An Infrastructure for Tractable Verification of JavaScript Programs

ANONYMOUS AUTHOR(S)

The dynamic nature of JavaScript, together with its complex semantics, makes it a difficult target for logic-based 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 JSIL Logic, a program logic for JSIL; and **(3)** verified JSIL Logic 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.

ACM Reference format:
Anonymous Author(s). 2017. An Infrastructure for Tractable Verification of JavaScript Programs. *PACM Progr. Lang.* 1, 1, Article 1 (January 2017), 42 pages.
DOI: 10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

JavaScript is the de facto language for programming client-side web applications. It is described by the international ECMAScript standard (ECMAScript Committee 2011), 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. Such tools have recently become tractable, mainly for C and Java, with compositional techniques that scale and properly engineered tools applied to real-world code. They typically target intermediate goto representations in order to dismantle the control flow constructs of their target language, simplifying their analyses. These tools include: Infer, Facebook's verification tool based on separation logic for finding bugs in C, Java and Objective-C (Calcagno et al. 2015); Java Pathfinder, a model-checking tool for Java bytecode programs (Visser et al. 2004); CBMC, a bounded model checker for C developed at Oxford and Amazon (Kroening and Tautschnig 2014); and WALA's analysis for Java using the Rosette symbolic analyser (Fink and Dolby 2015). These representations, however, are not suitable for dynamic languages such as JavaScript, which require extensible objects, dynamic fields and dynamic function calls. 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 based on separation logic and built on top of this infrastructure.

Our verification infrastructure consists of: **(1)** JS-2-JSIL, a logic-preserving compiler from JavaScript to our simple intermediate goto language JSIL; **(2)** JSIL Verify, a semi-automatic JSIL verification tool based on JSIL Logic, a sound program logic for JSIL; and **(3)** JSIL axiomatic specifications of the JavaScript internal functions, with

A note.

2017. 2475-1421/2017/1-ART1 $15.00
DOI: 10.1145/nnnnnnn.nnnnnnn

JSIL reference implementations verified using JSIL Verify. We demonstrate the feasibility of our infrastructure by developing JaVerT, the JavaScript Verification Toolchain (Figure 1), on top of it. In JaVerT, we create abstractions for the most important JavaScript concepts, such as prototype inheritance, scoping, and function closures, which streamline our reasoning and allow the developer to write readable specifications free of JavaScript-specific clutter. We use JaVerT to verify, for example, small critical Node.js libraries for common data structures, demonstrating our reasoning methodology using a JavaScript implementation of a priority queue.

JS-2-JSIL Compiler. Our JS-2-JSIL compiler from JavaScript to our intermediate goto language, JSIL, targets the strict mode of the ES5 English standard (ES5 Strict). ES5 Strict is a restricted variant of ES5 that intentionally has slightly different semantics compared with ES5, exhibiting 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 (Flanagan 2011), and is widely used by major industrial players: e.g. Google's V8 engine (Google 2017) and Facebook's React library (Facebook 2017). We believe that ES5 Strict is the correct starting point for our project on JavaScript verification. We cover a substantial part of ES5 Strict: the entire kernel plus all built-in libraries intertwined with the kernel. In doing so, we do not simplify the memory model of JavaScript in any way.

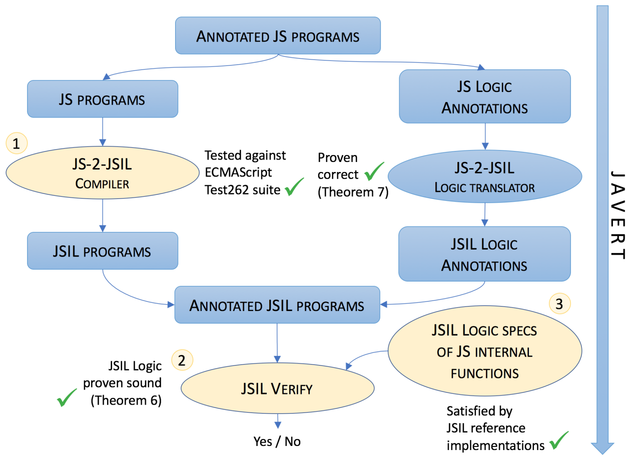

Annotated JS programs
JS programs
JS Logic Annotations
1
JS-2-JSIL Compiler
Tested against ECMAScript Test262 suite ✓
Proven correct ✓ (Theorem 7)
JS-2-JSIL Logic translator
JSIL programs
JSIL Logic Annotations
Annotated JSIL programs
3
JSIL Logic specs of JS internal functions
JSIL Logic proven sound (Theorem 6) ✓
2
JSIL Verify
Yes / No
Satisfied by JSIL reference implementations ✓
JAVERT

Fig. 1. JaVerT: JavaScript Verification Toolchain

Validation. There exists a direct correspondence between the lines of the ES5 standard and the compiled JSIL code. Moreover, we maintain a step-by-step connection between lines of the JS-2-JSIL code itself and lines of the standard as much as possible. We systematically test JS-2-JSIL against the official ECMAScript test suite, passing 100% of the appropriate tests. In a separate technical appendix (JaVerT Team 2017), we give a formal definition and correctness result for part of the compiler, [1] adapting standard techniques from compiler design literature (Barthe et al. 2005; Fournet et al. 2009) to the dynamic setting of JavaScript.

JSIL Verify. We introduce JSIL Verify, a semi-automatic JSIL verification tool based on JSIL Logic, a sound separation logic for JSIL. JSIL Verify contains a symbolic execution engine and an entailment engine, which uses the Z3 SMT solver (De Moura and Bjørner 2008) to discharge assertions in first-order logic with equality and arithmetic, while we handle the separation-logic assertions ourselves. To verify JSIL programs, we provide pre- and postconditions for functions, loop invariants and fold/unfold directives for user-defined predicates. We use JSIL Verify directly to verify our JSIL Logic specifications of the JavaScript internal functions against their JSIL reference implementations. This provides substantial initial validation for JSIL Verify, as we have verified more than 100 non-trivial specifications of the internal functions corresponding to approximately 1K lines of JSIL code.

JSIL Logic Specifications of JavaScript Internal Functions. JavaScript internal functions describe the fundamental inner workings of the language, such as object management and type conversions. They are not accessible by the programmer, but are called internally by all JavaScript commands. Their definitions in the standard are complex, operational, and are often intertwined, making it difficult for the user to fully grasp the control flow and allowed behaviours. In order to reason about JavaScript code, one first has to be able to reason about these internal functions. We provide functionally correct JSIL Logic *axiomatic specifications* of a large subset of JavaScript

[1]The definition and result for the full compiler would require substantial mechanised specification and proof, which is beyond our manpower.

internal functions. In creating these specifications, we leverage on a number of JavaScript-specific abstractions built on top of JSIL Logic, making the specifications more readable than the operational definitions of the standard. The remaining complexity arises from the internal functions themselves, and not our reasoning.

Validation. We obtain a strong guarantee of correctness for our axiomatic specifications by verifying, using JSIL Verify, that they are satisfied by their corresponding JSIL reference implementations. The reference implementations step-by-step follow the standard and are (indirectly) substantially tested tens of thousands of times via our testing of the JS-2-JSIL compiler against the official ECMAScript test suite.

Applications. Our axiomatic specifications of internal functions directly benefit JaVerT, since now the verification of JavaScript code only needs to use the specifications, not the underlying implementations. We believe that they can also have benefits outside JaVerT. For instance, from the axiomatic specifications we could construct executable specifications that could then be used for different types of symbolic analysis for JavaScript and would give us a mechanism to restrict the semantics of JavaScript in a principled way.

JaVerT Verification. Our goal is to verify critical JavaScript code, such as Node.js libraries describing frequently used data structures. We demonstrate JaVerT on a variation of a Node.js priority queue library that uses doubly linked lists (Jones 2016), simplified for exposition to singly-linked lists. The library code is annotated with assertions written in our JS assertion language, JS Logic; the annotations comprise specifications (pre- and postconditions) for library functions, together with loop invariants and unfold/fold instructions for any user-defined predicates, such as the queue predicate. We provide abstractions for the key concepts of JavaScript; in particular, for prototype inheritance, scoping, and function closures. These abstractions allow the developer to write readable, expressive specifications, without having to know the underlying representations of JavaScript semantics. Using the JS-2-JSIL logic translator, we translate the JS annotations to JSIL Logic; at the same time, we compile the JavaScript code to JSIL using JS-2-JSIL. The resulting annotated JSIL program is then automatically verified by JSIL Verify, taking advantage of our verified specifications of JavaScript internal functions.

Validation. We have purposefully designed the JSIL memory model to be as close as possible to the JS memory model so that we can easily relate functional properties of JavaScript programs to those of their compiled JSIL programs. We validate the JS-2-JSIL logic translator and establish a correctness result for the translation of specifications, under the assumption of the correctness of JS-2-JSIL. We also prove a correctness result for JaVerT, allowing us to lift JSIL verification to JavaScript verification.

Application. We have verified the stylised priority queue library that is our running example (see §3), as well as a number of JavaScript programs that show that our treatment of prototype chains, scoping, and function closures is correct. We show how to specify the queue library functions in a way that ensures functionally correct behaviour and encapsulation; this approach is general and can be applied to other libraries. Moreover, due to our abstractions of JavaScript internals, our specifications are of the same complexity as those available for C++ or Java programs. The JavaScript programs that we have verified can be found online at (JaVerT Team 2017).

2 RELATED WORK

This paper brings 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. The existing literature covers a wide range of program analysis techniques for JavaScript, including: type systems (Anderson et al. 2005; Bierman et al. 2014; Feldthaus and Møller 2014; Jensen et al. 2009; Microsoft 2014; Rastogi et al. 2015; Thiemann 2005), control flow analysis (Feldthaus et al. 2013; Park and Ryu 2015), pointer analysis (Jang and Choe 2009; Sridharan et al. 2012) and abstract interpretation (Andreasen and Møller 2014; Jensen et al. 2009; Kashyap et al. 2014), among others. We do not address this work in detail as we aim for a logic-based verification tool for JavaScript. We focus on compilers and intermediate representations (IRs) for JavaScript as well as relevant logic-based verification tools.

Separation-Logic-Based Verification Tools. Separation logic enables modular reasoning about programs which manipulate complex heap structures. It has been successfully used in verification tools for static languages: Smallfoot (Berdine et al. 2005) for a simple imperative while language; jStar (Distefano and Parkinson 2008) for Java; Verifast (Jacobs et al. 2011) for C and Java; Space Invader (Yang et al. 2008) and Abductor (Calcagno et al. 2009) for C; and Infer (Calcagno et al. 2015) for C, Java, Objective C, and C++.

We are developing a separation-logic-based tool for JavaScript. Given the tried-and-tested approach of the above-mentioned tools, our IR, JSIL, is an extremely simple language with only the most basic control flow and object management commands and a memory model as close to JavaScript as possible. Our only aim with JSIL was to be able to efficiently build a verification tool around it and easily state the required correctness results.

Due to the dynamic nature of JavaScript, we could not have compiled ES5 Strict to a language supported by one of the stated tools. 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, we would need to use custom abstractions to reason about them and their associated operations. Moreover, any program logic for JavaScript needs to take into account the JavaScript operators, such as `toInt32` and `toUInt32` (ECMAScript Committee 2011). In JSIL, these operators are supported natively, but it is not clear that they could be expressed using the assertion languages of existing tools.

We draw inspiration for JSIL Logic from the work of Gardner et al. (2012), who developed a separation logic for a small fragment of ES3 (e.g. no implicit coercions, only the `while` control-flow statement) with a simplified memory model (e.g. no property attributes). Nonetheless, their logic is complex, features non-standard separation logic connectives and is not suitable for automation. We improve substantially on this work. We do not simplify the semantics or the memory model of the language and JaVerT is only restricted by the coverage of JS-2-JSIL. We give a semantic definition of the Hoare triples of ES5 Strict which we interpret in JSIL Logic. JSIL Logic is sound, considerably simpler than that of Gardner et al. (2012), and has already been automated as part of JSIL Verify.

KJS. KJS (Ştefănescu et al. 2016; Park et al. 2015) is a tested executable semantics of JavaScript in the $\mathbb{K}$ framework (Roşu and Şerbănuţă 2010). It comes with a symbolic execution engine and can thus be used for formal analysis and verification of JavaScript programs, with specifications written in the reachability logic of $\mathbb{K}$. The authors have used KJS to verify several non-trivial heap-manipulating programs. However, they do not provide abstractions for reasoning about JavaScript scope and prototype chains, i.e. the user of $\mathbb{K}$ has to know the inner workings of the JavaScript semantics to specify and verify JavaScript programs. Moreover, in contrast to JaVerT, which jumps over the implementations of the internal functions by using their axiomatic specifications, KJS always needs to symbolically execute these implementations. This makes it impossible to use KJS to reason about prototype and scope chains whose concrete shape is not known a priori.

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

When it comes to the IRs in **(2)**, JSIL is similar to JSIR, and the IRs of WALA and TAJS. JSIR and WALA do not have associated compilers, which makes it difficult for us to discuss the precise nature of the differences between JSIL and JSIR/WALA, other than saying that JSIL is syntactically simpler. Our choices were determined by wanting JS-2-JSIL to follow closely our ES5 Strict operational semantics; the choices behind JSIR/WALA are not stated. TAJS includes a compiler, originally from ES3, 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 on the developed analyses and not the IR, the authors do not provide detailed information about the translation and their testing infrastructure.

One of our goals in the development of JS-2-JSIL was to cover all the corner cases of ES5 Strict. Thus, a strong connection between the generated JSIL code and the standard was imperative. Our design of JS-2-JSIL builds on the tradition of compilers that closely follow the operational semantics of the source language, such as the ML Kit Compiler (Birkedal et al. 1993). In that spirit, JS-2-JSIL mimics ES5 Strict by inlining in the generated JSIL code the internal steps performed by the ES5 Strict semantics, making them explicit. To achieve this, we based our compiler on the JSCert mechanised specification of ES5 (Bodin et al. 2013). Alternatively, we could have used KJS.

We have considered using S5 of Politz et al. (2012), which targets the full ES5 standard, as an interim stage during compilation. 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. The figure critical for us, the success rate of S5 on full ES5 Strict tests (those testing its unique features *and* the features common with ES5), was not reported. This means that we would have to redo S5 tests using our methodology and then fix the unfamiliar code in light of failing tests. Also, we would have to relate JS Logic and JSIL Logic via S5. This would be a very difficult task.

3 MOTIVATING EXAMPLE: PRIORITY QUEUE

Our motivating example is a priority queue library which uses an implementation based on singly-linked node lists. It is a variation on a Node.js priority queue library that uses doubly linked lists (Jones 2016), simplified for exposition. We use this example to illustrate the complexity of programming in JavaScript and the behavioural properties that we wish to capture using JaVerT. In §7, we use JaVerT to: **(1)** specify and verify all of the functions associated with the library; and **(2)** state conditions under which the library behaves as intended.

Our priority queue library is given in Figure 2 in lines 1-40, with a small client program given in lines 42-46. A priority queue is an object with a property `_head` pointing to a singly-linked list of node objects, with the nodes ordered in descending order of priority. Line 42 constructs a new priority queue, identified by the variable `q`, by calling the `PQ` function, which constructs a new priority queue and initially sets `_head` to `null`. Lines 43-46 call the `enqueue` and `dequeue` functions, manipulating the queue. For example, `q.enqueue(1,"last")` uses the `Node` function to construct a node object, say `n`, with priority (`pri`= 1), value (`val`="last"), and a pointer to the next node (initially `next = null`). It then calls the function `n.insert(nl)` which inserts `n` into an existing node list `nl`, returning the head of the new node list. To illustrate scoping (see §7.3), we keep track of how many nodes were created using the variable `counter` (lines 3,8). Also, we annotate all function literals with unique identifiers.

We would like to abstract the fact that our implementation of queue uses `Node` and instead present a well-known queue interface to the user. In Java, it would be possible to define a `Node` constructor and its associated functionalities to be private. In contrast, JavaScript does not provide a native way of declaring private functions and the standard way to establish some form of encapsulation is by using function closures: for example, the call of the function `Node` inside the body of `enqueue` (line 26) refers to the variable `Node` declared in the enclosing scope. This makes it impossible for the clients of the library to see the `Node` function and use it directly. However, they still can access and modify constructed nodes and `Node.prototype` through the `_head` property of the queue, breaking encapsulation. In §7, we give specifications of the queue library functions that ensure functionally correct behaviour and encapsulation. To achieve this, we must reason about a number of JavaScript concepts, most importantly *prototype inheritance* and *scoping*. In the remainder of this section, we describe these concepts, as well as the initial JavaScript heap and some features of JavaScript objects.

Initial Heap. Before the execution of any JavaScript program, an initial heap is established. It contains the global object, which holds all global variables such as `PriorityQueue`, `q` and `r` from the example. It also contains the functions of all JavaScript built-in libraries. In the example, we use the `Error` built-in function to construct a new error object when trying to dequeue an empty queue (line 32).

Extensible Objects, Dynamic Access, Internal Properties. JavaScript objects, unlike those of Java or C++, are *extensible*, i.e. properties can be added and removed from an object after creation. Moreover, property access

```
1  /* @id PQLib */
2  var PriorityQueue = (function () {
3    var counter = 0;
4
5    /* @id Node */
6    var Node = function (pri, val) {
7      this.pri = pri; this.val = val; this.next = null;
8      counter++;
9    }
10
11   /* @id insert */
12   Node.prototype.insert = function (nl) {
13     if (nl === null) { return this }
14     if (this.pri >= nl.pri) {
15       this.next = nl; return this
16     }
17     var tmp = this.insert (nl.next);
18     nl.next = tmp;
19     return nl
20   }
21
22   /* @id PQ */
23   var PQ = function () { this._head = null };
24   /* @id enqueue */
25   PQ.prototype.enqueue = function(pri, val) {
26     var n = new Node(pri, val);
27     this._head = n.insert(this._head);
28   };
29
30   /* @id dequeue */
31   PQ.prototype.dequeue = function () {
32     if (this._head === null) { throw new Error() }
33     var first = this._head;
34     this._head = this._head.next;
35
36     return {pri: first.pri, val: first.val};
37   };
38
39   return PQ;
40 })();
41
42 var q = new PriorityQueue();
43 q.enqueue(1, "last");
44 q.enqueue(3, "bar");
45 q.enqueue(2, "foo");
46 var r = q.dequeue();
```

Fig. 2. A stylised priority queue library (lines 1-40) and a simple client (lines 42-46).

in JavaScript is dynamic; we cannot guarantee statically which property of the object will be accessed. Finally, JavaScript objects have two types of properties: *internal* and *named*. Internal properties are hidden from the user, but are critical for the mechanisms underlying JavaScript, such as prototype inheritance. Standard JavaScript objects have three internal properties: `@proto`, `@class`, and `@extensible`. For example, all node objects constructed using the `enqueue` function have prototype `Node.prototype`, class `"Object"`, and are extensible.

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

Functions, Function objects. Functions are also stored in the JavaScript heap as objects. 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 those objects created using that function as the constructor. For example, `Node.prototype` is the prototype of all the node objects constructed using the `Node` function, and is the place to find the `insert` function.

Prototype-based inheritance. JavaScript models inheritance through prototype chains. JavaScript semantics looks up the value of a property of an object by first checking the object itself and then walking along the prototype chain, following the `@proto` internal properties. In our example, all node objects constructed using the `enqueue` function (line 26) have the prototype chain like the one given in Figure 3. To look up a property `p` starting from `n`, the semantics first checks if `n` 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`, and then of `Object.prototype`, returning the corresponding value if the property exists somewhere in the chain and `undefined` otherwise.

There are two main requirements necessary for the correct functioning of the priority queue library, which involve a subtle interaction of extensible objects, property descriptors, and prototype-based inheritance. All node objects constructed using the `Node` function should have access to the function `insert`. This means that the node objects themselves must not have the property `"insert"`. Also, we must always be able to construct a `Node` object. This means, due to the semantics of JavaScript, that `Node.Prototype` and `Object.Prototype` must not have properties `"pri"`, `"val"` and `"next"`, used in the node constructor, declared as non-writable. These two requirements, which we call *prototype safety*, are illustrated in Figure 3 by the red properties with value `None`. In §5, we introduce assertions that declare the absence of properties inside objects. Using these assertions, we are able to build abstractions that yield a simple, functionally correct, prototype-safe specification of the priority queue library, where much of the complexity of JavaScript can be hidden from view.

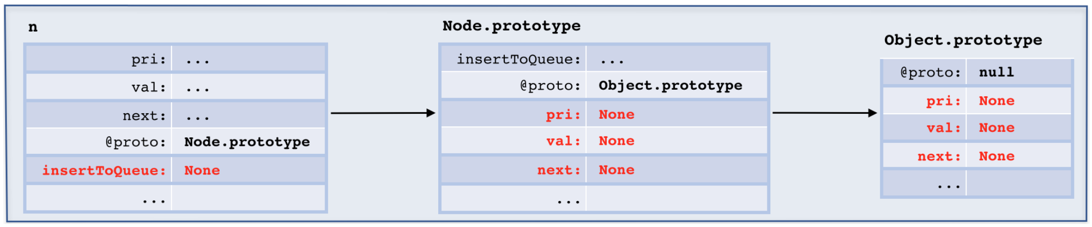


Fig. 3. Prototype safety for Node objects

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

We discuss scoping through the `enqueue` function, which uses four variables: `pri`, `val`, `n`, and `Node` in its body. The scope chain of `enqueue` contains the ERs corresponding to `enqueue`, `PQLib`, and global code. As `pri` and `val` are formal parameters and `n` is a local variable of `enqueue`, they are stored in the ER of `enqueue`. However, `Node` is not declared in `enqueue` and is not its formal parameter, so we have to look for it in the rest of the scope chain associated with `enqueue`, and we find it in the ER corresponding to `PQLib`. This means that when we reason about `enqueue`, we need to capture not only its ER, but also a part of the ER of `PQLib`. This is a difficult task for specification/verification, especially in the presence of multiple function closures that may need to share the same resources. In §7, we introduce abstractions that allow the developer to reason about scope and function closures without having to know any details about the internal representation of scopes in JavaScript.

4 JS-2-JSIL COMPILER

Our JS-2-JSIL compiler targets the strict mode of the ECMAScript 5 English standard (ES5 Strict). In this section, we first describe in detail the coverage of JS-2-JSIL (§4.1). Next, we briefly introduce JSIL, our intermediate language for JavaScript verification (§4.2), and show how compiled JSIL code directly reflects the ES5 standard by translating an assignment from our running example (§4.3). Finally, we discuss our approaches to validating JS-2-JSIL, focussing on extensive testing against the official ECMAScript Test262 test suite, where we have achieved a 100% success rate on the 8797 relevant tests (§4.4).

4.1 Compiler Coverage

We cover a substantial and fully representative part of the ES5 Strict language, 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 ES5 standard can broadly be divided into three parts: the syntax and the parser (chapters 1-7); the kernel of JavaScript, including scoping, internal functions, and language constructs (chapters 8-14); and built-in libraries that provide additional functionalities on top of the kernel (chapter 15). 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. The parser that we use, Esprima (Hidayat 2012), is widely used and is standard-compliant. We implement the entire kernel, except indirect eval, which 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. Additionally, we implement: the core of the Global library, associated with the global object; the constructors and basic functionalities for the String, Number, and Date libraries, together with the functions from those libraries used for testing features of the kernel. We do not implement the orthogonal RegExp and JSON libraries. The implementation of the remaining functionalities amounts to a (lengthy) technical exercise. In §8, we discuss what it means to extend our coverage from ES5 Strict to ES6 Strict as well as to full ES5/ES6.

4.2 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 of the JSIL Language

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

Literals : $\lambda \in \mathcal{L}it$ ::= $n \mid b \mid m \mid \mathsf{undefined} \mid \mathsf{null} \mid l \mid \mathsf{t} \mid \mathsf{empty} \mid \overline{\lambda}$

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

Basic Commands:

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

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

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

Notation : $\overline{\mathsf{x}}$, $\overline{\lambda}$, $\overline{\mathsf{e}}$, and $\overline{\mathsf{c}}$, respectively, denote lists of variables, literals, expressions, and commands.

Most syntactic constructs of JSIL either directly emulate those of JavaScript or are useful for JavaScript analysis. JSIL literals, $\lambda \in \mathcal{L}it$, include strings, numbers, booleans, JavaScript special values undefined and null, object locations l, types t, the special value empty, and lists of literals $\overline{\lambda}$. JSIL expressions, $\mathsf{e} \in \mathcal{E}_{\mathsf{JSIL}}$, include JSIL literals, JSIL variables x, a variety of unary and binary operators, and lists of expressions $\overline{\mathsf{e}}$.

Basic JSIL commands essentially provide the machinery for the management of extensible objects and do not affect control flow. They include skip, variable assignments, object creation, as well as standard 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. The two goto commands are standard: $\mathsf{goto}\ i$ jumps to the i-th command of the active procedure, and $\mathsf{goto}\ [\mathsf{e}]\ i, j$ jumps to the i-th command if e evaluates to true, and to the j-th otherwise. The dynamic procedure call $\mathsf{x} := \mathsf{e}(\overline{\mathsf{e}})\ \mathsf{with}\ j$ first dynamically obtains the procedure name and arguments by evaluating e and $\overline{\mathsf{e}}$, then executes the procedure supplying these arguments, and finally assigns its return value to 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, we have the ϕ-node command $\mathsf{x} := \phi(\mathsf{x}_1, \ldots, \mathsf{x}_n)$, interpreted as follows: there exist n paths via which this command can be reached during the execution of the program; the value assigned to x is x_i *iff*

[2]The Function constructor, much like indirect eval, may exit strict mode; we always execute the provided code in strict mode.

the i-th path was taken. We include ϕ-nodes in JSIL to allow direct support for Static-Single-Assignment (SSA), well-known to simplify analysis (Cytron et al. 1989). Our JS-2-JSIL compiler generates JSIL code directly in SSA.

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

4.3 JS-2-JSIL Compilation by Example

We illustrate compilation 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 introduce JavaScript references and describe how JavaScript scope chains are dealt with in JSIL.

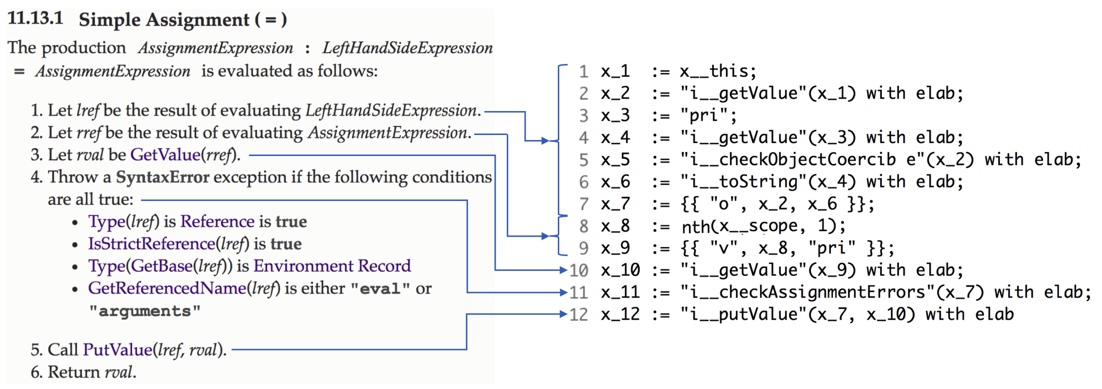
References. References are internals of JavaScript that appear, e.g., as a result of evaluating a left-hand side of an assignment, and represent resolved property bindings. A reference is a pair consisting of a base (normally an object location) and a property name (always a string), telling us where 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 ER (*variable reference*). To obtain the actual associated value, the reference needs to be dereferenced, which is performed by the GetValue internal function. In JSIL, we encode references as lists of three elements, containing the reference type ("o" or "v"), the base, and the property name.

Emulating Scope Lookup in JSIL. Scope chains in ES5 Strict are lists of ER locations, and are modelled as such in JSIL. 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. To take advantage of this, we use a special *closure clarification function* $\psi(m, x)$, which produces the index of the scope chain list corresponding to the ER within which x is defined during the execution of the function m.

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) We first 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 is step-by-step reflected in lines 1-7 of the JSIL code. We omit the details due to lack of space. In this case, the resulting reference is {{ "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 evaluate the right-hand side of the assignment, the variable pri, resolving it to a variable reference. 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 (see Figure 2), it is in the ER corresponding to Node, which is the second element of the scope chain list (line 8) (the first element is always the global object). The appropriate reference, {{ "v", x_8, "pri"}}, is then constructed in line 9. This code is automatically generated using the scope clarification function.



