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.
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 ECMA Script 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 ECMA Script standard are operational, intricate, and intertwined.

There has been little theoretical and practical work on logic-based reasoning about JavaScript. Gardner et al.\(^4\) have developed a separation logic for a tiny fragment of ECMA Script 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.\(^5\) 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.\(^6\) in this paper; we expect to build on the work of Cox et al.\(^7\) in future. On the more practical side, Swamy et al.\(^8\) have used the higher-order logic of F*\(^9\) to prove absence of runtime errors for higher-order ES3 programs using the Dijkstra monad, but have stopped short of proving functional correctness properties. Ștefanescu et al.\(^10\) 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. www3techs.com/technologies/details/cp-javascript/all
2. http://github.info
4. JaVerT is pronounced zhah-verh (IPA: ʒə vər̩̂h), like the name of the main antagonist of Victor Hugo’s ‘Les Miserables’. 

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, and a logic-preserving compiler from JavaScript to JSIL, called JS-2-JSIL. JS-2-JSIL is designed to be line-by-line close to the ECMAScript standard, without simplifying the behaviour in any way. 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

---

**Fig. 1. JaVerT: JavaScript Verification Toolchain**
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. 2005;]
We provide assertions and carefully designed abstractions in JS Logic, together with a translation of JS-2-JSIL is broader; we pass all 8797 tests applicable for our coverage (cf. § 6.1). 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.

Roșu and Șerbănucă [2010] have developed $\lambda$, 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 $\lambda$-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).

Ștefănescu et al. [2016] introduce a language-independent verification infrastructure that can be instantiated with a $\lambda$-interpretation of a language to automatically generate a symbolic verification tool for that language based on the $\lambda$ 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, 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 Ţefănescu 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.


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 λJS [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 compiled 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.

---

The $K$ 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 $K$ is underway and will support this.
We address the JavaScript speciﬁcation challenges highlighted in the introduction. To specify
we would have to relate JS Logic and JSIL Logic via S5. This would be a diﬃcult task.
The basic memory model of JavaScript is straightforward. The diﬃculty lies in the way in
which it

Therefore, we would have to redo S5 tests using our methodology and
thus, a strong connection between the generated JSIL code and the standard was imperative.
JavaScript internals. We must, therefore, build a number of predicates on top of JS Logic to describe
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
properties, function objects, string objects and the JavaScript initial heap. In §3.3.1, we introduce our basic predicates for describing object
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.

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

\[
\begin{align*}
\text{JS locations : } l & \in \mathcal{L} & \text{JS variables : } x & \in X_{JS} & \text{JS heap values : } \omega & \in \mathcal{V}_{JS} \text{ } := \text{ } v | \overline{v} | \text{fid} \\
\text{JS values : } v & \in \mathcal{V}_{JS} \text{ } := n | b | m | \text{undefined} | \text{null} | l & \text{JS heaps : } h & \in \mathcal{H}_{JS} \text{ } := \mathcal{L} \times X_{JS} \rightarrow \mathcal{V}_{JS}^h
\end{align*}
\]
A JavaScript heap, $h \in \mathcal{H}_{JS}$, 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 $L$. Property names and JS program variables are taken from a set of strings $X_{JS}$. JS values contain: numbers, $n$; booleans, $b$; strings, $m$; the special JavaScript values undefined and null; and object locations, $l$. JS heap values, $\omega \in \mathcal{V}_{JS}^l$, contain: JS values, $v \in \mathcal{V}_{JS}$; lists of JS values, $\overline{v}$; and function identifiers, $fid \in \mathcal{F}id$. 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 $h_1 \cup h_2$, a heap lookup by $h(l, x)$, and the empty heap by $\emptyset$.

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

\textbf{JS Logic Assertions}

\[ V \in \mathcal{V}_{JS}^l := \omega \mid \omega_{\text{set}} \mid \emptyset \quad E \in \mathcal{E}_{JS}^l := V \mid x \mid x \oplus E \mid E \oplus E \mid \text{sc} \mid \text{this} \]

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

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

JS logical values, $V \in \mathcal{V}_{JS}^l$, contain: JS heap values, $\omega$; sets of JavaScript heap values, $\omega_{\text{set}}$; and the special value $\emptyset$, read none, used to denote the absence of a property in an object (see §3.4). JS logical expressions, $E \in \mathcal{E}_{JS}^l$, contain: logical values, $V$; JS program variables, $x$; JS logical variables, $x$; unary and binary operators, $\oplus$ 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 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 types$(X_i : \tau_i |_{i=1}^n)$ states that variable $X_i$ has type $\tau_i$ for $0 \leq i \leq n$, where $X_i$ is either a program or a logical variable and $\tau$ ranges over JavaScript types, $\tau \in \text{Types}$. We say that an assertion is spatial if it contains cell assertions or emptyFields assertions. Otherwise, it is pure.

