal chain. In the case that the lookup procedure reaches the global object OProto and the field is not there, it returns an undefined value undef.
- For the field mutation command, $x.f = e$, if the field f is already in the object record referred to by x, it simply updates the value of f to e. If the field f does not exist in the object record referred to by x, it is added to the record dynamically with value e. This happens even if f exists in the parent object of x (prototype object), in which case it looks like an arbitrary "overriding". Therefore, the relationship between a prototype object and a child object can be diversified.
- For simplicity of presentation, we do not include local variable declarations **var** x. Therefore, variables in the programs are all considered as global variables except for function parameters.

3. Semantics

In this section, we present a big-step operational semantics for JS_{sl}. Inspired by [11] we will modify the usual definition of heaps used in separation logic.

3.1. Semantic Domains

The basic elements include values of the primitive types, which are integers, booleans, strings. We also have an undefined value **undef**, and a null value **null**.

$$\mathbf{Prim} = \mathbf{Int} \uplus \mathbf{Bool} \uplus \mathbf{Str} \uplus \{\mathbf{undef}, \mathbf{null}\}$$

A value is either a primitive value or a location, which is a subset of the integers.

$$\mathbf{Value} = \mathbf{Prim} \cup \mathbf{Loc}$$

e	::=	$const$	constants
	|	$p(e_1, ..., e_n)$	arithmetic ops
	|	x	variable
c	::=	**skip**	skip statement
	|	$x = e$	assignment
	|	$x = x'.f$	field lookup
	|	$x.f = e$	field mutation
	|	return e	return
	|	$x = \mathbf{func}\ f\ (x_1, ..., x_n)\ \{c\}$	function object
	|	$x = x'.f(e_1, ..., e_n)$	function call
	|	$x = f(e_1, ..., e_n)$	function call
	|	$x = \{f_1 : e_1, ..., f_n : e_n\}$	object literal
	|	$x = \mathbf{new}\ x$	object creation
	|	$x = \mathbf{new}\ x(e_1, .., e_n)$	obj creation via fun
	|	$c; c$	sequencing
	|	if (b) c else c	condtional
	|	while(b) c	iteration

Fig. 1. Syntax of JS_{sl}

As in separation logic, a program state is a pair of store and heap. The store is simply a mapping from variables to values.

$$s \in \mathbf{Store} = \mathbf{Var} \to \mathbf{Value}$$

A heap is then a (partial) mapping from locations to records.

$$h \in \mathbf{Heap} = \mathbf{Loc} \rightharpoonup \mathbf{Record}$$

Where a record is a reification of an object in the heap. A record contains a number of fields with associated values. We assume there is a set of field names **Field**. A record for an object is then a finite mapping from field names to their contents. We adopt a direct written-form for records as $[n_1 : v_1, \ldots]$. In addition, the record for each object has a field named @proto whose value is a reference to the record representing the prototype of this object. We assume a special constant cproto and its location $\mathsf{loc_{op}}$. It refers to the global object OProto (which is called Object.Prototype in JavaScript).

In JavaScript, functions are also objects and can be stored in objects as fields. To accommodate this, we introduce a subcategory $\mathbf{Func} \subset \mathbf{Record}$ for function objects where each function object contains exact three fields, with the field name as **body**, **args**, and @proto:

$$\begin{aligned}\mathbf{Func} \quad = \quad & \{[\mathbf{body} : c, \mathbf{args} : (x_1, .., x_n), @\mathsf{proto} : \mathsf{loc_{op}}] \\ & \mid c \in \mathbf{ProcBody} \wedge n \in \mathbb{N}\}\end{aligned}$$

where $(x_1, .., x_n)$ are parameters of the function, and c is a piece of code body belonging to the set **ProcBody**. Here we take the value of @proto as $\mathsf{loc_{op}}$ for every function object for brevity[2]. We use $\mathbb{N}$ to denote the set of natural numbers.

Now we can give the category **Record**:

$$\mathbf{Record} = (\mathbf{Field} \to \mathbf{Value}) \cup \mathbf{Func}$$

2. In JavaScript, the @proto field of every function objects refers to Function.Prototype. Our simplification makes no harm to the semantics.

A state (s, h) is simply a pair of a store and a heap.

$$(s, h) \in \mathbf{State} = \mathbf{Store} \times \mathbf{Heap}$$

3.2. Operational Semantics

The operational semantics defined below uses some ideas from the existing work for semantics of JavaScript [1], [2], [5]. In considering the semantics, we always suppose the program is well-formed.

The semantics of an expression is a value depending only on the store as shown below.

$$[\![e]\!] : \mathbf{Store} \to \mathbf{Value} \cup \{\mathbf{error}\}$$

Note that the evaluation of an expression may result in a normal value (in **Value**) or an error (**error**). The definition is standard and omitted for brevity. We will also use $s(e)$ to denote the value of e under the store s.

We define a big-step semantics where each transition rule is of the forms

$$c, (s, h) \to (s', h') \quad \text{or}$$
$$c, (s, h) \to \bot$$

which says that the execution of c at state (s, h) reaches finally the state (s', h'), or the execution fails into an abortion. We will use σ, probably with prime or subscript, to represent a state or $\bot$ in the semantic rules.

Semantics rules for basic commands are given in Fig 2. Semantics of skip, assignment and return are routine. If the evaluation of an expression (e) fails, the execution of some commands (x=e, return e) fails too. Here we use juxtaposition to represent function overriding, as what for $s[x \mapsto v]$, that is defined as

$$\delta\, \delta'(x) = \begin{cases} \delta'(x) & \text{when } x \in \mathsf{dom}(\delta'), \\ \delta(x) & \text{otherwise.} \end{cases}$$

We use $\delta \backslash S$ to denote a mapping obtained from δ by removing variables in S from its domain. That is, $\mathsf{dom}(\delta \backslash S) = \mathsf{dom}(\delta) \backslash S$ and $(\delta \backslash S)(x) = \delta(x)$, for any $x \in \mathsf{dom}(\delta) \backslash S$.

There are several rules for field lookup. Firstly, if the object has the field that is being looked up, then we bind the value of this field to variable x in the store (rule [op-lookup-field]). In the case that the object referred to by x' does not have field f, we need to traverse the so-called prototypal chain searching for the field. Thus we lookup the prototype of x' and run the lookup with the prototype (rule [op-lookup-proto]). This searching may succeed with a result, or it may end up in the scenario where it reaches the OProto object and the field does not exist in that object, in which case the undefined value will be bound to x (rule [op-lookup-undef]).