11.13.1 Simple Assignment (=)

The production *AssignmentExpression* : *LeftHandSideExpression* = *AssignmentExpression* is evaluated as follows:

1. Let *lref* be the result of evaluating *LeftHandSideExpression*.
2. Let *rref* be the result of evaluating *AssignmentExpression*.
3. Let *rval* be GetValue(*rref*).
4. Throw a **SyntaxError** exception if the following conditions are all true:
    - Type(*lref*) is Reference is **true**
    - IsStrictReference(*lref*) is **true**
    - Type(GetBase(*lref*)) is Environment Record
    - GetReferencedName(*lref*) is either `"eval"` or `"arguments"`
5. Call PutValue(*lref*, *rval*).
6. Return *rval*.

```
 1 x_1  := x__this;
 2 x_2  := "i__getValue"(x_1) with elab;
 3 x_3  := "pri";
 4 x_4  := "i__getValue"(x_3) with elab;
 5 x_5  := "i__checkObjectCoercib e"(x_2) with elab;
 6 x_6  := "i__toString"(x_4) with elab;
 7 x_7  := {{ "o", x_2, x_6 }};
 8 x_8  := nth(x__scope, 1);
 9 x_9  := {{ "v", x_8, "pri" }};
10 x_10 := "i__getValue"(x_9) with elab;
11 x_11 := "i__checkAssignmentErrors"(x_7) with elab;
12 x_12 := "i__putValue"(x_7, x_10) with elab
```

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

(3) Next, the obtained right-hand side reference is dereferenced using the `GetValue` internal function (§8.7.1 of the ES5 standard). Any call to an internal function gets translated to JSIL as a procedure call to our corresponding reference implementation, in this case `i__getValue` (line 10).

(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).

(6) In JavaScript, every statement returns a value. In JSIL, the compiler, when given a statement to compile, returns the list of corresponding JSIL commands and also variable that stores the return value of that statement. In this example, the compiler would return the presented code and the variable `x_10`.

This example illustrates how close JS-2-JSIL is to the ES5 standard. Out of the 12 lines of compiled JSIL code, 10 have a direct counterpart in the standard. 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 a simple JavaScript assignment: two expression evaluations, one syntactic check, and calls to two different internal functions. What is not visible yet, and will be shown in §6, is the hierarchy of internal functions behind `GetValue` and `PutValue`.

4.4 JS-2-JSIL: Validation

We take great care in validating JS-2-JSIL. As we have shown, there exists a direct correspondence between the lines of the ES5 standard and the compiled JSIL code. Furthermore, we maintain a step-by-step connection between lines of the JS-2-JSIL code itself and lines of the standard as much as possible. In a separate technical appendix (JaVerT Team 2017), we give a formal definition and correctness result for part of the compiler, adapting standard techniques from compiler design literature (Barthe et al. 2005; Fournet et al. 2009) to the dynamic setting of JavaScript. In the remainder of this section, we report in detail on our extensive testing of JS-2-JSIL.

We test JS-2-JSIL against ECMAScript Test262, the official test suite for JavaScript implementations. Currently, it has two available versions: an unmaintained version for ES5 and an actively maintained version for the ES6 standard. ES5 Test262 has poor support for ECMAScript implementations that enforce strict mode, which JSIL does, rendering any kind of systematic effort to target ES5 Strict tests borderline infeasible. This issue has been fully resolved in the ES6 version. 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. exclude all of the tests targeting ES6 features), this is a small price to pay for the overall increase in precision and correctness.

Due to the size of the test suite, it was imperative for us to automate as much of the process as possible. 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 and logs the results. We have also developed an accompanying GUI, which allows us to easily 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 potential regressions. The infrastructure is highly modular and can be reused for a systematic testing of other related projects, such as JSCert or S5.

To run the tests, we set up the compiler runtime, containing the JavaScript initial heap and the JSIL implementations of all JavaScript internal functions and essential functionalities of the built-in libraries. We setup the initial heap in full, including stubs for unimplemented functions (~750 lines of JSIL code). We implement, step-by-step following the English standard, the internal functions (~1K lines) and built-in library functions (~3.5K lines).

We perform the actual testing as follows: first, we compile to JSIL the official harness of ES6 Test262. Then, for each test, we: **(1)** compile its code 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.

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	1297
Compiler Coverage	**9172**
ES5/6 differences in semantics	345
Tests using non-implemented features	30
Applicable Tests	**8797**
Tests passed	**8797**
Tests failed	0

Fig. 5. Detailed testing results

The breakdown of the testing results is presented in Figure 5. The version of the ES6 Test262 test suite used in this study contains 21301 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 browsers, 699 tests), and internationalisation (189 tests), all of which are out of the scope of this project. Next, we exclude tests for ES5 non-strict features (890 tests); these include all tests for indirect eval and some tests for the Function constructor, which both allow the programmer to use non-strict code even when executing in strict mode. We obtain 10469 tests targeting ES5 Strict.

To filter down to the tests that should reflect the coverage of JS-2-JSIL, we remove 1297 tests for unimplemented built-in library functions (e.g. RegExp and JSON libraries). This leaves us with 9172 tests targeting JS-2-JSIL. Not all of these tests, however, are applicable. ES6 has introduced minor changes to the semantics of a few features with respect to ES5, and there are 345 tests that target such features. Also, 30 tests were testing features covered by the compiler by using non-implemented features, and were thus excluded. In the end, we have the final 8797 tests relevant to our JS-2-JSIL compiler, of which we pass 100%. This 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.

5 JSIL VERIFY

JSIL Verify is our semi-automatic verification tool for JSIL programs. In this section, we give a high-level overview of JSIL Verify and JSIL Logic, a sound program logic for JSIL. The full description of our symbolic execution and the detailed proof of the soundness theorem can be found in the Appendix. We have used JSIL Verify directly for the verification of JavaScript internal functions (described in §6) and we use it as part of JaVerT for the verification of compiled JavaScript code (described in §7).

5.1 JSIL Verify

Given a JSIL program annotated with the specifications of its procedures as well as additional proof annotations, JSIL Verify checks whether or not the procedures of the program satisfy their specifications. The high-level architecture of JSIL Verify is given in Figure 6. It consists of: **(1)** a symbolic execution engine that symbolically executes JSIL commands according to the proof rules of JSIL Logic (§5.2) and **(2)** an entailment engine for resolving frame inference and entailment questions, most of which we are able to efficiently handle ourselves; the rest we delegate to the Z3 SMT solver (De Moura and Bjørner 2008).

Recall that JSIL is a language with extensible objects and dynamic field access. This introduces an additional layer of complexity into our symbolic execution and entailment when compared to C++ or Java. For instance, since properties can be added to/deleted from objects after creation, we need to be able to reason explicitly about the absence of fields in an object; we outline our approach to this in §5.2. Also, as we do not necessarily statically know which field of an object we are accessing, we need to use our entailment engine on each property access. Since property accesses are very frequent and calls to Z3 are expensive, we needed to develop an efficient mechanism for resolving property names. Due to lack of space, we will give a detailed account of these and other technical details behind JSIL Verify in a separate future publication.

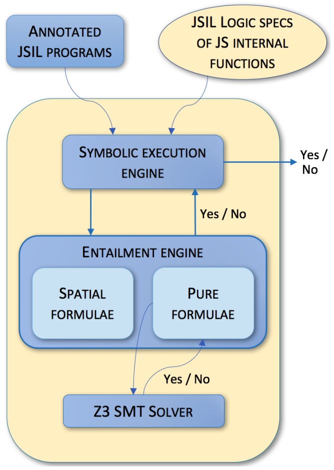


Fig. 6. Architecture of JSIL Verify

5.2 JSIL Logic

JSIL Operational Semantics. Here, we introduce only the JSIL semantic judgement for programs, needed to state our soundness result. The full account of JSIL semantics can be found in the Appendix. A JSIL variable store, $\rho \in \mathcal{S}to$, is a mapping from JSIL variables to JSIL values, and a JSIL heap, $\mathsf{h} \in \mathcal{H}_{\text{JSIL}}$, is a mapping from pairs of locations and property names to JSIL values. JSIL values, $\mathsf{v} \in \mathcal{V}_{\text{JSIL}}$, coincide with JSIL literals. The semantics of JSIL programs is described usingthe judgement $\mathsf{p} \vdash \langle \mathsf{h}, \rho, j, i \rangle \Downarrow_m \langle \mathsf{h}', \rho', \mathsf{v} \rangle$, meaning that the evaluation of procedure m of program p, starting from its i-th command, to which we have arrived from its j-th command, in the heap h and store ρ, generates the heap h', the store ρ', and returns the value v.

JSIL Assertions. JSIL assertions are composed of JSIL logical expressions, $E \in \mathcal{E}^L$, and include standard boolean assertions, except disjunction; the separating conjunction; existential quantification; and assertions for describing JSIL heaps. The $\mathtt{emp}$ assertion describes an empty heap; 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 that has the value denoted by E_3. The assertion $\mathsf{emptyFields}(E \mid E_1, ..., E_n)$ describes an object that has no fields other than possibly those denoted by $E_1, ..., E_n$.

$$\begin{array}{rcl} P, Q \in \mathcal{AS}_{\text{JSIL}} & ::= & \mathsf{true} \mid \mathsf{false} \mid E = E \mid E \leq E \mid P \land Q \mid \neg P \mid P * Q \mid \exists X.P \mid \\ & & \mathsf{emp} \mid (E, E) \mapsto E \mid \mathsf{emptyFields}(E \mid E_1, ..., E_n) \end{array} \quad (1)$$

JSIL logical expressions, $E \in \mathcal{E}^L$, are essentially JSIL expressions extended with logical variables X and the special value $\varnothing$ (read: *none*), used to denote the absence of a field in an object. We write $(l, x) \mapsto \varnothing$ to state that the object at location l has no field named x. JSIL logical values, $V \in \mathcal{V}^L_{\text{JSIL}}$, coincide with JSIL values together with the special logic value $\varnothing$.

An assertion may be satisfied by a JSIL logical context (H, ρ, ϵ), consisting of an *abstract heap* H; a JSIL store ρ, mapping JSIL variables to JSIL literals; and a logical environment ϵ, mapping JSIL logical variables to logical values. An abstract heap (Gardner et al. 2012) maps pairs of locations and field names to logical values. Formally, $H \in \mathcal{H}^{\emptyset}_{\text{JSIL}} : \mathcal{L} \times \mathcal{S}tr \rightharpoonup \mathcal{V}^L_{\text{JSIL}}$. We give the satisfiability relation for JSIL assertions in the Appendix.

JSIL Specifications. JSIL procedures can be annotated with specifications of the form $\{P\}\, m(\overline{\mathsf{x}})\, \{Q\}$, where m is the procedure name, $\overline{\mathsf{x}}$ the formal parameters of the procedure, and P and Q the pre- and postconditions of the procedure. Furthermore, each specification needs to be associated with a return mode $\mathit{fl} \in \{\mathsf{nm}, \mathsf{er}\}$, stating if the procedure returns in normal (nm) or in error mode (er). Formally, we define a *specification environment*, $\mathsf{S} : \mathcal{S}tr \rightharpoonup \mathcal{F}lag \rightharpoonup \mathcal{S}pec$, mapping procedure names and return modes onto specifications. To avoid clutter, we assume in the formalisation that each procedure has a single specification per return mode. Hence, if

$\mathsf{S}(m, \mathit{fl}) = S \in \mathcal{S}pec$, then S is the specification of m for the return mode fl. We say that a JSIL specification $\{P\}\, m(\overline{\mathsf{x}})\, \{Q\}$ is *valid* w.r.t. the return mode fl, if whenever m is executed in a state satisfying P, then, if it terminates, it will do so in a state satisfying Q with return mode fl. A specification environment is said to be *valid* for a JSIL program p, if all the specifications in its range are valid with respect to their corresponding return modes.

JSIL Symbolic Execution. Our goal is to use JSIL Logic to prove the specifications of JSIL procedures. JSIL basic commands can be given local axioms of the form $\{P\}\mathsf{bc}\{Q\}$. However, since JSIL programs contain goto commands, we cannot rely on the standard sequencing rule of Hoare logic to derive the specifications of sequences of JSIL commands. 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 $\mathsf{p}, \mathsf{S}, i, k \vdash^{j}_{m,\mathit{fl}} P \rightsquigarrow Q$, where S is the specification environment, P and Q are the pre- and postconditions of the i-th command of procedure m in program p, in the symbolic execution that reaches the i-th command of m from its j-th command and that then proceeds to the k-th command of m. Transitions are additionally annotated with the return mode fl associated with the specification that is currently being verified.

In order to give both axiomatic and operational semantics to the ϕ-assignment, we need to have the predecessors of each command ordered. We use $j \overset{l}{\mapsto}_m i$ to denote that the j-th command of m is the l-th predecessor of the i-th command of m (we omit l when it is not relevant). The preconditions of each command are computed from the postconditions of its predecessors using the transition relation. Figure 7 gives several rules for the symbolic execution of JSIL control flow commands (see §4.2 for the definition of ϕ-nodes).

$$\textsc{Basic Command}\quad \frac{\mathsf{p}_m(i) = \mathsf{bc} \qquad \{P\}\,\mathsf{bc}\,\{Q\}}{\mathsf{p}, \mathsf{S}, i, i+1 \vdash^{j}_{m,\mathit{fl}} P \rightsquigarrow Q} \qquad \textsc{Conditional Goto – True}\quad \frac{\mathsf{p}_m(i) = \mathsf{goto}\ [\mathsf{e}]\ k_1,\ k_2}{\mathsf{p}, \mathsf{S}, i, k_1 \vdash^{j}_{m,\mathit{fl}} P \rightsquigarrow P * \mathsf{e} \doteq \mathsf{true}}$$

$$\textsc{Phi-Assignment}\quad \frac{\mathsf{p}_m(i) = \mathsf{x} := \phi(\mathsf{x}_1, \ldots, \mathsf{x}_n) \qquad j \overset{l}{\mapsto}_m i}{\mathsf{p}, \mathsf{S}, i, i+1 \vdash^{j}_{m,\mathit{fl}} P \rightsquigarrow P * (\mathsf{x} \doteq \mathsf{x}_l)}$$

Fig. 7. Symbolic Execution of Control Flow Commands: $\mathsf{p}, \mathsf{S}, i, k \vdash^{j}_{m,\mathit{fl}} P \rightsquigarrow Q$

To establish the validity of specifications we construct *proof candidates*. As we do not support disjunction, we opt for a polyvariant analysis (Barthe and Rezk 2005). A proof candidate, $\mathsf{pd} \in \mathcal{D} : \mathcal{S}tr \times \mathcal{F}lag \times \mathbb{N} \rightharpoonup \wp(\mathcal{AS}_{\mathsf{JSIL}} \times \mathbb{N})$, maps each command in a procedure to a set of possible preconditions, associating each such precondition with the index of the command that led to it. To illustrate, if $(P, j) \in \mathsf{pd}(m, \mathit{fl}, i)$, then we have that, in the symbolic execution of m with return mode fl, P is the precondition of the i-th command of m that resulted from the symbolic execution of the j-th command of m. A proof candidate is then said to be *well-formed* iff **(1)** the set of preconditions of the first command of every procedure is the singleton set containing the precondition of the procedure itself and **(2)** all the postconditions of every command are included in the set of preconditions of its predecessors. Formally, given a program $\mathsf{p} \in \mathsf{P}$ and a specification environment S, a proof candidate pd is *well-formed* with respect to p and S, written $\mathsf{p}, \mathsf{S} \vdash \mathsf{pd}$, iff for all procedures m in p, and index i the following hold:

(1) $\mathsf{S}(m, \mathit{fl}) = \{P\}m(\overline{\mathsf{x}})\{Q\} \iff \mathsf{pd}(m, \mathit{fl}, 0) = \{(P, 0)\}$

(2) $(P, j) \in \mathsf{pd}(m, \mathit{fl}, i) \ \wedge (P \nvDash \mathsf{false}) \Rightarrow \big(\forall k.\, i \mapsto_m k \Rightarrow \exists Q.\, (Q, i) \in \mathsf{pd}(m, \mathit{fl}, k) \ \wedge \ \mathsf{p}, \mathsf{S}, i, k \vdash^{j}_{m,\mathit{fl}} P \rightsquigarrow Q\big)$

A specification environment S is said to be satisfied by a program p if there is a proof candidate pd, such that $\mathsf{p}, \mathsf{S} \vdash \mathsf{pd}$. JSIL symbolic execution rules are sound with respect to the JSIL operational semantics, meaning that if a specification environment S is satisfied by p, then S is valid for p (Theorem 5.1). We use $\lfloor H \rfloor$ to denote the concrete heap obtained by restricting the abstract heap H to the elements of its domain not mapped to $\varnothing$.

Theorem 5.1 (JSIL logic soundness). *For every JSIL specification environment* S*, program* p*, abstract heap* H*, store* ρ*, logical environment* ϵ*, procedure names* m*, and return flag* fl*, such that:* **(1)** S *is satisfied by* p, **(2)** $\mathsf{S}(m, \mathsf{fl}) = \{P\}\, m(\overline{\mathsf{x}})\, \{Q\}$, **(3)** $H, \rho, \epsilon \models P$*, it holds that:*

$$\forall \mathsf{h}_f, \rho_f, o.\, \mathsf{p} \vdash \langle \lfloor H \rfloor, \rho, -, 0 \rangle \Downarrow_m \langle \mathsf{h}_f, \rho_f, o \rangle \implies \exists H_f, \mathsf{v}.\ \lfloor H_f \rfloor = \mathsf{h}_f \ \wedge \ H_f, \rho_f, \epsilon \models Q \ \wedge \ o = \mathsf{fl}\langle \mathsf{v} \rangle$$

6 JSIL LOGIC SPECIFICATIONS OF JAVASCRIPT INTERNAL FUNCTIONS

JavaScript internal functions describe the building blocks of the language, including prototype chain traversal, object management, and type conversions. They are called by all JavaScript commands, and in order to be able to reason about JavaScript code, we have to first be able to reason efficiently about the internal functions. We achieve this by creating a number of crucial abstractions that hide the internals of JavaScript and, using these abstractions, give *axiomatic specifications* of a large subset of the internal functions. We verify, using JSIL Verify, that these specifications satisfy their JSIL reference implementations, and discuss how they benefit JaVerT and, possibly, other types of JavaScript analysis.

Axiomatic specifications. The definitions of the internal functions in the standard are operational, complex, and intertwined, making the allowed behaviours difficult to discern. Illustratively, in Figure 8 we show the call graphs of `GetValue` and `PutValue`, the two main internal functions that operate on references. We provide functionally correct JSIL Logic *axiomatic specifications* of a large subset of the internal functions, explicitly exposing all allowed behaviours. In creating these specifications, we leverage on a number of JavaScript-specific abstract predicates built on top of JSIL logic that make our specifications more readable than the operational definitions of the standard and serve to hide JavaScript internals from the developer. For example, `standardObject` hides internal properties, `dataField` abstracts over property descriptors, and `Pi` fully captures prototype inheritance.[3] They also allow us to coalesce multiple execution scenarios into one specification, greatly reducing the number of specifications that we need to write. Illustratively, although `GetProperty`, which models prototype chain traversal, has only one specification, during verification the predicates in its precondition get unfolded to seven different cases, each of which needs to be verified separately. For the functions shown in Figure 8, which form the core of JavaScript's object management, we have more than 100 specifications that get unfolded during verification to more than 300. Without our abstractions, specifying internal functions would not be feasible.

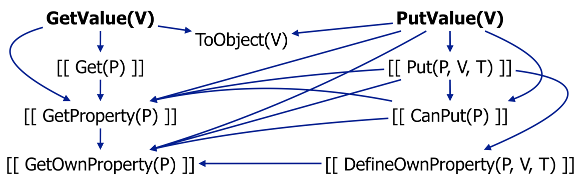


Fig. 8. Call graphs for GetValue and PutValue

Specification by Example: `GetValue`. `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"`, `val`, `prop` }}. If `val` does not denote an object, it is first converted to one using `ToObject`. Then, `GetValue` returns the value associated with the property `prop` of that object. If `v` is a variable reference whose base is not the global object, this value is obtained by directly inspecting the heap. Otherwise, `GetValue` traverses the prototype chain and obtains the appropriate value. Below, we show the specification 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, _, _, _) * DataDescriptor(desc) * descVal(desc, w)  }
                                        GetValue(v)
                                  { Precondition * (ret = w) }
```

In the precondition, we require an object reference `v`, pointing to a property `prop` of object `l`. 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 and returns `w`. This specification is simple, readable, and captures a behaviour that operationally involves calling three internal functions.

[3]We will discuss some of these predicates in more detail in §7, when we show the specification of our running example.

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

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

7 JAVERT

We use JaVerT to verify JavaScript programs specified using JS Logic assertions. We present the theory that underpins JaVerT: our JS Logic assertion language, focussing on the scoping abstractions (§7.1), and the correctness results for our JS-2-JSIL logic translator and JaVerT (§7.2). Finally, we specify the running example (§7.3). JaVerT is available online (JaVerT Team 2017), where it can be tested on the running example and other JavaScript programs.

7.1 JS Logic

We adapt the assertion language introduced in (Gardner et al. 2012) to allow for automation and extend it to target full ES5 Strict heaps and reason about ES5 Strict scope. The ES5 Strict scope assertions allow the user to reason about ES5 Strict scope without knowing the underlying details of how scope is represented in the JavaScript heap.

JS Assertions. JS logical expressions comprise JSIL logical expressions, together with the special logical expressions this and $\underline{1}$, respectively denoting the current this object and scope chain. JS assertions comprise JSIL assertions and: **(1)** an assertion $\mathsf{scope}(m, x : E_1, E_2)$ stating that the variable x is bound to the value denoted by E_1 in the scope chain denoted by E_2, corresponding to the function with identifier m, **(2)** an assertion $\mathsf{funObj}(m, E_1, E_2)$ describing the function object at the location denoted by E_1, representing the syntactic function with identifier m, with the scope chain denoted by E_2, and **(3)** an assertion $\mathsf{o\text{-}chains}(m_1 : E_1, m_2 : E_2)$ stating that E_1 denotes a scope chain of the function with identifier m_1 and E_2 a scope chain of the function with identifier m_2 and that these two scope chains maximally overlap. Finally, we use $\mathsf{scope}(x : E)$ as short for $\mathsf{scope}(m, x : E, \underline{1})$, where m denotes the identifier in which the assertion occurs.

The satisfiability relation for JS assertions has the form: $H, \rho, L, l_t, \epsilon \models_{s,m,\psi,\psi^o} P$, 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 of the this identifier, and **(5)** ϵ is the logical environment. In the following, we use the term *JavaScript logical context* to refer to tuples of the form $(H, \rho, L, l_t, \epsilon)$ and the term *execution context* when ϵ is not given. The satisfiability relation takes four additional parameters, which we generally leave implicit: **(1)** the 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 overlapping scope function ψ^o, which takes two function identifiers and returns $k - 1$, where k is the length of the overlap of their scope chains. We give the satisfiability relation for the special assertions; the remaining cases are in the Appendix. In the following, we use $L(i)$ to denote the i-th element of the list L and $s(m)$ to refer to the lambda abstraction corresponding to the function labelled with m in s.

Satisfiability relation for JS assertions: $H, \rho, L, l_t, \epsilon \models_{s,m,\psi,\psi^o} P$

Scope Assertion

$$H, \rho, L, l_t, \epsilon \models \mathsf{scope}(m', x : E_1, E_2) \iff v = [\![E_1]\!]^{\epsilon}_{\rho,l_t,L} \land L' = [\![E_2]\!]^{\epsilon}_{\rho,l_t,L} \land \psi(m', x) = i \land$$
$$(i = 0 \land L'(0) = l_g \land H = ((l_g, x) \mapsto \{\{\text{“}d\text{”}, v, \mathsf{true}, \mathsf{true}, \mathsf{false}\}\})) \lor (i \neq 0 \land H = ((L'(i), x) \mapsto v))$$

Function Object Assertion

$$H, \rho, L, l_t, \epsilon \models \mathsf{funObj}(m', E_1, E_2) \iff l_f = [\![E_1]\!]^{\epsilon}_{\rho,l_t,L} \land L' = [\![E_2]\!]^{\epsilon}_{\rho,l_t,L} \land \psi^o(m, m') = k \land \bigwedge_{0 \leq i \leq k} L(i) = L'(i) \land$$
$$H = ((l_f, @code) \mapsto s(m')) \uplus ((l_f, @scope) \mapsto L')$$

O-Chains Assertion

$$H, \rho, L, l_t, \epsilon \models \mathsf{o\text{-}chains}(m_1 : E_1, m_2 : E_2) \iff k = \psi^o(m_1, m_2) \land \forall_{0 \leq i \leq k} \left([\![E_1]\!]^{\epsilon}_{\rho,l_t,L}(i) = [\![E_2]\!]^{\epsilon}_{\rho,l_t,L}(i)\right)$$

The scope assertion $\mathsf{scope}(m, x : E_1, E_2)$ uses the scope clarification function to determine the identifier of the function in which x is defined when looked-up from within the code of m'. If x is a global variable (in which case it is found in the first element of the scope chain list), then it is associated with the appropriate data descriptor; otherwise, it is directly associated with its value. The function object assertion $\mathsf{funObj}(m', E_1, E_2)$ means that at the location denoted by E_1 there exists a function object whose identifier is m' and whose scope chain is denoted by E_2. It also explicitly captures the overlap between the scope chain of m' and the current scope chain. Finally, the assertion $\mathsf{o\text{-}chains}(m_1 : E_1, m_2 : E_2)$ states that E_1 and E_2 respectively denote scope chains of functions with identifiers m_1 and m_2, and that these two scope chains coincide in their overlap computed using ψ^o.

JS Specifications. Our goal is to use JaVerT to verify JavaScript programs annotated with JS Logic assertions. To this end, the function literals occurring in the JavaScript programs given to JaVerT can be annotated with specifications of the form $\{P\}\, m(\overline{x})\, \{Q\}$. As for JSIL, each specification is associated with a return mode $fl \in \{\mathsf{nm}, \mathsf{er}\}$. A JavaScript function specification $\{P\}\, m(\overline{x})\, \{Q\}$ for mode fl is valid w.r.t. its corresponding function if “whenever the function is executed in a state satisfying P, then, if it terminates, it does so in a state satisfying Q with return mode fl”. As before, we define a *JavaScript specification environment*, $\mathsf{S}_{\mathsf{JS}} : \mathcal{S}tr \rightharpoonup \mathcal{F}lag \rightharpoonup \mathcal{S}pec_{\mathsf{JS}}$, mapping function identifiers and return modes onto specifications and we assume, in the formalism, a single function specification per return mode. A JavaScript specification environment is *valid* w.r.t a given JavaScript program s (Definition 7.1) if and only if all the specifications in its co-domain are valid w.r.t. their corresponding functions. Given an annotated JavaScript program s, we use $\mathsf{specs}(s)$ to refer to the specification environment of s, i.e. the specification environment containing the specifications of all the function literals declared in s. To avoid clutter, we assume that the top level specification is associated with the special identifier main. In the following, we make use of an ES5 Strict semantic relation of the form $s, L, l_t \vdash \langle h, \rho \rangle \Downarrow_m \langle h_f, o \rangle$, meaning that, in the context of a program s, scope chain list L and the this identifier l_t, when executing the function of s whose identifier is m and whose parameter values are given by ρ in the heap h, one obtains the final heap h_f and outcome o.

Definition 7.1 (Valid JS specification environment). A JS specification environment $\mathsf{S}_{\mathsf{JS}} \in \mathcal{S}tr \rightharpoonup \mathcal{F}lag \rightharpoonup \mathcal{S}pec_{\mathsf{JS}}$ is said to be valid for a given program s if and only if for all function identifiers m, return modes fl, assertions P and Q, JS abstract heaps $H \in \mathcal{H}^{\emptyset}_{\mathsf{JS}}$, stores $\rho \in \mathcal{S}to$, scope chain lists L, locations l_t, and logical environments ϵ, such that $\mathsf{S}_{\mathsf{JS}}(m, fl) = \{P\}\, m(\overline{x})\, \{Q\}$ and $H, \rho, L, l_t, \epsilon \models P$, it holds that:

$$\forall h_f, o.\ s, L, l_t \vdash \langle \lfloor H \rfloor, \rho \rangle \Downarrow_m \langle h_f, o \rangle \implies \exists H_f, v.\ \lfloor H_f \rfloor = h_f \land H_f, \rho, L, l_t, \epsilon \models Q \land o = fl\langle v \rangle$$

7.2 JS-2-JSIL: Logic Translator

Translation of Assertions. Our goal is to use JSIL Verify to verify JavaScript specifications. To this end, we define a translation from JavaScript specifications to JSIL specifications and JavaScript programs to JSIL programs. As the JSIL memory model is very close to the JavaScript memory model, the translation of assertions is straightforward with the exception of the scope-related assertions, which is formally given below (the full translation can be found in the Appendix). The translation is implicitly parameterised with a function identifier m, the *scope clarification function* ψ, and the overlapping scope function ψ^o, described above.

Translation of Assertions and Specifications

Scope Assertion - Non-global

$$\frac{\psi(m,x) = i \neq 0 \quad \mathcal{T}(E_1) = E_1' \quad \mathcal{T}(E_2) = E_2'}{\mathcal{T}(\mathsf{scope}(m, x : E_1, E_2)) \triangleq (\mathsf{nth}\,(E_1', i), x) \mapsto E_2'}$$

Scope Assertion - global

$$\frac{\psi(m,x) = 0 \quad \mathcal{T}(E_1) = E_1' \quad \mathcal{T}(E_2) = E_2' \quad P' = \mathsf{nth}\,(E_2', 0) \doteq l_g}{\mathcal{T}(\mathsf{scope}(m, x : E_1, E_2)) \triangleq (l_g, x) \mapsto \{\{\text{“}d\text{”}, E', \mathsf{true}, \mathsf{true}, \mathsf{false}\}\} * P'}$$

Function Object Assertion

$$\frac{\psi^o(m,m') = k \quad \mathcal{T}(E_1) = E_1' \quad \mathcal{T}(E_2) = E_2'}{\mathcal{T}(\mathsf{funObj}(m', E_1, E_2)) \triangleq (E', @code) \mapsto m' * (E', @scope) \mapsto E_2' * \circledast_{0 \leq i \leq k}(\mathsf{nth}\,(\mathsf{x}_{sc}, i) \doteq \mathsf{nth}\,(E_2', i))}$$

O-Chains Assertion

$$\frac{\psi^o(m_1,m_2) = k \quad \mathcal{T}(E_1) = E_1' \quad \mathcal{T}(E_2) = E_2'}{\mathcal{T}(\mathsf{o\text{-}chains}(m_1 : E_1, m_2 : E_2)) \triangleq \circledast_{0 \leq i \leq k}(\mathsf{nth}\,(E_1', i) \doteq \mathsf{nth}\,(E_2', i))}$$

Function Specification

$$\frac{\mathcal{T}(P) = P' \quad \mathcal{T}(Q) = Q'}{\mathcal{T}(\{P\}\, m(\overline{\mathsf{x}})\, \{Q\}) \triangleq \{P'\}\, m(\mathsf{x}_{sc}, \mathsf{x}_{this}, \overline{\mathsf{x}})\, \{Q'\}}$$

The translation of JavaScript assertions closely follows their semantics. To translate a procedure specification, the formal parameters of the procedure are extended with the x_{sc} and x_{this} variables (respectively denoting the scope chain list and the this object), whereas the pre- and postconditions are translated using the translation for assertions. We translate a JavaScript specification environment to a JSIL specification environment by applying the translation of function specifications to all the elements in its co-domain. Put formally, $\mathcal{T}(\mathsf{S}_{\mathsf{JS}}) \triangleq \mathsf{S}_{\mathsf{JSIL}}$ if and only if $\mathsf{dom}(\mathsf{S}_{\mathsf{JS}}) = \mathsf{dom}(\mathsf{S}_{\mathsf{JSIL}})$ and for all $(m, fl) \in \mathsf{dom}(\mathsf{S}_{\mathsf{JS}})$, it holds that: $\mathsf{S}_{\mathsf{JSIL}}(m, fl) = \mathcal{T}(\mathsf{S}_{\mathsf{JS}}(m, fl))$.

Correctness of the Translation of Assertions. In order to establish the correctness of the translation of assertions, we need to relate JavaScript logical contexts with JSIL logical contexts. To this end, we need to precisely relate JavaScript heaps and values with JSIL heaps and values. JavaScript objects in the JavaScript heap correspond to JSIL objects in the JSIL heap. However, as object allocators are by default non-deterministic, we cannot require their locations to precisely coincide. To remedy this, we use a partial injective function $\beta : \mathcal{L} \rightharpoonup \mathcal{L}$ to allow for the renaming of object locations. Informally, $h \sim_\beta \mathsf{h}$ means that h and h coincide up to object locations and lambda abstractions. The object locations in h are mapped to the locations in h given by β and the function abstractions in h are mapped to their corresponding identifiers in h. The same holds for values $v \sim_\beta \mathsf{v}$. Furthermore, 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, respectively. Finally, Theorem 7.3 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 $\mathcal{T}(P)$.

Definition 7.2 (β-Correspondence for Logical Contexts). A JS logical context $(H_{\mathrm{JS}}, \rho, L, l_t, \epsilon)$ is said to be β-equal to a JSIL logical context $(H_{\mathrm{JSIL}}, \rho', \epsilon')$, written $H_{\mathrm{JS}}, \rho, L, l_t, \epsilon \simeq_\beta H_{\mathrm{JSIL}}, \rho', \epsilon'$ iff there is a JSIL store ρ'' such that: **(1)** $H_{\mathrm{JS}} \sim_\beta H_{\mathrm{JSIL}}$, **(2)** $\rho' = \rho'' \uplus [\mathsf{x}_{sc} \mapsto \beta(L), \mathsf{x}_{\mathsf{this}} \mapsto \beta(l_t)]$, **(3)** $\rho \sim_\beta \rho''$, and **(4)** $\epsilon \sim_\beta \epsilon'$.

Theorem 7.3 (Assertion translation correctness). *For any two JavaScript and JSIL logical contexts $(H_{\mathrm{JS}}, \rho, L, l_t, \epsilon)$ and $(H_{\mathrm{JSIL}}, \rho', \epsilon')$, such that $H_{\mathrm{JS}}, \rho, L, l_t, \epsilon \simeq_\beta H_{\mathrm{JSIL}}, \rho', \epsilon'$, it holds that: $H_{\mathrm{JS}}, \rho, L, l_t, \epsilon \models P$ iff $H_{\mathrm{JSIL}}, \rho', \epsilon' \models \mathcal{T}(P)$.*

Correctness of JaVerT. 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 verify ES5 Strict specification environments by appealing to JSIL logic. Theorem 7.4 establishes the correctness of the translation of specification environments, in that it states that a given JavaScript specification environment is valid for a program s *if and only if* the translation of the specification environment is valid for the compilation of s generated by a *correct* compiler. Finally, Theorem 7.5 shows how we can use JSIL logic together with the logic translator and a correct JS-2-JSIL compiler to verify JavaScript programs.

Theorem 7.4 (JS-2-JSIL Logic correspondence). *Given a correct JS-2-JSIL compiler C, for every annotated JS program s:* specs(s) *is valid for s iff* $\mathcal{T}$(specs(s)) *is valid for* $C(s)$.

Theorem 7.5 (Proving JS Hoare Triples in JSIL). *For every annotated JS program s, if there is a JSIL logic proof derivation* pd *such that* $C(s), \mathcal{T}(\text{specs}(s)) \vdash$ pd *for a correct JS-2-JSIL compiler C, then* specs(s) *is valid for s.*

The compiler correctness criterion is given in the Appendix. Informally, we say that a JS-2-JSIL compiler is *correct* if: "whenever a ES5 Strict program and its compilation are evaluated in two contexts 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 $\sim_\beta$".

7.3 Specification of the running example

We illustrate how to use JaVerT to verify JavaScript programs by specifying the functions given in the example. 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.

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:

