JaVerT: JavaScript Verification Toolchain

JOSÉ FRAGOSO SANTOS, Imperial College London, UK
PETAR MAKSIMOVIĆ, Imperial College London, UK and Mathematical Institute SASA, Serbia
DAIVA NAUDŽIŪNIENĖ, Imperial College London, UK
THOMAS WOOD, Imperial College London, UK
PHILIPPA GARDNER, Imperial College London, UK

The dynamic nature of JavaScript and its complex semantics make it a difficult target for logic-based verification. We introduce JaVerT, a semi-automatic JavaScript Verification Toolchain, based on separation logic and aimed at the specialist developer wanting rich, mechanically verified specifications of critical JavaScript code. To specify JavaScript programs, we design abstractions that capture its key heap structures (for example, prototype chains and function closures), allowing the developer to write clear and succinct specifications with minimal knowledge of the JavaScript internals. To verify JavaScript programs, we develop JaVerT, a verification pipeline consisting of: JS-2-JSIL, a well-tested compiler from JavaScript to JSIL, an intermediate goto language capturing the fundamental dynamic features of JavaScript; JSIL Verify, a semi-automatic verification tool based on a sound JSIL separation logic; and verified axiomatic specifications of the JavaScript internal functions. Using JaVerT, we verify functional correctness properties of: data-structure libraries (key-value map, priority queue) written in an object-oriented style; operations on data structures such as binary search trees (BSTs) and lists; examples illustrating function closures; and test cases from the official ECMAScript test suite. The verification times suggest that reasoning about larger, more complex code using JaVerT is feasible.

ACM Reference Format:
José Fragoso Santos, Petar Maksimović, Daiva Naudžiūnienė, Thomas Wood, and Philippa Gardner. 2018. JaVerT: JavaScript Verification Toolchain. *Proc. ACM Program. Lang.* 2, POPL, Article (January 2018), 33 pages. https://doi.org/10.1145/3158138

1 INTRODUCTION

Separation logic was developed in order to reason about programs that manipulate data structures in the heap. The reasoning has been shown to be tractable, with compositional techniques that scale [Reynolds 2002] and properly engineered tools applied to real-world code. In particular, separation logic has been used to reason about programs written in static languages: for example, the semi-automatic verification tool Verifast [Jacobs et al. 2011] for reasoning about C and Java programs; the automatic verification tool Infer [Calcagno et al. 2015], being developed at Facebook, for reasoning about C, Java, C++ and Objective C programs; and the interactive Coq development for reasoning about, for example, ML-like programs [Krebbers et al. 2017] using Iris [Jung et al. 2015]. In contrast, separation logic has hardly been used to reason about programs written in dynamic languages in general, and JavaScript in particular. The goal of this paper is to explore the extent to which techniques from separation logic, proven to work for C, C++, and Java, can

Authors' addresses: José Fragoso Santos, Imperial College London, UK, jose.fragoso.santos@imperial.ac.uk; Petar Maksimović, Imperial College London, UK, petar.maksimovic@imperial.ac.uk, Mathematical Institute SASA, Serbia; Daiva Naudžiūnienė, Imperial College London, UK, daiva.naudziuniene@imperial.ac.uk; Thomas Wood, Imperial College London, UK, thomas.wood@imperial.ac.uk; Philippa Gardner, Imperial College London, UK, p.gardner@imperial.ac.uk.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).
© 2018 Copyright held by the owner/author(s).
2475-1421/2018/1-ART
https://doi.org/10.1145/3158138

be applied to full, non-simplified JavaScript. This is part of a wider, more general study, done by ourselves and others, to assess the trade-off between static and dynamic analysis for JavaScript.

JavaScript is one of the most widespread dynamic languages: 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. It is used by 94.8% of websites[1], and is the most active language in both GitHub[2] and StackOverflow.[3] Standardised by the ECMAScript committee and natively supported by all major browsers, JavaScript is a complex and evolving language. Logic-based reasoning about JavaScript programs poses a number of significant challenges. To specify JavaScript programs, the challenge is to design assertions that fully capture the common heap structures of JavaScript, such as property descriptors, prototype chains for modelling inheritance, the variable store emulated in the heap, and function closures. Importantly, these assertions should abstract as much as possible from the details of the heap structures they describe, to provide a specification that makes sense to the JavaScript developer who has limited knowledge of the JavaScript internals. To verify JavaScript programs, the challenge is to handle the complexity of the JavaScript semantics, due to: (V1) the behaviour of JavaScript statements, which exhibit complicated control flow with several breaking mechanisms and ways of returning values; (V2) the fundamental dynamic behaviour associated with extensible objects, dynamic property access, and dynamic function calls; and (V3) the JavaScript internal functions, which underpin the JavaScript statements and whose definitions in the ECMAScript standard are operational, intricate, and intertwined.

There has been little theoretical and practical work on logic-based reasoning about JavaScript. Gardner et al. [2012] have developed a separation logic for a tiny fragment of ECMAScript 3 (ES3). In JavaScript, the program state resides in the object heap, imperfectly emulating the standard variable store. This work demonstrated that separation logic can be used to reason about this emulated variable store: for example, to specify when programs are safe from prototype poisoning attacks. Cox et al. [2014] have combined separation logic and abstract interpretation to show how to specify property iteration for a simple extensible object calculus. This work focussed on a simplified version of the JavaScript `for-in` statement. It is intractable to extend such logic-based analysis to full JavaScript. Instead, we must work with an intermediate representation. We build on the work of Gardner et al. [2012] in this paper; we expect to build on the work of Cox et al. [2014] in future. On the more practical side, Swamy et al. [2013] have used the higher-order logic of F* to prove absence of runtime errors for higher-order ES3 programs using the Dijkstra monad, but have stopped short of proving functional correctness properties. Ştefănescu et al. [2016] have built a verification tool for JavaScript based on their K framework and associated reachability logic. Their aim is to provide general analysis for languages interpreted in K, not specific analysis for JavaScript. We discuss this and other related work in more detail in §2.

In this paper, we present JaVerT,[4] a semi-automatic JavaScript Verification Toolchain for reasoning about JavaScript programs using separation logic, aimed at the specialist developer wanting rich, mechanically verified specifications of critical JavaScript code. JaVerT verifies functional correctness properties of JavaScript programs annotated with pre- and post-conditions, loop invariants, and instructions for folding and unfolding user-defined predicates. JaVerT specifications are written using JS Logic, our assertion language for JavaScript. JS Logic features a number of built-in predicates (§3) that allow the developer to specify JavaScript programs with only a minimal knowledge of JavaScript internals: for example, the `DataProp` predicate abstracts over data descriptors; the

[1]w3techs.com/technologies/details/cp-javascript/all/all
[2]http://githut.info
[3]https://exploratory.io/viz/Hidetaka-Ko/94368d12800a?cb=1469037012628.
[4]JaVerT is pronounced *zhah-vehr* (IPA: ʒaˈvɛʁ), like the name of the main antagonist of Victor Hugo's 'Les Miserables'.

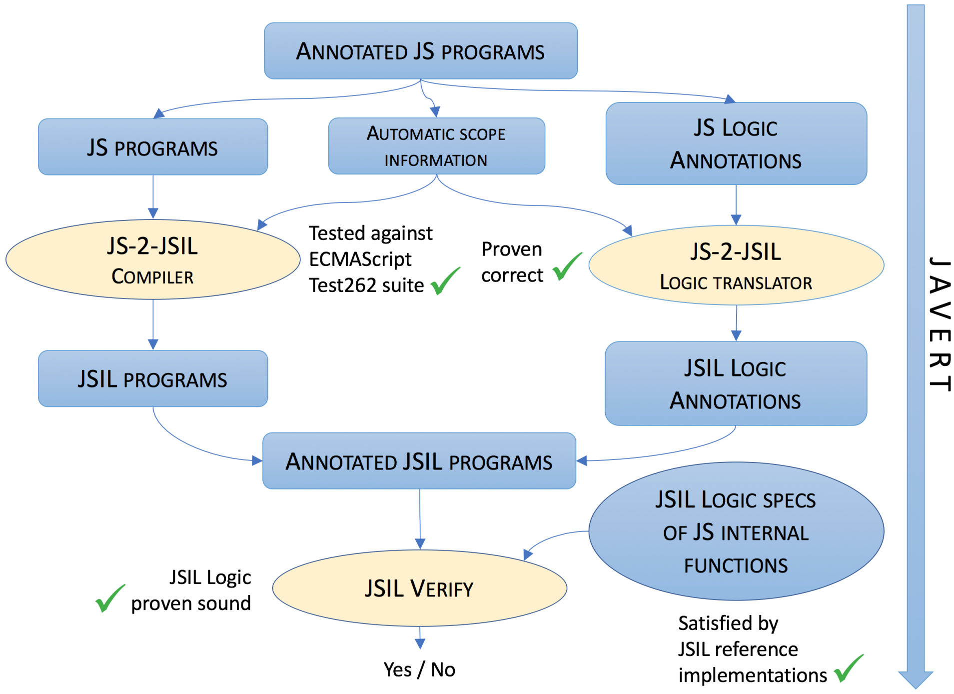

Fig. 1. JaVerT: JavaScript Verification Toolchain

`Pi` predicate captures prototype chains; the `Scope` predicate allows reasoning about basic variable scoping; and the `Closure` predicate precisely describes JavaScript function closures.

The structure of the JaVerT verification pipeline is illustrated in Figure 1 and is driven by the three verification challenges (V1)–(V3). To solve (V1), in §4 we introduce a simple intermediate goto language, JSIL,[5] and a logic-preserving compiler from JavaScript to JSIL, called JS-2-JSIL.[6] JS-2-JSIL is designed to be line-by-line close to the ECMAScript standard, without simplifying the behaviour in any way.[7] Instead of reasoning directly about code built from complex JavaScript statements, we use JS-2-JSIL to reason about compiled JSIL code built from simple JSIL statements. JSIL is designed so that its heap model subsumes the heap model of JavaScript. Hence, JavaScript and JSIL assertions coincide, making the JS-2-JSIL logic translator and its correctness proof straightforward.

JSIL retains the fundamental dynamic behaviour of JavaScript given by extensible objects, dynamic property access and dynamic function calls. To solve the verification challenge (V2), in §5 we introduce JSIL Verify, our semi-automatic verification tool for JSIL. JSIL Verify is based on JSIL Logic, a sound separation logic for JSIL. The development of JSIL Verify is challenging due to the dynamic behaviour of JSIL. JSIL Verify comprises a symbolic execution engine and an entailment engine, which uses the Z3 SMT solver [De Moura and Bjørner 2008] to discharge assertions in first-order logic with equality and arithmetic, while we handle the separation logic assertions. As with many tools based on separation logic, a key task during symbolic execution is to solve the frame inference problem. This is more challenging for us, due to the dynamic nature of JSIL.

We solve our final verification challenge (V3) in §5.3, by writing axiomatic specifications for the JavaScript internal functions in JSIL Logic and providing reference implementations in JSIL. The reference implementations are line-by-line close to the standard and are proven correct with respect to the axiomatic specifications using JSIL Verify. Our use of axiomatic specifications of the

[5]JSIL is pronounced *jis-suhl* (IPA: ˈdʒisəl) or *jay-sill* (IPA: ˈdʒeɪsil), not *jay-ess-aye-ell*.
[6]JS-2-JSIL is pronounced *jay-ess-to-JSIL*.
[7]JS-2-JSIL targets the strict mode of the ECMAScript 5 English standard. We discuss this choice in §4.2.

internal functions enables us to: keep the compiled JSIL code visually closer to the ECMAScript standard; and expose explicitly the allowed behaviours of the internal functions, in contrast with their intertwined operational definitions given in the standard.

For us, an important part of this project was to validate the components of JaVerT: the JS-2-JSIL compiler and logic translator; JSIL Verify; and the JSIL axiomatic specifications of the JavaScript internal functions. JS-2-JSIL has broad coverage and is systematically tested against the official ECMAScript test suite, passing all 8797 tests applicable for its coverage. JSIL Logic is sound with respect to its operational semantics. Since JSIL is designed so that the JSIL heap model subsumes the JavaScript heap model, the correctness of the logic translator is straightforward. JS-2-JSIL is logic-preserving, with JSIL verification lifting to JavaScript verification. JSIL Verify is validated by verifying that the reference implementations of the internal functions are correct with respect to their axiomatic specifications, and by verifying compiled JavaScript programs. The specifications of the internal functions are validated by verifying that they are satisfied by their well-tested corresponding JSIL reference implementations. Further details can be found in §6.

We also validate JaVerT as a whole by verifying specifications of JavaScript code. As JaVerT is a semi-automatic verification tool, we believe its target should be critical JavaScript code, such as Node.js libraries describing frequently used data structures. We have used JaVerT to verify a simple key-value map library (§3.4) and a priority queue library modelled after a real-world Node.js priority queue library of Jones [2016]. Libraries such as these, written in an object-oriented style, are typical for JavaScript. The code, however, no longer guarantees the expected good behavioural properties of these libraries due of the dynamic nature of JavaScript; our specifications do. In §3.5, we have verified an ID generator, a simple example illustrating JavaScript function closures and how they can be used to emulate data encapsulation. Our specifications capture the achieved degree of encapsulation. Further, we have verified operations on binary search trees, targeting set reasoning, and an insertion sort algorithm, targeting list reasoning. Finally, we have verified several programs from the ECMAScript Test262 test suite, which test complex language statements such as `switch` and `try-catch-finally`. Due to our predicates, our specifications successfully abstract over the JavaScript internals and are in the style of separation-logic specifications for C++ or Java. Our verification times suggest that JaVerT can be used to reason about larger, more complex code. A detailed discussion is given in §6.4.

JaVerT has two limitations that need to be addressed. Currently, we cannot reason about the `for-in` loop and higher-order functions. For the specification of `for-in`, we will leverage on the work of Cox et al. [2014], who have shown how to reason about property iteration in a simple extensible object calculus. Specifying the `for-in` of JavaScript is substantially more complicated because it only targets enumerable properties and iterates over the entire prototype chain. The verification of `for-in` will also push the set reasoning capabilities of Z3 to their limit. It is likely that we will need to implement complex set reasoning heuristics in JSIL Verify. Higher-order reasoning is known to be difficult for separation logic, involving the topos of trees of Birkedal et al. [2012]. Our current plan is to encode JSIL Logic in Iris [Jung et al. 2015], obtaining soundness for free.

2 RELATED WORK

This paper brings together a number of techniques associated with operational semantics, compilers and separation logic. Many of these techniques have been introduced for static languages. Their application to dynamic JavaScript is not straightforward.

Logic-based Verification of JavaScript Programs. The existing literature covers a wide range of analysis techniques for JavaScript programs, including: type systems [Anderson et al. 2005; Bierman et al. 2014; Feldthaus and Møller 2014; Jensen et al. 2009; Microsoft 2014; Rastogi et al.

2015; Thiemann 2005], control flow analysis [Feldthaus et al. 2013], pointer analysis [Jang and Choe 2009; Sridharan et al. 2012] and abstract interpretation [Andreasen and Møller 2014; Jensen et al. 2009; Kashyap et al. 2014; Park and Ryu 2015], among others. In contrast, there has been comparatively little work on logic-based verification of JavaScript programs.

Gardner et al. [2012] have developed a separation logic for a tiny fragment of ECMAScript 3, to reason about the variable store emulated in the JavaScript heap. We draw partial inspiration from this work: our property assertions are similar; our predicate for describing prototype chains is different. An extension of their logic to the full language is intractable. For example, the behaviour of the JavaScript assignment is described in the ECMAScript standard in terms of expression evaluation and calls to the internal functions `getValue` and `putValue`. This effectively means that the assignment is described by hundreds of possible pathways through the standard; each of these pathways would have to be a proof rule of the logic, making automation essentially impossible. The same issues would give rise to even greater complexity when applied to the complex control-flow given by, for example, the `switch` and `try–catch–finally` statements. Direct verification of JavaScript programs using separation logic is, therefore, not feasible. It is necessary to move to an intermediate representation (IR), with simpler commands and simpler control flow. This comment also applies to other logics for reasoning directly about JavaScript, such as the work combining separation logic with abstract interpretation to reason about `for–in` [Cox et al. 2014].

Swamy et al. [2013] use F* to prove absence of runtime errors for higher-order JavaScript programs. This is achieved by: annotating JavaScript programs with assertions and loop invariants in the logic of F*; compiling an annotated JavaScript program (a subset of ES3) to F*; using a type inference algorithm to generate verification conditions for the absence of runtime errors; automatically discharging these verification conditions using Z3. The authors state, but do not demonstrate, that this methodology is extensible to functional correctness. Their assertions, abstractions, and reasoning are all in the higher-order logic of F*. As they aim at safety, there are no abstractions that capture, for example, JavaScript prototype chains or function closures. Our goal is to provide systematic functional correctness specifications that resonate with the knowledge of the developer. We provide assertions and carefully designed abstractions in JS Logic, together with a translation to JSIL Logic, where the reasoning occurs, and prove that this reasoning lifts back to JavaScript.

Fournet et al. [2013] address safe library development: the developer writes library code in a subset of F* and compiles it to JavaScript (ES3). The compilation preserves all source program properties. As F* comes with an expressive type system, this approach can ensure code safety. Our agenda is different. We aim to verify functional correctness for existing JavaScript code. Ideas from this paper might help us generate defensive wrappers from our verified specifications.

Roşu and Şerbănuţă [2010] have developed $\mathbb{K}$, a term-rewriting framework for formalising the operational semantics of programming languages. In particular, they have developed KJS [Park et al. 2015] which provides a $\mathbb{K}$-interpretation of the core language and part of the built-in libraries of the ES5 standard. KJS has been tested against the official ECMAScript Test262 test suite and passed all 2782 tests for the core language; the testing results for the built-in libraries are not reported. The coverage of JS-2-JSIL is broader; we pass all 8797 tests applicable for our coverage (cf. §6.1).

Ştefănescu et al. [2016] introduce a language-independent verification infrastructure that can be instantiated with a $\mathbb{K}$-interpretation of a language to automatically generate a symbolic verification tool for that language based on the $\mathbb{K}$ reachability logic. They apply this infrastructure to KJS to generate a verification tool for JavaScript, which they use to verify functional correctness properties of operations for manipulating data structures such as binary search trees, AVL trees, and lists.

These examples, however, do not address the majority of critical JavaScript-specific features,[8] and also contain no JavaScript-specific abstractions. A developer thus has to consider all of the internals of JavaScript in order to specify JavaScript code, making the specification difficult and error-prone.

Our approach is entirely different. JaVerT is a specialised verification toolchain, addressing the reasoning challenges posed by JavaScript. We create layers of abstractions, allowing the developer to write specifications with only a minimal knowledge of the JavaScript internals. Similarly to Ştefănescu et al. [2016], we use JaVerT to verify correctness of data structure operations. In addition, we show how to reason about common JavaScript programming idioms, such as emulating OO-style programming via prototype-based inheritance and data encapsulation via function closures.

