An Infrastructure for Tractable Verification of JavaScript Programs

Abstract

The dynamic nature of JavaScript, together with its complex semantics, makes it a difficult target for symbolic verification techniques. To address this issue, we develop an infrastructure for tractable symbolic verification of JavaScript programs (ECMAScript 5 Strict mode), and present JaVerT, a semi-automatic JavaScript Verification Toolchain built on top of this infrastructure. The infrastructure consists of: **(1)** JS-2-JSIL, a compiler from JavaScript to JSIL, a simple intermediate goto language suitable for verification; **(2)** JSIL Verify, a semi-automatic JSIL verification tool based on separation logic; and **(3)** verified JSIL specifications of JavaScript internal functions. We design JS-2-JSIL to be step-by-step faithful to the ECMAScript standard and systematically test it against the official ECMAScript test suite, passing 100% of the appropriate tests. We provide JSIL reference implementations of the JavaScript internal functions and use JSIL Verify to show that these implementations satisfy their specifications. We demonstrate the feasibility of our verification infrastructure using JaVerT to specify and verify simple JavaScript programs, illustrating our ideas using an implementation of a priority queue. We believe that our infrastructure can be reused for other styles of program analysis for JavaScript.

1 Introduction

JavaScript is the de facto language for programming client-side web applications. It is described by the international ECMAScript standard [1], with which all major web browsers now comply. The highly dynamic nature of JavaScript, coupled with its intricate semantics, makes the understanding and development of correct JavaScript code notoriously difficult. Because of this, JavaScript developers still have very little tool support for catching errors early in development, contrasted with the abundance of tools (such as IDEs and specialised static analysis tools) available for more traditional languages such as C and Java. The transfer of analysis techniques to the domain of JavaScript is known to be a challenging task.

Our goal is to develop analysis tools for reasoning about JavaScript programs based on separation logic and symbolic verification. Symbolic verification has recently become tractable for C and Java, with compositional techniques that scale and properly engineered tools applied to real-world code: e.g. Facebook's Infer for C and Java [15], based on separation logic; Java Pathfinder, a model checking tool for Java bytecode programs [44]; CBMC, a bounded model checker for C, currently being adapted to Java at Amazon [32]; and WALA's analysis for Java using the Rosette symbolic analyser [22]. We are aware of little work on logic-based symbolic analysis for JavaScript, and no published work on verification tools based on such analysis. To address this, we develop an infrastructure for tractable symbolic verification of JavaScript programs (ECMAScript 5 Strict mode), and present JaVerT, a semi-automatic JavaScript Verification Toolchain built on top of this infrastructure.

Our verification infrastructure consists of: **(1)** JS-2-JSIL, a compiler from JavaScript to JSIL, our simple intermediate goto language suitable for verification; **(2)** JSIL Verify, a semi-automatic JSIL verification tool based on our JSIL separation logic; and **(3)** JSIL specifications of JavaScript internal functions, verified using JSIL Verify. An important part of our project has been the validation of this infrastructure. JS-2-JSIL is step-by-step faithful to the ECMAScript standard and is systematically tested against the official ECMAScript test suite, passing 100% of the appropriate tests. It also satisfies a correctness criterion expressed via a simple correspondence between JavaScript and JSIL heaps. JSIL Verify is based on our JSIL separation logic, proven sound with respect to our JSIL operational semantics. The JSIL reference implementations of JavaScript internal functions are step-by-step faithful to the standard and are verified with respect to their JSIL specifications using JSIL Verify.

Leibniz International Proceedings in Informatics

Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

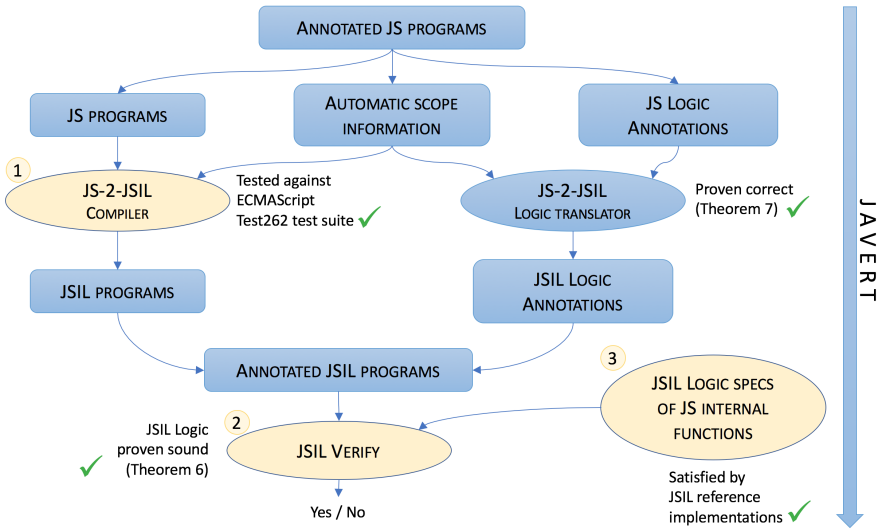


Figure 1 JaVerT: JavaScript Verification Toolchain using the JSIL Infrastructure

We demonstrate the feasibility of our infrastructure by developing the JaVaScript Verification Toolchain, JaVerT, on top of it. We use JaVerT to verify functionally correct specifications of simple JavaScript programs (Figure 1), illustrating our reasoning using a JavaScript implementation of a priority queue. We show how to develop natural JavaScript abstractions that make reasoning using JaVerT nearly as simple as reasoning about Java programs using a semi-automatic verification tool such as VeriFast [27]. Any additional complexity stems from the behaviour of JavaScript programs, and not from our reasoning.

JSIL. Many tools based on symbolic analysis [7, 8, 19, 27, 14, 15, 32, 22] target intermediate goto representations in order to fully dismantle the control flow constructs of their target language, thus simplifying their analyses. These representations, however, are not suitable for dynamic languages such as JavaScript, which require extensible objects, dynamic fields and dynamic function calls. For this reason, we have developed an intermediate goto language, JSIL, which we believe to be well-suited for symbolic verification of JavaScript programs. JSIL has only a small number of commands and a simple operational semantics with no corner cases or unexpected behaviours. We purposefully design the JSIL memory model to be as close as possible to the memory model of JavaScript so that we can easily relate functional properties of JavaScript programs with their translated JSIL programs.

JS-2-JSIL Compiler. The JS-2-JSIL compiler from JavaScript to JSIL targets the strict mode of the ECMAScript 5 English standard (ES5 Strict). It closely follows our operational semantics for ES5 Strict, which was inspired by the operational semantics of JSCert [12], the recent Coq specification of the ES5 standard. As in JSCert, we follow the ECMAScript standard step-by-step by using the pretty-big-step style of semantics [16]. This means that the structure of a compiled JSIL program directly reflects the description of the behaviour of the original JavaScript program in the English standard.

Why ES5 Strict? We aim to demonstrate the feasibility of symbolic verification for JavaScript. In such an academic project, however, it is difficult to understand how much one should initially aim for. We focus on ES5 Strict, a restricted variant of ES5 that intentionally has slightly different semantics compared with the full language, and exhibits better behavioural properties, such as being syntactically scoped. ES5 Strict is developed by the ECMAScript committee, is recommended for use by the committee and professional developers [13], and is widely used by major industrial players: e.g. Google's V8 engine [18] and Facebook's React library [14]. We believe that ES5 Strict is the correct starting point for symbolic verification for JavaScript.

Coverage. We cover a very large and fully representative fragment of ES5 Strict. In doing so, the memory model is not simplified in any way. We implement the entire kernel of ES5 Strict plus all of the built-in libraries intertwined with the kernel. The detailed breakdown of our coverage is shown in §4.3.

Validation by testing. We systematically test JS-2-JSIL against the new ECMAScript ES6 Test262 test suite, which organises tests by feature. This enables us to provide a more fine-grained analysis than was previously possible. To that end, we develop a continuous-integration testing infrastructure, which runs the entire test suite automatically on each commit and provides comprehensive feedback. We also design an accompanying GUI, which greatly simplifies our interaction with the database and allows us to filter tests easily. Using this infrastructure, we are able to identify, through several stages of filtering, precisely the tests appropriate for our coverage. We identify 10469 tests relevant for ES5 Strict and 8797 tests relevant for the JS-2-JSIL coverage, of which we pass 100%. A detailed account of the testing methodology and the breakdown of the results is given in §4.4. We highlight that this infrastructure is highly modular and can easily be reused, together with our methodology, for a systematic testing of related projects such as JSCert or S5 [36].

Validation by compiler correctness. We designed JS-2-JSIL so that that there is a simple correspondence between JavaScript and JSIL heaps, and a step-by-step connection to the standard. This allows us to define a straightforward correctness condition for JS-2-JSIL. In a separate technical appendix [42], we give a correctness proof using our formal ES5 Strict operational semantics, which describes a fragment of the language, adapting standard techniques from compiler design literature [6, 23] to the dynamic setting of JavaScript. The full result would require a substantial mechanised proof development.

JSIL Verify. We introduce JSIL Verify, our semi-automatic verification tool for JSIL. JSIL Verify is based on JSIL Logic, our sound separation logic for JSIL. It contains a symbolic execution engine and an entailment engine, which uses the Z3 SMT solver [18] to discharge assertions in first-order logic with equality and arithmetic, while we handle the separation logic assertions ourselves. To verify JSIL programs, we provide pre- and postconditions for functions, loop invariants and fold/unfold directives for user-defined predicates. Developers wishing to verify JavaScript programs will not need to use JSIL Verify directly. We, however, use it to verify JSIL specifications of JavaScript internal function.

JSIL Specifications of Internal Functions. JavaScript internal functions describe the fundamental inner workings of the language, such as prototype chain traversal (`GetProperty`), or property definition (`DefineOwnProperty`) and deletion (`DeleteProperty`). They are not accessible by the programmer, but are called internally by all JavaScript commands. Their definitions in the standard are complex, are given operationally, and are often intertwined, making it difficult for the user to fully grasp the control flow and allowed behaviours.

We provide functionally correct *axiomatic specifications* of the internal functions. In creating these specifications, we leverage on a number of JavaScript-specific abstractions built on top of JSIL Logic, which make the specifications much more readable than the operational definitions of the standard. The remaining complexity arises from the internal functions themselves, not our reasoning. We give JSIL reference implementations of the internal functions, substantially tested by the testing of JS-2-JSIL. Using JSIL Verify, we prove that these implementations satisfy their axiomatic specifications. These proofs can be seen both as further validation of the implementations of the internal functions as well as validation of the JSIL axiomatic specifications themselves, as the implementations closely follow the standard and are well tested.

We believe that our JSIL axiomatic specifications of the internal functions are an important contribution of the paper. They directly benefit JaVerT, since the verification of JavaScript

code only needs to use the specifications, not the underlying implementations. We also believe that they will prove helpful to the developer, since they are more concise than the descriptions in the standard. Finally, we hope that they will be useful for other forms of verification: for example, by the CBMC model checker [32].

Having stated the benefits, we also need to state one limitation of our JSIL specifications. Currently, JSIL logic does not support higher-order reasoning, so we cannot specify properties associated with getters and setters, or functions passed as function parameters. At this stage, this limitation is not an obstacle, as the JavaScript code we are targeting involves simple manipulation of familiar data structures such as a priority queue. However, as we progress and approach real-world code, we will need to adapt our work to provide a higher-order logic and verification tool for JSIL, inspired by the work on higher-order separation logics [38, 10, 40, 27] and the verification tool VeriFast [27] which is based on one such logic.

JaVerT. We introduce the JavaScript Verification Toolchain, JaVerT, described in Figure 1. The idea is to start from a JavaScript program, such as a library implementation of a priority queue, annotated by the programmer with assertions written in JS Logic, an assertion language in the style of separation logic. These annotations allow the programmer to write pre- and postconditions for functions and the entire program, as well as loop invariants and unfold/fold instructions for user-defined predicates, such as queue predicate. From the annotated JavaScript program, we automatically extract useful scope information. Given this information, the JavaScript program is compiled to a JSIL program using JS-2-JSIL, whereas JS Logic annotations are translated using our JS-2-JSIL logic translator to equivalent annotations in JSIL logic. The resulting annotated JSIL program is then automatically verified by JSIL Verify, making use of the verified JSIL specifications of JavaScript internal functions.

We validate the JS-2-JSIL logic translator by establishing a full correctness result for the assertion languages, and a partial correctness result for the program logics, which depends on the correctness of the JS-2-JSIL compiler.

2 Related Work

This paper pulls together a large amount of work on operational semantics, compilers, and separation logic. Much of this was previously developed for static languages. The application of this work to the dynamic and complex language that is JavaScript has not been straightforward. There is a wide range of literature covering different flavours of program analysis for JavaScript including: type systems [43, 2, 29, 33, 20, 9, 37], control flow analysis [34, 21], pointer analysis [28, 39] and abstract interpretation [31, 29, 3], among others. We do not address this work in detail as our aim is to develop a logic-based verification tool for JavaScript. Hence, we focus our discussion on compilers and intermediate representations for JavaScript as well as separation-logic-based verification tools.

Compilers and Intermediate Representations for JavaScript. There is a rich landscape of intermediate representations (IRs) for JavaScript. We can broadly divide these IRs into two categories: **(1)** those that work for analyses that are syntax-directed, following the abstract syntax tree (AST) of the program, such as λ_{JS} [25], S5 [36], and notJS [31]; and **(2)** those that aim at analyses based on the control-flow graph of the program, such as JSIR [30], WALA [39] and the IR of TAJS [29, 3]. The IRs in **(1)** are normally well-suited for high-level analysis, such as type-checking/inference [25, 36], whereas those in **(2)** are generally the target of separation-logic-based tools [7, 8, 19, 27, 14, 15] as well as tools for tractable symbolic evaluation [32, 13].

Our work develops a separation-logic-based tool for JavaScript, using an IR in the style of **(2)**. We have considered using an IR in **(1)** as an interim stage during compilation. λ_{JS} and notJS were not appropriate as their target is ES3, an older version of JavaScript. The

best candidate was S5 developed by Politz et al. [36], which targets the full ES5 standard. The compilation from ES5 to S5 is informally described in the paper, and is validated through testing against the ECMAScript test suite, achieving 70% success on all ES5 tests and 98% on tests designed specifically to test unique features of ES5 Strict. However, the figure critical for us, which is the success rate of S5 on full ES5 Strict tests (those testing its unique features *and* the features common with ES5), was not reported. Therefore, given our emphasis on covering all of the corner cases of the ES5 Strict semantics, using S5 was also not appropriate.

When it comes to the IRs in **(2)**, JSIL is similar to JSIR [30], and the IRs of WALA [22] and TAJS [29, 3]. Neither JSIR nor WALA have associated compilers or reference implementations of the JavaScript internal functions and built-in libraries. The absence of a compiler makes it impossible for us to discuss the precise nature of the differences between JSIL and JSIR/WALA. Our choices were determined by wanting JS-2-JSIL to follow closely our ES5 Strict operational semantics. TAJS does include a compiler, originally from ES3 but now extended with partial models of the ES5 standard library, the HTML DOM, and the browser API. However, as the focus of the TAJS papers is not placed on the IR, but rather on the developed analyses, the authors do not provide detailed information about the translation and their testing infrastructure.

One of our main goals in the development of JS-2-JSIL was to cover all the corner cases of the ES5 Strict semantics. Thus, a strong connection between the generated JSIL code and the standard was imperative. Our design of the JS-2-JSIL compiler builds on the tradition of compilers which closely follow the operational semantics of the source language, such as the ML Kit Compiler [11]. In the same spirit, JS-2-JSIL mimics ES5 Strict by inlining in the generated JSIL code the internal steps performed by the ES5 Strict semantics, making them explicit. For that, we based our compiler on the JSCert mechanised specification of ES5 [12]. Alternatively, we could have used KJS [35], a mechanised specification of JavaScript in the $\mathbb{K}$ framework.

Separation-Logic-Based Verification Tools. Separation logic provides modular reasoning about programs which manipulate complex heap structures. It has been successfully used in verification tools for static languages: Smallfoot [7] for a simple imperative while language; jStar [19] for Java; Verifast [27] for C and Java; Space Invader [45] and Abductor [14] for C; and Infer [15] for C, Java, Objective C, and C++. We believe that separation logic is a good fit for JavaScript verification. However, the transposition of the techniques developed for static languages to a highly complex dynamic language, such as JavaScript, is challenging.

We build on the work of Smith et al. [24], who developed a separation logic for a small fragment of ES3 (for example, no implicit coercions and only the `while` control-flow statement) with a simplified memory model (for example, no property descriptors), since their aim was to demonstrate that separation logic can be used to reason about the variable store emulated in the JavaScript heap. Despite their language simplification, the logic is complex and not suitable for automation. We improve substantially on this work. We do not simplify the semantics or the memory model of the language and the JaVerT toolchain is only restricted by the ES5 Strict coverage of the JS-2-JSIL compiler. We give a semantic definition of the Hoare triples of ES5 Strict which we interpret in JSIL Logic. JSIL Logic is sound, comparatively simpler than that of [24], and has already been automated as part of JSIL Verify.

Instead of developing JSIL Verify, we might have attempted to compile ES5 Strict to a language supported by an existing tool [7, 8, 19, 27, 14, 15]. This would have presented us with two problems worth mentioning. First, these tools all target static languages that do not support extensible objects or dynamic binding of procedure calls. Hence, JavaScript objects could not be directly encoded using the built-in constructs of these languages. Consequently, at the logical level, one would need to use custom abstractions to reason about JavaScript objects and their associated operations. Second, any program logic for JavaScript needs to take into

account all the JavaScript binary and unary operators, such as `toInt32` and `toUInt32` [1]. In JSIL, these operators are supported natively, but it is not clear that they could be expressed using the assertion languages of existing tools.

3 Overview

We describe a number of key concepts underpinning JavaScript, such as descriptors, prototype chains, scoping, and function objects, using a part of a priority queue implementation as our running example. We illustrate how to use JaVerT to verify JavaScript programs by specifying the functions given in the example. We highlight the need for having correct abstractions of various JavaScript concepts and demonstrate how this can be achieved, reaching the point where specifications are of the same complexity as those available for C++ or Java programs. We give a high-level description of our verification infrastructure and explain how to use it to create a verification tool for JavaScript.

3.1 Running Example: Priority Queue

We explain the fundamentals of JavaScript by appealing to the example given in Figure 2, which is part of an implementation of a priority queue in JavaScript, and the heap obtained from its execution (Figure 3). We implement a priority queue as a singly linked list of nodes (`Node`). Each node contains a priority (`pri`), a value (`val`), and the pointer to the next node in the queue (`next`). The nodes in the queue are ordered in descending order of priority. The functions for inspecting/manipulating the queue are stored in the node prototype object (`Node.prototype`), and are available to all node objects. One such function is `Node.prototype.insert`, which inserts a newly created node into an existing queue. It should be used in the form `q.insert(n)`, where `q` is the head of the queue into which we are inserting and `n` is the newly created node to be inserted. It returns the head of the new queue, obtained by correctly inserting `n` into the queue starting with `q`. To illustrate scope, we have a global `counter`, keeping track of how many nodes have been created, and we omit the code of `Node.prototype.insert`, to avoid clutter. Both `Node` and `insert` are labelled with unique identifiers that are used during verification.

```
1  var counter = 0;
2
3  /* @id Node */
4  var Node = function (pri, val) {
5        this.pri = pri; this.val = val; this.next = null; counter++
6  }
7
8  /* @id insert */
9  Node.prototype.insert = function (n) { ... }
10
11 var n1 = new Node(2, "foo"); var n2 = new Node (3, "bar"); var q = n1.insert(n2)
```

Figure 2 Running Example - fragment of a priority queue implemented in JavaScript

Initial Heap Before the execution of any JavaScript program, an initial heap must be established. It must contain a unique global object, which will hold all global variables, such as `counter`, `Node`, `n1`, `n2` and `q` from the running example. The initial heap must also contain the constructors and prototypes of all JavaScript built-in libraries, such as `Object`, `Function`, `Array`, and `String`, which are widely used by JavaScript programmers.

Object Properties JavaScript objects differ from those of C++ or Java in several defining ways. They are *extensible*, in that properties can be added to/removed from an object after creation. Also, JavaScript objects have two types of properties: *internal* and *named*. Internal properties are hidden from the user, are associated directly with values, and are critical for the mechanisms underlying JavaScript, such as prototype inheritance. Standard JavaScript objects have three internal properties: `@proto`, `@class`, and `@extensible`. For example, as seen in Figure 3, object `n1` is extensible, its prototype is `Node.prototype`, and its class is `"Object"`.