```
    Node(n, pri, val, next, np) := ObjectWithProto(n, np) * dataProp(n, "pri", pri) *
      dataProp(n, "val", val) * dataProp(n, "next", next) * (n, "insert") -> None * typeOf(pri) = Num
```

A node n is a JavaScript object whose prototype is np, and which has three data properties: "pri", "val", and "next", with values pri, val, and next. The value pri is a number, as it represents the priority; we do not restrict the type of val. The definition of Node uses two built-in predicates: ObjectWithProto and dataProp. ObjectWithProto(o, p) states that object o has prototype p, while its internal properties @extensible and @class have their default values, true and "Object". The built-in predicate assertion dataProp(o, p, v) denotes that the property p of object o holds a data descriptor with value v and all other attributes set to true. Finally, we require that n has no property "insert". By doing so, we fulfil part of the first requirement for prototype safety (cf. §3).

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" to null. Finally, each time a new node is created, the variable counter is incremented by 1. The specification of Node is:

```
{ ObjectWithProto(this, np) * (this, "pri") -> None * (this, "val") -> None * (this, "next") -> None * pri >= 0 }
{   * (this, "insert") -> None * NodeProto(np) * typeOf(pri) = Num * scope(counter: c) * typeOf(c) = Num      }
                                          Node(pri, val)
                      { Node(this, pri, val, null, np) * NodeProto(np) * scope(counter: c+1) }
```

The precondition of Node states: **(1)** that the keyword this must be initially bound to a JavaScript object whose prototype is np; **(2)** that the this object must not have the properties "pri", "val", "next", and "insert"; **(3)** that np is a valid node prototype, 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 counter c are of number 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 np; **(2)** np is still a valid node prototype; and **(3)** the variable counter is incremented by 1.

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, fulfilling the second part of the first requirement for prototype safety. The second requirement states that the node prototype cannot contain non-writable "pri", "val", or "next" properties. We go with a stronger specification, where these properties are not defined 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". Note that, as Node.prototype is shared between all node objects, we cannot inline the definition of NodeProto in the definition of Node. Were we to do that, we could no longer write a satisfiable assertion describing two distinct nodes using the standard separating conjunction.

To fully capture prototype safety, we also need to state that Object.prototype does not contain "pri", "val", and "next". For this, we use the predicate InitialHeap(), which assumes a sealed JavaScript initial heap, which, by default, does not contain these properties. Now, when we write, e.g. Node(n, _, _, _, np) * NodeProto(np) * InitialHeap(), we guarantee encapsulation: we can always create a new node, and we will always use the correct insert function.

The NodeList Predicate. Below, we recursively define a NodeList predicate, which captures a null-terminated list of Node objects singly linked via their next fields. In the base case, we have an empty list and, in the recursive case, a list starting with a node that has priority pri, value val, and points to the next node in the list next. The nodes in the node list are ordered by priority; hence, the priority of the of the head of the list is set as the maximum priority of the tail. All of the nodes in the node list share the same prototype np.

```
NodeList(nl, primax, np) :=
  nl = null * emp
```

```
NodeList(nl, primax, np) := exists pri, val, next.
   Node(nl, pri, val, next, np) * (pri <= primax) *
     NodeList(next, pri, np) * typeOf(primax) = Num
```

Specification of Node.prototype.insert. We are now ready to show the specification of Node.prototype.insert. The function insert is used for inserting a new node object into a node list, maintaining the descending order of priorities. The formal specification of insert is given below:

```
{ NodeList(this, pri_q, np) * Node(n, pri, val, null, np) * NodeProto(np) }
                              insert(n)
          { NodeList(ret, max(pri, pri_q), np) * NodeProto(np) }
```

The precondition states that the this is bound to the head of a node list with maximum priority pri_q and that n is bound to a node with priority pri, value val, and no next element. Notice that n is required to have the same prototype as the all the nodes in this. The postcondition states that insert returns the head of a node list with max priority max(pri, pri_q) (the keyword ret refers to the return value in the postcondition). Due to our abstractions, the specification of insert is fairly straightforward, despite the underlying complexity of JavaScript, and it resembles a specification that one might expect to write for a Java or C++ program.

Specification of PQLib. A priority queue object is simply an object storing a node list in its property "_head" and whose prototype exposes the methods enqueue, for inserting a new element into the priority queue, and dequeue, for popping the element with the highest priority. Due to lack of space, we omit the specifications of enqueue and dequeue, as well as the JaVerT predicates that capture priority queues and priority queue prototypes. All of these are available in the online version of JaVerT (JaVerT Team 2017). We conclude with the specification of the priority queue library function that returns the constructor of priority queues (the function with identifier PQ), as its specification makes use of the JS Logic assertions for reasoning about function closures.