The last rule in Fig. 2 defines the semantics for field mutations. If the current object has the field, the value of the field is mutated (rule [op-mutate-field]). JavaScript has a special treatment for the case when the field to be mutated does not exist in the object: a new field with the given name will be created dynamically for the object, and its value is set. This case is also covered by rule [op-mutate-field].

$$\frac{}{\mathsf{skip}, (s, h) \to (s, h)}\ [\text{op-skip}]$$

$$\frac{s(e) = v}{x = e, (s, h) \to (s[x \mapsto v], h)}\ [\text{op-assign}] \qquad \frac{s(e) = \mathbf{error}}{x = e, (s, h) \to \bot}\ [\text{op-assign-abt}]$$

$$\frac{s(e) = v}{\mathsf{return}\ e, (s, h) \to (s[\mathsf{res} \mapsto v], h)}\ [\text{op-return}] \qquad \frac{s(e) = \mathbf{error}}{\mathsf{return}\ e, (s, h) \to \bot}\ [\text{op-return-abt}]$$

$$\frac{x' \in \mathsf{dom}(s) \quad s(x') \in \mathsf{dom}(h) \quad h(s(x'))(f) = v}{x = x'.f, (s, h) \to (s[x \mapsto v], h)}\ [\text{op-lookup-field}]$$

$$\frac{\begin{array}{c} x' \in \mathsf{dom}(s) \quad s(x') \in \mathsf{dom}(h) \quad f \notin \mathsf{dom}(h(s(x'))) \\ h(s(x'))(@\mathsf{proto}) = s(x'') \quad x = x''.f, (s, h) \to \sigma \end{array}}{x = x'.f, (s, h) \to \sigma}\ [\text{op-lookup-proto}]$$

$$\frac{s(x') = \mathsf{loc}_{\mathsf{op}} \quad f \notin \mathsf{fields}(\mathsf{OProto})}{x = x'.f, (s, h) \to (s[x \mapsto \mathsf{undef}], h)}\ [\text{op-lookup-undef}]$$

$$\frac{x' \notin \mathsf{dom}(s) \quad \textbf{or else} \quad s(x') \notin \mathsf{dom}(h)}{x = x'.f, (s, h) \to \bot}\ [\text{op-lookup-field-abt}]$$

$$\frac{x \in \mathsf{dom}(s) \quad s(x) \in \mathsf{dom}(h) \quad h(s(x)) = r}{x.f = e, (s, h) \to (s, h[s(x) \mapsto r[f \mapsto s(e)]])}\ [\text{op-mutate-field}]$$

$$\frac{s(e) = \mathbf{error} \quad \textbf{or else} \quad x \notin \mathsf{dom}(s) \quad \textbf{or else} \quad s(x) \notin \mathsf{dom}(h)}{x.f = e, (s, h) \to \bot}\ [\text{op-mutate-field-abt}]$$

Fig. 2. Semantics for Basic Commands

Note that there are various cases, when the execution of commands fails: The evaluation of an expression gives an error, the variable to be assigned to does not exist, etc. Here we use **or else** to combine several failing cases. Note that A **or else** $B \equiv A \vee (\neg A \wedge B)$. We assume **or else** to be left associative: A **or else** B **or else** $C \equiv (A$ **or else** $B)$ **or else** C.

Semantics for function objects are in Fig. 3. In JavaScript, functions are represented as objects. Thus the rule [op-fun-decl] allocates a new record, and sets the **body** and **args** appropriately. The language has two kinds of function call: calling a function directly, or calling via a field reference from an object. There is a subtlety here with regards to what **this** refers to in the function body. In the case of directly calling a function object, **this** refers to the global object OProto. In the case where we call a function from an object, **this** refers to this calling object. In addition, we also assume that return values from functions are stored in a variable named **result**.

The semantics for directly calling a function is given by rule [op-fun-call-dir]: it looks up the body and parameters, evaluates the arguments and binds them to the parameters before executing the body. Note that in the execution, variable **this** is bound to the global object. Once the function has returned, we assign x the value of the **result** variable.

Calling function via a reference is slightly more complicated as we may need to traverse the prototype hierarchy in order

$$\frac{\ell \notin \mathsf{dom}(h) \quad r = [\mathbf{body} : c, \mathbf{args} : (x_1..x_n), @\mathsf{proto} : \mathsf{loc_{op}}]}{x = \mathbf{func}\ f\ (x_1, ..., x_n)\ \{c\}, (s,h) \to (s[x \mapsto \ell], h[\ell \mapsto r])}\ \textbf{[op-fun-decl]}$$

$$\frac{f \in \mathsf{dom}(s) \quad s(f) \in \mathsf{dom}(h) \quad h(s(f)) = r \quad r \in \mathbf{Func} \quad r(\mathbf{body}) = c \quad r(\mathbf{args}) = (x_1, .., x_n) \quad c, (s[\mathbf{this} \mapsto \mathsf{loc_{op}}, x_1 \mapsto s(e_1), .., x_n \mapsto s(e_n)], h) \to (s', h') \quad s_1 = (s' \backslash \{\mathbf{this}, x_1, .., x_n\})[x \mapsto s'(\mathbf{result})] \quad s_2 = s\ s_1}{x = f(e_1, ..., e_n), (s,h) \to (s_2, h')}\ \textbf{[op-fun-call-dir]}$$

$$\frac{f \notin \mathsf{dom}(s)\ \ \mathbf{or\ else}\ \ s(f) \notin \mathsf{dom}(h)\ \ \mathbf{or\ else}\ \ h(s(f)) = r \quad r \notin \mathbf{Func}\ \ \mathbf{or\ else}\ \ r(\mathbf{body}) = c \quad r(\mathbf{args}) = (x_1, .., x_n) \quad c, (s[\mathbf{this} \mapsto \mathsf{loc_{op}}, x_1 \mapsto s(e_1), .., x_n \mapsto s(e_n)], h) \to \bot}{x = f(e_1, ..., e_n), (s,h) \to \bot}\ \textbf{[op-fun-call-dir-abt]}$$

$$\frac{x' \in \mathsf{dom}(s) \quad s(x') \in \mathsf{dom}(h) \quad h(s(x')) = r \quad f \in \mathsf{dom}(r) \quad r(f) = r' \quad r' \in \mathbf{Func} \quad r'(\mathbf{body}) = c \quad r'(\mathbf{args}) = (x_1, .., x_n) \quad c, (s[\mathbf{this} \mapsto s(x'), x_1 \mapsto s(e_1), .., x_n \mapsto s(e_n)], h) \to (s', h') \quad s_1 = (s' \backslash \{\mathbf{this}, x_1, .., x_n\})[x \mapsto s'(\mathbf{result})] \quad s_2 = s\ s_1}{x = x'.f(e_1, ..., e_n), (s,h) \to (s_2, h')}\ \textbf{[op-fun-call-obj]}$$