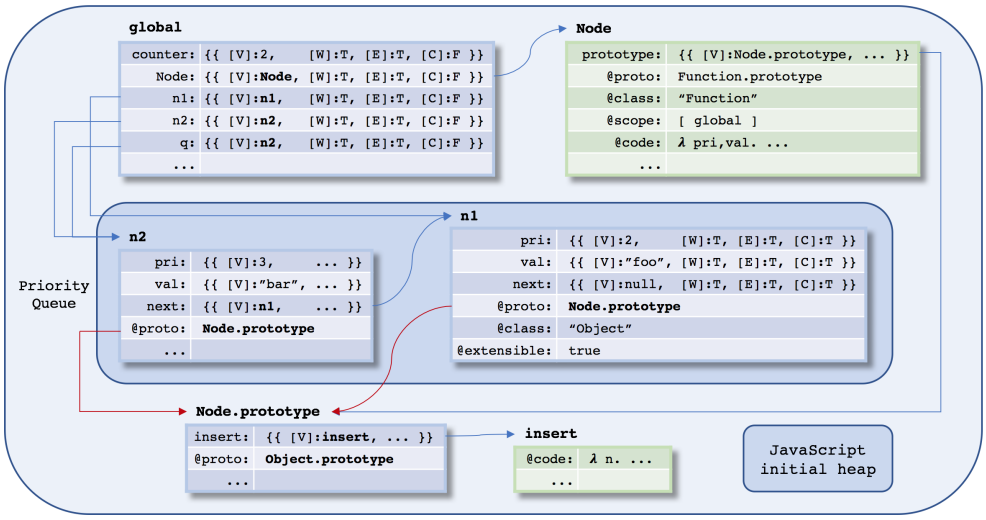


Figure 3 JavaScript heap obtained from the execution of the running example

Named properties can be thought of as object properties in the style of C++ or Java, except that they are not associated with values, but with *property descriptors*. Property descriptors are lists of *attributes*, which describe the ways in which a property can be accessed and/or modified. Depending on the attributes they contain, named properties can either be *data properties* or *accessor properties*. Data properties contain the value, writable, enumerable, and configurable attributes (denoted by [V], [W], [E], and [C]), whereas accessor properties contain get and set attributes (denoted by [G] and [S]), as well as [E] and [C]. The attributes have the following semantics: [V] holds the actual value of the property; [W] describes whether or not the property's value can be changed; [E] indicates whether or not the property will be included in a for-in enumeration; [C] allows or disallows any change to the other attributes (except for value, which it does not affect), as well as any change in the type of the property (data to accessor and vice versa); [G] and [S] play a role similar to getters and setters of Java and provide property encapsulation. Let us illustrate how JavaScript uses descriptors. If a property of an object was created using a property accessor (e.g. `this.pri = pri`), it will be writable, enumerable, and configurable (e.g. `pri`, `val`, and `next` in the object `n1`). On the other hand, if a property was declared as a variable, it will not be configurable (e.g. `counter` and `n1` in the global object).

Functions, Function objects. Functions are also stored in the JavaScript heap as objects. For instance, the functions defined in the code are represented in the final heap by the objects labelled with their respective identifiers, `Node` and `insert`. Each function object has three specific properties: **(1)** `@code`, storing the code of the original function, **(2)** `@scope`, storing a representation of the scope in which the function was defined, and **(3)** `prototype`, storing the prototype of objects created using that function as the constructor. For example, `Node.prototype` will be the prototype of all instances of `Node`, such as `n1` and `n2`.

Prototype-based inheritance. All node objects constructed using `new Node(...)` share the same prototype, which is the `Node.prototype` object. In order to determine the value of a property `p` of a given instance of `Node`, say `n1`, the semantics first checks whether `n1` has the property `p`, in which case the property lookup yields its value. Otherwise, the semantics checks if `p` belongs to the properties of `Node.prototype`, then of `Object.prototype`, and so on.

There is an interesting point to be made when it comes to the interaction of descriptors and prototype-based inheritance. Suppose that we wished to set a default priority of 0 for all nodes by assigning 0 to `"pri"` in the `Node.prototype` object as follows:

```
1 Object.defineProperty(Node.prototype, "pri", { value: 0, writable: false} )
```

The above code creates a non-writable property "pri" with value 0 in Node.prototype. While this may seem reasonable, as the default value of "pri" should not be updated, the JavaScript standard forbids assignment to a property of an object if a non-writable property of the same name exists in the prototype chain of that object. This means that any program executing new Node(...) will fail, as the code of Node includes an assignment to the property "pri".

Variable binding. In JavaScript, scope is modelled using environment records. An environment record (ER) is an internal object created upon the invocation of a function that maps the variables declared in the body of that function and its formal parameters to their respective values. Variables are resolved with respect to a list of ER locations, called a *scope chain*. For instance, the scope chain associated with the execution of the function Node contains the following two ERs, which are not represented in Figure 3 due to space constraints:

```
1 global  : { counter : -, ... }        ER-Node : { pri: -, val : - }
```

3.2 Specification of the running example

As previously described, JaVerT, given a program whose functions are annotated with specifications in the form of pre- and post-conditions written in JS Logic, verifies whether or not the code of each function satisfies its specification. JaVerT features a number of built-in predicates, allowing the user to abstract the complexity of JavaScript semantics and write specifications in a logically clear and concise manner. Moreover, the users are allowed to define their own predicates and use them to reason about more complex JavaScript structures. Here, we illustrate how to use JaVerT by specifying the functions given in the running example.

The Node Predicate. Our first aim is to specify the Node(pri, val) function. In order to do that, we need to create the appropriate abstraction/user-defined predicate for nodes, i.e. to state, using the assertion language of JaVerT, what it means to be a node:

```
Node(n, pri, val, next, nproto) := ObjectWithProto(n, nproto) * dataProp(n, "pri", pri) *
  dataProp(n, "val", val) * dataProp(n, "next", next) * types(pri: Num, val: Str)
```

A node n is a JavaScript object whose prototype is nproto, and which has three data properties: "pri", "val", and "next", with values pri, val, and next. The value pri has to be a number, as it represents the priority, whereas, for this example, we chose the value val to be a string.

Let us discuss the abstractions present in the definition of Node. ObjectWithProto(o, p) exposes only the @proto internal property, stating that the object o has prototype p, while the internal properties @extensible and @class have their default values (true and "Object", respectively). The dataProp(o, p, v) abstraction denotes that the property p of object o holds a data descriptor with value v and all other attributes set to true. This is almost equivalent[1] to the standard notion of a property p of object o having the value v, as in C++ or Java.

We will also be using the scope abstraction, which hides from the user the complexity of JavaScript scoping. In the specification below, it is sufficient for the user to state that the variable counter is visible inside the function Node, without specifying in which environment record it is actually defined. The precise meaning of scope will be given in §7.1.

There is an important point to be made here regarding these abstractions that we use. We are specifying JavaScript programs while remaining faithful to the ECMAScript standard, where a simple command such as a property assignment triggers many internal functions and inspects many internal properties. In separation logic, specifications capture the entire program footprint, i.e. all of the resources that are used/modified by the program, meaning that

[1] To be equivalent, we need [C] : false, as object properties in C++ and Java cannot be deleted.

our specifications invariably have to include all of these internal resources. JavaScript programmers cannot be expected to have a detailed knowledge of the internals and should not be required to specify their existence/management. In that context, having abstractions such as `ObjectWithProto(...)` and `dataProp(...)`, which allow the user to forget about the internals of JavaScript, is essential for bringing the tool closer to the programmer.

Specification of the `Node` function. The `Node` function is to be used as the constructor of node objects, e.g. `n1 = new Node(2, "foo")`. Therefore, at the beginning of every valid execution of `Node`, the keyword `this` is bound to a new object whose prototype is the `Node.prototype` object. The function `Node` then extends the `this` object with the properties `"pri"` and `"val"`, setting their values to `pri` and `val`, and the field `"next"`, set to `null`. Finally, each time a new node is created, the global variable `counter` is incremented by 1. The specification of `Node` is:

$$\left\{ \begin{array}{l} \texttt{ObjectWithProto(this, nproto) * ((this, "pri") -> None) * ((this, "val") -> None) *} \\ \texttt{\quad ((this, "next") -> None) * NodeProto(nproto) * (pri >= 0) * scope(counter: c) *} \\ \texttt{\quad\quad types(pri, c: Num, val: Str)} \end{array} \right\}$$
$$\texttt{Node(pri, val)}$$
$$\left\{ \texttt{Node(this, pri, val, null, nproto) * NodeProto(nproto) * scope(counter: c+1)} \right\}$$

The precondition of `Node` states: **(1)** that the keyword `this` must be initially bound to a JavaScript object whose prototype is `nproto`; **(2)** that the `this` object must not have the properties `"pri"`, `"val"`, and `"next"`, (we denote the absence of a field using the keyword `None`); **(3)** that `nproto` is a valid node prototype, which will be explained shortly; **(4)** that the priority `pri` is non-negative; **(5)** that the JavaScript variable `counter` (visible in the scope of the function `Node`) has the value denoted by the logical variable `c`; and **(6)** that the priority `pri` and the counter `c` are of number type, whereas the node value `val` is of string type.

The postcondition states that after the execution of the body of `Node`: **(1)** the keyword `this` is bound to a `Node` object with priority `pri`, value `val`, no next node, and prototype `nproto`; **(2)** `nproto` is still a valid node prototype; and **(3)** the variable `counter` is incremented by 1.

We can see that this specification is fairly straightforward, despite the underlying complexity of JavaScript, and that it resembles a specification that one might expect to write for a Java or C++ program. This is made possible precisely by the abstractions that we have used.

The `NodeProto` predicate. The `NodeProto(np)` predicate describes what it means for an object `np` to be a valid node prototype. First, it needs to capture all of the properties of the node prototype object, such as the `insert` function shown in the example. From §3.1, we also know that the node prototype cannot contain non-writable `"pri"`, `"val"`, or `"next"` properties. We choose to go with a stronger specification, where these properties are not allowed at all:

```
NodeProto(np) := objectWithProto(np, $lobj_proto) * dataProp(np, "insert", iloc) *
  fun_obj(insert, iloc, _) * ((np, "pri") -> None) * ((np, "val") -> None) *
    ((np, "next") -> None)
```

This predicate states that a node prototype `np`: **(1)** is a JavaScript object with prototype `Object.prototype` (`$lobj_proto` denotes the location of the built-in `Object.prototype` object), **(2)** has a property `"insert"` bound to the location `iloc` of the function object representing in memory the function labelled with the identifier `insert`, and **(3)** does not have the properties `"pri"`, `"val"`, and `"next"`. There is one more detail that needs to be expanded on, and it has to do with the interplay between separation logic and the prototype inheritance of JavaScript. As `Node.prototype` is shared between all node objects, we cannot simply inline the definition of `NodeProto` in the definition of the `Node` predicate. Were we to do that, we could no longer write a satisfiable assertion describing two distinct nodes using the standard separating conjunction.

The assertion `fun_obj(f_id, f_loc, f_prototype)` describes the function object stored at the location `f_loc`, representing the function labelled with `f_id` in the code and having a `prototype`

property with value f_prototype. All internal properties of any given function object can be determined from the identifier of the function that it represents.

The Queue Predicate. Now, let us turn to the definition of the Queue predicate:

```
Queue(q, primax, nproto) :=
  (q == null) * emp
```

```
Queue(q, primax, nproto) := exists pri, val, next.
  Node(q, pri, val, next, nproto) * (pri <= primax) *
    Queue(next, pri, nproto) * types(primax : Num)
```

As mentioned before, a queue Queue(q, primax, nproto) is a null-terminated list of Node objects singly linked via their next fields. All of the nodes in the queue share the same prototype nproto and have priority not greater than primax. Given our choice of the underlying data structure, the Queue predicate needs to be recursive. In the base case, we have an empty queue, meaning that q has to equal null. The maximum priority primax does not need to be specified. In the recursive case, the queue starts with a node that has priority pri, value val, and points to the next node in the queue next. The tail of the queue is also a queue, starting with the node next and maximum priority pri, which has to be not greater than primax.

Specification of Node.prototype.insert. Finally, we are ready to show the specification of Node.prototype.insert. As described earlier, the function insert is used for inserting a new node object into a queue of node objects. For example, q = n1.insert(n2) adds the node n2 to the queue whose head is n1 and returns the head of the extended priority queue which is then assigned to q. The formal specification of insert is given below:

$$\left\{ \texttt{Queue(this, pri\_q, nproto) * Node(n, pri, val, null, nproto) * NodeProto(nproto)} \right\}$$
$$\texttt{insert(n)}$$
$$\left\{ \texttt{Queue(ret, max(pri, pri\_q), nproto) * NodeProto(nproto)} \right\}$$

The precondition states that this is bound to the head of a priority queue with max priority pri_q, whose node elements all have the valid node prototype nproto. It also states that the parameter n is bound to a node object with priority pri, val, no next element, and prototype nproto. The postcondition states that insert returns the head of a priority queue with max priority max(pri, pri_q) (the keyword ret refers to the return value in the post-condition), that all node elements of this queue have the (still valid) node prototype nproto.

4 JS-2-JSIL

We introduce JSIL, our intermediate language for JavaScript verification (§4.1), and illustrate the compilation process by translating an assignment from our running example (§4.2). We describe the coverage of JS-2-JSIL (§4.3) and validate JS-2-JSIL by extensive testing against the official ECMAScript Test262 test suite, achieving a 100% success rate on the 8797 relevant tests (§4.4). We also formulate a correctness criterion for JS-2-JSIL, expressed via a simple correspondence between JavaScript and JSIL heaps. (§4.5).

4.1 The JSIL Language

JSIL is a simple dynamic goto language with top-level procedures and commands operating on object heaps. JSIL is dynamic in the sense that it supports extensible objects, dynamic field access, and dynamic procedure calls.

Syntax. Most syntactic constructs of JSIL either directly emulate those of JavaScript or are useful for JavaScript analysis. JSIL literals include the standard datatypes: strings m, numbers n, and booleans b, and the JavaScript special values undefined and null. JSIL values include the literals, object locations l, special values empty and error, types $\mathtt{t}$, and lists of values $\overline{\mathtt{v}}$. JSIL expressions include JSIL values, JSIL variables $\mathtt{x}$, a variety of unary and binary operators, the typeof operator that returns the type of an expression, and lists of expressions $\overline{\mathtt{e}}$, with the operator nth $(\mathtt{e}_1, \mathtt{e}_2)$ for accessing the $\mathtt{e}_2$-th element of list $\mathtt{e}_1$.

Syntax of the JSIL Language

Strings: $m \in \mathcal{S}tr$ Numbers: $n \in \mathcal{N}um$ Booleans: $b \in \mathcal{B}ool$ Locations: $l \in \mathcal{L}$

Variables: $\mathtt{x} \in \mathcal{X}_{\text{JSIL}}$ Literals: $\lambda \in \mathcal{L}it ::= n \mid b \mid m \mid \mathsf{undefined} \mid \mathsf{null}$

Types : $\mathtt{t} \in \mathtt{Types} ::= \mathtt{Num} \mid \mathtt{Bool} \mid \mathtt{Str} \mid \mathtt{Undef} \mid \mathtt{Null} \mid \mathtt{Empty} \mid \mathtt{Obj} \mid \mathtt{List} \mid \mathtt{Type}$

Values : $\mathtt{v} \in \mathcal{V}_{\text{JSIL}} ::= \lambda \mid l \mid \mathsf{empty} \mid \mathsf{error} \mid \mathtt{t} \mid \overline{\mathtt{v}}$

Expressions : $\mathtt{e} \in \mathcal{E}_{\text{JSIL}} ::= \mathtt{v} \mid \mathtt{x} \mid \ominus\, \mathtt{e} \mid \mathtt{e} \oplus \mathtt{e} \mid \mathsf{typeof}\,(\mathtt{e}) \mid \overline{\mathtt{e}} \mid \mathsf{nth}\,(\mathtt{e}, \mathtt{e})$

Basic Commands:

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

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

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

Notation : $\overline{\mathtt{x}}$, $\overline{\mathtt{v}}$, $\overline{\mathtt{e}}$, respectively, denote lists of variables, values, and expressions.

Basic JSIL commands essentially provide the machinery for the management of extensible objects and do not affect control flow. They include the skip command, variable assignments, object creation, as well as a number of dynamic operations on objects: field access, field assignment, field deletion, membership check, and field collection.

JSIL also includes commands related to control flow: conditional and unconditional gotos, dynamic procedure calls and ϕ-node commands, used for reasoning. The two goto commands are standard: goto i transfers control to the i-th command of the active procedure, and goto $[\mathtt{e}]\ i,\ j$ transfers control to the i-th command if $\mathtt{e}$ evaluates to `true`, and to the j-th otherwise. The dynamic procedure call $\mathtt{x} := \mathtt{e}(\overline{\mathtt{e}})$ with j first dynamically obtains the procedure name by evaluating the expression $\mathtt{e}$, then the arguments by evaluating the list of expressions $\overline{\mathtt{e}}$, then executes the procedure supplying these arguments, and finally assigns its return value to $\mathtt{x}$. If the procedure does not raise an error, the control is transferred to the next command; otherwise, it is transferred to the j-th command. The dynamic nature of procedures is inherited from the dynamic functions of JavaScript. Finally, the most complex command of the language is the ϕ-node command $\mathtt{x} := \phi(\mathtt{x}_1, \ldots, \mathtt{x}_n)$. Intuitively, this command can be interpreted as follows: there exist n paths via which this command can be reached during the execution of the program; the value assigned to $\mathtt{x}$ will be $\mathtt{x}_i$ *iff* the i-th path was taken. We include ϕ-nodes in JSIL to allow for direct support for Static-Single-Assignment (SSA), well-known to simplify analysis [17]. Moreover, our JS-2-JSIL compiler generates JSIL code directly in SSA form.

A JSIL program $\mathtt{p} \in \mathtt{P}$ is a set of top-level procedures proc $m(\overline{\mathtt{x}})\{\overline{\mathtt{c}}\}$, where $m \in \mathcal{S}tr$ is the name of the procedure, $\overline{\mathtt{x}}$ its sequence of formal parameters, and its body $\overline{\mathtt{c}}$ is a *command list* consisting of a numbered sequence of JSIL commands. We use $\mathtt{p}_m$ and $\mathtt{p}_m(i)$ to refer, respectively, to procedure m of program $\mathtt{p}$ 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_{ret} and i_{err}, that, when jumped to, respectively cause it to return normally or return an error. Also, each procedure has two dedicated variables, `ret` and `err`. When a procedure jumps to i_{ret}, it returns normally with the return value `ret`; when it jumps to i_{err}, it returns an error, with the error value `err`, which contains information about the error.

Semantics. We hope that the behaviour of JSIL is understood from familiarity with standard goto languages and our explanation of how we differ from this standard (extensible objects, dynamic fields and procedure calls). We omit the full semantics due to lack of space, and introduce only the judgements needed to state our results.

All JSIL procedures are top-level, meaning that each procedure is executed in its own

dedicated variable store. A variable store, $\rho \in \mathcal{S}to$, is a mapping from JSIL variables to JSIL values, and a JSIL heap, $\mathtt{h} \in \mathcal{H}_{\mathtt{JSIL}}$, is a mapping from locations and JSIL variables to JSIL values. JSIL expressions do not have side effects and do not depend on heap values. Given a JSIL expression $\mathtt{e}$ and a store ρ, $[\![\mathtt{e}]\!]_\rho$ denotes the value of $\mathtt{e}$ with respect to ρ.

The semantics of JSIL basic commands is described by the function $[\![.]\!] : \mathtt{BCmd} \times \mathcal{H}_{\mathtt{JSIL}} \times \mathcal{S}to \rightharpoonup \mathcal{H}_{\mathtt{JSIL}} \times \mathcal{S}to \times \mathcal{V}_{\mathtt{JSIL}}$. Informally, the judgement $[\![\mathtt{bc}]\!]_{\mathtt{h},\rho} = (\mathtt{h}', \rho', \mathtt{v})$ means that the evaluation of $\mathtt{bc}$ in the heap $\mathtt{h}$ and store ρ results in the heap $\mathtt{h}'$ and the store ρ'.

The effects of control flow commands are captured in the semantics of JSIL programs, described using the relation $\Downarrow \subseteq (\mathcal{H}_{\mathtt{JSIL}} \times \mathcal{S}to \times \mathbb{N} \times \mathbb{N}) \times \mathcal{S}tr \times (\mathcal{H}_{\mathtt{JSIL}} \times \mathcal{S}to \times \mathcal{V}_{\mathtt{JSIL}})$. Informally, the judgement $\mathtt{p} \vdash \langle \mathtt{h}, \rho, j, i \rangle \Downarrow_m \langle \mathtt{h}', \rho', \mathtt{v} \rangle$ means that the evaluation of procedure m of program $\mathtt{p}$, starting from its i-th command, to which we have arrived from its j-th command, in the heap $\mathtt{h}$ and store ρ, generates the heap $\mathtt{h}'$, the store ρ', and returns the value $\mathtt{v}$.

4.2 JS-2-JSIL: Compilation by Example

We illustrate the compilation process from JavaScript to JSIL using an assignment from our running example, namely `this.pri = pri` from the function `Node`. This seemingly innocuous statement has non-trivial behaviour and triggers a number of JavaScript internal functions. Before we show this, however, we need to describe how JavaScript scope chains are dealt with in JSIL, and we must also introduce another important JavaScript concept: *references*.

References. References are internals of JavaScript that appear, e.g., as a result of evaluating a left-hand side of an assignment, and represent resolved property bindings. A reference can be viewed as a pair consisting of a base (normally an object location) and a property name (always a string), telling us where in the heap we can find the property we are looking for. The base can hold the location of a standard object (*object reference*) or that of an environment record (*variable reference*). The global object is the only object that can be part of both object and variable references. To obtain the actual associated value (e.g. when it is used in the right-hand side of an assignment), the reference needs to be dereferenced. In the ES5 standard, dereferencing an object reference is different from dereferencing a variable reference. For the former, one has to inspect the entire prototype chain of the object. For the latter, one only has to inspect the object itself. In JSIL, we encode references as lists of three elements, containing the reference type (`"o"` or `"v"`), the base, and the referenced name.