Verification Tools based on Separation Logic. Separation logic enables compositional reasoning about programs which manipulate complex heap structures. It has been successfully used in verification tools for static languages: Smallfoot [Berdine et al. 2005a] for a simple imperative while language; jStar [Distefano and Parkinson 2008] for Java; Verifast [Jacobs et al. 2011] for C and Java; Space Invader [Yang et al. 2008] and Abductor [Calcagno et al. 2011] for C; and Infer [Calcagno et al. 2015] for C, Java, Objective C, and C++.

All of these verification tools compile to simple goto IRs, designed especially for the language under consideration. These IRs cannot be reused for JavaScript verification, as these tools target static languages that do not support the fundamental dynamic aspects of JavaScript (V2). Therefore, we would have to use custom-made abstractions to describe JavaScript object cells, losing native support for reasoning about object properties and having to axiomatise property operations. We attempted to do this using the CoreStar theorem prover, obtaining prohibitive performance even for simple examples. Moreover, any program logic for JavaScript needs to take into account the JavaScript operators, such as `toInt32` [ECMAScript Committee 2011], and it is not clear that these operators could be expressed using the assertion languages of existing tools.

Compilers and IRs for JavaScript. There is a rich landscape of IRs for JavaScript, broadly divided into two categories: **(1)** those for syntax-directed analyses, following the abstract syntax tree of the program, such as λ_{JS} [Guha et al. 2010], S5 [Politz et al. 2012], and notJS [Kashyap et al. 2014]; and **(2)** those for analyses based on the control-flow graph of the program, such as JSIR [Livshits 2014], WALA [Sridharan et al. 2012] and the IR of TAJS [Andreasen and Møller 2014; Jensen et al. 2009]. SAFE [Lee et al. 2012], an analysis framework for JavaScript, provides IRs in both categories. The IRs in **(1)** are normally well-suited for high-level analysis, such as type-checking/inference, whereas those in **(2)** are generally the target of separation-logic tools and tools for tractable symbolic evaluation [Cadar et al. 2008; Kroening and Tautschnig 2014]. We believe that an IR for logic-based JavaScript verification should belong to the latter category.

Our aim for JSIL was to: **(1)** natively support the fundamental dynamic features of JavaScript (V2); **(2)** have JSIL heaps be identical to JavaScript heaps, to keep correctness proofs simple; and **(3)** keep JSIL minimal to simplify JSIL logic. For control flow, JSIL has only conditional and unconditional `goto` statements. Having gotos in an IR for JavaScript verification is reasonable, because: first, separation-logic-based verification tools commonly have goto IRs; second, JavaScript has complex control flow statements with many corner cases (for example, `switch` and `try-catch-finally`), which can be naturally decompiled to gotos; third, JavaScript supports a restricted form of goto statements, via labelled statements, breaks, and continues. We have *only* gotos because we have not encountered the need for more structured loops: our invariants are always JavaScript assertions; and the JavaScript internal and built-in functions implemented in JSIL use only simple loops.

[8] The $\mathbb{K}$ framework currently does not support predicates whose footprint captures some, but not all, properties of an object. Therefore, it cannot be used to reason generally about dynamic property access, prototype inheritance, or function closures. We were informed by the authors that a new development of $\mathbb{K}$ is underway and will support this.

JSIL is similar to JSIR, and the IRs of WALA and TAJS. JSIR and the IR of WALA do not have associated JavaScript compilers, and the design choices have not been stated so it is difficult to compare with JSIL. JSIL is syntactically simpler. TAJS includes a well-tested compiler, targeted for ES3 (which is substantially different from ES5), but now extended with partial models of the ES5 standard library, the HTML DOM, and the browser API. Since TAJS was designed for type analysis and abstract interpretation, the IR that it uses is slightly more high-level than those typically used for logic-based symbolic verification. The IR of SAFE based on control flow is not documented.

One of our main goals in the development of JS-2-JSIL was to be fully compliant with ES5 Strict. Thus, a strong connection between the generated JSIL code and the standard was imperative. Our design of JS-2-JSIL builds on the tradition of compilers that closely follow the operational semantics of the source language, such as the ML Kit Compiler [Birkedal et al. 1993]. In that spirit, JS-2-JSIL mimics ES5 Strict by inlining in the generated JSIL code the internal steps performed by the ES5 Strict semantics, making them explicit. To achieve this, we based our compiler on the JSCert mechanised specification of ES5 [Bodin et al. 2014]. Alternatively, we could have used KJS.

We have considered using S5 of Politz et al. [2012], which targets ES5, as an interim stage during compilation. The compilation from ES5 to S5 is informally described in this paper, and is validated through testing against the ECMAScript test suite, with 70% success on all ES5 tests and 98% on tests for unique features of ES5 Strict. The figure critical for us, the success rate of S5 on full ES5 Strict tests (those testing its unique features *and* the features common with ES5), was not reported. Therefore, we would have to redo S5 tests using our methodology and fix the unfamiliar code in light of failing tests. Also, to prove correctness of our assertion translation and, ultimately, JaVerT, we would have to relate JS Logic and JSIL Logic via S5. This would be a difficult task.

3 SPECIFYING JAVASCRIPT PROGRAMS

We address the JavaScript specification challenges highlighted in the introduction. To specify JavaScript programs, we need to design assertions that fully capture the key heap structures of JavaScript, such as property descriptors, prototype chains for modelling inheritance, the variable store emulated in the heap using scope chains, and function closures. We start by introducing the memory model of ES5 Strict and the JS Logic assertions in §3.1. We would like the user of JaVerT to be able to specify JavaScript programs clearly and concisely, with only a minimal knowledge of JavaScript internals. We must, therefore, build a number of predicates on top of JS Logic to describe common JavaScript heap structures. In §3.2, we introduce our basic predicates for describing object properties, function objects, string objects and the JavaScript initial heap. In §3.3, we introduce the `Pi` predicate, which precisely captures the prototype chains of JavaScript. In §3.4, we provide a general approach for specifying JavaScript libraries written in a typical object-oriented (OO) style, using a simple key-value map as the example. For such libraries, we give specifications that ensure *prototype safety* of library operations, in that they describe the conditions under which these operations exhibit the desired behaviour. Finally, in §3.5, we show how to specify variable scoping and function closures, using an ID generator example to show how our specifications can be used to capture the degree of encapsulation obtained from using function closures.

3.1 JavaScript Specifications: Preliminaries

The basic memory model of JavaScript is straightforward. The difficulty lies in the way in which it is used to emulate the variable store and to provide prototype inheritance using prototype chains.

JavaScript Memory Model

JS locations : $l \in \mathcal{L}$ JS variables : $x \in \mathcal{X}_{\mathrm{JS}}$ JS heap values : $\omega \in \mathcal{V}^h_{\mathrm{JS}} ::= v \mid \overline{v} \mid fid$

JS values: $v \in \mathcal{V}_{\mathrm{JS}} ::= n \mid b \mid m \mid \mathsf{undefined} \mid \mathsf{null} \mid l$ JS heaps : $h \in \mathcal{H}_{\mathrm{JS}} : \mathcal{L} \times \mathcal{X}_{\mathrm{JS}} \rightharpoonup \mathcal{V}^h_{\mathrm{JS}}$

A JavaScript heap, $h \in \mathcal{H}_{\text{JS}}$, is a partial function mapping pairs of object locations and property names to JS heap values. Object locations are taken from a set of locations $\mathcal{L}$. Property names and JS program variables are taken from a set of strings $\mathcal{X}_{\text{JS}}$. JS values contain: numbers, n; booleans, b; strings, m; the special JavaScript values `undefined` and `null`; and object locations, l. JS heap values, $\omega \in \mathcal{V}^h_{\text{JS}}$, contain: JS values, $v \in \mathcal{V}_{\text{JS}}$; lists of JS values, $\overline{v}$; and function identifiers, $\mathit{fid} \in \mathcal{F}\mathit{id}$. Function identifiers, *fid*, are associated with syntactic functions in the JavaScript code and are used to represent function bodies in the heap uniquely. This choice differs from the approach of Gardner et al. [2012], where function bodies are also JS heap values. The ECMAScript standard does not prescribe how function bodies should be represented and our choice closely connects the JavaScript and JSIL heap models. Given a heap h, we denote a heap cell by $(l, x) \mapsto v$ when $h(l, x) = v$, the union of two disjoint heaps by $h_1 \uplus h_2$, a heap lookup by $h(l, x)$, and the empty heap by emp.

JS Logic assertions mostly follow those introduced by Gardner et al. [2012]; the main difference is that we do not use the sepish connective $\circledast$, which was used to describe overlapping prototype chains. We discuss this difference in §3.4 and §5.3.

JS Logic Assertions

$$V \in \mathcal{V}^L_{\text{JS}} ::= \omega \mid \omega_{\text{set}} \mid \varnothing \qquad E \in \mathcal{E}^L_{\text{JS}} ::= V \mid x \mid \mathrm{x} \mid \ominus E \mid E \oplus E \mid \mathsf{sc} \mid \mathsf{this}$$

$$\tau \in \mathsf{Types} ::= \mathsf{Num} \mid \mathsf{Bool} \mid \mathsf{Str} \mid \mathsf{Undef} \mid \mathsf{Null} \mid \mathsf{Obj} \mid \mathsf{List} \mid \mathsf{Set} \mid \mathsf{Type}$$

$$P, Q \in \mathcal{AS}_{\text{JS}} ::= \mathsf{true} \mid \mathsf{false} \mid E = E \mid E \leq E \mid P \land Q \mid \neg P \mid P * Q \mid \exists \mathrm{x}.P \mid \mathsf{emp} \mid (E, E) \mapsto E \mid \mathsf{emptyFields}(E \mid E) \mid \mathsf{types}(X_i : \tau_i|_{i=1}^n)$$

JS logical values, $V \in \mathcal{V}^L_{\text{JS}}$, contain: JS heap values, ω; sets of JavaScript heap values, ω_{set}; and the special value $\varnothing$, read *none*, used to denote the absence of a property in an object (see §3.4). JS logical expressions, $E \in \mathcal{E}^L_{\text{JS}}$, contain: logical values, V; JS program variables, x; JS logical variables, x; unary and binary operators, $\ominus$ and $\oplus$ respectively; and the special expressions, `sc` and this, referring respectively to the current scope chain (see §3.5) and the current this object (see §3.4). JS Logic assertions are constructed from: basic boolean constants, comparisons, and connectives; the separating conjunction; existential quantification; and assertions for describing heaps and declaring typing information. The `emp` assertion describes an empty heap. The cell assertion, $(E_1, E_2) \mapsto E_3$, describes an object at the location denoted by E_1 with a property denoted by E_2 that has the value denoted by E_3. The assertion $\mathsf{emptyFields}(E_1 \mid E_2)$ states that the object at the location denoted by E_1 has no properties other than possibly those included in the set denoted by E_2. The assertion $\mathsf{types}(X_i : \tau_i \mid_{i=1}^n)$ states that variable X_i has type τ_i for $0 \leq i \leq n$, where X_i is either a program or a logical variable and τ ranges over JavaScript types, $\tau \in$ `Types`. We say that an assertion is *spatial* if it contains cell assertions or `emptyFields` assertions. Otherwise, it is *pure*.

JaVerT specifications have the form $\{P\}\, \mathit{fid}(\overline{x})\, \{Q\}$, where P and Q are the pre- and postconditions of the function with identifier *fid*, and $\overline{x}$ is its list of formal parameters. We treat global code as a function with identifier `main`. Each specification has a return mode $\mathit{fl} \in \{\mathsf{nm}, \mathsf{er}\}$, indicating if the function returns normally or with an error. If it returns normally, the return value is stored in the (dedicated) variable `ret`; otherwise, the error value is stored in the variable `err`. Intuitively, given a JavaScript program s and return mode *fl*, a specification $\{P\}\, \mathit{fid}(\overline{x})\, \{Q\}$ is *valid* if s contains a function with identifier *fid* and "whenever *fid* is executed in a state satisfying P, then, if it terminates, it does so in a state satisfying Q, with return mode *fl*". The formal definition is given in §4.3.

3.2 Basic JS Logic Predicates

We start by introducing the basic predicates for describing JavaScript object properties, function objects, string objects and the JS initial heap. These predicates constitute the building blocks of our specifications and are widely used throughout the paper.

Object Properties. JavaScript objects have two types of properties: *internal* and *named*. Internal properties have no analogue with C++ or Java. They are hidden from the user, are associated directly with JS values, and are critical for the mechanisms underlying JavaScript such as prototype inheritance. We prefix internal properties with the `@` symbol, to distinguish them from named properties. Standard JavaScript objects have three internal properties, `@proto`, `@class`, and `@extensible`, which respectively denote the prototype of the object, the class of the object, and whether the object can be extended with new properties.

JaVerT has two built-in predicates for describing internal properties of JavaScript objects. The `JSObject(o, p)` predicate states that object `o` has prototype `p`, and its internal properties `@class` and `@extensible` have their default values, `"Object"` and `true`. Its general version, the `JSObjGen(o, p, c, e)` predicate, allows the user to specify the values of `@class` and `@extensible` as `c` and `e`.

```
JSObjectGen(o, p, c, e) := types(o : Obj, p : Str, c : Str, e : Bool) *
                           (o, "@proto") -> p * (o, "@class") -> c * (o, "@extensible") -> e
   JSObject(o, p)       := JSObjectGen(o, p, "Object", true)
```

Named properties are similar to object properties in C++ or Java, except that they are not associated with values but with *property descriptors*, which are lists of *attributes* that describe the ways in which a property can be accessed and/or modified. Depending on the attributes they contain, descriptors can be *data descriptors* or *accessor descriptors*. For lack of space, we focus on data descriptors.

Data descriptors contain the *value*, *writable*, *enumerable*, and *configurable* attributes, denoted by `[V]`, `[W]`, `[E]`, and `[C]`, respectively. The `[V]` holds the actual property value. The `[W]` describes whether the value `[V]` may be modified. The `[E]` indicates whether the property is included in for-in enumerations. The `[C]` denotes whether `[W]`, `[E]` or the property type (data or accessor property) may be modified. Note that the modifiability of `[V]` is determined by `[W]` and is thus not controlled by `[C]`.

We represent descriptors as five-element lists; the first element states the descriptor type and the remaining four represent values of appropriate attributes; for example, `["d", "foo", true, false, true]` is a writable, non-enumerable, and configurable data descriptor with value `"foo"`.

Depending on their associated descriptor, JavaScript named properties can be *data properties* or *accessor properties*. Again, we focus only on data properties. JaVerT has two built-in predicates for describing data properties. The `DataProp(o, p, v)` predicate states that the property `p` of object `o` holds a data descriptor with value `v` and all other attributes set to `true`. The more general predicate, `DataPropGen(o, p, v, w, e, c)`, allows the user to specify the values of the remaining attributes. We also define a predicate `DescVal(desc, v)`, stating that the data descriptor `desc` has value attribute `v`.

Function Objects. In JavaScript, functions are also stored as objects in the heap. In addition to the `@proto`, `@class`, and `@extensible` internal properties common to all objects, function objects also have the `@code` property, storing the function identifier of the original function, and the `@scope` property, storing the scope chain associated with the function object (discussed in detail in §3.5).

JaVerT offers the `FunctionObject(o, fid, sc)` predicate, which describes the function object `o`, whose internal properties `@code` and `@scope` have values given by the function identifier, `fid`, and the location of the scope chain, `sc`, respectively.

String Objects. String objects are native wrappers for primitive strings. Every string object has an internal property `@pv` holding its corresponding primitive string value. String objects differ from standard JavaScript objects in that they expose indexing properties (the i-th character of a string) that do not exist in the heap. For instance, the statement `var s = new String("foo"); s[0]` evaluates to the string `"f"`, even though the object bound to `s` does not have the named property `"0"`. To reason about properties of string objects, we define the `SCell(o, p, d)` predicate, which states that property `p` of string object `o` is associated with either a property descriptor or the value `None`.

In the definition of SCell, we use the predicate IsStringIndex(s, i), which holds if and only if i is a non-negative integer smaller than the length of the string s. Also, we use the operators s−nth and str2num to retrieve the nth element of a string and convert a string to a number, respectively.

```
SCell (o, p, d) :=
  types(o : Obj, p : Str) * (o, "@pv") -> pv * ! IsStringIndex(pv, str2num(p)) * (o, p) -> d,
  types(o : Obj, p : Str) * (o, "@pv") -> pv * IsStringIndex(pv, str2num(p)) * (o, p) -> None *
    c = s-nth(pv, str2num(p)) * d = [ "d", c, false, false, false ]
```

The SCell(o, p, d) predicate has two cases (disjuncts), which are separated with a comma. In both cases, o has to denote an object, and p has to denote a string. In the first case, p is not a string index of the primitive string, in which case the associated value is looked up in the heap. In the second case, p is a string index of the primitive string, in which case the associated data descriptor is ["d", c, false, false, false], as string indexes are not enumerable, writable, or configurable.

Please note that, in the specifications, we denote negation by the ! symbol. Also, we do not distinguish between program variables (parameters of predicates and functions, for example, o, p, and d) and logical variables (for example, pv and c), which are implicitly existentially quantified.

JS Initial Heap. Prior to execution of a JavaScript program, an *initial heap* is established, containing the global object and the objects associated with built-in libraries (for example, Object, Function and String), as well as their prototypes. We provide predicates that describe the built-in library objects, as well as the entire initial heap. These predicates come in two flavours: frozen, where changes to the target object(s) are not allowed; and open, where changes are allowed. For instance, InitialHeap() and ObjProtoF() describe the open initial heap and the frozen Object.prototype, respectively.

3.3 Specifying Prototype Inheritance

JavaScript models inheritance through prototype chains. In order to retrieve the value of an object property, first the object itself is inspected. If the property is not present, then the prototype chain is traversed (following the @proto internal properties), checking for the property at each object. In general, prototype chains can be of arbitrary length (typically finishing at Object.prototype) but cannot be circular. Prototype chain traversal is additionally complicated in the presence of String objects, which have indexing properties that do not exist in the heap.

While in some cases it is reasonable to expect the precise structure of a prototype chain to be known *a priori*, there are cases in which this is not possible. For instance, consider the following function for obtaining the value associated with property p in the prototype chain of object o, which only returns the value of p if it is public, for some black-boxed notion of being public captured by the JavaScript function isPublic(p).

```
function getPublicProp (o, p) { if (isPublic(p)) { return o[p] } else { return null } }
```

We should, ideally, be able to specify this function without knowing anything about the concrete shape of the prototype chain of o, other than the value to which it maps the property p.

