Imperial College London
Department of Computing

An Infrastructure for Tractable Verification of JavaScript Programs

Daiva Naudžiūnienė

September 2017

Supervised by Prof Philippa Gardner

Submitted in part fulfilment of the requirements for the degree of
Doctor of Philosophy in Computing of Imperial College London
and the Diploma of Imperial College London

Declaration

I herewith certify that all material in this dissertation which is not my own work has been properly acknowledged.

Daiva Naudžiūnienė

Copyright Declaration

Abstract

The highly dynamic nature of JavaScript, coupled with its intricate semantics, makes the understanding and development of correct JavaScript code notoriously difficult. We believe that logic-based verification has much to offer to JavaScript. In particular, separation logic has been successfully applied to verification tools for static languages. However, it has hardly been used to reason about programs written in dynamic languages in general, and JavaScript in particular.

This thesis presents JaVerT, a semi-automatic **Ja**vaScript **Ver**ification **T**oolchain for tractable logic-based verification of JavaScript programs. JaVerT verifies JavaScript programs annotated with function specifications in the form of pre- and postconditions, loop invariants, and annotations for the folding and unfolding of user-defined predicates. We design natural JavaScript abstractions that allow JavaScript developers wishing to verify JavaScript programs to not think about almost any internals of the language.

The actual process of how JaVerT verifies the given annotated JavaScript program is not visible to the JavaScript developer, and is achieved using our JSIL verification infrastructure. This infrastructure includes: JSIL, a simple goto language, suitable for logic-based verification of JavaScript; JSIL Logic, a sound separation logic for JSIL; and JSIL Verify, a semi-automatic verification tool, based on JSIL Logic. The joining ingredient of JaVerT is a *JavaScript frontend* to our JSIL verification infrastructure, tightly connecting programs and reasoning at the level of JavaScript to programs and reasoning at the level of JSIL. This frontend includes a well-tested compiler from JavaScript code to JSIL code, a translator from JavaScript Logic to JSIL Logic, and well-tested JSIL reference implementations and verified axiomatic specifications of the JavaScript internal functions.

We demonstrate the feasibility of JaVerT to specify and verify simple data structure libraries, illustrating our ideas using an implementation of a priority queue. Our given specifications ensure *prototype safety* of library operations, in that they describe the conditions under which these operations exhibit the desired behaviour.

‘Have no fear of perfection - you’ll never reach it.’
Salvador Dalí

Contents

List of Figures

1. Introduction

JavaScript is the de facto language for programming client-side web applications. It is developed by the ECMAScript Committee and described by the international ECMAScript standard [1], with which all major web browsers now comply. JavaScript started off as a scripting language for small applications that manipulate the static content of web pages. Today, JavaScript has grown to be one of the most widely used programming languages in the world. It has moved beyond client-side applications and onto server-side platforms and small embedded devices. JavaScript developers have a large number of frameworks and libraries at their disposal, and the applications that they build are of considerable size. The highly dynamic nature of JavaScript, coupled with its intricate semantics, makes the understanding and development of correct JavaScript code notoriously difficult.

This thesis presents logic-based verification of JavaScript programs, by pulling together a large amount of work on operational semantics, compilers, and separation logic. Even though much of this work was previously developed for static languages, the application to the dynamic and complex language that is JavaScript has not been straightforward and brought about a number of significant challenges. To specify JavaScript programs, the challenge **(C1)** is to design assertions that fully capture the common heap structures of JavaScript, such as prototype chains for modelling inheritance and the variable store emulated in the heap. Importantly, these assertions should abstract as much as possible from the details of the heap structures they describe. To verify JavaScript programs, the challenge is to handle the sheer complexity of the JavaScript semantics, due to: **(C2)** the behaviour of JavaScript statements, which exhibit complicated control flow with several breaking mechanisms and ways of returning values; **(C3)** the fundamental dynamic behaviour associated with extensible objects, dynamic property accesses, and dynamic function calls; and **(C4)** the JavaScript internal functions, which underpin the JavaScript statements and whose definitions in the ECMAScript standard are operational, intricate, and intertwined.

Symbolic verification has recently become tractable for C and Java, with compositional techniques that scale and properly engineered tools applied to real-world code: for example, Facebook's Infer for C, C++, Objective C, and Java [14], based on separation logic; Java Pathfinder, a model checking tool for Java bytecode programs [71]; CBMC, a bounded model checker for C, currently being adapted to Java at Amazon [42]; and WALA's analysis for Java using the Rosette symbolic analyser [27]. Such tools often use intermediate goto representations in order to unravel the control flow constructs of their target language, thus simplifying their analyses. These representations can help us tackle our second challenge **(C2)**, however, they are not inherently suitable for dynamic languages such as JavaScript, and fall short for our third challenge **(C3)**.

