Towards Logic-based Verification of JavaScript Programs

José Fragoso Santos[1], Philippa Gardner[1], Petar Maksimović[1,2], Daiva Naudžiūnienė[1]

[1] Imperial College London
[2] Mathematical Institute of the Serbian Academy of Sciences and Arts

Abstract. In this position paper, we argue for what we believe is a correct pathway to achieving scalable symbolic verification of JavaScript based on separation logic. We highlight the difficulties imposed by the language, the current state-of-the-art in the literature, and the sequence of steps that needs to be taken. We briefly describe JaVerT, our semi-automatic toolchain for JavaScript verification.

1 Introduction

JavaScript is one of the most widespread languages for Web programming today: it is the de facto language for client-side Web applications; it is used for server-side scripting via Node.js; and it is even run on small embedded devices with limited memory. Standardised by the ECMAScript committee and natively supported by all major browsers, JavaScript is a complex and evolving language.

The ubiquitous use of JavaScript, especially in security-critical contexts, mandates a high level of trust in the written code. However, the dynamic nature of JavaScript, coupled with its intricate semantics, makes the understanding and development of correct JavaScript code notoriously difficult. It is because of this complexity that JavaScript developers still have very little tool support for catching errors early in development, contrasted with the abundance of tools (such as IDEs and specialised static analysis tools) available for more traditional languages, such as C and Java. The transfer of analysis techniques to the domain of JavaScript is known to be a challenging task.

In this position paper, we argue for what we believe is a correct pathway to achieving scalable, logic-based symbolic verification of JavaScript, highlighting the difficulties imposed by the language, the current state-of-the-art in the literature, and the sequence of steps that needs to be taken. Using our approach, we illustrate how to give functionally correct specifications of JavaScript programs, written in a separation logic for JavaScript. We aim to have such specifications be as agnostic as possible to the internals of JavaScript and provide an interface that gives meaningful feedback to the developer. We give a brief description of JaVerT, our semi-automatic toolchain for JavaScript verification.

2 Motivation

We illustrate the complexity of JavaScript by appealing to a JavaScript priority queue library, which uses an implementation based on singly-linked node lists. It is a variation on a Node.js priority queue library that uses doubly linked lists [18], simplified for exposition. We use this example to showcase the intricacies of JavaScript semantics as well as some of the major challenges that need to be addressed before JavaScript programs can be verified.

```
1  /* @id PQLib */
2  var PriorityQueue = (function () {
3
4    /* @id Node */
5    var Node = function (pri, val) {
6      this.pri = pri;
7      this.val = val;
8      this.next = null;
9    }
10
11   /* @id insert */
12   Node.prototype.insert =
13   function (nl) {
14     if (nl === null) {
15       return this
16     }
17     if (this.pri >= nl.pri) {
18       this.next = nl;
19       return this
20     }
21     var tmp = this.insert (nl.next);
22     nl.next = tmp;
23     return nl
24   }
25
26   /* @id PQ */
27   var PQ = function () {
28     this._head = null
29   };
30   /* @id enqueue */
31   PQ.prototype.enqueue =
32   function(pri, val) {
33     if (counter > 42) {
34       throw new Error()
35     }
36     var n = new Node(pri, val);
37     this._head = n.insert(this._head);
38   };
39
40   /* @id dequeue */
41   PQ.prototype.dequeue =
42   function () {
43     if (this._head === null) {
44       throw new Error()
45     }
46     var first = this._head;
47     this._head = this._head.next;
48     counter--;
49     return {pri: first.pri,
50       val: first.val};
51   };
52
53    return PQ;
54 })();
55
56 var q = new PriorityQueue();
57 q.enqueue(1, "last");
58 q.enqueue(3, "bar");
59 q.enqueue(2, "foo");
60 var r = q.dequeue();
```

Fig. 1. A simple JavaScript priority queue library (lines 1-56) and client (lines 58-62). For verification purposes, each function literal is annotated with a unique identifier.

A Priority Queue Library

In Figure 1, we present the priority queue library (lines 1-56) together with a simple client program (lines 58-62). The priority queue is implemented as an object with property `_head` pointing to a singly-linked list of node objects, ordered in descending order of priority. A new priority queue object is constructed using the `PQ` function (lines 28-31), which declares that property `_head` has value `null`, that is, that the queue is initially empty. The `enqueue` and `dequeue` functions (lines 32-53) provide the functionality to enqueue and dequeue nodes of the queue. These functions should be accessible by all priority queue objects. This is accomplished by following the standard JavaScript prototype inheritance paradigm, which, in this case, means storing these two functions within the object `PQ.prototype`.

The `enqueue` function constructs a new node object, and then adds it to the node list in the appropriate place given by its priority. A node object is constructed using the `Node` function (lines 5-11) which declares three properties, a priority, a value and a pointer to the next node in the node list, and increments the variable `counter`, which keeps track of how many nodes were created (lines 3,10) by the library. We limit the number of nodes that a library can create (lines 35-37) to illustrate scoping further. The node object is then inserted into the node list using the `insert` function (lines 13-26) which, again using prototype inheritance, is a property of `Node.prototype` and is accessible by all node objects.

Let us now show how the example actually works. Our first step is to initialise the priority queue library in lines 1-50. This involves: **(1)** setting up the functionalities of node objects (lines 5-26); **(2)** setting up the functionalities of priority queue objects (lines 28-53); and **(3)** providing the interface from the priority queue library to the client (line 55). At this point, the client can construct a new, empty priority queue, by calling `new PriorityQueue()`, and enqueue and dequeue nodes of the queue, by calling the `enqueue` and `dequeue` functions.