Assume that we have a predicate Pi(o, p, d, ...), describing the resource of the prototype chain of o in which property p is mapped onto a data descriptor d, and may require additional parameters. Also assume that Public(p) is a predicate that holds if and only if isPublic(p) returns true. Then, we can specify getPublicProp (o, p) as follows:

```
{ Pi(o, p, d, ...) * DescVal(d, v) * Public(p) * ... }
              getPublicProp(o, p)
    { Pi(o, p, d, ...) * Public(p) * ret = v * ... }
```

This specification states that, when getPublicProp gets as input a public property p in object o, it returns the value associated with that property in the prototype chain of o. It is general, as it makes no assumptions on the structure of the prototype chain. For clarity, we have omitted the assertions capturing the resource corresponding to the function `isPublic`. We have also not repeated the `DescVal` predicate in the postcondition, since it is pure.

For our `Pi` predicate, we take inspiration from the prototype-chain predicate of Gardner et al. [2012]. Their predicate describes prototype chains of standard objects with simple values, whereas ours describes prototype chains for property descriptors and accounts for the subtle combination of standard objects and string objects, capturing the full prototype inheritance of JavaScript.

We define the `Pi` predicate, `Pi (o, p, d, lo, lc)`, stating that property `p` has value `d` in the prototype chain of `o`. The value `d` can either be a property descriptor or the value `undefined`. The two additional parameters, `lo` and `lc`, denote lists that respectively capture the locations and classes of the objects in the prototype chain up to and including the object in which `p` is found, or of all objects if the property is not found. These two parameters arise because of the complexity of the internal functions and are justified in §5.3. The JavaScript programmer does not need to consider these parameters and can always pass logical variables in their place. Below is the full definition of the `Pi` predicate, with four base cases and two recursive cases.

```
Pi (o, p, d, lo, lc) :=
 T * lo = [o] * lc = [c] * (o, "@class") -> c * !(c = "String") * (o, p) -> d * !(d = None),
 T * lo = [o] * lc = [c] * (o, "@class") -> c *  (c = "String") * SCell(o, p, d) * !(d = None),
 T * lo = [o] * lc = [c] * (o, "@class") -> c * !(c = "String") *
     (o, @proto) -> null * (o, p) -> None * d = undefined,
 T * lo = [o] * lc = [c] * (o, "@class") -> c *  (c = "String") *
     (o, @proto) -> null * SCell(o, p, None) * d = undefined,
 T * lo = o :: lop * lc = c :: lcp * (o, "@class") -> c * !(c = "String") * (o, p) -> None *
     lop = op :: lop' * (o, "@proto") -> op * Pi(op, p, d, lop, lcp),
 T * lo = o :: lop * lc = c :: lcp * (o, "@class") -> c *  (c = "String") * SCell(o, p, None) *
     lop = op :: lop' * (o, "@proto") -> op * Pi(op, p, d, lop, lcp)
```

where `T` denotes the assertion `types(o : Obj, p : Str)`.

3.4 Specifying OO-style Libraries: Prototype Safety

JavaScript programmers rely on prototype-based inheritance to emulate the standard class-based inheritance mechanism of static OO languages when implementing JavaScript libraries. However, as JavaScript objects are extensible, it is possible to break the functionality of such libraries by adding properties either to the constructed objects or to their prototype chains. This makes the specifications of these libraries challenging as they not only need to capture the resources that must be present in the heap, but also the resources that *must not be present* in the heap if the library code is to run as intended. We highlight a general methodology for specifying such libraries, introducing the notion of *prototype safety* to specify when libraries behave as intended.

Example: Key-Value Map. We illustrate how JaVerT is used to specify JavaScript OO-style libraries, using the JavaScript implementation of a *key-value map* given in Figure 2 (left). It contains four functions: `Map`, for constructing an empty map; `get`, for retrieving the value associated with the key given as input; `put`, for inserting a new *key-value pair* into the map and updating existing keys; and `validKey`, for deciding if a key is valid or not. This library implements a *key-value map* as an object with property `_contents`, denoting the object used to store the map contents. The named properties of `_contents` and their value attributes correspond to the map keys and values, respectively. As the functions `get`, `put`, and `validKey` are to be shared between all map objects, they are defined as properties of `Map.prototype`, which is the prototype of the objects that are created using `Map` as a constructor (for example, using `new Map()` in the client examples of Figure 2 (right)).

Language: Breaking the Library. In order to guarantee that this library works as intended, we must make sure that: **(1)** every time one calls `get`, `put` or `validKey` on a map object, one reaches the appropriate functions defined within its prototype; **(2)** one can always successfully construct an

```
function Map () { this._contents = {} }

Map.prototype.get = function (k) {
    if (this._contents.hasOwnProperty(k)) {
        return this._contents[k]
    } else { return null }
}

Map.prototype.put = function (k, v) {
   var contents = this._contents;
   if (this.validKey(k)) {
      contents[k] = v;
   } else { throw new Error("Invalid_Key") }
}

Map.prototype.validKey = function (k) { ... }
```

Client 1:

```
var m = new Map();
m.get = "foo"
```

Client 2:

```
var mp = Map.prototype;
var desc = { value: 0, writable: false };
Object.defineProperty(mp, "_contents", desc)
```

Client 3:

```
var m = new Map ();
m.put("hasOwnProperty", "bar")
```

Fig. 2. JavaScript OO-style Map implementation (left); three library-breaking clients (right).

object map using the `Map` constructor; and **(3)** one can always retrieve the value of a key previously inserted into a map as well as insert a new valid key-value pair into a map. In Figure 2 (right), we show how a user can misuse the library, effectively breaking **(1)**-**(3)**. To break **(1)**, one simply has to override `get` or `put` on the constructed map object (Client 1). To break **(2)**, it suffices to assign an arbitrary *non-writable* value to `_contents` in `Map.prototype` (Client 2). To break **(3)**, one can insert a key-value pair with `"hasOwnProperty"` as a key into the map. By doing this, `"hasOwnProperty"` in the prototype chain of `_contents` is overridden and subsequent calls to `get` will fail (Client 3).

JaVerT: Capturing Prototype Safety. In general, the specification of a given library must ensure that all prototype chains are consistent with correct library behaviour by stating which resources must not be present for its code to run correctly. In particular, constructed objects cannot redefine properties that are to be found in their prototypes; and prototypes cannot define as *non-writable* those properties that are to be present in their instances. We refer to these two criteria as *prototype safety*, and illustrate how it can be achieved through the specification of the key-value map.

We define a *map object predicate* below, `Map`, using the auxiliary predicate `KVPairs`, which captures the resource of the key-value pairs in the map, and the `validKey(k)` predicate, which holds if and only if the JavaScript function `ValidKey(k)` returns `true`[9]. Intuitively, the `Map(m, mp, kvs, keys)` predicate captures the resource of a map object `m` with prototype `mp`, key-value pairs `kvs` (a set of pairs whose first component is a string[10]), and keys `keys` (a set of strings). We write `-u-` for set union and omit the brackets around singleton sets when the meaning is clear from the context.

```
Map (m, mp, kvs, keys) := JSObject(m, mp) *
  DataProp(m, "_contents", c) * JSObject(c, Object.prototype) *
  (m, "get") -> None * (m, "put") ->  None * (m, "validKey") ->  None *
  (c, "hasOwnProperty") ->  None * KVPairs(c, kvs, keys) * emptyFields(c, keys -u- "hasOwnProperty")
```

```
KVPairs (o, kvs, keys) :=
  (kvs = { }) * (keys = { }),
  (kvs = (key, value) -u- kvs') * (keys = key -u- keys') *
    ValidKey(key) * DataProp(o, key, value) * KVPairs(o, kvs', keys')
```

The definition of `Map` achieves the first requirement for prototype safety by stating that a map object `m` cannot have the properties `"get"`, `"put"`, and `"validKey"`, and that the object bound to `_contents` cannot have the property `"hasOwnProperty"`. The `emptyFields` predicate, together with the prototype safety requirement `(c, "hasOwnProperty") -> None`, ensures that there are no other properties in the contents of the map except for the keys.

[9] We treat the `ValidKey` predicate as a black box, other than requiring that `hasOwnProperty` is not a valid key.
[10] We model pairs as two-element lists and, for clarity, use the pair notation.

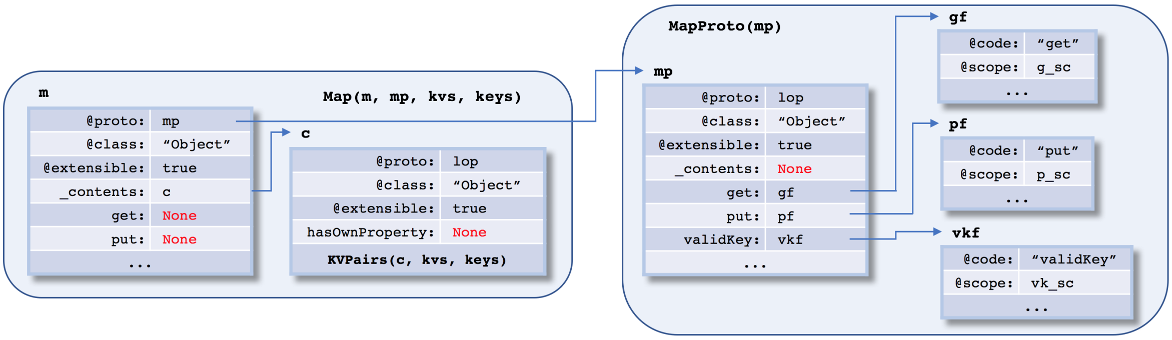

Fig. 3. Graphical representation of Map (m, mp, kvs, keys) ∗ MapProto (mp)

Observe that the definition of Map does not include the resource of a map prototype. Since Map.prototype is shared between all map objects, we cannot include the resource of a map prototype in the definition of Map. Were we to do that, we could no longer write a satisfiable assertion describing two distinct map objects using the standard separating conjunction. Below, we show the definition of MapProto, stating that a valid map prototype has the properties "get", "put", and "validKey", respectively assigned to the appropriate functions (see §3.2). The definition of MapProto achieves the second requirement for prototype safety by stating that a map prototype cannot have the property "_contents". We could have weakened this definition, stating that a map prototype can have the property "_contents", as long as it is writable. In Figure 3, we give a graphical representation of the assertion Map (m, mp, kvs, keys) ∗ MapProto (mp).

```
MapProto (mp) :=  JSObject(mp, Object.prototype) * (mp, "_contents") -> None) *
   DataProp(mp, "get", gf)        * FunctionObject(gf, "get", g_sc) *
   DataProp(mp, "put", pf)        * FunctionObject(pf, "put", p_sc) *
   DataProp(mp, "validKey", vkf) * FunctionObject(vkf, "validKey", vk_sc)
```

We are now in the position to specify the functions of the map library. In particular, below we show how to use the map object predicate and the map prototype predicate to specify put(k, v).

```
{ Map(this, mp, kvs -u- (k, v'), ks) *        { Map(this, mp, kvs, ks) * MapProto(mp) *
       MapProto(mp) * ObjProtoF() }             !(k -in- ks) * ValidKey(k) * ObjProtoF() }
             put(k, v)                                        put(k, v)
{ Map(this, mp, kvs -u- (k, v), ks) *         { Map(this, mp, kvs -u- (k, v), ks -u- k) *
       MapProto(mp) * ObjProtoF() }                    MapProto(mp) * ObjProtoF() }

      { Map(this, mp, kvs, ks) * MapProto(mp) * !ValidKey(k) * ObjProtoF() }
                                   put(k, v)
    { Map(this, mp, kvs, ks) * MapProto(mp) * ErrorObject(err) * ObjProtoF() }
```

The first specification captures the case in which the key of key-value pair to be inserted already exists in the map, while the second one captures the case in which it does not. The third specification captures the error case, when the given key is not valid. Since put calls the function validKey, all of its specifications must include the MapProto(mp) predicate, that captures the location of validKey.

Recall that the prototype safety requirements of the library extend to Object.prototype as well. This resource is captured by the built-in ObjProtoF() predicate, describing the frozen Object.prototype object (see §3.2). Here, the user can instead choose to use the open version of the predicate, ObjProto(), allowing for a more flexible initial heap. In that case, they would have to manually specify the prototype safety requirements, as we have done for maps and the map prototype.

3.5 Specifying Scoping and Function Closures

Example: Identifier Generator. We illustrate variable scoping and function closures using a JavaScript identifier (ID) generator, shown in Figure 4. The function makeIdGen takes a string prefix, and returns a new ID generator, which is an object with two properties: `getId`, storing a function for creating fresh IDs; and `reset`, storing a function for resetting the ID generator. `getId` ensures that the returned ID is fresh by using a counter, stored in variable `count`, which is appended to the generated ID string of the form `prefix + '_id_'` and is incremented afterwards.

```
1  var makeIdGen = function (prefix) {
2    var count = 0;
3
4    var getId = function () {
5      return prefix + '_id_' + (count++)
6    };
7
8    var reset = function () { count = 0 };
9
10   return { getId: getId, reset: reset }
11 }
12 var ig1 = makeIdGen("foo");
13 var ig2 = makeIdGen("bar");
14 var id1 = ig1.getId();
```

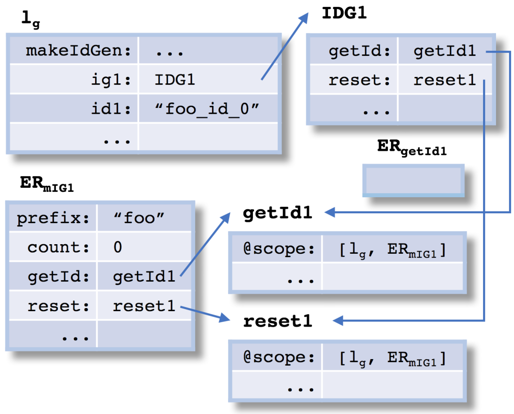

Fig. 4. Identifier Generator (left); partial post-execution heap (right)

The variable `count` is not intended to be directly accessible by programs using `makeIdGen`, but rather only through the `getId` and `reset` functions. In Java, `count` can be declared private. In JavaScript, however, there is no native mechanism for encapsulation and the standard approach of establishing some form of encapsulation is to use function closures. In our example, once an ID generator is created, the variables `count` and `prefix` remain accessible only from within the code of `getId` and `reset`, making it impossible for client code (such as lines 12-14 of the example) to access or modify them directly. In the general case, however, full encapsulation cannot be guaranteed.

Language: Scope resolution in ES5 Strict. In JavaScript, scope is modelled in the heap using *environment records* (ERs). An ER is an internal object, created upon the invocation of a function, mapping variables declared in the body of that function and its parameters to their respective values. For example, each time `makeIdGen` is called, two new function objects representing `getId` and `reset` are created in the heap, as well as a new ER for that particular execution of `makeIdGen`. In particular, after executing `makeIdGen("foo")`, we get the objects `getId1` and `reset1`, as well as the $\mathsf{ER}_{\mathtt{mIG1}}$ environment record (Figure 4 (right)); the execution of `makeIdGen("bar")` is similar.

Variables are resolved with respect to a list of ER locations, called a *scope chain*. When executing a function *fid*, its scope chain consists of the list found in the `@scope` field of the function object corresponding to *fid*, extended with the ER of *fid* created for that execution. For instance, during the execution of `ig1.getId()`, the scope chain will be $[l_g, \mathsf{ER}_{\mathtt{mIG1}}, \mathsf{ER}_{\mathtt{getId1}}]$. We can also observe that, for example, function objects `getId1` and `reset1` share the $[l_g, \mathsf{ER}_{\mathtt{mIG1}}]$ part of their scope chains.

When trying to determine the value of a variable `x` during the execution of a function *fid*, the semantics inspects the scope chain of *fid* and, if no binding for `x` is found, the prototype chain of the global object. However, as ES5 Strict is lexically scoped, we can statically determine if `x` is defined in the scope chain of *fid* and, if so, in which ER it is defined. Therefore, we do not model the scope inspection procedure as a list traversal, but use instead a *scope clarification function*, $\psi : \mathcal{S}tr \times \mathcal{S}tr \rightharpoonup \mathbb{N}$, for determining which ER in the scope chain of a given function defines a given variable. For instance, $\psi(\texttt{"getId"}, \texttt{"makeIdGen"}) = 0$ and $\psi(\texttt{"getId"}, \texttt{"count"}) = 1$, as the variables `makeIdGen` and `count` are defined, respectively, in the first and the second ER in the scope chain of `getId`. We also use the *overlapping scope function*, $\psi^o : \mathcal{S}tr \times \mathcal{S}tr \rightharpoonup \mathbb{N}$, which takes two function identifiers and returns the length of the overlap of their scope chains. For instance, $\psi^o(\texttt{"getId"}, \texttt{"reset"}) = 2$, as `getId` and `reset` share the global object and the ER of `makeIdGen`.

JaVerT: Specifying Scoping. To capture variable scoping, we introduce the `Scope` predicate. The `Scope(x : v, sch, fid)` predicate states that the variable `x` has value `v` in the scope chain denoted by `sch` of the function literal with identifier `fid`. In the general case, this predicate corresponds to the JS Logic assertion $(\mathtt{nth}\,(\mathtt{sch}, n),\ \mathtt{x}) \mapsto \mathtt{v}$, where `nth` is the binary list indexing operator and $n = \psi(\mathtt{fid},\ \mathtt{x})$. For instance, the predicate `Scope(count : c, gi_sc, getId)` unfolds to $(\mathtt{nth}\,(\mathtt{gi\_sc}, 1),\ \texttt{"count"}) \mapsto \mathtt{c}$ as $\psi(\mathtt{getId},\ \mathtt{count}) = 1$. We can also use `Scope(x : v)` as syntactic sugar for `Scope(x : v, sc, fid)`, where `sc` is the special logical expression denoting the current scope chain and `fid` is the identifier of the current function.

$$\mathsf{Scope}(\mathsf{x} : \mathsf{v}, \mathsf{sch}, \mathsf{fid}) := (\mathsf{nth}(\mathsf{sch}, \mathsf{n}), \mathsf{x}) \mapsto \mathsf{v},\ \text{when } \mathsf{n} = \psi(\mathsf{fid}, \mathsf{x}) \neq 0;$$
$$\mathsf{Scope}(\mathsf{x} : \mathsf{v},\ \_, \mathsf{fid}) := (\mathsf{l_g}, \mathsf{x}) \mapsto [\text{"d"}, \mathsf{v},\ \_,\ \_,\ \_],\ \text{when } \psi(\mathsf{fid}, \mathsf{x}) = 0.$$

```
{ Scope(prefix: p) * Scope(count: c) *
        types(p: Str, c: Num)          }
              getId()
{ Scope(prefix: p) * Scope(count: c+1) *
  (ret = p ++ "_id_" ++ numToString(c)) }
```

Fig. 5. Specification of `getId`

To illustrate `Scope`, we specify `getId` in Figure 5. `getId` uses the `prefix` and `count` variables, defined in the ER of `makeIdGen`. We capture this in the precondition with `Scope(prefix: p) * Scope(count: c)`. We also state that the value of `prefix` (`p`) is a string and the value of `count` (`c`) is a number. After execution, the value of `prefix` remains the same, while the value of `count` is incremented, which we capture with `Scope(prefix: p) * Scope(count: c+1)`. The return value is described using string concatenation (++) and number-to-string conversion (`numToString`). This specification again highlights the importance of our abstractions: to specify `getId`, the user does not need to know anything about the internal representation of scope chains. We revisit this specification shortly in the context of encapsulation.