JaVerT specifications have the form $\{P\} fid(\overline{x}) \{Q\}$, 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 $fl \in \{\text{nm}, \text{er}\}$, 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 $\{P\} fid(\overline{x}) \{Q\}$ 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.

\textbf{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.

```
JSObject(o, p, c, e) := types(o : Obj, p : Str, c : Str, e : Bool) *
    (o,@proto") -> p * (o,"@class") -> c * (o,"@extensible") -> e
JSObject(o) := 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 §3.3). 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 SCCell(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) * ... }

This specification states, 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
494 Pi predicate, with four base cases and two recursive cases.

\[
P_i (o, p, d, lo, lc) :=
\]

\[
T * lo = [o] * lc = [c] * (o, "@class") -> c * ![c = "String"] * (o, p) -> d * ![d = None],
\]

\[
T * lo = [o] * lc = [c] * (o, "@class") -> c * ![c = "String"] * SCell(o, p, d) * ![d = None],
\]

\[
(o, @proto) -> null * (o, p) -> None * d = undefined,
\]

\[
T * lo = [o] * lc = [c] * (o, "@class") -> c * ![c = "String"] *
\]

\[
(o, @proto) -> null * SCell(o, p, None) * d = undefined,
\]

\[
T * lo = o :: lop * lc = c :: lcp * (o, "@class") -> c * ![c = "String"] * (o, p) -> None *
\]

\[
lop = op :: lop * (o, "@proto") -> op * Pi(op, p, d, lop, lcp),
\]

\[
T * lo = o :: lop * lc = c :: lcp * (o, "@class") -> c * ![c = "String"] * SCell(o, p, None) *
\]

\[
lop = op :: lop * (o, "@proto") -> op * Pi(op, p, d, lop, lcp)
\]

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,
494 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 be present in the heap if the library code
494 is to run as intended. We highlight a general methodology for specifying such libraries, introducing
494 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
494 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
494 as an object with property _contents, denoting the object used to store the map contents. The
494 named properties of _contents and their value attributes correspond to the map keys and values,
494 respectively. As the functions get, put, and validKey are to be shared between all map objects, they
494 are defined as properties of Map.prototype, which is the prototype of the objects that are created
494 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
494 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:
1 var m = new Map();
2 m.get("foo")

CLIENT 2:
1 var mp = Map.prototype;
2 var desc = { value: 0, writable: false }; 3 Object.defineProperty(mp, "_contents", desc)

CLIENT 3:
1 var m = new Map();
2 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. 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), 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) := JSONObject(m, mp) *
  DataProp(m, "._contents", c) * JSONObject(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.

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

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)`. 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)`. 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 `getId` and `reset`, as well as the `ER_{IG1}` 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_y, ER_{IG1}, ER_{getId}]`. We can also observe that, for example, function objects `getId` and `reset` share the `[l_y, ER_{getId}]` 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*, \( \psi : \text{Str} \times \text{Str} \rightarrow \mathbb{N} \), for determining which ER in the scope chain of a given function defines a given variable. For instance, \( \psi("getId","makeIdGen") = 0 \) and \( \psi("getId","count") = 1 \), 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*, \( \psi^o : \text{Str} \times \text{Str} \rightarrow \mathbb{N} \), which takes two function identifiers and returns the length of the overlap of their scope chains. For instance, \( \psi^o("getId","reset") = 2 \), 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 (nth(sch, n), x) \mapsto v, where nth is the binary list indexing operator and n = \psi(fid, x). For instance, the predicate \text{Scope(count} \cdot c, gi_sc, getId) unfolds to \text{Scope(count} \cdot c, \text{count} = 1). We can also use \text{Scope(x: v)} as syntactic sugar for \text{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.

\[
\text{Scope(x: v, sch, fid)} := (\text{nth(sch, n), x}) \mapsto v, \text{ when } n = \psi(fid, x) \neq 0;
\]

\[
\text{Scope(x: v, } \_\_\text{, fid)} := (1g, x) \mapsto "d", v, \_\_\_\_\_, \text{ when } \psi(fid, x) = 0.
\]

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 \text{Scope(prefix} \cdot p) \cdot \text{Scope(count} \cdot 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 \text{Scope(prefix} \cdot p) \cdot \text{Scope(count} \cdot 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.

\[
\text{IdGenerator(ig, p, c) := types(p: Str, c: Num) *} \text{JSObject(o, Object.prototype) *} \\
\text{DataProp(ig, "getId", gif) *} \text{FunctionObject(gif, "getId", gi_sc) *} \\
\text{DataProp(ig, "reset", rf) *} \text{FunctionObject(rf, "reset", r_sc) *}
\]

\[
\text{Scope(count} \cdot c, gi_sc, getId) \times \text{Scope(prefix} \cdot p, gi_sc, getId) \times \text{OChains(getId: gi_sc, reset: r_sc)}
\]

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 \text{Scope(count} \cdot c, gi_sc, getId) \times \text{Scope(count} \cdot c, r_sc, reset), but this is duplicated resource. We need a predicate that captures the scope chain overlap between two functions.

The \text{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 \exists_{0 \leq i < n} \text{nth} (f_sc, i) = \text{nth} (g_sc, i), where n = \psi^0(f, g). In particular, \text{OChains(getId: gi_sc, reset: r_sc)} unfolds to \text{nth} (gi_sc, 0) = \text{nth} (r_sc, 0) *
nth \((g_i, c, 1) = nth (r, sc, 1), as \psi(\text{getId, reset}) = 2. That is, the \(g_i, sc\) and \(r, sc\) coincide on their first two ERs, namely the global object and the ER of \(mainGen\).

\[chains(f : f, sc, g : g, sc) := \Theta_{g \leq c}(nth(f, sc, i) = nth(g, sc, i)), \text{ where } n = \psi(f, g)\]

The \(chains\) 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 \(chain\), as shown in the fourth line of \(IdGen\).

When function closures get more involved, it can be tedious to write all necessary \(chains\) predicates. We offer a more compact predicate, \(Closure\), expressible in terms of \(Scope\) and \(chain\).

The \(Closure\) predicate states that the variables \(x_1, \ldots, x_n\) with values \(v_1, \ldots, v_n\) are all shared between functions \(f_1, \ldots, f_m\), whose scope chains are given by \(f_1, sc, \ldots, f_m, sc\), and that these scope chains all maximally overlap pairwise. Using \(Closure\), we can rewrite the last line of \(IdGen\) as \(Closure\) \((\text{count: } c, \text{prefix: } p; \text{getId: } g, sc, \text{reset: } r, sc)\).

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 \(ig1, ig2, \text{ and } id1\) 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".

\[
\begin{align*}
\text{Scope}(\text{makeIdGen}: mIG) \ast \text{FunctionObject}(mIG, "makeIdGen", mIG, sc) \ast \\
\quad \text{Scope}(ig1: undefined) \ast \text{Scope}(ig2: undefined) \ast \text{Scope}(id1: undefined)
\end{align*}
\]

\[
\begin{align*}
\text{var } ig1 = \text{makeIdGen}("foo"), ig2 = \text{makeIdGen}("bar"), id1 = ig1, \text{getId}();
\end{align*}
\]

\[
\begin{align*}
\text{Scope}(ig1: mIG) \ast \text{FunctionObject}(mIG, "makeIdGen", mIG, sc) \ast \\
\quad \text{Scope}(ig1: IDGI) \ast \text{Scope}(ig2: IDG2) \ast \text{Scope}(id1: id) \\