We demonstrate how this library can be used via a small client program (lines 58-62). Line 58 constructs an empty queue, identified by the variable `q`. In doing so, the node counter associated with `q` is set to zero, as no nodes have yet been created (line 3). Lines 59-62 call the `enqueue` and `dequeue` functions, for adding and removing elements from the queue. For example, the command statement `q.enqueue(1,"last")` in line 59 inserts a new node with priority `1` and value `"last"` into the (at this point empty) queue `q`. To do so, it first checks if the node limit has been reached and, since the value of the node counter is zero, it proceeds. Next, it uses the `Node` function to construct a new node object (line 38), say `n`, with the given priority (`pri=1`), value (`val="last"`), and a pointer to the next node (initially `next = null`). Finally, it then calls `n.insert(this._head)` (line 39), which inserts `n` into the existing node list at `this._head`, returns the head of the new node list and stores it in `this._head`. In this case, since we are inserting the node `n` into an empty queue, this head of the new node list will be `n`. The statements `q.enqueue(3, "bar")` and `q.enqueue(2, "foo")` behave in a similar way. After their execution, we have a queue containing three elements and the node counter is equal to 3. Finally, the statement `var r = q.dequeue( )` removes the first element from the queue by swinging the `_head` pointer to the second element of the node list, decreases the node counter to 2, creates a new object containing the priority property with value 3 and the value property with value `"bar"`, and returns the address of this new object.

Ideally, it should be possible to abstract the details of `Node` so that the client works with the functionalities of the priority queue. In Java, it is possible to define a `Node` constructor and its associated functionalities to be private. In JavaScript, there is no native mechanism that provides encapsulation. Instead, the standard approach to establish some form of encapsulation is to use function closures. For example, the call of the function `Node` inside the body of `enqueue` (line 38) refers to the variable `Node` declared in the enclosing scope. This makes it impossible for the clients of the library to see the `Node` function and use it directly.

However, they still can access and modify constructed nodes and `Node.prototype` through the `_head` property of the queue, breaking encapsulation. Our goal is to provide specifications of the queue library functions that ensure functionally correct behaviour and behavioural properties of encapsulation.

The Complexity of JavaScript

JavaScript is a highly dynamic language, featuring a number of non-standard concepts and behaviours. In this section, we describe the JavaScript initial heap and elaborate on the challenges that need to be addressed for tractable JavaScript verification to be possible.

Initial Heap. Before the execution of any JavaScript program, an *initial heap* has to be established. It contains the *global object*, which holds all global variables such as `PriorityQueue`, `q` and `r` from the example. It also contains the functions of all JavaScript built-in libraries, widely used by developers: for example, `Object`, `Function` and `Error`. In the example, the `Error` built-in function is used to construct a new error object when trying to dequeue an empty queue (line 36).

Internal Functions. In the ECMAScript standard, the semantics of JavaScript is described operationally, that is, the behaviour of each JavaScript expression and statement is broken down into a number of steps. These steps heavily rely on a wide variety of *internal functions*, which capture the fundamental inner workings of the language; most notably, object property management (e.g. creation (`DefineOwnProperty`), lookup (`GetValue`), mutation (`PutValue`) and deletion (`Delete`)) and type conversions (e.g. `ToString` and `ToNumber`).

To better understand the extent of the use of the internal functions, consider the JavaScript assignment `o["foo"]` = 42. According to its definition in the standard, it calls the internal functions five times: `GetValue` thrice, and `ToString` and `PutValue` once. This, however, is only at top-level: `GetValue`, in turn, calls `Get`, which calls `GetProperty`, which calls `GetOwnProperty` and possibly itself recursively; `PutValue` calls `Put`, which calls `CanPut` and `DefineOwnProperty`, which calls `GetOwnProperty`. In the end, a simple JavaScript assignment will make more than ten and, in some cases, even more than twenty calls to various internal functions. The more complex a JavaScript command is, the greater the number of the internal functions that it calls. Therefore, in order to be able to reason about JavaScript programs, one first has to tackle the internal functions. This brings us to the following challenge:

Challenge: To reason robustly and abstractly about the JavaScript internal functions.

JavaScript Objects. Objects in JavaScript differ C++ and Java objects in several defining ways. First, JavaScript objects are *extensible*, that is, properties can be added and removed from an object after creation. Second, property access in JavaScript is *dynamic*; we cannot guarantee statically which property of the

object will be accessed. Third, JavaScript objects have two types of properties: *internal* and *named*.

Internal properties are hidden from the user, but are critical for the mechanisms underlying JavaScript, such as prototype inheritance. To illustrate, standard objects have three internal properties: `@proto`, `@class`, and `@extensible`. For example, all node objects constructed using the `Node` function have prototype `Node.prototype`, class `"Object"`, and are extensible. JavaScript objects constructed by some of the built-in libraries can have additional internal properties. For example, a `String` object, associated with a string literal, has properties that represent the characters of that literal.

Named properties, which correspond to standard fields of C++ and Java objects, are associated not with values, but instead with *property descriptors*, which are lists of *attributes* that describe the ways in which a property can be accessed or modified. Depending on the attributes they contain, named properties can either be *data properties* or *accessor properties*. Here, we focus on data properties, which have the following attributes: *value*, holding the actual value of the property; *writable*, describing if the value can be changed; *configurable*, allowing property deletion and any change to non-value attributes; and *enumerable*, stating if a property may be used in a `for−in` enumeration. The values of these attributes depend on how the property is created. For example, if a property of an object is created using a property accessor (for example, `this.pri = pri`), then by default it is writable, configurable and enumerable. On the other hand, if a property is declared as a variable, then by default it is not configurable (for example, `q` in the global object).

Additionally, certain JavaScript commands and functions, such as `for−in` or `Object.keys`, traverse over all enumerable properties of an object. As JavaScript objects are extensible, these properties need not be known statically. Also, the `for−in` loop may modify the object over which it is traversing. This behaviour is difficult to capture and further illustrates the dynamic nature of JavaScript.