We believe that separation logic has much to offer JavaScript as it provides a natural way to reason modularly about the JavaScript heap. Previous work introduced a separation logic for a small fragment of JavaScript [30] and showed that separation logic can be used to reason about the JavaScript variable store emulated in the heap. This logic considers simplified JavaScript semantics. For instance, it does

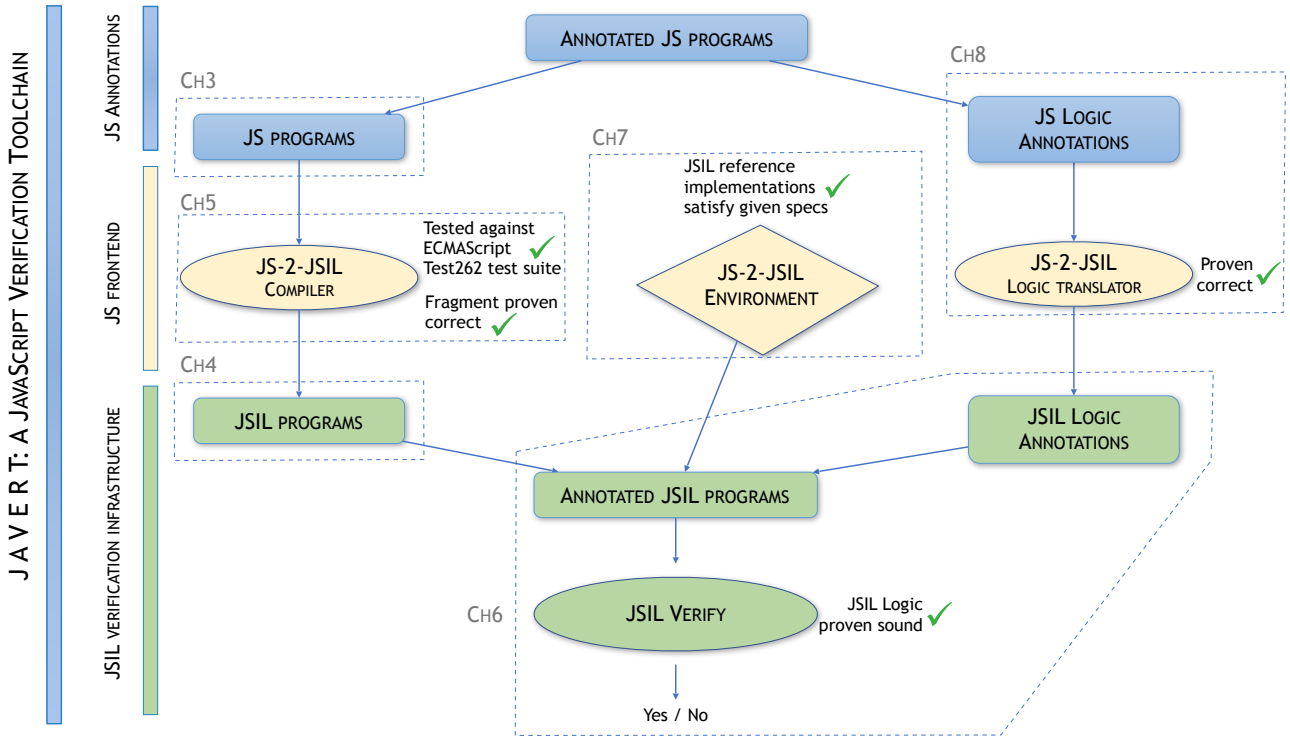


Figure 1.1.: JaVerT: A JavaScript Verification Toolchain

not take into account getters and setters, descriptors, implicit coercions, or any of the intricate control flow commands of the language. Even in such a simplified setting, this proposed logic is very complex due to the (remaining) complexity of JavaScript. We did build a prototype tool, JuS [29], based on the logic, and were able to semi-automatically reason about very simple programs that manipulate the JavaScript variable store. Even though this work partly solves our first challenge **(C1)**, it is not feasible to extend the program logic of JavaScript to the full language. In particular, one would have to re-develop a comprehensive logic with a great many number of ad hoc axioms, which would be extremely difficult to prove sound, let alone automate. We believe that working directly with JavaScript is too complex in the context of verification and we contend that the correct approach is to first translate JavaScript programs to a simpler language and only then to apply automatic reasoning techniques. With such an approach, we would move a substantial part of JavaScript complexity from the logic to the translation.