JaVerT: Specifying Function Closures. The major challenge associated with specifying function closures in JavaScript comes from the fact that, in contrast to static languages such as Java and ML, the JavaScript variable store is emulated in the heap and constitutes spatial resource. Since scope chains often overlap, one can easily specify duplicated resources and end up with unsatisfiable assertions. We illustrate this challenge by specifying the `makeIdGen` function (Figure 6).

In the precondition, the only information we require is that `prefix` is a string. In the postcondition, we would like to have an `IdGenerator(ig, p, c)` predicate, which captures that the object `ig` is an ID generator with prefix `p` and count `c`. Let us first look at only the first three lines, which are standard.

```
IdGenerator(ig, p, c) := types(p: Str, c: Num) * JSObject(o, Object.prototype) *
  DataProp(ig, "getId", gif) * FunctionObject(gif, "getId", gi_sc) *
  DataProp(ig, "reset", rf)  * FunctionObject(rf, "reset", r_sc) *
  Scope(count: c, gi_sc, getId) * Scope(prefix: p, gi_sc, getId) * OChains(getId: gi_sc, reset: r_sc)
```

```
{ types(prefix: Str) }
   makeIdGen(prefix)
{ IdGenerator(ret, prefix, 0) }
```

Fig. 6. Specification of `makeIdGen`

We have that the object `ig` is a standard JS object. It has two properties, `getId` and `reset`, associated with two function objects, respectively corresponding to functions with identifiers `getId` and `reset`, and whose scope chains are respectively denoted by `gi_sc` and `r_sc`. Now, what remains to be specified is that both `getId` and `reset` have access to the same variables `prefix` and `count` in the environment record of `makeIdGen`. We could naively try to capture this with the assertion `Scope(count: c, gi_sc, getId) * Scope(count: c, r_sc, reset)`, but this is duplicated resource. We need a predicate that captures the scope chain overlap between two functions.

The `OChains(f: f_sc, g: g_sc)` predicate states that the scope chains `f_sc` (associated with function `f`) and `g_sc` (associated with function `g`) were created during the same execution of their innermost enclosing function, that is, that their scope chains *maximally overlap*. In the general case, this predicate corresponds to the (pure) JS Logic assertion $\circledast_{0 \le i < n} \mathtt{nth}\,(\mathtt{f\_sc}, i) = \mathtt{nth}\,(\mathtt{g\_sc}, i)$, where $n = \psi^o(\mathtt{f},\ \mathtt{g})$. In particular, `OChains(getId: gi_sc, reset: r_sc)` unfolds to $\mathtt{nth}\,(\mathtt{gi\_sc}, 0) = \mathtt{nth}\,(\mathtt{r\_sc}, 0)\ *$

nth (gi_sc, 1) = nth (r_sc, 1), as ψ^o(getId, reset) = 2. That is, the gi_sc and r_sc coincide on their first two ERs, namely the global object and the ER of mainIdGen.

$$\texttt{OChains}(\texttt{f} : \texttt{f\_sc}, \texttt{g} : \texttt{g\_sc}) := \circledast_{0 \le i < n}(\texttt{nth}(\texttt{f\_sc}, \texttt{i}) = \texttt{nth}(\texttt{g\_sc}, \texttt{i})), \text{ where } \texttt{n} = \psi^o(f, g)$$

The OChains predicate is used together with Scope to capture function closures. First, we specify variables required by multiple closures in a single scope chain using Scope and then state the overlap between these scope chains using OChains, as shown in the fourth line of IdGenerator.

When function closures get more involved, it can be tedious to write all necessary OChains predicates. We offer a more compact predicate, Closure, expressible in terms of Scope and OChains. The Closure(x1 : v1, ... xn : vn; f1 : f1_sc, ..., fm : fm_sc) predicate states that the variables x1, . . ., xn with values v1, . . ., vn are all shared between functions f1, . . ., fm, whose scope chains are given by f1_sc, . . ., fm_sc, and that these scope chains all maximally overlap pairwise. Using Closure, we can rewrite the last line of IdGenerator as Closure(count: c, prefix: p; getId: gi_sc, reset: r_sc).

We also give the specification of the client program in lines 12-14 of Figure 4 (left). In the precondition, we have the function object corresponding to makeIdGen and that the variables ig1, ig2, and id1 all hold the value undefined. The postcondition differs in that the variables ig1 and ig2 hold ID generators with respective prefixes foo and bar and respective count values 1 and 0, and that the variable id1 holds the generated identifier "foo_id_0".

```
{     Scope(makeIdGen : mIG) * FunctionObject(mIG, "makeIdGen", mIG_sc) *
  Scope(ig1 : undefined) * Scope(ig2 : undefined) * Scope (id1 : undefined) }
       var ig1 = makeIdGen("foo"), ig2 = makeIdGen("bar"), id1 = ig1.getId();
{     Scope(makeIdGen : mIG) * FunctionObject(mIG, "makeIdGen", mIG_sc) *
           Scope(ig1: IDG1) * Scope(ig2: IDG2) * Scope(id1: id) *
  IdGenerator(IDG1, "foo", 1) * IdGenerator(IDG2, "bar", 0) * id = "foo_id_0" }
```

JaVerT: Encapsulation. The specification of get shown in Figure 5, albeit correct, does not reflect the key property of the counter implementation, which is encapsulation. That is, since the variable count is not accessible by the clients using the get function, it should not be exposed in the specification of get either. We revisit this specification to demonstrate how to capture encapsulation.

First, we extend the IdGenerator predicate to maintain information about the scope chain in which the ID generator was created:

```
IdGenerator(ig, p, c, ig_sc) := types(p: Str, c: Num) * JSObject(ig, Object.prototype) *
  DataProp(ig, "getId", gif) * FunctionObject(gif, getId, gi_sc) *
  DataProp(ig, "reset", rf)  * FunctionObject(rf, reset, r_sc) *
  Closure(count: c, prefix: p; getId: gi_sc, reset: r_sc, makeIdGen: ig_sc).
```

With this definition in place, the postcondition of makeIdGen (Figure 7, left) can be restated as IdGenerator(ig, prefix, 0, sc), where, as mentioned earlier, sc denotes the scope chain in which this IdGenerator is to be executed. We can now state the specification of get in terms of the IdGenerator predicate (Figure 7, right). Note that, in the precondition, we now need to make sure that the instance of the get function that we are executing is, in fact, the one captured by the IdGenerator.

```
{ types(prefix: Str) }
makeIdGen(prefix)
{ IdGenerator(ig, prefix, 0, sc) }
```

```
{ (this = ig) * OChains(getId: sc, makeIdGen: ig_sc) *
        IdGenerator(ig, prefix, c, ig_sc)) }
getId()
{ IdGenerator(ig, prefix, c + 1, ig_sc) *
  (ret = prefix ++ "_id_" ++ numToString(c)) }
```

Fig. 7. Revisited specifications of makeIdGen (left) and getId (right).

This specification of get no longer exposes the internal state of the ID generator and hints at encapsulation. In general, using function closures in JavaScript does not guarantee encapsulation, and client programs can still access and modify parts of the internal state that are intended to be private, as shown by the priority queue example of Fragoso Santos et al. [2017]. One way of achieving full encapsulation would be to disallow the unfolding of predicates by client programs (in this case, the `IdGenerator` predicate), in the style of Parkinson and Bierman [2005, 2008].

4 JS-2-JSIL: LOGIC-PRESERVING COMPILER

We describe how we use our verification pipeline to move the reasoning from JavaScript to JSIL, solving the verification challenge (V1) of coping with the complexity of JavaScript commands. We introduce JSIL, our intermediate language for JavaScript verification in §4.1. Using an example assignment, we demonstrate how JS-2-JSIL compiles JavaScript to JSIL in §4.2. In §4.3, we introduce JSIL Logic assertions, show how annotations are translated from JS Logic to JSIL Logic by the JS-2-JSIL Logic Translator, and prove correct the translation of assertions and specifications.

4.1 The JSIL Language

JSIL is a simple goto language with top-level procedures and commands operating on object heaps. It natively supports the dynamic features of JavaScript, namely extensible objects, dynamic property access, and dynamic procedure calls.

Syntax of the JSIL Language

Numbers: $n \in \mathcal{N}um$ Booleans: $b \in \mathcal{B}ool$ Strings: $m \in \mathcal{S}tr$ Locations: $l \in \mathcal{L}$ Variables: $\mathtt{x} \in \mathcal{X}_{\mathtt{JSIL}}$

Types: $\tau \in \mathsf{Types}$ Literals: $\lambda \in \mathcal{L}it ::= n \mid b \mid m \mid \mathsf{undefined} \mid \mathsf{null} \mid l \mid \tau \mid fid \mid \mathsf{empty} \mid \lambda_{\mathtt{lst}} \mid \lambda_{\mathtt{set}}$

Expressions: $\mathtt{e} \in \mathcal{E}_{\mathtt{JSIL}} ::= \lambda \mid \mathtt{x} \mid \ominus\, \mathtt{e} \mid \mathtt{e} \oplus \mathtt{e}$

Basic Commands: $\mathtt{bc} \in \mathtt{BCmd} ::= \mathsf{skip} \mid \mathtt{x} := \mathtt{e} \mid \mathtt{x} := \mathsf{new}\,() \mid \mathtt{x} := [\mathtt{e}, \mathtt{e}] \mid [\mathtt{e}, \mathtt{e}] := \mathtt{e} \mid$
$\mathsf{delete}\,(\mathtt{e}, \mathtt{e}) \mid \mathtt{x} := \mathsf{hasField}\,(\mathtt{e}, \mathtt{e}) \mid \mathtt{x} := \mathsf{getFields}\,(\mathtt{e})$

Commands: $\mathtt{c} \in \mathtt{Cmd} ::= \mathtt{bc} \mid \mathsf{goto}\ i \mid \mathsf{goto}\ [\mathtt{e}]\ i,\ j \mid \mathtt{x} := \mathtt{e}(\overline{\mathtt{e}})\ \mathsf{with}\ j \mid \mathtt{x} := \phi(\overline{\mathtt{x}})$

Procedures : $\mathtt{proc} \in \mathtt{Proc} ::= \mathsf{proc}\ fid(\overline{\mathtt{x}})\{\overline{\mathtt{c}}\}$

Notation : $\overline{\mathtt{x}}$, $\lambda_{\mathtt{lst}}$, $\overline{\mathtt{e}}$, and $\overline{\mathtt{c}}$, respectively, denote lists of variables, literals, expressions, and commands. $\lambda_{\mathtt{set}}$ denotes a set of literals.

JSIL literals, $\lambda \in \mathcal{L}it$, include JavaScript literals, as well as procedure identifiers fid, types τ, the special value empty, and lists and sets of literals. JSIL expressions, $\mathtt{e} \in \mathcal{E}_{\mathtt{JSIL}}$, include JSIL literals, JSIL program variables $\mathtt{x}$, and a variety of unary and binary operators.

The JSIL basic commands provide the machinery for the management of extensible objects and do not affect control flow. They include skip , variable assignment, object creation, property access, property assignment, property deletion, membership check, and property collection.

The JSIL commands include JSIL basic commands and commands related to control flow: conditional and unconditional gotos; dynamic procedure calls; and ϕ-node commands. The two goto commands are standard: $\mathsf{goto}\ i$ jumps to the i-th command of the active procedure, and $\mathsf{goto}\ [\mathtt{e}]\ i,\ j$ jumps to the i-th command if $\mathtt{e}$ evaluates to `true`, and to the j-th otherwise. The dynamic procedure call $\mathtt{x} := \mathtt{e}(\overline{\mathtt{e}})\ \mathsf{with}\ j$ first obtains the procedure name and arguments by evaluating $\mathtt{e}$ and $\overline{\mathtt{e}}$, respectively, then executes the appropriate procedure with these arguments, and finally assigns its return value to $\mathtt{x}$. Control is transferred to the next command if the procedure does not raise an error, or to the j-th command otherwise. Finally, the ϕ-node command $\mathtt{x} := \phi(\mathtt{x}_1, \ldots, \mathtt{x}_n)$ is interpreted as follows: there exist n paths via which this command can be reached during the execution of the program; the value assigned to $\mathtt{x}$ is $\mathtt{x}_i$ if and only if the i-th path was taken. We include ϕ-nodes in JSIL to directly support Static-Single-Assignment (SSA), well-known to simplify analysis [Cytron et al. 1989]. The JS-2-JSIL compiler generates JSIL code directly in SSA.

A JSIL program $\mathsf{p} \in \mathsf{P}$ is a set of top-level procedures $\mathsf{proc}\ \mathit{fid}(\overline{\mathsf{x}})\{\overline{\mathsf{c}}\}$, where fid is the name of the procedure, $\overline{\mathsf{x}}$ its sequence of formal parameters, and its body $\overline{\mathsf{c}}$ is a *command list* consisting of a numbered sequence of JSIL commands. We use $\mathsf{p}_{\mathit{fid}}$ and $\mathsf{p}_{\mathit{fid}}(i)$ to refer, respectively, to procedure fid of program p and to the i-th command of that procedure. Every JSIL program contains a special procedure `main` , corresponding to the entry point of the program. JSIL procedures do not explicitly return. Instead, each procedure has two special command indexes, i_{nm} and i_{er}, that, when jumped to, respectively cause it to return normally or return an error. Also, each procedure has two dedicated variables, `ret` and `err`. When a procedure jumps to i_{nm}, it returns normally with the return value `ret`; when it jumps to i_{er}, it returns an error, with the error value `err`.

JSIL Operational Semantics. We introduce the JSIL semantic judgement for program behaviour; the full JSIL semantics is omitted due to lack of space. A JSIL variable store, $\rho \in \mathcal{S}to$, is a mapping from JSIL variables to JSIL values, and a JSIL heap, $\mathsf{h} \in \mathcal{H}_{\mathsf{JSIL}}$, is a mapping from pairs of locations and property names (strings) to JSIL values, $\mathsf{v} \in \mathcal{V}_{\mathsf{JSIL}}$, which coincide with the JSIL literals. The JSIL semantic judgement has the form $\mathsf{p} \vdash \langle \mathsf{h}, \rho, j, i \rangle \Downarrow_{\mathit{fid}} \langle \mathsf{h}', \rho', o \rangle$, meaning that the evaluation of procedure fid of program p, starting from its i-th command, to which we arrived from its j-th command, in the heap h and store ρ, generates the heap h', the store ρ', and returns the outcome o. JSIL outcomes are of the form $\mathit{fl}\langle \mathsf{v} \rangle$, where $\mathit{fl} \in \{\mathsf{nm}, \mathsf{er}\}$ denotes the return mode of the function.

4.2 JS-2-JSIL: Compilation by Example

The JS-2-JSIL compiler targets the strict mode of the ES5 English standard (ES5 Strict). ES5 Strict is a variant of ES5 that intentionally has slightly different semantics, exhibiting better behavioural properties, such as being lexically scoped. It is developed by the ECMAScript committee, is recommended for use by the committee and professional developers [Flanagan 1998], and is widely used by major industrial players: for example, Google's V8 engine [Google 2017] and Facebook's React library [Facebook 2017]. We believe that ES5 Strict is the correct starting point for JavaScript verification.

We illustrate how JS-2-JSIL compiles JavaScript code to JSIL code using an assignment from our key-value map example (§3.4): the assignment `contents[k] = v` from the function `put`. This seemingly innocuous statement has non-trivial behaviour and triggers a number of JavaScript internal functions, as shown below. First, however, we need to introduce JavaScript references.

References. References are JS internals that appear, for example, as a result of evaluating a left-hand side of an assignment, and represent resolved property bindings. They consist of a base (normally an object location) and a property name (a string), telling us where in the heap we can find the property we are looking for. The base can hold the location either of a standard object (*object reference*) or of an ER (*variable reference*). To obtain the associated value, the reference needs to be dereferenced, which is performed by the `GetValue` internal function. In JSIL, we encode references as three-element lists, containing the reference type (`"o"` or `"v"`), the base, and the property name.

Compiling the Assignment. We are now ready to go line-by-line through the compilation of the assignment `contents[k] = v`, which is given in Figure 8.

(1) We first evaluate the the property accessor `contents[k]` and obtain the corresponding reference. Evaluation of property accessors is described in §11.2.1 of the ES5 standard, and is line-by-line reflected in lines 1-9 of the JSIL code. The resulting reference, `["o", x_2_v, x_4_s ]`, points to the property denoted by `k` of the object denoted by `contents`.
(2) Next, we evaluate the variable `v`. Here, we need to understand within which ER `v` is defined; as it is a parameter of the `put` function, it will be in the ER corresponding to `put`, i.e. the second element of the scope chain (line 10). The appropriate reference, `["v", x_7, "v"]`, is then constructed in line 11. This code is automatically generated using the scope clarification function.

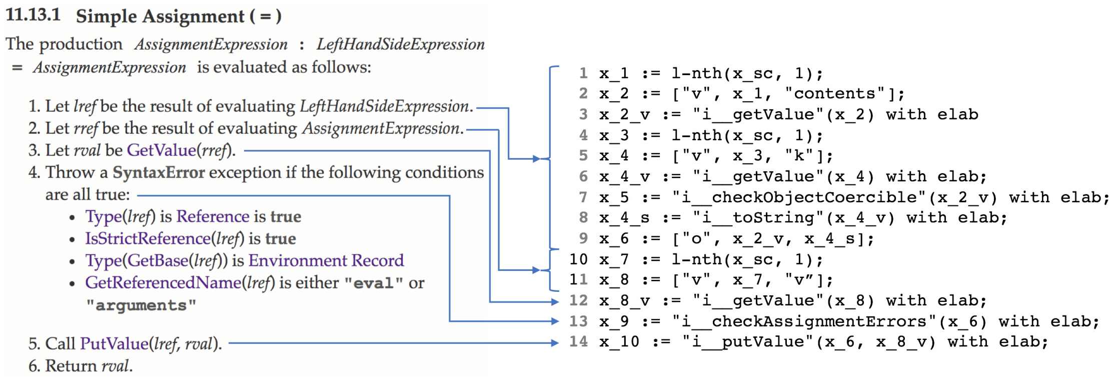

Fig. 8. Compiling `contents[k] = v` to JSIL by closely following the ES5 Standard.

(3) Next, the obtained right-hand-side reference is dereferenced using the `GetValue` internal function (ES5 standard, §8.7.1). Any call to an internal function gets translated to JSIL as a procedure call to our corresponding reference implementation, in this case `i__getValue` (line 12).

(4) In ES5 Strict, the identifiers `eval` and `arguments` may not appear as the left-hand side of an assignment (for example, `eval` = 42), and this step enforces this restriction. We do not inline the conditions every time, but instead call a JSIL procedure `i__checkAssignmentErrors` (line 13), which takes as a parameter a reference and throws a syntax error if the conditions are met.