Emulating Scope Chains in JSIL. A scope chain in ES5 Strict can be viewed as a list of ER locations. When trying to determine the value of a given variable `x` in the body of a given function `f`, the semantics needs to inspect the entire scope chain of `f` and, if it does not find a binding for `x`, the prototype chain of the global object. However, ES5 Strict is syntactically scoped and we can statically determine if a given variable is defined in a given scope chain and if so, in which ER it is defined. This means that we do not need to represent scope chains as lists of ER locations nor model the scope inspection procedure as a list traversal. Instead, we represent scope chains as special *scope chain objects*, mapping function identifiers to the locations of the corresponding ERs. Then, when trying to determine the value of a statically defined variable, we first need to determine the identifier of the function in which it is defined, which we do using a special *scope clarification function* (explained below), obtaining the corresponding function identifier. Then, we get the ER corresponding to that function identifier from the current scope chain object, and from there, read the value of the variable.

Scope Clarification Function. Given a function identifier m and a variable x, a *scope clarification function* ψ outputs the identifier m' of the function that defines x in the scope of the body of the function with identifier m. More intuitively, $\psi(m, \mathtt{x})$ tells us in which function is $\mathtt{x}$ defined, if we are currently executing function m.

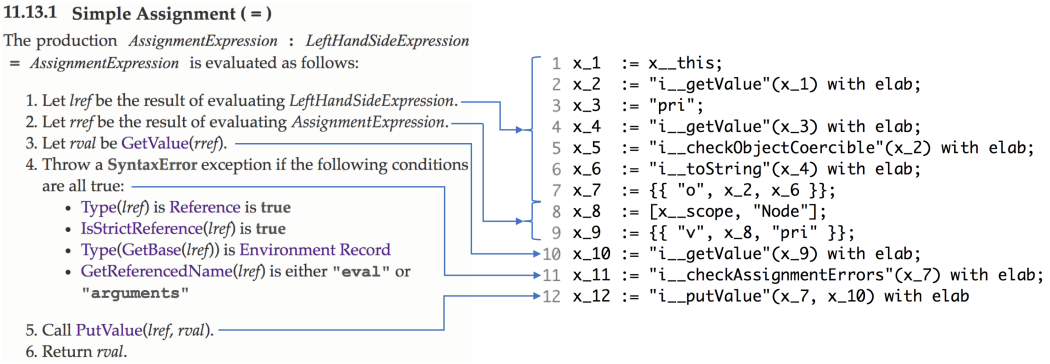


Figure 4 Compiling `this.pri = pri` to JSIL by closely following the ES5 Standard.

Compiling the Assignment. We are now ready to go step-by-step through the compilation of the assignment `this.pri = pri`, which is given in Figure 4.

1. In the first step, we evaluate the left-hand side expression of the assignment (the property accessor `this.pri`) and obtain the corresponding reference. The evaluation of property accessors is described in §11.2.1 of the ES5 standard, and it is line-by-line reflected in lines 1-7 of the JSIL code. We omit the details due to lack of space, but we do note that the reference resulting from the evaluation of a property accessor is always an object reference. In this case, this is the reference {{ `"o"`, `x_2`, `x_6` }} which, given the running example, points to the property `pri` of the object being constructed using the `new Node(...)` command.
2. Next, we do the same for the right-hand side of the assignment, the variable `pri`. Here, we need to perform variable resolution and the resulting reference will be a variable reference. For the JSIL translation, given how we emulate JavaScript scope chains, we only need to understand within which ER `pri` is defined. As `pri` is a parameter of the `Node` function (cf. Figure 2), it will be in the ER corresponding to `Node`, which we obtain from the scope chain object (line 8). The appropriate reference, {{ `"v"`, `x_8`, `"pri"`}}, is then constructed in line 9. This code is automatically generated using the scope clarification function.
3. Next, the obtained right-hand side reference is dereferenced and we get the actual value to be assigned. The dereferencing is done by the `GetValue` internal function (§8.7.1 of the ES5 standard). As we provide reference implementations of all internal functions, any call to an internal function gets translated to JSIL as a procedure call to our corresponding reference implementation, `i__getValue` (line 10). We elaborate further on `GetValue` in §6.2.
4. In ES5 Strict, the identifiers `eval` and `arguments` may not appear as the left-hand side of an assignment (e.g. `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 10), 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 internal `PutValue` function (§8.7.2 of the ES5 standard), which is translated to JSIL directly, as a procedure call to our reference implementation (line 12). We delay the details of `PutValue` until §6.2.
6. In JavaScript, every statement returns a value. In JSIL, the compiler, when given a statement to compile, returns not only the list of corresponding JSIL commands, but also the variable that stores the return value of that statement. In this particular case, the compiler would return, in addition to the presented code, the variable `x_10`.

This example illustrates how close JS-2-JSIL is to the ECMAScript standard. Out of the 12 lines of compiled JSIL code, 10 have a direct counterpart in the standard, while the remaining ones deal with scoping, which is the only notion where an observable difference is expected. This example also reveals part of the complexity behind the JavaScript assignment. We have seen two expression evaluations, one syntactic check, and calls to two different internal functions.

What is not visible yet, and will be shown in §6.2, is the hierarchy of internal functions behind `GetValue` and `PutValue`. This level of complexity is the precise reason why JavaScript analysis is difficult and why verification tools for JavaScript have not yet been developed.

4.3 JS-2-JSIL: Compiler Coverage

The ES5 standard can broadly be divided into three parts: the first part (chapters 1-7) introduces the syntax and details the parser; the second part (chapters 8-14) addresses the kernel of JavaScript, including scoping, internal functions, and language constructs; the third part (chapter 15) describes a number of built-in libraries that provide additional functionalities on top of the kernel. Strict mode features are addressed via notes interspersed throughout the standard, and most of them are also summarised in Annex C.

We do not target the correctness of the JavaScript parser, and view that as a separate project. The parser that we use, Esprima [26], is widely used and is standard-compliant. We implement the entire kernel, except the indirect eval, which by default exits strict mode, falling out of the scope of our project. As for the built-in libraries, we implement in JSIL the parts intertwined with the kernel. We implement the entire Object, Function[2], Array, Boolean, Math, and Error libraries. We implement the core of the Global library, associated with the global object; remaining functionalities target URI processing and number parsing and are orthogonal. We implement 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 not implement the RegExp, and JSON libraries, as they are orthogonal.

In summary, we cover a very large and fully representative fragment of ES5 Strict, as witnessed by our test suite coverage, detailed in §4.4. In doing so, we do not simplify the memory model of JavaScript in any way. The implementation of the remaining functionalities amounts to a (lengthy) technical exercise.

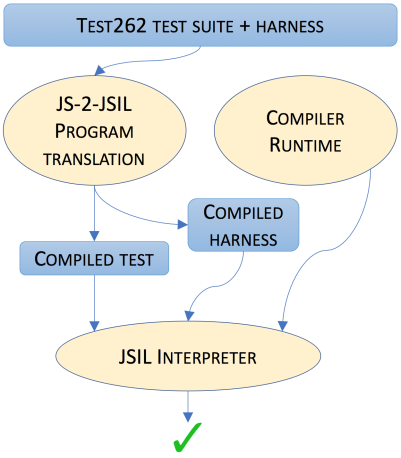
JSIL Bootstrap for JavaScript. Before we are able to run the compiled JSIL code of a JavaScript program, we need to have the basic support infrastructure in place. This infrastructure includes the setup of the JavaScript initial heap, as well as the implementations of all JavaScript internal functions and the essential functionalities of the built-in libraries. Most of these features are not directly accessible in JavaScript and as such cannot be translated using the compiler; they need to be implemented directly in JSIL.

We setup the initial heap in full, including stubs for unimplemented functions. This setup requires a fair amount of precision and takes approximately 750 lines of JSIL code. We implement all internal functions (43 of them, approx. 1000 lines of JSIL code), line-by-line following the English standard. Finally, our implementation of the built-in library functions takes approximately 3500 lines of JSIL code, again following the standard. Onward, we refer to the JSIL implementations of internal and built-in functions as the *compiler runtime*.

4.4 JS-2-JSIL Validation: Testing

ECMAScript Test262 is the official test suite for JavaScript implementations. Currently, there are two available versions of the suite: 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, which JSIL does. Tests are inconsistently flagged with respect to the mode in which they can be run, and a significant number of test cases that should be common to both strict and non-strict modes of the language contain errors preventing them

[2] The Function constructor, much like indirect eval, may exit strict mode, which we do not support. The code provided in the constructor is always executed in strict mode.



ECMAScript ES6 Test Suite	21301
ES6 constructs/libraries	8489
Annexes/Internationalisation	888
Parsing	565
Non-strict tests	890
ES5 Strict Tests	**10469**
Tests for non-implemented features	1309
Compiler Coverage	**9160**
ES5/6 differences in semantics	345
Tests using non-implemented features	18
Applicable Tests	**8797**
Tests passed	**8797**
Tests failed	0

Figure 5 Validation by testing (left); Detailed testing results (right)

from being executed correctly in strict mode. This renders any kind of systematic effort to target ES5 Strict tests borderline infeasible. All such errors have been fixed in the latest version of the tests for ES6/2015. Also, a considerable effort was made to ensure that *all* tests are correctly flagged for strict mode. For this reason, we have opted to test JS-2-JSIL using the latest version of ES6 Test262. While this does mean that we need to do more filtering to arrive at the applicable tests (e.g. we have to exclude all of the tests targeting ES6 features), this is a small price to pay for the overall increase in precision and correctness.

Addressing Incompleteness. Test262 assumes a complete implementation of JavaScript on which to run the tests. Often, a test for one language feature will make use of unrelated language features to test its result. While it is legitimate to ignore test cases that directly test unimplemented features, several options are available when these features are used for testing relevant constructs: expand the implementation to cover the missing dependency, filter out the test, or rewrite the test to avoid the unimplemented feature. We have opted for a combination of the first two solutions, and do not include rewritten tests in our testing results.

Testing Infrastructure. Due to the scale of the project and the size of the test suite, it was imperative for us to automate as much of the process as possible. To this end, we have created a continuous-integration testing infrastructure that, upon each commit to the repository of JS-2-JSIL, runs the entire Test262 suite automatically *en masse* in parallel. The individual test results for each run are logged to a database, and are used as the basis for the test analysis and filtering. We have also developed an accompanying GUI, which simplifies our interaction with the database and allows us to group tests by feature, by pass or fail, by output/error messages, etc. It also allows us to efficiently understand the progress between test runs and pinpoint any regressions that might have occurred. The infrastructure is highly modular and can be reused for a systematic testing of other related projects, such as JSCert or S5.

Running Tests. We perform the actual testing as shown in Figure 5 (left). First, we compile to JSIL the official harness of ES6 Test262. Then, for each test, we: **(1)** compile the code of the test to JSIL; **(2)** execute, using our JSIL interpreter, the JSIL code obtained by concatenating the compiled harness, the compiled test, and the compiler runtime; and **(3)** if the execution terminated normally, we declare that the test has passed.

Test Filtering and Testing Results. The breakdown of the testing results is presented in Figure 5 (right). The version of the ES6 Test262 test suite used in this study[3] contains 21301

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

individual test cases. We first filter out the test cases aimed at ES6 language constructs and libraries (8489 tests), parsing (565 tests), specification annexes (describing language extensions for web browsers, 699 tests), and the internationalisation API (189 tests), all of which fall out of the scope of this project. Next, we exclude tests for ES5 non-strict features (890 tests); we note that these include all tests for indirect eval and a number of tests for the Function constructor, which both allow the programmer to use non-strict code even when explicitly executing the code in strict mode. We obtain 10469 tests that target ES5 Strict.

Next, to filter down to the tests that should reflect the coverage of JS-2-JSIL, we remove 1309 tests for built-in library functions that have not been implemented, such as those in the Date and JSON libraries. This leaves us with the total of 9160 tests that target 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 that target such features[4]. Also, 18 tests were testing features covered by the compiler, but in doing so were using non-implemented features, and were thus excluded.

In the end, we have the final 8797 test relevant to our JS-2-JSIL compiler, of which we pass 100%. This result gives us a strong guarantee of the correctness of JS-2-JSIL and constitutes a solid foundation for our next step, the verification of compiled JSIL programs.

4.5 JS-2-JSIL: Heap Correspondence and Compiler Correctness

In order to be able to talk about the correctness of JS-2-JSIL, it is necessary to precisely relate JavaScript heaps and values obtained from the execution of JavaScript statements with the heaps and values obtained from the execution of their compiled JSIL counterparts.

The Memory Model of ES5 Strict. A JavaScript heap maps pairs of locations and property names to heap values and scope chains. Property names and JavaScript variables are taken from the same set of strings $\mathcal{X}_{\text{JS}}$. Heap values include literal values and locations (same as in JSIL), lambda abstractions $\lambda\overline{x}.s^m$, and descriptors, modelled as lists of values $\overline{v}$. We denote a heap cell by $(l, x) \mapsto v$, the union of two disjoint heaps by $h_1 \uplus h_2$, lookup by $h(l, x)$, and the empty heap by emp. Finally, each JavaScript statement returns an outcome, which can either be a value v, a function return value $\mathsf{ret}\, v$, the empty value empty, or the error value error.

ES5 Strict Memory Model

JS variables : $x \in \mathcal{X}_{\text{JS}}$	JS scope chains : $L \in \mathcal{S}ch_{\text{JS}} ::= [\,] \mid (l, m) :: L$
JS values : $v \in \mathcal{V}_{\text{JS}} ::= \lambda \mid l \mid \overline{v}$	JS heaps : $h \in \mathcal{H}_{\text{JS}}$: $\mathcal{L} \times \mathcal{X}_{\text{JS}} \rightharpoonup (\mathcal{V}^h_{\text{JS}} \cup \mathcal{S}ch_{\text{JS}})$
JS heap values : $\omega \in \mathcal{V}^h_{\text{JS}} ::= v \mid \lambda\overline{x}.s^m$	Outcomes : $o \in \mathcal{O}$ $::= v \mid \mathsf{ret}\, v \mid \mathsf{empty} \mid \mathsf{error}$

The Correspondence Relation. JavaScript objects in the JavaScript heap correspond to JSIL objects in the JSIL heap. As object allocators are by default non-deterministic, we use a partial injective function $\beta : \mathcal{L} \rightharpoonup \mathcal{L}$ to track the mapping between object locations created during the JavaScript execution of a statement s and those created during the JSIL execution of its compiled code. In Fig. 6, we express the correspondence between ES5 Strict heaps and JSIL heaps using a family of β-correspondence relations, in the style of [4].

JavaScript heaps can contain either heap values ω or scope chains L. Given our design of JSIL heaps, the values are easily relatable and are equal up to the values of the locations in their respective domains (Fig. 6, β-correspondence on outcomes (All rules) and on heaps (Other Properties, Heap Composition)).

JavaScript scope chains are represented in JSIL using *scope chain objects*. The initial scope chain object is scope chain object used for execution of top-level code, which we denote by

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

On outcomes ($\sim_\beta : \mathcal{O} \times \mathcal{O}$):

$$\frac{o \in \mathcal{L}it \cup \{\mathsf{empty}, \mathsf{error}\}}{o \sim_\beta o} \quad \text{Constants and Literals} \qquad l \sim_\beta \beta(l) \quad \text{Locations} \qquad \frac{(v_i \sim_\beta v_i')_{i=1}^n}{\{v_i\}_{i=1}^n \sim_\beta \{v_i'\}_{i=1}^n} \quad \text{Lists of values}$$

On heaps ($\sim_\beta : \mathcal{H}_{\text{JS}} \times \mathcal{H}_{\text{JSIL}}$):

$$\frac{l \sim_\beta l' \quad v \sim_\beta v' \quad x \notin \{@scope, @code\}}{(l, x) \mapsto v \sim_\beta (l', x) \mapsto v'} \quad \text{Standard Properties} \qquad \frac{h_1 \sim_\beta \mathtt{h}_1 \quad h_2 \sim_\beta \mathtt{h}_2}{h_1 \uplus h_2 \sim_\beta \mathtt{h}_1 \uplus \mathtt{h}_2} \quad \text{Heap Composition}$$

Function Objects - Body
$(l, @code) \mapsto \lambda \overline{x}.s^m \sim_\beta (\beta(l), @code) \mapsto m$

Function Objects - Scope
$(l, @scope) \mapsto L \sim_\beta (\beta(l), @scope) \mapsto l_{sc} \uplus \mathtt{SC}_\beta(L)$

Figure 6 Correspondence between ES5 Strict and JSIL heaps (β-correspondence).

$\mathtt{SC}_0(l_{sc})$ and define, given a location l_{sc}, as follows: $\mathtt{SC}_0(l_{sc}) ::= l_{sc} \mapsto \{\mathsf{main} : l_g, @proto : \mathsf{null}\}$. Furthermore, given an arbitrary scope chain L, a function $\beta : \mathcal{L} \rightharpoonup \mathcal{L}$, and a location l_{sc}, we define the scope chain object corresponding to L stored at l_{sc}, written $\mathtt{SC}_\beta(L, l_{sc})$, as follow: $\mathtt{SC}_\beta(L, l_{sc}) ::= \mathtt{SC}_0(l_{sc}) \uplus l_{sc} \mapsto \{(l_i, m_i) \mid_{i=1}^n\}$.

We are now able to represent JavaScript function objects as JSIL objects. In particular, the property @*code* of a function object is mapped onto its identifier in the JSIL heap and the property @*scope* is mapped to the location of the corresponding scope-chain object representing. For instance, the function object `Node` of the running example is modelled in JSIL as:

```
Node : { @code : "Node", @scope: $l_sc, ...}     $l_sc : { global: $lg }
```

Compiler Correctness. In order to formally state what it means for a JS-2-JSIL compiler to be correct, we assume an ES5 Strict semantic relation $\vdash \langle h, s\rangle \Downarrow \langle h', o\rangle$, where: **(i)** h is the initial heap and s the statement to evaluate, and **(ii)** h' is the final heap and o the *outcome* of the evaluation. Furthermore, we assume that a JS-2-JSIL compiler is formally captured by a function $\mathcal{C}$, mapping JavaScript statements to JSIL programs. Finally, the compiler correctness criterion is given in Definition 1. Informally, we say that a JS-2-JSIL compiler is *correct* if: "whenever a ES5 Strict program and its compilation are evaluated in two heaps related by $\sim_\beta$, the evaluation of the source program terminates *if and only if* the evaluation of its compilation also terminates, in which case the final heaps and the computed values are also related by $\sim_\beta$.

▶ **Definition 1** (Compiler Correctness). Given a JS statement s, a JSIL program $\mathtt{p}$, two heaps h and $\mathtt{h}$, such that $\mathcal{C}(s) = \mathtt{p}$ and $h \simeq_\beta \mathtt{h}$, it holds that: $\vdash \langle h, s\rangle \Downarrow \langle h_f, o\rangle \;\Leftrightarrow\; \mathtt{p} \vdash \langle \mathtt{h}, [], \_, 0\rangle \Downarrow_{\mathtt{main}} \langle \mathtt{h}_f, \rho_f, o'\rangle$, in which case there exists β' extending β such that $h_f \simeq_{\beta'} \mathtt{h}_f$ and $o \sim_{\beta'} o'$.

The β-correspondence relation is critical for defining and proving the correctness of a JS-2-JSIL compiler. We have proven our compiler correct for a fragment of ES5 Strict. This proof can be found in a separate technical appendix [42]. We do believe that the proof can be extended to the full compiler, but only in a fully mechanised setting.

5 JSIL Logic

We present JSIL logic and prove it sound with respect to the operational semantics of JSIL. Using the JSIL assertion language, we will later specify the JavaScript internal functions and verify that these specifications are satisfied by the corresponding reference implementations.

JSIL Assertions. JSIL assertions are composed of JSIL logical expressions $E \in \mathcal{E}^L$, which are, essentially, JSIL expressions extended with logical variables and the special value $\varnothing$ (read *none*, described below), and include standard boolean assertions, except disjunction; the separating conjunction; existential quantification; and assertions for describing JSIL heaps.

$$P, Q \in \mathcal{AS}_{\text{JSIL}} \quad ::= \quad \text{true} \mid \text{false} \mid E = E \mid E \;\leq\; E \mid P \wedge Q \mid \neg P \mid P * Q \mid \exists X.P \mid \mathsf{emp} \mid (E, E) \mapsto E \mid \mathsf{emptyFields}(E, E_1, ..., E_n) \tag{1}$$

An assertion may be satisfied by a triple (H, ρ, ϵ), consisting of an *abstract heap* H, a JSIL store ρ, and a logical environment ϵ, mapping logical variables to logical values. An abstract heap [24] is a mapping from pairs of locations and field names to logical values. Logical values include JSIL values as well as the special value $\varnothing$, used to denote the absence of a given field in an object. That is, we write $(l, x) \mapsto \varnothing$ to state that the object at location l has no field named x. To define abstract heaps, we thus extend the range of JSIL heaps (§4.1) with the $\varnothing$ value. Put formally, $H \in \mathcal{H}^{\emptyset}_{\texttt{JSIL}} : \mathcal{L} \times \mathcal{S}tr \rightharpoonup \mathcal{V}_{\texttt{JSIL}} \cup \{\varnothing\}$. The 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 mapped onto the value denoted by E_3. The assertion $\mathsf{emptyFields}(E, E_1, ..., E_n)$ describes an object that has no fields other than possibly those denoted by E_1, ..., E_n. All other assertions are interpreted in the standard way. The satisfaction relation for JSIL assertions can be found in the Appendix.