To solve our four challenges **(C1)**-**(C4)**, we introduce JaVerT, a semi-automatic **Ja**vaScript **Ver**ification **T**oolchain based on separation logic. In particular, we consider a strict mode of the fifth version of the standard (ES5), which we call *ES5 Strict*. The ECMAScript committee developed the strict mode intentionally opting for a different semantics that exhibits better behavioural properties. Albeit somewhat simpler than full ES5, ES5 Strict retains a high level of complexity.

The structure of JaVerT is given in Figure 1.1 and can be divided into three parts. The first part of JaVerT addresses the specification of JavaScript programs, which solves our first challenge **(C1)**. To verify JavaScript programs, we first need to specify them, which we do by placing appropriate annotations into the code. These annotations take the form of pre- and postconditions, loop invariants, as well as instructions for folding and unfolding predicates, and are written in an assertion language in the style of separation logic, which we call JS Logic. We show how to develop natural JavaScript

abstractions that make reasoning using JaVerT nearly as simple as reasoning about Java programs using a semi-automatic verification tool such as VeriFast [36]; any additional complexity stems from the behaviour of JavaScript programs, and not from our reasoning. A key idea is that JavaScript developers wishing to verify JavaScript programs would only need to know how to use these abstractions and would not need to think about any internals of the language. Given an annotated JavaScript program, JaVerT indicates whether the JavaScript program satisfies its specification by yielding a yes/no output. The actual process of how JaVerT verifies the given annotated JavaScript program is not visible to the JavaScript developer.

The second part of JaVerT is our JSIL verification infrastructure. This infrastructure includes: JSIL, a simple goto language that we have developed, and which is suitable for symbolic verification of JavaScript; JSIL Logic, a sound separation logic for JSIL; and JSIL Verify, a semi-automatic verification tool for JSIL, based on JSIL Logic. Given a JSIL program and its specification, written using annotations in JSIL logic, JSIL Verify checks if the JSIL program satisfies its specification and provides a yes/no answer. The annotated JSIL program can be both compiled from a given annotated JavaScript program or written directly in JSIL. The JSIL verification infrastructure solves our third challenge **(C3)**, as JSIL retains the fundamental dynamic behaviour of JavaScript associated with extensible objects, property accesses and function calls.

The final ingredient of JaVerT is a *JavaScript frontend* to our JSIL verification infrastructure, tightly connecting programs and reasoning at the level of JavaScript to programs and reasoning at the level of JSIL. This frontend includes the JS-2-JSIL compiler, a well-tested compiler from JavaScript code to JSIL code; the JS-2-JSIL logic translator, a translator from JS Logic to JSIL Logic; and JS-2-JSIL *environment*, well-tested JSIL reference implementations and verified axiomatic specifications of the JavaScript internal functions. To verify an annotated JavaScript program, the program is first compiled to a JSIL program using the JS-2-JSIL compiler and its annotations are translated to JSIL annotations using the logic translator. The obtained annotated JSIL program, together with the JS-2-JSIL environment, is then fed to JSIL Verify, which produces the final yes/no answer. Instead of reasoning directly about code built from complex JavaScript statements, we use JS-2-JSIL to reason about compiled JSIL code built from simple JSIL statements. This solves our second challenge **(C2)**. We solve our final, fourth challenge **(C4)** by providing the JS-2-JSIL environment.

An important part of our project has been the validation of JaVerT. We validate JSIL verification infrastructure by proving our JSIL Logic sound with respect to our JSIL operational semantics. We also validate all three parts of our JavaScript frontend. The JS-2-JSIL compiler is step-by-step faithful to the ECMAScript standard and is systematically tested against the official ECMAScript test suite, passing 100% of the appropriate tests. Moreover, we prove the JS-2-JSIL compiler correct for a fragment of ES5 Strict. We validate the JS-2-JSIL logic translator by establishing a full correctness result for the assertion languages, and a partial correctness result for the program logics, which depends on the correctness of the full JS-2-JSIL compiler. Finally, the JSIL reference implementations of the JS-2-JSIL environment are step-by-step faithful to the standard and are verified with respect to their JSIL specifications using JSIL Verify.