$$\frac{x' \notin \mathsf{dom}(s)\ \ \mathbf{or\ else}\ \ s(x') \notin \mathsf{dom}(h)\ \ \mathbf{or\ else}\ \ h(s(x')) = r \quad f \notin \mathsf{dom}(r)\ \ \mathbf{or\ else}\ \ r(f) = r' \quad r' \notin \mathbf{Func}\ \ \mathbf{or\ else}\ \ r'(\mathbf{body}) = c \quad r'(\mathbf{args}) = (x_1, .., x_n) \quad c, (s[\mathbf{this} \mapsto s(x'), x_1 \mapsto s(e_1), .., x_n \mapsto s(e_n)], h) \to \bot}{x = x'.f(e_1, ..., e_n), (s,h) \to \bot}\ \textbf{[op-fun-call-obj-abt]}$$

$$\frac{x' \in \mathsf{dom}(s) \quad s(x') \in \mathsf{dom}(h) \quad h(s(x')) = r \quad f \notin \mathsf{dom}(r) \quad r(@\mathsf{proto}) = s(x'') \quad x = x''.f(e_1, ..., e_n), (s,h) \to \sigma}{x = x'.f(e_1, ..., e_n), (s,h) \to \sigma}\ \textbf{[op-fun-call-proto]}$$

$$\frac{s(x') = \mathsf{loc_{op}} \quad f \notin \mathsf{fields}(\mathsf{OProto})}{x = x'.f(e_1, ..., e_n), (s,h) \to (s[x \mapsto \mathsf{undef}], h)}\ \textbf{[op-fun-undef]}$$

Fig. 3. Semantics for Function Objects and Invocation

$$\frac{s(e_1) = v_1, ..., s(e_n) = v_n \quad r = [f_1 : v_1, ..., f_n : v_n, @\mathsf{proto} : \mathsf{loc_{op}}]}{x = \{f_1 : e_1, ..., f_n : e_n\}, (s,h) \to (s[x \mapsto \ell], h[\ell \mapsto r])}\ \textbf{[op-obj-crt-literal]}$$

$$\frac{s(e_i) = \mathbf{error} \text{ for some } i}{x = \{f_1 : e_1, ..., f_n : e_n\}, (s,h) \to \bot}\ \textbf{[op-obj-crt-literal-abt]}$$

$$\frac{x' \in \mathsf{dom}(s) \quad s(x') \in \mathsf{dom}(h) \quad s(x') = \ell' \quad \ell \notin \mathsf{dom}(h)}{x = \mathbf{new}\ x', (s,h) \to (s[x \mapsto \ell], h[\ell \mapsto [@\mathsf{proto} : \ell']])}\ \textbf{[op-obj-crt-proto]}$$

$$\frac{x' \in \mathsf{dom}(s) \quad s(x') \in \mathsf{dom}(h) \quad s(x') = \ell' \quad \ell \notin \mathsf{dom}(h) \quad h(\ell') \in \mathbf{Func} \quad h(\ell') = r \quad r(\mathbf{body}) = c \quad r(\mathbf{args}) = (x_1, .., x_n) \quad c, (s[\mathbf{this} \mapsto \mathsf{loc_{op}}, x_1 \mapsto s(e_1), .., x_n \mapsto s(e_n)], h) \to (s', h') \quad s_1 = (s' \backslash \{\mathbf{this}, x_1, .., x_n\})[x \mapsto \ell] \quad s_2 = s\ s_1}{x = \mathbf{new}\ x'(e_1, .., e_n), (s,h) \to (s_2, h'[\ell \mapsto [@\mathsf{proto} : \mathsf{loc_{op}}]])}\ \textbf{[op-obj-crt-fun-construct]}$$

$$\frac{x' \notin \mathsf{dom}(s)\ \ \mathbf{or\ else}\ \ s(x') \notin \mathsf{dom}(h)}{x = \mathbf{new}\ x', (s,h) \to \bot}\ \textbf{[op-obj-crt-proto-abt]}$$

Fig. 4. Semantics for Object Creations

$$\frac{c_1, (s,h) \to (s', h') \quad c_2, (s', h') \to \sigma}{c_1; c_2 \to \sigma}\ \textbf{[op-sequential]} \qquad \frac{c_1, (s,h) \to \bot}{c_1; c_2 \to \bot}\ \textbf{[op-sequential-abt]}$$

$$\frac{s(b) = \mathsf{true} \quad c_1, (s,h) \to \sigma}{\mathsf{if}(b)\ c_1\ \mathsf{else}\ c_2, (s,h) \to \sigma}\ \textbf{[op-conditional-true]} \qquad \frac{s(b) = \mathsf{false} \quad c_2, (s,h) \to \sigma}{\mathsf{if}(b)\ c_1\ \mathsf{else}\ c_2, (s,h) \to \sigma}\ \textbf{[op-conditional-false]}$$

$$\frac{s(b) = \mathbf{error}}{\mathsf{if}(b)\ c_1\ \mathsf{else}\ c_2, (s,h) \to \bot}\ \textbf{[op-conditional-abt]} \qquad \frac{s(b) = \mathsf{false}}{\mathsf{while}\ (b)\ c, (s,h) \to (s,h)}\ \textbf{[op-iteration-false]}$$

$$\frac{s(b) = \mathsf{true} \quad c; \mathsf{while}\ (b)\ c, (s,h) \to \sigma}{\mathsf{while}\ (b)\ c, (s,h) \to \sigma}\ \textbf{[op-iteration-true]} \qquad \frac{s(b) = \mathbf{error}}{\mathsf{while}\ (b)\ c, (s,h) \to \bot}\ \textbf{[op-iteration-abt]}$$

Fig. 5. Semantics for Control Structures

to find the real function object to execute. The simple case is that the current calling object has the field with the name which refers to the function (rule [op-fun-call-obj]). In this case we have to follow the reference from the field f in the object referred to by x' in order to get access to the function object. Once we have this, the rule is very similar to [op-fun-call-dir]. To ensure that **this** has a correct reference, we bind it to the calling object r before execute function body c.

In the case that the current calling object does not have the field f, we need to look up its prototype object (rule [op-fun-call-proto]). Of course, this searching may fail when we reach the global object OProto, in this case we return the special undefined object and bind x to it (rule [op-fun-undef]). Here we use fields(OProto) to denote the set of field names of OProto.

Note that the rules for field lookups and function calls all reflect the prototypal inheritance property of JavaScript: the fields and functions in prototypes are inherited by the objects.