In summary, JavaScript objects have an additional, highly non-trivial layer of complexity related to object property management with respect to objects in C++ or Java. Furthermore, this complexity cannot be captured natively by the existing tools for verifying C++ or Java programs (see §3.2 for a more detailed discussion). This constitutes an important challenge:

Challenge: To reason about extensible objects, dynamic property access, property descriptors, and property traversal.

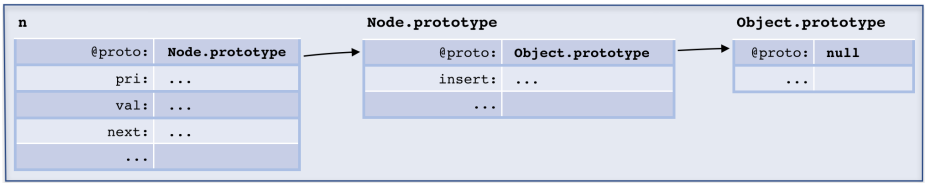

Fig. 2. The prototype chain of Node objects.

Prototype-based inheritance. JavaScript models inheritance through prototype chains. To look up the value of a property of an object, we first check the object itself. If the property is not there, we walk along the prototype chain, following the `@proto` internal properties, checking for the property at each object. In our example, all node objects constructed using the `enqueue` function (line 38) have a prototype chain like the one given in Figure 2. There, the lookup of property `val` starting from object `n` only needs to check `n`. The lookup of property `insert` starting from `n` first checks `n`, which does not have the property, then checks `Node.Prototype`, which does. In general, prototype chains can be of arbitrary length, typically finishing at `Object.prototype`, but they cannot be circular. Moreover, prototype chain traversal is additionally complicated in the presence of `String` objects, which have properties that do not exist in the heap.

Prototype chain traversal is one of the fundamental building blocks of the JavaScript language and is prominently featured in the behaviour of almost every JavaScript command. This brings us to our next challenge:

Challenge: To reason about prototype chains of arbitrary complexity.

Functions, Function objects. Functions are also stored in the JavaScript heap as objects. Each function object has three specific internal properties: **(1)** `@code`, storing the code of the original function; **(2)** `@scope`, storing a representation of the scope in which the function was defined; and **(3)** `prototype`, storing the prototype of those objects created using that function as the constructor. For example, `Node.prototype` is the prototype of all node objects constructed using the `Node` function, and is the place to find the `insert` function.

There are two main challenges related to reasoning about function objects. The first involves the interaction between function objects and scoping, which we address in the following paragraph. The second has to do with higher-order functions. Namely, JavaScript has full support for higher-order functions, meaning that a function can take another function as an argument, or that a function can return another function as a result. This behaviour is not easily captured, particularly in a program logic setting, but is often used in practice and verification of JavaScript programs should ultimately be able to tackle it.

Challenge: To reason about higher-order functions of arbitrary complexity.

Scoping, Function Closures. In JavaScript, scope is modelled using environment records (ERs). An ER is an internal object, created upon the invocation of a function, that maps the variables declared in the body of that function and its formal parameters to their respective values. Variables are resolved with respect to a list of ER locations, called a *scope chain*. In the non-strict mode of JavaScript, standard JavaScript objects can also be part of a scope chain. In strict mode, the only JavaScript object that can be part of a scope chain is the global object, which is treated as the ER of the global code. Since functions in JavaScript can be nested (e.g. `Node`, `enqueue`, `dequeue`) and can also be returned as outcomes of other functions (e.g. the `PQ` function is returned by `PQLib`), it is possible to create complex relationships between scope chains of various functions.

We discuss scoping through the `enqueue` function, which uses five variables in its body: `pri`, `val`, `n`, `Node`, and `counter`. The scope chain of `enqueue` contains the ERs corresponding to `enqueue`, `PQLib`, and global code. As `pri` and `val` are formal parameters and `n` is a local variable of `enqueue`, they are stored in the ER of `enqueue`. However, `Node` and `counter` are not declared in `enqueue` and are not its formal parameters, so we have to look for them in the rest of the scope chain associated with `enqueue`, and we find them in the ER corresponding to `PQLib`. This means that when we reason about `enqueue`, we need to capture not only its ER, but also a part of the ER of `PQLib`. We should also note that while the value of `Node` is static, the value of `counter` is changed both by `Node` and by `dequeue`, and that this change is visible by all of the functions of the library. Overall, the interaction of scope chains in JavaScript is very intricate, especially in the presence of multiple function closures. Therefore, our next challenge is:

Challenge: To reason about scope chains and function closures of arbitrary complexity.

Specification of JavaScript libraries.

There are two requirements necessary for the correct functioning of the priority queue library. First, the intention of the library developer is that all node objects constructed using the `Node` function should have access to the function `insert`. This means that the node objects themselves must not have the property `"insert"`. Second, we must always be able to construct a `Node` object. This means, due to the semantics of JavaScript, that `Node.Prototype` and `Object.Prototype` must not have properties `"pri"`, `"val"` and `"next"`, used in the node constructor, declared as non-writable. We call these two requirements *prototype safety*. We aim to provide a library specification for the priority queue that ensures prototype safety, and believe that we have identified a desired pattern of library behaviour suitable for JavaScript data structure libraries developed for Node.js.

Challenge: To provide specifications of JavaScript libraries that ensure prototype safety.

Hiding JavaScript internals. Our priority queue example illustrates some of the complexities of JavaScript: extensible objects, prototype-based inheritance, functions, scoping, and function closures. There is, in addition, much complexity that is not exposed to the JavaScript developer: for example, property descriptors, internal functions, as well as implicit type coercions, where values of one type are coerced at runtime to values of another type in order to delay error reporting. We would like to provide specifications that are as opaque as possible to such hidden features: since the code does not expose them, the specification should not expose them either. However, all of these features have to be taken into account when verifying that a program satisfies a specification. One solution is to provide abstractions that hide these internal details from view.