1.1. Contributions

The main result of the thesis is JaVerT, the semi-automatic JavaScript Verification Toolchain for tractable symbolic verification of JavaScript programs, based on separation logic. To achieve this, we note these four contributions:

1. *Natural JavaScript Abstractions*, that describe common JavaScript heap structures, such as prototype chains and the variable store emulated in the heap, without exposing internals of JavaScript.

2. *Validated JSIL Verification Infrastructure*, which includes JSIL, a simple goto language, retaining the fundamental dynamic behaviour of JavaScript associated with extensible objects, property accesses and function calls; a sound program logic for JSIL that handles this dynamic behaviour; and JSIL Verify, a semi-automatic JSIL verification tool based on JSIL logic. One limitation of the JSIL logic is that it does not support higher-order reasoning.

3. *Validated JavaScript Frontend*, which includes

   a) *the JS-2-JSIL Compiler* from JavaScript to our simple intermediate goto language JSIL. We design the JS-2-JSIL compiler so that it closely follows ES5 Strict. We implement the entire core language (except the indirect eval, which by default exits strict mode), as well as the built-in libraries that are strongly intertwined with the core language. We substantially test the JS-2-JSIL compiler using the official ECMAScript test suite. Moreover, we prove that the JS-2-JSIL compiler is correct with respect to our operational semantics of a representative fragment of ES5 Strict. We note that, currently, the JS-2-JSIL compiler requires the entire program in order to compile it to JSIL.

   b) *the JS-2-JSIL Logic Translator* from JS Logic to JSIL Logic. We validate the JS-2-JSIL logic translator by establishing a full correctness result for the assertion languages, and a partial correctness result for the program logics. The full correctness for the program logics would require to prove the correctness of the JS-2-JSIL compiler for the full ES5 Strict.

   c) *the JS-2-JSIL Environment*, containing reference implementations and axiomatic specifications for JavaScript internal functions. The specifications directly benefit JaVerT, since the verification of JavaScript code only needs to use the specifications, not the underlying implementations. The specifications of the JS-2-JSIL environment are validated by verifying that they are satisfied by their well-tested corresponding JSIL reference implementations.

4. *JavaScript Verification*, achieved using JSIL verification infrastructure via JavaScript frontend. JaVerT verifies functional correctness properties of JavaScript programs annotated with pre- and postconditions, loop invariants, and instructions for folding and unfolding predicates. JaVerT specifications are written using JS Logic, which features a number of natural JavaScript abstractions. As JaVerT is a semi-automatic verification tool, we believe its target should be critical JavaScript code, such as JavaScript libraries describing frequently used data structures. For such libraries, we give specifications that ensure prototype safety of library operations, in that they describe the conditions under which these operations exhibit the desired behaviour.

1.2. Thesis Outline

We start with the background theory in §2, where we cover: literature on different flavours of program analysis for JavaScript; provide a background theory on existing operational semantics for JavaScript; discuss a rich landscape of existing intermediate representations for JavaScript; and discuss existing verification tools based on separation logic for other programming languages.

In the main body of the thesis (§3 - §8), we describe the verification of JavaScript programs using JaVerT, illustrating our reasoning with a JavaScript implementation of a priority queue. We conclude in §9. What follows is a detailed outline of the main body of the thesis, also illustrated in Figure 1.1.

§3 The JavaScript Language. We describe the JavaScript language with its key concepts and explain why we choose to work with ES5 Strict. Using small examples, we highlight difficult parts of JavaScript, such as prototype-based inheritance, the lack of encapsulation in the presence of prototype chains, the variable store emulated in the heap, the confusing `this`, tricky type conversions, dynamic property accesses and dynamic function calls. We present a priority queue implementation as our running example and describe the behaviour of ES5 Strict constructs in more detail. We use this example to showcase the major challenges that need to be addressed before JavaScript programs can be verified. We formally define the memory model of full ES5 Strict, and introduce a representative fragment of ES5 Strict with its operational semantics, illustrating the complexity of the language. Our operational semantics was inspired by the operational semantics of JSCert [9], the recent Coq specification of the ES5 standard. We use the memory model to prove a full correctness result of the translation from the JavaScript assertion language to the JSIL assertion language in §8. Moreover, we use the operational semantics to prove the correctness of the formally defined part of the JS-2-JSIL compiler in §5.