\quad \text{IdGen}(id1, "foo", 1) \ast \text{IdGen}(id2, "bar", 0) \ast id = "foo_id_0"
\end{align*}
\]

**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 \(IdGen\) predicate to maintain information about the scope chain in which the ID generator was created:

\[
\text{IdGen}(ig, p, c, ig) := \text{types}(p: Str, c: Num) \ast \text{JSONObject}(ig, Object.prototype) \ast \\
\text{DataProp}(ig, "getId", gif) \ast \text{FunctionObject}(gif, getId, gi, sc) \ast \\
\text{DataProp}(ig, "reset", rf) \ast \text{FunctionObject}(rf, reset, r, sc) \ast \\
\text{Closure}(\text{count: } c, \text{prefix: } p; \text{getId: } g, sc, \text{reset: } r, sc, \text{makeIdGen: } ig, sc).
\]

With this definition in place, the postcondition of \(makeIdGen\) (Figure 7, left) can be restated as \(IdGen\) \((ig, \text{prefix}, 0, sc)\), where, as mentioned earlier, \(sc\) denotes the scope chain in which this \(IdGen\) is to be executed. We can now state the specification of \(get\) in terms of the \(IdGen\) 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 \(IdGen\).

\[
\begin{align*}
\text{types}(\text{prefix: } Str) \ast \\
\text{makeIdGen}(\text{prefix}) \ast \\
\text{IdGen}(\text{ig, prefix, sc}) \ast \\
\quad (\text{this is } ig) \ast \text{chains}(\text{getId: } sc, \text{makeIdGen: } ig, sc) \ast \\
\quad \text{IdGen}(ig, \text{prefix, c, ig, sc}) \ast \\
\quad \text{getId}()
\end{align*}
\]

**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 fgenerator 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: $n \in \mathbb{N}$  
Booleans: $b \in \mathbb{B}$  
Strings: $m \in \mathbb{S}$  
Locations: $l \in \mathcal{L}$  
Variables: $x \in \mathcal{X}_{\text{JSIL}}$

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

Expressions: $e \in \mathcal{E}_{\text{JSIL}} := \lambda \mid x \mid \emptyset \mid e \oplus e$

Basic Commands: $bc \in \mathcal{B}_{\text{Cmd}} := \text{skip} \mid x := e \mid x := \text{new}() \mid x := [e, e] \mid [e, e] := e \mid \text{delete}(e, e) \mid x := \text{hasField}(e, e) \mid x := \text{getFields}(e)$

Commands: $c \in \mathcal{C}_{\text{Cmd}} := bc \mid \text{goto } i \mid \text{goto } [e] i \mid \text{goto } e(\bar{e}) \mid x := \phi(\bar{x})$

Procedures: $\text{proc } \in \mathcal{P}_{\text{Proc}} := \text{proc } \text{fid}(\bar{x})\{\bar{c}\}$

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

JSIL literals, $\lambda \in \mathcal{L}_{\text{It}}$, include JavaScript literals, as well as procedure identifiers $\text{fid}$, types $\tau$, the special value empty, and lists and sets of literals. JSIL expressions, $e \in \mathcal{E}_{\text{JSIL}}$, include JSIL literals, JSIL program variables $x$, 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 $\text{skip}$, 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 $\phi$-node commands. The two goto commands are standard: $\text{goto } i$ jumps to the $i$-th command of the active procedure, and $\text{goto } [e] i$, $j$ jumps to the $i$-th command if $e$ evaluates to $\text{true}$, and to the $j$-th otherwise. The dynamic procedure call $x := e(\bar{e})$ with $j$ first obtains the procedure name and arguments by evaluating $e$ and $\bar{e}$, respectively, then executes the appropriate procedure with these arguments, and finally assigns its return value to $x$. Control is transferred to the next command if the procedure does not raise an error, or to the $j$-th command otherwise. Finally, the $\phi$-node command $x := \phi(x_1, \ldots, x_n)$ 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 $x$ is $x_j$ if and only if the $i$-th path was taken. We include $\phi$-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 $p \in \mathcal{P}$ is a set of top-level procedures $\text{proc } fid(x)\{\overline{c}\}$, where $fid$ is the name of the procedure, $x$ its sequence of formal parameters, and its body $\overline{c}$ is a command list consisting of a numbered sequence of JSIL commands. We use $p_{fid}$ and $p_{fid}(i)$ 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, $\text{ret}$ and $\text{err}$. When a procedure jumps to $i_{nm}$, it returns normally with the return value $\text{ret}$; when it jumps to $i_{er}$, it returns an error, with the error value $\text{err}$.