Challenge: To create abstractions that hide the internals of JavaScript as much as possible and allow the developer to write specifications in the style of C++ and Java specifications.

3 A Pathway to JavaScript Verification

Logic-based symbolic verification has recently become tractable for C and Java, with compositional techniques that scale and properly engineered tools applied to real-world code: for example, Infer, Facebook's tool based on separation logic for reasoning about for C, C++, Objective-C and Java [6]; Java Pathfinder, a model checking tool for Java bytecode programs [27]; CBMC, a bounded model checker for C, currently being adapted to Java at Amazon [20]; and WALA's analysis for Java using the Rosette symbolic analyser [12].

There has been little work on logic-based symbolic verification for JavaScript. As far as we are aware, the only relevant work is KJS [22,8], a tested executable semantics of JavaScript in the $\mathbb{K}$ framework [24] which is equiped with a symbolic execution engine. The aim of K is to provide a unified environment for analysing programming languages such as C, Java and JavaScript. Specifications are written in the reachability logic of $\mathbb{K}$, and the authors use KJS to specify operations on data structures, such as lists, binary search trees (BSTs) and AVL trees, and to verify the correctness of several sorting algorithms. This work does not address many of the challenges that laid out in the previous section. For example, it does not provide a general, abstract way of reasoning about prototype chains, scope chains, or function closures; the concrete shape of a prototype chain or a scope chain always needs to be known. It does not provide JavaScript-specific abstractions, so the specifications are cumbersome and reveal all JavaScript internals. The internal functions are always executed in full. More generally, previously proven function specifications cannot be reused to jump over function calls, so the function bodies always have to be executed. This significantly diminishes the scalability of KJS. We argue that a more JavaScript-specific approach is needed in order to make JavaScript verification tractable.

3.1 Choosing the Battleground

We believe that separation logic has much to offer JavaScript, since it provides a natural way of reasoning modularly about the JavaScript heap. Gardner, Smith and Maffeis developed a sound separation logic for a small fragment of JavaScript with many syntactic and semantic simplifications [13]. Their goal was to demonstrate that separation logic can be used to reason about the variable store emulated in the JavaScript heap. This approach is not extensible to the entire language. For example, consider the general assignment `e1 = e2`, where `e1` and `e2` are arbitrary JavaScript expressions. Under the hood, this assignment evaluates these two expressions and calls the `GetValue` and `PutValue` internal functions. The evaluation of each expression, as well as each of these two internal functions has tens of cases, so combining these case together would result in hundreds of axioms for the JavaScript assignment alone. Such a logic would be extremely difficult to prove sound, let alone automate. In order to reason about JavaScript, we need to move to a simple intermediate representation.

Working directly with JavaScript is not tractable for verification based on program logics. We need a simple intermediate representation.

3.2 Moving to a Simpler World

Our conclusion that some sort of an intermediate representation (IR) is necessary for JavaScript verification is not surprising. Most analysis tools, both for JavaScript [14,23,19,17,1,26] and other languages [2,3,9,15,7,6,12], use an IR. The next step is to understand what the desired features of an IR for logic-based JavaScript verification are. We believe that the following criteria need to be met.

1. **Expressiveness.** JavaScript is a highly dynamic language, with extensible objects, dynamic field access, and dynamic function calls. These features create an additional level of complexity for JavaScript when compared to other object-oriented languages such as C++ and Java. They should be supported natively by the IR.
2. **Simple control flow.** JavaScript has complicated control flow constructs: for example, `for-in`, which iterates on the fields of an object; `try-catch-finally` for handling exceptions; and the breaking out of loops to arbitrary labelled points in the code. Logic-based symbolic verification tools today typically work on IRs with simple control flow. In particular, many of the separation-logic tools for analysing C, C++, and Java use goto-based IRs: for example, [2,3,9,15,7,6]. This suggests that our IR for JavaScript should be based on simple low-level control flow constructs.

One option is to use an IR that has already been developed for analysing JavaScript code. We can broadly divide these IRs into two categories: **(1)** those that work for analyses that are syntax-directed, following the abstract syntax tree (AST) of the program, such as λ_{JS} [14], S5 [23], and notJS [19]; and **(2)** those that aim at analyses based on the control-flow graph of the program, such as JSIR [21], WALA [12,26] and the IR of TAJS [17,1]. The IRs in **(1)** are normally well-suited for high-level analysis, such as type-checking and type inference [14,23], whereas those belonging to **(2)** are generally the target of separation-logic-based tools, such as Smallfoot [2], Slayer [3], JStar [9], VeriFast [15], Abductor [7], and Infer [6], as well as tools for tractable symbolic evaluation such as CBMC [20] and Klee[5].

We believe that an IR for JavaScript verification should belong to **(2)**. The JSIR [21] and WALA [12,26] IRs both capture the dynamic features of JavaScript and provide low-level control flow constructs. However, neither JSIR nor WALA have associated compilers. In addition, they do not provide reference implementations of the JavaScript internal functions and built-in libraries, which makes it very difficult for us to assess their usability. TAJS [17,1] does include a compiler, originally for ECMAScript 3 (ES3) but now extended with partial models of the ES5 standard library, the HTML DOM, and the browser API. As TAJS is used for type analysis and abstract interpretation, its IR is more high-level than those typically used for logic-based symbolic verification. In addition, we believe that the aim for verification should be at least ECMAScript 5 (ES5) [10], which is substantially different from ES3 and essentially provides the core language for the more recent ES6 and ES7.