Rules in Fig. 4 define the semantics for object creations. When we meet a statement which asks to create an object by giving an object literal, we build a record which defines initial values for each field, especially, the value of field @proto is OProto (rule [op-obj-crt-literal]). An object can also be created by taking an existing object as its prototype, that is called as prototypal inheritance. In this case, we make a new object record whose @proto field is set to the prototype object (rule [op-obj-crt-proto]). Similarly, a function object can be created by taking an existing function object as its prototype (rule [op-obj-crt-fun-construct]). Note that in the latter case, the function x' gets executed in this process.

The semantics of sequential, conditional, and iteration structures are standard, as given in Fig. 5.

4. An Axiomatic System for JS_{sl}

As a first step to support mechanised verification for JavaScript code, we propose in this section a set of inference rules in the style of a separation logic. Separation logic extends Hoare logic with extra operators to facilitate reasoning about heap-allocated data structures. Apart from this, another benefit of using separation logic is that we do not need to carry the whole program state during verification, thanks to the elegant frame rule offered by separation logic. This can lead to better scalability when the system is mechanised.

4.1. Assertion Language

To specify properties for JavaScript code, apart from the usual logical operations from first-order logic, we use operations from separation logic to help specify heap-allocated objects. The syntax for the assertion language *AssnSL* is

$$\begin{array}{lll} P & \in & \mathit{AssnSL} \\ P & ::= & \Pi|\Sigma \\ \Pi & ::= & \mathbf{true} \mid \mathbf{false} \mid x \circledcirc e \mid \Pi \wedge \Pi \mid \Pi \vee \Pi \\ \Sigma & ::= & \mathbf{emp} \mid x \mapsto [f_1 : e_1, .., f_n : e_n] \mid \Sigma * \Sigma \mid \Sigma -\!\!* \Sigma \end{array}$$

Note that x denotes a variable name, e, $e_1, .., e_n$ represent expressions, and $f_1, .., f_n$ denote field names. The $\circledcirc$ denotes a relational operator in $\{=, <, >, \leq, \geq\}$.

Similar to other separation logic based verification systems (e.g. [12], [13]), an assertion ($\Pi|\Sigma$ which denotes $\Pi \wedge \Sigma$) in our system represents a symbolic heap, where the pure part Π denotes heap-insensitive properties and the heap part Σ specifies heap-sensitive properties. The formula $\mathbf{emp}$ denotes an empty heap, and the formula $x \mapsto [f_1 : e_1, .., f_n : e_n]$ denotes a heap-allocated object referred to by x, which contains fields $f_1, .., f_n$ whose values are $e_1, .., e_n$, respectively. The formula $\Sigma_1 * \Sigma_2$ (resp. $\Sigma_1 -\!\!* \Sigma_2$) denotes the separation conjunction (resp. separation implication) of two heap formulae Σ_1 and Σ_2.

JavaScript allows dynamic extension of objects, i.e., new fields can be added to an object on the fly. This means that a singleton heap formula $x \mapsto [f_1 : e_1, .., f_n : e_n]$ may only specify a partial object. To accommodate this, we need to amend slightly the usual memory model for separation logic (with respect to the separation conjunction). The semantics of our assertions, represented by the judgement $s, h \models P$, is defined as follows.

$$\begin{array}{lll} s, h \models \Pi \mid \Sigma & \text{iff} & s \models \Pi \text{ and } s, h \models \Sigma \\ s, h \models \mathbf{emp} & \text{iff} & \mathsf{dom}(h) = \emptyset \\ s, h \models x \mapsto [f_1{:}e_1..f_n{:}e_n] & \text{iff} & \mathsf{dom}(h) = \{s(x)\} \text{ and } h(s(x)) = [f_1 : s(e_1), .., f_n : s(e_n)] \\ s, h \models \Sigma_1 * \Sigma_2 & \text{iff} & \exists h_1, h_2 \cdot h_1 \# h_2 \text{ and } h = h_1 \cdot h_2 \text{ and } s, h_1 \models \Sigma_1 \text{ and } s, h_2 \models \Sigma_2 \\ s, h \models \Sigma_1 -\!\!* \Sigma_2 & \text{iff} & \forall h_1 \cdot (\mathsf{dom}(h_1) \cap \mathsf{dom}(h) = \emptyset \text{ and } s, h_1 \models \Sigma_1) \text{ implies } s, h * h_1 \models \Sigma_2 \end{array}$$

The semantics for pure formulae $s \models \Pi$ is standard and omitted for brevity. Our amendment to the memory model is the definition of heap disjointness. To allow partial objects being specified separately, we relax the usual disjointness definition to what follows:

$$\begin{array}{rll} h_1 \# h_2 \text{ iff} & \mathsf{dom}(h_1) \cap \mathsf{dom}(h_2) = \emptyset \text{ or} \\ & \text{forall } \ell \in \mathsf{dom}(h_1) \cap \mathsf{dom}(h_2) \cdot \\ & \mathsf{dom}(h_1(\ell)) \cap \mathsf{dom}(h_2(\ell)) = \emptyset \end{array}$$

This allows us to represent a partial view of a heap-allocated object in our specifications. This flexibility does not cause any practical problems as our system always maintains a more complete view of an object via normalisation:

$$x \mapsto [f_1{:}e_1..f_n{:}e_n] * x \mapsto [f_{n+1}{:}e_{n+1}..f_{n+m}{:}e_{n+m}] \rightsquigarrow x \mapsto [f_1{:}e_1..f_{n+m}{:}e_{n+m}]$$

$$\frac{}{\{\Pi \mid \Sigma\}\mathbf{skip}\{\Pi \mid \Sigma\}}\ [\text{sl-skip}] \qquad \frac{}{\{\Pi[e/x] \mid \Sigma\}x = e\{\Pi \mid \Sigma\}}\ [\text{sl-assign}]$$

$$\frac{}{\{\Pi[e/\mathbf{result}] \mid \Sigma\}\mathbf{return}\ e\{\Pi \mid \Sigma\}}\ [\text{sl-return}]$$

$$\frac{r = [..., f : v, ...]}{\{\Pi \mid x' \mapsto r\}x = x'.f\{(\exists x \cdot \Pi) \wedge x{=}v \mid x' \mapsto r\}}\ [\text{sl-lookup-field}]$$

$$\frac{x' \notin \mathsf{LV}(\Sigma_1) \quad f \notin \mathsf{dom}(r) \quad r(@\mathsf{proto}) = x'' \quad \{\Pi_1 \mid \Sigma_1\}x = x''.f\{\Pi_2 \mid \Sigma_2\}}{\{\Pi_1 \mid x' \mapsto r * \Sigma_1\}x = x'.f\{\Pi_2 \mid x' \mapsto r * \Sigma_2\}}\ [\text{sl-lookup-proto}]$$

$$\frac{f \notin \mathsf{fields}(\mathsf{OProto})}{\{\Pi \mid x' \mapsto \mathsf{cproto}\}x = x'.f\{\Pi \wedge x{=}\mathsf{undef} \mid x' \mapsto \mathsf{cproto}\}}\ [\text{sl-lookup-undef}]$$

$$\frac{}{\{\Pi \mid x \mapsto r\}x.f = e\{\Pi \mid x \mapsto r[f \mapsto e]\}}\ [\text{sl-mutate-field}]$$