(5) The actual assignment is performed by calling the `PutValue` internal function (ES5 standard, §8.7.2), translated to JSIL as a procedure call to our reference implementation (line 14).

(6) In JavaScript, every statement returns a value. JS-2-JSIL, when given a statement, returns the list of corresponding JSIL commands and the variable that stores the return value of that statement. In this example, JS-2-JSIL returns the presented code and the variable `x_8_v`.

This example illustrates the following important points about JS-2-JSIL:

- *Our compilation from JavaScript to JSIL closely follows the ES5 standard.* Out of the 14 lines of compiled JSIL code, 8 have a direct counterpart in the standard. The remaining six deal with scoping, where a difference is expected due to our use of the closure clarification function.
- *JS-2-JSIL moves a substantial part of the complexity of JavaScript from the reasoning to the compiled code.* As discussed in §2, program-logic-based verification is not feasible for JavaScript due to the complexity of its constructs. JS-2-JSIL moves this complexity to the compiled JSIL code. There are more lines of JSIL to be analysed when compared to the original JS code (for example, the key-value map example compiles to 354 lines of JSIL code), but JSIL logic is very simple, making this analysis tractable. However, the fundamental dynamic features of JavaScript cannot be compiled away; they remain in JSIL and JSIL Logic and are resolved by JSIL Verify, as described in §5.
- *JS-2-JSIL maintains the level of abstraction of the ES5 standard.* By this, we refer to the fact that the compilation never inlines function bodies. A function call in the ES5 standard is always compiled to a procedure call in JSIL. For example, a call to an internal function in the standard (lines 3 and 5 of Figure 8, left) is translated to a call to a JSIL reference implementation of that internal function (lines 12 and 14 of Figure 8, right)). One tangible benefit of this approach is that it makes the resulting compiled JSIL code much more readable and visually closer to the ES5 standard.

Compiling Function Literals. Each ES5 Strict function literal `function fid(x1, ..., xn) { ... }` is compiled to a JSIL procedure `procedure fid(xsc, xthis, x1, ..., xn) { ... }`, whose name is the identifier of the original function and whose first two arguments are bound, respectively, to the scope chain and the `this` object active during the evaluation of the function body. The remaining arguments correspond to the original arguments of the function.

4.3 JS-2-JSIL Logic Translator

JaVerT verifies programs annotated with pre- and postconditions, loop invariants, and instructions for folding and unfolding of user-defined predicates. The JSIL Logic Translator translates these annotations to equivalent annotations in JSIL Logic, and then integrates them into the compiled JSIL code. It also automatically inserts additional fold/unfold annotations for the `Pi` predicate, as they are required by some of the internal functions (see §5.3 for more details).

JSIL Logic Assertions

$$V \in \mathcal{V}^L_{\texttt{JSIL}} ::= \mathsf{v} \mid \varnothing \qquad\qquad E \in \mathcal{E}^L_{\texttt{JSIL}} ::= V \mid \mathsf{x} \mid x \mid \ominus E \mid E \oplus E$$

$$\tau \in \mathsf{Types} ::= \mathtt{Num} \mid \mathtt{Bool} \mid \mathtt{Str} \mid \mathtt{Undef} \mid \mathtt{Null} \mid \mathtt{Obj} \mid \mathtt{List} \mid \mathtt{Set} \mid \mathtt{Type}$$

$$P, Q \in \mathcal{AS}_{\texttt{JSIL}} ::= \mathsf{true} \mid \mathsf{false} \mid E = E \mid E \leq E \mid P \wedge Q \mid \neg P \mid P * Q \mid \exists \mathsf{x}.P \mid \\ \mathsf{emp} \mid (E, E) \mapsto E \mid \mathsf{emptyFields}(E \mid E) \mid \mathsf{types}(X_i : \tau_i|_{i=1}^n)$$

JSIL Logic Assertions. There is a strong correspondence between JavaScript and JSIL at the level of the logics. JSIL logical values, $V \in \mathcal{V}^L_{\texttt{JSIL}}$, consist of JSIL values extended with $\varnothing$, subsuming JS logical values. JSIL logical expressions, $E \in \mathcal{E}^L_{\texttt{JSIL}}$, coincide with JS logical expressions, except that they do not contain `sc` and `this`. JSIL types coincide with JavaScript types. Finally, as ES5 Strict heaps are by design a proper subset of JSIL heaps, we have that JSIL Logic assertions, $P, Q \in \mathcal{AS}_{\texttt{JSIL}}$, coincide with JS Logic assertions.

JS-2-JSIL: Logic Translation. Translating JS Logic assertions to JSIL Logic assertions amounts to replacing the occurrences of the `sc` and `this` special logical values of JS Logic with the variables `xsc` and `xthis` of JSIL logic, which hold their associated values at the JSIL level. The translation of a JS Logic assertion P to JSIL Logic is denoted by $\mathcal{T}(P)$.

Translation Correctness: Assertions. We define satisfiability for JSIL Logic assertions with respect to *abstract heaps*, which differs from concrete heaps in that they may map object properties to the special value $\varnothing$. The satisfiability relation for JSIL Logic assertions has the form: $H, \rho, \epsilon \models P$, where: **(1)** H is an abstract heap; **(2)** ρ is a JSIL variable store; **(3)** and ϵ is a JSIL logical environment, mapping JSIL logical variables to JSIL values. The satisfiability relation for JSIL Logic assertions builds on the semantics of JSIL logical expressions. A JSIL logical expression E is interpreted with respect to ρ and ϵ, written $[\![E]\!]^\epsilon_\rho$. Both the satisfiability relation and the expression interpretation are mostly standard; we show the non-standard cases below. We also use a function TypeOf, which given a JSIL value, outputs its type.

Interpretation of JSIL Logic Expressions and Satisfiability Relation for Assertions (fragment)

Semantics of Logical Expressions: $[\![V]\!]^\epsilon_\rho \triangleq V \quad [\![\mathsf{x}]\!]^\epsilon_\rho \triangleq \rho(x) \quad [\![x]\!]^\epsilon_\rho \triangleq \epsilon(\mathsf{x})$

Satisfiability Relation:

$$H, \rho, \epsilon \models \mathsf{emptyFields}(E_1 \mid E_2) \quad \Leftrightarrow H = \biguplus_{m \notin \{[\![E_2]\!]^\epsilon_\rho\}}(([\![E_1]\!]^\epsilon_\rho, m) \mapsto \varnothing)$$

$$H, \rho, \epsilon \models \mathsf{types}(X_i : \tau_i|_{i=1}^n) \quad \Leftrightarrow H = \mathsf{emp} \text{ and for all } i \in \{1, \ldots, n\}, \mathsf{TypeOf}([\![E]\!]^\epsilon_\rho) = \tau_i$$

Satisfiability of JS Logic assertions, $H, \rho, L, l_t, \epsilon \models P$, is defined analogously, except that JS logical expressions are interpreted not only with respect to the JS store ρ and JS logical environment ϵ, but also the current scope chain L and the binding of the this object l_t. Given how close the semantics of JS and JSIL assertions are, it immediately follows that:

$$H, \rho, L, l_t, \epsilon \models P \iff H, \rho[\mathtt{xsc} \mapsto L, \mathtt{xthis} \mapsto l_t], \epsilon \models \mathcal{T}(P)$$

Translation Correctness: Specifications. First, we define what it means for a JSIL Logic specification to be valid. This definition is expressed in terms of the JSIL semantic judgement, $\mathsf{p} \vdash \langle \mathsf{h}, \rho, j, i \rangle \Downarrow_{\mathit{fid}} \langle \mathsf{h}', \rho', o \rangle$, given in §4.1. Also, it makes use of a deabstraction function $\lfloor . \rfloor : \mathcal{H}^\emptyset_{\texttt{JSIL}} \rightarrow$

$\mathcal{H}_{\text{JSIL}}$, transforming abstract JSIL heaps to concrete JSIL heaps. Intuitively, $\lfloor H \rfloor$ denotes the concrete JSIL heap obtained by removing the cells of H that are mapped to $\varnothing$.

Definition 4.1 (Validity of JSIL Logic Specifications). A JSIL Logic specification $\{P\}\, \mathit{fid}(\overline{x})\, \{Q\}$ for return mode fl is valid with respect to a program p, written $\mathsf{p}, \mathit{fl} \vDash \{P\}\, \mathit{fid}(\overline{x})\, \{Q\}$, if and only if, for all logical contexts (H, ρ, ϵ), heaps h_f, stores ρ_f, flags fl', and JSIL values v, it holds that:

$$\begin{array}{c} H, \rho, \epsilon \models P \ \wedge\ \mathsf{p} \vdash \langle \lfloor H \rfloor, \rho, -, 0 \rangle \Downarrow_{\mathit{fid}} \langle \mathsf{h}_f, \rho_f, \mathit{fl}'\langle \mathsf{v} \rangle \rangle \implies \\ \mathit{fl}' = \mathit{fl} \ \wedge\ \exists H_f \,.\, H_f, \rho_f, \epsilon \models Q \ \wedge\ \lfloor H_f \rfloor = \mathsf{h}_f \end{array}$$

The validity of JS Logic specifications is defined in a similar way, with respect to an ES5 Strict semantic relation of the form $s, L, l_t \vdash \langle h, \rho \rangle \Downarrow_{\mathit{fid}} \langle h_f, o \rangle$, meaning that, given a JavaScript program s, scope chain list L and the this object l_t, when executing the function of s with identifier fid and parameter values given by ρ in the heap h, one obtains the final heap h_f and outcome o. We write $s, \mathit{fl} \vDash \{P\}\, \mathit{fid}(\overline{x})\, \{Q\}$ to denote that a JS Logic specification $\{P\}\, \mathit{fid}(\overline{x})\, \{Q\}$ for return mode fl is valid with respect to a JavaScript program s.

To be able to state the next theorem, we lift the translation of assertions to specifications: $\mathcal{T}(\{P\}\, \mathit{fid}(\overline{x})\, \{Q\}) = \{\mathcal{T}(P)\}\, \mathit{fid}(\mathtt{xsc}, \mathtt{xthis}, \overline{x})\, \{\mathcal{T}(Q)\}$. Also, we say that a JS-2-JSIL compiler C is correct if compiled programs preserve the behaviour of their original versions. Put formally:

$$s, L, l_t \vdash \langle h, \rho \rangle \Downarrow_{\mathit{fid}} \langle h_f, \mathit{fl}\langle v \rangle \rangle \iff \exists \rho_f \,.\, C(s) \vdash \langle h, \rho[\mathtt{xsc} \mapsto L, \mathtt{xthis} \mapsto l_t], -, 0 \rangle \Downarrow_{\mathit{fid}} \langle h_f, \rho_f, \mathit{fl}\langle v \rangle \rangle$$

Due to our extensive validation, which we discuss in detail in §6.1, we strongly believe that the JS-2-JSIL compiler is correct. Finally, Theorem 4.2 states that under the assumption of a correct compiler, a JavaScript specification is valid if and only if its translated JSIL specification is valid.

Theorem 4.2 (JS-2-JSIL Logic correspondence). *Given a* correct *JS-2-JSIL compiler, C, for any JavaScript program s, return mode fl, and JS specification $\{P\}\, \mathit{fid}(\overline{x})\, \{Q\}$, it holds that:*

$$s, \mathit{fl} \vDash \{P\}\, \mathit{fid}(\overline{x})\, \{Q\} \iff C(s), \mathit{fl} \vDash \mathcal{T}(\{P\}\, \mathit{fid}(\overline{x})\, \{Q\})$$

5 JSIL VERIFY

We present JSIL Verify, a semi-automatic verification tool for JSIL, and discuss how it tackles the verification challenge of reasoning about the dynamic features of JavaScript (V2). Given a JSIL program annotated with the specifications of its procedures, JSIL Verify checks whether the program procedures satisfy their specifications. JSIL Verify consists of: **(1)** a symbolic execution engine based on JSIL Logic, the sound separation logic for JSIL, presented in §5.1; and **(2)** an entailment engine for resolving frame inference and entailment questions, presented in §5.2. Finally, in §5.3, we explain how we used JSIL Verify to specify and verify the JSIL implementations of the JavaScript internal functions and how these specifications are used in the verification of compiled JavaScript code (V3).

5.1 JSIL Verify: Symbolic Execution

Axiomatic Semantics of Basic Commands. The Hoare triples for the JSIL basic commands are of the form $\{P\}\mathsf{bc}\{Q\}$, and are interpreted as: "if bc is executed in a state satisfying P, then, if it terminates, it will do so in a state satisfying Q". We assume that JSIL programs are in SSA form, taking away the need for standard substitutions in many of the axioms. Below, we give selected axioms for the JSIL basic commands. We write $E_1 \doteq E_2$ to denote $E_1 = E_2 \wedge \mathsf{emp}$. The Get Fields axiom states that if the object bound to e *only* contains the properties denoted by X_1, ..., X_n, then, after execution of $\mathsf{x} := \mathsf{getFields}\,(\mathsf{e})$, x will be bound to a list containing precisely X_1, ..., X_n in an order described by the ord predicate, which stands for an implementation-dependent ordering of property names. The Property Deletion axiom forbids the deletion of *@proto* properties. The

Object Creation axiom states that the new object at x only contains the *@proto* property with value null. The remaining axioms are straightforward.

Axiomatic Semantics of Basic Commands (selected axioms): $\{P\}\mathsf{bc}\{Q\}$

Property Access
$$\frac{P \equiv (\mathsf{e}_1, \mathsf{e}_2) \mapsto \mathrm{X} * \mathrm{X} \neq \varnothing}{\{P\}\ \mathsf{x} := [\mathsf{e}_1, \mathsf{e}_2]\ \{P * \mathsf{x} \doteq \mathrm{X}\}}$$

Get Fields
$$\frac{P \equiv ((\mathsf{e}, \mathrm{X}_i) \mapsto \mathrm{Y}_i|_{i=1}^{n}) * \mathsf{emptyFields}(\mathsf{e} \mid \{\mathrm{X}_i|_{i=1}^{n}\}) * (\mathrm{Y}_i \neq \varnothing|_{i=1}^{n})}{\{P\}\ \mathsf{x} := \mathsf{getFields}\ (\mathsf{e})\ \{P * (\mathsf{x} \doteq [\mathrm{X}_1, \ldots, \mathrm{X}_n]) * (\mathsf{ord}\ (\mathsf{x}) \doteq \mathsf{true})\}}$$

Property Assignment
$$\{(\mathsf{e}_1, \mathsf{e}_2) \mapsto \_\}\ \ [\mathsf{e}_1, \mathsf{e}_2] := \mathsf{e}_3\ \ \{(\mathsf{e}_1, \mathsf{e}_2) \mapsto \mathsf{e}_3\}$$

Property Deletion
$$\frac{P \equiv (\mathsf{e}_1, \mathsf{e}_2) \mapsto \mathrm{X} * \quad \mathrm{X} \neq \varnothing * \mathsf{e}_2 \neq @proto}{\{P\}\ \mathsf{delete}\ (\mathsf{e}_1, \mathsf{e}_2)\ \{(\mathsf{e}_1, \mathsf{e}_2) \mapsto \varnothing\}}$$

Object Creation
$$\frac{Q = (\mathsf{x}, @proto) \mapsto \mathsf{null} * \quad \mathsf{emptyFields}(\mathsf{x} \mid \{@proto\})}{\{\mathsf{emp}\}\ \mathsf{x} := \mathsf{new}\ ()\ \{Q\}}$$

Symbolic Execution. Our goal is to use symbolic execution to prove the specifications of JSIL procedures. As procedures may call other procedures, we group specifications in *specification environments*, $\mathsf{SE} : \mathcal{F}id \rightharpoonup \mathcal{F}lag \rightharpoonup \mathcal{S}pec$, mapping procedure identifiers and return modes to specifications. To avoid clutter, we assume in the formalisation that each procedure has a single specification per return mode. Hence, $\mathsf{SE}(\mathit{fid}, \mathit{fl}) = \mathit{spec}$ means that *spec* is the specification of the procedure with identifier *fid* for the return mode *fl*. In the following, we use the terms symbolic state and assertion interchangeably. Below, we give all of the operational rules of the symbolic execution. Rules have the form $\mathsf{p}, \mathit{fid}, \mathsf{SE}, \mathit{fl} \vdash \langle P, k, i \rangle \rightsquigarrow \langle Q, j \rangle$, meaning that: **(1)** we are currently symbolically executing the code of the procedure with identifier *fid* in the JSIL program p assuming the specification environment SE; **(2)** the symbolic execution of the entire procedure must terminate with return mode *fl*; and **(3)** the symbolic execution of the i-th command on P results in Q when j is the index of the next command to be executed, whilst k is the index of the command executed before i. As p, *fid*, SE, and *fl* do not change during symbolic execution, we leave them implicit. In the operational rules, we write $\mathsf{post}(\mathit{spec})$ to denote the postcondition of *spec*.