Another option is to consider using or adapting an IR supported by an existing separation-logic-based tool [2,3,9,15,7,6], where we would have to provide the compiler from JavaScript, but the analysis for the IR could be reused. There are two problems worth mentioning with this approach. First, these tools all target static languages that do not support extensible objects or dynamic function calls. Hence, JavaScript objects could not be directly encoded using the built-in constructs of these languages. Consequently, at the logical level, one would need to use custom abstractions to reason about JavaScript objects and their associated operations, effectively losing most of the native reasoning features of the tool in question. Second, any program logic for JavaScript needs to take into account the JavaScript binary and unary operators, such as `toInt32` [10], and it is not clear that these operators would be expressible using the assertion languages of existing tools. This brings us to the following conclusion:

JavaScript requires a dedicated low-level control-flow-based IR for verification: the simpler the IR, the better.

We have developed a simple JavaScript IR for our verification toolchain, called JSIL. It comprises only the most basic control flow commands (unconditional and conditional gotos), the object management commands needed to support extensible objects and dynamic field access, and top-level procedures. In the following section, we use JSIL to discuss what it means to design and trust the compilation and verification process. However, the methodology and principles that we lay out are general and apply to any verification IR for JavaScript.

3.3 Trusted compilation of JavaScript

The development of an appropriate IR is tightly connected with the development of the compiler from JavaScript to the IR, which brings up two challenges that need to be addressed:

1. The compilation has to capture all of the behaviours and corner cases of the JavaScript semantics, and must come with strong guarantees of correctness.
2. The reasoning at JavaScript level has to be strongly connected with the reasoning at the IR level; we refer to this as logic-preserving compilation.

To answer these two challenges, we unpack what it means to show that a compiler can be trusted.

– **Correctness by design.** This approach assesses the compiler by looking at the structure of the compiler code and at examples of compiled code. It is greatly simplified by using semantics-driven compilation, where the compiler and the compiled code follow the steps of the JavaScript English standard line-by-line as much as possible. This approach is feasible, because the JavaScript standard is given operationally, in an almost pseudo-code format. Given the complexity of JavaScript, this approach, albeit quite informal in nature, can give some confidence, particularly to the compiler developer, when it comes to compiler correctness. Ultimately, however, it is not formal enough to be sufficient on its own.

– **Correctness by testing.** JavaScript is a real-world language that comes with an official test suite, the ECMAScript Test262 [11]. Although Test262 is known not to be comprehensive, it features over 20,000 tests that extensively test most of the JavaScript behaviour. Correctly compiled JavaScript code should pass all of the appropriate tests.
– **Semantics-preserving compilation.** This correctness condition for the compiler is standard in compiler literature. It requires formalising the semantics and memory model of JavaScript, formalising the semantics and memory model of IR, giving a correspondence between the two memory models, and, with this correspondence, proving that the semantics of the JavaScript and compiled IR code match. For real-world languages, either a pen-and-paper proof is given for a representative fragment or a mechanised proof is given, in a proof assistant such as Coq, for the entire language.
– **Logic-preserving compilation.** This correctness condition for the compiler is not commonly emphasised in the analysis literature. It assumes semantic-preserving compilation and additionally requires: giving a strong correspondence between JavaScript and IR assertions; relating the semantics of JavaScript triples with the semantics of the IR triples; and proving a soundness result for the IR proof rules. In this way, one can formally lift IR verification to JavaScript verification.

What follows is an insight into our design process for JSIL and JS-2-JSIL, and the main lessons that we have learnt from it. We knew we wanted to achieve logic-preserving compilation, using a separation logic for reasoning about heap manipulation. From the start, a fundamental decision was to make the JavaScript and JSIL memory models as close to each other as possible. In the end, the only difference is the modelling of scope chains. We also chose to design JS-2-JSIL so that the compiled code follows the ECMAScript standard line-by-line, which meant that the choices for JSIL were quite apparent. This approach proved to be important for JS-2-JSIL. It leverages on the operational aspect of the standard, making the inspection and debugging of compiled code considerably easier.

Semantics-driven compilation is greatly beneficial.

We believe that testing is an indispensable part of establishing compiler correctness for real-world languages such as JavaScript. Regardless of how precise proof of correctness may be, there still is plenty of room for discrepancies to arise: for example, the implementation of the compiler might inadvertently deviate from its formalisation; or the formalised JavaScript semantics might deviate from the standard. For us, it was testing that guided our debugging process; without it, we would not be able to claim correctness of JS-2-JSIL.

Extensive testing of the compiled code is essential.

When writing a language compiler, one might claim that correctness-by-design and correctness-by-testing are sufficient: there is a clear design structure to the compiler that can be checked by looking at the code and by testing. This is

not enough when using the compiler for logic-based verification. In this case, we require logic-preserving compilation which formally connects JavaScript verification with JSIL verification. Logic-preserving compilation depends on semantic-preserving compilation, which is difficult to prove for such a complex language as JavaScript. We give a pen-and-paper proof correctness proof for a representative fragment of the language. We have given thought to providing a Coq proof of correctness, leveraging on our previous JSCert mechanised specification of JavaScript [4]. However, the process of formalising JSIL and JS-2-JSIL, and then proving the correctness is beyond our manpower. In contrast, the proof that the compiler is logic preserving is comparatively straightforward due to the simple correspondence between the JavaScript and JSIL memory models. Moreover, we noticed that the complexity of our proofs is strongly related to the complexity of this correspondence.

Semantic- and logic-preserving compilation is essential for verification. A simple correspondence between JavaScript and IR heaps is essential for containing the complexity of any correctness proofs.

3.4 Tackling the Javascript Internal Functions

The internal functions are described in the standard only in terms of pseudo-code and not JavaScript. They must, therefore, be implemented directly in the IR. With these implementations, we have identified two options on how to use them in verification.