Fig. 6. Axiomatic Rules for Basic Commands

4.2. Inference Rules

Similar to other separation logic based verification systems, we also have the frame rule:

$$\frac{\{P\}\ c\ \{Q\} \quad \mathit{modified}(c) \cap \mathit{vars}(R) = \emptyset}{\{P * R\}\ c\ \{Q * R\}}\ [\text{sl-frame}]$$

Note that $\mathit{modified}(c)$ denotes the program variables modified by *c*, and $\mathit{vars}(R)$ represents the set of free variables in *R*. With the support of this frame rule, when we define other inference rules, we only need to specify in the pre/post-conditions the local state that may be accessed by the code in focus (so called the memory footprint).

Our system also supports rules of consequence which allow the strengthening of precondition and weakening of the postcondition. They are omitted here for brevity.

Inference rules for basic commands are given in Fig. 6. Rules for skip, assignment, and return command are straightforward. The field lookup command is much complicated,

$$\frac{r = [\mathbf{body} : c, \mathbf{args} : (x_1, ..., x_n), @\mathsf{proto} : \mathsf{loc_{op}}]}{\{\Pi \mid \mathbf{emp}\} x = \mathbf{func}\ f\ (x_1, ..., x_n)\ \{c\}\{\Pi \mid x{\mapsto}r\}}\ [\text{sl-fun-decl}]$$

$$\frac{\begin{array}{c}\Sigma \equiv (\Sigma_1 * x' \mapsto [\mathbf{body} : c, \mathbf{args} : (x_1..x_n), @\mathsf{proto{:}loc_{op}}]) \\ \Pi_1 = (\exists \mathbf{this}, x_1, .., x_n \cdot \Pi) \wedge \mathbf{this}{=}\mathsf{cproto} \wedge x_1{=}e_1 \wedge .. \wedge x_n{=}e_n \\ \{\Pi_1 \mid \Sigma\} c \{\Pi_2 \mid \Sigma_2\}\end{array}}{\{\Pi \mid \Sigma\} x = x'(e_1, ..., e_n)\{(\exists \cdot \Pi_2) \wedge x{=}\mathbf{result} \mid \Sigma_2\}}\ [\text{sl-fun-call-dir}]$$

$$\frac{\begin{array}{c}\Sigma \equiv (\Sigma_1 * x' {\mapsto} [f{:}x'', ..] * x''{\mapsto}[\mathbf{body} : c, \mathbf{args} : (x_1..x_n), ..]) \\ \Pi_1 = (\exists \mathbf{this}, x_1, .., x_n \cdot \Pi) \wedge \mathbf{this}{=}x' \wedge x_1{=}e_1 \wedge .. \wedge x_n{=}e_n \\ \{\Pi_1 \mid \Sigma\} c \{\Pi_2 \mid \Sigma_2\}\end{array}}{\{\Pi \mid \Sigma\} x = x'.f(e_1, ..., e_n)\{(\exists x \cdot \Pi_2) \wedge x{=}\mathbf{result} \mid \Sigma_2\}}\ [\text{sl-fun-call-obj}]$$

$$\frac{f \notin \mathsf{fields}(x') \quad x'.@\mathsf{proto} = x'' \quad \{\Pi \mid \Sigma\} x = x''.f(e_1, ..., e_n)\{\Pi_1 \mid \Sigma_1\}}{\{\Pi \mid \Sigma\} x = x'.f(e_1, ..., e_n)\{\Pi_1 \mid \Sigma_1\}}\ [\text{sl-fun-call-proto}]$$

$$\frac{f \notin \mathsf{fields}(\mathsf{OProto}) \quad \Sigma \equiv x' {\mapsto} \mathsf{cproto}}{\{\Pi \mid \Sigma\} x = x'.f(e_1, ..., e_n)\{\Pi \wedge x{=}\mathsf{undef} \mid \Sigma\}}\ [\text{sl-fun-undef}]$$

Fig. 7. Axiomatic Rules for Function and Invocation

we have three inference rules to deal with it, like in the operational semantics. Firstly if current object contains the field, we fetch corresponding value in the object directly, as in rule [sl-lookup-field]. If current object does not contain the field, then rule [sl-lookup-proto] says that we go one step following the @proto chain to the prototype object and attempt to find the field there. If the search reaches the global object, OProto, then it either returns a value (when the field exists in the global object) or returns the undefined value, as described in [sl-lookup-undef]. The function $\mathsf{LV}(\Sigma)$ is defined as follows.

$$\begin{array}{lcl}\mathsf{LV}(\mathbf{emp}) & ::= & \emptyset \\ \mathsf{LV}(x{\mapsto}r) & ::= & \{x\} \\ \mathsf{LV}(\Sigma_1 * \Sigma_2) & ::= & \mathsf{LV}(\Sigma_1) \cup \mathsf{LV}(\Sigma_2)\end{array}$$

The rule for field mutation is fairly standard as [sl-mutate-field]. If the field does not exist, it is added to the record. If it does exist, the value of the field is modified in the record. Both cases are covered by the same rule.

The inference rules related to functions are given in Fig. 7. The rule for function declaration is straightforward: it extends the heap with a new record to represent the function object (rule [sl-fun-decl]).

When calling a function directly, the body and parameters of the function are extracted, and the arguments are bound to the parameters, then we attempt to prove the body. The rule describing this is [sl-fun-call-dir]. In this kind of function invocation **this** denotes the global object.