**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, $\rho \in \mathcal{Sto}$, is a mapping from JSIL variables to JSIL values, and a JSIL heap, $h \in \mathcal{H}_{\text{JSIL}}$, is a mapping from pairs of locations and property names (strings) to JSIL values, $v \in \mathcal{V}_{\text{JSIL}}$, which coincide with the JSIL literals. The JSIL semantic judgement has the form $p \rightarrow \langle h, \rho, j, i \rangle \Downarrow_{fid} (h', \rho', o)$, 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 $\rho$, generates the heap $h'$, the store $\rho'$, and returns the outcome $o$. JSIL outcomes are of the form $\text{fl}(v)$, where $\text{fl} \in \{\text{nm}, \text{err}\}$ 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 $\text{contents}[k] = v$ 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 $\text{contents}[k] = v$, which is given in Figure 8.

1. We first evaluate the the property accessor $\text{contents}[k]$ 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 $\text{contents}$.

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

\[
V \in \mathcal{V}_{JSIL}^L := v \mid \not\exists \quad E \in \mathcal{E}_{JSIL}^L := V \mid x \mid x \odot E \mid E \oplus E
\]

\( r \in \text{Types} := \text{Num} \mid \text{Bool} \mid \text{Str} \mid \text{Undef} \mid \text{Null} \mid \text{Obj} \mid \text{List} \mid \text{Set} \mid \text{Type} \)

\( P, Q \in \mathcal{A}\mathcal{S}_{JSIL} := \text{true} \mid \text{false} \mid E = E \mid E \leq E \mid P \land Q \mid \neg P \mid P \lor Q \mid \exists x.P \mid\)

\( \text{emp} \mid (E,E) \mapsto E \mid \text{emptyFields}(E \mid E) \mid \text{types}(X_1 : \tau_1 \mid \cdot \mid) \)

JSIL Logic Assertions. There is a strong correspondence between JavaScript and JSIL at the level of the logics. JSIL logical values, \( V \in \mathcal{V}_{JSIL}^L \), consist of JSIL values extended with \( \not\exists \), subsuming JS logical values. JSIL logical expressions, \( E \in \mathcal{E}_{JSIL}^L \), 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, \( P, Q \in \mathcal{A}\mathcal{S}_{JSIL} \), 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 \( x\sc \) and \( x\text{this} \) 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 satisifiability 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 \( \not\exists \). The satisifiability relation for JSIL Logic assertions has the form: \( H, \rho, \epsilon \models P \), where: (1) \( H \) is an abstract heap; (2) \( \rho \) is a JSIL variable store; (3) and \( \epsilon \) is a JSIL logical environment, mapping JSIL logical variables to JSIL values. The satisifiability relation for JSIL Logic assertions builds on the semantics of JSIL logical expressions. A JSIL logical expression \( E \) is interpreted with respect to \( \rho \) and \( \epsilon \), written \( \llbracket E \rrbracket_{\rho, \epsilon}^\epsilon \). Both the satisifiability relation and the expression interpretation are mostly standard; we show the non-standard cases below. We also use a function \( \text{.TypeOf} \), which given a JSIL value, outputs its type.

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

Semantics of Logical Expressions:

\[
\llbracket V \rrbracket_{\rho}^\epsilon \triangleq V \quad \llbracket x \rrbracket_{\rho}^\epsilon \triangleq \rho(x) \quad \llbracket x \rrbracket_{\rho}^\epsilon \triangleq \epsilon(x)
\]

Satisifiability Relation:

\[