- **Inlining.** The entire body of an internal function is inlined every time the function is supposed to be called in the compiled code.
- **Axiomatic specification.** Internal functions are treated as procedures of the IR and, as such, are fully axiomatically specified. Calls to internal functions are treated as standard procedure calls of the IR.

We do not believe that inlining is a viable option. Given the sheer number of calls to the internal functions and their intertwined nature, the size of the compiled code would quickly spiral out of control. We would also entirely lose the visual correspondence between the compiled code and the standard. Moreover, the bulk of verification time would be spent inside this code and the overall verification process would be very slow.

With axiomatic specifications, on the other hand, the calls to internal functions are featured in the compiled code as procedure calls to their IR implementations. In that sense, the compiled code reflects the English standard. During verification, the only check that has to be made is that the current symbolic state entails a precondition of the specification, which is both at a higher level of abstraction as well as faster than running the body of the function every time.

Axiomatic specifications of the internal functions are essential for tractable JavaScript verification.

Creating axiomatic specifications does not come without its challenges. The definitions of the internal functions are often intertwined, making it difficult to fully grasp the control flow and allowed behaviours. Specifying such dependencies axiomatically involves the joining of the specifications of all nested function calls at the top level, which results in numerous branchings. Also, some of the internal functions feature higher-order and, although it is possible to add higher-order reasoning to separation logic [25], the soundness result is known to be difficult. We believe that the resulting specifications, however, will be much more readable than the operational definitions of the standard. We also hope that they can also be easily reused for other types of analyses, by leveraging on executable code created from the axiomatic specifications.

3.5 JavaScript Verification Toolchain

We are currently developing a JavaScript verification toolchain (JaVerT), which targets the strict mode of the ES5 standard. It requires the JavaScript code to be annotated with assertions written in our assertion language for JavaScript (JS Logic). These annotations comprise specifications (the pre- and postconditions) for functions and global code, together with loop invariants and unfold/fold instructions for any user-defined predicates, such as a predicate for describing priority queues. JaVerT also features a number of built-in predicates that provide abstractions for the key concepts of JavaScript; in particular, for prototype inheritance, scoping, function objects and function closures. Such predicates enable the developer to move away from the complexity of the JavaScript semantics and write specifications in a logically clear and concise manner.

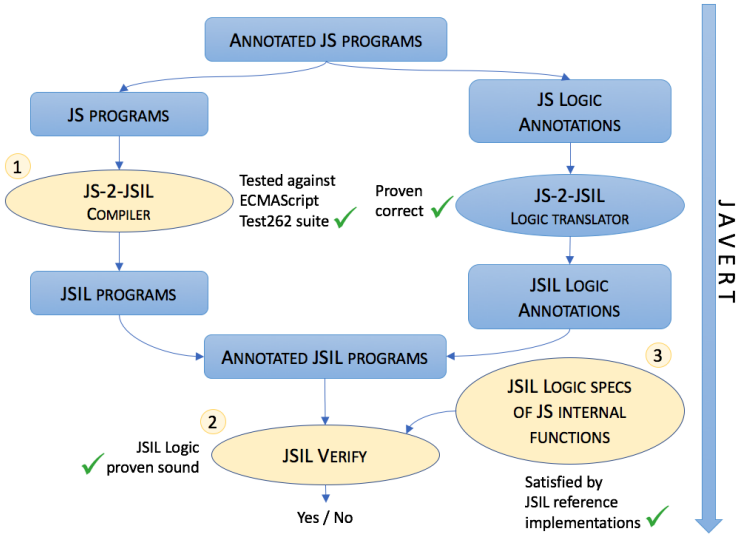

Fig. 3. JaVerT: JavaScript Verification Toolchain

Figure 3 presents the architecture of JaVerT, which rests on an infrastructure that consists of three components: **(1)** JS-2-JSIL, our semantics-preserving[3] and logic-preserving compiler from JavaScript to JSIL which has been tested using the official Test262 test suite, passing all the appropriate tests; **(2)** JSIL Verify, our semi-automatic tool for JSIL verification, based on a sound program logic for JSIL (JSIL Logic); and **(3)** our JSIL Logic axiomatic specifications of the JavaScript internal functions, which have been verified using JSIL Verify against their corresponding JSIL implementations.

Given a JavaScript program annotated with JS Logic specifications, JaVerT uses our JS-2-JSIL compiler to translate it to JSIL and the JS-2-JSIL logic translator to translate JS Logic annotations to JSIL Logic. The resulting annotated JSIL program is then automatically verified by JSIL Verify, taking advantage of our specifications of the JavaScript internal functions.

Thus far, we have used JaVerT to specify and verify a variety of heap-manipulating programs, including operations on lists (e.g. insertion sort), priority queues and BSTs, as well as a number of small JavaScript programs that showcase our treatment of prototype chains, scoping, and function closures. All examples can be found online at [16] and are continually being updated.

4 Specifying the Priority Queue Library

We illustrate JaVerT specifications by specifying the `enqueue` and `dequeue` methods of the priority queue library, given in Figure 1. We show how these specifications are used to verify the client program given in lines 58-62 of the example.

In order to specify `enqueue` and `dequeue`, we first need to have a predicate `Queue`, describing a priority queue, and the predicate `QueueProto`, describing the priority queue prototype. The predicate `Queue(lq, qp, np, pri_q, len)` describes a priority queue at location `lq`, whose prototype is `qp`, whose nodes have node prototype `np`, whose maximum priority is `pri_q`, and which contains `len` nodes. The predicate `QueueProto(qp, np, c)` describes a priority queue prototype at location `qp` for those priority queues built from node objects whose node prototype is `np`. The parameter `c` records the value of the variable `counter` of the example (line 3), and holds the total number of existing node objects.