Calling a function from an object is slightly tricky as in the operational semantics. We have three rules for this too, because we may need to search the prototype chain for the definition. Firstly, if the object has the function (rule [sl-fun-call-obj]) defined, then the function object is accessed, and again we attempt to prove the body. Note that **this** should now refer to the object that initiates the calling, so we need to make a substitution in the pre-condition. If the object does not have the function f then we look up its prototype in rule [sl-fun-call-proto]. If the searching ends at the global object, and it attempts anything not available at the global object, the value representing undefined is returned and assign to variable x (rule [sl-fun-undef]).

$$\frac{r = [f_1 : e_1, ..., f_n : e_n]}{\{\Pi \mid \mathbf{emp}\}\ x = \{f_1 : e_1, .., f_n : e_n\}\ \{\Pi \mid x{\mapsto}r\}}\ [\text{sl-obj-crt-literal}]$$

$$\frac{}{\{\Pi \mid x'{\mapsto}r\}\ x := \mathbf{new}\ x'\ \{\Pi \mid x{\mapsto}[@\mathsf{proto} : x'] * x'{\mapsto}r\}}\ [\text{sl-obj-crt-proto}]$$

$$\frac{\begin{array}{c}\Sigma \equiv (\Sigma_1 * x' \mapsto [\mathbf{body} : c, \mathbf{args} : (x_1..x_n), @\mathsf{proto{:}loc_{op}}]) \\ \Pi_1 = (\exists \mathbf{this}, x_1, .., x_n \cdot \Pi) \wedge \mathbf{this}{=}\mathsf{cproto} \wedge x_1{=}e_1 \wedge .. \wedge x_n{=}e_n \\ \{\Pi_1 \mid \Sigma\} c \{\Pi_2 \mid \Sigma_2\} \\ \Sigma_3 = ((x{\mapsto}\_) {-\!\!*}\ \Sigma_2) * x{\mapsto}[@\mathsf{proto} : \mathsf{cproto}]\end{array}}{\{\Pi \mid \Sigma\} x = \mathbf{new}\ x'(e_1, ..., e_n)\{\exists x \cdot \Pi_2 \mid \Sigma_3\}}\ [\text{sl-obj-crt-fun-construct}]$$

Fig. 8. Axiomatic Rules for Object Creation

$$\frac{\{\Pi \mid \Sigma\} c_1 \{\Pi_1 \mid \Sigma_1\} \quad \{\Pi_1 \mid \Sigma_1\} c_2 \{\Pi_2 \mid \Sigma_2\}}{\{\Pi \mid \Sigma\} c_1; c_2 \{\Pi_2 \mid \Sigma_2\}}\ [\text{sl-sequential}]$$

$$\frac{\{\Pi \wedge b \mid \Sigma\} c_1 \{\Pi_2 \mid \Sigma_2\} \quad \{\Pi \wedge \neg b \mid \Sigma\} c_2 \{\Pi_2 \mid \Sigma_2\}}{\{\Pi \mid \Sigma\} \mathbf{if}\ b\ \mathbf{then}\ c_1\ \mathbf{else}\ c_2 \{\Pi_2 \mid \Sigma_2\}}\ [\text{sl-conditional}]$$

$$\frac{\{\Pi \wedge b \mid \Sigma\} c \{\Pi \mid \Sigma\}}{\{\Pi \mid \Sigma\} \mathbf{while}\ b\ \{c\}\{\Pi \wedge \neg b \mid \Sigma\}}\ [\text{sl-iteration}]$$

Fig. 9. Axiomatic Rules for Control Structures

The inference rules for object creation (via object literal or from a prototype) are given in Fig. 8. The [sl-obj-crt-literal] rule starts with an empty heap and allocates a new object on the heap. JavaScript also supports the creation of a new object via prototypal inheritance (resp. via a function constructor), as signified by the [sl-obj-crt-proto] rule (resp. the [sl-obj-crt-fun-construct] rule). When an object is created this way, a new object is created, and its prototype field is set to the prototype object.

The inference rules for control structures, i.e. sequential composition, conditional and while-loops, are standard as in Hoare logic, as shown in Fig. 9.

4.3. Soundness

We now confirm the soundness of our verification system. We will first give two definitions, then formalise the soundness theorem.

Definition 1 (Validity)**.** A specification $\{P\}c\{Q\}$ is *valid*, denoted $\models \{P\}c\{Q\}$, if for all s, h, if $s, h \models P$ and $c, (s, h) \rightarrow (s', h')$ for some s', h', then $s', h' \models Q$.

Definition 2 (Soundness)**.** A verification system for JS_{sl} is *sound* if all provable specifications are indeed valid, that is, if $\vdash \{P\}c\{Q\}$, then $\models \{P\}c\{Q\}$.

Theorem 1. *The verification system presented earlier in this section is sound.*

As is indicated by Definition 2 above, we need to show that, for any P, c, Q, if $\vdash \{P\}c\{Q\}$, then $\models \{P\}c\{Q\}$. The proof can be accomplished by structural induction over c. Due to space limitation, we omit the proof here.

4.4. Example

In this section a nontrivial example is exhibited to illustrate some characteristic features of the logic.

Here we create first an object contains two fields f1 and f2 with value of 1 and a function named f2 respectively. We have a nested function fn3 in the body of f2 with a conditional statement as the function body. The object is assigned to variable obj. After that, field f2 and undefined field f3 are both alerted (shown on screen). A mutation operation on the field f3 is created and alerted later on. At last the function f2 is called from obj, and the result is assigned to variable res and alerted for completing the program. The code is:

```
<script type="text/javascript">
obj = {
    f1: 1,
    f2: function(n) {
        var fn3 = function() {
            if (n >= 10) { return 2;}
            else { return 3;}
        }
        return fn3();
    }
};
alert(obj.f2(11)); alert(obj.f3);
obj.f3 = 5; alert(obj.f3);
res = obj.f2(1); alert(res);
</script>
```

This example can be written in JS_{sl} below.

```
obj = {
    f1: 1,
    f2: func(n) {
        fn3 = func() {
            if (n >= 10) return 2;
            else return 3;
        }
        x = fn3();
        return x;
    }
};
obj.f3 = 5;
res = obj.f2(1);
```

Note that thee "alert" has no effect in our program, as it only produces an output. We omit it here.

Suppose we need to prove the following specification, with C denoting the above code:

$$\{\mathbf{true} \mid \mathbf{emp}\}$$
$$C$$
$$\{\text{res} = 3 \mid \text{obj} \mapsto [\text{f1} : 1, \text{f2} : O_{\text{f2}}, \text{f3} : 5] * \mathbf{true}\}$$

An outline of the verification in our system is given below. In the proof we use O_{f2} (resp. O_{fn3}) to represent the function object referred to by f2 (resp. fn3) for brevity.

$$\begin{aligned} O_{\text{f2}} \;=\; & [\mathbf{body} : \{\text{fn3} = \mathbf{func}()\{\mathbf{if}(\text{n} >= 10)\mathbf{return}\ 2; \\ & \qquad\qquad \mathbf{else\ return}\ 3;\}; \\ & \qquad \text{x} = \text{fn3}(); \mathbf{return}\ \text{x}\}, \\ & \mathbf{args} : (n), @\mathsf{proto} : \mathsf{OProto}] \\ O_{\text{fn3}} \;=\; & [\mathbf{body} : \mathbf{if}(\text{n} >= 10)\mathbf{return}\ 2; \mathbf{else\ return}\ 3, \\ & \mathbf{args} : (), @\mathsf{proto} : \mathsf{OProto}] \end{aligned}$$

Note that we only outline the main steps of proof structure below. The framed boxes denote the verification of the function bodies which is inlined in the proof for illustration purpose. We can easily introduce function specifications so that each function can be verified separately against its specification at declaration.