H, \rho, \epsilon \models \text{emptyFields}(E_1 \mid E_2) \quad \Leftrightarrow \quad H = \cup_m \llbracket E_1 \rrbracket_{\rho,m} \mid (\llbracket E_2 \rrbracket_{\rho,m} \mapsto \not\exists)
\]

\[
H, \rho, \epsilon \models \text{types}(X_1 : \tau_1 \mid \cdot \mid) \quad \Leftrightarrow \quad H = \text{emp} \quad \text{and \ for \ all } i \in \{1, \ldots, n\}, \text{TypeOf}(\llbracket E_i \rrbracket_{\rho}^\epsilon) = \tau_i
\]

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

\[
H, \rho, L, l_i, \epsilon \models P \quad \Leftrightarrow \quad H, \rho[x\sc \mapsto L, x\text{this} \mapsto l_i], \epsilon \models \mathcal{T}(P)
\]

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, \( \rho \vdash (h, \rho, j, l) \downarrow_{fd} (h', \rho', o) \), given in §4.1. Also, it makes use of a deabstraction function \( \mathcal{L} \) : \( \mathcal{H}_{JSIL} \to \mathcal{H}_{JSIL} \).
\[ \mathcal{H}_{\text{JSIL}}, \text{transforming abstract JSIL heaps to concrete JSIL heaps. Intuitively, } [H] \text{ denotes the concrete JSIL heap obtained by removing the cells of } H \text{ that are mapped to } \emptyset. \]

**Definition 4.1** (Validity of JSIL Logic Specifications). A JSIL Logic specification \{P\} \{Q\} for return mode \(fl\) is valid with respect to a program \(p\), written \(p, fl \vdash \{P\} \{Q\}\), if and only if, for all logical contexts \((H, \rho, \epsilon)\), heaps \(h_f\), stores \(\rho_f\), flags \(fl'\), and JSIL values \(\nu\), it holds that:

\[
\begin{align*}
H, \rho, \epsilon &\models P \land p \vdash \{[H], \rho, -, 0\} \Downarrow_{\text{fid}} \langle h_f, \rho_f, fl'(\nu) \rangle \\
fl' &= fl \land \exists H_f. H_f, \rho_f, \epsilon \models Q \land [H_f] = h_f
\end{align*}
\]

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

\[ \mathcal{T} (\{P\} \{Q\}) = \{ \mathcal{T}(P) \} \{ \mathcal{T}(Q) \} \]

Also, we say that a JS-2-JSIL compiler \(C\) is correct if compiled programs preserve the behaviour of their original versions. Put formally:

\[
s, L, l_f \vdash \langle h, \rho \rangle \Downarrow_{\text{fid}} \langle h_f, fl(\nu) \rangle \iff \exists \rho_f . C(s) \vdash \langle h, \rho|_{\text{sc} \leftarrow L, \text{this} \leftarrow l_f}, -, 0 \rangle \Downarrow_{\text{fid}} \langle h_f, \rho_f, fl(\nu) \rangle
\]

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\} \{Q\}, it holds that:

\[
s, fl \vdash \{P\} \{Q\} \iff C(s), fl \models \mathcal{T}(\{P\} \{Q\})
\]

### 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\}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 \(E_1 \equiv E_2\) to denote \(E_1 = E_2 \land \text{emp}\). The Get Fields axiom states that if the object bound to \(e\) only contains the properties denoted by \(X_1, \ldots, X_n\), then, after execution of \(x := \text{getFields}(e)\), \(x\) will be bound to a list containing precisely \(X_1, \ldots, X_n\) in an order described by the \text{ord}\ predicate, which stands for an implementation-dependent ordering of property names. The Property Deletion axiom forbids the deletion of @proto properties.
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): $[P]bc\{Q\}$

<table>
<thead>
<tr>
<th>Property Access</th>
<th>Get Fields</th>
<th>Property Deletion</th>
<th>Object Creation</th>
</tr>
</thead>
<tbody>
<tr>
<td>$P \equiv (e_1,e_2) \mapsto X * X \neq \emptyset$</td>
<td>$P \equiv ((e,X_i) \mapsto Y_i)</td>
<td>(x \neq \emptyset) } * (Y_i \neq \emptyset)$</td>
<td>$P \equiv (e_1,e_2) \mapsto X * X \neq \emptyset * \emptyset$</td>
</tr>
<tr>
<td>$[P] x := [e_1,e_2] [P \times x \neq X]$</td>
<td>$(P \times x := \text{getFields}(e) (P \times (x \in [X_1, \ldots, X_n]) \times (\text{ord}(x) \neq \text{true})$</td>
<td>$(P \times \text{delete}(e_1, e_2) {e_1, e_2} \mapsto \emptyset)$</td>
<td>$(\text{emp}) x := \text{new} () {Q}$</td>
</tr>
</tbody>
</table>