Operational Rules for JSIL Logic Symbolic Execution: $\mathsf{p}, \mathit{fid}, \mathsf{SE}, \mathit{fl} \vdash \langle P, k, i \rangle \rightsquigarrow \langle Q, j \rangle$

Basic Command
$$\frac{\mathsf{p}_{\mathit{fid}}(i) = \mathsf{bc} \qquad \{P\}\ \mathsf{bc}\ \{Q\}}{\langle P, -, i \rangle \rightsquigarrow \langle Q, i+1 \rangle}$$

Frame Rule
$$\frac{\langle P, i, j \rangle \rightsquigarrow \langle Q, k \rangle \qquad i \notin \{i_{\mathsf{nm}}, i_{\mathsf{er}}\}}{\langle P * R, i, j \rangle \rightsquigarrow \langle Q * R, k \rangle}$$

Phi-Assignment
$$\frac{\mathsf{p}_{\mathit{fid}}(i) = \mathsf{x} := \phi(\mathsf{x}_1, \ldots, \mathsf{x}_n) \qquad j \overset{k}{\mapsto}_{\mathit{fid}} i}{\langle P, j, i \rangle \rightsquigarrow \langle P * (\mathsf{x} \doteq \mathsf{x}_k), i+1 \rangle}$$

Goto
$$\frac{\mathsf{p}_{\mathit{fid}}(i) = \mathsf{goto}\ k}{\langle P, -, i \rangle \rightsquigarrow \langle P, k \rangle}$$

Cond. Goto - True
$$\frac{\mathsf{p}_{\mathit{fid}}(i) = \mathsf{goto}\ [\mathsf{e}]\ k_1,\ k_2}{\langle P, -, i \rangle \rightsquigarrow \langle P * \mathsf{e} \doteq \mathsf{true}, k_1 \rangle}$$

Cond. Goto - False
$$\frac{\mathsf{p}_{\mathit{fid}}(i) = \mathsf{goto}\ [\mathsf{e}]\ k_1,\ k_2}{\langle P, -, i \rangle \rightsquigarrow \langle P * \mathsf{e} \doteq \mathsf{false}, k_2 \rangle}$$

Consequence
$$\frac{\langle P, i, j \rangle \rightsquigarrow \langle Q, k \rangle \qquad P' \Rightarrow P \qquad Q \Rightarrow Q'}{\langle P', i, j \rangle \rightsquigarrow \langle Q', k \rangle}$$

Existential Elimination
$$\frac{\langle P, i, j \rangle \rightsquigarrow \langle Q, k \rangle \qquad i \notin \{i_{\mathsf{nm}}, i_{\mathsf{er}}\}}{\langle (\exists \mathrm{X}.\, P), i, j \rangle \rightsquigarrow \langle (\exists \mathrm{X}.\, Q), k \rangle}$$

Procedure Call - Normal
$$\frac{\mathsf{p}_{\mathit{fid}}(i) = \mathsf{x} := \mathsf{e}_0(\mathsf{e}_i|_{i=1}^{n_1})\ \mathsf{with}\ j \qquad \mathsf{SE}(\mathit{fid}', \mathsf{nm}) = \{P\}\ \mathit{fid}'(\mathsf{x}_1, \ldots, \mathsf{x}_{n_2})\ \{Q * \mathsf{ret} \doteq \mathsf{e}\} \qquad \mathsf{e}_i = \mathsf{undefined}\ |_{i=n_1+1}^{n2}}{\langle (P[\mathsf{e}_i/\mathsf{x}_i|_{i=1}^{n_2}] * \mathsf{e}_0 \doteq \mathit{fid}'), i \rangle \rightsquigarrow \langle (Q[\mathsf{e}_i/\mathsf{x}_i|_{i=1}^{n_2}] * \mathsf{e}_0 \doteq \mathit{fid}' * \mathsf{x} \doteq \mathsf{e}[\mathsf{e}_i/\mathsf{x}_i|_{i=1}^{n_2}]), i+1 \rangle}$$

Procedure Call - Error
$$\frac{\mathsf{p}_{\mathit{fid}}(i) = \mathsf{x} := \mathsf{e}_0(\mathsf{e}_i|_{i=1}^{n_1})\ \mathsf{with}\ j \qquad \mathsf{SE}(\mathit{fid}', \mathsf{er}) = \{P\}\ \mathit{fid}'(\mathsf{x}_1, \ldots, \mathsf{x}_{n_2})\ \{Q * \mathsf{err} \doteq \mathsf{e}\} \qquad \mathsf{e}_i = \mathsf{undefined}\ |_{i=n_1+1}^{n2}}{\langle (P[\mathsf{e}_i/\mathsf{x}_i|_{i=1}^{n_2}] * \mathsf{e}_0 \doteq \mathit{fid}'), i \rangle \rightsquigarrow \langle (Q[\mathsf{e}_i/\mathsf{x}_i|_{i=1}^{n_2}] * \mathsf{e}_0 \doteq \mathit{fid}' * \mathsf{x} \doteq \mathsf{e}[\mathsf{e}_i/\mathsf{x}_i|_{i=1}^{n_2}]), j \rangle}$$

Normal Return
$$\frac{\mathit{fl} = \mathsf{nm} \qquad Q \vdash \mathsf{post}(\mathsf{SE}(\mathit{fid}, \mathsf{nm}))}{\langle Q, -, i_{\mathsf{nm}} \rangle \rightsquigarrow \langle Q, i_{\mathsf{nm}} \rangle}$$

Error Return
$$\frac{\mathit{fl} = \mathsf{er} \qquad Q \vdash \mathsf{post}(\mathsf{SE}(\mathit{fid}, \mathsf{er}))}{\langle Q, -, i_{\mathsf{er}} \rangle \rightsquigarrow \langle Q, i_{\mathsf{er}} \rangle}$$

We discuss the non-standard rules. The Normal Return rule first checks if the symbolic execution is associated with a nm-mode specification, and then checks if the current symbolic state entails the postcondition of that specification. This rule cannot be used during the symbolic execution of an er-mode specification, as the first check would fail. The Error Return rule is analogous. The Procedure Call - Normal rule checks if the current symbolic state entails the precondition of the nm-specification of the procedure being called, in which case the rule updates the symbolic state with the postcondition of that procedure. The Procedure Call - Error rule is analogous.

The reader may notice that the symbolic execution rules presented above are not syntax-directed. Therefore, we needed to develop a strategy for applying the Frame and Consequence rules. In practice, we apply both rules before the symbolic execution of every basic command and procedure call.

Soundness of Symbolic Execution. Since JSIL programs contain goto operations, we cannot rely on the standard sequential composition rule of Hoare logic to derive specifications for sequences of JSIL commands. Instead, we introduce *proof candidates.* A proof candidate, $\mathsf{pd} \in \mathcal{D} : \mathcal{F}id \times \mathcal{F}lag \times \mathbb{N} \rightharpoonup \wp(\mathcal{AS}_{\mathsf{JSIL}} \times \mathbb{N})$, maps each command in a procedure to a set of possible preconditions, associating each such precondition with the index of the command that led to it. To illustrate, if $(P, j) \in \mathsf{pd}(\mathit{fid}, \mathit{fl}, i)$, then P is the precondition of the i-th command of procedure fid that resulted from the symbolic execution of its j-th command during the symbolic execution associated with the fl-mode specification of fid. A proof candidate is a valid proof derivation *iff* it is *well-formed* (Definition 5.1 below), meaning that **(1)** the set of preconditions of the first command of every procedure contains the precondition of the procedure itself and **(2)** one can symbolically execute every command on all of its possible preconditions. In the definition, we use $i \mapsto_{\mathit{fid}} j$ to denote that i is an immediate predecessor of j, and $i \overset{k}{\mapsto}_{\mathit{fid}} j$ to state that i is the k-th element of the list containing all the predecessors of j in chronological order.

Definition 5.1 (Well-formed proof candidate). Given a program $\mathsf{p} \in \mathsf{P}$ and a specification environment $\mathsf{SE} \in \mathcal{S}tr \rightharpoonup \mathcal{F}lag \rightharpoonup \mathcal{S}pec$, we say that a proof candidate $\mathsf{pd} \in \mathcal{D}$ is *well-formed* with respect to p and SE, written $\mathsf{p}, \mathsf{SE} \vdash \mathsf{pd}$, if and only if for all procedures fid in p, and index i the following statements hold:

(1) $\forall \mathit{fl}, P, Q.\ \mathsf{SE}(\mathit{fid}, \mathit{fl}) = \{P\}\mathit{fid}(\overline{\mathsf{x}})\{Q\} \iff \mathsf{pd}(\mathit{fid}, \mathit{fl}, 0) = \{(P, 0)\}$

(2) $\forall \mathit{fl}, P, k.\ (P, k) \in \mathsf{pd}(\mathit{fid}, \mathit{fl}, i) \ \wedge\ (P \nvdash \mathsf{false}) \implies$

$$\Big(\forall j.\ i \mapsto_{\mathit{fid}} j \implies \exists Q.\ (Q, i) \in \mathsf{pd}(\mathit{fid}, \mathit{fl}, j) \ \wedge\ \mathsf{p}, \mathit{fid}, \mathsf{SE}, \mathit{fl} \vdash \langle P, k, i \rangle \rightsquigarrow \langle Q, j \rangle\Big)$$
$$\vee\ \Big(i \in \{i_{\mathsf{nm}}, i_{\mathsf{er}}\} \implies \mathsf{p}, \mathit{fid}, \mathsf{SE}, \mathit{fl} \vdash \langle P, k, i \rangle \rightsquigarrow \langle P, i \rangle\Big)$$

The operational rules for JSIL symbolic execution are sound with respect to the JSIL operational semantics. Hence, if we have that there is a well-formed proof candidate derivation with respect to a program p and specification environment SE, then we have that all of the the specifications in the co-domain of SE are valid.

Theorem 5.2 (Soundness of Symbolic Execution for JSIL). *For all JSIL programs* p *and specification environments* SE*, if there exists a proof candidate* $\mathsf{pd} \in \mathcal{D}$ *such that* $\mathsf{p}, \mathsf{SE} \vdash \mathsf{pd}$*, then:*

$$\forall \mathit{fid}, \mathit{fl}, P, Q, \overline{\mathsf{x}}\ .\ \mathsf{SE}(\mathit{fid}, \mathit{fl}) = \{P\}\mathit{fid}(\overline{\mathsf{x}})\{Q\} \implies \mathsf{p}, \mathit{fl} \models \{P\}\mathit{fid}(\overline{\mathsf{x}})\{Q\}$$

5.2 JSIL Verify: Entailment Engine

Frame Inference. As JSIL features dynamic property access, the field of a cell assertion is an arbitrary logical expression and not a concrete string. This makes symbolic evaluation of object manipulation commands non-trivial. Consider, for instance, the property assignment $[\mathsf{e}_1, \mathsf{e}_2] := \mathsf{e}_3$. To symbolically execute this command in a symbolic state P, JSIL Verify must solve the following instance of the frame inference problem (FIP) $P \vdash (o, p) \mapsto - * ?F$, where $?F$ denotes the resources to be framed off. In this case, solving the FIP involves: **(1)** traversing all the cell assertions $(E_1, E_2) \mapsto -$ in P, checking for each one whether $P \vdash \mathsf{e}_i = E_i \;|_{i=1,2}$; and **(2)** traversing all the emptyFields assertions $\mathsf{emptyFields}(E_1 \mid E_2)$ in P, checking for each one whether $P \vdash \mathsf{e}_1 = E_1$ and $P \vdash \mathsf{e}_2 \notin E2$ (for the case in which the required resource is captured by the emptyFields assertion).

Similarly to Berdine et al. [2005b], given the FIP $P \vdash Q * [?F]$, what we do first is decompose P and Q into pairs of the form (Σ, Π), where Σ and Π denote, respectively, their spatial and pure parts. Hence, what we are left with is $(\Sigma_p, \Pi_p) \vdash (\Sigma_q, \Pi_q) * [?F]$, which can then be further decomposed into: **(i)** $(\Sigma_p, \Pi_p) \vdash (\Sigma_q, \mathsf{True}) * [?F]$ and the pure entailment **(ii)** $\Pi_p \vdash \Pi_q$. Below, we present a proof system for solving **(i)**, which we rewrite, for readability, as $\Sigma_p \mid \Pi_p \vdash \Sigma_q * [?F]$. We note that this proof system makes use of a pure entailment oracle in order to check entailments between pure assertions of the form $\Pi_1 \vdash \Pi_2$.

Proof System for Frame Inference - $\Sigma_1 \mid \Pi \vdash \Sigma_2 * [?F]$

Cell-Cell
$$\frac{\Pi \vdash E_i = E_i' \;|_{i=1,2,3} \qquad \Sigma_1 \mid \Pi \vdash \Sigma_2 * [?F]}{\Sigma_1 * (E_1, E_2) \mapsto E_3 \mid \Pi \vdash \Sigma_2 * (E_1', E_2') \mapsto E_3' * [?F]}$$

Frame
$$\frac{\Sigma_1 \mid \Pi \vdash \Sigma_2 * [?F]}{\Sigma_1 * \Sigma \mid \Pi \vdash \Sigma_2 * [?F * \Sigma]}$$

Emp
$$\mathsf{emp} \mid \Pi \vdash \mathsf{emp} * [\mathsf{emp}]$$

EmptyFields-None-Cell
$$\frac{\Pi \vdash E_1 = E_1' \qquad \Pi \vdash E_2' \notin E_2 \qquad \Sigma_1 * \mathsf{emptyFields}(E_1 \mid E_2 \cup \{ E_2' \}) \mid \Pi \vdash \Sigma_2 * [?F]}{\Sigma_1 * \mathsf{emptyFields}(E_1 \mid E_2) \mid \Pi \vdash \Sigma_2 * (E_1', E_2') \mapsto \varnothing * [?F]}$$

EmptyFields-EmptyFields-Extra-Resource-Left
$$\frac{\Pi \vdash E_0 = E_0' \qquad \Pi \vdash E \uplus \{E_i \,|_{i=1}^{k}\} = E' \qquad \Sigma_1 * \circledast_{1 \leq i \leq k} (E_0, E_i) \mapsto \varnothing \mid \Pi \vdash \Sigma_2 * [?F]}{\Sigma_1 * \mathsf{emptyFields}(E_0 \mid E) \mid \Pi \vdash \Sigma_2 * \mathsf{emptyFields}(E_0' \mid E') * [?F]}$$

EmptyFields-EmptyFields-Extra-Resource-Right
$$\frac{\Pi \vdash E_0 = E_0' \qquad \Pi \vdash E \backslash \{E_i \,|_{i=1}^{k}\} = E' \qquad \Sigma_1 \mid \Pi \vdash \Sigma_2 * [?F]}{\Sigma_1 * \circledast_{1 \leq i \leq k} (E_0, E_i) \mapsto \varnothing * \mathsf{emptyFields}(E_0 \mid E) \mid \Pi \vdash \Sigma_2 * \mathsf{emptyFields}(E_0' \mid E') * [?F]}$$

The Cell-Cell, Frame, and Emp rules are standard, whereas the remaining three deal with negative resource and are tightly connected to the dynamic nature of JSIL and, by extension, JavaScript. They are all based on the following insight: $\mathsf{emptyFields}(E_1 \mid E_2) * E_1 \doteq E_1' * E_2' \dot{\notin} E_2 \Leftrightarrow \mathsf{emptyFields}(E_1 \mid E_2 \cup \{E_2'\}) * (E_1', E_2') \mapsto \varnothing$, which shows how a single none-cell can be taken out of or put into an `emptyFields` assertion, highlighting how the footprint of `emptyFields` is contravariant on the cardinality of the set E_2. The EmptyFields-None-Cell rule places the left-to-right direction of this equivalence into the context of the FIP. The remaining two rules, EmptyFields-EmptyFields-Extra-Resource-Left and EmptyFields-EmptyFields-Extra-Resource-Right, illustrate the two scenarios in which an `emptyFields` assertion for the same object exists on both sides of the FIP. In the first rule, the footprint of `emptyFields` on the left-hand-side is greater than that of the `emptyFields` on the right. There, we have to carry the extra resource, $\circledast_{1 \leq i \leq k} (E_0, E_i) \mapsto \varnothing$, into the left-hand-side of the remaining derivation. In the second rule, the extra resource is present immediately on the left-hand-side of the FIP, and no `emptyFields` are carried over into the remaining derivation. Note that, in the first rule, the union $E \uplus \{E_i \,|_{i=1}^{k}\}$ in the premise has to be disjoint to avoid resource duplication. In the second rule, this is taken care of by the separating conjunction.

Consider, for example, the symbolic execution of the compilation of `put(k, v)` from §3.4 on a symbolic state P, such that the key to be inserted, k, is valid and not contained in the given map. Then, to symbolically execute the compilation of `contents[k] = v`, we must prove that k is not defined in `contents`, which implies solving the following FIP: $P \vdash (\mathrm{contents}, \mathrm{k}) \mapsto \varnothing * [?F]$, with $P = \mathsf{emptyFields}(\mathrm{contents} \mid \mathrm{keys} \cup \{ \mathrm{hOP} \}) * \Sigma * \Pi$ and $\Pi = \mathit{validKey}(\mathrm{k}) \wedge \mathrm{k} \notin \mathrm{keys}$, where Σ denotes the remaining spatial resource and hOP denotes the string `hasOwnProperty`. Figure 9 shows

$$\dfrac{\dfrac{\Pi \vdash \mathsf{k} \notin \mathsf{keys} \cup \{\,\mathsf{hOP}\,\} \qquad \dfrac{\mathsf{emp} \mid \Pi \vdash \mathsf{emp} * [\mathsf{emp}]}{\Sigma_F \mid \Pi \vdash \mathsf{emp} * [\Sigma_F]}\ \textsc{Emp Frame}}{\mathsf{emptyFields}(\mathsf{contents} \mid \mathsf{keys} \cup \{\,\mathsf{hOP}\,\}) \mid \Pi \vdash (\mathsf{contents}, \mathsf{k}) \mapsto \varnothing * [\Sigma_F]}\ \textsc{EF - None}}{\mathsf{emptyFields}(\mathsf{contents} \mid \mathsf{keys} \cup \{\,\mathsf{hOP}\,\}) * \Sigma \mid \Pi \vdash (\mathsf{contents}, \mathsf{k}) \mapsto \varnothing * [\Sigma_F * \Sigma]}\ \textsc{Frame}$$

$$\Pi = \mathit{validKey}(\mathsf{k}) * \mathsf{k} \notin \mathsf{keys} \qquad \Sigma_F = \mathsf{emptyFields}(\mathsf{contents} \mid \mathsf{keys} \cup \{\,\mathsf{hOP}, \mathsf{k}\,\})$$

Fig. 9. Example - proof system for frame inference - derivation

the appropriate derivation, concluding that: $?F = \mathsf{emptyFields}(\mathsf{contents} \mid \mathsf{keys} \cup \{\,\mathsf{hOP}, \mathsf{k}\,\}) * \Sigma$. Intuitively, the computed frame $?F$ coincides with the spatial part of the original symbolic state P except that the property k is removed from the infinite footprint of the emptyFields assertion.

Pure Entailment. JSIL Verify discharges the pure entailments of the form $\Pi_1 \vdash \Pi_2$ to the Z3 SMT solver [De Moura and Bjørner 2008]. To this end, it encodes JSIL Logic pure assertions as Z3 formulae. Z3 gives native support for arithmetic, bit-vectors, arrays, and uninterpreted functions. It additionally supports the definition of new algebraic datatypes. We encode JSIL Logic values as a Z3 algebraic data type taking advantage of Z3 native types when possible, and specify the operations for the JSIL value types not natively supported using uninterpreted functions.

5.3 JSIL Logic Specifications of JavaScript Internal Functions

JavaScript internal functions describe the building blocks of the language, including prototype chain traversal, object management, and type conversions. They are called extensively by all JavaScript commands. Therefore, in order to reason about JavaScript code, we have to first be able to reason efficiently about the internal functions. However, their definitions in the ES5 standard are operational, complex, and intertwined, making the allowed behaviours difficult to discern. To illustrate, in Figure 10 we show the call graphs of `GetValue` and `PutValue`, the two main internal functions operating on references.

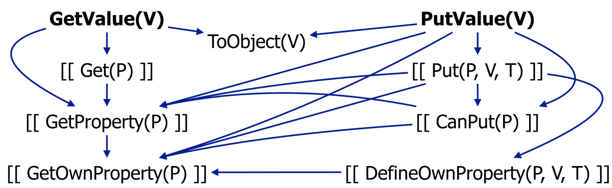

Fig. 10. Call graphs for GetValue and PutValue

Symbolic execution of internal functions. In §4.2, we showed how JS-2-JSIL compiles calls to internal functions in the standard to procedure calls to their reference implementations in JSIL. As such, in order to symbolically execute these calls, we need the specifications of internal functions.