```
                                      { emp }
                                      PQLib()
{            funObj(PQ, ret, pqSC) * dataProp(ret, "prototype", pqp) * scope(PQ, Node: nf, pqSC) *           }
{ funObj(Node, nf, nSC) * dataProp(nf, "prototype", np) * scope(Node, counter: 0, nSC) * QProto(pqp, np, pqSC) }
```

The postcondition states that this function returns a function object corresponding to the function with identifier `PQ`, with scope chain `pqSC` and `"prototype"` property `pqp`. We now reach the first function closure: the `enqueue` function of `PQ.prototype` uses `Node`, hence, we need to state that `Node` is accessible through the scope chain of `PQ`. We do this using the scope assertion of JS Logic, `scope(PQ, Node: nf, pqSC)`, which states that the scope chain of `PQ`, bound to `pqSC`, has access to the variable `Node`, bound to the value denoted by `nf`. Next, we have that `nf` represents a function object corresponding to the function with identifier `Node`, with scope chain `nSC` and `"prototype"` property `np`, and we arrive to the second closure: the `Node` function uses `counter`. Similarly to the previous case, we capture this with `scope(Node, counter: 0, nSC)`. Finally, we state that `pqp` denotes a valid prototype of priority queues.

In this example, it was not the case that two or more closures shared the same variable. When that happens, the solution is to specify one of them and then use the `o−chains` assertion to connect the others. To illustrate, suppose that `enqueue` also needed to use `counter`. In that case, the specification of the two closures that capture `counter` would be `scope(PQ, counter: 0, pqSC) * o-chains(Node: nSC, PQ: pqSC)`. It would then be possible to use entailment to shift between `scope(PQ, counter: 0, pqSC)` and `scope(Node, counter: 0, nSC)`, depending on which of those assertions was required in a given context. In this way, one can specify closures in a principled way, without exposing the underlying scope representation.

8 CONCLUSIONS AND FUTURE WORK

We have developed an 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. We have taken great care to validate our infrastructure, by extensively testing JS-2-JSIL against the ECMAScript test suite, proving the soundness of JSIL Logic, giving correctness results for the JS-2-JSIL logic translator and JaVerT, and creating axiomatic specifications of JavaScript internal functions, as well as verifying their reference implementations using JSIL Verify. We have introduced abstractions for key JavaScript concepts that allow the developer to write JavaScript specifications that resemble specifications for C++ and Java code, with only minimal knowledge about the JavaScript inner workings. For instance, the developer needs to know that prototype chains and scope chains exist and refer to them using logical variables in the specifications; however, they do not need to know how scope chains and prototype chains are represented in the heap or reason about their actual structure. We have highlighted the subtleties of JavaScript by showing the approach to specifying prototype safety and function closures. Finally, we have verified the code of a JavaScript priority queue library.

There are several directions for future work that we are currently exploring. We are in the process of developing an automated tool based on bi-abduction (Calcagno et al. 2009) in the style of Infer, for verifying large JavaScript codebases. We do, however, believe that the semi-automated tool will always have a role to play in the development of functionally correct library specifications. Furthermore, we plan to extend JSIL logic with support for higher-order reasoning, allowing us to reason about JavaScript getters/setters and arbitrary functions as parameters. We plan to investigate the possibility of moving from ES5 Strict to ES6 Strict, as well as to ES5/ES6. Moving to ES6 Strict would only involve extending our work with new language constructs: the specifications of the internal functions would remain the same; the currently supported constructs would still be relevant. Moving to ES5/ES6 would involve a change in how we deal with scoping, in that we would have to model scope lookup using an inductive predicate for capturing the footprint of a dynamic scope chain traversal instead of a built-in assertion that only captures the exact environment record in which a variable is defined.

We hope that our infrastructure will be useful for other styles of JavaScript analysis. So far, we have built a prototype JSIL front-end to CBMC (CBMC Team 2016), 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

Christopher Anderson, Paola Giannini, and Sophia Drossopoulou. 2005. Proceedings of the Towards Type Inference for JavaScript. In *19th European Conference Object-Oriented Programming (Lecture Notes in Computer Science)*. Springer, 428–452.

Esben Andreasen and Anders Møller. 2014. Determinacy in Static Analysis for jQuery. In *OOPSLA*.

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

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

J. Berdine, C. Calcagno, and P. O'Hearn. 2005. Smallfoot: Modular Automatic Assertion Checking with Separation Logic. In *FMCO*.

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

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

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

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

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

C. Calcagno, Dino Distefano, Peter O'Hearn, and Hongseok Yang. 2009. Compositional shape analysis by means of bi-abduction. In *POPL*. DOI: http://dx.doi.org/10.1145/1480881.1480917

CBMC Team. 2016. The JSIL front end of CBMC. https://github.com/diffblue/cbmc/pull/51, https://github.com/diffblue/cbmc/pull/91. (2016).

Andrei Ştefănescu, Daejun Park, Shijiao Yuwen, Yilong Li, and Grigore Roşu. 2016. Semantics-Based Program Verifiers for All Languages. In *Proceedings of the 31th Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'16)*. ACM, 74–91. DOI: http://dx.doi.org/10.1145/2983990.2984027

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

Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'08/ETAPS'08)*. Springer-Verlag, Berlin, Heidelberg, 337–340. http://dl.acm.org/citation.cfm?id=1792734.1792766

Dino Distefano and M. Parkinson. 2008. jStar: Towards practical verification for Java. In *OOPSLA*. http://doi.acm.org/10.1145/1449764.1449782

ECMAScript Committee. 2011. *The 5th edition of the ECMAScript Language Specification.* Technical Report. ECMA.

Facebook. 2017. react.js. https://facebook.github.io/react/. (2017).

Asger Feldthaus and Anders Møller. 2014. 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*.

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

Stephen Fink and Julian Dolby. 2015. WALA — The T.J. Watson Libraries for Analysis. http://wala.sourceforge.net/. (2015).

David Flanagan. 2011. *JavaScript - The Definitive Guide.* O'Reilly.

Cédric Fournet, Gurvan Le Guernic, and Tamara Rezk. 2009. A security-preserving compiler for distributed programs: from information-flow policies to cryptographic mechanisms. In *Proceedings of the 2009 ACM Conference on Computer and Communications Security, CCS 2009, Chicago, Illinois, USA, November 9-13, 2009*, Ehab Al-Shaer, Somesh Jha, and Angelos D. Keromytis (Eds.). ACM, 432–441. DOI: http://dx.doi.org/10.1145/1653662.1653715

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

Google. 2017. V8. http://v8project.blogspot.co.uk. (2017).

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

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

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

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

JaVerT Team. 2017. JaVerT. http://goo.gl/au69SV. (2017).

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

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

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

Daniel Kroening and Michael Tautschnig. 2014. CBMC – C Bounded Model Checker. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS) (LNCS)*, Vol. 8413. Springer, 389–391.

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

Microsoft. 2014. *TypeScript language specification*. Technical Report. Microsoft.

Changhee Park and Sukyoung Ryu. 2015. Scalable and Precise Static Analysis of JavaScript Applications via Loop-Sensitivity. In *ECOOP*. 735–756.

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

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

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

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

Manu Sridharan, Julian Dolby, Satish Chandra, Max Schäfer, and Frank Tip. 2012. Correlation Tracking for Points-To Analysis of JavaScript. In *ECOOP*. 435–458.

Peter Thiemann. 2005. 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)*. Springer, 408–422.

Willem Visser, Corina S. Păsăreanu, and Sarfraz Khurshid. 2004. Test Input Generation with Java PathFinder. In *Proceedings of the 2004 ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '04)*. ACM, New York, NY, USA, 97–107. DOI: http://dx.doi.org/10.1145/1007512.1007526

Hongseok Yang, Oukseh Lee, Josh Berdine, C. Calcagno, Byron Cook, Dino Distefano, and Peter O'Hearn. 2008. Scalable Shape Analysis for Systems Code. In *CAV '08: Proc. of the 20th international conference on Computer Aided Verification*. Springer-Verlag, Berlin, Heidelberg, 385–398. DOI: http://dx.doi.org/10.1007/978-3-540-70545-1_36

A JSIL SYNTAX AND SEMANTICS

Syntax of the JSIL Language.

STRINGS: $m \in \mathcal{Str}$ NUMBERS: $n \in \mathcal{Num}$ BOOLEANS: $b \in \mathcal{Bool}$ LOCATIONS: $l \in \mathcal{L}$

VARIABLES: $\mathsf{x} \in \mathcal{X}_{\mathsf{JSIL}}$ LITERALS: $\lambda \in \mathcal{Lit} \triangleq n \mid b \mid m \mid \mathsf{undefined} \mid \mathsf{null} \mid \mathsf{empty} \mid l \mid \mathsf{t} \mid \overline{\lambda}$

TYPES : $\mathsf{t} \in \mathsf{Types} \triangleq \mathsf{Num} \mid \mathsf{Bool} \mid \mathsf{Str} \mid \mathsf{Undef} \mid \mathsf{Null} \mid \mathsf{Empty} \mid \mathsf{Obj} \mid \mathsf{Type} \mid \mathsf{List}$

EXPRESSIONS : $\mathsf{e} \in \mathcal{E}_{\mathsf{JSIL}} \triangleq \lambda \mid \mathsf{x} \mid \ominus\, \mathsf{e} \mid \mathsf{e} \oplus \mathsf{e} \mid \overline{\mathsf{e}}$

UNARY OPERATORS : $\ominus \triangleq \mathsf{not} \mid \mathsf{head} \mid \mathsf{tail} \mid \mathsf{and} \mid \mathsf{length}_s \mid \mathsf{toString}$
$\mid \mathsf{toNumber} \mid \mathsf{typeOf} \mid \mathsf{ord} \mid \ldots$

BINARY OPERATORS : $\oplus \triangleq + \mid - \mid * \mid / \mid \% \mid = \mid < \mid <_s \mid \mathsf{and} \mid \mathsf{or} \mid :: \mid @$
$\mid \mathsf{nth} \mid @_s \mid \mathsf{nth}_s \mid \ldots$

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

PROCEDURES : $\mathsf{proc} \in \mathsf{Proc} \triangleq \mathsf{proc}\ m(\overline{\mathsf{x}})\{\overline{\mathsf{c}}\}$

Notation: $\overline{\lambda}$, $\overline{\mathsf{x}}$, $\overline{\mathsf{e}}$, $\overline{\mathsf{c}}$, respectively, denote lists of literals, variables, expressions, and commands.

The memory model of JSIL

PROPERTY NAMES : $pn \in \mathcal{P}_{\mathsf{JSIL}} \subset \mathcal{Str}$

JSIL VALUES : $\mathsf{v} \in \mathcal{V}_{\mathsf{JSIL}} \triangleq \mathcal{Lit}$

JSIL STORES : $\rho \in \mathcal{Sto} : \mathcal{X}_{\mathsf{JSIL}} \rightharpoonup \mathcal{V}_{\mathsf{JSIL}}$

JSIL HEAPS : $\mathsf{h} \in \mathcal{H}_{\mathsf{JSIL}} : \mathcal{L} \times \mathcal{P}_{\mathsf{JSIL}} \rightharpoonup \mathcal{V}_{\mathsf{JSIL}}$

JSIL OUTCOMES : $\mathsf{o} \in \mathcal{O}_{\mathsf{JSIL}} \triangleq \mathsf{nm}\langle \mathsf{v} \rangle \mid \mathsf{er}\langle \mathsf{v} \rangle$

Semantics of JSIL Expressions: $\llbracket \mathsf{e} \rrbracket_\rho = \mathsf{v}$

LITERAL
$$\frac{}{\llbracket \lambda \rrbracket_\rho \triangleq \lambda}$$

VARIABLE
$$\frac{}{\llbracket \mathsf{x} \rrbracket_\rho \triangleq \rho(\mathsf{x})}$$

UNARY OPERATOR
$$\frac{}{\llbracket \ominus\, \mathsf{e} \rrbracket_\rho \triangleq \overline{\ominus}(\llbracket \mathsf{e} \rrbracket_\rho)}$$

BINARY OPERATOR
$$\frac{}{\llbracket \mathsf{e}_1 \oplus \mathsf{e}_2 \rrbracket_\rho \triangleq \overline{\oplus}(\llbracket \mathsf{e}_1 \rrbracket_\rho, \llbracket \mathsf{e}_2 \rrbracket_\rho)}$$

EXPRESSION LIST
$$\frac{}{\llbracket \{\{\mathsf{e}_1, \ldots, \mathsf{e}_n\}\} \rrbracket_\rho \triangleq \{\{\llbracket \mathsf{e}_1 \rrbracket_\rho, \ldots, \llbracket \mathsf{e}_n \rrbracket_\rho\}\}}$$

Semantics of Basic Commands: $\llbracket \mathsf{bc} \rrbracket_{h,\rho} = (\mathsf{h}', \rho', \mathsf{v})$

SKIP
$$\llbracket \mathsf{skip} \rrbracket_{\mathsf{h},\rho} \triangleq (\mathsf{h}, \rho, \mathsf{empty})$$

ASSIGNMENT
$$\frac{\llbracket \mathsf{e} \rrbracket_\rho = \mathsf{v} \quad \rho' = \rho[\mathsf{x} \mapsto \mathsf{v}]}{\llbracket \mathsf{x} := \mathsf{e} \rrbracket_{\mathsf{h},\rho} \triangleq (\mathsf{h}, \rho', \mathsf{v})}$$

PROPERTY ACCESS
$$\frac{\mathsf{h}(\llbracket \mathsf{e}_1 \rrbracket_\rho, \llbracket \mathsf{e}_2 \rrbracket_\rho) = \mathsf{v} \quad \rho' = \rho[\mathsf{x} \mapsto \mathsf{v}]}{\llbracket \mathsf{x} := [\mathsf{e}_1, \mathsf{e}_2] \rrbracket_{\mathsf{h},\rho} \triangleq (\mathsf{h}, \rho', \mathsf{v})}$$

PROPERTY ASSIGNMENT
$$\frac{\llbracket \mathsf{e}_3 \rrbracket_\rho = \mathsf{v} \quad \mathsf{h}' = \mathsf{h}[(\llbracket \mathsf{e}_1 \rrbracket_\rho, \llbracket \mathsf{e}_2 \rrbracket_\rho) \mapsto \mathsf{v}]}{\llbracket [\mathsf{e}_1, \mathsf{e}_2] := \mathsf{e}_3 \rrbracket_{\mathsf{h},\rho} \triangleq (\mathsf{h}', \rho, \mathsf{v})}$$

PROPERTY DELETION
$$\frac{\mathsf{h} = \mathsf{h}' \uplus (\llbracket \mathsf{e}_1 \rrbracket_\rho, \llbracket \mathsf{e}_2 \rrbracket_\rho) \mapsto \mathsf{v} \quad \llbracket \mathsf{e}_2 \rrbracket_\rho \neq @\mathit{proto}}{\llbracket \mathsf{delete}\,(\mathsf{e}_1, \mathsf{e}_2) \rrbracket_{\mathsf{h},\rho} \triangleq (\mathsf{h}', \rho, \mathsf{true})}$$

OBJECT CREATION
$$\frac{\mathsf{h}' = \mathsf{h} \uplus (l, @\mathit{proto}) \mapsto \mathsf{null} \quad \rho' = \rho[\mathsf{x} \mapsto l] \quad (l, -) \notin \mathsf{dom}(\mathsf{h})}{\llbracket \mathsf{x} := \mathsf{new}\,() \rrbracket_{\mathsf{h},\rho} \triangleq (\mathsf{h}', \rho', l)}$$

MEMBER CHECK - TRUE
$$\frac{(\llbracket \mathsf{e}_1 \rrbracket_\rho, \llbracket \mathsf{e}_2 \rrbracket_\rho) \in \mathsf{dom}(\mathsf{h}) \quad \rho' = \rho[\mathsf{x} \mapsto \mathsf{true}]}{\llbracket \mathsf{x} := \mathsf{hasField}\,(\mathsf{e}_1, \mathsf{e}_2) \rrbracket_{\mathsf{h},\rho} \triangleq (\mathsf{h}, \rho', \mathsf{true})}$$

MEMBER CHECK - FALSE
$$\frac{(\llbracket \mathsf{e}_1 \rrbracket_\rho, \llbracket \mathsf{e}_2 \rrbracket_\rho) \notin \mathsf{dom}(\mathsf{h}) \quad \rho' = \rho[\mathsf{x} \mapsto \mathsf{false}]}{\llbracket \mathsf{x} := \mathsf{hasField}\,(\mathsf{e}_1, \mathsf{e}_2) \rrbracket_{\mathsf{h},\rho} \triangleq (\mathsf{h}, \rho', \mathsf{false})}$$

GET FIELDS
$$\frac{\begin{array}{c}\llbracket \mathsf{e} \rrbracket_\rho = l \quad \mathsf{h} = (\mathsf{h}' \uplus (l, pn_1) \mapsto - \uplus \ldots \uplus (l, pn_n) \mapsto -) \quad (l, -) \notin \mathsf{dom}(\mathsf{h}') \\ \{\{pn_1, \ldots, pn_n\}\} = \mathsf{v} \quad \mathit{Ord}(\{\{pn_1, \ldots, pn_n\}\}) \quad \rho' = \rho[\mathsf{x} \mapsto \mathsf{v}]\end{array}}{\llbracket \mathsf{x} := \mathsf{getFields}\,(\mathsf{e}) \rrbracket_{\mathsf{h},\rho} \triangleq (\mathsf{h}, \rho', \mathsf{v})}$$

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

Basic Command

$$\frac{\mathsf{p}_m(i) = \mathsf{bc} \in \mathsf{BCmd} \quad [\![\mathsf{bc}]\!]_{\mathsf{h},\rho} = (\mathsf{h}', \rho', -) \quad \mathsf{p} \vdash \langle \mathsf{h}', \rho', i, i+1 \rangle \Downarrow_m \langle \mathsf{h}'', \rho'', \mathsf{o} \rangle}{\mathsf{p} \vdash \langle \mathsf{h}, \rho, \_, i \rangle \Downarrow_m \langle \mathsf{h}'', \rho'', \mathsf{o} \rangle}$$

Goto

$$\frac{\mathsf{p}_m(i) = \mathsf{goto}\ j \quad \mathsf{p} \vdash \langle \mathsf{h}, \rho, i, j \rangle \Downarrow_m \langle \mathsf{h}', \rho', \mathsf{o} \rangle}{\mathsf{p} \vdash \langle \mathsf{h}, \rho, \_, i \rangle \Downarrow_m \langle \mathsf{h}', \rho', \mathsf{o} \rangle}$$

Cond. Goto - True

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

Cond. Goto - False

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

Normal Return

$$\vdash \langle \mathsf{h}, \rho, \_, i_{\mathsf{nm}} \rangle \Downarrow_m \langle \mathsf{h}, \rho, \mathsf{nm}\langle \rho(\mathsf{xret}) \rangle \rangle$$

Error Return

$$\vdash \langle \mathsf{h}, \rho, \_, i_{\mathsf{er}} \rangle \Downarrow_m \langle \mathsf{h}, \rho, \mathsf{er}\langle \rho(\mathsf{xerr}) \rangle \rangle$$

Procedure Call - Normal

$$\frac{\begin{array}{l} \mathsf{p}_m(i) = \mathsf{x} := \mathsf{e}(\mathsf{e}_1, ..., \mathsf{e}_{n_1})\ \mathsf{with}\ j \quad [\![\mathsf{e}]\!]_\rho = m' \\ \mathsf{p}(m') = \mathsf{proc}\ \ m'(\mathsf{y}_1, ..., \mathsf{y}_{n_2})\{\bar{\mathsf{c}}\} \\ \forall_{1 \leq n \leq n_1} \mathsf{v}_n = [\![\mathsf{e}_n]\!]_\rho \quad \forall_{n_1 < n \leq n_2} \mathsf{v}_n = \mathsf{undefined} \\ \mathsf{p} \vdash \langle \mathsf{h}, \emptyset[\mathsf{y}_i \mapsto \mathsf{v}_i|_{i=1}^{n_2}], 0, 0 \rangle \Downarrow_{m'} \langle \mathsf{h}', \rho', \mathsf{nm}\langle \mathsf{v} \rangle \rangle \\ \mathsf{p} \vdash \langle \mathsf{h}', \rho[\mathsf{x} \mapsto \mathsf{v}], i, i+1 \rangle \Downarrow_m \langle \mathsf{h}'', \rho'', \mathsf{o} \rangle \end{array}}{\mathsf{p} \vdash \langle \mathsf{h}, \rho, \_, i \rangle \Downarrow_m \langle \mathsf{h}'', \rho'', \mathsf{o} \rangle}$$

Procedure Call - Error

$$\frac{\begin{array}{l} \mathsf{p}_m(i) = \mathsf{x} := \mathsf{e}(\mathsf{e}_1, ..., \mathsf{e}_{n_1})\ \mathsf{with}\ j \quad [\![\mathsf{e}]\!]_\rho = m' \\ \mathsf{p}(m') = \mathsf{proc}\ \ m'(\mathsf{y}_1, ..., \mathsf{y}_{n_2})\{\bar{\mathsf{c}}\} \\ \forall_{1 \leq n \leq n_1} \mathsf{v}_n = [\![\mathsf{e}_n]\!]_\rho \quad \forall_{n_1 < n \leq n_2} \mathsf{v}_n = \mathsf{undefined} \\ \mathsf{p} \vdash \langle \mathsf{h}, \emptyset[\mathsf{y}_i \mapsto \mathsf{v}_i|_{i=1}^{n_2}], 0, 0 \rangle \Downarrow_{m'} \langle \mathsf{h}', \rho', \mathsf{er}\langle \mathsf{v} \rangle \rangle \\ \mathsf{p} \vdash \langle \mathsf{h}', \rho[\mathsf{x} \mapsto \mathsf{v}], i, j \rangle \Downarrow_m \langle \mathsf{h}'', \rho'', \mathsf{o} \rangle \end{array}}{\mathsf{p} \vdash \langle \mathsf{h}, \rho, \_, i \rangle \Downarrow_m \langle \mathsf{h}'', \rho'', \mathsf{o} \rangle}$$

Phi-Assignment

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

Remark: The unary operator ord returns true when applied to a list of lexicographically ordered strings. Otherwise, it returns false. Analogously, the semantic predicate *Ord* holds for lists of lexicographically ordered strings.

Lemma A.1 (Return Values). *For any JSIL program* p*, heaps* h *and* h_f*, stores* ρ *and* ρ_f*, identifiers i and j, value* v*, procedure identifier m, the following implications hold:*

$$\mathsf{p} \vdash \langle \mathsf{h}, \rho, i, j \rangle \Downarrow_m \langle \mathsf{h}_f, \rho_f, \mathsf{nm}\langle \mathsf{v} \rangle \rangle \implies \rho_f(\mathsf{xret}) = \mathsf{v}$$
$$\mathsf{p} \vdash \langle \mathsf{h}, \rho, i, j \rangle \Downarrow_m \langle \mathsf{h}_f, \rho_f, \mathsf{er}\langle \mathsf{v} \rangle \rangle \implies \rho_f(\mathsf{xerr}) = \mathsf{v}$$

Proof. Both implications are proven by induction on the derivation of the semantic judgement. All cases are proven directly using the induction hypothesis. □

B JSIL LOGIC

JSIL Logic Assertions

$$
\begin{array}{rcl}
\textsc{Logical Values} : V \in \mathcal{V}^L_{\mathsf{JSIL}} & \triangleq & \mathsf{v} \mid \varnothing \mid \overline{V} \\
\textsc{Logical Expressions} : E \in \mathcal{E}^L_{\mathsf{JSIL}} & \triangleq & V \mid \mathsf{x} \mid \mathrm{X} \mid \ominus E \mid E \oplus E \mid \overline{E}
\end{array}
$$

$$
\begin{array}{rcll}
\textsc{JSIL Assertions} : P \in \mathcal{AS}_{\mathsf{JSIL}} & \triangleq & \mathrm{true} \mid \mathrm{false} \mid \neg P \mid P \wedge P \mid \exists \mathrm{X}.P & \textsc{Pure Boolean} \\
& & \mid E = E \mid E \leq E & \textsc{Equalities} \\
& & \mid \mathsf{emp} \mid (E, E) \mapsto E & \textsc{Heap} \\
& & \mid P * P \mid \mathsf{emptyFields}(E \mid \overline{E}) &
\end{array}
$$

$\textsc{Notation}$: $E \neq E \triangleq \neg(E = E), E > E \triangleq \neg(E \leq E), E < E \triangleq E \leq E \wedge E \neq E, E \geq E \triangleq \neg(E < E)$

Additional JSIL Logic Notions

$$
\begin{array}{lll}
\text{JSIL logical environments} : & \epsilon \in \mathcal{E}nv_{\mathsf{JSIL}} & : \mathcal{X}^L_{\mathsf{JSIL}} \rightharpoonup \mathcal{V}^L_{\mathsf{JSIL}} \\
\text{JSIL abstract heaps} & : H \in \mathcal{H}^{\emptyset}_{\mathsf{JSIL}} & : \mathcal{L} \times \mathcal{P}_{\mathsf{JSIL}} \rightharpoonup \mathcal{V}^L_{\mathsf{JSIL}}
\end{array}
$$

Semantics of JSIL Logical Expressions and Assertions

Logical Expressions (Semantics):

$$
\begin{array}{lcl@{\qquad}lcl}
[\![V]\!]^{\epsilon}_{\rho} & \triangleq & V & [\![\ominus E]\!]^{\epsilon}_{\rho} & \triangleq & \overline{\ominus}([\![E]\!]^{\epsilon}_{\rho}) \\
[\![\mathsf{x}]\!]^{\epsilon}_{\rho} & \triangleq & \rho(\mathsf{x}) & [\![E_1 \oplus E_2]\!]^{\epsilon}_{\rho} & \triangleq & \overline{\oplus}([\![E_1]\!]^{\epsilon}_{\rho}, [\![E_2]\!]^{\epsilon}_{\rho}) \\
[\![\mathrm{X}]\!]^{\epsilon}_{\rho} & \triangleq & \epsilon(\mathrm{X}) & [\![\{\{E_1, ..., E_1\}\}]\!]^{\epsilon}_{\rho} & \triangleq & \{\{[\![E_1]\!]^{\epsilon}_{\rho}, ..., [\![E_n]\!]^{\epsilon}_{\rho}\}\}
\end{array}
$$

Assertions (Satisfiability Relation):

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

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

$\textsc{Skip}$

$$\{\mathsf{emp}\}\ \mathsf{skip}\ \{\mathsf{emp}\}$$

$\textsc{Object Creation}$

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

$\textsc{Property Assignment}$

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

$\textsc{Property Access}$

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

$\textsc{Var Assignment}$

$$\{\mathsf{emp}\}\ \mathsf{x} := \mathsf{e}\ \{\mathsf{x} \doteq \mathsf{e}\}$$

$\textsc{Property Deletion}$

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

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