These two abstractions, which we will not unfold in detail here, capture, among others, the resource associated with the `Node`, `insert`, `enqueue`, and `dequeue` function objects, as well as the resource corresponding to the function closures of `enqueue` and `dequeue`: in particular, for `enqueue`, we need the variable property `Node` from the ER of `PQLib`; and, for `dequeue`, we need the variable resource `counter` from that same ER. They also capture the resources necessary to express prototype safety for both `Node` and `PQ`, which we describe using a technique from [13] for reasoning about the absence of properties in an object. We explicitly require the `insert` property of node object `n`, and the `pri`, `val`, and `next` properties of `Node.prototype` and `Object.prototype` not to be in the heap, as illustrated

[3] The formal result that the compiler is semantics-preserving has been done for a fragment of the language.

in Figure 4 by the properties in red with value None. Note that the Queue and QueueProto predicates do not expose the internals of JavaScript, such as property descriptors and scope chains. Moreover, they do not expose functions not accessible to the client, such as the Node function. They do expose Node.prototype via the np parameter, but this is expected since the client can access it through the _head property of a queue.

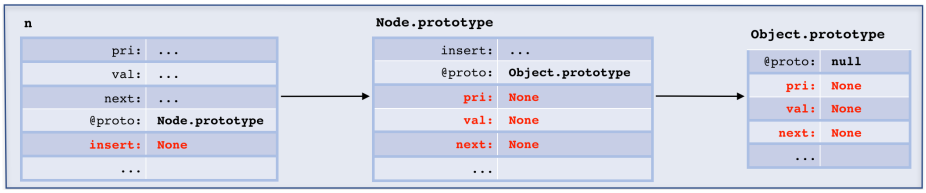

Fig. 4. Prototype safety for Node objects

The following specification of enqueue states that it should be executed on a priority queue of arbitrary length len, that the total number of existing nodes c needs to be not greater than 42, and that it receives two arguments pri and val with pri of type Num. The postcondition states that enqueue returns a priority queue with len + 1 nodes and maximum priority max(pri_q, pri), and that the total number of nodes has increased by one. Due to space requirements, we omit the specification of enqueue corresponding to the error case in which the total number of existing nodes is greater than 42.

```
{ Queue(this, qp, np, pri_q, len) * QueueProto(qp, np, c) *
                types(pri: Num) * c <= 42 }
                    enqueue(pri, val)
{ Queue(ret, qp, np, max(pri_q, pri), len+1) * QueueProto(qp, np, c+1) }
```

The following specification of dequeue states that it should be executed on a priority queue with length len greater than 0 and maximum priority pri_q. The postcondition states that, afterwards, the length of the queue has decreased by one, its priority has not increased, and the overall total number of nodes has decreased by one. The function also returns a standard object with two fields, pri with value pri_q and val with value #val which is existentially quantified. We prefix existentially quantified variables with a '#'. In the postcondition, the standardObject and dataField abstractions hide the internal properties and property descriptors of JavaScript objects. Again, due to space requirements, we omit the specification of dequeue where the queue from which we are dequeueing is empty and an error is thrown.

```
{ Queue(this, qp, np, pri_q, len) * QueueProto(qp, np, c) * len > 0 }
                        dequeue()
{   Queue(this, qp, np, pri', len-1) * QueueProto(qp, np, c-1) * pri' <= pri_q *
 standardObject(ret) * dataField(ret, "pri", pri_q) * dataField(ret, "val", #val) }
```

Given the specifications of enqueue and dequeue, we can verify the client program in lines 59-62. We show a proof sketch below, where we use the assertion

`scope(x: v)` to state that variable `x` has value `v` in the current scope. Starting from an empty queue with maximum priority 0, we create three nodes, obtaining a queue with three nodes and maximum priority 3. Then, we dequeue the head of the queue (which we can do, as we know that the queue has 3 nodes), obtaining a queue with 2 nodes and existentially quantified priority `#pri` not greater than 3. Moreover, in the end, the variable `r` is bound to an object with two fields: `pri`, with value 3; and `val`, with value `#val` which is existentially quantified.

```
{ scope(q: qv) * Queue(qv, qp, np, 0, 0) * QueueProto(qp, np, 0) * scope(r: undefined) }
                q.enqueue(1, "last"); q.enqueue(3, "bar"); q.enqueue(2, "foo")
{ scope(q: qv) * Queue(qv, qp, np, 3, 3) * QueueProto(qp, np, 3) * scope(r: undefined) }
                var r = q.dequeue()
{    scope(q: qv) * Queue(qv, qp, np, #pri, 2) * QueueProto(qp, np, 2) * #pri <= 3 *
 scope(r: #r) * standardObject(#r) * dataField(#r, "pri", 3) * dataField(#r, "val", #val) }
```

These specifications show that it is possible to successfully abstract over JavaScript internals, allowing both the library developer and the client developer to write specifications that are as free as possible from JavaScript-specific clutter.

4.1 Discussion

We conclude with a brief discussion of two important aspects of specifying JavaScript libraries: capturing prototype safety; and enforcing encapsulation. The situation for prototype safety is straightforward. It is not possible to verify a specification of client code if it compromises prototype safety. The situation for encapsulation is more subtle. In the example, a client can break encapsulation by modifying node objects or `Node.prototype`. There are ways of breaking encapsulation that we could choose to allow. The client could, for instance, add more functionalities to `Node.prototype` or add more properties to node objects, and this would not break the existing functionalities. However, there are ways of breaking encapsulation that we should certainly disallow. The client could, for instance, change the values of the `pri`, `val`, or `next` properties of a node object, or change the implementation of the `insert` function in `Node.prototype`. One way to ensure full encapsulation would be to keep the `Queue` and `QueueProto` predicates opaque to the client code. Hence, in order to be successfully verified, client code can only interact with a priority queue via its established interface, that being the `enqueue` and `dequeue` methods. By keeping library predicates opaque, we make sure that client code cannot break the existing abstractions.