JSIL Basic Commands – Axiomatic Semantics. Our Hoare triples for the JSIL basic commands are of the form $\{P\}\texttt{bc}\{Q\}$, and have a partial fault-avoiding interpretation: "if bc is executed in a state satisfying P, then it will not fault and, if it terminates, it will do so in a state satisfying Q". Importantly, we assume that JSIL programs are in SSA form, i.e. that each variable can be assigned to only once. This takes away the need for standard substitutions in many of the axioms. The axioms for basic commands are given below. We use the notation $E_1 \doteq E_2$ as a shorthand for $E_1 = E_2 \wedge \texttt{emp}$.

Axiomatic Semantics of Basic Commands: : $\{P\}\texttt{bc}\{Q\}$

Skip
$$\{\texttt{emp}\}\ \mathsf{skip}\ \{\texttt{emp}\}$$

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

Var Assignment
$$\{P\}\ \texttt{x} := \texttt{e}\ \{P * \texttt{x} \doteq \texttt{e}\}$$

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

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

Deletion
$$\frac{P = (\texttt{e}_1, \texttt{e}_2) \mapsto \_}{\{P\}\ \mathsf{delete}\,(\texttt{e}_1, \texttt{e}_2)\ \{(\texttt{e}_1, \texttt{e}_2) \mapsto \varnothing\}}$$

Member Check - True
$$\frac{P = (\texttt{e}_1, \texttt{e}_2) \mapsto V * V \neq \varnothing}{\{P\}\ \texttt{x} := \mathsf{hasField}\,(\texttt{e}_1, \texttt{e}_2)\ \{P * \texttt{x} \doteq \mathsf{true}\}}$$

Member Check - False
$$\frac{P = (\texttt{e}_1, \texttt{e}_2) \mapsto \varnothing}{\{P\}\ \texttt{x} := \mathsf{hasField}\,(\texttt{e}_1, \texttt{e}_2)\ \{P * \texttt{x} \doteq \mathsf{false}\}}$$

Get Fields
$$\frac{P = ((\texttt{e}, X_i) \mapsto Y_i|_{1 \leq i \leq n}) * \mathsf{emptyFields}(\texttt{e}, X_1, ..., X_n) * (Y_i \neq \varnothing|_{1 \leq i \leq n})}{\{P\}\ \texttt{x} := \mathsf{getFields}\,(\texttt{e})\ \{P * \texttt{x} \doteq \{\{X_1, ..., X_n\}\}\}}$$

The Object Creation axiom states that the new object at x only contains the $@proto$ property with value null. The Get Fields axiom states that if the object bound to e only contains the properties denoted by X_1, ..., X_n, then, after the execution of $\texttt{x} := \mathsf{getFields}\,(\texttt{e})$, x will be bound to a list containing precisely X_1, ..., X_n. The remaining axioms are straightforward.

The basic command axioms are sound with respect to their operational semantics. This is captured in Lemma B, where abstract heaps are related to concrete heaps using an *erasure* function, $\lfloor . \rfloor : \mathcal{H}^{\emptyset}_{\texttt{JSIL}} \to \mathcal{H}_{\texttt{JSIL}} : \lfloor H \rfloor(l, x) = H(l, x) \stackrel{\text{def}}{\iff} (l, x) \in \mathsf{dom}(H) \ \wedge \ H(l, x) \neq \varnothing$.

▶ **Lemma 2** (Soundness of Basic Commands). *For all basic commands* $\texttt{bc} \in \texttt{BCmd}$, *abstract heaps* $H \in \mathcal{H}^{\emptyset}_{\texttt{JSIL}}$, *stores* $\rho \in \mathcal{S}to$, *logical environments* $\epsilon \in \mathcal{E}nv$, *and assertions* $P, Q \in \mathcal{AS}_{\texttt{JSIL}}$, *if* $\{P\}\,\texttt{bc}\,\{Q\}$ *and* $H, \rho, \epsilon \models P$, *then* $\exists_{H', \rho', \texttt{v}}.\ [\![\texttt{bc}]\!]_{\lfloor H \rfloor, \rho} = (h', \rho', \texttt{v}) \ \wedge \ \lfloor H' \rfloor = h' \ \wedge \ H', \rho', \epsilon \models Q$.

JSIL Control Flow Commands – Proof Derivations. Our goal is to use symbolic execution to prove the specifications of JSIL procedures. To this end, we define a *specification environment*, $\texttt{S} : \mathcal{S}tr \mapsto \mathcal{S}pec$, mapping procedure names onto specifications. In order to avoid clutter, we assume, in the formalisation of the logic that each procedure has a single specification. It would be straightforward to extend the current formalisation to allow for multiple

specification per procedure. A *procedure specification*, $S \in \mathcal{S}pec$, is of the form $\{P\}\, m(\overline{\mathtt{x}})\, \{Q\}$, where m is the procedure name, $\overline{\mathtt{x}}$ denote the formal parameters of the procedure, and P and Q are the pre- and postconditions of the procedure.

Since JSIL programs may contain goto operations, we cannot rely on the standard sequencing rule of Hoare logic to derive the specifications of JSIL command sequences. To remedy this, we define a transition system for relating the postcondition of every command to the preconditions of the commands that may immediately follow its execution. Transitions have the form $\mathtt{p}, \mathtt{S}, i \vdash^{j}_{m} P \rightsquigarrow Q$, where $\mathtt{S}$ is the specification environment, P and Q are the pre- and postconditions of the i-th command of procedure m in program $\mathtt{p}$, and j is the index of the command from which the symbolic execution reaches i.

In the following, we define a successor relation (Definition 3) of the form $i \mapsto^{P}_{m} j$, stating that the j-th command of procedure m may execute immediately after the i-th command of procedure m, provided that assertion P holds. When the i-th command is a conditional goto with guard $\mathtt{e}$, P is set to the assertion $\mathtt{e}$=true when the guard holds, and otherwise to $\mathtt{e}$=false.

▶ **Definition 3** (Successor relation)**.** Given a JSIL program $\mathtt{p} \in \mathtt{P}$, and a procedure m in $\mathtt{p}$, the *successor relation*, $\mapsto_m \subseteq \mathbb{N} \times \mathbb{N} \times \mathcal{AS}_{\mathtt{JSIL}}$, is defined as follows:

$$\begin{aligned}\mapsto_m \triangleq\ & \{(i, j, \text{true}) \mid \mathtt{p}_m(i) = \mathsf{goto}\ j\} \\ & \cup \{(i, j, \mathtt{e}{=}\text{true}), (i, k, \mathtt{e}{=}\text{false}) \mid \mathtt{p}_m(i){=}\mathsf{goto}\ [\mathtt{e}]\ j,\ k\} \\ & \cup \{(i, i{+}1, \text{true}) \mid i \notin \{i_{err}, i_{ret}\} \wedge \forall \mathtt{e}, j, k, (\mathtt{p}_m(i){\neq}\mathsf{goto}\ j \wedge \mathtt{p}_m(i){\neq}\mathsf{goto}\ [\mathtt{e}]\ j,\ k)\}\end{aligned}$$

We say that i is a predecessor of j *iff* j is a successor of i. We say that i is the k-th predecessor of j, written $i \overset{k}{\mapsto}_m j$, if it is the k-th element of the list containing all the predecessors of j in increasing order. We conflate the notations $i \overset{k}{\mapsto}_m j$ and $i \mapsto^{P}_{m} j$, writing $i \overset{k}{\mapsto}{}^{P}_{m} j$ instead. We can now give the proof rules for symbolically executing control flow commands.

Control Flow Commands - Symbolic Execution: $\mathtt{p}, \mathtt{S}, i \vdash^{j}_{m} P \rightsquigarrow Q$

Basic Command

$$\frac{\mathtt{p}_m(i) = \mathtt{bc} \quad \{P\}\, \mathtt{bc}\, \{Q\}}{\mathtt{p}, \mathtt{S}, i \vdash^{j}_{m} P \rightsquigarrow Q}$$

Goto

$$\frac{\mathtt{p}_m(i) = \mathsf{goto}\ k}{\mathtt{p}, \mathtt{S}, i \vdash^{j}_{m} P \rightsquigarrow P}$$

Cond. Goto

$$\frac{\mathtt{p}_m(i) = \mathsf{goto}\ [\mathtt{e}]\ k_1,\ k_2}{\mathtt{p}, \mathtt{S}, i \vdash^{j}_{m} P \rightsquigarrow P}$$

ϕ-Assignment

$$\frac{\mathtt{p}_m(i) = \mathtt{x} := \phi(\mathtt{x}_1, ..., \mathtt{x}_n) \quad j \overset{k}{\mapsto}_m i}{\mathtt{p}, \mathtt{S}, i \vdash^{j}_{m} P \rightsquigarrow P * (\mathtt{x} \doteq \mathtt{x}_k)}$$

Frame Rule

$$\frac{\mathtt{p}, \mathtt{S}, i \vdash^{j}_{m} P \rightsquigarrow Q}{\mathtt{p}, \mathtt{S}, i \vdash^{j}_{m} P * R \rightsquigarrow Q * R}$$

Existential Elimination

$$\frac{\mathtt{p}, \mathtt{S}, i \vdash^{j}_{m} P \rightsquigarrow Q}{\mathtt{p}, \mathtt{S}, i \vdash^{j}_{m} \exists \mathrm{X}.\, P \rightsquigarrow \exists \mathrm{X}.\, Q}$$

Consequence

$$\frac{\mathtt{p}, \mathtt{S}, i \vdash^{j}_{m} P \rightsquigarrow Q \quad P' \Rightarrow P \quad Q \Rightarrow Q'}{\mathtt{p}, \mathtt{S}, i \vdash^{j}_{m} P' \rightsquigarrow Q'}$$

Procedure Call

$$\frac{\mathtt{p}_m(i) = \mathtt{x} := \mathtt{e}_0(\mathtt{e}_1, ..., \mathtt{e}_k) \quad \mathtt{S}(m') = \{P\}\, m'(\mathtt{x}_1, ..., \mathtt{x}_n)\, \{Q * \mathtt{ret} \doteq \mathtt{e}\} \quad \forall_{k<j\leq n}\, \mathtt{e}_j = \mathsf{undefined}}{\mathtt{p}, \mathtt{S}, i \vdash^{j}_{m} (P[\mathtt{e}_i/\mathtt{x}_i|^{n}_{i=1}] * \mathtt{e}_0 \doteq m') \rightsquigarrow (Q[\mathtt{e}_i/\mathtt{x}_i|^{n}_{i=1}] * \mathtt{e}_0 \doteq m' * \mathtt{x} \doteq \mathtt{e}[\mathtt{e}_i/\mathtt{x}_i|^{n}_{i=1}])}$$

To establish procedure correctness, we construct *proof candidates*. As we do not support disjunction, we opt for a polyvariant analysis [5]. A proof candidate, $\mathtt{pd} \in \mathcal{D} : \mathcal{S}tr \times \mathbb{N} \rightharpoonup \wp(\mathcal{AS}_{\mathtt{JSIL}} \times \mathbb{N})$, maps each command in a procedure to a set of possible preconditions, associating each such precondition with the index of the command that led to it. For instance, if $(P, j) \in \mathtt{pd}(m, i)$, then P is the precondition of the i-th command of procedure m that resulted from the symbolic execution of its j-th command.

A proof candidate is a valid proof derivation *iff* it is *well-formed* (Definition 4). We assume, without loss of generality, that the last return index always corresponds to the command skip. A proof candidate $\mathtt{pd}$ is *well-formed* if and only if **(1)** the set of preconditions of the first

command of every procedure is the singleton set containing the precondition of the procedure itself and all postconditions of the command associated with the return label coincide with the postcondition of the procedure itself;[5] **(2)** $\mathtt{pd}$ maps the error label onto the empty set; and **(3)** all possible preconditions of every command are included in the set of postconditions of one of its predecessors (computed using the transition relation defined above).

▶ **Definition 4** (Well-formed proof candidate)**.** Given a program $\mathtt{p} \in \mathtt{P}$ and a specification function $\mathtt{S} \in \mathcal{S}tr \rightharpoonup \mathcal{S}pec$, a proof candidate $\mathtt{pd} \in \mathcal{D}$ is *well-formed* with respect to $\mathtt{p}$ and $\mathtt{S}$, written $\mathtt{p}, \mathtt{S} \vdash \mathtt{pd}$, iff for all procedures m in $\mathtt{p}$, and indexes $i, j \in \mathsf{labs}_m$ the following hold:

1. $\{P\}m(\overline{\mathtt{x}})\{Q\} = \mathtt{S}(m) \iff \mathtt{pd}(m,0) = \{(P,0)\} \wedge \exists_{k_1,\ldots,k_n}.\, \mathtt{pd}(m, i_{ret}) = \{(Q,k_i)\}_{1 \leq i \leq n}$
2. $\mathtt{pd}(m, i_{err}) = \emptyset$
3. $(P,k) \in \mathtt{pd}(m,i) \;\wedge\; i \mapsto_m^{P'} j \;\wedge\; (P \wedge P' \not\models \mathsf{false}) \implies$
   $\exists Q.\, (Q,i) \in \mathtt{pd}(m,j) \;\wedge\; \mathtt{p}, \mathtt{S}, i \vdash_m^k P \wedge P' \rightsquigarrow Q$

JSIL Logic rules are sound with respect to the JSIL operational semantics, meaning that if there is a well-formed proof candidate derivation for a JSIL program $\mathtt{p}$, then $\mathtt{p}$ will not fault and, if it terminates, it will do so in a state satisfying its postcondition (Theorem 6).

▶ **Definition 5** (Valid JSIL specification environment)**.** A specification environment $\mathtt{S} \in \mathcal{S}tr \rightharpoonup \mathcal{S}pec$ is said to be valid for a given program $\mathtt{p} \in \mathtt{P}$ if and only if for all abstract JSIL heaps $H \in \mathcal{H}^{\emptyset}_{\mathtt{JSIL}}$, stores $\rho \in \mathcal{S}to$, logical environments ϵ, and procedure names m:

if $\quad \{P\}\, m(\overline{\mathtt{x}})\, \{Q\} \in \mathtt{S}_m \wedge H, \rho, \epsilon \models P$

then $\forall \mathtt{h}, \rho'. \neg(\mathtt{p} \vdash \langle \lfloor H \rfloor, \rho, -, 0\rangle \Downarrow_m \langle \mathtt{h}, \rho', \mathsf{error}\rangle)$ **(fault-avoidance)**

$\wedge\; \forall \mathtt{h}_f, \rho_f, \mathtt{v}.\; \mathtt{p} \vdash \langle \lfloor H \rfloor, \rho, -, 0\rangle \Downarrow_m \langle \mathtt{h}_f, \rho_f, \mathtt{v}\rangle \Rightarrow \exists H_f.\; \lfloor H_f \rfloor = \mathtt{h}_f \wedge H_f, \rho_f, \epsilon \models Q$ **(safety)**

▶ **Theorem 6** (JSIL logic soundness)**.** *For any JSIL program* $\mathtt{p}$ *and specification environment* $\mathtt{S}$*, if there exists a proof candidate* $\mathtt{pd} \in \mathcal{D}$ *such that* $\mathtt{p}, \mathtt{S} \vdash \mathtt{pd}$*, then* $\mathtt{S}$ *is valid for* $\mathtt{p}$*.*

6 Specifying JavaScript Internal Functions

We illustrate our specifications of JavaScript internal functions using `GetValue` and `PutValue`, two internal functions that deal with references. We also present our `Pi` predicate, the first abstraction to precisely capture the prototype chains of JavaScript.

6.1 Capturing JavaScript prototype chains: the `Pi` predicate

To design the `Pi` predicate correctly, we need to understand the resources required for a property lookup `o.prop` in the setting of prototype inheritance. First, the `Pi` predicate has to feature **(1)** the location `l` of the object `o`, **(2)** the property `prop`, as well as **(3)** the result of the lookup, `val`. Next, we need to capture the actual prototype chain. For this, we need **(4)** a list `lloc` containing locations of the objects in the chain up to and including the one in which the property is found (or all of them if it is not found). We need two additional parameters due to the fact that String objects in JavaScript are treated differently when it comes to property lookup: they have properties that do not exist in the heap. To capture this behaviour, which we cannot detail due to space restrictions, we require **(5)** the list `lcls` of values of the `@class` internal property and **(6)** the list `lpv` of values of the `@primitiveValue` internal property, for all objects in the list `lloc`. The precise abstraction needed in order to reason about JavaScript prototype chains is `Pi(l, prop, val, lloc, lcls, lpv)`.

[5] As we assume the last command to always be `skip`, its postconditions coincide with its preconditions.

We show the definitions of `Pi` for the base case in which the property is defined in the (standard) object, and the recursive case, in which the property is not defined in the object:

```
Pi (l, prop, val, {{ l }}, {{ cls }}, {{ "" }}) :
  ((l, "@class") -> cls) * (! (cls = "String")) *
    isNamedProperty(prop) * ((l, prop) -> val) * DataDescriptor(val);

Pi (l, prop, val, {{ l :: lp :: lloc }}, {{ cls :: lcls }}, {{ pv :: lpv }}) :
   ((l, "@proto") -> lp) * ((l, "@class") -> cls) * (! (cls = "String")) *
     isNamedProperty(prop) * ((l, prop) -> None) *
       Pi (lp, prop, val, {{ lp :: lloc }}, lcls, lpv);
```

The base case reads as follows: the lookup of a named property `prop` of the object at location `l` yields the data descriptor `val`, the property is defined in the object itself, and the object is not a String object. String objects have their own set of base cases. The recursive case is slightly more complex, and reads: the lookup of a named property `prop` in the object at location `l` yields the value `val`, the property is not defined in the object itself, the object is not a String object, the prototype of the object is at location `lp`, and the lookup of `prop` in the prototype yields the value `val`. We illustrate these definitions using the object `n1` of the running example (whose prototype chain contains `n1`, `Node.prototype`, and `Object.prototype`), and the property lookups `n1.pri`, yielding the data descriptor {{ [V]: 2, [W]: T, [E]: T, [C]: T }} and `n1.foo`, yielding `undefined`. The `Pi` predicate for the former lookup, using the shown base case is:

`Pi(n1, "pri", {{ [V]: 2, [W]: T, [E]: T, [C]: T }}, {{ n1 }}, {{"Object"}}, {{""}}).`

The `Pi` predicate for the latter lookup, on the other hand, requires two unfoldings of the shown recursive case:

`Pi(n1, "foo", undefined, {{ n1 :: Node.prototype :: Object.prototype }},`
`{{"Object"::"Object"::"Object"}}, {{""::""::""}}).`

6.2 Specification by Example: `GetValue` and `PutValue`

The definitions of JavaScript internal functions in the ECMAScript standard are complex and often intertwined, making it difficult for the user to fully grasp the control flow and allowed behaviours. To illustrate: `GetValue` calls `Get`, which calls `GetProperty`, which calls `GetOwnProperty`; `PutValue` calls `Put`, which calls `CanPut`, which calls `DefineOwnProperty`, which calls `GetOwnProperty`. Specifying such dependencies axiomatically involves the joining of the specifications of all nested functions at the top level, which is highly non-trivial and results in numerous branchings. The resulting specifications, however, are much more readable than the operational definitions of the standard.

The `GetValue` internal function. `GetValue(v)` is the JavaScript internal function that performs dereferencing. It takes one parameter: the value `v` to be dereferenced. If `v` is not a reference, it is returned immediately. If `v` is a reference with the base `undefined`, a JavaScript reference error is thrown. Otherwise, `v` = {{ "o"/"v", l, prop }} and, in that case, `GetValue` returns the value associated with the property `prop` of object `l`. If `v` is a variable reference whose base is not the global object, this value is obtained by directly inspecting the heap. Otherwise, `GetValue` uses the `Get` internal function to traverse the prototype chain and obtain the appropriate value. Here, we show the specification[6] of `GetValue` for the case in which `v` is an object reference and the corresponding property is defined as a data descriptor.

```
{ (v = {{ "o", l, prop }}) * Pi (l, prop, desc, lloc, lcls, lpv) * }
{   DataDescriptor(desc) * desc_val(desc, w)                       }
                              GetValue(v)
                      { Precondition * (ret = w) }
```

[6] Due to lack of space, we omit typing information from all specifications.

In the precondition, we require an object reference v, pointing to a property prop of object l.[7] We also have that prop is defined in the prototype chain of l and that the corresponding data descriptor desc has value w. The postcondition states that, in this case, GetValue does not affect any resources[8] and returns w.

The PutValue internal function. PutValue(v, w) is, in a sense, the dual of GetValue(v). It takes two parameters: values v and w. The value v is expected to be a reference whose base is not undefined, and an error is thrown otherwise. When v = {{ "o"/"v", l, prop }} is a reference, the goal of PutValue is to assign the value w to the property prop of object l, and in order to do that, it relies on the CanPut and DefineOwnProperty internal functions. CanPut(l, prop) tells us if assigning to the property prop of object l is allowed, whereas DefineOwnProperty(l, prop, w) performs the actual assignment. The main difficulty of specifying PutValue lies in the joining of the specifications of these two functions, which feature numerous branchings and corner cases. As a result, PutValue has over twenty-five non-trivial specifications. Here, we show two of them, both relevant to the running example.