We provide functional correctness JSIL Logic *axiomatic specifications* that explicitly expose the allowed behaviours for all cases of the internal functions that do not use higher-order reasoning, accounting for approximately 90% of all possible cases. In creating these specifications, we leverage on the built-in predicates of §3 and, in particular, on the `Pi` predicate, without which the specification of internal functions would be impossible. Using JSIL Verify, we verify that our axiomatic specifications are satisfied by their corresponding, well-tested JSIL reference implementations.

Several `GetValue` and `PutValue` specifications require the `Pi` predicate to be folded. To account for this, JS-2-JSIL automatically inserts folding and unfolding annotations before and after such calls. This is illustrated in Figure 11 for the last command of the compiled JSIL code of the assignment `contents[k] = v` in Figure 8. This way, we ensure that prototype chains are folded only when needed and, therefore, do not require the sepish connective of Gardner et al. [2012].

```
   [ @fold Pi(x_6, x_8_v, _, _, _) ]
14 x_10 := "i__putValue"(x_6, x_8_v) with elab;
   [ @unfold Pi ]
```

Fig. 11. Automatic fold/unfold annotations

Finally, observe that when we insert a new key into the map, in order for the `Pi` predicate to be automatically folded for the precondition of `PutValue` in Figure 11, JSIL Verify must prove that the supplied key does not exist in the prototype chain, which includes solving the FIP described in §5.2.

Specification by Example: `PutValue`. `PutValue(v, w)` is the JavaScript internal function that takes a reference `v` and a value `w`, and assigns `w` to the property pointed to by reference `v`. Let us consider the case in which `v` is an object reference of the form `v = [ "o", o, p ]`. In this case, `PutValue` assigns the descriptor `["d", w, T, T, T]` to the property `p` of `o`. Below, we present two specifications of `PutValue(v, w)`, where `v` is an object reference `["o", o, p]`, `o` is an extensible object that is not a string or an array object, and the property `p` is not defined in the prototype chain of `o`.

This example illustrates why we need lists of object locations and classes exposed in the `Pi` predicate. Depending on the length of the prototype chain of `o`, the post-conditions vary slightly. In both cases, the property `p` is defined in the object with the appropriate descriptor, the link from `o` to its prototype is exposed, `o` remains extensible, and the return value is `empty`. However, when `o` is not at the end of the prototype chain (right), we also have to specify (using another `Pi` predicate) the tail of the prototype chain of `o`, in which `p` is still undefined. Since we need to be able to distinguish these two cases given only the parameters of the `Pi`, we have to expose the location list.

```
{ v = ["o", o, p] *
  Pi (o, p, undefined, {o}, {c}) *
  !(c = "String") * !(c = "Array") *
  (o, "@extensible") -> true }
PutValue(v, w)
{ Pi (o, p, ["d", w, true, true, true], [o], [c]) *
  (o, "@proto") -> null * (o, "@extensible") -> true *
  ret = empty }
```

```
{ v = ["o", o, p] *
  Pi (o, p, undefined, o :: op :: lop, c :: lc) *
  !(c = "String") * !(c = "Array") *
  (o, "@extensible") -> true }
PutValue(v, w)
{ Pi (o, p, ["d", w, true, true, true], [o], [c]) *
  (o, "@proto") -> op * (o, "@extensible") -> true *
  Pi (op, p, undefined, op :: lop, lc) *
  ret = empty }
```

Similarly, the classes of objects have to be exposed as parameters of the `Pi` because certain internal functions behave differently depending on the object class. Specifically, `GetOwnProperty` behaves differently for strings, and `DefineOwnProperty` behaves differently for arrays. This is even more pronounced in ES6, with the introduction of proxies, which override all internal functions.

6 VALIDATION AND EVALUATION

We focus on the validation and evaluation of the JS-2-JSIL compiler (§6.1), the JSIL Verify tool (§6.2), our axiomatic specifications of the internal functions (§6.3), and JaVerT as a whole (§6.4).

6.1 JS-2-JSIL: A Trusted Logic-Preserving Compiler

The JS-2-JSIL compiler covers a substantial, fully representative part of ES5 Strict. It does not simplify the memory model or the semantics of JavaScript in any way. As illustrated in §4.2, there is a direct correspondence between the lines of the ES5 standard and the compiled JSIL code. Furthermore, we maintain, as much as possible, a step-by-step connection between lines of the JS-2-JSIL code itself and lines of the standard. We extensively test JS-2-JSIL against the official ECMAScript test suite, Test262, passing all 8797 applicable tests. In her PhD thesis, Naudžiūnienė [2018], also gives a formal definition and correctness result for part of the compiler, adapting techniques from compiler design literature [Barthe et al. 2005; Fournet et al. 2009] to the dynamic setting of JavaScript. A full correctness result would be feasible only in a mechanised setting: for example, by formalising JS-2-JSIL in Coq and then leveraging on JSCert, the mechanised operational semantics of ES5 in Coq of Bodin et al. [2014]. This effort, however, is beyond our manpower.

Compiler Coverage. We implement the entire kernel of ES5 Strict, except indirect eval, which exits strict mode. We implement the entire Object, Function[11], Array, Boolean, Math, and Error built-in libraries. Additionally, we implement: the core of the Global library, associated with the global object; the constructors and basic functionalities for the String, Number, and Date libraries, together with the functions from those libraries used for testing features of the kernel. We do

[11]The Function constructor, just as indirect eval, may exit strict mode; we always execute the provided code in strict mode.

not implement the orthogonal RegExp and JSON libraries. The implementation of the remaining functionalities amounts to a (lengthy) technical exercise.

Table 1. Detailed testing results

ECMAScript ES6 Test Suite	**21301**
ES6 constructs/libraries	8489
Annexes/Internationalisation	888
Parsing	565
Non-strict tests	890
ES5 Strict Tests	**10469**
Tests for non-impl. features	1297
Compiler Coverage	**9172**
ES5/6 differences in semantics	345
Tests using non-impl. features	30
Applicable Tests	**8797**
Tests passed	**8797**
Tests failed	0

Testing Methodology and Results. We test JS-2-JSIL against ECMAScript Test262, the official test suite for JavaScript implementations. Currently, Test262 has two available versions: an unmaintained version for ES5 and an actively maintained version for the ES6 standard. ES5 Test262 has poor support for ECMAScript implementations that enforce strict mode, rendering systematic efforts to target ES5 Strict tests borderline infeasible. This issue has been fully resolved in the ES6 version of the test suite.

On the other hand, there do exist certain disadvantages in using a more recent version of the test suite than the implementation was designed for; some test cases are no longer applicable and need to be excluded. Also, the specification was comprehensively redrafted and a number of new features were introduced for ES6. Luckily, the committee took great care in minimising the number of backwards incompatible changes and, as a result, only a small proportion of test cases needed to be altered between the two versions. These test cases can be identified and excluded from the results. Tests for new features are easily identifiable due to the structure of the test suite. On the whole, the strong negatives of a poorly maintained ES5 version of the test suite overshadowed the minor difficulties of having to track the incompatible changes and new features between versions of the specification. We have thus opted to test JS-2-JSIL using the latest version of ES6 Test262.

We have created a continuous-integration testing infrastructure that, on each commit to the JaVerT repository, runs Test262 automatically and logs the results. We have also developed an accompanying GUI, which allows us to easily group tests, efficiently understand the progress between test runs and pinpoint any potential regressions. To run the tests, we set up the *compiler runtime*, containing the JS initial heap and our JSIL implementations of JS internal and built-in functions. We setup the initial heap in full (~750 loc). We implement all internal functions (~1 Kloc) and a large part of the built-in libraries (~3.5 Kloc), following line-by-line the English standard.

We perform the testing as follows. First, we compile to JSIL the official harness of ES6 Test262. Then, for each test, we compile its code to JSIL. We then execute, in our JSIL interpreter, the JSIL program obtained by concatenating the compiled harness, the compiled test, and the compiler runtime. If the execution terminates normally, we declare that the test has passed.

The breakdown of the testing results is presented in Table 1. The version of the ES6 Test262 test suite that we have used[12] contains 21301 test cases. We first filter down to the 10469 tests targeting ES5 Strict, removing the cases aimed at ES6 language constructs and libraries, specification annexes, internationalisation, parsing, and ES5 non-strict features. Next, we remove the 1297 tests for unimplemented built-in library functions (for example, the JSON library), leaving us with 9172 tests targeting JS-2-JSIL. Not all of these tests, however, are applicable. ES6 has introduced minor changes to the semantics of a few features with respect to ES5, and there are 345 tests targeting such features.[13] Also, 30 tests were testing features covered by the compiler by using non-implemented features, and were thus excluded. In the end, we have the final 8797 tests relevant to JS-2-JSIL, of which we pass 100%. This gives us a solid guarantee of the correctness of our JS-2-JSIL compiler.

[12] http://github.com/tc39/test262/tree/91d06f

[13] For example, the `length` property of Function objects is configurable in ES6, but was not configurable in ES5.

This guarantee ultimately holds up to the coverage of the Test262 test suite, which is known to be extensive, but is not complete. Moreover, it is stressed by the ECMAScript committee that Test262, despite its widespread use, is not an official conformance test suite.

6.2 JSIL Verify: Scalable JSIL Verification

As discussed in §5, JSIL Verify natively supports the fundamental dynamic features of JavaScript: extensible objects, dynamic property access and dynamic procedure calls. These dynamic features introduce an additional level of complexity compared with the static features in the IRs underlying the familiar separation-logic tools. Therefore, the key aspect that the evaluation of JSIL Verify needs to address is its scalability.

We evaluate JSIL Verify by verifying that our JSIL implementations of JavaScript internal functions satisfy their axiomatic specifications. We have 186 specifications targeting 1K lines of JSIL code. These specifications are non-trivial and the underlying code makes extensive use of the dynamic features of JSIL, as the internal functions are written in a general way in the standard. We conclude that JSIL Verify is able to handle tractably the dynamic features, as it quickly verifies all 186 specifications of the JavaScript internal functions in 3.62 seconds. We have identified that a sizeable amount of that time is spent during the folding of predicates, the unification of pre-conditions for procedure calls, and, more generally, the calls to Z3, which we minimise using a number of heuristics and simplifications. We have found no reason to believe that JSIL verification with JSIL Verify would not scale to much larger code. We revisit this discussion in §6.4.

6.3 JS Internal Functions: Verified Axiomatic Specifications

Using JSIL Verify, we verify that our axiomatic specifications of the internal functions are satisfied by the corresponding JSIL reference implementations. These implementations follow the ES5 standard line-by-line and are (indirectly) substantially tested via our testing of the JS-2-JSIL compiler against Test262. These results can be interpreted in two ways: they provide validation of the JSIL axiomatic specifications, as the implementations closely follow the standard and are well tested; and, at the same time, they provide further validation of the implementations of the internal functions.

Our axiomatic specifications of the internal functions directly increase the scalability of JaVerT, as they allow it to step over the underlying implementations rather than executing them every time. We envisage that these specifications will be useful beyond JaVerT. For example, starting from our axiomatic specifications, we could create executable specifications of the internal functions, that could then be used for different types of symbolic analysis for JavaScript. They would also provide a mechanism for restricting the semantics of JavaScript in a principled way. If, for instance, we would like to perform an analysis that wishes to abstract a semantic feature of JavaScript, say type coercion, we would generate executable specifications of the internal functions without taking into account the axiomatic specifications that describe type coercion. This would be much more robust than altering the code of the internal functions manually.

6.4 JaVerT: Verifying JavaScript Programs

We have verified a number of further examples in addition to the Map and Id Generator examples shown in §3, including: a priority queue library, modelled after a real-world `Node.js` priority queue library [Jones 2016]; operations on binary search trees (BSTs), which target set reasoning in Z3; and an insertion sort algorithm, which targets list reasoning in Z3. We have also verified several Test262 programs, testing complex language statements such as the `switch` and `try-catch-finally`. The statistics for these examples are shown in Figure 2. The columns of the table denote: the name of the example; the number of lines of JS code; the number of lines of compiled JSIL code; the number of verified specifications; and the obtained verification time.

Table 2. JaVerT Verification Statistics

Example	#JS	#JSIL	#specs	t(s)
Key-value map	23	523	9	3.37
ID Generator	16	330	4	0.73
Priority queue	46	1003	10	7.14
BST	70	1032	5	7.38
Insertion sort	24	415	2	1.78
Test262 examples	113	1367	16	3.46

Understanding the scalability of JaVerT amounts to understanding how the size of the compiled JSIL code corresponds to the size of the original JavaScript code and the scalability of JSIL Verify in the presence of the reasoning patterns of JavaScript. As Figure 2 shows, the compiled JSIL code has approximately ten to twenty-five times more lines of code than its JavaScript counterpart. Also, it takes about 0.5 seconds to verify one hundred lines of compiled JSIL code. With JaVerT requiring annotations in the form of pre- and postconditions, loop invariants and folding/unfolding of user-defined predicates, we estimate that users will only be able to annotate eventually up to thousands of lines of JavaScript code, not tens of thousands. For us, the results presented in Table 2 indicate that JaVerT can meet this scalability goal. Importantly, we note that, although the specification of data structure libraries requires a potentially large annotational bootstrap, in terms of defining all of the abstractions capturing the data structures, the ratio of annotations to code decreases rapidly as the library code and verified client code grow.

When it comes to verification, there is little work to compare JaVerT against. In fact, there is only KJS, the instantiation of the general $\mathbb{K}$ verification framework to JavaScript. We compare the performance of JaVerT and KJS on the BST and insertion sort examples, which we have in common. On a machine with an Intel Core i7-4960X CPU 3.60GHz and DDR3 RAM 64GB, KJS takes 35.7 seconds to verify the correctness of the BST operations, and 44.8 seconds to verify the insertion sort algorithm. On a machine with an Intel Core i7-4980HQ CPU 2.80 GHz and DDR3 RAM 16GB, JaVerT verifies the same BST operations in 7.38 seconds, and the insertion sort algorithm in 1.78 seconds. This difference in speed is not surprising, because KJS implements proof search with automatic unfolding and folding of recursive predicates, which requires fewer code annotations than JaVerT, but is computationally intensive. The remaining KJS examples amount to using predicates describing more complex data structures, such as AVL trees and red-black trees. We do not envisage major issues with verifying them using JaVerT, as they do not exercise any JavaScript-specific features and only depend on designing the abstractions correctly. Such abstractions are standard in separation logic. On the other hand, we were unable to verify our examples that illustrate dynamic property access using the KJS tool because, at the time, KJS did not have support for predicates whose footprint captures some, but not all properties of an object: for example, the `Pi` predicate.

7 CONCLUSIONS AND FUTURE WORK

We believe JaVerT constitutes an important step towards the verification of real-world JavaScript programs. It is built on top of a trusted, systematically validated infrastructure and it successfully tackles a number of challenges that are critical for tractable reasoning about JavaScript. We contain the complexity of reasoning about complex JavaScript statements by compiling JavaScript to JSIL (V1). We reason efficiently about the fundamental dynamic features of JavaScript using JSIL Verify (V2), the first verification tool based on separation logic to natively support such features. We provide verified axiomatic specifications of the internal functions (V3).

We design key abstractions that allow the developer to capture fundamental JavaScript concepts: the `Scope` predicate to reason about basic variable scoping; the `Pi` predicate to capture the prototype inheritance of JavaScript; and the `Closure` predicate to talk about shared variables in JavaScript function closures. The `Pi` and `Closure` predicates are carefully designed to resolve the tension between the overlapping of prototype and scope chains, and the heap separation inherent to separation logic. Our specifications can be used by a developer who has minimal knowledge of JavaScript internals. To demonstrate this, we specify: a key-value map implementation, written in a typical OO-style, where our specifications ensure *prototype safety* for library operations; and a simple ID generator, where we show how our specifications can be used to capture the degree of encapsulation obtained from using function closures.

Our immediate next steps are to prove properties of programs using the `for-in` statement, leveraging on the work of Cox et al. [2014], and to extend JSIL Logic with higher-order reasoning by encoding it in Iris [Jung et al. 2015], in order to be able to reason about JavaScript getters/setters and arbitrary functions passed as parameters.

JaVerT was designed so that the trust in its infrastructure is maximised. To validate JSIL Verify further rigorously, we will encode JSIL Logic in Coq, leveraging on the Iris framework; adapt JSIL Verify to produce, for each verified specification, a Coq proof term supposedly certifying it; and use Coq to verify formally that this proof term indeed certifies it.

In terms of coverage, we expect to move JS-2-JSIL to ES6 Strict at some point, extending it with the new language constructs of ES6. The existing specifications of the internal functions will remain the same and our abstractions will still be directly relevant. We may later move to full ES6, where we would have to model scope lookup using an inductive predicate for capturing the footprint of a dynamic scope chain traversal, similar to the one used by Gardner et al. [2012].

There are several ways to improve the overall usability of JaVerT, the most important of which is giving meaningful feedback to the developer when specifications cannot be verified. This is a non-trivial problem that requires a precise lifting of error messages from JSIL back to JavaScript, which is possible, given our correctness results. Also, we have observed that JaVerT specifications for prototype safety and function closures follow specific patterns, parts of which could be inferred automatically. This gives room to the possibility of providing specification templates for the developer. Finally, JaVerT currently supports only verified client code. An interesting goal would be to automatically synthesise defensive wrappers for verified library code, so that verified libraries can be safely integrated with non-verified client code.

We will develop an automated tool based on bi-abduction [Calcagno et al. 2011] for verifying large JavaScript codebases, but believe that the semi-automatic JaVerT will always have a role to play in the development of functional correctness specifications of critical libraries. We may investigate how to reason about the DOM using JaVerT, building on the work of Raad et al. [2016]. We are also looking for ways to reuse the infrastructure behind JaVerT for other styles of JavaScript analysis. Concretely, we are building a JSIL front-end to Rosette [Torlak and Bodík 2013, 2014], where we aim to use the symbolic execution of Rosette to obtain a bug-finding tool for JavaScript. We expect that such a tool could also help the developer with the debugging of JaVerT specifications. Our goal is to establish our JSIL infrastructure as a common platform for JavaScript verification.

ACKNOWLEDGMENTS

We would like to thank Azalea Raad for the many lengthy, insightful conversations about the formalism underpinning JaVerT. We would also like to thank Beatrix de Wilde, César Roux dit Buisson, Iván Matellanes, Thomas Pointon, and Aubrianna Zhu, whose work during their MEng, MSc, and UROP projects improved the infrastructure of JaVerT. Finally, we would like to thank the anonymous reviewers for their comments, which have improved the overall quality of the paper.

Fragoso Santos, Gardner, and Maksimović were supported by the EPSRC Grant 'Certified Verification of Client-Side Web Programs' (EP/K032089/1), the EPSRC Programme Grant 'REMS: Rigorous Engineering for Mainstream Systems' (EP/K008528/1), and the Department of Computing at Imperial College London. Naudžiūnienė and Wood were supported by an EPSRC DTA award. Maksimović was partially supported by the Serbian Ministry of Education and Science through the Mathematical Institute of the Serbian Academy of Sciences and Arts, projects ON174026 and III44006.