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

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

Symbolic Execution of Control Flow Commands: $\mathsf{p}, \mathsf{S}, i, k \vdash_m^j P \rightsquigarrow Q$

Basic Command
$$\frac{\mathsf{p}_m(i) = \mathsf{bc} \quad \{P\}\, \mathsf{bc}\, \{Q\}}{\mathsf{p}, \mathsf{S}, i, i+1 \vdash_{m,fl}^j P \rightsquigarrow Q}$$

Goto
$$\frac{\mathsf{p}_m(i) = \mathsf{goto}\ k}{\mathsf{p}, \mathsf{S}, i, k \vdash_{m,fl}^j P \rightsquigarrow P}$$

Cond. Goto - True
$$\frac{\mathsf{p}_m(i) = \mathsf{goto}\ [\mathsf{e}]\ k_1,\ k_2}{\mathsf{p}, \mathsf{S}, i, k_1 \vdash_{m,fl}^j P \rightsquigarrow P * \mathsf{e} \doteq \mathsf{true}}$$

Cond. Goto - False
$$\frac{\mathsf{p}_m(i) = \mathsf{goto}\ [\mathsf{e}]\ k_1,\ k_2}{\mathsf{p}, \mathsf{S}, i, k_2 \vdash_{m,fl}^j P \rightsquigarrow P * \mathsf{e} \doteq \mathsf{false}}$$

Phi-Assignment
$$\frac{\mathsf{p}_m(i) = \mathsf{x} := \phi(\mathsf{x}_1, ..., \mathsf{x}_n) \quad j \overset{k}{\mapsto}_m i}{\mathsf{p}, \mathsf{S}, i, i+1 \vdash_{m,fl}^j P \rightsquigarrow P * (\mathsf{x} \doteq \mathsf{x}_k)}$$

Frame Rule
$$\frac{\mathsf{p}, \mathsf{S}, i, k \vdash_{m,fl}^j P \rightsquigarrow Q \quad i \notin \{i_{\mathsf{nm}}, i_{\mathsf{er}}\}}{\mathsf{p}, \mathsf{S}, i, k \vdash_{m,fl}^j P * R \rightsquigarrow Q * R}$$

Existential Elimination
$$\frac{\mathsf{p}, \mathsf{S}, i, k \vdash_{m,fl}^j P \rightsquigarrow Q \quad i \notin \{i_{\mathsf{nm}}, i_{\mathsf{er}}\}}{\mathsf{p}, \mathsf{S}, i, k \vdash_{m,fl}^j \exists \mathsf{X}.\, P \rightsquigarrow \exists \mathsf{X}.\, Q}$$

Consequence
$$\frac{\mathsf{p}, \mathsf{S}, i, k \vdash_{m,fl}^j P \rightsquigarrow Q \quad P' \Rightarrow P \quad Q \Rightarrow Q'}{\mathsf{p}, \mathsf{S}, i, k \vdash_{m,fl}^j P' \rightsquigarrow Q'}$$

Procedure Call - Normal
$$\frac{\begin{array}{l}\mathsf{p}_m(i) = \mathsf{x} := \mathsf{e}_0(\mathsf{e}_1, ..., \mathsf{e}_{n_1})\ \mathsf{with}\ k \\ \mathsf{S}(m', \mathsf{nm}) = \{P\}\, m'(\mathsf{x}_1, ..., \mathsf{x}_{n_2})\, \{Q * \mathsf{xret} \doteq \mathsf{e}\} \quad \mathsf{e}_n = \mathsf{undefined}\,|_{n=n_1+1}^{n2}\end{array}}{\mathsf{p}, \mathsf{S}, i, i+1 \vdash_{m,fl}^j (P[\mathsf{e}_i/\mathsf{x}_i|_{i=1}^{n_2}] * \mathsf{e}_0 \doteq m') \rightsquigarrow (Q[\mathsf{e}_i/\mathsf{x}_i|_{i=1}^{n_2}] * \mathsf{e}_0 \doteq m' * \mathsf{x} \doteq \mathsf{e}[\mathsf{e}_i/\mathsf{x}_i|_{i=1}^{n_2}])}$$

Procedure Call - Error
$$\frac{\begin{array}{l}\mathsf{p}_m(i) = \mathsf{x} := \mathsf{e}_0(\mathsf{e}_1, ..., \mathsf{e}_{n_1})\ \mathsf{with}\ k \\ \mathsf{S}(m', \mathsf{er}) = \{P\}\, m'(\mathsf{x}_1, ..., \mathsf{x}_{n_2})\, \{Q * \mathsf{xerr} \doteq \mathsf{e}\} \quad \mathsf{e}_n = \mathsf{undefined}\,|_{n=n_1+1}^{n2}\end{array}}{\mathsf{p}, \mathsf{S}, i, k \vdash_{m,fl}^j (P[\mathsf{e}_i/\mathsf{x}_i|_{i=1}^{n_2}] * \mathsf{e}_0 \doteq m') \rightsquigarrow (Q[\mathsf{e}_i/\mathsf{x}_i|_{i=1}^{n_2}] * \mathsf{e}_0 \doteq m' * \mathsf{x} \doteq \mathsf{e}[\mathsf{e}_i/\mathsf{x}_i|_{i=1}^{n_2}])}$$

Normal Return
$$\mathsf{p}, \mathsf{S}, i_{\mathsf{nm}}, i_{\mathsf{nm}} \vdash_{m,\mathsf{nm}}^j \mathsf{post}(m, \mathsf{nm}, \mathsf{S}) \rightsquigarrow \mathsf{post}(m, \mathsf{nm}, \mathsf{S})$$

Error Return
$$\mathsf{p}, \mathsf{S}, i_{\mathsf{er}}, i_{\mathsf{er}} \vdash_{m,\mathsf{er}}^j \mathsf{post}(m, \mathsf{er}, \mathsf{S}) \rightsquigarrow \mathsf{post}(m, \mathsf{er}, \mathsf{S})$$

Definition B.1 (Successor relation). Given a JSIL program $\mathsf{p} \in \mathsf{P}$, and a procedure m in p, the *successor relation*, $\mapsto_m \subseteq \mathbb{N} \times \mathbb{N} \times \mathcal{AS}_{\mathsf{JSIL}}$, is defined as follows:

$$\begin{aligned}\mapsto_m \triangleq\ & \{(i, i{+}1) \mid i \notin \{i_{\mathsf{nm}}, i_{\mathsf{er}}\} \land \mathsf{p}_m(i) = \mathsf{bc}\} \\ & \cup \{(i, j) \mid \mathsf{p}_m(i) = \mathsf{goto}\ j\} \ \cup\ \{(i, j), (i, k) \mid \mathsf{p}_m(i) = \mathsf{goto}\ [\mathsf{e}]\ j,\ k\} \\ & \cup\ \{(i, k), (i, i{+}1) \mid \mathsf{p}_m(i) = \mathsf{x} := \mathsf{e}_0(\mathsf{e}_1, ..., \mathsf{e}_{n_1})\ \mathsf{with}\ k\} \\ & \cup \{(i_{\mathsf{nm}}, i_{\mathsf{nm}}), (i_{\mathsf{er}}, i_{\mathsf{er}})\}\end{aligned}$$

Lemma B.2 (Substitution Lemma, Expressions). *Let* $\mathsf{vars}(\mathsf{e}) \subseteq \{\mathsf{x}_i\ |_{i=1}^{n}\}$. *Then:*

$$[\![\mathsf{e}[\mathsf{e}_i/\mathsf{x}_i\ |_{i=1}^{n}]]\!]_\rho \ = \ [\![\mathsf{e}]\!]_{\emptyset[\mathsf{x}_i \mapsto [\![\mathsf{e}_i]\!]_\rho |_{i=1}^{n}]}.$$

Proof. By induction on the structure of e. We have that: $\mathsf{e} \in \mathcal{E}_{\mathsf{JSIL}} \triangleq \lambda \mid \mathsf{x} \mid \ominus\, \mathsf{e} \mid \mathsf{e} \oplus \mathsf{e} \mid \bar{\mathsf{e}}$. As literals are unaffected by substitution, and unary and binary operators and lists are trivially covered by the induction hypothesis, the only case that we need to address is when $\mathsf{e} = \mathsf{x}_i$, for some $i \in \{1, \ldots, n\}$. In that case, our goal becomes:

$$[\![\mathsf{e}_i]\!]_\rho = [\![\mathsf{x}_i]\!]_{\emptyset[\mathsf{x}_i \mapsto [\![\mathsf{e}_i]\!]_\rho |_{i=1}^n]}$$

which holds directly from the definition of $[\![\ ]\!]_\rho$. □

Lemma B.3 (Substitution Lemma, Logical Expressions). *Let* $\mathsf{vars}(E) \subseteq \{\mathsf{x}_i \,|_{i=1}^n\}$. *Then:*

$$[\![E[\mathsf{e}_i/\mathsf{x}_i \,|_{i=1}^n]]\!]_\rho^\epsilon = [\![E]\!]^\epsilon_{\emptyset[\mathsf{x}_i \mapsto [\![\mathsf{e}_i]\!]_\rho |_{i=1}^n]}.$$

Proof. Analogous to the previous lemma, by induction on the structure of E. □

Lemma B.4 (Substitution Lemma, Assertions). *Let* $\mathsf{vars}(P) \subseteq \{\mathsf{x}_i \,|_{i=1}^n\}$. *Then:*

$$H, \rho, \epsilon \models P[\mathsf{e}_i/\mathsf{x}_i \,|_{i=1}^n] \;\Leftrightarrow\; H, \emptyset[\mathsf{x}_i \mapsto [\![\mathsf{e}_i]\!]_\rho \,|_{i=1}^n], \epsilon \models P$$

Proof. We will prove the cases when $P \equiv E_1 = E_2$ and $P \equiv (E_1, E_2) \mapsto E_3$. The remaining cases are either unaffected by substitution, are proven directly using the induction hypothesis, or are proven analogously.

- Let $P \equiv E_1 = E_2$. Then, using Lemma B.3, we have:

$$\begin{array}{c}
H, \rho, \epsilon \models P[\mathsf{e}_i/\mathsf{x}_i \,|_{i=1}^n] \Leftrightarrow \\
H, \rho, \epsilon \models (E_1 = E_2)[\mathsf{e}_i/\mathsf{x}_i \,|_{i=1}^n] \Leftrightarrow \\
[\![E_1[\mathsf{e}_i/\mathsf{x}_i \,|_{i=1}^n]]\!]_\rho^\epsilon = [\![E_2[\mathsf{e}_i/\mathsf{x}_i \,|_{i=1}^n]]\!]_\rho^\epsilon \Leftrightarrow \\
[\![E_1]\!]^\epsilon_{\emptyset[\mathsf{x}_i \mapsto [\![\mathsf{e}_i]\!]_\rho |_{i=1}^n]} = [\![E_2]\!]^\epsilon_{\emptyset[\mathsf{x}_i \mapsto [\![\mathsf{e}_i]\!]_\rho |_{i=1}^n]} \Leftrightarrow \\
[\![E_1]\!]^\epsilon_{\emptyset[\mathsf{x}_i \mapsto [\![\mathsf{e}_i]\!]_\rho |_{i=1}^n]} = [\![E_2]\!]^\epsilon_{\emptyset[\mathsf{x}_i \mapsto [\![\mathsf{e}_i]\!]_\rho |_{i=1}^n]} \Leftrightarrow \\
H, \emptyset[\mathsf{x}_i \mapsto [\![\mathsf{e}_i]\!]_\rho \,|_{i=1}^n], \epsilon \models E_1 = E_2 \Leftrightarrow \\
H, \emptyset[\mathsf{x}_i \mapsto [\![\mathsf{e}_i]\!]_\rho \,|_{i=1}^n], \epsilon \models P
\end{array}$$

- Let $P \equiv (E_1, E_2) \mapsto E_3$. Then, using Lemma B.3, we have:

$$\begin{array}{c}
H, \rho, \epsilon \models P[\mathsf{e}_i/\mathsf{x}_i \,|_{i=1}^n] \Leftrightarrow \\
H, \rho, \epsilon \models ((E_1, E_2) \mapsto E_3)[\mathsf{e}_i/\mathsf{x}_i \,|_{i=1}^n] \Leftrightarrow \\
H = ([\![E_1[\mathsf{e}_i/\mathsf{x}_i \,|_{i=1}^n]]\!]_\rho^\epsilon, [\![E_2[\mathsf{e}_i/\mathsf{x}_i \,|_{i=1}^n]]\!]_\rho^\epsilon) \mapsto [\![E_3[\mathsf{e}_i/\mathsf{x}_i \,|_{i=1}^n]]\!]_\rho^\epsilon \Leftrightarrow \\
H = ([\![E_1]\!]^\epsilon_{\emptyset[\mathsf{x}_i \mapsto [\![\mathsf{e}_i]\!]_\rho |_{i=1}^n]}, [\![E_2]\!]^\epsilon_{\emptyset[\mathsf{x}_i \mapsto [\![\mathsf{e}_i]\!]_\rho |_{i=1}^n]}) \mapsto [\![E_3]\!]^\epsilon_{\emptyset[\mathsf{x}_i \mapsto [\![\mathsf{e}_i]\!]_\rho |_{i=1}^n]} \Leftrightarrow \\
H, \emptyset[\mathsf{x}_i \mapsto [\![\mathsf{e}_i]\!]_\rho \,|_{i=1}^n], \epsilon \models (E_1, E_2) \mapsto E_3 \Leftrightarrow \\
H, \emptyset[\mathsf{x}_i \mapsto [\![\mathsf{e}_i]\!]_\rho \,|_{i=1}^n], \epsilon \models P
\end{array}$$

□

Lemma B.5 (Frame Property and Soundness for Basic Commands).

$$\begin{array}{l}
\forall P, \mathsf{bc}, Q.\ \{P\}\,\mathsf{bc}\,\{Q\} \Rightarrow \\
\quad \forall H, \rho, \epsilon.\ H, \rho, \epsilon \models P \Rightarrow \\
\quad\quad \forall \hat{H}_1, \hat{H}_2, \hat{\mathsf{h}}_f, \rho_f, \mathsf{v}.\ [\![\mathsf{bc}]\!]_{\lfloor H \uplus \hat{H}_1 \uplus \hat{H}_2 \rfloor, \rho} = (\hat{\mathsf{h}}_f, \rho_f, \mathsf{v}) \Rightarrow \\
\quad\quad\quad \exists H_f.\ [\![\mathsf{bc}]\!]_{\lfloor H \uplus \hat{H}_1 \rfloor, \rho} = (\lfloor H_f \uplus \hat{H}_1 \rfloor, \rho_f, \mathsf{v}) \,\wedge\, H_f, \rho_f, \mathsf{v} \models Q \,\wedge\, \hat{\mathsf{h}}_f = \lfloor H_f \uplus \hat{H}_1 \uplus \hat{H}_2 \rfloor.
\end{array}$$

Proof. For convenience, we name the hypotheses as follows:

- **H1**: $\{P\}\,\mathsf{bc}\,\{Q\}$
- **H2**: $H, \rho, \epsilon \models P$
- **H3**: $[\![\mathsf{bc}]\!]_{\lfloor H \uplus \hat{H}_1 \uplus \hat{H}_2 \rfloor, \rho} = (\hat{\mathsf{h}}_f, \rho_f, \mathsf{v})$

Our goal is to show that there exists a JSIL abstract heap H_f, such that:

- **G1**: $[\![\mathsf{bc}]\!]_{\lfloor H \uplus \hat{H}_1 \rfloor, \rho} = (\lfloor H_f \uplus \hat{H}_1 \rfloor, \rho_f, \mathsf{v})$

- **G2**: $H_f, \rho_f, \epsilon \models Q$.
- **G3**: $\hat{\mathsf{h}}_f = \lfloor H_f \uplus \hat{H}_1 \uplus \hat{H}_2 \rfloor$.

We proceed by case analysis on the derivation of $\{P\}\, \mathsf{bc}\, \{Q\}$. We prove the representative cases, whereas the remaining ones are proven analogously.

- [Skip] We have that $\mathsf{bc} = \mathsf{skip}$ and, applying **H1**, that $P = \mathsf{emp}$ and $Q = \mathsf{emp}$. From the satisfiability of JSIL assertions and **H2**, we obtain that $H = \mathsf{emp}$. From **H3** and the semantics of JSIL basic commands, we obtain that $\hat{\mathsf{h}}_f = \lfloor \hat{H}_1 \uplus \hat{H}_2 \rfloor$, $\rho_f = \rho$, and $\mathsf{v} = \mathsf{empty}$. We choose $H_f = \mathsf{emp}$ as our witness. The goals then become:
  - **G1**: $[\![\mathsf{skip}]\!]_{\lfloor \hat{H}_1 \rfloor, \rho} = (\lfloor \hat{H}_1 \rfloor, \rho, \mathsf{empty})$
  - **G2**: $\mathsf{emp}, \rho, \epsilon \models \mathsf{emp}$.
  - **G3**: $\lfloor \hat{H}_1 \uplus \hat{H}_2 \rfloor = \lfloor \mathsf{emp} \uplus \hat{H}_1 \uplus \hat{H}_2 \rfloor$

  and all hold directly from the definitions and hypotheses.

- [Property Assignment] We have that $\mathsf{bc} = [\mathsf{e}_1, \mathsf{e}_2] := \mathsf{e}_3$ and, applying **H1**, that $P = (\mathsf{e}_1, \mathsf{e}_2) \mapsto \_$ and $Q = (\mathsf{e}_1, \mathsf{e}_2) \mapsto \mathsf{e}_3$. From the satisfiability of JSIL assertions and **H2**, we obtain that $H = ([\![\mathsf{e}_1]\!]_\rho, [\![\mathsf{e}_2]\!]_\rho) \mapsto V$, for some value V, possibly $\varnothing$. From **H3** and the semantics of JSIL basic commands, we obtain that $\hat{\mathsf{h}}_f = \lfloor ([\![\mathsf{e}_1]\!]_\rho, [\![\mathsf{e}_2]\!]_\rho) \mapsto [\![\mathsf{e}_3]\!]_\rho \uplus \hat{H}_1 \uplus \hat{H}_2 \rfloor$, $\rho_f = \rho$, and $\mathsf{v} = [\![\mathsf{e}]\!]_{\rho_3}$. We choose $H_f = ([\![\mathsf{e}_1]\!]_\rho, [\![\mathsf{e}_2]\!]_\rho) \mapsto [\![\mathsf{e}_3]\!]_\rho$ as our witness. The goals then become:
  - **G1**: $[\![[\mathsf{e}_1, \mathsf{e}_2] := \mathsf{e}_3]\!]_{\lfloor ([\![\mathsf{e}_1]\!]_\rho, [\![\mathsf{e}_2]\!]_\rho) \mapsto V \uplus \hat{H}_1 \rfloor, \rho} = (\lfloor ([\![\mathsf{e}_1]\!]_\rho, [\![\mathsf{e}_2]\!]_\rho) \mapsto [\![\mathsf{e}_3]\!]_\rho \uplus \hat{H}_1 \rfloor, \rho, [\![\mathsf{e}]\!]_{\rho_3})$
  - **G2**: $([\![\mathsf{e}_1]\!]_\rho, [\![\mathsf{e}_2]\!]_\rho) \mapsto [\![\mathsf{e}_3]\!]_\rho, \rho, \epsilon \models (\mathsf{e}_1, \mathsf{e}_2) \mapsto \mathsf{e}_3$.
  - **G3**: $\lfloor ([\![\mathsf{e}_1]\!]_\rho, [\![\mathsf{e}_2]\!]_\rho) \mapsto [\![\mathsf{e}_3]\!]_\rho \uplus \hat{H}_1 \uplus \hat{H}_2 \rfloor = \lfloor ([\![\mathsf{e}_1]\!]_\rho, [\![\mathsf{e}_2]\!]_\rho) \mapsto [\![\mathsf{e}_3]\!]_\rho \uplus \hat{H}_1 \uplus \hat{H}_2 \rfloor$.

  and all hold directly from the definitions and hypotheses, noting that $[\![\mathsf{e}_3]\!]_\rho \neq \varnothing$.

- [Var Assignment] We have that $\mathsf{bc} = \mathsf{x} := \mathsf{e}$ and, applying **H1**, that $P = \mathsf{emp}$ and $Q = \mathsf{x} \doteq \mathsf{e}$. From the satisfiability of JSIL assertions and **H2**, we obtain that $H = \mathsf{emp}$. From **H3** and the semantics of JSIL basic commands, we obtain that $\hat{\mathsf{h}}_f = \lfloor \hat{H}_1 \uplus \hat{H}_2 \rfloor$, $\rho_f = \rho[\mathsf{x} \mapsto [\![\mathsf{e}]\!]_\rho]$, and $\mathsf{v} = [\![\mathsf{e}]\!]_\rho$. We choose $H_f = \mathsf{emp}$ as our witness. The goals then become:
  - **G1**: $[\![\mathsf{x} := \mathsf{e}]\!]_{\lfloor \hat{H}_1 \rfloor, \rho} = (\lfloor \hat{H}_1 \rfloor, \rho[\mathsf{x} \mapsto [\![\mathsf{e}]\!]_\rho], [\![\mathsf{e}]\!]_\rho)$
  - **G2**: $\mathsf{emp}, \rho[\mathsf{x} \mapsto [\![\mathsf{e}]\!]_\rho], \epsilon \models \mathsf{x} \doteq \mathsf{e}$
  - **G3**: $\lfloor \hat{H}_1 \uplus \hat{H}_2 \rfloor = \lfloor \mathsf{emp} \uplus \hat{H}_1 \uplus \hat{H}_2 \rfloor$

  and all hold directly from the definitions and hypotheses.

- [Object Creation] We have that $\mathsf{bc} = \mathsf{x} := \mathsf{new}\,()$ and, applying **H1**, that $P = \mathsf{emp}$ and $Q = (\mathsf{x}, @proto) \mapsto \mathsf{null} \ast \mathsf{emptyFields}(\mathsf{x} \mid \{\{@proto\}\})$. From the satisfiability of JSIL assertions and **H2**, we obtain that $H = \mathsf{emp}$. From **H3** and the semantics of JSIL basic commands, we obtain that $\hat{\mathsf{h}}_f = (l, @proto) \mapsto \mathsf{null} \uplus \lfloor \hat{H}_1 \uplus \hat{H}_2 \rfloor$, $\rho_f = \rho[\mathsf{x} \mapsto l]$, and $\mathsf{v} = l$, for a fresh location l. We know that $l \notin \mathrm{dom}(\hat{H}_1 \uplus \hat{H}_2)$ from **H3**. We choose $H_f = (l, @proto) \mapsto \mathsf{null} \uplus \left(\biguplus_{m \neq @proto} (l, m) \mapsto \varnothing\right)$ as our witness. Note that then $\lfloor H_f \rfloor = (l, @proto) \mapsto \mathsf{null}$. The goals become:
  - **G1**: $[\![\mathsf{new}\,()]\!]_{\lfloor \hat{H}_1 \rfloor, \rho} = (\lfloor (l, @proto) \mapsto \mathsf{null} \uplus H_1 \rfloor, \rho[\mathsf{x} \mapsto l], l)$
  - **G2**: $H_f, \rho[\mathsf{x} \mapsto l], \epsilon \models (\mathsf{x}, @proto) \mapsto \mathsf{null} \ast \mathsf{emptyFields}(\mathsf{x} \mid \{\{@proto\}\})$
  - **G3**: $(l, @proto) \mapsto \mathsf{null} \uplus \lfloor \hat{H}_1 \uplus \hat{H}_2 \rfloor = \lfloor H_f \uplus \hat{H}_1 \uplus \hat{H}_2 \rfloor$.

  and all hold directly from the definitions and hypotheses, noting that $l \notin \mathrm{dom}(\hat{H}_1 \uplus \hat{H}_2)$.

- [Property Deletion] We have that $\mathsf{bc} = \mathsf{delete}\,(\mathsf{e}_1, \mathsf{e}_2)$ and, applying **H1**, that $P = (\mathsf{e}_1, \mathsf{e}_2) \mapsto \mathsf{X} \ast \mathsf{X} \neq \varnothing$ and $Q = (\mathsf{e}_1, \mathsf{e}_2) \mapsto \varnothing$. From the satisfiability of JSIL assertions and **H2**, we obtain that $H = ([\![\mathsf{e}_1]\!]_\rho, [\![\mathsf{e}_2]\!]_\rho) \mapsto \epsilon(\mathsf{X})$, where $[\![\mathsf{e}_2]\!]_\rho \neq @proto$. Note that $\lfloor H \rfloor = H$ since $\mathsf{X} \neq \varnothing$. From **H3** and the semantics of JSIL basic commands, we obtain that $\hat{h}_f = \lfloor \hat{H}_1 \uplus \hat{H}_2 \rfloor$, $\rho_f = \rho$, and $v = \mathsf{true}$. We choose $H_f = ([\![e_1]\!]_\rho, [\![e_2]\!]_\rho) \mapsto \varnothing$ as our witness, noting that $\lfloor H_f \rfloor = \mathsf{emp}$. The goals become:

- **G1**: $[\![\mathsf{bc}]\!]_{\lfloor ([\![e_1]\!]_\rho, [\![e_2]\!]_\rho) \mapsto \epsilon(X) \uplus \hat{H}_1 \rfloor, \rho} = (\lfloor \hat{H}_1 \rfloor, \rho, \mathsf{true})$
- **G2**: $([\![e_1]\!]_\rho, [\![e_2]\!]_\rho) \mapsto \varnothing, \rho, \epsilon \models (e_1, e_2) \mapsto \varnothing$
- **G3**: $\lfloor \hat{H}_1 \uplus \hat{H}_2 \rfloor = \lfloor ([\![e_1]\!]_\rho, [\![e_2]\!]_\rho) \mapsto \varnothing \uplus \hat{H}_1 \uplus \hat{H}_2 \rfloor$.

and all follow directly from the definitions and hypotheses, noting that the disjoint union in **G4** is well-defined due to **H3**.

- [Property Access] We have that $\mathsf{bc} = x := [e_1, e_2]$ and, applying **H1**, that $P = (e_1, e_2) \mapsto X * X \neq \varnothing$ and $Q = (e_1, e_2) \mapsto X * X \neq \varnothing * x \doteq X$. From the satisfiability of JSIL assertions and **H2**, we obtain that $H = ([\![e_1]\!]_\rho, [\![e_2]\!]_\rho) \mapsto \epsilon(X)$. Note that $\lfloor H \rfloor = H$ since $X \neq \varnothing$. From **H3** and the semantics of JSIL basic commands, we obtain that $\hat{h}_f = \lfloor H \uplus \hat{H}_1 \uplus \hat{H}_2 \rfloor$, $\rho_f = \rho[x \mapsto \epsilon(X)]$, and $v = \epsilon(X)$. We choose $H_f = H = ([\![e_1]\!]_\rho, [\![e_2]\!]_\rho) \mapsto \epsilon(X)$ as our witness. The goals then become:
  - **G1**: $[\![x := [e_1, e_2]]\!]_{\lfloor ([\![e_1]\!]_\rho, [\![e_2]\!]_\rho) \mapsto \epsilon(X) \uplus \hat{H}_1 \rfloor, \rho} = (\lfloor ([\![e_1]\!]_\rho, [\![e_2]\!]_\rho) \mapsto \epsilon(X) \uplus \hat{H}_1 \rfloor, \rho[x \mapsto \epsilon(X)], \epsilon(X))$
  - **G2**: $([\![e_1]\!]_\rho, [\![e_2]\!]_\rho) \mapsto \epsilon(X), \rho[x \mapsto \epsilon(X)], \epsilon \models (e_1, e_2) \mapsto X * X \neq \varnothing * x \doteq X$.
  - **G3**: $\lfloor H \uplus \hat{H}_1 \uplus \hat{H}_2 \rfloor = \lfloor H \uplus \hat{H}_1 \uplus \hat{H}_2 \rfloor$.

  and all hold directly from the definitions and hypotheses, noting that $\epsilon(X) \neq \varnothing$.

- [Member Check - True] Analogous to the Property Access case, with only the return value being different.
- [Member Check - False] Analogous to the Property Access case, noting that $H_1 \uplus H_2$ cannot contain the cell $([\![e_1]\!]_\rho, [\![e_2]\!]_\rho)$, since the disjoint union $H \uplus \hat{H}_1 \uplus \hat{H}_2$ is well-defined.
- [Get Fields] We have that $\mathsf{bc} = x := \mathsf{getFields}\,(e)$ and, applying **H1**, that

$$P = ((e, X_i) \mapsto Y_i|_{i=1}^n) * \mathsf{emptyFields}(e \mid X_i|_{i=1}^n) * (Y_i \neq \varnothing|_{i=1}^n) * (\mathsf{ord}\,(x) \doteq \mathsf{true})$$

and $Q = P * x \doteq \{\{X_1, ..., X_n\}\}$. From the satisfiability of JSIL assertions and **H2**, we obtain that

$$H = \left(\biguplus_{p \notin \{\epsilon(X_i)|_{i=1}^n\}} ([\![e]\!]_\rho, p) \mapsto \varnothing \right) \uplus \left(\biguplus_{i=1}^n ([\![e]\!]_\rho, \epsilon(X_i)) \mapsto \epsilon(Y_i) \right)$$

where $\epsilon(Y_i) \neq \varnothing$ for $i = 1, ..., n$ and $Ord(\{\{\epsilon(X_i)|_{i=1}^n\}\})$. Therefore, we conclude that:

$$\lfloor H \rfloor = \left(\biguplus_{i=1}^n ([\![e]\!]_\rho, \epsilon(X_i)) \mapsto \epsilon(Y_i) \right).$$

From **H3**, the semantics of JSIL basic commands, and recalling that $Ord(\{\{\epsilon(X_i)|_{i=1}^n\}\})$, we obtain that $\hat{h}_f = \lfloor H \uplus \hat{H}_1 \uplus \hat{H}_2 \rfloor$, $\rho_f = \rho[x \mapsto \{\{\epsilon(X_i)|_{i=1}^n\}\}]$, and $v = \{\{\epsilon(X_i)|_{i=1}^n\}\}$. Note that due to the construction of H and the disjoint union $H \uplus \hat{H}_1 \uplus \hat{H}_2$ being well defined, all the fields of the object corresponding to e are captured by H. We choose $H_f = H$ as our witness. The goals become:

- **G1**: $[\![x := \mathsf{getFields}\,(e)]\!]_{\lfloor H \uplus \hat{H}_1 \rfloor, \rho} = (\lfloor H \uplus \hat{H}_1 \rfloor, \rho[x \mapsto \{\{\epsilon(X_i)|_{i=1}^n\}\}], \{\{\epsilon(X_i)|_{i=1}^n\}\})$
- **G2**: $H, \rho[x \mapsto \{\{\epsilon(X_i)\ |_{i=1}^n\}\}], \epsilon \models P * x \doteq \{\{X_i|_{i=1}^n\}\}$.
- **G3**: $\lfloor H \uplus \hat{H} \rfloor = \lfloor H \uplus \hat{H} \rfloor$.

and all hold directly from the definitions and hypotheses.

□

Lemma B.6 (Frame Property and Soundness for Control Flow Commands). *For any derivation* $\mathsf{pd} \in \mathcal{D}$*, program* $\mathsf{p} \in \mathsf{P}$*, specification environment* $\mathsf{S} \in \mathcal{S}tr \rightharpoonup \mathcal{S}pec$*, flag* fl*, abstract heaps* $H, \hat{H}_1, \hat{H}_2 \in \mathcal{H}^{\emptyset}$*, store* $\rho \in \mathcal{S}to$*, logical environment* ϵ*, procedure name* m*, and command labels* i *and* k *such that:*

- **H1***:* $\mathsf{p}, \mathsf{S} \vdash \mathsf{pd}$
- **H2***:* $(P, k) \in \mathsf{pd}(m, \mathsf{fl}, i)$
- **H3***:* $H, \rho, \epsilon \models P$
- **H4***:* $\mathsf{p} \vdash \langle \lfloor H \uplus \hat{H}_1 \uplus \hat{H}_2 \rfloor, \rho, k, i \rangle \Downarrow_m \langle \mathsf{h}_f, \rho_f, o \rangle$

It follows that there is an abstract heap H_f *and a value* v *such that:*

- **G1***:* $o = \mathsf{fl}\langle \mathsf{v} \rangle$
- **G2***:* $\mathsf{p} \vdash \langle \lfloor H \uplus \hat{H}_1 \rfloor, \rho, k, i \rangle \Downarrow_m \langle \lfloor H_f \uplus \hat{H}_1 \rfloor, \rho_f, o \rangle$
- **G3***:* $H_f, \rho_f, \epsilon \models \mathsf{post}(m, \mathsf{fl}, \mathsf{S})$
- **G4***:* $\mathsf{h}_f = \lfloor H_f \uplus \hat{H}_1 \uplus \hat{H}_2 \rfloor$

Proof. We proceed by induction on the derivation of **H4**, using case analysis on the rule applied to obtain **H4**.

[Basic Command] It follows that $\mathsf{p}_m(i) = \mathsf{bc}$ for a given basic command bc. We conclude, using **H5** and the semantics of JSIL, that there is a heap h', a store ρ', and value v', such that:

$$[\![\mathsf{bc}]\!]_{\lfloor H \uplus \hat{H}_1 \uplus \hat{H}_2 \rfloor, \rho} = (\mathsf{h}', \rho', \mathsf{v}') \ \textbf{(I1)} \qquad \mathsf{p} \vdash \langle \mathsf{h}', \rho', i, i{+}1 \rangle \Downarrow_m \langle \mathsf{h}_f, \rho_f, o \rangle \ \textbf{(I2)}$$