We first describe the case in which we try to assign a value to a property of an object that has not been previously defined in the prototype chain of that object. As this case involves adding a new property to an object, we will succeed only if the object is extensible.

```
{ (v = {{ "o", l, prop }}) *
    Pi (l, prop, undefined, (l :: lloc), (cls :: lcls), (pv :: lpv)) *
      (lloc = lp :: rest) * (! (cls = "Array")) * ((l, "@extensible") -> T) }

                          PutValue(v, w)

{ PreconditionExceptPi * (ret = empty) *
    Pi (l, prop, {{ "d", w, T, T, T }}, {{ l }}, {{ cls }}, {{ pv }}) *
      ((l, "@proto") -> lp) * Pi (lp, prop, undefined, lloc, lcls, lpv) }
```

The precondition states that the property prop of object l is not defined in the prototype chain of l, and that l is extensible. We also require l not to be an Array object. As the defineOwnProperty internal function for JavaScript arrays differs from that for standard objects, PutValue for arrays has a different specification. The postcondition illustrates the subtlety of the Pi predicate. First, all assertions from the precondition except the Pi(...) still hold, which we (liberally) denote by PreconditionExceptPi in the postcondition. The return value is stated to be the empty value, as required by the standard. As for the Pi, by adding the property prop to l, we break the prototype chain of the precondition into two: a single-element chain containing only l, where the property is now defined with the appropriate descriptor, and the rest of the original chain, in which the property is still undefined. This separation leaves a hanging resource (l, "@proto") → lp, hidden in the original Pi but now stated explicitly.

We conclude with the specification of PutValue(v, w) for the error case described in §3.1, when we are attempting to assign a value to a property of an object that is not yet defined in the object itself, but it is defined in its prototype chain and there it is not writable:

```
{ (v = {{ "o", l, prop }}) * Pi (l, prop, desc, lloc, lcls, lpv) *
    (lloc = (l :: lp :: rest)) * DataDescriptor(desc) *
      desc_writ(desc, F) * ((l, "@extensible") -> T)) }

                          PutValue(v, w)

                { Precondition * isTypeError(err) }
```

In the precondition, we have that prop is present in the prototype chain of l, not in l itself, and we have that the associated data descriptor desc is not writable. We also require of the

[7] This first line is common to all three specifications shown in this section.
[8] To save space, we write Precondition instead of repeating the entire precondition.

object to be extensible, because a different specification should be applied otherwise. The postcondition states that we have not affected any of the previously existing resources, and that we are throwing a JavaScript `TypeError`.

7 JaVerT

We use JaVerT to verify JavaScript programs specified using separation logic assertions. JaVerT is available online at [42], where the user can verify the running example of the paper as well as other simple JavaScript programs. Here, we present the theory that allows us to use JaVerT. We adapt the assertion language introduced in [24] to allow for automation and extend it to target full ES5 Strict heaps. We provide a translation from JS assertions to JSIL assertions, and prove that translation correct (§7.1). We give a semantic definition of Hoare triples for ES5 Strict programs and show how they can be verified using JS-2-JSIL, JSIL logic, and specifications of JavaScript internal functions (§7.2).

7.1 JS-2-JSIL: Logic Translator

JS Assertions. JS logical expressions include JSIL logical expressions as well as the special this logical expression. JS assertions include JSIL assertions as well as: **(1)** an assertion $\mathtt{scope}(x : E)$ stating that variable x is bound to the value denoted by E in the current scope and **(2)** an assertion $\mathsf{funObj}(id, E)$ describing the function object at the location denoted by E, which represents the syntactic function labelled with the identifier id.

Below, we give the satisfiability relation for these two special assertions, while the remaining cases are given in the Appendix. The satisfiability relation for JS assertions has the form: $H, \rho, L, l_t, \epsilon \models_{s,m,\psi,\psi^+} P$, where: **(1)** H is the abstract JS heap, **(2)** ρ is a mapping from the JavaScript function parameters to their initial values, **(3)** L is the scope chain, **(4)** l_t is the location bound to the this identifier, and **(5)** ϵ is the logical environment (i.e. a mapping from logical variables to logical values). The satisfiability relation takes four additional parameters, which we will generally leave implicit: **(1)** the entire JavaScript program, s, **(2)** the identifier of the function in which the assertion occurs, m, **(3)** the scope clarification function of s, ψ, and **(4)** the visibility function of s, ψ^+, which maps the identifier of each function in s to a list containing the the identifiers of all the functions that enclose it. In the following, we use: **(i)** $L(m)$ to denote to the location associated of the environment record associated with m in L, **(ii)** $\mathsf{ids}(L)$ to denote the list of function identifiers in L, and **(iii)** $s(m)$ to refer to the lambda abstraction corresponding to the function labelled with m in s.

Satisfiability relation for JS assertions: $H, \rho, L, l_{this}, \epsilon \models P$

Scope Assertion

$$H, \rho, L, l_t, \epsilon \models \mathsf{scope}(x : E) \iff$$
$$(\psi(m, x) = \mathsf{global} \land H = ((l_g, x) \mapsto \{\{\text{“}d\text{”}, [\![E]\!]^{\epsilon}_{\rho,l_t}, \mathsf{true}, \mathsf{true}, \mathsf{false}\}\}))$$
$$\lor (\psi(m, x) = m' \land m' \neq \mathsf{global} \land H = ((L(m'), x) \mapsto [\![E]\!]^{\epsilon}_{\rho,l_t}))$$

Function Object Assertion

$$H, \rho, L, l_t, \epsilon \models \mathsf{funObj}(m', E) \iff$$
$$l_f = [\![E]\!]^{\epsilon}_{\rho,l_t} \land H = ((l_f, @code) \mapsto s(m')) \uplus ((l_f, @scope) \mapsto L) \land \mathsf{ids}(L) = \psi^+(m')$$

Translation. In the Appendix, we give the complete translation from JS Logic assertions to JSIL Logic assertions. Here, we show the translation of the scope lookup and function object assertions and of JavaScript function specifications, as the remaining cases are straightforward. The translation is implicitly parameterised with a function identifier m, the *closure clarification function* ψ, and the visibility function ψ^+, described above. The translation of the scope assertion $\mathtt{scope}(x : E)$ uses the scope clarification function to determine the function in which

x is defined. If x is defined in the global object, it needs to be associated with a data descriptor, whereas if it is defined in an ER, it is associated directly with its corresponding value. To translate a procedure specification, the formal parameters of the procedure are augmented with the $\mathtt{x}_{sc}$ and $\mathtt{x}_{this}$ variables denoting the locations of the scope chain and this objects. When translating the pre- and postcondition, since P and Q may contain scope assertions, we must also produce the resources describing the expected scope chain object. We lift the translation to logical environments by applying it to all logical expressions in the logical environments.

Translation of Assertions and Specifications

Scope Assertion - Non-global

$$\frac{\psi_m(x) = m' \neq \mathsf{global} \quad \mathcal{T}(E) = E'}{\mathcal{T}(\mathsf{scope}(x : E)) \triangleq (X_{m'}, x) \mapsto E'}$$

Scope Assertion - global

$$\frac{\psi_m(x) = \mathsf{global} \quad \mathcal{T}(E) = E'}{\mathcal{T}(\mathsf{scope}(x : E)) \triangleq (l_g, x) \mapsto \{\{\text{“}d\text{”}, E', \mathsf{true}, \mathsf{true}, \mathsf{false}\}\}}$$

Function Object

$$\frac{\mathcal{T}(E) = E' \quad \psi^+(m') = [\mathsf{main}, m_1, ..., m_n] \quad P = ((X_{sc}, m_1) \mapsto X_1) * ... * ((X_{sc}, m_n) \mapsto X_n) * ((X_{sc}, \mathsf{main}) \mapsto l_g)}{\mathcal{T}(\mathsf{funObj}(m', E)) \triangleq \exists_{X_{sc}, X_1, ..., X_n}. (E', @code) \mapsto m' * (E', @scope) \mapsto X_{sc} * P}$$

Function Specification

$$\frac{\mathcal{T}(P) = P' \quad \mathcal{T}(Q) = Q' \quad \psi^+(m) = [\mathsf{main}, m_1, ..., m_n] \quad P'' = (\mathtt{x}_{sc}, \mathsf{main}) \mapsto l_g * (\mathtt{x}_{sc}, m_1) \mapsto Y_{m_1} * ... * (\mathtt{x}_{sc}, m_n) \mapsto Y_{m_n}}{\mathcal{T}(\{P\}\, m(\overline{\mathtt{x}})\, \{Q\}) \triangleq \{P'' * P'\}\, m(\mathtt{x}_{sc}, \mathtt{x}_{this}, \overline{\mathtt{x}})\, \{P'' * Q'\}}$$

Below, we give the compiled specification of the function `Node` given in §3. Note that this specification uses the predicates `Node` and `NodeProto` exactly as they are defined in §3, since their definitions are not changed by the translation of assertions. In a nutshell, the translation of this specification simply adds the footprint of the scope chain of the function to both the pre- and postconditions and replaces the occurrences of `this` for `x_this` and of the assertion `scope(counter: c)` for its appropriate translation.

```
{ ((x_sc, "global") -> $lobj_global) * ((x_sc, "Node") -> _) * (pri >= 0) *
  ObjectWithProto(this, nproto) * ((x_this, "pri") -> None) *
    ((this, "val") -> None) * ((x_this, "next") -> None) * NodeProto(nproto) *
      (($lobj_global, counter) -> {{"d", c, true, true, false}}) * }

                        Node(x_sc, x_this, pri, val)

{ ((x_sc, "global") -> $lobj_global) * ((x_sc, "Node") -> _) *
  Node(x_this, pri, val, null, nproto) * NodeProto(nproto) *
    (($lobj_global, counter) -> {{"d", c+1, true, true, false}}) }
```

In the following, we use $\rho \sim_\beta \rho'$ and $\epsilon \sim_\beta \epsilon'$ to denote the point-wise extension of $\sim_\beta$ to the domains of ρ and ϵ. In order to state the correctness of the translation of assertions, we need to extend the β-correspondence to logical contexts: we say that $H_{\mathtt{JS}}, \rho, L, l_t \simeq_\beta H_{\mathtt{JSIL}}, \rho'$ holds iff **(1)** $H_{\mathtt{JS}} \sim_\beta H_{\mathtt{JSIL}}$, **(2)** $\rho \sim_\beta \rho'$, and **(3)** $\rho'(\mathtt{x}_{\mathsf{this}}) \mapsto \beta(l_t)$. Theorem 7 states that for any JS and JSIL logical contexts in the β-correspondence relation, the JS context satisfies a JS assertion P if and only if the JSIL context satisfies its translation $\mathcal{T}(P)$.

▶ **Theorem 7** (Assertion translation correctness). *For any two JavaScript and JSIL logical contexts $H_{\mathtt{JS}}, \rho, L, l_t$ and $H_{\mathtt{JSIL}}, \rho'$, such that $H_{\mathtt{JS}}, \rho, L, l_t \simeq_\beta H_{\mathtt{JSIL}}, \rho'$ and $\epsilon \sim_\beta \epsilon'$, it holds that: $H_{\mathtt{JS}}, \rho, L, l_t, \epsilon \models P$ iff $H_{\mathtt{JSIL}}, \rho', \epsilon'[X_{m_1} \mapsto \beta(L(m_1)), ..., X_{m_n} \mapsto \beta(L(m_n))] \models \mathcal{T}(P)$, where $\mathsf{ids}(L) = \{m_1, ..., m_n\}$.*

7.2 JS Logic

We give a semantic definition of the Hoare triples for ES5 Strict programs based on its operational semantics. Given a correct compiler from ES5 Strict to JSIL, we can use our verification

infrastructure together with our translation from JavaScript assertions to JSIL assertions to validate ES5 Strict Hoare triples by appealing to the JSIL logic.

JS-2-JSIL Logic correspondence. In Theorem 8 below, we establish a correspondence between Hoare triples at JS level and specification environments at the JSIL level. JS Hoare triples have a partial, fault-avoiding semantic interpretation defined by: $\{P\}\, s\, \{Q\}$ holds if and only if "whenever s is executed in a state satisfying P, then it does not fault, and, if it terminates, it does so in a state satisfying Q". Informally, Theorem 8 states that a valid JS Hoare triple corresponds to a valid JSIL specification environment and vice-versa. In order to precisely relate JS Hoare triples with specification environments, we extend the translation function for assertions and specifications to programs. Program translation is done piecemeal by extracting its functions and translating their specifications. The specification of the program itself is translated as the top level main procedure. Let specs be a function that, given a statement s as input, returns the specifications of the function literals declared in s. The *program translation* is then defined as:

Top Level Specification

$$\frac{\mathcal{T}(P) = P' \quad \mathcal{T}(Q) = Q' \quad \{P'\}\ \mathsf{main}\ (\mathtt{x}_{sc}, \mathtt{x}_{this})\ \{Q'\} \in \mathtt{S} \quad \forall \{\hat{P}\}\, m(\overline{\mathtt{x}})\, \{\hat{Q}\} \in \mathsf{specs}(s) \Rightarrow \mathcal{T}(\{\hat{P}\}\, m(\overline{\mathtt{x}})\, \{\hat{Q}\}) \in \mathtt{S}}{\mathcal{T}(\{P\}\, s\, \{Q\}) \triangleq \mathtt{S}}$$

▶ **Theorem 8** (JS-2-JSIL Logic correspondence). *Given a correct JS-2-JSIL compiler $\mathcal{C}$, for every JS statement s and JS assertions P and Q: $\{P\}\, s\, \{Q\}$ iff $\mathcal{T}(\{P\}\, s\, \{Q\})$ is a valid specification environment for $\mathcal{C}(s)$.*

▶ **Theorem 9** (Proving JS Hoare Triples in JSIL). *For every JS statement s and JS assertions P and Q: $(\exists \mathtt{pd} \cdot \mathcal{C}(s), \mathcal{T}(\{P\}\, s\, \{Q\}) \vdash \mathtt{pd}) \Rightarrow \{P\}\, s\, \{Q\}$*

8 Conclusions and Future Work

We have developed a JSIL verification infrastructure for tractable symbolic verification of JavaScript programs (ECMAScript 5 Strict mode) and the semi-automatic JavaScript Verification Toolchain, JaVerT, built on top of this infrastructure. As far as we are aware, JaVerT is the first logic-based symbolic verifier for JavaScript. We have taken great care to validate our infrastructure, with substantial filtered testing using the ECMAScript test suite, partial correctness of the JS-2-JSIL compiler and logic translator, the soundness of JSIL logic, and the formal specification of the JavaScript internal functions and the verification of their JSIL reference implementations using JSIL Verify.

We have substantial experience with using JSIL Verify to validate the internal function specifications. We are gaining experience with JaVerT, currently verifying functionally correct properties of JavaScript implementations of simple data structures, such as the priority queue. Our immediate aim is to hone JaVerT and better understand its capabilities and limitations. In future, we will extend our JSIL logic with higher-order reasoning, and develop an automated tool based on bi-abduction [14] for verifying large JavaScript code. Nonetheless, we believe that the semi-automated tool will always have a role to play in the development of functionally correct library specifications.

We believe that our infrastructure can be used for other styles of JavaScript analysis. We have built a prototype JSIL front-end to CBMC [41], with the aim of finding cross-scripting vulnerabilities, and are investigating the viability of a JSIL front-end to the Rosette symbolic analyser. Our ultimate goal is to establish our JSIL infrastructure as a common platform for JavaScript verification.

References

1 The 5th edition of ECMA 262. ECMAScript Language Specification. Technical report, ECMA, 2011.

2 Christopher Anderson, Paola Giannini, and Sophia Drossopoulou. Proceedings of the towards type inference for JavaScript. In *19th European Conference Object-Oriented Programming*, Lecture Notes in Computer Science, pages 428–452. Springer, 2005.

3 Esben Andreasen and Anders Møller. Determinacy in static analysis for jquery. In *OOPSLA*, 2014.

4 Anindya Banerjee and David A. Naumann. Secure information flow and pointer confinement in a java-like language. In *Proceedings of the 15th IEEE Computer Security Foundations Workshop*, CSF'15, pages 253–267. IEEE Computer Society, 2002.

5 Gilles Barthe and Tamara Rezk. Non-interference for a JVM-like language. In *Proceedings of the ACM SIGPLAN International Workshop on Types in Languages Design and Implementation (TLDI'05)*, pages 103–112, 2005.

6 Gilles Barthe, Tamara Rezk, and Ando Saabas. Proof obligations preserving compilation. In Theodosis Dimitrakos, Fabio Martinelli, Peter Y. A. Ryan, and Steve A. Schneider, editors, *Formal Aspects in Security and Trust, Third International Workshop, FAST 2005, Newcastle upon Tyne, UK, July 18-19, 2005, Revised Selected Papers*, volume 3866 of *Lecture Notes in Computer Science*, pages 112–126. Springer, 2005.

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

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

9 Gavin M. Bierman, Martín Abadi, and Mads Torgersen. Understanding TypeScript. In *Proceedings of the 28th European Conference on Object-Oriented Programming (ECOOP'14)*, Lecture Notes in Computer Science, pages 257–281. Springer, 2014.

10 Lars Birkedal, Rasmus Ejlers Møgelberg, Jan Schwinghammer, and Kristian Støvring. First steps in synthetic guarded domain theory: step-indexing in the topos of trees. *Logical Methods in Computer Science*, 8(4), 2012.

11 Lars Birkedal, Nick Rothwell, Mads Tofte, and David N. Turner. The ml kit, version 1. Technical report, Technical Report 93/14 DIKU, 1993.

12 Martin Bodin, Arthur Charguéraud, Daniele Filaretti, Philippa Gardner, Sergio Maffeis, Daiva Naudziuniene, Alan Schmitt, and Gareth Smith. A Trusted Mechanised JavaScript Specification. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL'13, pages 87–100. ACM Press, 2013.

13 Cristian Cadar, Daniel Dunbar, and Dawson R. Engler. KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs. In Richard Draves and Robbert van Renesse, editors, *8th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2008, December 8-10, 2008, San Diego, California, USA, Proceedings*, pages 209–224. USENIX Association, 2008.

14 C. Calcagno, Dino Distefano, Peter O'Hearn, and Hongseok Yang. Compositional shape analysis by means of bi-abduction. In *POPL*, 2009.

15 Cristiano Calcagno, Dino Distefano, Jérémy Dubreil, Dominik Gabi, Pieter Hooimeijer, Martino Luca, Peter O'Hearn, Irene Papakonstantinou, Jim Purbrick, and Dulma Rodriguez. Moving fast with software verification. In *NASA Formal Methods*, pages 3–11. Springer, 2015.

16 Arthur Charguéraud. Pretty-big-step semantics. In *Programming Languages and Systems*, volume 7792 of *Lecture Notes in Computer Science*, pages 41–60. Springer Berlin Heidelberg, 2013.

17 Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. An efficient method of computing static single assignment form. In *Conference Record of the*

Sixteenth Annual ACM Symposium on Principles of Programming Languages, Austin, Texas, USA, January 11-13, 1989, pages 25–35. ACM Press, 1989.

18 Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'08/ETAPS'08, pages 337–340, Berlin, Heidelberg, 2008. Springer-Verlag.

19 Dino Distefano and M. Parkinson. jStar: Towards practical verification for Java. In *OOPSLA*, 2008.

20 Asger Feldthaus and Anders Møller. Checking correctness of TypeScript interfaces for JavaScript libraries. In *Proceedings of the 29th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA*, 2014.

21 Asger Feldthaus, Max Schäfer, Manu Sridharan, Julian Dolby, and Frank Tip. Efficient construction of approximate call graphs for javascript ide services. In *2013 35th International Conference on Software Engineering (ICSE)*, pages 752–761. IEEE, 2013.

22 Stephen Fink and Julian Dolby. WALA — The T.J. Watson Libraries for Analysis. http://wala.sourceforge.net/.

23 Cédric Fournet, Gurvan Le Guernic, and Tamara Rezk. A security-preserving compiler for distributed programs: from information-flow policies to cryptographic mechanisms. In Ehab Al-Shaer, Somesh Jha, and Angelos D. Keromytis, editors, *Proceedings of the 2009 ACM Conference on Computer and Communications Security, CCS 2009, Chicago, Illinois, USA, November 9-13, 2009*, pages 432–441. ACM, 2009.

24 Philippa Gardner, Sergio Maffeis, and Gareth Smith. Towards a program logic for JavaScript. In *Proceedings of the 40th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL'13, pages 31–44. ACM Press, 2012.

25 Arjun Guha, Claudiu Saftoiu, and Shriram Krishnamurthi. The essence of Javascript. In *Proceedings of the 24th European Conference on Object-Oriented Programming (ECOOP)*, Lecture Notes in Computer Science, pages 126–150. Springer, 2010.

26 Ariya Hidayat. Esprima : ECMAScript parsing infrastructure for multipurpose analysis. http://esprima.org/, 2012.

27 Bart Jacobs, Jan Smans, Pieter Philippaerts, Frédéric Vogels, Willem Penninckx, and Frank Piessens. Verifast: A powerful, sound, predictable, fast verifier for c and java. In *NASA Formal Methods*, pages 41–55. Springer, 2011.

28 Dongseok Jang and Kwang-Moo Choe. Points-to analysis for javascript. In *Proceedings of the 2009 ACM symposium on Applied Computing*, pages 1930–1937. ACM, 2009.

29 Simon Holm Jensen, Anders Møller, and Peter Thiemann. Type analysis for JavaScript. In *Proceedings of the 16th International Static Analysis Symposium (SAS)*, volume 5673 of *Lecture Notes in Computer Science*, pages 238–255. Springer, 2009.

30 JSIR, An Intermediate Representation for JavaScript Analysis. `http://too4words.github.io/jsir/`.

31 Vineeth Kashyap, Kyle Dewey, Ethan A. Kuefner, John Wagner, Kevin Gibbons, John Sarracino, Ben Wiedermann, and Ben Hardekopf. JSAI: a static analysis platform for javascript. In *FSE*, pages 121–132, 2014.