$\{\mathbf{true} \mid \mathbf{emp}\}$
obj $= \{\text{f1} : 1, \text{f2} : \ldots\}$;
$\{\mathbf{true} \mid \text{obj} \mapsto [\text{f1} : 1, \text{f2} : O_{\text{f2}}]\}$
obj.f3 $= 5$;
$\{\mathbf{true} \mid \text{obj} \mapsto [\text{f1} : 1, \text{f2} : O_{\text{f2}}, \text{f3} : 5]\}$
res $=$ obj.f2(1)

$\{n{=}1 \mid \text{obj} \mapsto [\text{f1} : 1, \text{f2} : O_{\text{f2}}, \text{f3} : 5]\}$
fn3 $= \mathbf{func}()\{...\}$
$\{n{=}1 \mid \text{obj} \mapsto [\text{f1} : 1, \text{f2} : O_{\text{f2}}, \text{f3} : 5] * \text{fn3} \mapsto O_{\text{fn3}}\}$
x $=$ fn3()

$\{\text{n}{=}1 \mid \text{obj} \mapsto [\text{f1} : 1, \text{f2} : O_{\text{f2}}, \text{f3} : 5] * \text{fn3} \mapsto O_{\text{fn3}}\}$
$\mathbf{if}(\text{n} >= 10)\mathbf{return}\ 2; \mathbf{else\ return}\ 3$
$\{\text{n}{=}1 \wedge \mathbf{result} = 3 \mid \text{obj} \mapsto [\text{f1} : 1, \text{f2} : O_{\text{f2}}, \text{f3} : 5] * \mathbf{true}\}$

$\{\text{n}{=}1 \wedge \text{x} = 3 \mid \text{obj} \mapsto [\text{f1} : 1, \text{f2} : O_{\text{f2}}, \text{f3} : 5] * \mathbf{true}\}$
$\mathbf{return}\ \text{x}$
$\{\mathbf{result} = 3 \mid \text{obj} \mapsto [\text{f1} : 1, \text{f2} : O_{\text{f2}}, \text{f3} : 5] * \mathbf{true}\}$

$\{\text{res}{=}3 \mid \text{obj} \mapsto [\text{f1} : 1, \text{f2} : O_{\text{f2}}, \text{f3} : 5] * \mathbf{true}\}$

5. Related Work

Almost every modern web browser has a JavaScript interpreter for enriching the dynamic web interactions but at the cost of its safety level [14]. The complexity and dynamic features of JavaScript cause to an impending need for having a logic to define the entire language.

On the subtle semantics of JavaScript there are many investigations. Maffeis et al. [1] provide a large operational semantics mainly for the ECMAScript standard language [15]. However, their semantics are too verbose to be applied directly for verification. Guha et al. [5] give an operational semantics for λ_{js} that is taken as a core for JavaScript excluding certain functions such as *eval*, *this*, and *with*. Since separation logic [6], [16], [17] is a promising new approach tailored for specifying and verifying properties of heap allocated data, our semantic model for JS_{sl} is amended from the application of separation logic to the domain of OO languages from [11]. Our semantic model can be a step stone for JavaScript verification in separation logic [16], [18].

Apart from the work on semantic models, researchers have also tried to work on type analysis on JavaScript. The work has developed from early work on type analysis for other languages [19]–[21]. Some work proposed type systems for

JavaScript program static analysis. Thiemann's type framework focused on the abstract domain design and soundness proof [22]. Anderson et al. [23] focused on definite or potential objects with their sub-langauge JS_0. They did not model type modification and absence of property since that are more difficult to predict than recency-based system. In the later work, Heidegger and Thiemann followed up on a recency-based type system for core JavaScript with additional inference algorithm [24]. Jensen, Moller and THiemann presented a sound and precise type analysis system which support full version of ECMAScript262 for catching errors before code loaded in browsers or rather proving the absence of built-in functions and objects [2]. Logozzo and Venter proposed a compiler that can specialise numerical Float64 variables to Int32 variable to improve JavaScript program performance [25]. However, all these type system relies on similar assumptions, such as no further properties are defined after the initialization and that the type of properties rarely changes. These assumptions are crucial for applicability of their results,but this really restricts the applicability of the work to real JavaScript programs. In fact, Richards, Lebresne, Burg and Vitek [26] enumerated nine explicit and implicit assumptions that are commonly found in JavaScript analysis, and provided supporting evidence to either confirm or invalidate these assumptions. Ratanaworabhan et al. [27] concurrently performed similar study and produced similar results as Lebresne and Vitek.

There are some studies on JavaScript's security behaviour [28], [29], but these studies are restricted to particular security properties. Lebresne et al. have explored a small scale of study of JavaScript and their preliminary results [30] are consistent with Richards et al. [26]. ADsafe [10] as a most popular safe subset of JavaScript is powerful enough to allow guest code to perform valuable interactions, while at the same time preventing malicious or accidental damage or intrusion. Moreover, the ADsafe subset can be checked mechanically by tools like JSLint so that no human inspection is necessary to review guest code for safety. However, JSLint is only a JavaScript syntax checker and validator, and ADsafe can be too restrictive a subset. Our work aims to build a static verifier for a less restrictive (hence more flexible) subset of the JavaScript langugage.

Separation logic has been used to formally verify and reason about conventional programming languages such as Java, C++ with reference-based heap models, e.g. [7], [9], [12], [13], [31], [32]. In this work we make a first step towards the verification of the JavaScript language using separation logic.

6. Conclusions

In this paper we have made the first step towards an axiomatic system for the JavaScript language. To simplify the presentation, we have concentrated on a core subset of JavaScript, for which we have created an operational semantics with state transition rules. Based on this, the assertions and their semantics are set up, and the verification rules are formalised. Our verification system is based on a variant of separation logic, which allows a better support for reasoning about heap allocated data. We have also stated the soundness of the verification system, and provided an example for illustration. Possible future works include (1) extending the system to cover more JavaScript features, (2) mechanising the verification system for practical use, and (3) extending the logic with permissions [33] to facilitate reasoning about security properties concerning access permissions for external code. It might be possible to benefit from the recent advances on automated program verification using separation logic (e.g. [12], [13], [31], [32], [34]).

Acknowledgement This work was supported in part by the EPSRC project EP/G042322. We would like to thank the anonymous referees for valuable comments.