Using **H1** and **H2**, we conclude that there is an assertion Q such that: $(Q, i) \in \mathsf{pd}(m, fl, i+1)$ **(I3)**, and $\mathsf{p}, \mathsf{S}, i, i{+}1 \vdash^k_{m,fl} P \rightsquigarrow Q$ **(I4)**. In order to re-establish the premises of the lemma, so we can apply the induction hypothesis to **I2**, we need to show that there is an abstract heap H' such that: $H', \rho', \epsilon \models Q$ and $\lfloor H' \uplus \hat{H}_1 \uplus \hat{H}_2 \rfloor = \mathsf{h}'$. We prove that such an abstract heap exists by induction on the derivation of $\mathsf{p}, \mathsf{S}, i, i{+}1 \vdash^k_{m,fl} P \rightsquigarrow Q$. More concretely, we have to prove that, given $[\![\mathsf{bc}]\!]_{\lfloor H \uplus \hat{H}_1 \uplus \hat{H}_2 \rfloor, \rho} = (\mathsf{h}', \rho', \mathsf{v}')$, $H, \rho, \epsilon \models P$, and $\mathsf{p}, \mathsf{S}, i, i{+}1 \vdash^k_{m,fl} P \rightsquigarrow Q$, there must exist an abstract heap H' such that $\lfloor H' \uplus \hat{H}_1 \uplus \hat{H}_2 \rfloor = \mathsf{h}'$ (*goal 1*), $H', \rho', \epsilon \models Q$ (*goal 2*), and $[\![\mathsf{bc}]\!]_{\lfloor H \uplus \hat{H}_1 \rfloor, \rho} = (\lfloor H' \uplus \hat{H}_1 \rfloor, \rho', \mathsf{v}')$ (*goal 3*). In the following, we proceed by case analysis on the last rule applied to obtain $\mathsf{p}, \mathsf{S}, i, i{+}1 \vdash^k_{m,fl} P \rightsquigarrow Q$.

- [Basic Command] We conclude that: $\{P\}\, \mathsf{bc}\, \{Q\}$ **(C1.1)**. Applying the Frame Property and Soundness for Basic Commands (Lemma B.5) to **C1.1**, **H3**, and **I1**, we conclude that there is an abstract heap H' such that $[\![\mathsf{bc}]\!]_{\lfloor H \uplus \hat{H}_1 \rfloor, \rho} = (\lfloor H' \uplus \hat{H}_1 \rfloor, \rho', \mathsf{v}')$, $H', \rho', \epsilon \models Q$, and $\mathsf{h}' = \lfloor H' \uplus \hat{H}_1 \uplus \hat{H}_2 \rfloor$ (*goals 1-3*).

- [Frame Rule] We conclude that there are three assertions P', Q', and R, such that: $P = P' * R$ **(C2.1)**, $Q = Q' * R$ **(C2.2)**, and $\mathsf{p}, \mathsf{S}, i, i{+}1 \vdash^k_{m,fl} P' \rightsquigarrow Q'$ **(C2.3)**. From **H3** and **C2.1**, it follows that there are two abstract heaps H'_p and H_r such that $H = H'_p \uplus H_r$ **(C2.4)**, $H'_p, \rho, \epsilon \models P'$ **(C2.5)**, and $H_r, \rho, \epsilon \models R$ **(C2.6)**. From **I1** and **C2.4**, we conclude that: $[\![\mathsf{bc}]\!]_{\lfloor (H'_p \uplus H_r) \uplus \hat{H}_1 \uplus \hat{H}_2 \rfloor, \rho} = (\mathsf{h}', \rho', \mathsf{v}')$ **(C2.7)**. Using the associativity of $\uplus$, we get from **C2.7**, that $[\![\mathsf{bc}]\!]_{\lfloor H'_p \uplus (H_r \uplus \hat{H}_1) \uplus \hat{H}_2 \rfloor, \rho} = (\mathsf{h}', \rho', \mathsf{v}')$ **(C2.8)**. Applying the $\underline{\text{inner induction hypothesis}}$ to **C2.8**, **C2.5**, and **C2.3**, we conclude that there is an abs. heap H'_q such that: $\lfloor H'_q \uplus (H_r \uplus \hat{H}_1) \uplus \hat{H}_2 \rfloor = \mathsf{h}'$ **(C2.9)**, $H'_q, \rho', \epsilon \models Q'$ **(C2.10)**, and $[\![\mathsf{bc}]\!]_{\lfloor H'_p \uplus (H_r \uplus \hat{H}_1) \rfloor, \rho} = (\lfloor H'_q \uplus (H_r \uplus \hat{H}_1) \rfloor, \rho', \mathsf{v}')$ **(C2.11)**. We now claim that $H'_q \uplus H_r$ is our witness, having to show that $\lfloor (H'_q \uplus H_r) \uplus \hat{H}_1 \uplus \hat{H}_2 \rfloor = \mathsf{h}'$, $H'_q \uplus H_r, \rho', \epsilon \models Q$, and $[\![\mathsf{bc}]\!]_{\lfloor H \uplus \hat{H}_1 \rfloor, \rho} = (\lfloor (H'_q \uplus H_r) \uplus \hat{H}_1 \rfloor, \rho', \mathsf{v}')$. Given the associativity of $\uplus$, we conclude, from **C2.9**, that $\lfloor (H'_q \uplus H_r) \uplus \hat{H}_1 \uplus \hat{H}_2 \rfloor = \mathsf{h}'$ (*goal 1*) and, from **C2.11**, that $[\![\mathsf{bc}]\!]_{\lfloor (H'_p \uplus H_r) \uplus \hat{H}_1 \rfloor, \rho} = (\lfloor (H'_q \uplus H_r) \uplus \hat{H}_1 \rfloor, \rho', \mathsf{v}')$ (*goal 3*). From **C2.2**, **C2.6**, and **C2.10**, it follows that $H'_q \uplus H_r, \rho', \epsilon \models Q$ (*goal 3*).

- [Consequence] We conclude that there are two assertions P' and Q', such that: $P \Rightarrow P'$ (**C3.1**), $Q' \Rightarrow Q$ (**C3.2**), and $\mathsf{p}, \mathsf{S}, i, i{+}1 \vdash^{k}_{m,fl} P' \rightsquigarrow Q'$ (**C3.3**). From **H3** and **C3.1**, we conclude that $H, \rho, \epsilon \models P'$ (**C3.4**). Applying the $\underline{\text{inner induction hypothesis}}$ to **I1**, **C3.4**, and **C3.3**, it follows that there is an abstract heap H' such that: $\lfloor H' \uplus \hat{H}_1 \uplus \hat{H}_2 \rfloor = \mathsf{h}'$ (*goal 1*), $H', \rho', \epsilon \models Q'$ (**C3.5**), and $[\![\mathsf{bc}]\!]_{\lfloor H \uplus \hat{H}_1 \rfloor, \rho} = (\lfloor H' \uplus \hat{H}_1 \rfloor, \rho', \mathsf{v}')$ (*goal 3*). From **C3.2** and **C3.5**, we conclude that $H', \rho, \epsilon \models Q$ (*goal 2*).
- [Elimination] We conclude that there are two assertions P' and Q', such that: $P = \exists \mathsf{X}.\, P'$ (**C3.1**), $Q = \exists \mathsf{X}.\, Q'$ (**C3.2**), and $\mathsf{p}, \mathsf{S}, i, i{+}1 \vdash^{k}_{m,fl} P' \rightsquigarrow Q'$ (**C3.3**). From **H3** and **C3.1**, it follows that there is a value v such that $H, \rho, \epsilon[\mathsf{X} \mapsto \mathsf{v}] \models P'$ (**C3.4**). Applying the $\underline{\text{inner induction hypothesis}}$ to **I1**, **C3.4**, and **C3.3**, we conclude that there is an abstract heap H' such that: $\lfloor H' \uplus \hat{H}_1 \uplus \hat{H}_2 \rfloor = \mathsf{h}'$ (*goal 1*), $H', \rho', \epsilon[\mathsf{X} \mapsto \mathsf{v}] \models Q'$ (**C3.5**), and $[\![\mathsf{bc}]\!]_{\lfloor H \uplus \hat{H}_1 \rfloor, \rho} = (\lfloor H' \uplus \hat{H}_1 \rfloor, \rho', \mathsf{v}')$ (*goal 3*). From **C3.2** and **C3.5**, we conclude that $H, \rho, \epsilon \models Q$ (*goal 2*).
- [All Other Cases] No other rule may have been applied in the derivation of $\mathsf{p}, \mathsf{S}, i, i{+}1 \vdash^{k}_{m,fl} P \rightsquigarrow Q$ because bc is a basic command. Hence, we do not have to analyse those cases.

We have established that there is an abstract heap H' such that $\lfloor H' \uplus \hat{H}_1 \uplus \hat{H}_2 \rfloor = \mathsf{h}'$ (**I5**), $H', \rho', \epsilon \models Q$ (**I6**), and $[\![\mathsf{bc}]\!]_{\lfloor H \uplus \hat{H}_1 \rfloor, \rho} = (\lfloor H' \uplus \hat{H}_1 \rfloor, \rho', \mathsf{v}')$ (**I7**). Applying the $\underline{\text{induction hypothesis}}$ to **H1**, **I3**, **I6**, and **I2**, we have that there is an abs. heap H_f and value v such that: $o = fl\langle \mathsf{v} \rangle$ (**G1**), $\mathsf{p} \vdash \langle \lfloor H' \uplus \hat{H}_1 \rfloor, \rho', i, i+1 \rangle \Downarrow_m \langle \lfloor H_f \uplus \hat{H}_1 \rfloor, \rho_f, o \rangle$ (**I8**), $H_f, \rho_f, \epsilon \models \mathsf{post}(m, fl, \mathsf{S})$ (**G3**), and $\mathsf{h}_f = \lfloor H_f \uplus \hat{H}_1 \uplus \hat{H}_2 \rfloor$ (**G4**). Recalling that $\mathsf{p}_m(i) = \mathsf{bc}$, it follows from **I7** and **I8** that $\mathsf{p} \vdash \langle \lfloor H \uplus \hat{H}_1 \rfloor, \rho, k, i \rangle \Downarrow_m \langle \lfloor H_f \uplus \hat{H}_1 \rfloor, \rho_f, o \rangle$ (**G1**).

[Goto] It follows that $\mathsf{p}_m(i) = \mathsf{goto}\ j$ for a given command index j. We conclude, using **H4** and the semantics of JSIL, that:

$$\mathsf{p} \vdash \langle \lfloor H \uplus \hat{H}_1 \uplus \hat{H}_2 \rfloor, \rho, i, j \rangle \Downarrow_m \langle \mathsf{h}_f, \rho_f, o \rangle \ \textbf{(I1)}$$

Using **H1** and **H2**, we conclude that there is an assertion Q such that: $(Q, i) \in \mathsf{pd}(m, fl, j)$ (**I2**) and $\mathsf{p}, \mathsf{S}, i, j \vdash^{k}_{m,fl} P \rightsquigarrow Q$ (**I3**). In order to re-establish the premises of the lemma, so we can apply the induction hypothesis to **I1**, we need to show that $H, \rho, \epsilon \models Q$. Like in the previous case, we proceed by induction on the derivation of $\mathsf{p}, \mathsf{S}, i \vdash^{k}_{m,fl} P \rightsquigarrow Q$. More concretely, given that $H, \rho, \epsilon \models P$ and $\mathsf{p}, \mathsf{S}, i, j \vdash^{k}_{m,fl} P \rightsquigarrow Q$, we need to show that $H, \rho, \epsilon \models Q$.

- [Goto] We conclude that: $\mathsf{p}, \mathsf{S}, i, j \vdash^{k}_{m,fl} P \rightsquigarrow P$ (**C1.1**), from which it follows that $Q = P$ (**C1.2**). From **C1.2** and **H3**, it follows that $H, \rho, \epsilon \models Q$ (*goal*).
- [Frame Rule] We conclude that there are three assertions P', Q', and R, such that: $P = P' * R$ (**C2.1**), $Q = Q' * R$ (**C2.2**), and $\mathsf{p}, \mathsf{S}, i, j \vdash^{k}_{m,fl} P' \rightsquigarrow Q'$ (**C2.3**). From **H3** and **C2.1** it follows that there are two abstract heaps H'_p and H_r such that $H = H'_p \uplus H_r$ (**C2.4**), $H'_p, \rho, \epsilon \models P'$ (**C2.5**), and $H_r, \rho, \epsilon \models R$ (**C2.6**). Applying the $\underline{\text{inner induction hypothesis}}$ to **C2.5** and **C2.3**, we conclude that $H'_p, \rho, \epsilon \models Q'$ (**C2.7**). From **C2.4**, **C2.6**, and **C2.7**, it follows that $H, \rho, \epsilon \models Q$ (*goal*).
- [Consequence] We conclude that there are two assertions P' and Q', such that: $P \Rightarrow P'$ (**C3.1**), $Q' \Rightarrow Q$ (**C3.2**), and $\mathsf{p}, \mathsf{S}, i, j \vdash^{k}_{m,fl} P' \rightsquigarrow Q'$ (**C3.3**). From **H3** and **C3.1**, we conclude that $H, \rho, \epsilon \models P'$ (**C3.4**). Applying the $\underline{\text{inner induction hypothesis}}$ to **C3.4**, and **C3.3**, it follows that $H, \rho, \epsilon \models Q'$ (**C3.5**). From **C3.2** and **C3.5**, we have that $H, \rho, \epsilon \models Q$ (*goal*).
- [Elimination] We conclude that there are two assertions P' and Q', such that: $P = \exists \mathsf{X}.\, P'$ (**C3.1**), $Q = \exists \mathsf{X}.\, Q'$ (**C3.2**), and $\mathsf{p}, \mathsf{S}, i, j \vdash^{k}_{m,fl} P' \rightsquigarrow Q'$ (**C3.3**). From **H3** and **C3.1**, it follows that there is a value v

such that $H, \rho, \epsilon[\mathsf{X} \mapsto \mathsf{v}] \models P'$ (**C3.4**). Applying the inner induction hypothesis to **C3.4** and **C3.3**, we conclude that $H, \rho', \epsilon[\mathsf{X} \mapsto \mathsf{v}] \models Q'$ (**C3.5**). From **C3.2** and **C3.5**, we conclude that $H, \rho, \epsilon \models Q$ (*goal*).

- [All Other Cases] No other rule may have been applied in the derivation of $\mathsf{p}, \mathsf{S}, i, j \vdash^{k}_{m,fl} P \rightsquigarrow Q$ because $\mathsf{p}_m(i) = \mathsf{goto}\ j$.

Having established that $H, \rho, \epsilon \models Q$ (**I4**), we can apply the induction hypothesis to **H1**, **I2**, **I4**, and **I1** to conclude that there is an abstract heap H_f and JSIL value v such that: $o = fl\langle \mathsf{v} \rangle$ (**G1**), $\mathsf{p} \vdash \langle \lfloor H' \uplus \hat{H}_1 \rfloor, \rho', i, i+1 \rangle \Downarrow_m \langle \lfloor H_f \uplus \hat{H}_1 \rfloor, \rho_f, o \rangle$ (**I5**), $H_f, \rho_f, \epsilon \models \mathsf{post}(m, fl, \mathsf{S})$ (**G3**), and $\mathsf{h}_f = \lfloor H_f \uplus \hat{H}_1 \uplus \hat{H}_2 \rfloor$ (**G4**). Recalling that $\mathsf{p}_m(i) = \mathsf{goto}\ j$, it follows from **I5** that $\mathsf{p} \vdash \langle \lfloor H \uplus \hat{H}_1 \rfloor, \rho, k, i \rangle \Downarrow_m \langle \lfloor H_f \uplus \hat{H}_1 \rfloor, \rho_f, o \rangle$ (**G2**).

[Conditional Goto - True] It follows that $\mathsf{p}_m(i) = \mathsf{goto}\ [\mathsf{e}]\ j_1,\ j_2$ for two command indexes j_1 and j_2 and a JSIL expression e. We conclude, using **H4** and the semantics of JSIL, that:

$$[\![\mathsf{e}]\!]_\rho = \mathsf{true}\ (\mathbf{I1}) \qquad \mathsf{p} \vdash \langle \lfloor H \rfloor, \rho, i, j_1 \rangle \Downarrow_m \langle h_f, \rho_f, o \rangle\ (\mathbf{I2})$$

Using **H1** and **H2**, we conclude that there is an assertion Q such that: $(Q, i) \in \mathsf{pd}(m, fl, j_1)$ (**I3**), and $\mathsf{p}, \mathsf{S}, i, j_1 \vdash^{k}_{m,fl} P \rightsquigarrow Q$ (**I4**) . In order to re-establish the premises of the lemma, so we can apply the induction hypothesis to **I2**, we need to show that $H, \rho, \epsilon \models Q$. As in the previous case, we proceed by induction on the derivation of $\mathsf{p}, \mathsf{S}, i, j_1 \vdash^{k}_{m,fl} P \rightsquigarrow Q$. More concretely, given that $H, \rho, \epsilon \models P$ and $\mathsf{p}, \mathsf{S}, i, j_1 \vdash^{k}_{m,fl} P \rightsquigarrow Q$, we need to show that $H, \rho, \epsilon \models Q$.

- [Conditional Goto - True] We conclude that: $\mathsf{p}, \mathsf{S}, i, j_1 \vdash^{k}_{m,fl} P \rightsquigarrow P\ \wedge\ \mathsf{e} = \mathsf{true}$. Noting that e does not contain any logical variables, we conclude that: $[\![\mathsf{e}]\!]_\rho = [\![\mathsf{e}]\!]^{\epsilon}_{\rho}$ (**C1.1**). From **I1** and **C1.1**, it follows that $[\![\mathsf{e}]\!]^{\epsilon}_{\rho} = \mathsf{true}$ (**C1.2**). From **H3** and **C1.2**, we conclude that $H, \rho, \epsilon \models (P\ \wedge\ \mathsf{e} = \mathsf{true})$ (*goal*).
- [Frame Rule, Consequence, Elimination] These cases exactly coincide with the corresponding ones in the proof of [Goto].
- [All Other Cases] No other rule may have been applied in the derivation of $\mathsf{p}, \mathsf{S}, i, j_1 \vdash^{k}_{m,fl} P \rightsquigarrow Q$ because $\mathsf{p}_m(i) = \mathsf{goto}\ [\mathsf{e}]\ j_1,\ j_2$.

Having established that $H, \rho, \epsilon \models Q$ (**I5**), we can apply the induction hypothesis to **H1**, **I3**, **I5**, and **I2** to conclude that there is an abstract heap H_f and JSIL value v such that: $o = fl\langle \mathsf{v} \rangle$ (**G1**), $\mathsf{p} \vdash \langle \lfloor H' \uplus \hat{H}_1 \rfloor, \rho', i, i+1 \rangle \Downarrow_m \langle \lfloor H_f \uplus \hat{H}_1 \rfloor, \rho_f, o \rangle$ (**I6**), $H_f, \rho_f, \epsilon \models \mathsf{post}(m, fl, \mathsf{S})$ (**G3**), and $\mathsf{h}_f = \lfloor H_f \uplus \hat{H}_1 \uplus \hat{H}_2 \rfloor$ (**G4**). Recalling that $\mathsf{p}_m(i) = \mathsf{goto}\ [\mathsf{e}]\ j_1,\ j_2$, it follows from **I1** and **I6** that $\mathsf{p} \vdash \langle \lfloor H \uplus \hat{H}_1 \rfloor, \rho, k, i \rangle \Downarrow_m \langle \lfloor H_f \uplus \hat{H}_1 \rfloor, \rho_f, o \rangle$ (**G2**).

[Procedure Call - Normal] It follows that $\mathsf{p}_m(i) = \mathsf{x} := \mathsf{e}(\mathsf{e}_1, ..., \mathsf{e}_{n_1})\ \mathsf{with}\ j$ for a given *JSIL* variable x, n JSIL expressions $\mathsf{e}, \mathsf{e}_1, ..., \mathsf{e}_{n_1}$, and index j. We conclude, using **H4** and the semantics of JSIL, that:

$$[\![\mathsf{e}]\!]_\rho = m'\ (\mathbf{I1}) \qquad \mathsf{p}(m') = \mathsf{proc}\ m'(\mathsf{y}_1, ..., \mathsf{y}_{n_2})\{\bar{\mathsf{c}}\}\ (\mathbf{I2}) \qquad \forall_{1 \leq n \leq n_1} \mathsf{v}_n = [\![\mathsf{e}_n]\!]_\rho\ (\mathbf{I3})$$
$$\forall_{n_1 < n \leq n_2} \mathsf{v}_n = \mathsf{undefined}\ (\mathbf{I4}) \quad \mathsf{p} \vdash \langle \lfloor H \uplus \hat{H}_1 \uplus \hat{H}_2 \rfloor, \emptyset[\mathsf{y}_i \mapsto \mathsf{v}_i|_{i=1}^{n_2}], 0, 0 \rangle \Downarrow_{m'} \langle \mathsf{h}', \rho', \mathsf{nm}\langle \mathsf{v}' \rangle \rangle\ (\mathbf{I5})$$
$$\mathsf{p} \vdash \langle \mathsf{h}', \rho[\mathsf{x} \mapsto \mathsf{v}'], i, i+1 \rangle \Downarrow_m \langle \mathsf{h}_f, \rho_f, o \rangle\ (\mathbf{I6})$$

Using **H1** and **H2**, we conclude that there is an assertion Q such that: $(Q, i) \in \mathsf{pd}(m, fl, i{+}1)$ (**I7**) and $\mathsf{p}, \mathsf{S}, i, i{+}1 \vdash^{k}_{m,fl} P \rightsquigarrow Q$ (**I8**). In order to re-establish the premises of the lemma, so we can apply the induction hypothesis to **I6**, we need to show that there is an abstract heap H' such that: $H', \rho[\mathsf{x} \mapsto \mathsf{v}], \epsilon \models Q$, and $\lfloor H' \uplus \hat{H}_1 \uplus \hat{H}_2 \rfloor = \mathsf{h}'$. We prove that such an abstract heap exists by induction on the derivation of $\mathsf{p}, \mathsf{S}, i, i{+}1 \vdash^{k}_{m,fl} P \rightsquigarrow Q$. More concretely, we have to prove that, given $\mathsf{p} \vdash \langle \lfloor H \uplus \hat{H}_1 \uplus \hat{H}_2 \rfloor, \emptyset[\mathsf{y}_i \mapsto \mathsf{v}_i|_{i=1}^{n_2}], 0, 0 \rangle \Downarrow_{m'} \langle \mathsf{h}', \rho', \mathsf{nm}\langle \mathsf{v}' \rangle \rangle$, $H, \rho, \epsilon \models P$, and $\mathsf{p}, \mathsf{S}, i, i{+}1 \vdash^{k}_{m,fl} P \rightsquigarrow Q$, there must exist an abstract heap H' such that $\lfloor H' \uplus \hat{H}_1 \uplus \hat{H}_2 \rfloor = \mathsf{h}'$ (*goal 1*),

$H', \rho[\mathsf{x} \mapsto \mathsf{v}'], \epsilon \models Q$ (*goal 2*), and $\mathsf{p} \vdash \langle \lfloor H \uplus \hat{H}_1 \rfloor, \emptyset[\mathsf{y}_i \mapsto \mathsf{v}_i|_{i=1}^{n_2}], 0, 0 \rangle \Downarrow_{m'} \langle \lfloor H' \uplus \hat{H}_1 \rfloor, \rho', \mathsf{nm}\langle \mathsf{v}' \rangle \rangle$ (*goal 3*). In the following, we proceed by case analysis on the last rule applied to obtain $\mathsf{p}, \mathsf{S}, i, i{+}1 \vdash_{m,fl}^{k} P \rightsquigarrow Q$.

- [Procedure Call - Normal] We conclude that there are two assertions P' and Q' such that $P = P'[\mathsf{e}_i/\mathsf{x}_i|_{i=1}^{n_3}] * \mathsf{e}_0 \doteq m''$ (**C1.1**), $Q = Q'[\mathsf{e}_i/\mathsf{x}_i|_{i=1}^{n_3}] * \mathsf{e}_0 \doteq m'' * \mathsf{x} \doteq \mathsf{e}'[\mathsf{e}_i/\mathsf{x}_i|_{i=1}^{n_3}]$ (**C1.2**), $\mathsf{S}(m', \mathsf{nm}) = \{P'\}\, m''(\mathsf{x}_1, ..., \mathsf{x}_{n_3})\, \{Q' * \mathtt{xret} \doteq \mathsf{e}'\}$ (**C1.3**), where $\mathsf{e}_n = \mathsf{undefined}\ |_{n=n_1+1}^{n_3}$. From **H3** and **C1.1**, it follows that $[\![\mathsf{e}]\!]_{\rho}^{\epsilon} = m''$ (**C1.4**). Noting that $[\![\mathsf{e}]\!]_{\rho} = [\![\mathsf{e}]\!]_{\rho}^{\epsilon}$, we conclude, from **I1** and **C1.4**, that $m' = m''$ (**C1.5**), $n_2 = n_3$ (**C1.6**), and $(\mathsf{x}_1, ..., \mathsf{x}_{n_3}) = (\mathsf{y}_1, ..., \mathsf{y}_{n_2})$ (**C1.7**). For convenience, we use **C1.5**-**C1.7** to rewrite **C1.1**-**C1.3** as follows: $P = P'[\mathsf{e}_i/\mathsf{y}_i|_{i=1}^{n_2}] * \mathsf{e}_0 \doteq m'$ (**C1.8**), $Q = Q'[\mathsf{e}_i/\mathsf{y}_i|_{i=1}^{n_2}] * \mathsf{e}_0 \doteq m' * \mathsf{x} \doteq \mathsf{e}'[\mathsf{e}_i/\mathsf{y}_i|_{i=1}^{n_2}]$ (**C1.9**), $\mathsf{S}(m', \mathsf{nm}) = \{P'\}\, m'(\mathsf{y}_1, ..., \mathsf{y}_{n_2})\, \{Q' * \mathtt{xret} \doteq \mathsf{e}'\}$ (**C1.10**), where $\mathsf{e}_n = \mathsf{undefined}\ |_{n=n_1+1}^{n_2}$.

  Noting that all the specs in S are well-formed, we conclude, from **C1.10**, that $\mathsf{vars}(P') \cup \mathsf{vars}(Q') \cup \mathsf{vars}(\mathsf{e}) \subseteq \{\mathsf{y}_1, ..., \mathsf{y}_{n_2}\}$ (**C1.11**). Applying the Substitution Lemma for Assertions (Lemma B.4) to **H3**, **C1.11**, and **C1.8**, we conclude that $H, \emptyset[\mathsf{y}_i \mapsto [\![\mathsf{e}_i]\!]_{\rho}^{\epsilon}|_{i=1}^{n_2}], \epsilon \models P'$ (**C1.12**). For $1 \leq i \leq n_1$, it holds that $[\![\mathsf{e}_i]\!]_{\rho}^{\epsilon} = [\![\mathsf{e}_i]\!]_{\rho}$ (**C1.13**), because e_i does not contain logical variables. For $n_1 < i \leq n_2$, it holds that $[\![\mathsf{e}_i]\!]_{\rho}^{\epsilon} = \mathsf{undefined}$ (**C1.14**), because $\mathsf{e}_i = \mathsf{undefined}$. From **I3**, **I4**, **C1.13**, and **C1.14**, it follows that $\emptyset[\mathsf{y}_i \mapsto \mathsf{v}_i|_{i=1}^{n_2}] = \emptyset[\mathsf{y}_i \mapsto [\![\mathsf{e}_i]\!]_{\rho}^{\epsilon}|_{i=1}^{n_2}]$ (**C1.15**). From **C1.12** and **C1.15**, we have that $H, \emptyset[\mathsf{y}_i \mapsto \mathsf{v}_i|_{i=1}^{n_2}], \epsilon \models P'$ (**C1.16**). From **H1** and **C1.10**, we conclude that $(P', 0) \in \mathsf{pd}(m, \mathsf{nm}, 0)$ (**C1.17**). We can now apply the outer induction hypothesis to **H1**, **C1.17**, **C1.16**, and **I5**, obtaining that there is a heap H': $\mathsf{p} \vdash \langle \lfloor H \uplus \hat{H}_1 \rfloor, \emptyset[\mathsf{y}_i \mapsto \mathsf{v}_i|_{i=1}^{n_2}], 0, 0 \rangle \Downarrow_{m'} \langle \lfloor H' \uplus \hat{H}_1 \rfloor, \rho', \mathsf{nm}\langle \mathsf{v}' \rangle \rangle$ (*goal 3*), $H', \rho', \epsilon \models \mathsf{post}(m', \mathsf{nm}, \mathsf{S})$ (**C1.18**), and $\mathsf{h}' = \lfloor H' \uplus \hat{H}_1 \uplus \hat{H}_2 \rfloor$ (*goal 1*). From **C1.10** and **C1.18**, it follows that $H', \rho', \epsilon \models Q' * \mathtt{xret} \doteq \mathsf{e}'$ (**C1.19**). Since we only consider programs in SSA, we conclude, from *goal 3*, that $\rho' \geq \emptyset[\mathsf{y}_i \mapsto \mathsf{v}_i|_{i=1}^{n_2}]$ (**C1.20**). From **C1.11**, **C1.15**, **C1.19**, and **C1.20**, we have that $H', \emptyset[\mathsf{y}_i \mapsto [\![\mathsf{e}_i]\!]_{\rho}^{\epsilon}|_{i=1}^{n_2}], \epsilon \models Q'$ (**C1.21**). Applying the Substitution Lemma for Assertions (Lemma B.4) to **C1.11** and **C1.21**, we conclude that $H', \rho, \epsilon \models Q'[\mathsf{e}_i/\mathsf{y}_i|_{i=1}^{n_2}]$ (**C1.22**). Applying Lemma A.1 to **I5**, it follows that $\rho'(\mathtt{xret}) = \mathsf{v}'$, from which we have, using **C1.19**, that $[\![\mathsf{e}']\!]_{\rho'}^{\epsilon} = \mathsf{v}'$ (**C1.23**). From **C1.11**, **C1.15**, **C1.20**, and **C1.23**, we have that $[\![\mathsf{e}']\!]^{\epsilon}_{\emptyset[\mathsf{y}_i \mapsto [\![\mathsf{e}_i]\!]_{\rho}^{\epsilon}|_{i=1}^{n_2}]} = \mathsf{v}'$ (**C1.24**). Applying the Substitution Lemma for Logical Expressions (Lemma B.3) to **C1.24**, we conclude that $[\![\mathsf{e}'[\mathsf{e}_i/\mathsf{y}_i|_{i=1}^{n_2}]]\!]_{\rho}^{\epsilon} = \mathsf{v}'$ (**C1.25**). Since we only consider programs in SSA, we know that $\mathsf{x} \notin \bigcup_{i=1}^{n_2} \mathsf{vars}(\mathsf{e}_i)$, from which it follows that $\mathsf{x} \notin \mathsf{vars}(Q'[\mathsf{e}_i/\mathsf{y}_i|_{i=1}^{n_2}]) \cup \mathsf{vars}(\mathsf{e}'[\mathsf{e}_i/\mathsf{y}_i|_{i=1}^{n_2}])$ (**C1.26**). Combining **C1.22**, **C1.25**, and **C1.26**, we get that $H', \rho[\mathsf{x} \mapsto \mathsf{v}'], \epsilon \models Q'[\mathsf{e}_i/\mathsf{y}_i|_{i=1}^{n_2}] * \mathsf{x} \doteq \mathsf{e}'[\mathsf{e}_i/\mathsf{y}_i|_{i=1}^{n_2}]$ (*goal 2*).

- [Frame Rule, Consequence, Elimination] These cases exactly coincide with the corresponding ones in the proof of [Basic Command].

- [All Other Cases] No other rule may have been applied in the derivation of $\mathsf{p}, \mathsf{S}, i, i{+}1 \vdash_{m,fl}^{k} P \rightsquigarrow Q$ because $\mathsf{p}_m(i) = \mathsf{x} := \mathsf{e}(\mathsf{e}_1, ..., \mathsf{e}_{n_1})$ with j.