32 Daniel Kroening and Michael Tautschnig. CBMC – C bounded model checker. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 8413 of *LNCS*, pages 389–391. Springer, 2014.

33 Microsoft. TypeScript language specification. Technical report, Microsoft, 2014.

34 Changhee Park and Sukyoung Ryu. Scalable and precise static analysis of javascript applications via loop-sensitivity. In *ECOOP*, pages 735–756, 2015.

35 Daejun Park, Andrei Stefănescu, and Grigore Roşu. Kjs: A complete formal semantics of javascript. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2015, pages 346–356, New York, NY, USA, 2015. ACM.

36 Joe Gibbs Politz, Matthew J. Carroll, Benjamin S. Lerner, Justin Pombrio, and Shriram Krishnamurthi. A Tested Semantics for Getters, Setters, and Eval in JavaScript. In *Proceedings of the 8th Symposium on Dynamic Languages*, 2012.

37 Aseem Rastogi, Nikhil Swamy, Cédric Fournet, Gavin Bierman, and Panagiotis Vekris. Safe & efficient gradual typing for TypeScript. In *Proceedings of the 42nd ACM Symposium on Principles of Programming Languages*. ACM Press, 2015.

38 Jan Schwinghammer, Lars Birkedal, Bernhard Reus, and Hongseok Yang. Nested hoare triples and frame rules for higher-order store. *Logical Methods in Computer Science*, 7(3), 2011.

39 Manu Sridharan, Julian Dolby, Satish Chandra, Max Schäfer, and Frank Tip. Correlation tracking for points-to analysis of javascript. In *ECOOP*, pages 435–458, 2012.

40 Kasper Svendsen and Lars Birkedal. Impredicative concurrent abstract predicates. In *Programming Languages and Systems - 23rd European Symposium on Programming, ESOP 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014, Proceedings*, pages 149–168, 2014.

41 CBMC Team. The JSIL front end of CBMC. https://github.com/diffblue/cbmc/pull/51, https://github.com/diffblue/cbmc/pull/91.

42 JSIL Team. JSIL as a Service. http://goo.gl/au69SV, 2016.

43 Peter Thiemann. Towards a type system for analysing JavaScript programs. In *Proceedings of the 14th European Symposium on Programming Languages and Systems*, Lecture Notes in Computer Science, pages 408–422. Springer, 2005.

44 Willem Visser, Corina S. Păsăreanu, and Sarfraz Khurshid. Test input generation with java pathfinder. In *Proceedings of the 2004 ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA '04, pages 97–107, New York, NY, USA, 2004. ACM.

45 Hongseok Yang, Oukseh Lee, Josh Berdine, C. Calcagno, Byron Cook, Dino Distefano, and Peter O'Hearn. Scalable shape analysis for systems code. In *CAV '08: Proc. of the 20th international conference on Computer Aided Verification*, pages 385–398, Berlin, Heidelberg, 2008. Springer-Verlag.

A JSIL Semantics

Semantics of JSIL Expressions: $[\![\mathtt{e}]\!]_\rho = \mathtt{v}$

Literal

$$\frac{}{[\![\mathtt{v}]\!]_\rho \triangleq \mathtt{v}}$$

Variable

$$\frac{}{[\![\mathtt{x}]\!]_\rho \triangleq \rho(\mathtt{x})}$$

Unary Operator

$$\frac{\mathtt{v} = [\![\mathtt{e}]\!]_\rho}{[\![\ominus \mathtt{e}]\!]_\rho \triangleq \overline{\ominus}(\mathtt{v})}$$

Binary Operator

$$\frac{\mathtt{v}_1 = [\![\mathtt{e}_1]\!]_\rho \quad \mathtt{v}_2 = [\![\mathtt{e}_2]\!]_\rho}{[\![\mathtt{e}_1 \oplus \mathtt{e}_2]\!]_\rho \triangleq \overline{\oplus}(\mathtt{v}_1, \mathtt{v}_2)}$$

TypeOf

$$\frac{\mathtt{v} = [\![\mathtt{e}]\!]_\rho \qquad \mathtt{t} = \mathsf{TypeOf}(\mathtt{v})}{[\![\mathsf{typeof}\,(\mathtt{e})]\!]_\rho \triangleq \mathtt{t}}$$

Expression List

$$\frac{}{[\![\{\{\mathtt{e}_1, ..., \mathtt{e}_n\}\}]\!]_\rho \triangleq \{\{[\![\mathtt{e}_1]\!]_\rho, ..., [\![\mathtt{e}_n]\!]_\rho\}\}}$$

Nth-Access

$$\frac{\{\{\mathtt{v}_0, ..., \mathtt{v}_n\}\} = [\![\mathtt{e}_1]\!]_\rho \qquad k = [\![\mathtt{e}_2]\!]_\rho \qquad 0 \leq k \leq n}{[\![\mathsf{nth}\,(\mathtt{e}_1, \mathtt{e}_2)]\!]_\rho \triangleq \mathtt{v}_k}$$

Semantics of Basic Commands: $[\![\mathtt{bc}]\!]_{h,\rho} = (h', \rho', \mathtt{v})$

Skip

$$[\![\mathsf{skip}\,]\!]_{\mathtt{h},\rho} \triangleq (\mathtt{h}, \rho, \mathsf{empty})$$

Assignment

$$\frac{[\![\mathtt{e}]\!]_\rho = \mathtt{v} \quad \rho' = \rho[\mathtt{x} \mapsto \mathtt{v}]}{[\![\mathtt{x} := \mathtt{e}]\!]_{\mathtt{h},\rho} \triangleq (\mathtt{h}, \rho', \mathtt{v})}$$

Object Creation

$$\frac{\mathtt{h}' = \mathtt{h} \uplus (l, @proto) \mapsto \mathsf{null} \quad \rho' = \rho[\mathtt{x} \mapsto l]}{[\![\mathtt{x} := \mathsf{new}\,()]\!]_{\mathtt{h},\rho} \triangleq (\mathtt{h}', \rho', \mathsf{true})}$$

Property Projection

$$\frac{\mathtt{v} = \mathtt{h}([\![\mathtt{e}_1]\!]_\rho, [\![\mathtt{e}_2]\!]_\rho) \quad \rho' = \rho[\mathtt{x} \mapsto \mathtt{v}]}{[\![\mathtt{x} := [\mathtt{e}_1, \mathtt{e}_2]]\!]_{\mathtt{h},\rho} \triangleq (\mathtt{h}, \rho', \mathtt{v})}$$

Property Assignment

$$\frac{\mathtt{v} = [\![\mathtt{e}_3]\!]_\rho \quad \mathtt{h}' = \mathtt{h}[([\![\mathtt{e}_1]\!]_\rho, [\![\mathtt{e}_2]\!]_\rho) \mapsto \mathtt{v}]}{[\![[\mathtt{e}_1, \mathtt{e}_2] := \mathtt{e}_3]\!]_{\mathtt{h},\rho} \triangleq (\mathtt{h}', \rho, \mathtt{v})}$$

Property Deletion - Existing

$$\frac{\mathtt{h} = \mathtt{h}' \uplus ([\![\mathtt{e}_1]\!]_\rho, [\![\mathtt{e}_2]\!]_\rho) \mapsto \mathtt{v}}{[\![\mathsf{delete}\,(\mathtt{e}_1, \mathtt{e}_2)]\!]_{\mathtt{h},\rho} \triangleq (\mathtt{h}', \rho, \mathsf{true})}$$

Property Deletion - Non-Existing

$$\frac{([\![\mathtt{e}_1]\!]_\rho, [\![\mathtt{e}_2]\!]_\rho) \notin \mathsf{dom}(\mathtt{h})}{[\![\mathsf{delete}\,(\mathtt{e}_1, \mathtt{e}_2)]\!]_{\mathtt{h},\rho} \triangleq (\mathtt{h}, \rho, \mathsf{true})}$$

Member Check - True

$$\frac{([\![\mathtt{e}_1]\!]_\rho, [\![\mathtt{e}_2]\!]_\rho) \in \mathsf{dom}(\mathtt{h}) \quad \rho' = \rho[\mathtt{x} \mapsto \mathsf{true}]}{[\![\mathtt{x} := \mathsf{hasField}\,(\mathtt{e}_1, \mathtt{e}_2)]\!]_{\mathtt{h},\rho} \triangleq (\mathtt{h}, \rho', \mathsf{true})}$$

Member Check - False

$$\frac{([\![\mathtt{e}_1]\!]_\rho, [\![\mathtt{e}_2]\!]_\rho) \notin \mathsf{dom}(\mathtt{h}) \quad \rho' = \rho[\mathtt{x} \mapsto \mathsf{false}]}{[\![\mathtt{x} := \mathsf{hasField}\,(\mathtt{e}_1, \mathtt{e}_2)]\!]_{\mathtt{h},\rho} \triangleq (\mathtt{h}, \rho', \mathsf{false})}$$

GetFields

$$\frac{\begin{array}{l} l = [\![\mathtt{e}]\!]_\rho \quad \mathtt{h} = (\mathtt{h}' \uplus (l, m_1) \mapsto - \uplus \ ... \ \uplus (l, m_n) \mapsto -) \quad l \notin \mathsf{dom}(\mathtt{h}') \\ \mathtt{v} = \{\{m_1, ..., m_n\}\} \quad \rho' = \rho[\mathtt{x} \mapsto \mathtt{v}] \end{array}}{[\![\mathtt{x} := \mathsf{getFields}\,(\mathtt{e})]\!]_{\mathtt{h},\rho} \triangleq (\mathtt{h}, \rho', \mathtt{v})}$$

Semantics of control flow commands: $\mathtt{p} \vdash \langle h, \rho, i, j\rangle \Downarrow_m \langle h', \rho', \mathtt{v}\rangle$

Basic Command

$$\frac{\begin{array}{l} \mathtt{p}_m(i) = \mathtt{c} \in \mathtt{BCmd} \\ [\![\mathtt{c}]\!]_{\mathtt{h},\rho} = (\mathtt{h}', \rho', -) \\ \mathtt{p} \vdash \langle \mathtt{h}', \rho', i, i+1\rangle \Downarrow_m \langle \mathtt{h}'', \rho'', \mathtt{v}\rangle \end{array}}{\mathtt{p} \vdash \langle \mathtt{h}, \rho, \_, i\rangle \Downarrow_m \langle \mathtt{h}'', \rho'', \mathtt{v}\rangle}$$

Goto

$$\frac{\begin{array}{l} \mathtt{p}_m(i) = \mathsf{goto}\ j \\ \mathtt{p} \vdash \langle \mathtt{h}, \rho, i, j\rangle \Downarrow_m \langle \mathtt{h}', \rho', \mathtt{v}\rangle \end{array}}{\mathtt{p} \vdash \langle \mathtt{h}, \rho, \_, i\rangle \Downarrow_m \langle \mathtt{h}', \rho', \mathtt{v}\rangle}$$

Cond. Goto - True

$$\frac{\begin{array}{l} \mathtt{p}_m(i) = \mathsf{goto}\ [\mathtt{e}]\ j,\ k \quad [\![\mathtt{e}]\!]_\rho = \mathsf{true} \\ \mathtt{p} \vdash \langle \mathtt{h}, \rho, i, j\rangle \Downarrow_m \langle \mathtt{h}', \rho', \mathtt{v}\rangle \end{array}}{\mathtt{p} \vdash \langle \mathtt{h}, \rho, \_, i\rangle \Downarrow_m \langle \mathtt{h}', \rho', \mathtt{v}\rangle}$$

Cond. Goto - False

$$\frac{\begin{array}{l} \mathtt{p}_m(i) = \mathsf{goto}\ [\mathtt{e}]\ j,\ k \quad [\![\mathtt{e}]\!]_\rho = \mathsf{false} \\ \mathtt{p} \vdash \langle \mathtt{h}, \rho, i, k\rangle \Downarrow_m \langle \mathtt{h}', \rho', \mathtt{v}\rangle \end{array}}{\mathtt{p} \vdash \langle \mathtt{h}, \rho, \_, i\rangle \Downarrow_m \langle \mathtt{h}', \rho', \mathtt{v}\rangle}$$

Normal Return

$$\vdash \langle \mathtt{h}, \rho, \_, i_{ret}\rangle \Downarrow_m \langle \mathtt{h}, \rho, \rho(\mathtt{ret})\rangle$$

Error Return

$$\vdash \langle \mathtt{h}, \rho, \_, i_{err}\rangle \Downarrow_m \langle \mathtt{h}, \rho, \mathsf{error}\rangle$$

PROCEDURE CALL

$$\frac{\mathtt{p}_m(i) = \mathtt{x} := \mathtt{e}(\mathtt{e}_1, ..., \mathtt{e}_{n_1}) \text{ with } j \qquad [\![\mathtt{e}]\!]_\rho = m' \qquad \mathtt{p}(m') = \text{proc } m'(\mathtt{y}_1, ..., \mathtt{y}_{n_2})\{\mathtt{cl}\} \quad \forall_{1\leq n\leq n_1} \mathtt{v}_n = [\![\mathtt{e}_n]\!]_\rho \quad \forall_{n_1<n\leq n_2} \mathtt{v}_n = \text{undefined} \qquad \mathtt{p} \vdash \langle \mathtt{h}, [\mathtt{y}_1 \mapsto \mathtt{v}_1, \cdots, \mathtt{v}_{n_2} \mapsto \mathtt{v}_{n_2}], 0, 0\rangle \Downarrow_{m'} \langle \mathtt{h}', \rho', \mathtt{v}\rangle \quad \mathtt{v} \neq \text{error} \Rightarrow k = i+1 \quad \mathtt{v} = \text{error} \Rightarrow k = j \qquad \mathtt{p} \vdash \langle \mathtt{h}', \rho[\mathtt{x} \mapsto \mathtt{v}], i, k\rangle \Downarrow_m \langle \mathtt{h}'', \rho'', \mathtt{v}'\rangle}{\mathtt{p} \vdash \langle \mathtt{h}, \rho, \_, i\rangle \Downarrow_m \langle \mathtt{h}'', \rho'', \mathtt{v}'\rangle}$$

PHI-ASSIGNMENT

$$\frac{\mathtt{p}_m(j) = \mathtt{x} := \phi(\mathtt{x}_1, ..., \mathtt{x}_n) \quad i \overset{k}{\mapsto}_m j \quad \mathtt{p} \vdash \langle \mathtt{h}, \rho[\mathtt{x} \mapsto \rho(\mathtt{x}_k)], j, j+1\rangle \Downarrow_m \langle \mathtt{h}', \rho', \mathtt{v}\rangle}{\mathtt{p} \vdash \langle \mathtt{h}, \rho, i, j\rangle \Downarrow_m \langle \mathtt{h}', \rho', \mathtt{v}\rangle}$$

B JSIL Logic

JSIL Logic: Logical Values, Logical Expressions, Assertions

$$V \in \mathcal{V}^L_{\text{JSIL}} ::= \mathtt{v} \mid \varnothing \mid \overline{V}$$

$$E \in \mathcal{E}^L_{\text{JSIL}} ::= V \mid \mathtt{x} \mid \mathrm{X} \mid \ominus E \mid E \oplus E \mid \text{typeof}\,(E) \mid \{\{E_1, ..., E_n\}\} \mid \text{nth}\,(E_1, E_2)$$

$$P, Q \in \mathcal{AS}_{\text{JSIL}} ::= \text{true} \mid \text{false} \mid E_1 = E_2 \mid E_1 \leq E_2 \quad \text{Boolean}$$
$$\mid P \wedge Q \mid \neg P \mid \exists \mathtt{x}.P \quad \text{Classical}$$
$$\mid \mathtt{emp} \mid (E, E) \mapsto E \mid \text{emptyFields}(E, E_1, ..., E_n) \mid P * Q \quad \text{JS heaps}$$

Semantics of JSIL Logical Expressions and Assertions

Logical Expressions (Semantics):

$$\begin{array}{lcl}
[\![V]\!]^\epsilon_\rho & \triangleq & V \\
[\![\mathtt{x}]\!]^\epsilon_\rho & \triangleq & \rho(\mathtt{x}) \\
[\![\mathrm{X}]\!]^\epsilon_\rho & \triangleq & \epsilon(\mathrm{X}) \\
[\![\ominus E]\!]^\epsilon_\rho & \triangleq & \overline{\ominus}([\![E]\!]^\epsilon_\rho) \\
[\![E_1 \oplus E_2]\!]^\epsilon_\rho & \triangleq & \overline{\oplus}([\![E_1]\!]^\epsilon_\rho, [\![E_2]\!]^\epsilon_\rho) \\
[\![\text{typeof}\ E]\!]^\epsilon_\rho & \triangleq & \text{TypeOf}([\![E]\!]^\epsilon_\rho) \\
[\![\{\{E_1, ..., E_1\}\}]\!]^\epsilon_\rho & \triangleq & \{\{[\![E_1]\!]^\epsilon_\rho, ..., [\![E_n]\!]^\epsilon_\rho\}\} \\
[\![\text{nth}\,(E_1, E_2)]\!]^\epsilon_\rho & \triangleq & V_k \text{ if } [\![E_1]\!]^\epsilon_\rho = \{\{V_0, ..., V_n\}\},\ [\![E_2]\!]^\epsilon_\rho = k, \text{ and } 0 \leq k \leq n
\end{array}$$

Assertions (Satisfiability Relation):

$$\begin{array}{ll}
H, \rho, \epsilon \models \text{true} & \Leftrightarrow \text{always} \\
H, \rho, \epsilon \models \text{false} & \Leftrightarrow \text{never} \\
H, \rho, \epsilon \models E_1 = E_2 & \Leftrightarrow [\![E_1]\!]^\epsilon_\rho = [\![E_2]\!]^\epsilon_\rho \\
H, \rho, \epsilon \models E_1 \leq E_2 & \Leftrightarrow [\![E_1]\!]^\epsilon_\rho \leq [\![E_2]\!]^\epsilon_\rho \\
H, \rho, \epsilon \models P \wedge Q & \Leftrightarrow H, \rho, \epsilon \models P \wedge Q \text{ and } H, \rho, \epsilon \models Q \\
H, \rho, \epsilon \models \neg P & \Leftrightarrow H, \rho, \epsilon \not\models P \\
H, \rho, \epsilon \models \exists \mathrm{X}.P & \Leftrightarrow \exists V \in \mathcal{V}^L_{\text{JSIL}}.\, H, \rho, \epsilon[\mathrm{X} \mapsto V] \models P \\
H, \rho, \epsilon \models \mathtt{emp} & \Leftrightarrow H = \mathtt{emp} \\
H, \rho, \epsilon \models (E_1, E_2) \mapsto E_3 & \Leftrightarrow H = ([\![E_1]\!]^\epsilon_\rho, [\![E_2]\!]^\epsilon_\rho) \mapsto [\![E_3]\!]^\epsilon_\rho \\
H, \rho, \epsilon \models \text{emptyFields}(E, E_1, ..., E_n) & \Leftrightarrow \forall m \in \mathcal{S}tr.\, m \notin \{[\![E_1]\!]^\epsilon_\rho, ..., [\![E_n]\!]^\epsilon_\rho\} \Rightarrow H([\![E]\!]^\epsilon_\rho, m) = \varnothing \\
H, \rho, \epsilon \models P * Q & \Leftrightarrow \exists H_1, H_2.\, H = H_1 \uplus H_2 \wedge (H_1, \rho, \epsilon \models P) \wedge (H_2, \rho, \epsilon \models Q)
\end{array}$$

▶ (Lemma 2, Soundness of Basic Commands). All basic commands satisfy soundness: for any basic command $\mathtt{bc} \in \mathtt{BCmd}$, abstract heap $H \in \mathcal{H}^{\not\varnothing}$, store $\rho \in \mathcal{S}to$, logical environment $\epsilon \in \mathcal{E}nv$, and assertions $P, Q \in \mathcal{AS}_{\text{JSIL}}$ such that: $\{P\}\,\mathtt{bc}\,\{Q\}$ and $H, \rho, \epsilon \models P$, it holds that:

$$\exists_{H', \rho'}.\, [\![\mathtt{bc}]\!]_{\lfloor H \rfloor, \rho} = (\lfloor H' \rfloor, \rho', -) \ \wedge\ H', \rho', \epsilon \models Q$$

Proof: We proceed by case analysis on the rule applied to get $\{P\}\,\mathtt{bc}\,\{Q\}$. For convenience, we name the hypotheses of the lemma:

- **Hyp.** 1: $\{P\}\,\mathtt{bc}\,\{Q\}$ and
- **Hyp.** 2: $H, \rho, \epsilon \models P$

We need to prove that there is an abstract heap H', a store ρ', and a value $\mathtt{v}$ such that:

- **Goal** 1: $[\![\mathtt{bc}]\!]_{\lfloor H \rfloor, \rho} = (\lfloor H' \rfloor, \rho', \mathtt{v})$
- **Goal** 2: $H', \rho', \epsilon \models Q$

[Skip] It follows that $\mathtt{bc} = \mathsf{skip}$ and, applying **Hyp.** 1, that $P = Q = \mathsf{emp}$. Hence, using **Hyp.** 2, we have that $H = \mathsf{emp}$. We conclude that $H' = \mathsf{emp}$, $\rho' = \rho$, and $\mathtt{v} = \mathsf{empty}$, because $[\![\mathsf{skip}]\!]_{\mathsf{emp},\rho} = (\mathsf{emp}, \rho, \mathsf{empty})$. Finally, noting that $\mathsf{emp}, \rho, \epsilon \models \mathsf{emp}$, we conclude that: $H', \rho', \epsilon \models Q$.