REFERENCES

Christopher Anderson, Paola Giannini, and Sophia Drossopoulou. 2005. Towards Type Inference for JavaScript. In *Proceedings of the 19th European Conference on Object-Oriented Programming, ECOOP 2005, Glasgow, UK, July 25-29, 2005. (LNCS)*, Andrew P. Black (Ed.), Vol. 3586. Springer, 428–452. https://doi.org/10.1007/11531142_19

Esben Andreasen and Anders Møller. 2014. Determinacy in static analysis for jQuery, See [Black and Millstein 2014], 17–31. https://doi.org/10.1145/2660193.2660214

Gilles Barthe, Tamara Rezk, and Ando Saabas. 2005. Proof Obligations Preserving Compilation. In *Revised Selected Papers of the 3rd International Workshop on Formal Aspects in Security and Trust, FAST 2005, Newcastle upon Tyne, UK, July 18-19, 2005 (LNCS)*, Theodosis Dimitrakos, Fabio Martinelli, Peter Y. A. Ryan, and Steve A. Schneider (Eds.), Vol. 3866. Springer, 112–126. https://doi.org/10.1007/11679219_9

Josh Berdine, Cristiano Calcagno, and Peter W. O'Hearn. 2005a. Smallfoot: Modular Automatic Assertion Checking with Separation Logic. In *Revised Lectures of the 4th International Symposium on Formal Methods for Components and Objects, FMCO 2005, Amsterdam, The Netherlands, November 1-4, 2005 (LNCS)*, Frank S. de Boer, Marcello M. Bonsangue, Susanne Graf, and Willem P. de Roever (Eds.), Vol. 4111. Springer, 115–137. https://doi.org/10.1007/11804192_6

Josh Berdine, Cristiano Calcagno, and Peter W. O'Hearn. 2005b. Symbolic Execution with Separation Logic. In *Proceedings of the 3rd Asian Symposium on Programming Languages and Systems, APLAS 2005, Tsukuba, Japan, November 2-5, 2005 (LNCS)*, Kwangkeun Yi (Ed.), Vol. 3780. Springer, 52–68. https://doi.org/10.1007/11575467_5

Gavin M. Bierman, Martín Abadi, and Mads Torgersen. 2014. Understanding TypeScript. In *Proceedings of the 28th European Conference on Object-Oriented Programming, ECOOP 2014, Uppsala, Sweden, July 28 - August 1, 2014 (LNCS)*, Richard E. Jones (Ed.), Vol. 8586. Springer, 257–281. https://doi.org/10.1007/978-3-662-44202-9_11

Lars Birkedal, Rasmus Ejlers Møgelberg, Jan Schwinghammer, and Kristian Støvring. 2012. First Steps in Synthetic Guarded Domain Theory: Step-indexing in the Topos of Trees. *Logical Methods in Computer Science* 8, 4 (2012). https://doi.org/10.2168/LMCS-8(4:1)2012

Lars Birkedal, Nick Rothwell, Mads Tofte, and David N. Turner. 1993. *The ML Kit, Version 1.* Technical Report. Technical Report 93/14 DIKU.

Andrew P. Black and Todd D. Millstein (Eds.). 2014. *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages and Applications, OOPSLA 2014, part of SPLASH 2014, Portland, OR, USA, October 20-24, 2014.* ACM. http://dl.acm.org/citation.cfm?id=2660193

Martin Bodin, Arthur Charguéraud, Daniele Filaretti, Philippa Gardner, Sergio Maffeis, Daiva Naudžiūnienė, Alan Schmitt, and Gareth Smith. 2014. A Trusted Mechanised JavaScript Specification. In *Proceedings of the 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2014, San Diego, CA, USA, January 20-21, 2014, Proceedings*, Suresh Jagannathan and Peter Sewell (Eds.). ACM, 87–100. https://doi.org/10.1145/2535838.2535876

Cristian Cadar, Daniel Dunbar, and Dawson R. Engler. 2008. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. In *the 8th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2008, December 8-10, 2008, San Diego, California, USA, Proceedings*, Richard Draves and Robbert van Renesse (Eds.). USENIX Association, 209–224. http://www.usenix.org/events/osdi08/tech/full_papers/cadar/cadar.pdf

Cristiano Calcagno, Dino Distefano, Jérémy Dubreil, Dominik Gabi, Pieter Hooimeijer, Martino Luca, Peter W. O'Hearn, Irene Papakonstantinou, Jim Purbrick, and Dulma Rodriguez. 2015. Moving Fast with Software Verification. In *the 7th International NASA Symposium on Formal Methods, NFM 2015, Pasadena, CA, USA, April 27-29, 2015, Proceedings*, Klaus Havelund, Gerard J. Holzmann, and Rajeev Joshi (Eds.). LNCS, Vol. 9058. Springer, 3–11. https://doi.org/10.1007/978-3-319-17524-9_1

Cristiano Calcagno, Dino Distefano, Peter W. O'Hearn, and Hongseok Yang. 2011. Compositional Shape Analysis by Means of Bi-Abduction. *J. ACM* 58, 6, 26:1–26:66. https://doi.org/10.1145/2049697.2049700

Arlen Cox, Bor-Yuh Evan Chang, and Xavier Rival. 2014. Automatic Analysis of Open Objects in Dynamic Language Programs. In *Proceedings of the 21st International Symposium on Static Analysis, SAS 2014, Munich, Germany, September 11-13, 2014 (LNCS)*, Markus Müller-Olm and Helmut Seidl (Eds.), Vol. 8723. Springer, 134–150. https://doi.org/10.1007/978-3-319-10936-7_9

Andrei Ştefănescu, Daejun Park, Shijiao Yuwen, Yilong Li, and Grigore Roşu. 2016. Semantics-Based Program Verifiers for All Languages. In *Proceedings of the 31th Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2016)*. ACM, 74–91. https://doi.org/10.1145/2983990.2984027

Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. 1989. An Efficient Method of Computing Static Single Assignment Form. In *Conference Record of the Sixteenth Annual ACM Symposium on Principles of Programming Languages, Austin, Texas, USA, January 11-13, 1989.* ACM Press, 25–35. https://doi.org/10.1145/75277.75280

Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2008/ETAPS 2008)*. Springer-Verlag, Berlin, Heidelberg, 337–340. http://dl.acm.org/citation.cfm?id=1792734.1792766

Dino Distefano and Matthew J. Parkinson. 2008. jStar: Towards Practical Verification for Java. In *Proceedings of the 23rd Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2008, October 19-23, 2008, Nashville, TN, USA*, Gail E. Harris (Ed.). ACM, 213–226. https://doi.org/10.1145/1449764.1449782

ECMAScript Committee. 2011. *The 5th Edition of the ECMAScript Language Specification*. Technical Report. ECMA. http://www.ecma-international.org/ecma-262/5.1/ECMA-262.pdf.

Facebook. 2017. react.js: A JavaScript Library for Building User Interfaces. https://facebook.github.io/react/.

Asger Feldthaus and Anders Møller. 2014. Checking Correctness of TypeScript Interfaces for JavaScript Libraries, See [Black and Millstein 2014], 1–16. https://doi.org/10.1145/2660193.2660215

Asger Feldthaus, Max Schäfer, Manu Sridharan, Julian Dolby, and Frank Tip. 2013. Efficient Construction of Approximate Call Graphs for JavaScript IDE Services. In *Proceedings of the 35th International Conference on Software Engineering, ICSE 2013, San Francisco, CA, USA, May 18-26, 2013*, David Notkin, Betty H. C. Cheng, and Klaus Pohl (Eds.). IEEE Computer Society, 752–761. https://doi.org/10.1109/ICSE.2013.6606621

David Flanagan. 1998. *JavaScript: The Definitive Guide* (3rd ed.). O'Reilly & Associates, Inc., Sebastopol, CA, USA.

Cédric Fournet, Gurvan Le Guernic, and Tamara Rezk. 2009. A Security-preserving Compiler for Distributed Programs: from Information-flow Policies to Cryptographic Mechanisms. In *Proceedings of the 2009 ACM Conference on Computer and Communications Security, CCS 2009, Chicago, Illinois, USA, November 9-13, 2009*, Ehab Al-Shaer, Somesh Jha, and Angelos D. Keromytis (Eds.). ACM, 432–441. https://doi.org/10.1145/1653662.1653715

Cedric Fournet, Nikhil Swamy, Juan Chen, Pierre-Evariste Dagand, Pierre-Yves Strub, and Benjamin Livshits. 2013. Fully Abstract Compilation to JavaScript. In *Proceedings of the 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2013)*. ACM, New York, NY, USA, 371–384. https://doi.org/10.1145/2429069.2429114

José Fragoso Santos, Philippa Gardner, Petar Maksimović, and Daiva Naudžiūnienė. 2017. Towards Logic-Based Verification of JavaScript Programs. In *the 26th International Conference on Automated Deduction, CADE 26, Gothenburg, Sweden, August 6-11, 2017, Proceedings (LNCS)*, Leonardo de Moura (Ed.), Vol. 10395. Springer, 8–25. https://doi.org/10.1007/978-3-319-63046-5_2

Philippa Gardner, Sergio Maffeis, and Gareth David Smith. 2012. Towards a Program Logic for JavaScript. In *Proceedings of the 39th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2012, Philadelphia, Pennsylvania, USA, January 22-28, 2012*, John Field and Michael Hicks (Eds.). ACM, 31–44. https://doi.org/10.1145/2103656.2103663

Google. 2017. The V8 JavaScript Engine. https://v8project.blogspot.ie/.

Arjun Guha, Claudiu Saftoiu, and Shriram Krishnamurthi. 2010. The Essence of Javascript. In *Proceedings of the 24th European Conference on Object-oriented Programming (ECOOP'10)*. Springer-Verlag, Berlin, Heidelberg, 126–150. http://dl.acm.org/citation.cfm?id=1883978.1883988

Bart Jacobs, Jan Smans, Pieter Philippaerts, Frédéric Vogels, Willem Penninckx, and Frank Piessens. 2011. VeriFast: A Powerful, Sound, Predictable, Fast Verifier for C and Java. In *the 3rd International NASA Symposium on Formal Methods, NFM 2011, Pasadena, CA, USA, April 18-20, 2011, Proceedings*, Mihaela Gheorghiu Bobaru, Klaus Havelund, Gerard J. Holzmann, and Rajeev Joshi (Eds.). LNCS, Vol. 6617. Springer, 41–55. https://doi.org/10.1007/978-3-642-20398-5_4

Dongseok Jang and Kwang-Moo Choe. 2009. Points-to Analysis for JavaScript. In *Proceedings of the 2009 ACM Symposium on Applied Computing (SAC 2009)*. ACM, New York, NY, USA, 1930–1937. https://doi.org/10.1145/1529282.1529711

Simon Holm Jensen, Anders Møller, and Peter Thiemann. 2009. Type Analysis for JavaScript. In *Proceedings of the 16th International Symposium on Static Analysis, SAS 2009, Los Angeles, CA, USA, August 9-11, 2009. (LNCS)*, Jens Palsberg and Zhendong Su (Eds.), Vol. 5673. Springer, 238–255. https://doi.org/10.1007/978-3-642-03237-0_17

Jason Jones. 2016. Priority Queue Data Structure. https://github.com/jasonsjones/queue-pri.

Ralf Jung, David Swasey, Filip Sieczkowski, Kasper Svendsen, Aaron Turon, Lars Birkedal, and Derek Dreyer. 2015. Iris: Monoids and Invariants As an Orthogonal Basis for Concurrent Reasoning. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, (POPL 2015)*. ACM, New York, NY, USA, 637–650. https://doi.org/10.1145/2676726.2676980

Vineeth Kashyap, Kyle Dewey, Ethan A. Kuefner, John Wagner, Kevin Gibbons, John Sarracino, Ben Wiedermann, and Ben Hardekopf. 2014. JSAI: a static analysis platform for JavaScript. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, (FSE-22), Hong Kong, China, November 16 - 22, 2014*, Shing-Chi Cheung, Alessandro Orso, and Margaret-Anne D. Storey (Eds.). ACM, 121–132. https://doi.org/10.1145/2635868.2635904

Robbert Krebbers, Amin Timany, and Lars Birkedal. 2017. Interactive Proofs in Higher-order Concurrent Separation Logic. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, (POPL 2017)*. ACM, New York, NY, USA, 205–217. https://doi.org/10.1145/3009837.3009855

Daniel Kroening and Michael Tautschnig. 2014. CBMC - C Bounded Model Checker - (Competition Contribution). In *Proceedings of the 20th International Conference Tools and Algorithms for the Construction and Analysis of Systems, TACAS 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014. (LNCS)*, Erika Ábrahám and Klaus Havelund (Eds.), Vol. 8413. Springer, 389–391. https://doi.org/10.1007/978-3-642-54862-8_26

Hongki Lee, Sooncheol Won, Joonho Jin, Junhee Cho, and Sukyoung Ryu. 2012. SAFE: Formal Specification and Implementation of a Scalable Analysis Framework for ECMAScript. In *Proceedings of the 19th International Workshop on Foundations of Object-Oriented Languages (FOOL 2012)*.

Ben Livshits. 2014. JSIR, An Intermediate Representation for JavaScript Analysis. http://too4words.github.io/jsir/.

Microsoft. 2014. *TypeScript Language Specification.* Technical Report. Microsoft.

Daiva Naudžiūnienė. 2018. *An Infrastructure for Tractable Verification of JavaScript Programs.* Ph.D. Dissertation. Imperial College London, London, UK. Advisor(s) Philippa Gardner.

Changhee Park and Sukyoung Ryu. 2015. Scalable and Precise Static Analysis of JavaScript Applications via Loop-Sensitivity. In *Proceedings of the 29th European Conference on Object-Oriented Programming, ECOOP 2015, July 5-10, 2015, Prague, Czech Republic (LIPIcs)*, John Tang Boyland (Ed.), Vol. 37. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 735–756. https://doi.org/10.4230/LIPIcs.ECOOP.2015.735

Daejun Park, Andrei Stefănescu, and Grigore Roşu. 2015. KJS: A Complete Formal Semantics of JavaScript. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2015).* ACM, New York, NY, USA, 346–356. https://doi.org/10.1145/2737924.2737991

Matthew J. Parkinson and Gavin M. Bierman. 2005. Separation Logic and Abstraction. In *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2005, Long Beach, California, USA, January 12-14, 2005*, Jens Palsberg and Martín Abadi (Eds.). ACM, 247–258. https://doi.org/10.1145/1040305.1040326

Matthew J. Parkinson and Gavin M. Bierman. 2008. Separation logic, Abstraction and Inheritance. In *Proceedings of the 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2008, San Francisco, California, USA, January 7-12, 2008*, George C. Necula and Philip Wadler (Eds.). ACM, 75–86. https://doi.org/10.1145/1328438.1328451

Joe Gibbs Politz, Matthew J. Carroll, Benjamin S. Lerner, Justin Pombrio, and Shriram Krishnamurthi. 2012. A Tested Semantics for Getters, Setters, and Eval in JavaScript. In *Proceedings of the 8th Symposium on Dynamic Languages, DLS 2012, Tucson, AZ, USA, October 22, 2012*, Alessandro Warth (Ed.). ACM, 1–16. https://doi.org/10.1145/2384577.2384579

Azalea Raad, José Fragoso Santos, and Philippa Gardner. 2016. DOM: Specification and Client Reasoning. In *Proceedings of the 14th Asian Symposium on Programming Languages and Systems, APLAS 2016, Hanoi, Vietnam, November 21-23, 2016, (LNCS)*, Atsushi Igarashi (Ed.), Vol. 10017. 401–422. https://doi.org/10.1007/978-3-319-47958-3_21

Aseem Rastogi, Nikhil Swamy, Cédric Fournet, Gavin M. Bierman, and Panagiotis Vekris. 2015. Safe & Efficient Gradual Typing for TypeScript. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015, Mumbai, India, January 15-17, 2015*, Sriram K. Rajamani and David Walker (Eds.). ACM, 167–180. https://doi.org/10.1145/2676726.2676971

John C. Reynolds. 2002. Separation Logic: A Logic for Shared Mutable Data Structures. In *the 17th IEEE Symposium on Logic in Computer Science (LICS 2002), 22-25 July 2002, Copenhagen, Denmark, Proceedings.* IEEE Computer Society, 55–74. https://doi.org/10.1109/LICS.2002.1029817

Grigore Roşu and Traian Florin Şerbănuţă. 2010. An Overview of the K Semantic Framework. *Journal of Logic and Algebraic Programming* 79, 6 (2010), 397–434. https://doi.org/10.1016/j.jlap.2010.03.012

Manu Sridharan, Julian Dolby, Satish Chandra, Max Schäfer, and Frank Tip. 2012. Correlation Tracking for Points-To Analysis of JavaScript. In *the 26th European Conference on Object-Oriented Programming, ECOOP 2012, Beijing, China, June 11-16, 2012, Proceedings (LNCS)*, James Noble (Ed.), Vol. 7313. Springer, 435–458. https://doi.org/10.1007/978-3-642-31057-7_20

Nikhil Swamy, Joel Weinberger, Cole Schlesinger, Juan Chen, and Benjamin Livshits. 2013. Verifying Higher-order Programs with the Dijkstra Monad. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2013, Seattle, WA, USA, June 16-19, 2013*, Hans-Juergen Boehm and Cormac Flanagan (Eds.). ACM, 387–398. https://doi.org/10.1145/2491956.2491978

Peter Thiemann. 2005. Towards a Type System for Analyzing JavaScript Programs. In *Programming Languages and Systems, 14th European Symposium on Programming,ESOP 2005, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2005, Edinburgh, UK, April 4-8, 2005, Proceedings (LNCS)*, Shmuel Sagiv (Ed.), Vol. 3444. Springer, 408–422. https://doi.org/10.1007/978-3-540-31987-0_28

Emina Torlak and Rastislav Bodík. 2013. Growing Solver-aided Languages with Rosette. In *ACM Symposium on New Ideas in Programming and Reflections on Software, Onward! 2013, part of SPLASH 2013, Indianapolis, IN, USA, October 26-31, 2013*, Antony Hosking, Patrick Eugster, and Robert Hirschfeld (Eds.). ACM, 135–152. https://doi.org/10.1145/2509578.2509586

Emina Torlak and Rastislav Bodík. 2014. A Lightweight Symbolic Virtual Machine for Solver-aided Host Languages. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2014, Edinburgh, United Kingdom - June 09 - 11, 2014*, Michael F. P. O'Boyle and Keshav Pingali (Eds.). ACM, 530–541. https://doi.org/10.1145/2594291.2594340

Hongseok Yang, Oukseh Lee, Josh Berdine, C. Calcagno, Byron Cook, Dino Distefano, and Peter O'Hearn. 2008. Scalable Shape Analysis for Systems Code. In *CAV 2008: Proceedings of the 20th international conference on Computer Aided Verification.* Springer-Verlag, Berlin, Heidelberg, 385–398. https://doi.org/10.1007/978-3-540-70545-1_36