References

[1] S. Maffeis, J. C. Mitchell, and A. Taly, "An operational semantics for javascript," *APLAS*, vol. 5356, pp. 307–325, 2008. [Online]. Available: http://www.doc.ic.ac.uk/~maffeis/aplas08.pdf

[2] S. H. Jensen, A. Moller, and P. Thiemann, "Type analysis for javascript," Los Angeles, August 2009, pp. 238–255.

[3] R. Chugh, J. A. Meister, R. Jhala, and S. Lerner, "Staged infomration flow for javascript," *PLDI (SIGPLAN Conference on Programming Language Design and Implementation)*, June 2009.

[4] S. Maffeis, J. C. Mithcell, and A. Taly, "Language-based isolation of untrusted javascript," 2009.

[5] A. Guha, C. Saftoiu, and S. shnamurthi, "The essence of javascript," *ECOOP (European Conference on Object-Oriented Programming)*, 2010.

[6] J. C. Reynolds, "Separation logic: A logic for shared mutable data structures," *Logic in Computer Science, Symposium on*, vol. 0, pp. 55–74, 2002. [Online]. Available: http://dx.doi.org/10.1109/LICS.2002.1029817

[7] C. Luo, G. He, and S. Qin, "A heap model for java bytecode to support separation logic," *The 15th Asia-Pacific Software Engineering Conference(APSEC)*, December 2008.

[8] M. Parkinson, "Local reasoning for java," Ph.D. dissertation, Cambridge University, August 2005.

[9] W. Chin, C. David, H. Nguyen, and S. Qin, "Enhancing modular oo verification with separation logic," *The 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, January 2008.

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

[11] R. Middelkoop, K. Huizing, and R. Kuiper, "A separation logic proof system for a class-based language," in *Proceedings of the Workshop on Logics for Resources, Processes and Programs (LRPP)*, 2004.

[12] C. Calcagno, D. Distefano, P. O'Hearn, and H. Yang, "Compositional shape analysis by means of bi-abduction," in *36th POPL*. Savannah, Georgia, USA: ACM Press, January 2009.

[13] W.-N. Chin, C. David, H. H. Nguyen, and S. Qin, "Automated verification of shape, size and bag properties via user-defined predicates in separation logic," *Science of Computer Programming*, vol. In Press, Corrected Proof, 2010.

[14] K. Chandra, S. S. Chandra, and S. S. Chandra, "A comparison of vbscript, javascript, and jscript," *J. Comput. Small Coll.*, vol. 19, no. 1, pp. 323–335, October 2003. [Online]. Available: http://portal.acm.org/citation.cfm?id=948737.948782

[15] "Final Draft Of Standard ECMA-262 5th Edition ECMAScript Language," 2009. [Online]. Available: http://www.ecma-international.org

[16] S. S. Ishtiaq and P. W. O'Hearn, "Bi as an assertion language for mutable data structures," *SIGPLAN Not.*, vol. 36, no. 3, pp. 14–26, 2001. [Online]. Available: http://dx.doi.org/10.1145/373243.375719

[17] J. Reynolds, "An overview of separation logic," 2005, pp. 460–469. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-69149-5_49

[18] J. C. Reynolds, "Separation logic: A logic for shared mutable data structures," in *Proceedings of the 17th IEEE Symposium on Logic in Computer Science*. IEEE, July 2002, pp. 55–74.

[19] J. O. Graver and R. E. Johnson, "A type system for smalltalk," *In Proc. 17th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL*, pp. 136–150, 1990.

[20] R. Cartwright and M. Fagan, "Soft typing," *In Proc. ACM SIGPLAN Con- ference on Programming Language Design and Implementation (PLDI)*, June 1991.

[21] A. K. Wright and R. Cartwright, "A practical soft type system for scheme," *ACM Transactions on Programming Languages and Systems*, vol. 19, no. 1, pp. 87–152, 1997.

[22] P. Thiemann, "Towards a type system for analyzing javascript programs," *In Proc. Programming Languages and Systems, 14th European Symposium on Pro- gramming, ESOP*, April 2005.

[23] C. Anderson, P. Giannini, and S. Drossopoulou, "Towards type inference for javascript," *In Proc. 19th European Conference on Object-Oriented Programming, ECOOP*, vol. 3586, July 2005.

[24] P. Heidegger and P. Thiemann, "Recency types for dynamically-typed object-based languages," *In Proc. International Workshops on Foundations of Object-Oriented Languages, FOOL*, January 2009.

[25] F. Logozzo and H. Venter, "Rata: Rapid atomic type analysis by abstract interpretation - application to javascript optimization," 2010, pp. 66–83. [Online]. Available: http://research.microsoft.com/pubs/115734/aitypes.pdf

[26] G. Richards, S. Lebresne, B. Burg, and J. Vitek, "An analysis of the dynamic behavior of javascript programs," Toronto, June 2010. [Online]. Available: http://sss.cs.purdue.edu/projects/dynjs/pldi275-richards.pdf

[27] P. Ratanaworabhan, B. Livshits, and B. Zom, "Jsmeter: Comparing the behavior of javascript benchmarks with real web applications," *In USENIX Conference on Web Application Development (WebApps)*, June 2010.

[28] B. Feinstein and D. Peck, "Caffeinemonkey: Automatedcollection, detection and analysis of malicious javascript," *In Black Hat, Las Vegas, USA*, 2007.

[29] C. Yue and H. Wang, "Characterizing insecure javascript practices on the web," *In 18th International World Wide Web Conference*, p. 961, April 2009.

[30] S. Lebresne, G. Richards, J. Ostlund, T. Wrigstad, and J. Vitek, "Understanding the dynamics of javascript," *In Workshop on Script to Program Evolution (STOP)*, 2009.

[31] D. Distefano, P. W. O'Hearn, and H. Yang, "A local shape analysis based on separation logic," in *Proceedings of the 12th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, ser. Lecture Notes in Computer Science, vol. 3920. Vienna: Springer-Verlag, April 2006, pp. 287–302.

[32] H. H. Nguyen, C. David, S. Qin, and W.-N. Chin, "Automated verification of shape and size properties via separation logic." in *8th VMCAI*, ser. LNCS, vol. 4349, 2007.

[33] R. Bornat, C. Calcagno, P. O'Hearn, and M. Parkinson, "Permission accounting in separation logic," *SIGPLAN Not.*, vol. 40, no. 1, pp. 259–270, 2005.

[34] W. Chin, C. David, H. Nguyue, and S. Qin, "Automated verification of shape, size and bag properties," *12th IEEE International Conference on Engineering of Complex Computer Systems (ICECCS)*, pp. 307–320, July 2007.