[Assignment] It follows that $\mathtt{bc} = \mathtt{x} := \mathtt{e}$ and, applying **Hyp.** 1, that $Q = P * \mathtt{x} \doteq \mathtt{e}$. Using the semantics of JSIL, we conclude that:

$$(\lfloor H' \rfloor, \rho', \mathtt{v}) = [\![\mathtt{x} := \mathtt{e}]\!]_{\lfloor H \rfloor, \rho} = (\lfloor H \rfloor, \rho[\mathtt{x} \mapsto [\![\mathtt{e}]\!]_\rho], [\![\mathtt{e}]\!]_\rho)$$

Observe that:

$$\begin{array}{l} H, \rho, \epsilon \models P \\ \Rightarrow\ H, \rho[\mathtt{x} \mapsto [\![\mathtt{e}]\!]_\rho], \epsilon \models P \text{ (because } \mathtt{x} \text{ does not occur in } P) \\ \Rightarrow\ H, \rho[\mathtt{x} \mapsto [\![\mathtt{e}]\!]_\rho], \epsilon \models P\ \wedge\ \mathtt{x} = \mathtt{e} \\ \Rightarrow\ H, \rho', \epsilon \models Q \end{array}$$

Hence, if we let $H' = H$, it follows that: $H', \rho', \epsilon \models Q$, which concludes the proof.

[Object Creation] It follows that $\mathtt{bc} = \mathtt{x} := \mathsf{new}\,()$ and, applying **Hyp.** 1, that $P = \mathsf{emp}$ and $Q = (\mathtt{x}, @proto) \mapsto \mathsf{null} * \mathsf{emptyFields}(\mathtt{x}, \{@proto\})$. Hence, using **Hyp.** 2, we have that $H = \mathsf{emp}$. Furthermore, using the semantics of JSIL, we obtain that:

$$(h', \rho', \mathtt{v}) = [\![\mathtt{x} := \mathsf{new}\,()]\!]_{\mathsf{emp},\rho} = ((l, @proto) \mapsto \mathsf{null}, \rho[\mathtt{x} \mapsto l], l)$$

for a given fresh location l. Hence, if we let:

$$H' = \Big(\biguplus_{p \neq @proto} (l, p) \mapsto \varnothing \Big) \uplus (l, @proto) \mapsto \mathsf{null}$$

it follows that $\lfloor H' \rfloor = h'$ and $H', \rho', \epsilon \models Q$, which concludes the proof.

[Property Projection] It follows that $\mathtt{bc} = \mathtt{x} := [\mathtt{e}_1, \mathtt{e}_2]$ and, applying **Hyp.** 1, that $P = (\mathtt{e}_1, \mathtt{e}_2) \mapsto \mathrm{V} * \mathrm{V} \not\doteq \varnothing$ and $Q = P * \mathtt{x} \doteq \mathrm{V}$, for some logical variable V. Hence, using **Hyp.** 2, we have that:

$$H = ([\![\mathtt{e}_1]\!]_\rho, [\![\mathtt{e}_2]\!]_\rho) \mapsto \epsilon(\mathrm{V})$$

Note that $\lfloor H \rfloor = H$; therefore, using the semantics of JSIL, we conclude that:

$$(h', \rho', \mathtt{v}) = [\![\mathtt{x} := [\mathtt{e}_1, \mathtt{e}_2]]\!]_{H,\rho} = (H, \rho[\mathtt{x} \mapsto \epsilon(\mathrm{V})], \epsilon(\mathrm{V}))$$

Hence, letting $H' = H = h'$ and because $\mathtt{x} \notin \mathsf{vars}(\mathtt{e}_1) \cup \mathsf{vars}(\mathtt{e}_2)$, we conclude that $H', \rho', \epsilon \models Q$. More concretely, we know that $H', \rho', \epsilon \models P$ from **Hyp.** 2 and we know $\mathsf{emp}, \rho', \epsilon \models \mathtt{x} \doteq \mathrm{V}$ from the definition of $\models$.

[Property Assignment] It follows that $\mathtt{bc} = [\mathtt{e}_1, \mathtt{e}_2] := \mathtt{e}_3$ and, applying **Hyp.** 1, that $P = (\mathtt{e}_1, \mathtt{e}_2) \mapsto \_$ and $Q = (\mathtt{e}_1, \mathtt{e}_2) \mapsto \mathtt{e}_3$. Hence, using **Hyp.** 2, we have that:

$$H = ([\![\mathtt{e}_1]\!]_\rho, [\![\mathtt{e}_2]\!]_\rho) \mapsto V$$

for some value V. Therefore, using the semantics of JSIL, we conclude that:

$$(h', \rho', \mathtt{v}) = [\![[\mathtt{e}_1, \mathtt{e}_2] := \mathtt{e}_3]\!]_{H,\rho} = ((([\![\mathtt{e}_1]\!]_\rho, [\![\mathtt{e}_2]\!]_\rho) \mapsto [\![\mathtt{e}_3]\!]_\rho, \rho, [\![\mathtt{e}_3]\!]_\rho)$$

Hence, letting $H' = h' = ([\![\mathtt{e}_1]\!]_\rho, [\![\mathtt{e}_2]\!]_\rho) \mapsto [\![\mathtt{e}_3]\!]_\rho$, we conclude that $H', \rho', \epsilon \models Q$.

[Property Deletion - Existing] It follows that $\mathtt{bc} = \mathsf{delete}\,(\mathtt{e}_1, \mathtt{e}_2)$ and, applying **Hyp.** 1, that $P = (\mathtt{e}_1, \mathtt{e}_2) \mapsto V * V \neq \varnothing$ and $Q = (\mathtt{e}_1, \mathtt{e}_2) \mapsto \varnothing$, for some logical variable V. Hence, using **Hyp.** 2, we have that:

$$H = ([\![\mathtt{e}_1]\!]_\rho, [\![\mathtt{e}_2]\!]_\rho) \mapsto \epsilon(\mathrm{V})$$

Note that $\lfloor H \rfloor = H$; therefore, using the semantics of JSIL, we conclude that:

$$(h', \rho', \mathtt{v}) = [\![\mathsf{delete}\,(\mathtt{e}_1, \mathtt{e}_2)]\!]_{H,\rho} = (\mathsf{emp}, \rho[\mathtt{x} \mapsto \mathsf{true}], \mathsf{true})$$

Hence, letting $H' = ([\![\mathtt{e}_1]\!]_\rho, [\![\mathtt{e}_2]\!]_\rho) \mapsto \varnothing$ and recalling that $\mathtt{x} \notin \mathsf{vars}(\mathtt{e}_1) \cup \mathsf{vars}(\mathtt{e}_2)$, we conclude that $H', \rho', \epsilon \models Q$ and $\lfloor H' \rfloor = h'$.

[Member Check - True] It follows that $\mathtt{bc} = \mathtt{x} := \mathsf{hasField}\,(\mathtt{e}_1, \mathtt{e}_2)$ and, applying **Hyp.** 1, that $P = (\mathtt{e}_1, \mathtt{e}_2) \mapsto \mathrm{V} \wedge V \neq \varnothing$ and $Q = P * \mathtt{x} \doteq \mathsf{true}$. Hence, using **Hyp.** 2, we have that:

$$H = ([\![\mathtt{e}_1]\!]_\rho, [\![\mathtt{e}_2]\!]_\rho) \mapsto \epsilon(\mathrm{V})$$

Note that $\lfloor H \rfloor = H$; therefore, using the semantics of JSIL, we conclude that:

$$(h', \rho', \mathtt{v}) = [\![\mathtt{x} := \mathsf{hasField}\,(\mathtt{e}_1, \mathtt{e}_2)]\!]_{H,\rho} = (H, \rho[\mathtt{x} \mapsto \mathsf{true}], \mathsf{true})$$

Hence, letting $H' = H$ and noting that $\mathtt{x} \notin \mathsf{vars}(\mathtt{e}_1) \cup \mathsf{vars}(\mathtt{e}_2)$, we conclude that $H', \rho', \epsilon \models Q$.

The remaining cases follow analogously. □

▶ **Definition 10** (Weak Locality of Basic Commands). A JSIL basic command bc is weakly local with respect to an assertion P if and only if for any assertion Q, abstract heap $H \in \mathcal{H}^{\varnothing}$, store $\rho \in \mathcal{S}to$, and logical environment $\epsilon \in \mathcal{E}nv$, such that $\{P\}\,\mathtt{bc}\,\{Q\}$ and $H, \rho, \epsilon \models P$ the following two properties hold:

- *Frame Property: for all $\hat{H}, \hat{H}_f, \rho_f$, and* v*, it holds that:*

$$[\![\mathtt{bc}]\!]_{\lfloor H \uplus \hat{H} \rfloor, \rho} = (\lfloor \hat{H}_f \rfloor, \rho_f, \mathtt{v}) \implies \exists H_f.\ [\![\mathtt{bc}]\!]_{\lfloor H \rfloor, \rho} = (\lfloor H_f \rfloor, \rho_f, \mathtt{v}) \wedge \hat{H}_f = H \uplus H_f$$

- *Safety Monotonicity:*

$$\forall h_f, \hat{H}.\ [\![\mathtt{bc}]\!]_{\lfloor H \rfloor, \rho} = (h_f, \rho_f, \mathtt{v}) \implies \exists \hat{h}_f.\ [\![\mathtt{bc}]\!]_{\lfloor H \uplus \hat{H} \rfloor, \rho} = (\hat{h}_f, \rho_f, \mathtt{v})$$

▶ **Lemma 11** (Weak Locality of Basic Commands). *All basic commands satisfy the weak locality property; i.e. they exhibit the safety monotonicity and frame properties.*

Proof: As in the proof of Lemma B, we proceed by case analysis on the rule applied to get $\{P\}\,\mathtt{bc}\,\{Q\}$; the proof is analogous. □

▶ **Lemma 12** (Soundness). *For any derivation $\mathtt{pd} \in \mathcal{D}$, program $\mathtt{p} \in \mathtt{P}$, specification function $\mathtt{S} \in \mathcal{S}tr \rightharpoonup \mathcal{S}pec$, abstract heap $H \in \mathcal{H}^{\varnothing}$, store $\rho \in \mathcal{S}to$, logical environment ϵ, procedure name m, and command label i such that:*

- $\mathtt{pd} \vdash \mathtt{p}, \mathtt{S}$
- $(P, k) \in \mathtt{pd}(m, i)$ *and* $H, \rho, \epsilon \models P$
- $\mathtt{p} \vdash \langle \lfloor H \rfloor, \rho, k, i \rangle \Downarrow_m \langle h_f, \rho_f, \mathtt{v} \rangle$

It follows that there is an abstract heap H_f such that: $\lfloor H_f \rfloor = h_f$ and $H_f, \rho_f, \epsilon \models Q$, where $(Q, -) \in \mathtt{pd}(m, i_{ret})$.

Proof: For convenience we name the hypotheses:

- **Hyp.** 1: $\mathtt{pd} \vdash \mathtt{p}, \mathtt{S}$
- **Hyp.** 2: $(P, k) \in \mathtt{pd}(m, i)$ and $H, \rho, \epsilon \models P$
- **Hyp.** 3: $\mathtt{p} \vdash \langle \lfloor H \rfloor, \rho, k, i \rangle \Downarrow_m \langle h_f, \rho_f, \mathtt{v} \rangle$

We proceed by induction on the derivation of **Hyp.** 3.

[Basic Command] It follows that $\mathtt{p}_m(i) = \mathtt{bc}$ for a given basic command $\mathtt{bc}$. We conclude, using **Hyp.** 3 and the semantics of JSIL, that there is a heap h', a store ρ', and value $\mathtt{v}'$, such that:

$$[\![\mathtt{bc}]\!]_{\lfloor H \rfloor, \rho} = (h', \rho', \mathtt{v}') \qquad \mathtt{p} \vdash \langle h', \rho', i, i+1 \rangle \Downarrow_m \langle h_f, \rho_f, \mathtt{v} \rangle$$

Using **Hyp.** 1 and **Hyp.** 2, we conclude that there are two assertions P and Q and an index k such that: $H, \rho, \epsilon \models P$, $(P, k) \in \mathtt{pd}(m, i)$, $(Q, i) \in \mathtt{pd}(m, i+1)$, and $\mathtt{p}, \mathtt{S}, i \vdash^k_m P \rightsquigarrow Q$. We now need to show that there is an abstract heap H' such that: $H', \rho', \epsilon \models Q$ and $\lfloor H' \rfloor = h'$. We prove that such an abstract heap exists by induction on the derivation of $\mathtt{p}, \mathtt{S}, i \vdash^k_m P \rightsquigarrow Q$. More concretely, given that $[\![\mathtt{bc}]\!]_{\lfloor H \rfloor, \rho} = (h', \rho', \mathtt{v}')$, $H, \rho, \epsilon \models P$, and $\mathtt{p}, \mathtt{S}, i \vdash^k_m P \rightsquigarrow Q$, there must exist an abstract heap H' such that $\lfloor H' \rfloor = h'$ and $H', \rho', \epsilon \models Q$.

- [Basic Command] We conclude that: $\{P\}\, \mathtt{bc}\, \{Q\}$. Applying Lemma B, it follows that there is an abstract heap H' such that $\lfloor H' \rfloor = h'$ and $H', \rho', \epsilon \models Q$.
- [Frame Rule] We conclude that there are three assertions P', Q', and R', such that: $P = P' * R$, $Q = Q' * R$, and $\mathtt{p}, \mathtt{S}, i \vdash^k_m P' \rightsquigarrow Q'$. It follows that there are two abstract heaps H_p and H_r such that $H = H_p \uplus H_r$, $H_p, \rho, \epsilon \models P'$, and $H_r, \rho, \epsilon \models R$. Using the frame property of basic commands (Lemma 11), we conclude that there is a heap h'_p such that $[\![\mathtt{bc}]\!]_{\lfloor H_p \rfloor, \rho} = (h'_p, \rho', \mathtt{v}')$ and $h' = h'_p \uplus \lfloor H_r \rfloor$. Applying the inner induction hypothesis to $[\![\mathtt{bc}]\!]_{\lfloor H_p \rfloor, \rho} = (h'_p, \rho', \mathtt{v}')$, $H_p, \rho, \epsilon \models P'$, and $\mathtt{p}, \mathtt{S}, i \vdash^k_m P' \rightsquigarrow Q'$, we conclude that there is an abstract heap H'_p such that: $H'_p, \rho', \epsilon \models Q'$ and $\lfloor H'_p \rfloor = h'_p$. Recalling that $H_r, \rho, \epsilon \models R$, we conclude that $H'_p \uplus H_r, \rho', \epsilon \models Q * R$.
- [Consequence] We conclude that there are two assertions P' and Q', such that: $P \Rightarrow P'$, $Q' \Rightarrow Q$, and $\mathtt{p}, \mathtt{S}, i \vdash^k_m P' \rightsquigarrow Q'$. Since $P \Rightarrow P'$ and $H, \rho, \epsilon \models P$, it follows that $H, \rho, \epsilon \models P'$. Applying the inner induction hypothesis to $[\![\mathtt{bc}]\!]_{\lfloor H \rfloor, \rho} = (h', \rho', \mathtt{v}')$, $H, \rho, \epsilon \models P'$, and $\mathtt{p}, \mathtt{S}, i \vdash^k_m P' \rightsquigarrow Q'$, we conclude that there is an abstract heap H' such that: $H', \rho', \epsilon \models Q'$ and $\lfloor H' \rfloor = h'$. Since $Q' \Rightarrow Q$ and $H, \rho, \epsilon \models Q'$, we conclude that $H, \rho, \epsilon \models P'$.
- [Elimination] We conclude that there are two assertions P' and Q', such that: $P = \exists \mathrm{X}.\, P'$, $Q = \exists \mathrm{X}.\, Q'$, and $\mathtt{p}, \mathtt{S}, i \vdash^k_m P' \rightsquigarrow Q'$. From $H, \rho, \epsilon \models P$, it follows that there is a value $\mathtt{v}$ such that $H, \rho, \epsilon[\mathrm{X} \mapsto \mathtt{v}] \models P'$. Applying the inner induction hypothesis to $[\![\mathtt{bc}]\!]_{\lfloor H \rfloor, \rho} = (h', \rho', \mathtt{v}')$, $H, \rho, \epsilon[\mathrm{X} \mapsto \mathtt{v}] \models P'$, and $\mathtt{p}, \mathtt{S}, i \vdash^k_m P' \rightsquigarrow Q'$, we conclude that there is an abstract heap H' such that $H', \rho', \epsilon[\mathrm{X} \mapsto \mathtt{v}] \models Q'$ and $\lfloor H' \rfloor = h'$. We can therefore conclude, using the definition of satisfaction relations ($\models$) that there is an abstract heap H' such that: $H', \rho', \epsilon \models Q$.
- [All Other Cases] No other rule may have been applied in the derivation of $\mathtt{p}, \mathtt{S}, i \vdash^k_m P \rightsquigarrow Q$ because $\mathtt{bc}$ is a basic command. Hence, we do not have to analyse those cases.

Having established that there is an abstract heap H' such that $\lfloor H' \rfloor = h'$ and $H', \rho', \epsilon \models Q$, we conclude that there is an abstract heap H' such that $H', \rho, \epsilon \models Q$, $(Q, i) \in \mathtt{pd}(m, i+1)$ and $\mathtt{p} \vdash \langle \lfloor H' \rfloor, \rho', i, i+1 \rangle \Downarrow_m \langle h_f, \rho_f, \mathtt{v} \rangle$. Using **Hyp.** 1, we can apply the induction hypothesis to conclude the claim of the lemma.

[Goto] It follows that $\mathtt{p}_m(i) = \mathsf{goto}\ j$ for a given command index j. We conclude, using **Hyp.** 3 and the semantics of JSIL, that:

$$\mathtt{p} \vdash \langle \lfloor H \rfloor, \rho, i, j \rangle \Downarrow_m \langle h_f, \rho_f, \mathtt{v} \rangle$$

Using **Hyp.** 1 and **Hyp.** 2, we conclude that there are two assertions P and Q such that: $H, \rho, \epsilon \models P$, $(P, k) \in \mathtt{pd}(m, i)$, $(Q, i) \in \mathtt{pd}(m, j)$, and $\mathtt{p}, \mathtt{S}, i \vdash^k_m P \rightsquigarrow Q$. We now need to show that there is an abstract heap H' such that $H', \rho, \epsilon \models Q$ and $\lfloor H' \rfloor = \lfloor H \rfloor$. Like in the previous case, we prove that such an abstract heap exists by induction on the derivation of $\mathtt{p}, \mathtt{S}, i \vdash^k_m P \rightsquigarrow Q$. More concretely, given that $H, \rho, \epsilon \models P$ and $\mathtt{p}, \mathtt{S}, i \vdash^k_m P \rightsquigarrow Q$, there must exist an abstract heap H' such that $\lfloor H' \rfloor = \lfloor H \rfloor$ and $H', \rho, \epsilon \models Q$.

- [Goto] We conclude that: $\mathtt{p}, \mathtt{S}, i \vdash^k_m P \rightsquigarrow P$. By letting $H' = H$, it immediately follows that there exists an abstract heap H' such that $\lfloor H' \rfloor = \lfloor H \rfloor$ and $H', \rho, \epsilon \models Q$.
- [Frame Rule, Consequence, Elimination] These cases are similar to those in the previous case.
- [All Other Cases] No other rule may have been applied in the derivation of $\mathtt{p}, \mathtt{S}, i \vdash^k_m P \rightsquigarrow Q$ because $\mathtt{p}_m(i) = \mathsf{goto}\ j$.

Having established that there is an abstract heap H' such that $\lfloor H' \rfloor = \lfloor H \rfloor$ and $H', \rho, \epsilon \models Q$, we conclude that there is an abstract heap H' such that $H', \rho, \epsilon \models Q$, $(Q, i) \in \mathtt{pd}(m, j)$ and $\mathtt{p} \vdash \langle \lfloor H' \rfloor, \rho, i, j \rangle \Downarrow_m \langle h_f, \rho_f, \mathtt{v} \rangle$. Using **Hyp.** 1, we can apply the induction hypothesis to conclude the claim of the lemma.

[Conditional Goto - True] It follows that $\mathtt{p}_m(i) = \mathsf{goto}\ [\mathtt{e}]\ j_1,\ j_2$ for two command indexes command label j_1 and j_2 and a JSIL expression $\mathtt{e}$. We conclude, using **Hyp.** 3 and the semantics of JSIL, that:

$$[\![\mathtt{e}]\!]_\rho = \mathsf{true} \qquad \mathtt{p} \vdash \langle \lfloor H \rfloor, \rho, i, j_1 \rangle \Downarrow_m \langle h_f, \rho_f, \mathtt{v} \rangle$$

Using **Hyp.** 1 and **Hyp.** 2, we conclude that there are two assertions P and Q such that: $H, \rho, \epsilon \models P$, $(P, k) \in \mathtt{pd}(m, i)$, $(Q, i) \in \mathtt{pd}(m, j_1)$, and $\mathtt{p}, \mathtt{S}, i \vdash^k_m P \ \wedge\ \mathtt{e} = \mathsf{true} \rightsquigarrow Q$. Since $\mathtt{e}$ does not contain any logical variables, it follows that: $[\![\mathtt{e}]\!]_\rho = [\![\mathtt{e}]\!]_\rho \epsilon$. Therefore, we conclude that $[\![\mathtt{e}]\!]_\rho \epsilon = \mathsf{true}$, which, together with $H, \rho, \epsilon \models P$, implies that $H, \rho, \epsilon \models P \ \wedge\ \mathtt{e} = \mathsf{true}$. We now need to show that there is an abstract heap H' such that $H', \rho, \epsilon \models Q$ and $\lfloor H' \rfloor = \lfloor H \rfloor$. We prove that such an abstract heap exists by induction on the derivation of $\mathtt{p}, \mathtt{S}, i \vdash_m P \wedge \mathtt{e} = \mathsf{true} \rightsquigarrow Q$. More concretely, given that $H, \rho, \epsilon \models P$ and $\mathtt{p}, \mathtt{S}, i \vdash^k_m P \rightsquigarrow Q$, there must exist an abstract heap H' such that $\lfloor H' \rfloor = \lfloor H \rfloor$ and $H', \rho, \epsilon \models Q$.