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, SE : $\text{fid} \rightarrow \text{Flag} \rightarrow \text{Spec}$, 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, $\text{SE}(\text{fid}, \text{fl}) = \text{Spec}$ means that spec is the specification of the procedure with identifier $\text{fid}$ 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 $p, \text{fid}, \text{SE}, \text{fl} + \langle P, k, i \rangle \sim \langle Q, j \rangle$, meaning that: (1) we are currently symbolically executing the code of the procedure with identifier $\text{fid}$ in the JSIL program $p$ assuming the specification environment $\text{SE}$; (2) the symbolic execution of the entire procedure must terminate with return mode $\text{fl}$; 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 $p, \text{fid}, \text{SE}, \text{fl}$ do not change during symbolic execution, we leave them implicit. In the operational rules, we write $\text{post}(\text{spec})$ to denote the postcondition of $\text{spec}$.

Operational Rules for JSIL Logic Symbolic Execution: $p, \text{fid}, \text{SE}, \text{fl} + \langle P, k, i \rangle \sim \langle Q, j \rangle$
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 preconditions 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, pd ∈ D : F ⊓ F × F ⊓ N × N ⊓ ϕ(ASJMIL × 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) ∈ pd(fd, 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 → fid j to denote that i is an immediate predecessor of j, and i → fid 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 p ∈ P and a specification environment SE ∈ Str → F × F, for a program p ∈ P and a specification environment SE ∈ Str → F × F, we say that a proof candidate pd ∈ D is well-formed with respect to p and SE, written p, SE ⊢ pd, if and only if for all procedures fid in p, and index i the following statements hold:

1. ∀ fl, P, Q. SE(fid, fl) = {P} fid(red)(Q) ⇐⇒ pd(fid, fl, 0) = {(P, 0)}
2. ∀ fl, P, k. (P, k) ∈ pd(fid, fl, i) ∧ (P ′ = false) ⇒
   (i → fid j) ⇒ ∃ Q. (Q, i) ∈ pd(fid, fl, j) ∧ p, fid, SE, fl ⊢ (P, k, i) ⊢ (Q, j)
   ∨ (i ∈ {imm, iter} ⇒ p, fid, SE, fl ⊢ (P, k, i) ⊢ (P, i))

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 pd ∈ D such that p, SE ⊢ pd, then:

∀ fid, fl, P, Q, red. SE(fid, fl) = {P} fid(red)(Q) ⇒ p, fl ⊢ {P} fid(red)(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 [e₁, e₂] := e₃. To symbolically execute this command in a symbolic state P, JSIL Verify must solve the following instance of the frame inference problem (FIP) P ⊢ (o, p) ⊢ ⊥ ?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 \(P \vdash e_1 = E_1\) \(|_{i=1,2}\); and (2) traversing all the emptyFields
assertions emptyFields\((E_1 | E_2) \mapsto -\) in \(P\), checking for each one whether \(P \vdash e_2 = E_1\) and \(P \vdash e_2 \neq E_2\)
(for the case in which the required resource is captured by the emptyFields assertion).

Similarly to Berdine et al. [2005b], given the FIP \(P \vdash Q \mapsto \Sigma\), what we do first is decompose \(P\)
and \(Q\) into pairs of the form \((\Sigma, \Pi)\), where \(\Sigma\) and \(\Pi\) denote, respectively, their spatial and pure parts.

Hence, what we are left with is \((\Sigma_p, \Pi_p) \mapsto (\Sigma_q, \Pi_q) \mapsto \Sigma\), which can then be further decomposed
into: (i) \((\Sigma_p, \Pi_p) \mapsto (\Sigma_q, \text{True}) \mapsto \Sigma\) 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 \(\Sigma_p \mid \Pi_p \vdash \Sigma_q \mapsto \Sigma\). 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 - \(\Sigma_1 \mid \Pi \vdash \Sigma_2 \mapsto \Sigma\)

<table>
<thead>
<tr>
<th>Rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cell-Cell</td>
</tr>
<tr>
<td>(\Pi \vdash E_1 = E_1' \mid_{i=1,2,3} \Sigma_1 \mid \Pi \vdash \Sigma_2 \mapsto \Sigma)</td>
</tr>
<tr>
<td>Frame</td>
</tr>
<tr>
<td>(\Sigma_1 \mapsto (E_1, E_2) \mapsto E_3 \mid \Pi \vdash \Sigma_2 \mapsto (E_1', E_2') \mapsto E_3' \mapsto \Sigma)</td>