We have established that there is an abstract heap H' such that $\lfloor H' \uplus \hat{H}_1 \uplus \hat{H}_2 \rfloor = \mathsf{h}'$ (**I9**), $H', \rho[\mathsf{x} \mapsto \mathsf{v}'], \epsilon \models Q$ (**I10**), and $\mathsf{p} \vdash \langle \lfloor H \uplus \hat{H}_1 \rfloor, \emptyset[\mathsf{y}_i \mapsto \mathsf{v}_i|_{i=1}^{n_2}], 0, 0 \rangle \Downarrow_{m'} \langle \lfloor H' \uplus \hat{H}_1 \rfloor, \rho', \mathsf{nm}\langle \mathsf{v}' \rangle \rangle$ (**I11**). Applying the induction hypothesis to **H1**, **I7**, **I10**, and **I6**, we conclude that there is an abstract heap H_f and JSIL value v such that: $o = fl\langle \mathsf{v} \rangle$ (**G1**), $\mathsf{p} \vdash \langle \lfloor H' \uplus \hat{H}_1 \rfloor, \rho[\mathsf{x} \mapsto \mathsf{v}'], i, i+1 \rangle \Downarrow_m \langle \lfloor H_f \uplus \hat{H}_1 \rfloor, \rho_f, o \rangle$ (**I12**), $H_f, \rho_f, \epsilon \models \mathsf{post}(m, fl, \mathsf{S})$ (**G3**), $\mathsf{h}_f = \lfloor H_f \uplus \hat{H}_1 \uplus \hat{H}_2 \rfloor$ (**G4**). Recalling that $\mathsf{p}_m(i) = \mathsf{x} := \mathsf{e}(\mathsf{e}_1, ..., \mathsf{e}_{n_1})$ with j, it follows from **I1**-**I4**, **I11**, and **I12** that $\mathsf{p} \vdash \langle \lfloor H \uplus \hat{H}_1 \rfloor, \rho, k, i \rangle \Downarrow_m \langle \lfloor H_f \uplus \hat{H}_1 \rfloor, \rho_f, o \rangle$ (**G2**).

[Phi-Assignment] It follows that $\mathsf{p}_m(i) = \mathsf{x} := \phi(\mathsf{x}_1, ..., \mathsf{x}_n)$ for some *JSIL* variables $\mathsf{x}, \mathsf{x}_1, ..., \mathsf{x}_n$. We conclude, using **H4** and the semantics of JSIL, that:

$$i \overset{k}{\mapsto}_m j \ \textbf{(I1)} \qquad \mathsf{p} \vdash \langle \mathsf{h}, \rho[\mathsf{x} \mapsto \rho(\mathsf{x}_k)], j, j+1 \rangle \Downarrow_m \langle \mathsf{h}', \rho', o \rangle \ \textbf{(I2)}$$

for a given index k. Using **H1** and **H2**, we conclude that there is an assertion Q such that: $(Q, i) \in \mathsf{pd}(m, \mathit{fl}, i{+}1)$ **(I3)** and $\mathsf{p}, \mathsf{S}, i, i{+}1 \vdash^k_m P \rightsquigarrow Q$ **(I4)**. In order to re-establish the premises of the lemma, so we can apply the induction hypothesis to **I2**, we need to show that $H, \rho[\mathsf{x} \mapsto \rho(\mathsf{x}_k)], \epsilon \models Q$. As in the previous cases, we proceed by induction on the derivation of $\mathsf{p}, \mathsf{S}, i, i{+}1 \vdash^k_m P \rightsquigarrow Q$. More concretely, given that $H, \rho, \epsilon \models P$ and $\mathsf{p}, \mathsf{S}, i, i{+}1 \vdash^k_{m,\mathit{fl}} P \rightsquigarrow Q$, we need to show that $H, \rho[\mathsf{x} \mapsto \rho(\mathsf{x}_k)], \epsilon \models Q$. In the following, we proceed by case analysis on the last rule applied to obtain $\mathsf{p}, \mathsf{S}, i, i{+}1 \vdash^k_{m,\mathit{fl}} P \rightsquigarrow Q$.

- [Phi-Assignment] We conclude that $Q = P * (\mathsf{x} \doteq \mathsf{x}_k)$ **(C1.1)**. Because we assume programs are in SSA form, we conclude that $\mathsf{x} \notin \mathsf{vars}(P)$ **(C1.2)**. From **H3** and **C1.2**, we have that $H, \rho[\mathsf{x} \mapsto \rho(\mathsf{x}_k)], \epsilon \models P$ **(C1.3)**. Using the satisfiability of JSIL assertions, we get that $H, \rho[\mathsf{x} \mapsto \rho(\mathsf{x}_k)], \epsilon \models \mathsf{x} \doteq \mathsf{x}_k$ **(C1.4)**. Combining **C1.3** and **C1.4**, we conclude that $H, \rho[\mathsf{x} \mapsto \rho(\mathsf{x}_k)], \epsilon \models Q$ (*goal*).

- [Frame Rule, Consequence, Elimination] These cases exactly coincide with the corresponding ones in the proof of [Goto].

- [All Other Cases] No other rule may have been applied in the derivation of $\mathsf{p}, \mathsf{S}, i, i{+}1 \vdash^k_{m,\mathit{fl}} P \rightsquigarrow Q$ because $\mathsf{p}_m(i) = \mathsf{x} := \phi(\mathsf{x}_1, ..., \mathsf{x}_n)$.

Having established $H, \rho[\mathsf{x} \mapsto \rho(\mathsf{x}_k)], \epsilon \models Q$ **(I5)**, we can apply the induction hypothesis to **H1**, **I3**, **I5**, and **I2**, concluding that there is an abstract heap H_f and a JSIL value v such that: $o = \mathit{fl}\langle \mathsf{v} \rangle$ **(G1)**, $\mathsf{p} \vdash \langle \lfloor H \uplus \hat{H}_1 \rfloor, \rho[\mathsf{x} \mapsto \rho(\mathsf{x}_k)], i, i+1 \rangle \Downarrow_m \langle \lfloor H_f \uplus \hat{H}_1 \rfloor, \rho_f, o \rangle$ **(I6)**, $H_f, \rho_f, \epsilon \models \mathsf{post}(m, \mathit{fl}, \mathsf{S})$ **(G3)**, $\mathsf{h}_f = \lfloor H_f \uplus \hat{H}_1 \uplus \hat{H}_2 \rfloor$ **(G4)**. Recalling that $\mathsf{p}_m(i) = \mathsf{x} := \phi(\mathsf{x}_1, ..., \mathsf{x}_n)$, it follows from **I1** and **I6** that $\mathsf{p} \vdash \langle \lfloor H \uplus \hat{H}_1 \rfloor, \rho, k, i \rangle \Downarrow_m \langle \lfloor H_f \uplus \hat{H}_1 \rfloor, \rho_f, \mathsf{v} \rangle$ **(G2)**.

[Normal Return] It follows that $i = i_{\mathsf{nm}}$ **(I1)**, $\mathsf{h}_f = \lfloor H \uplus \hat{H}_1 \uplus \hat{H}_2 \rfloor$ **(I2)**, $\rho_f = \rho$ **(I3)**, and $o = \mathsf{nm}\langle \rho(\mathtt{xret}) \rangle$ **(I4)**. Using **H1** and **H2**, we conclude that there is an assertion Q such that $\mathsf{p}, \mathsf{S}, i, i \vdash^k_{m,\mathit{fl}} P \rightsquigarrow Q$ **(I5)**. From **I1** and **I5**, it follows that $\mathit{fl} = \mathsf{nm}$ **(I6)**. In the following, we use H as our witness for H_f and $\rho(\mathtt{xret})$ as our witness for v. Hence, we now need to prove that $H, \rho, \epsilon \models \mathsf{post}(m, \mathsf{nm}, \mathsf{S})$. We proceed by induction on the derivation of $\mathsf{p}, \mathsf{S}, i_{\mathsf{nm}}, i_{\mathsf{nm}} \vdash^k_{m,\mathit{fl}} P \rightsquigarrow Q$. More concretely, given that $H, \rho, \epsilon \models P$ and $\mathsf{p}, \mathsf{S}, i_{\mathsf{nm}}, i_{\mathsf{nm}} \vdash^k_{m,\mathit{fl}} P \rightsquigarrow Q$, we need to show that $H, \rho, \epsilon \models \mathsf{post}(m, \mathsf{nm}, \mathsf{S})$. In the following, we proceed by case analysis on the last rule applied to obtain $\mathsf{p}, \mathsf{S}, i_{\mathsf{nm}}, i_{\mathsf{nm}} \vdash^k_{m,\mathit{fl}} P \rightsquigarrow Q$.

- [Normal Return] We conclude that $P = Q = \mathsf{post}(m, \mathsf{nm}, \mathsf{S})$ **(C1.1)**. From **H3** and **C1.1**, we conclude that $H, \rho, \epsilon \models \mathsf{post}(m, \mathsf{nm}, \mathsf{S})$.

- [Consequence] This case is proven as the corresponding one in the proof of [Goto].

- [All Other Cases] No other rule may have been applied in the derivation of $\mathsf{p}, \mathsf{S}, i_{\mathsf{nm}}, i_{\mathsf{nm}} \vdash^k_{m,\mathit{fl}} P \rightsquigarrow Q$ because i_{nm} has no successor but itself.

Having proven that $H_f, \rho_f, \epsilon \models \mathsf{post}(m, \mathsf{nm}, \mathsf{S})$ **(G3)**, we proceed to the remaining goals. We obtain **G1** directly from **I4** and **I6**. Using the semantics of JSIL and **I1**, we get that: $\mathsf{p} \vdash \langle \lfloor H \uplus \hat{H}_1 \rfloor, \rho, k, i \rangle \Downarrow_m \langle \lfloor H \uplus \hat{H}_1 \rfloor, \rho, \mathsf{nm}\langle \rho(\mathtt{xret}) \rangle \rangle$ **(I7)**. From **I3**, **I4**, and **I7**, we obtain **G2**.

[Error Return] It follows that $i = i_{\mathsf{er}}$ **(I1)**, $\mathsf{h}_f = \lfloor H \uplus \hat{H}_1 \uplus \hat{H}_2 \rfloor$ **(I2)**, $\rho_f = \rho$ **(I3)**, and $o = \mathsf{er}\langle \rho(\mathtt{xerr}) \rangle$ **(I4)**. Using **H1** and **H2**, we conclude that there is an assertion Q such that $\mathsf{p}, \mathsf{S}, i, i \vdash^k_{m,\mathit{fl}} P \rightsquigarrow Q$ **(I5)**. From **I1** and **I5**, it follows that $\mathit{fl} = \mathsf{er}$ **(I6)**. In the following, we use H as our witness for H_f and $\rho(\mathtt{xerr})$ as our witness for v. Hence, we now need to prove that $H, \rho, \epsilon \models \mathsf{post}(m, \mathsf{er}, \mathsf{S})$. We proceed by induction on the derivation of $\mathsf{p}, \mathsf{S}, i_{\mathsf{er}}, i_{\mathsf{er}} \vdash^{k}_{m,fl} P \rightsquigarrow Q$. More concretely, given that $H, \rho, \epsilon \models P$ and $\mathsf{p}, \mathsf{S}, i_{\mathsf{er}}, i_{\mathsf{er}} \vdash^{k}_{m,fl} P \rightsquigarrow Q$, we need to show that $H, \rho, \epsilon \models \mathsf{post}(m, \mathsf{er}, \mathsf{S})$. In the following, we proceed by case analysis on the last rule applied to obtain $\mathsf{p}, \mathsf{S}, i_{\mathsf{er}}, i_{\mathsf{er}} \vdash^{k}_{m,fl} P \rightsquigarrow Q$.

- [Error Return] We conclude that $P = Q = \mathsf{post}(m, \mathsf{er}, \mathsf{S})$ (**C1.1**). From **H3** and **C1.1**, we conclude that $H, \rho, \epsilon \models \mathsf{post}(m, \mathsf{er}, \mathsf{S})$.
- [Consequence] This case is proven as the corresponding one in the proof of [Goto].
- [All Other Cases] No other rule may have been applied in the derivation of $\mathsf{p}, \mathsf{S}, i_{\mathsf{er}}, i_{\mathsf{er}} \vdash^{k}_{m,fl} P \rightsquigarrow Q$ because i_{nm} has no successor but itself.

Having proven that $H_f, \rho_f, \epsilon \models \mathsf{post}(m, \mathsf{er}, \mathsf{S})$ (**G3**), we proceed to the remaining goals. We obtain **G1** directly from **I4** and **I6**. Using the semantics of JSIL and **I1**, we get that: $\mathsf{p} \vdash \langle \lfloor H \uplus \hat{H}_1 \rfloor, \rho, k, i \rangle \Downarrow_m \langle \lfloor H \uplus \hat{H}_1 \rfloor, \rho, \mathsf{er}\langle \rho(\mathtt{xerr}) \rangle \rangle$ (**I7**). From **I3**, **I4**, and **I7**, we obtain **G2**.

□

Lemma B.7 (Soundness of Basic Commands). *For all basic commands* $\mathsf{bc} \in \mathsf{BCmd}$*, abstract heaps* $H \in \mathcal{H}^{\emptyset}_{\mathtt{JSIL}}$*, stores* $\rho \in \mathcal{S}to$*, logical environments* $\epsilon \in \mathcal{E}nv$*, and assertions* $P, Q \in \mathcal{AS}_{\mathtt{JSIL}}$*, if* $\{P\}\, \mathsf{bc}\, \{Q\}$*,* $H, \rho, \epsilon \models P$*, and* $[\![\mathsf{bc}]\!]_{\lfloor H \rfloor, \rho} = (h', \rho', \mathsf{v})$*, then there is an abstract heap* H' *such that* $H', \rho', \epsilon \models Q$*.*

Proof. Follows immediately from Lemma B.5 by picking $\hat{H}_1 = \hat{H}_2 = \mathsf{emp}$. □

(Theorem 5.1 - JSIL logic soundness). *For any JSIL program* p *and specification environment* S*, if there exists a proof candidate* $\mathsf{pd} \in \mathcal{D}$ *such that* $\mathsf{p}, \mathsf{S} \vdash \mathsf{pd}$*, then* S *is valid for* p*.*

Proof. We have to prove that for every procedure name m, return mode fl, JSIL abstract heap H, concrete heap h_f, stores ρ and ρ_f, environment ϵ, and outcome o such that $\mathsf{S}(m, fl) = \{P\}\, m(\overline{\mathsf{x}})\, \{Q\}$ (**I1**), $H, \rho, \epsilon \models P$ (**I2**), and $\mathsf{p} \vdash \langle \lfloor H \rfloor, \rho, -, 0 \rangle \Downarrow_m \langle \mathsf{h}_f, \rho_f, o \rangle$ (**I3**), it holds that there is an abstract heap H_f and value v such that $\lfloor H_f \rfloor = \mathsf{h}_f$ (**G1**), $H_f, \rho_f, \epsilon \models Q$ (**G2**), and $o = fl\langle \mathsf{v} \rangle$ (**G3**). From $\mathsf{p}, \mathsf{S} \vdash \mathsf{pd}$, we conclude that $\mathsf{pd}(m, fl, 0) = \{(P, 0)\}$ (**I4**). We can now apply the Frame Property and Soundness for Control Flow Commands (Lemma B.6) to $\mathsf{p}, \mathsf{S} \vdash \mathsf{pd}$, **I4**, **I2**, and **I3**, by choosing $\hat{H}_1 = \hat{H}_2 = \mathsf{emp}$, concluding that there exists an abstract heap H_f and a JSIL value v such that **G1**-**G3** hold. □

C β-CORRESPONDENCE

We introduce a family of β-correspondence relations to connect a number of notions in JavaScript to appropriate notions in JSIL. These relations are all based on finite injective functions $\beta : \mathcal{L} \rightharpoonup \mathcal{L}$, mapping locations of JavaScript objects to locations of JSIL objects. For simplicity, we assume that β always maps literal locations of the JavaScript initial heap (global object, Object.prototype, etc.) to themselves.

β-correspondence for outcomes

Constants and Literals
$$\frac{o \in \mathcal{L}it \cup \{\mathsf{empty}\}}{o \sim_\beta o}$$

Locations
$$l \sim_\beta \beta(l)$$

Lists of values
$$\frac{(v_i \sim_\beta v'_i)|_{i=1}^n}{\{\{v_i|_{i=1}^n\}\} \sim_\beta \{\{v'_i|_{i=1}^n\}\}}$$

Normal return
$$\frac{v \sim_\beta v'}{\mathsf{nm}\langle v\rangle \sim_\beta \mathsf{nm}\langle v'\rangle}$$

Error return
$$\frac{v \sim_\beta v'}{\mathsf{er}\langle v\rangle \sim_\beta \mathsf{er}\langle v'\rangle}$$

β-correspondence for heaps/abstract heaps

Empty heap
$$\mathsf{emp} \sim_\beta \mathsf{emp}$$

Standard Properties
$$\frac{l \sim_\beta l' \quad v \sim_\beta v' \quad x \neq @code}{(l, x) \mapsto v \sim_\beta (l', x) \mapsto v'}$$

Heap Composition
$$\frac{h_1 \sim_\beta \mathsf{h}_1 \quad h_2 \sim_\beta \mathsf{h}_2}{h_1 \uplus h_2 \sim_\beta \mathsf{h}_1 \uplus \mathsf{h}_2}$$

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

Remaining β-correspondence

Stores
$$\frac{\mathsf{dom}(\rho_{\mathsf{JS}}) = \mathsf{dom}(\rho_{\mathsf{JSIL}}) \quad \forall x \in \mathsf{dom}(\rho_{\mathsf{JS}}).\ \rho_{\mathsf{JS}}(x) \sim_\beta \rho_{\mathsf{JSIL}}(x)}{\rho_{\mathsf{JS}} \sim_\beta \rho_{\mathsf{JSIL}}}$$

Execution Contexts
$$\frac{H_{\mathsf{JS}} \sim_\beta H_{\mathsf{JSIL}} \quad \rho_{\mathsf{JSIL}} = \rho'_{\mathsf{JSIL}} \uplus [\mathsf{x}_{sc} \mapsto \beta(L), \mathsf{x}_{\mathsf{this}} \mapsto \beta(l_t)] \quad \rho_{\mathsf{JS}} \sim_\beta \rho'_{\mathsf{JSIL}}}{H_{\mathsf{JS}}, \rho_{\mathsf{JS}}, L, l_t \simeq_\beta H_{\mathsf{JSIL}}, \rho_{\mathsf{JSIL}}}$$

Logical Environments
$$\frac{\mathsf{dom}(\epsilon_{\mathsf{JS}}) = \mathsf{dom}(\epsilon_{\mathsf{JSIL}}) \quad \forall x \in \mathsf{dom}(\epsilon_{\mathsf{JS}}).\ \epsilon_{\mathsf{JS}}(x) \sim_\beta \epsilon_{\mathsf{JSIL}}(x)}{\epsilon_{\mathsf{JS}} \sim_\beta \epsilon_{\mathsf{JSIL}}}$$

Logical Contexts
$$\frac{H_{\mathsf{JS}} \sim_\beta H_{\mathsf{JSIL}} \quad \rho_{\mathsf{JSIL}} = \rho'_{\mathsf{JSIL}} \uplus [\mathsf{x}_{sc} \mapsto \beta(L), \mathsf{x}_{\mathsf{this}} \mapsto \beta(l_t)] \quad \rho_{\mathsf{JS}} \sim_\beta \rho'_{\mathsf{JSIL}} \quad \epsilon_{\mathsf{JS}} \sim_\beta \epsilon_{\mathsf{JSIL}}}{H_{\mathsf{JS}}, \rho_{\mathsf{JS}}, L, l_t, \epsilon_{\mathsf{JS}} \simeq_\beta H_{\mathsf{JSIL}}, \rho_{\mathsf{JSIL}}, \epsilon_{\mathsf{JSIL}}}$$

We extend the function application β in the standard way to JS and JSIL values, outcomes, logical values, logical expressions, heaps, abstract heaps, stores, and logical environments. We also extend the notion of domain to those categories to capture the object locations featured in them. We state the following lemmas:

Lemma C.1 (Properties of β-correspondence). *Let T range over the syntactic categories for which $\sim_\beta$ is defined (outcomes, heaps, abstract heaps, stores, logical environments). We have that the following properties hold:*

(1) $|\mathsf{dom}(T_{\mathsf{JS}})| = |\mathsf{dom}(T_{\mathsf{JSIL}})|$
(2) $\beta(\mathsf{dom}(T_{\mathsf{JS}})) = \mathsf{dom}(T_{\mathsf{JSIL}})$; $\mathsf{dom}(T_{\mathsf{JS}}) = \beta^{-1}(\mathsf{dom}(T_{\mathsf{JSIL}}))$
(3) $T_{\mathsf{JS}} \sim_\beta T_{\mathsf{JSIL}} \Leftrightarrow \beta(T_{\mathsf{JS}}) \sim_{\mathsf{id}_{\mathsf{dom}(\beta)}} T_{\mathsf{JSIL}} \Leftrightarrow T_{\mathsf{JS}} \sim_{\mathsf{id}_{\mathsf{dom}(\beta)}} \beta^{-1}(T_{\mathsf{JSIL}})$
(4) $T_{\mathsf{JS}} \sim_{\beta_1} T'_{\mathsf{JSIL}} \Rightarrow T_{\mathsf{JS}} \sim_{\beta_2} T''_{\mathsf{JSIL}} \Rightarrow \beta_2(\beta_1^{-1}(T'_{\mathsf{JSIL}}) = T''_{\mathsf{JSIL}}$

D JS-2-JSIL — LOGIC

JS Logic: Logical Values, Logical Expressions, Assertions

$V \in \mathcal{V}^L_{\mathrm{JS}}$	$::=$	$v \mid \varnothing \mid \overline{V}$	
$E \in \mathcal{E}^L_{\mathrm{JS}}$	$::=$	$V \mid \mathsf{x} \mid \mathsf{X} \mid \ominus E \mid E \oplus E \mid \mathsf{typeOf}\,(E) \mid \{\{E_1, ..., E_n\}\} \mid \mathsf{nth}\,(E_1, E_2) \mid \mathsf{this}$	
$P, Q \in \mathcal{AS}_{\mathrm{JS}}$	$::=$	$\mathsf{true} \mid \mathsf{false} \mid E_1 = E_2 \mid E_1 \leq E_2$	Boolean
		$\mid P \wedge Q \mid \neg P \mid \exists \mathsf{x}.P$	Classical
		$\mid \mathsf{emp} \mid (E, E) \mapsto E \mid \mathsf{emptyFields}(E \mid E_1, ..., E_n) \mid P * Q$	Spatial
		$\mid \mathsf{scope}(m, x : E_1, E_2) \mid \mathsf{funObj}(m, E_1, E_2) \mid \mathsf{o\text{-}chains}(m_1 : E_1, m_2 : E_2)$	Scoping and Fun. Objs.

Semantics of JS Logical Expressions and Assertions

Logical Expressions (Semantics):

$[\![V]\!]^{\epsilon}_{\rho,l_t,L}$	$\triangleq$	V
$[\![\mathsf{x}]\!]^{\epsilon}_{\rho,l_t,L}$	$\triangleq$	$\rho(\mathsf{x})$
$[\![\mathsf{X}]\!]^{\epsilon}_{\rho,l_t,L}$	$\triangleq$	$\epsilon(\mathsf{X})$
$[\![\ominus E]\!]^{\epsilon}_{\rho,l_t,L}$	$\triangleq$	$\overline{\ominus}([\![E]\!]^{\epsilon}_{\rho,l_t,L})$
$[\![E_1 \oplus E_2]\!]^{\epsilon}_{\rho,l_t,L}$	$\triangleq$	$\overline{\oplus}([\![E_1]\!]^{\epsilon}_{\rho,l_t,L}, [\![E_2]\!]^{\epsilon}_{\rho,l_t,L})$
$[\![\mathsf{typeOf}\ E]\!]^{\epsilon}_{\rho,l_t,L}$	$\triangleq$	$\mathsf{TypeOf}([\![E]\!]^{\epsilon}_{\rho,l_t,L})$
$[\![\{\{E_1, ..., E_1\}\}]\!]^{\epsilon}_{\rho,l_t,L}$	$\triangleq$	$\{\{[\![E_1]\!]^{\epsilon}_{\rho,l_t,L}, ..., [\![E_n]\!]^{\epsilon}_{\rho,l_t,L}\}\}$
$[\![\mathsf{nth}\,(E_1, E_2)]\!]^{\epsilon}_{\rho,l_t,L}$	$\triangleq$	V_k if $[\![E_1]\!]^{\epsilon}_{\rho,l_t,L} = \{\{V_0, ..., V_n\}\}$, $[\![E_2]\!]^{\epsilon}_{\rho,l_t,L} = k$, and $0 \leq k \leq n$
$[\![\mathsf{this}]\!]^{\epsilon}_{\rho,l_t,L}$	$\triangleq$	l_t
$[\![\underline{1}]\!]^{\epsilon}_{\rho,l_t,L}$	$\triangleq$	L

Assertions (Satisfiability Relation) for $C = H, \rho, L, l_t, \epsilon$:

$C \models \mathsf{true}$	$\Leftrightarrow$ always
$C \models \mathsf{false}$	$\Leftrightarrow$ never
$C \models E_1 = E_2$	$\Leftrightarrow [\![E_1]\!]^{\epsilon}_{\rho,l_t,L} = [\![E_2]\!]^{\epsilon}_{\rho,l_t,L}$
$C \models E_1 \leq E_2$	$\Leftrightarrow [\![E_1]\!]^{\epsilon}_{\rho,l_t,L} \leq [\![E_2]\!]^{\epsilon}_{\rho,l_t,L}$
$C \models P \wedge Q$	$\Leftrightarrow H, \rho, L, l_t, \epsilon \models P \wedge Q$ and $H, \rho, L, l_t, \epsilon \models Q$
$C \models \neg P$	$\Leftrightarrow H, \rho, L, l_t, \epsilon \not\models P$
$C \models \exists \mathsf{X}.P$	$\Leftrightarrow \exists V \in \mathcal{V}^L_{\mathrm{JSIL}}.\, H, \rho, L, l_t, \epsilon[\mathsf{X} \mapsto V] \models P$
$C \models \mathsf{emp}$	$\Leftrightarrow H = \mathsf{emp}$
$C \models (E_1, E_2) \mapsto E_3$	$\Leftrightarrow H = ([\![E_1]\!]^{\epsilon}_{\rho,l_t,L}, [\![E_2]\!]^{\epsilon}_{\rho,l_t,L}) \mapsto [\![E_3]\!]^{\epsilon}_{\rho,l_t,L}$
$C \models \mathsf{emptyFields}(E \mid E_1, ..., E_n)$	$\Leftrightarrow H = \biguplus_{m \notin M} H([\![E]\!]^{\epsilon}_{\rho,l_t,L}, m) = \varnothing \ \wedge\ M = \{[\![E_i]\!]^{\epsilon}_{\rho,l_t,L}\,\vert^{n}_{i=1}\}$
$C \models P * Q$	$\Leftrightarrow \exists H_1, H_2.\, H = H_1 \uplus H_2 \wedge (H_1, \rho, L, l_t, \epsilon \models P) \wedge (H_2, \rho, L, l_t, \epsilon \models Q)$
$C \models \mathsf{scope}(m', x : E_1, E_2)$	$\Leftrightarrow v = [\![E_1]\!]^{\epsilon}_{\rho,l_t,L} \ \wedge\ L' = [\![E_2]\!]^{\epsilon}_{\rho,l_t,L} \ \wedge\ \psi(m', x) = i\ \wedge$ $((i = 0 \ \wedge\ L'(0) = l_g \ \wedge\ H = ((l_g, x) \mapsto \{\{\text{“}d\text{”}, v, \mathsf{true}, \mathsf{true}, \mathsf{false}\}\}))$ $\vee (i \neq 0 \ \wedge\ H = ((L'(i), x) \mapsto v)))$
$C \models \mathsf{funObj}(m', E_1, E_2)$	$\Leftrightarrow l_f = [\![E_1]\!]^{\epsilon}_{\rho,l_t,L} \ \wedge\ L' = [\![E_2]\!]^{\epsilon}_{\rho,l_t,L}\ \wedge$ $H = ((l_f, @code) \mapsto s(m')) \uplus ((l_f, @scope) \mapsto L') \wedge$ $\psi^{o}(m, m') = k \ \wedge\ \bigwedge_{0 \leq i \leq k} L(i) = L'(i)$
$C \models \mathsf{o\text{-}chains}(m_1 : E_1, m_2 : E_2)$	$\Leftrightarrow L_1 = [\![E_1]\!]^{\epsilon}_{\rho,l_t,L} \ \wedge\ L_2 = [\![E_2]\!]^{\epsilon}_{\rho,l_t,L} \ \wedge\ \psi^{o}(m_1, m_2) = k \ \wedge\ \bigwedge_{0 \leq i \leq k} L_1(i) = L_2(i)$

Lemma D.1 (Renaming locations using β preserves satisfiability). *It holds that:*

(1) $H_{\text{JS}}, \rho_{\text{JS}}, L, l_t, \epsilon_{\text{JS}} \models P \Leftrightarrow \beta(H_{\text{JS}}), \beta(\rho_{\text{JS}}), \beta(L), \beta(l_t), \beta(\epsilon_{\text{JS}}) \models P$
(2) $H_{\text{JSIL}}, \rho_{\text{JSIL}}, \epsilon_{\text{JSIL}} \models P \Leftrightarrow \beta(H_{\text{JSIL}}), \beta(\rho_{\text{JSIL}}), \beta(\epsilon_{\text{JSIL}}) \models P$

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

$$\mathcal{T}_e(V) = V \qquad \mathcal{T}_e(\mathsf{x}) = \mathsf{x}$$

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

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

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

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

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

$$\mathcal{T}_e(\mathsf{this}) = \mathsf{x}_{this} \qquad \mathcal{T}_e(\underline{1}) = \mathsf{x}_{sc}$$

Function Specification

$$\frac{\mathcal{T}(P) = P' \quad \mathcal{T}(Q) = Q' \quad \overline{\mathsf{x}}' = (\mathsf{x}_{sc}, \mathsf{x}_{this}, \overline{\mathsf{x}})}{\mathcal{T}(\{P\}\, m(\overline{\mathsf{x}})\, \{Q\}) \triangleq \{P'\}\, m(\overline{\mathsf{x}}')\, \{Q'\}}$$

JavaScript Specification Environment

$$\frac{\mathrm{dom}(\mathsf{S}_{\text{JS}}) = \mathrm{dom}(\mathsf{S}_{\text{JSIL}}) \quad \forall\, (m, fl) \in \mathrm{dom}(\mathsf{S}_{\text{JS}})\,.\, \mathsf{S}_{\text{JSIL}}(m, fl) = \mathcal{T}(\mathsf{S}_{\text{JS}}(m, fl))}{\mathcal{T}(\mathsf{S}_{\text{JS}}) \triangleq \mathsf{S}_{\text{JSIL}}}$$

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

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

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

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

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

$$\frac{\mathcal{T}_e(E_1) = E_1' \quad \mathcal{T}_e(E_2) = E_2' \quad \mathcal{T}_e(E_3) = E_3'}{\mathcal{T}_a((E_1, E_2) \mapsto E_3) \triangleq (E_1', E_2') \mapsto E_3'}$$

$$\frac{\forall_{0 \leq i \leq n}\, E_i' = \mathcal{T}_e(E_i)}{\mathcal{T}_a(\mathsf{emptyFields}(E_0 \mid E_1, \ldots, E_n) \triangleq \mathsf{emptyFields}(E_0' \mid E_1', \ldots, E_n')}$$

$$\frac{\psi(m, x) = i \neq 0 \quad \mathcal{T}(E_1) = E_1' \quad \mathcal{T}(E_2) = E_2'}{\mathcal{T}_a(\mathsf{scope}(m, x : E_1, E_2)) \triangleq (\mathsf{nth}\,(E_2', i), x) \mapsto E_1'}$$

$$\frac{\psi(m, x) = 0 \quad \mathcal{T}(E_1) = E_1' \quad \mathcal{T}(E_2) = E_2' \quad desc = \{\{\text{“}d\text{”}, E_1', \mathsf{true}, \mathsf{true}, \mathsf{false}\}\}}{\mathcal{T}(\mathsf{scope}(m, x : E_1, E_2)) \triangleq ((l_g, x) \mapsto desc) * (\mathsf{nth}\,(E_2', 0) \doteq l_g)}$$

$$\frac{\mathcal{T}(E_1) = E_1' \quad \mathcal{T}(E_2) = E_2' \quad \psi^o(m, m') = k \quad P_1 = \circledast_{0 \leq i \leq k} (\mathsf{nth}\,(\mathsf{x}_{sc}, i) \doteq \mathsf{nth}\,(E_2', i)) \quad P_2 = (E_1', @code) \mapsto m' * (E_2', @scope) \mapsto E_2'}{\mathcal{T}_a(\mathsf{funObj}(m', E_1, E_2)) \triangleq P_1 * P_2}$$

$$\frac{\psi^o(m_1, m_2) = k \quad \mathcal{T}(E_1) = E_1' \quad \mathcal{T}(E_2) = E_2'}{\mathcal{T}_a(\mathsf{o\text{-}chains}(m_1 : E_1, m_2 : E_2)) \triangleq \circledast_{0 \leq i \leq k} (\mathsf{nth}\,(E_1', i) \doteq \mathsf{nth}\,(E_2', i))}$$