- [Goto] We conclude that: $\mathtt{p}, \mathtt{S}, i \vdash^k_m P \rightsquigarrow P$. By letting $H' = H$, it immediately follows that there exists an abstract heap H' such that $\lfloor H' \rfloor = \lfloor H \rfloor$ and $H', \rho, \epsilon \models Q$.
- [Frame Rule, Consequence, Elimination] These cases are similar to those in the previous case.
- [All Other Cases] No other rule may have been applied in the derivation of $\mathtt{p}, \mathtt{S}, i \vdash^k_m P \rightsquigarrow Q$ because $\mathtt{p}_m(i) = \mathsf{goto}\ [\mathtt{e}]\ j,\ k$.

□

▶ **Lemma 13** (Fault-Avoidance). *For any derivation* $\mathtt{pd} \in \mathcal{D}$*, program* $\mathtt{p} \in \mathtt{P}$*, specification function* $\mathtt{S} \in \mathcal{S}tr \rightharpoonup \mathcal{S}pec$*, abstract heap* $H \in \mathcal{H}^{\not\emptyset}$*, store* $\rho \in \mathcal{S}to$*, logical environment* ϵ*, procedure name* m*, and command label* i *such that:*

- $\mathtt{pd} \vdash \mathtt{p}, \mathtt{S}$
- $(P, k) \in \mathtt{pd}(m, i)$ *and* $H, \rho, \epsilon \models P$

It follows that there is no JSIL heap $\mathtt{h}$ *and store* ρ' *such that* $\mathtt{p} \vdash \langle \lfloor H \rfloor, \rho, -, 0 \rangle \Downarrow_m \langle \mathtt{h}, \rho', \mathsf{error} \rangle)$.

Proof: Analogous to Lemma 12. □

▶ (Theorem 6 - JSIL logic soundness). For any JSIL program $\mathtt{p}$ and specification environment $\mathtt{S}$, if there exists a proof candidate $\mathtt{pd} \in \mathcal{D}$ such that $\mathtt{p}, \mathtt{S} \vdash \mathtt{pd}$, then $\mathtt{S}$ is valid for $\mathtt{p}$.

Proof: Suppose that $\{P\}\, m(\overline{\mathtt{x}})\, \{Q\} \in \mathtt{S}_m$, $H, \rho, \epsilon \models P$, and $\mathtt{p}, \mathtt{S} \vdash \mathtt{pd}$. Then it follows that:

- Using Lemma 13, that there is no JSIL heap $\mathtt{h}$ and store ρ' such that $\mathtt{p} \vdash \langle \lfloor H \rfloor, \rho, -, 0 \rangle \Downarrow_m \langle \mathtt{h}, \rho', \mathsf{error} \rangle)$.
- Suppose that: $\mathtt{p} \vdash \langle \lfloor H \rfloor, \rho, -, 0 \rangle \Downarrow_m \langle \mathtt{h}_f, \rho_f, \mathtt{v} \rangle$ for some JSIL heap $\mathtt{h}_f$, store ρ_f and value $\mathtt{v}$. Then, using Lemma 12, we conclude that there exists an abstract heap H_f, such that $\lfloor H_f \rfloor = \mathtt{h}_f$ and $H_f, \rho_f, \epsilon \models Q$, which concludes the proof.

□

C JS-2-JSIL- Logic

JS Logic: Logical Values, Logical Expressions, Assertions

$$V \in \mathcal{V}^L_{\text{JS}} ::= \mathtt{v} \mid \varnothing \mid \overline{V}$$

$$E \in \mathcal{E}^L_{\text{JS}} ::= V \mid \mathtt{x} \mid \mathrm{X} \mid \ominus\ E \mid E \oplus E \mid \mathsf{typeof}\ (E) \mid \{\{E_1, ..., E_n\}\} \mid \mathsf{nth}\ (E_1, E_2) \mid \mathsf{this}$$

$$\begin{array}{lll} P, Q \in \mathcal{AS}_{\text{JS}} ::= \mathsf{true} \mid \mathsf{false} \mid E_1 = E_2 \mid E_1 \leq E_2 & & \text{Boolean} \\ \mid P \wedge Q \mid \neg P \mid \exists \mathtt{x}.P & & \text{Classical} \\ \mid \mathtt{emp} \mid (E, E) \mapsto E \mid \mathsf{emptyFields}(E, E_1, ..., E_n) \mid P * Q & & \text{Heaps} \\ \mid \mathtt{scope}(x : E) \mid \mathsf{funObj}(m', E) & & \text{Scoping and Fun. Objs.} \end{array}$$

Semantics of JS Logical Expressions and Assertions

Logical Expressions (Semantics):

$$\begin{array}{lcl} [\![V]\!]^{\epsilon}_{\rho,l_t} & \triangleq & V \\ [\![\mathtt{x}]\!]^{\epsilon}_{\rho,l_t} & \triangleq & \rho(\mathtt{x}) \\ [\![\mathrm{X}]\!]^{\epsilon}_{\rho,l_t} & \triangleq & \epsilon(\mathrm{X}) \\ [\![\ominus\ E]\!]^{\epsilon}_{\rho,l_t} & \triangleq & \overline{\ominus}([\![E]\!]^{\epsilon}_{\rho,l_t}) \\ [\![E_1 \oplus E_2]\!]^{\epsilon}_{\rho,l_t} & \triangleq & \overline{\oplus}([\![E_1]\!]^{\epsilon}_{\rho,l_t}, [\![E_2]\!]^{\epsilon}_{\rho,l_t}) \\ [\![\mathsf{typeof}\ E]\!]^{\epsilon}_{\rho,l_t} & \triangleq & \mathsf{TypeOf}([\![E]\!]^{\epsilon}_{\rho,l_t}) \\ [\![\{\{E_1, ..., E_1\}\}]\!]^{\epsilon}_{\rho,l_t} & \triangleq & \{\{[\![E_1]\!]^{\epsilon}_{\rho,l_t}, ..., [\![E_n]\!]^{\epsilon}_{\rho,l_t}\}\} \\ [\![\mathsf{nth}\ (E_1, E_2)]\!]^{\epsilon}_{\rho,l_t} & \triangleq & V_k \text{ if } [\![E_1]\!]^{\epsilon}_{\rho,l_t} = \{\{V_0, ..., V_n\}\},\ [\![E_2]\!]^{\epsilon}_{\rho,l_t} = k, \text{ and } 0 \leq k \leq n \\ [\![\mathsf{this}]\!]^{\epsilon}_{\rho,l_t} & \triangleq & l_t \end{array}$$

Assertions (Satisfiability Relation):

$$\begin{array}{ll} H, \rho, L, l_t, \epsilon \models \mathsf{true} & \Leftrightarrow \text{always} \\ H, \rho, L, l_t, \epsilon \models \mathsf{false} & \Leftrightarrow \text{never} \\ H, \rho, L, l_t, \epsilon \models E_1 = E_2 & \Leftrightarrow [\![E_1]\!]^{\epsilon}_{\rho,l_t} = [\![E_2]\!]^{\epsilon}_{\rho,l_t} \\ H, \rho, L, l_t, \epsilon \models E_1 \leq E_2 & \Leftrightarrow [\![E_1]\!]^{\epsilon}_{\rho,l_t} \leq [\![E_2]\!]^{\epsilon}_{\rho,l_t} \\ H, \rho, L, l_t, \epsilon \models P \wedge Q & \Leftrightarrow H, \rho, L, l_t, \epsilon \models P \wedge Q \text{ and } H, \rho, L, l_t, \epsilon \models Q \\ H, \rho, L, l_t, \epsilon \models \neg P & \Leftrightarrow H, \rho, L, l_t, \epsilon \not\models P \\ H, \rho, L, l_t, \epsilon \models \exists \mathrm{X}.P & \Leftrightarrow \exists V \in \mathcal{V}^L_{\text{JSIL}}.\ H, \rho, L, l_t, \epsilon[\mathrm{X} \mapsto V] \models P \\ H, \rho, L, l_t, \epsilon \models \mathtt{emp} & \Leftrightarrow H = \mathsf{emp} \\ H, \rho, L, l_t, \epsilon \models (E_1, E_2) \mapsto E_3 & \Leftrightarrow H = ([\![E_1]\!]^{\epsilon}_{\rho,l_t}, [\![E_2]\!]^{\epsilon}_{\rho,l_t}) \mapsto [\![E_3, l_t]\!]^{\epsilon}_{\rho} \\ H, \rho, L, l_t, \epsilon \models \mathsf{emptyFields}(E, E_1, ..., E_n) & \Leftrightarrow \forall m \in \mathcal{S}tr.\ m \notin \{[\![E_1]\!]^{\epsilon}_{\rho,l_t}, ..., [\![E_n]\!]^{\epsilon}_{\rho,l_t}\} \\ & \quad \implies H([\![E]\!]^{\epsilon}_{\rho,l_t}, m) = \varnothing \\ H, \rho, L, l_t, \epsilon \models P * Q & \Leftrightarrow \exists H_1, H_2.\ H = H_1 \uplus H_2 \wedge (H_1, \rho, L, l_t, \epsilon \models P) \\ & \quad \wedge (H_2, \rho, L, l_t, \epsilon \models Q) \\ H, \rho, L, l_t, \epsilon \models \mathtt{scope}(x : E) & \Leftrightarrow (\psi(m, x) = \mathsf{global} \wedge H = ((l_g, x) \mapsto desc) \\ & \quad \vee (\psi(m, x) = m' \neq \mathsf{global} \wedge H = ((L(m'), x) \mapsto [\![E]\!]^{\epsilon}_{\rho,l_t}) \\ & \quad \text{for } desc = \{\{\text{“}d\text{”}, [\![E]\!]^{\epsilon}_{\rho,l_t}, \mathsf{true}, \mathsf{true}, \mathsf{false}\}\} \\ H, \rho, L, l_t, \epsilon \models \mathsf{funObj}(m', E) & \Leftrightarrow l_f = [\![E]\!]^{\epsilon}_{\rho,l_t} \wedge \mathsf{ids}(L) = \psi^+(m') \\ & \quad H = ((l_f, @code) \mapsto s(m')) \uplus ((l_f, @scope) \mapsto L) \end{array}$$

Translation from JS Logical Expressions/Assertions to JSIL Logical Expressions/Assertions

Logical Environments : $\mathcal{T}_\epsilon : \mathcal{E}nv_{\text{JS}} \rightharpoonup \mathcal{E}nv_{\text{JSIL}}$

$$\mathcal{T}_\epsilon(\epsilon) = \{(\mathrm{X}, \mathcal{T}_v(V)) \mid (\mathrm{X}, V) \in \epsilon\}$$

Logical Expressions : $\mathcal{T}_e : \mathcal{E}^L_{\text{JS}} \rightharpoonup \mathcal{E}^L_{\text{JSIL}}$

$$\mathcal{T}_e(V) = V \qquad \mathcal{T}_e(\mathtt{x}) = \mathtt{x} \qquad \mathcal{T}_e(\mathrm{X}) = \mathrm{X}$$

$$\frac{\mathcal{T}_e(E) = E'}{\mathcal{T}_e(\ominus E) \triangleq E'}$$

$$\frac{\mathcal{T}_e(E_1) = E_1' \quad \mathcal{T}_e(E_2) = E_2'}{\mathcal{T}_e(E_1 \oplus E_2) \triangleq E_1' \oplus E_2'}$$

$$\frac{\mathcal{T}_e(E) = E'}{\mathcal{T}_e(\mathsf{typeof}\ E) \triangleq \mathsf{typeof}\ E'}$$

$$\frac{\mathcal{T}_e(E_1) = E_1' \ \cdots \ \mathcal{T}_e(E_n) = E_n'}{\mathcal{T}_e(\{\{E_1, ..., E_n\}\}) \triangleq \{\{E_1', ..., E_n'\}\}}$$

$$\frac{\mathcal{T}_e(E_1) = E_1' \qquad \mathcal{T}_e(E_2) = E_2'}{\mathcal{T}_e(\mathsf{nth}\ (E_1, E_2)) \triangleq \mathsf{nth}\ (E_1', E_2')}$$

Assertions : $\mathcal{T}_a : \mathcal{AS}_{\text{JS}} \rightharpoonup \mathcal{AS}_{\text{JSIL}}$

$$\mathcal{T}_a(\text{true}) = \text{true} \qquad \mathcal{T}_a(\text{false}) = \text{false} \qquad \frac{\mathcal{T}_a(P) = P'}{\mathcal{T}_a(\neg P) \triangleq \neg P'}$$

$$\frac{\mathcal{T}_a(P) = P' \quad \mathcal{T}_a(Q) = Q'}{\mathcal{T}_a(P \wedge Q) \triangleq P' \wedge Q'} \qquad \frac{\mathcal{T}_a(P) = P'}{\mathcal{T}_a(\exists X.P) \triangleq \exists X.P'}$$

$$\frac{\mathcal{T}_e(E_1) = E_1' \quad \mathcal{T}_e(E_2) = E_2'}{\mathcal{T}_a(E_1 = E_2) \triangleq E_1' = E_2'}$$

$$\frac{\mathcal{T}_e(E_1) = E_1' \quad \mathcal{T}_e(E_2) = E_2'}{\mathcal{T}_a(E_1 \leq E_2) \triangleq E_1' \leq E_2'} \qquad \mathcal{T}_a(\mathsf{emp}) = \mathsf{emp}$$

$$\frac{\mathcal{T}_a(P) = P' \quad \mathcal{T}_a(Q) = Q'}{\mathcal{T}_a(P * Q) \triangleq P' * Q'}$$

$$\frac{\psi_m(x) = m' \neq \mathsf{global} \qquad \mathcal{T}(E) = E'}{\mathcal{T}_a(\mathsf{scope}(x : E)) \triangleq (X_{m'}, x) \mapsto E'}$$

$$\frac{\psi_m(x) = \mathsf{global} \qquad \mathcal{T}(E) = E'}{\mathcal{T}_a(\mathsf{scope}(x : E)) \triangleq (l_g, x) \mapsto \{\{\text{“}d\text{”}, E', \mathsf{true}, \mathsf{true}, \mathsf{false}\}\}}$$

$$\frac{\begin{array}{c}\mathcal{T}(E) = E' \qquad \psi^+(m') = [\mathsf{main}, m_1, ..., m_n] \\ P_1 = ((X_{sc}, m_1) \mapsto X_1) * ... * ((X_{sc}, m_n) \mapsto X_n) \\ P_2 = (E', @code) \mapsto m' * (E', @scope) \mapsto X_{sc}\end{array}}{\begin{array}{c}\mathcal{T}_a(\mathsf{funObj}(m', E)) \triangleq \\ \exists X_{sc}, X_1, ..., X_n . ((X_{sc}, \mathsf{main}) \mapsto l_g) * P_1 * P_2\end{array}}$$

Specifications

$$\frac{\begin{array}{c}\mathcal{T}(P) = P' \qquad \mathcal{T}(Q) = Q' \qquad \psi^+(m) = [\mathsf{main}, m_1, ..., m_n] \\ P'' = (\mathtt{x}_{sc}, \mathsf{main}) \mapsto l_g * (\mathtt{x}_{sc}, m_1) \mapsto Y_{m_1} * ... * (\mathtt{x}_{sc}, m_n) \mapsto Y_{m_n}\end{array}}{\mathcal{T}(\{P\}\, m(\overline{\mathtt{x}})\, \{Q\}) \triangleq \{P'' * P'\}\, m(\mathtt{x}_{sc}, \mathtt{x}_{this}, \overline{\mathtt{x}})\, \{P'' * Q'\}}$$

▶ **Lemma 14** (Translation of Logical Expressions - Correctness). *For any two variable stores ρ and ρ', logical environments ϵ and ϵ', location l_t, and function $\beta : \mathcal{L} \rightharpoonup \mathcal{L}$, such that: $\rho \sim_\beta \rho'$, $\epsilon \sim_\beta \epsilon'$, $\rho'(\mathtt{x}_{this}) = l_t$, it holds that: $[\![E]\!]^\epsilon_{\rho, l_t} \sim_\beta [\![\mathcal{T}_e(E)]\!]^{\epsilon'}_{\rho'}$.*

▶ (Theorem 7 - Assertion translation correctness). For any two JS and JSIL execution contexts $H_{\text{JS}}, \rho, L, l_t$ and H_{JSIL}, ρ', such that $H_{\text{JS}}, \rho, L, l_t \simeq_\beta H_{\text{JSIL}}, \rho'$ and $\epsilon \sim_\beta \epsilon'$, it holds that: $H_{\text{JS}}, \rho, L, l_t, \epsilon \models P$ if and only if $H_{\text{JSIL}}, \rho', \epsilon'[X_{m_1} \mapsto \beta(L(m_1)), ..., X_{m_n} \mapsto \beta(L(m_n))] \models \mathcal{T}(P)$, where $\mathsf{ids}(L) = \{m_1, ..., m_n\}$.

Proof: We proceed by induction on the structure of P.

1. $P = \text{true}$, $P = \text{false}$, or $P = \mathsf{emp}$. Directly from the definitions of satisfiability and the translation by setting $H_{\text{JSIL}} = H_{\text{JSIL}} = \mathsf{emp}$

2. $P = \mathsf{scope}(x : E)$. There are two cases to consider: either $\psi_m(x) = \mathsf{global}$ or $\psi_m(x) = \mathsf{global}$.
   - If $\psi_m(x) = \mathsf{global}$, we conclude that:

a. $H_{\mathtt{JS}} = ((l_g, x) \mapsto \{\{\text{“}d\text{”}, [\![E]\!]^{\epsilon}_{\rho,l_t}, \mathsf{true}, \mathsf{true}, \mathsf{false}\}\})$, thus following that:
$H_{\mathtt{JSIL}} = ((l_g, x) \mapsto \{\{\text{“}d\text{”}, [\![\mathcal{T}(E)]\!]^{\epsilon'}_{\rho'}, \mathsf{true}, \mathsf{true}, \mathsf{false}\}\})$.

b. $\mathcal{T}(P) = (l_g, x) \mapsto \{\{\text{“}d\text{”}, E', \mathsf{true}, \mathsf{true}, \mathsf{false}\}\}$

Applying Lemma 14, we conclude that $[\![E]\!]^{\epsilon}_{\rho,l_t} \sim_{\beta} [\![\mathcal{T}(E)]\!]^{\epsilon'}_{\rho'}$. From which we conclude the result follows.

- If $\psi_m(x) = m' \neq \mathsf{global}$, we conclude that:
  a. $H_{\mathtt{JS}} = ((L(m'), x) \mapsto [\![E]\!]^{\epsilon}_{\rho,l_t})$ thus following that $H_{\mathtt{JSIL}} = ((\beta(L(m')), x) \mapsto [\![\mathcal{T}(E)]\!]^{\epsilon'}_{\rho'})$.
  b. $\mathcal{T}(P) = (l_g, x) \mapsto (X_{m'}, x) \mapsto E'$, from which the result follows immediately.

3. $P = \mathsf{funObj}(m', E)$. We conclude that:
- $H_{\mathtt{JS}} = ((l_f, @code) \mapsto s(m')) \uplus ((l_f, @scope) \mapsto L$, thus following that $H_{\mathtt{JSIL}} = ((\beta(l_f), @code) \mapsto m') \uplus ((\beta(l_f), @scope) \mapsto l_{sc} \uplus \uplus \mathtt{SC}_{\beta}(L))$.
- $P' = \exists_{X_{sc}, X_1, \ldots, X_n} ((X_{sc}, m_1) \mapsto X_1) * \ldots * ((X_{sc}, m_n) \mapsto X_n) * ((X_{sc}, \mathsf{main}) \mapsto l_g * (E', @code) \mapsto m' * (E', @scope) \mapsto X_{sc}$

The result follows immediately from the above.

The remaining cases follow by simple application of the induction hypothesis. □

▶ (Theorem 8 - JS-2-JSIL Logic correspondence). Given a correct JS-2-JSIL compiler $\mathcal{C}$, for every JS statement s and JS assertions P and Q: $\{P\}\, s\, \{Q\}$ iff $\mathcal{T}(\{P\}\, s\, \{Q\})$ is a valid specification environment for $\mathcal{C}(s)$.

Proof: Immediate corollary of Theorem 7. □

▶ (Theorem 9 - Proving JS Hoare Triples in JSIL). For every JS statement s and JS assertions P and Q: $(\exists \mathsf{pd} \cdot \mathcal{C}(s), \mathcal{T}(\{P\}\, s\, \{Q\}) \vdash \mathsf{pd}) \Rightarrow \{P\}\, s\, \{Q\}$

Proof: Immediate corollary of Theorems 6 and 8. □