</tr>
<tr>
<td>Emp</td>
</tr>
<tr>
<td>(\Sigma_1 \mid \Pi \vdash \Sigma_2 \mapsto \Sigma)</td>
</tr>
<tr>
<td>Emp</td>
</tr>
<tr>
<td>(\Pi \vdash E_1 \vdash E_1' \mid E_2 \mid \Sigma_1 \mid \Pi \vdash \Sigma_2 \mapsto \Sigma)</td>
</tr>
<tr>
<td>EmptyFields-None-Cell</td>
</tr>
<tr>
<td>(\Pi \vdash E_1 = E_1' \mid E_2 \mid \Pi \vdash E_1' \neq E_2 \mid \Sigma_1 \mid \Pi \vdash \Sigma_2 \mapsto \Sigma)</td>
</tr>
<tr>
<td>EmptyFields-EmptyFields-Extra-Resource-Left</td>
</tr>
<tr>
<td>(\Pi \vdash E_0 = E_0' \mid E \mid E \cup {E_i \mid k &lt; l} = E' \mid \Sigma_1 \mid \Pi \vdash \emptyFields(E_0, E_i) \mapsto \emptyFields(E_0', E_i') \mapsto \emptyFields(E_0, E) \mid \Pi \vdash \Sigma_2 \mapsto \Sigma)</td>
</tr>
<tr>
<td>EmptyFields-EmptyFields-Extra-Resource-Right</td>
</tr>
<tr>
<td>(\Pi \vdash E_0 = E_0' \mid E \setminus {E_i \mid k &lt; l} = E' \mid \Sigma_1 \mid \Pi \vdash \emptyFields(E_0, E_i) \mapsto \emptyFields(E_0', E_i') \mapsto \emptyFields(E_0, E) \mid \Pi \vdash \Sigma_2 \mapsto \Sigma)</td>
</tr>
</tbody>
</table>

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: \(\emptyFields(E_1 | E_2) \mapsto \Sigma_1 \mapsto \emptyFields(E_1' | E_2') \mapsto \emptyFields(E_1 | E_2') \mapsto \emptyFields(E_1, E_2) \mapsto \emptyFields(E_1' | E_2') \mapsto \emptyFields(E_1', E_2') \mapsto \emptyFields(E_1' | E_2') \mapsto \emptyFields(E_1, E_2) \mapsto \emptyFields(E_1' | E_2') \mapsto \emptyFields(E_1, E_2)
\)
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-Extra-Resource-Left
and 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, \(\emptyFields(E_0, E_i) \mapsto \emptyFields(E_0', E_i') \mapsto \emptyFields(E_0, E)
\)
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 \(E \cup \{E_i \mid k < l\}\) 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 \(\text{put}(k, v)\) 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 \(\text{contents}[k] = v\), we must prove that \(k\) is not
defined in \(\text{contents}\), which implies solving the following FIP: \(P \vdash (\text{contents} | k) \mapsto \emptyFields(E_1 | E_2') \mapsto \emptyFields(E_1', E_2') \mapsto \emptyFields(E_1, E_2) \mapsto \emptyFields(E_1', E_2') \mapsto \emptyFields(E_1, E_2)
\)
with \(P \mapsto \emptyFields(\text{contents} \mid \text{keys} \cup \{\text{hOP}\}) \mapsto \Sigma \mid \Pi \mapsto \emptyFields(\text{contents} \mid \text{keys} \cup \{\text{hOP}\}) \mapsto \emptyFields(E_1 | E_2') \mapsto \emptyFields(E_1', E_2') \mapsto \emptyFields(E_1, E_2) \mapsto \emptyFields(E_1', E_2') \mapsto \emptyFields(E_1, E_2)
\)
and \(\Sigma \mapsto \emptyFields(\text{contents} \mid \text{keys} \cup \{\text{hOP}\}) \mapsto \emptyFields(E_1 | E_2') \mapsto \emptyFields(E_1', E_2') \mapsto \emptyFields(E_1, E_2) \mapsto \emptyFields(E_1', E_2') \mapsto \emptyFields(E_1, E_2)
\)
and \(\Sigma \mapsto \emptyFields(\text{contents} \mid \text{keys} \cup \{\text{hOP}\}) \mapsto \emptyFields(E_1 | E_2') \mapsto \emptyFields(E_1', E_2') \mapsto \emptyFields(E_1, E_2) \mapsto \emptyFields(E_1', E_2') \mapsto \emptyFields(E_1, E_2)
\).
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].

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.

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\(^\text{11}\), 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.

**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\(^\text{12}\) 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.\(^\text{13}\) 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.

\(^\text{12}\)http://github.com/tc39/test262/tree/91d06f

\(^\text{13}\)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.
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 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

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