Lemma D.2 (Translation of Logical Expressions - Correctness). *For any three variable stores ρ, ρ', and ρ'', logical environments ϵ and ϵ', location l_t, scope chain L, and function $\beta : \mathcal{L} \rightharpoonup \mathcal{L}$, such that: $\rho \sim_\beta \rho''$, $\rho' = \rho'' \uplus [\mathsf{x}_{sc} \mapsto \beta(L), \mathsf{x}_{\text{this}} \mapsto \beta(l_t)]$, $\epsilon \sim_\beta \epsilon'$, it holds that: $[\![E]\!]^{\epsilon}_{\rho, l_t, L} \sim_\beta [\![\mathcal{T}_e(E)]\!]^{\epsilon'}_{\rho'}$.*

Lemma D.3 (Translation and variables). *It holds that:*

$$\mathsf{vars}(\mathcal{T}(P)) = \mathsf{vars}(P) \ \vee\ \mathsf{vars}(\mathcal{T}(P)) = \mathsf{vars}(P) \cup \{\mathtt{x}_{sc}, \mathtt{x}_{this}\}.$$

Translation of Heaps

Heap Composition
$\langle H_1 \uplus H_2 \rangle_{JSIL} \triangleq \langle H_1 \rangle_{JSIL} \uplus \langle H_2 \rangle_{JSIL}$

Code Property
$\langle (l, @code) \mapsto \lambda \overline{x}.s^m \rangle_{JSIL} \triangleq (l, x) \mapsto m$

Standard Properties
$$\frac{x \neq @code}{\langle (l, x) \mapsto v \rangle_{JSIL} \triangleq (l, x) \mapsto v}$$

Logical Context
$$\frac{\rho' = \rho \uplus [\mathtt{x}_{sc} \mapsto L, \mathtt{x}_{this} \mapsto l_t]}{\langle (H, \rho, L, l_t, \epsilon) \rangle_{JSIL} \triangleq (\langle H \rangle_{JSIL}, \rho', \epsilon)}$$

Heap Composition
$\langle H_1 \uplus H_2 \rangle^s_{JS} \triangleq \langle H_1 \rangle^s_{JS} \uplus \langle H_2 \rangle^s_{JS}$

Code Property
$\langle (l, @code) \mapsto m \rangle^s_{JS} \triangleq (l, x) \mapsto s(m)$

Standard Properties
$$\frac{x \neq @code}{\langle (l, x) \mapsto v \rangle^s_{JS} \triangleq (l, x) \mapsto v}$$

Logical Context
$$\frac{\rho = \rho' \uplus [\mathtt{x}_{sc} \mapsto L, \mathtt{x}_{this} \mapsto l_t]}{\langle (H, \rho, \epsilon) \rangle^s_{JS} \triangleq (\langle H \rangle^s_{JS}, \rho', L, l_t, \epsilon)}$$

Lemma D.4 (JS $\longrightarrow$ JSIL - Logical Contexts). *For any JavaScript logical context* $(H, \rho, L, l_t, \epsilon)$*, it holds that:* $H, \rho, L, l_t, \epsilon \simeq_{\mathsf{id}} \langle H, \rho, L, l_t, \epsilon \rangle_{JSIL}$*, where* $\mathsf{id} : \mathcal{L} \rightharpoonup \mathcal{L}$ *denotes the identity function.*

Proof. Immediate using the definition of $\simeq_{\mathsf{id}}$ for logical contexts. □

Lemma D.5 (JSIL $\longrightarrow$ JS - Logical Contexts). *For any JSIL logical context* (H, ρ, ϵ)*, it holds that:* $\langle H, \rho, \epsilon \rangle^s_{JS} \simeq_{\mathsf{id}} H, \rho, \epsilon$*, where* $\mathsf{id} : \mathcal{L} \rightharpoonup \mathcal{L}$ *denotes the identity function, provided that* $\langle H, \rho, \epsilon \rangle^s_{JS}$ *is defined.*

Proof. Immediate using the definition of $\simeq_{\mathsf{id}}$ for logical contexts. □

(Theorem 7.3 - Assertion translation correctness). *For any two JavaScript and JSIL logical contexts* $(H_{\mathrm{JS}}, \rho, L, l_t, \epsilon)$ *and* $(H_{\mathrm{JSIL}}, \rho', \epsilon')$*, such that* $H_{\mathrm{JS}}, \rho, L, l_t, \epsilon \simeq_\beta H_{\mathrm{JSIL}}, \rho', \epsilon'$*, it holds that:* $H_{\mathrm{JS}}, \rho, L, l_t, \epsilon \models P$ iff $H_{\mathrm{JSIL}}, \rho', \epsilon' \models \mathcal{T}_a(P)$.

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

(1) $P = \mathsf{false}$, $\mathcal{T}_a(P) = \mathsf{false}$. It can never be the case that either $H_{\mathrm{JS}}, \rho, L, l_t, \epsilon \models \mathsf{false}$ or $H_{\mathrm{JSIL}}, \rho', \epsilon' \models \mathsf{false}$ holds. Hence, the result holds vacuously.

(2) $P = \mathsf{true}$. $\mathcal{T}_a(P) = \mathsf{true}$. It is always the case that both $H_{\mathrm{JS}}, \rho, L, l_t, \epsilon \models \mathsf{true}$ and $H_{\mathrm{JSIL}}, \rho', \epsilon' \models \mathsf{true}$ hold, from which the result follows.

(3) $P = \exists X.P'$. $\mathcal{T}_a(P) \triangleq \exists X.\mathcal{T}_a(P')$. Suppose $H_{\mathrm{JS}}, \rho, L, l_t, \epsilon \models P$, we conclude (using satisfiability for JS assertions) that there is a value V, such that $H_{\mathrm{JS}}, \rho, L, l_t, \epsilon[\mathrm{X} \mapsto V] \models P'$. Using the hypothesis, we conclude that $H_{\mathrm{JS}}, \rho, L, l_t, [\mathrm{X} \mapsto V] \simeq_\beta H_{\mathrm{JSIL}}, \rho', \epsilon'[\mathrm{X} \mapsto V']$, where $V \sim_\beta V'$. Applying the induction hypothesis, we conclude that $H_{\mathrm{JSIL}}, \rho', \epsilon'[\mathrm{X} \mapsto V'] \models \mathcal{T}_a(P')$, from which it follows that $H_{\mathrm{JSIL}}, \rho', \epsilon' \models \mathcal{T}_a(P)$.

(4) $P = \mathsf{emp}$. $\mathcal{T}_a(P) = \mathsf{emp}$. Suppose $H_{\mathrm{JS}}, \rho, L, l_t, \epsilon \models P$, we conclude (using the definition of satisfiability for JS assertions) that $H_{\mathrm{JS}} = \mathtt{emp}$. From the definition of $\simeq_\beta$, we conclude that $H_{\mathrm{JSIL}} = \mathtt{emp}$. Using the satisfiability relation for JSIL assertions, we conclude that: $H_{\mathrm{JSIL}}, \rho', \epsilon' \models \mathcal{T}_a(P)$. The converse direction of the equivalence is proven analogously.

(5) $P = (E_1, E_2) \mapsto E_3$. $\mathcal{T}_a(P) = (\mathcal{T}_e(E_1), \mathcal{T}_e(E_2)) \mapsto \mathcal{T}_e(E_3)$. Suppose $H_{\mathrm{JS}}, \rho, L, l_t, \epsilon \models P$, we conclude (using satisfiability for JS assertions) that $H_{\mathrm{JS}} = (\llbracket E_1 \rrbracket^\epsilon_{\rho, l_t, L}, \llbracket E_2 \rrbracket^\epsilon_{\rho, l_t, L}) \mapsto \llbracket E_3 \rrbracket^\epsilon_{\rho, l_t, L}$. From the definition of

$\simeq_\beta$, we conclude that $H_{\text{JSIL}} = (l, m) \mapsto V$, where $l = \beta(\llbracket E_1 \rrbracket^\epsilon_{\rho,l_t,L})$, $m = \llbracket E_2 \rrbracket^\epsilon_{\rho,l_t,L}$, and $\llbracket E_3 \rrbracket^\epsilon_{\rho,l_t,L} \sim_\beta V$. Using Lemma D.2, we conclude that $\llbracket E_i \rrbracket^\epsilon_{\rho,l_t,L} \sim_\beta \llbracket \mathcal{T}_e(E_i) \rrbracket^{\epsilon'}_{\rho'}$, for $i = 1, 2, 3$, from which it follows that $\llbracket \mathcal{T}_e(E_1) \rrbracket^{\epsilon'}_{\rho'} = l$, $\llbracket \mathcal{T}_e(E_2) \rrbracket^{\epsilon'}_{\rho'} = m$, and $\llbracket \mathcal{T}_e(E_3) \rrbracket^{\epsilon'}_{\rho'} = V$. Using the satisfiability relation for JSIL assertions, we conclude that: $H_{\text{JSIL}}, \rho', \epsilon' \models \mathcal{T}_a(P)$. The converse direction of the equivalence is proven analogously.

(6) $P = \mathsf{emptyFields}(E_0 \mid E_1, ..., E_n)$. $\mathcal{T}_a(P) = \mathsf{emptyFields}(E'_0 \mid E'_1, ..., E'_n)$, where $E'_i = \mathcal{T}_e(E_i)$ for $i = 0, ..., n$. Suppose $H_{\text{JS}}, \rho, L, l_t, \epsilon \models P$, we conclude (using satisfiability for JS assertions) that $H_{\text{JS}} = \biguplus_{m \notin M} H(l, m) = \varnothing$, where for $l = \llbracket E_0 \rrbracket^\epsilon_{\rho,l_t,L}$ and $M = \{\llbracket E_i \rrbracket^\epsilon_{\rho,l_t,L} \,|^n_{i=1}\}$. From the definition of $\simeq_\beta$, we conclude that $H_{\text{JSIL}} = \biguplus_{m \notin M} H(\beta(l), m) = \varnothing$. Using Lemma D.2, we conclude that $\llbracket E_i \rrbracket^\epsilon_{\rho,l_t,L} \sim_\beta \llbracket \mathcal{T}_e(E_i) \rrbracket^{\epsilon'}_{\rho'}$, for $i = 0, ..., n$, from which it follows that $\llbracket \mathcal{T}_e(E_0) \rrbracket^{\epsilon'}_{\rho'} = \beta(l)$ and $M = \{\llbracket \mathcal{T}_e(E_1) \rrbracket^{\epsilon'}_{\rho'}, ..., \llbracket \mathcal{T}_e(E_n) \rrbracket^{\epsilon'}_{\rho'}\}$. Using the satisfiability relation for JSIL assertions, we conclude that: $H_{\text{JSIL}}, \rho', \epsilon' \models \mathcal{T}_a(P)$. The converse direction of the equivalence is proven analogously.

(7) $P = \mathsf{scope}(m, x : E_1, E_2)$. There are two cases to consider: either $\psi(m, x) = 0$ or $\psi(m, x) \neq 0$.

- If $\psi(m, x) = 0$, we have that $\mathcal{T}_a(\mathsf{scope}(x : E)) = (l_g, x) \mapsto \{\{\text{“}d\text{”}, E'_1, \mathsf{true}, \mathsf{true}, \mathsf{false}\}\} * (\mathsf{nth}\,(E'_2, 0) \doteq l_g)$, where $E'_1 = \mathcal{T}_e(E_1)$ and $E'_2 = \mathcal{T}_e(E_2)$. Using satisfiability for JS assertions, we conclude that: $H = (l_g, x) \mapsto \{\{\text{“}d\text{”}, \llbracket E_1 \rrbracket^\epsilon_{\rho,l_t,L}, \mathsf{true}, \mathsf{true}, \mathsf{false}\}\}$ and $(\llbracket E_2 \rrbracket^\epsilon_{\rho,l_t,L})(0) = l_g$. From the definition of $\simeq_\beta$, we conclude that $H_{\text{JSIL}} = (l_g, x) \mapsto \{\{\text{“}d\text{”}, \mathsf{v}, \mathsf{true}, \mathsf{true}, \mathsf{false}\}\}$, where $\llbracket E_1 \rrbracket^\epsilon_{\rho,l_t} \sim_\beta \mathsf{v}$ (note that we always assume that $\beta(l_g) = l_g$). Using Lemma D.2, we conclude that $\llbracket E_1 \rrbracket^\epsilon_{\rho,l_t,L} \sim_\beta \llbracket \mathcal{T}_e(E_1) \rrbracket^{\epsilon'}_{\rho'}$ and $\llbracket E_2 \rrbracket^\epsilon_{\rho,l_t,L} \sim_\beta \llbracket \mathcal{T}_e(E_2) \rrbracket^{\epsilon'}_{\rho'}$, from which it follows that $\mathsf{v} = \llbracket \mathcal{T}_e(E_1) \rrbracket^{\epsilon'}_{\rho'}$ and $l_g = (\llbracket \mathcal{T}_e(E_2) \rrbracket^{\epsilon'}_{\rho'})(0)$. From $H_{\text{JSIL}} = (l_g, x) \mapsto \{\{\text{“}d\text{”}, \mathsf{v}, \mathsf{true}, \mathsf{true}, \mathsf{false}\}\}$, $\mathsf{v} = \llbracket \mathcal{T}_e(E_1) \rrbracket^{\epsilon'}_{\rho'}$, and $l_g = (\llbracket \mathcal{T}_e(E_2) \rrbracket^{\epsilon'}_{\rho'})(0)$, we conclude that $H_{\text{JSIL}}, \rho', \epsilon' \models \mathcal{T}_a(P)$. The converse direction of the equivalence is proven using an analogous argument.
- If $\psi(m, x) = i \neq 0$, we conclude that $\mathcal{T}_a(\mathsf{scope}(m, x : E_1, E_2)) = (\mathsf{nth}\,(\mathcal{T}_e(E_2), i), x) \mapsto \mathcal{T}_e(E_1)$. Using satisfiability for JS assertions, we conclude that $H_{\text{JS}} = ((\llbracket E_2 \rrbracket^\epsilon_{\rho,l_t,L})(i), x) \mapsto \llbracket E_1 \rrbracket^\epsilon_{\rho,l_t,L}$. From the definition of $\simeq_\beta$, we conclude that $H_{\text{JSIL}} = (\beta((\llbracket E_2 \rrbracket^\epsilon_{\rho,l_t,L})(i)), x) \mapsto \mathsf{v}$, where $\llbracket E_1 \rrbracket^\epsilon_{\rho,l_t} \sim_\beta \mathsf{v}$. Using Lemma D.2, we conclude that $\llbracket E_1 \rrbracket^\epsilon_{\rho,l_t} \sim_\beta \llbracket \mathcal{T}_e(E_1) \rrbracket^{\epsilon'}_{\rho'}$ and $\llbracket E_2 \rrbracket^\epsilon_{\rho,l_t} \sim_\beta \llbracket \mathcal{T}_e(E_2) \rrbracket^{\epsilon'}_{\rho'}$, from which it follows that $\mathsf{v} = \llbracket \mathcal{T}_e(E) \rrbracket^{\epsilon'}_{\rho'}$ and $\llbracket \mathcal{T}_e(E_2) \rrbracket^{\epsilon'}_{\rho'} = \beta(\llbracket E_2 \rrbracket^\epsilon_{\rho,l_t,L})$. Using the semantics of logical expressions, we conclude that $\llbracket \mathsf{nth}\,(\mathcal{T}_e(E_2), i) \rrbracket^{\epsilon'}_{\rho'} = \beta(\llbracket E_2 \rrbracket^\epsilon_{\rho,l_t,L})(i) = \beta((\llbracket E_2 \rrbracket^\epsilon_{\rho,l_t,L})(i))$. From $H_{\text{JSIL}} = (\beta((\llbracket E_2 \rrbracket^\epsilon_{\rho,l_t,L})(i)), x) \mapsto \mathsf{v}$, $\mathsf{v} = \llbracket \mathcal{T}_e(E_1) \rrbracket^{\epsilon'}_{\rho'}$, and $\llbracket \mathsf{nth}\,(\mathcal{T}_e(E_2), i) \rrbracket^{\epsilon'}_{\rho'} = \beta((\llbracket E_2 \rrbracket^\epsilon_{\rho,l_t,L})(i))$, we conclude that $H_{\text{JSIL}}, \rho', \epsilon' \models \mathcal{T}_a(P) = (\mathsf{nth}\,(\mathcal{T}_e(E_2), i), x) \mapsto \mathcal{T}_e(E_1)$. The converse direction of the equivalence is proven using an analogous argument.

(8) $P = \mathsf{funObj}(m', E_1, E_2)$. $\mathcal{T}_a(P) = \circledast_{0 \leq i \leq k} P_i * (E'_1, @code) \mapsto m' * (E'_1, @scope) \mapsto E'_2$, where $P_i = \mathsf{nth}\,(\mathsf{x}_{sc}, i) \doteq \mathsf{nth}\,(E'_2, i)$, $E'_1 = \mathcal{T}_e(E_1)$, and $E'_2 = \mathcal{T}_e(E_2)$. Using satisfiability for JS assertions, we conclude that: $H_{\text{JS}} = ((l_f, @code) \mapsto s(m')) \uplus ((l_f, @scope) \mapsto L')$ and $\bigwedge_{0 \leq i \leq k} L(i) = L'(i)$, where $k = \psi^o(m, m')$, $\llbracket E_1 \rrbracket^\epsilon_{\rho,l_t,L} = l_f$, and $\llbracket E_2 \rrbracket^\epsilon_{\rho,l_t,L} = L'$. From the definition of $\simeq_\beta$, we conclude that $H_{\text{JSIL}} = (\beta(l_f), @code) \mapsto m' * (\beta(l_f), @scope) \mapsto \beta(L')$. Using Lemma D.2, we conclude that $\llbracket E_1 \rrbracket^\epsilon_{\rho,l_t} \sim_\beta \llbracket \mathcal{T}_e(E_1) \rrbracket^{\epsilon'}_{\rho'}$ and $\llbracket E_2 \rrbracket^\epsilon_{\rho,l_t} \sim_\beta \llbracket \mathcal{T}_e(E_2) \rrbracket^{\epsilon'}_{\rho'}$, from which it follows that $\llbracket E'_1 \rrbracket^{\epsilon'}_{\rho'} = \beta(l_f)$ and $\llbracket E'_2 \rrbracket^{\epsilon'}_{\rho'} = \beta(L')$. From $H_{\text{JSIL}} = (\beta(l_f), @code) \mapsto m' * (\beta(l_f), @scope) \mapsto \beta(L')$, $\llbracket E'_2 \rrbracket^{\epsilon'}_{\rho'} = \beta(L')$, and $\llbracket E'_1 \rrbracket^{\epsilon'}_{\rho'} = \beta(l_f)$, it follows that $H_{\text{JSIL}}, \rho', \epsilon' \models (E', @code) \mapsto m' * (E'_1, @scope) \mapsto E'_2$ **(I1)**.
From the definition of $\simeq_\beta$, we conclude that $\rho'(\mathsf{x}_{sc}) = \beta(L)$. Recalling that $\llbracket E'_2 \rrbracket^{\epsilon'}_{\rho'} = \beta(L')$, we have (using

satisfiability for JSIL assertions) that $H_{\text{JSIL}}, \rho', \epsilon' \models \mathsf{nth}\,(\mathsf{x}_{sc}, i) \doteq \mathsf{nth}\,(\mathrm{X}, i)$ *if and only if* $\beta(L)(i) = \beta(L')(i)$. Since $\bigwedge_{0 \le i \le k} L(i) = L'(i)$, we conclude that: $H_{\text{JSIL}}, \rho', \epsilon' \models \circledast_{0 \le i \le k} \mathsf{nth}\,(\mathsf{x}_{sc}, i) \doteq \mathsf{nth}\,(E_2', i)$ **(I2)**.
The result follows from **I1** and **I2**. The converse direction of the equivalence is proven using an analogous argument (observing that β is injective).

(9) $P = \text{o-chains}(m_1 : E_1, m_2 : E_2)$. $\mathcal{T}_a(P) = \circledast_{0 \le i \le k}(\mathsf{nth}\,(E_1', i) \doteq \mathsf{nth}\,(E_2', i))$, where $\psi^o(m_1, m_2) = k$, $E_1' = \mathcal{T}_e(E_1)$, and $E_2' = \mathcal{T}_e(E_2)$. Using satisfiability for JS assertions, we conclude that: $\bigwedge_{0 \le i \le k} L_1(i) = L_2(i)$, where $L_1 = [\![E_1]\!]^{\epsilon}_{\rho, l_t, L}$ and $L_2 = [\![E_2]\!]^{\epsilon}_{\rho, l_t, L}$. Using Lemma D.2, we conclude that $[\![E_1]\!]^{\epsilon}_{\rho, l_t} \sim_\beta [\![\mathcal{T}_e(E_1)]\!]^{\epsilon'}_{\rho'}$ and $[\![E_2]\!]^{\epsilon}_{\rho, l_t} \sim_\beta [\![\mathcal{T}_e(E_2)]\!]^{\epsilon'}_{\rho'}$, from which it follows that $[\![\mathcal{T}_e(E_1)]\!]^{\epsilon'}_{\rho'} = \beta(L_1)$ and $[\![\mathcal{T}_e(E_1)]\!]^{\epsilon'}_{\rho'} = \beta(L_2)$. Using the injectivity of β, we conclude that $\bigwedge_{0 \le i \le k} L_1(i) = L_2(i)$ holds if and only if $\bigwedge_{0 \le i \le k} \beta(L_1)(i) = \beta(L_2)(i)$, from which the result follows.

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

Definition D.6 (Compiler Correctness). A JS-2-JSIL compiler C is correct if for all JS statements s, JS execution context (h, ρ, L, l_t), JSIL execution context (h, ρ'), function identifier m, and function β, such that: $h, \rho, L, l_t \simeq_\beta \mathsf{h}, \rho'$, it holds that:

$$s, L, l_t \vdash \langle h, \rho \rangle \Downarrow_m \langle h_f, o \rangle \;\Leftrightarrow\; C(s) \vdash \langle \mathsf{h}, \rho', 0 \rangle \Downarrow_m \langle \mathsf{h}_f, \rho_f', o' \rangle$$

and there exists $\beta' \ge \beta$ such that $h_f \simeq_{\beta'} \mathsf{h}_f$, $o \sim_{\beta'} o'$, and $\rho_f' \ge \rho'$.

(Theorem 7.4 - JS-2-JSIL Logic correspondence). *Given a correct JS-2-JSIL compiler C, for every annotated JS program s:* specs(s) *is valid for s iff $\mathcal{T}$(*specs(s)*) is valid for $C(s)$.*

Proof. The fact that dom(specs(s)) $= \mathcal{T}$(specs(s) is an immediate consequence of the definition of the translation. Now, we have to prove that for every function identifier m and return mode fl, it holds that specs$(s)(m, fl)$ is valid if and only if $\mathcal{T}$(specs$(s)(m, fl)$) is valid. Suppose that specs$(s)(m, fl) = \{P\}\, m(\vec{x})\, \{Q\}$, we have to show that: $\{P\}\, m(\vec{x})\, \{Q\}$ is valid if and only if $\{\mathcal{T}(P)\}\, m(\mathsf{x}_{sc}, \mathsf{x}_{this}, \vec{x})\{\mathcal{T}(Q)\}$ is valid. In the following, we prove the left-to-right direction of the equivalence. The other direction is analogous.

Assuming that $\{P\}\, m(\vec{x})\, \{Q\}$ is valid, we need to prove that $\{\mathcal{T}(P)\}\, m(\mathsf{x}_{sc}, \mathsf{x}_{this}, \vec{x})\, \{\mathcal{T}(Q)\}$ is also valid. To this end, we need to show that for every JSIL logical context H, ρ, ϵ such that $H, \rho, \epsilon \models \mathcal{T}(P)$ and $\mathsf{p} \vdash \langle \lfloor H \rfloor, \rho, 0 \rangle \Downarrow_m \langle \mathsf{h}_f, \rho_f, \mathsf{v} \rangle$, for some $\mathsf{h}_f, \rho_f, \mathsf{v}$, there exists an abstract heap H_f such that $\lfloor H_f \rfloor = \mathsf{h}_f$ and $H_f, \rho_f, \epsilon \models \mathcal{T}(Q)$. Hence, assuming that:

- $\{P\}\, m(\vec{x})\, \{Q\}$ is valid **(H1)**
- $H, \rho, \epsilon \models \mathcal{T}(P)$ **(H2)**
- $\mathsf{p} \vdash \langle \lfloor H \rfloor, \rho, 0 \rangle \Downarrow_m \langle \mathsf{h}_f, \rho_f, o \rangle$, for some h_f, ρ_f, o **(H3)**

Our goal is to show that there is an abstract JSIL heap H_f and a JSIL value v such that:

- $\lfloor H_f \rfloor = \mathsf{h}_f$ **(G1)**
- $H_f, \rho_f, \epsilon \models \mathcal{T}(Q)$ **(G2)**
- $o = fl\langle \mathsf{v} \rangle$ **(G3)**

(1) From Lemma D.5, we get that $\langle (H, \rho, \epsilon) \rangle^s_{JS} \simeq_{\text{id}_1} (H, \rho, \epsilon)$, for $\text{id}_1 = \text{id}|_{\text{dom}(H)}$, from which it follows that: $\langle H \rangle^s_{JS}, \rho, L, l_t \simeq_{\text{id}_1} H, \hat{\rho}$, where $\rho = \hat{\rho} \uplus [\mathsf{x}_{sc} \mapsto L, \mathsf{x}_{this} \mapsto l_t]$ **(I1)**.
(2) From **I1**, we conclude, using compiler correctness (Definition D.6) and **H3**, that there exists a JavaScript heap h_f and an outcome $\hat{o}$ such that: $s, L, l_t \vdash \langle \lfloor \langle H \rangle^s_{JS} \rfloor, \hat{\rho} \rangle \Downarrow_m \langle h_f, \hat{o} \rangle$ **(I2.1)**, $h_f \sim_\beta \mathsf{h}_f$ **(I2.2)**, and $\hat{o} \sim_\beta o$ **(I2.3)**, for some $\beta \ge \text{id}_1$ **(I2.4)**.
(3) From **I1**, we conclude, using Theorem 7.3 and **H2**, that $\langle H \rangle^s_{JS}, \hat{\rho}, L, l_t, \epsilon \models P$ **(I3)**.

(4) From **I2.1** and **I3**, we conclude, using **H1**, that there exists an abstract JavaScript heap $\hat{H}_f$ and a JavaScript value v such that $\hat{H}_f, \hat{\rho}, L, l_t, \epsilon \models Q$ (**I4.1**), $\lfloor \hat{H}_f \rfloor = h_f$ (**I4.2**), and $\hat{o} = \mathit{fl}\langle v \rangle$ (**I4.3**). From **I2.3** and **I4.3**, we conclude that there is a value v such that $o = \mathit{fl}\langle v \rangle$ (where $v \sim_\beta \mathsf{v}$), which gives us **G3**.

(5) We now claim that $H_f = \beta(\langle \hat{H}_f \rangle_{JSIL})$ is our witness, which means that we have to show that: $\lfloor H_f \rfloor = \mathsf{h}_f$ (**G1**) and $H_f, \rho_f, \epsilon \models \mathcal{T}(Q)$ (**G2**). We prove the two goals below.

- **G1**. Noting that $\lfloor \beta(\langle \hat{H}_f \rangle_{JSIL}) \rfloor = \beta(\langle \lfloor \hat{H}_f \rfloor \rangle_{JSIL})$ and using **I4.2**, we conclude that: $\lfloor \beta(\langle \hat{H}_f \rangle_{JSIL}) \rfloor = \beta(\langle h_f \rangle_{JSIL})$. Noting that $\hat{H}_f \sim_{\mathsf{id}_2} \langle \hat{H}_f \rangle_{JSIL}$ (Lemma D.4) and using **I4.2**, we conclude that $h_f \sim_{\mathsf{id}_2} \lfloor \langle \hat{H}_f \rangle_{JSIL} \rfloor$ (**I5.1**). Applying Lemma 15(4) to **I2.2** and **I5.1**, we conclude that $\beta(\lfloor \langle \hat{H}_f \rangle_{JSIL} \rfloor) = \mathsf{h}_f$. Finally, observing that $\beta(\lfloor \langle \hat{H}_f \rangle_{JSIL} \rfloor) = \lfloor \beta(\langle \hat{H}_f \rangle_{JSIL}) \rfloor$, we conclude **G1**.
- **G2**. From Lemma D.4, we get that $\hat{H}_f, \hat{\rho}, L, l_t, \epsilon \simeq_{\mathsf{id}_3} \langle \hat{H}_f, \hat{\rho}, L, l_t, \epsilon \rangle_{JSIL}$ (**I6.1**), where $\mathsf{id}_3 = \mathsf{id}|_{\mathsf{dom}(\hat{H}_f)}$ Applying Theorem 7.3 (Assertion translation correctness - left to right) to **I4.1** and **I6.1**, we conclude $\langle \hat{H}_f, \hat{\rho}, L, l_t, \epsilon \rangle_{JSIL} \models \mathcal{T}(Q)$, from which it follows that that: $\langle \hat{H}_f \rangle_{JSIL}, \hat{\rho} \uplus [\mathsf{x}_{sc} \mapsto L, \mathsf{x}_{this} \mapsto l_t], \epsilon \models \mathcal{T}(Q)$ (**I6.2**). Applying Lemma D.1 to (**I6.2**), we conclude that $\beta(\langle \hat{H}_f \rangle_{JSIL}), \beta(\hat{\rho} \uplus [\mathsf{x}_{sc} \mapsto L, \mathsf{x}_{this} \mapsto l_t]), \beta(\epsilon) \models \mathcal{T}(Q)$ (**I6.3**). Noting that all locations in the range of ϵ are in the domain of id_1 and that $\beta \geq \mathsf{id}_1$, we conclude that $\beta(\epsilon) = \epsilon$ (**I6.4**). Analogously, we know that $\beta(\hat{\rho} \uplus [\mathsf{x}_{sc} \mapsto L, \mathsf{x}_{this} \mapsto l_t]) = \hat{\rho} \uplus [\mathsf{x}_{sc} \mapsto L, \mathsf{x}_{this} \mapsto l_t]$ (**I6.5**). From **I6.3** - **I6.5**, we conclude that: $\beta(\langle \hat{H}_f \rangle_{JSIL}), \hat{\rho} \uplus [\mathsf{x}_{sc} \mapsto L, \mathsf{x}_{this} \mapsto l_t], \epsilon \models \mathcal{T}(Q)$ (**I6.6**). From compiler correctness (Definition D.6) and **H3**, we conclude that $\rho_f \geq \rho$ (**I6.7**). Observe that $\mathsf{vars}(\mathcal{T}(Q)) \subseteq \mathsf{vars}(Q) \cup \{\mathsf{x}_{sc}, \mathsf{x}_{this}\}$ (**I6.8**). Furthermore, because $\{P\}\, m(\vec{x})\, \{Q\}$ is a legal JavaScript specification, we conclude that $\mathsf{vars}(Q) \cup \mathsf{vars}(P) \subset \vec{x} = \mathsf{dom}(\hat{\rho})$ (**I6.9**). From **I6.8** and **I6.9**, we conclude that $\mathsf{vars}(\mathcal{T}(Q)) \subseteq \vec{x} \cup \{\mathsf{x}_{sc}, \mathsf{x}_{this}\} = \mathsf{dom}(\rho)$ (**I6.10**). From **I6.6**, **I6.7**, and **I6.10**, we conclude that $\beta(\langle \hat{H}_f \rangle_{JSIL}), \rho_f, \epsilon \models \mathcal{T}(Q)$.

□

(Theorem 7.5 - Proving JS Hoare Triples in JSIL). *For every annotated JS program s, if there is a JSIL logic proof derivation* pd *such that* $C(s), \mathcal{T}(\mathsf{specs}(s)) \vdash \mathsf{pd}$ *for a correct JS-2-JSIL compiler C, then* $\mathsf{specs}(s)$ *is valid for s.*

Proof. Immediate corollary of Theorems 5.1 and 7.4. □