Acknowledgments. Fragoso Santos, Gardner, and Maksimović were supported by the EPSRC Programme Grant REMS: Rigorous Engineering for Mainstream Systems (EP/K008528/1), and the Department of Computing in Imperial College London. Naudžiūnienė was supported by an EPSRC DTA award. Maksimović was also partially supported by the Serbian Ministry of Education and Science through the Mathematical Institute of Serbian Academy of Sciences and Arts, projects ON174026 and III44006.

References

1. E. Andreasen and A. Møller. Determinacy in static analysis for jquery. In *OOPSLA*, 2014.
2. J. Berdine, C. Calcagno, and P. O'Hearn. Smallfoot: Modular automatic assertion checking with separation logic. In *FMCO*, 2005.
3. J. Berdine, B. Cook, and S. Ishtiaq. Slayer: Memory safety for systems-level code. In *CAV*, 2011.
4. M. Bodin, A. Charguéraud, D. Filaretti, P. Gardner, S. Maffeis, D. Naudziuniene, A. Schmitt, and G. Smith. A Trusted Mechanised JavaScript Specification. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL'14, pages 87–100. ACM Press, 2014.
5. C. Cadar, D. Dunbar, and D. R. Engler. KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs. In R. Draves and R. van Renesse, editors, *8th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2008, December 8-10, 2008, San Diego, California, USA, Proceedings*, pages 209–224. USENIX Association, 2008.
6. C. Calcagno, D. Distefano, J. Dubreil, D. Gabi, P. Hooimeijer, M. Luca, P. O'Hearn, I. Papakonstantinou, J. Purbrick, and D. Rodriguez. Moving fast with software verification. In *NASA Formal Methods*, pages 3–11. Springer, 2015.
7. C. Calcagno, D. Distefano, P. O'Hearn, and H. Yang. Compositional shape analysis by means of bi-abduction. In *POPL*, 2009.
8. A. Ştefănescu, D. Park, S. Yuwen, Y. Li, and G. Roşu. Semantics-based program verifiers for all languages. In *Proceedings of the 31th Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'16)*, pages 74–91. ACM, Nov 2016.
9. D. Distefano and M. Parkinson. jStar: Towards practical verification for Java. In *OOPSLA*, 2008.
10. ECMAScript Committee. The 5th edition of the ECMAScript Language Specification. Technical report, ECMA, 2011.
11. ECMAScript Committee. Test262 test suite. https://github.com/tc39/test262, 2017.
12. S. Fink and J. Dolby. WALA — The T.J. Watson Libraries for Analysis. http://wala.sourceforge.net/, 2015.
13. P. Gardner, S. Maffeis, and G. Smith. Towards a program logic for JavaScript. In *Proceedings of the 40th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL'13, pages 31–44. ACM Press, 2012.
14. A. Guha, C. Saftoiu, and S. Krishnamurthi. The essence of Javascript. In *Proceedings of the 24th European Conference on Object-Oriented Programming (ECOOP)*, Lecture Notes in Computer Science, pages 126–150. Springer, 2010.
15. B. Jacobs, J. Smans, P. Philippaerts, F. Vogels, W. Penninckx, and F. Piessens. Verifast: A powerful, sound, predictable, fast verifier for c and java. In *NASA Formal Methods*, pages 41–55. Springer, 2011.
16. JaVerT Team. Javert. http://goo.gl/au69SV, 2017.
17. S. H. Jensen, A. Møller, and P. Thiemann. Type analysis for JavaScript. In *Proceedings of the 16th International Static Analysis Symposium (SAS)*, volume 5673 of *Lecture Notes in Computer Science*, pages 238–255. Springer, 2009.
18. J. Jones. Priority queue data structure. https://github.com/jasonsjones/queue-pri, 2016.

19. V. Kashyap, K. Dewey, E. A. Kuefner, J. Wagner, K. Gibbons, J. Sarracino, B. Wiedermann, and B. Hardekopf. JSAI: a static analysis platform for javascript. In *FSE*, pages 121–132, 2014.
20. D. Kroening and M. Tautschnig. CBMC – C bounded model checker. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 8413 of *LNCS*, pages 389–391. Springer, 2014.
21. B. Livshits. JSIR, An Intermediate Representation for JavaScript Analysis, 2014. `http://too4words.github.io/jsir/`.
22. D. Park, A. Stefănescu, and G. Roşu. Kjs: A complete formal semantics of javascript. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2015, pages 346–356, New York, NY, USA, 2015. ACM.
23. J. G. Politz, M. J. Carroll, B. S. Lerner, J. Pombrio, and S. Krishnamurthi. A Tested Semantics for Getters, Setters, and Eval in JavaScript. In *Proceedings of the 8th Symposium on Dynamic Languages*, 2012.
24. G. Roşu and T. F. Şerbănuţă. An overview of the K semantic framework. *Journal of Logic and Algebraic Programming*, 79(6):397–434, 2010.
25. J. Schwinghammer, L. Birkedal, B. Reus, and H. Yang. Nested hoare triples and frame rules for higher-order store. *Logical Methods in Computer Science*, 7(3), 2011.
26. M. Sridharan, J. Dolby, S. Chandra, M. Schäfer, and F. Tip. Correlation tracking for points-to analysis of javascript. In *ECOOP*, pages 435–458, 2012.
27. W. Visser, C. S. Păsăreanu, and S. Khurshid. Test input generation with java pathfinder. In *Proceedings of the 2004 ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA '04, pages 97–107, New York, NY, USA, 2004. ACM.