§4 The JSIL Language. Many tools based on symbolic analysis [4, 22, 36, 13, 14, 42, 27] target intermediate goto representations in order to fully dismantle the control flow constructs of their target language, thus simplifying their analyses. These representations, however, are not suitable for dynamic languages such as JavaScript, which require extensible objects, dynamic properties and dynamic function calls.

For this reason, we have developed an intermediate goto language, JSIL, which we believe to be well-suited for logic-based verification of JavaScript programs. JSIL has only a small number of commands and a simple operational semantics with no corner cases or unexpected behaviours. We purposefully design JSIL so that its memory model subsumes the memory model of JavaScript to be able easily relate functional properties of JavaScript programs with their translated JSIL programs.

We describe an operational semantics of JSIL commands and provide an example of a JSIL procedure. We use the JSIL operational semantics to prove the correctness of the JS-2-JSIL compiler in §5 and to prove the soundness of the JSIL logic in §6.

§5 The JS-2-JSIL Compiler. The JS-2-JSIL compiler from JavaScript to JSIL targets ES5 Strict. It closely follows our operational semantics for ES5 Strict. As in JSCert, we follow the ECMAScript standard step-by-step by using the pretty-big-step style of semantics [15]. This means that the structure of a compiled JSIL program directly reflects the description of the behaviour of the original JavaScript program in the English standard.

We cover a large and fully representative fragment of ES5 Strict. In doing so, the memory model

is not simplified in any way. We implement the entire core of ES5 Strict which includes language constructs, such as expressions, statements, and internal functions. The only exception is indirect eval, which by default exits strict mode. JavaScript has a collection of built-in libraries. While most built-in libraries provide additional functionality to the core of JavaScript, some of them are strongly intertwined with the core language. We cover the built-in libraries intertwined with the core language.

We systematically test the JS-2-JSIL compiler against the new ECMAScript 6 Test262 test suite, which organises tests by feature. This enables us to provide a more fine-grained analysis than was previously possible. We identify 10469 tests relevant for ES5 Strict and 8797 tests relevant for the JS-2-JSIL compiler coverage, of which we pass 100%.

We designed the JS-2-JSIL compiler so that that there is a simple correspondence between JavaScript and JSIL heaps, and a step-by-step connection to the standard. This allows us to define a straightforward correctness condition for the JS-2-JSIL compiler. We give a correctness proof using our formal ES5 Strict operational semantics of the fragment, defined in §3. The full result would require a substantial mechanised proof development.

§6 JSIL Verification Infrastructure. We present JSIL verification based on separation logic. We introduce the JSIL assertion language that we use to write specifications for JSIL programs. To verify that a JSIL program satisfies its specification, we use our JSIL program logic. We prove JSIL logic to be sound with respect to the operational semantics of JSIL, presented in §4.

JSIL Verify, a semi-automatic verification tool for JSIL, is based on JSIL logic. The frame inference problem that JSIL Verify has to solve is more complex than those featured in tools, based on separation logic, for static languages such as C and Java. Namely, as JSIL features dynamic property access, the property of a cell assertion is an arbitrary logical expression and not a concrete string. This makes symbolic evaluation of object management commands non-trivial.

To verify JSIL programs, we provide pre- and postconditions for functions, loop invariants and fold/unfold directives for user-defined inductive predicates. Developers wishing to verify JavaScript programs will not need to use JSIL Verify directly. We, however, use it to verify JSIL specifications of JavaScript internal functions.

§7 The JS-2-JSIL Environment. The JS-2-JSIL environment includes specifications and JSIL reference implementations of JavaScript internal functions. JavaScript internal functions describe the fundamental inner workings of the language, such as prototype chain traversal (`GetProperty`), or property definition (`DefineOwnProperty`) and deletion (`DeleteProperty`). They are not accessible by the developer, but are called internally by all JavaScript commands. Their definitions in the standard are complex, are given operationally, and are often intertwined, making it difficult for the user to fully grasp the control flow and allowed behaviours.

To be able to verify JavaScript programs we need to provide JSIL *axiomatic specifications* of the internal functions. In creating these specifications, we leverage on a number of JavaScript-specific abstractions built on top of JSIL Logic, which make the specifications much more readable than the operational definitions of the standard. The remaining complexity arises from the internal functions themselves, not our reasoning.

Our JSIL reference implementations of the JS-2-JSIL environment closely follow ECMAScript standard and are substantially tested by the testing of the JS-2-JSIL compiler. Moreover, using JSIL Verify, we prove that these implementations satisfy their axiomatic specifications. These proofs can be seen

both as further validation of the implementations of the environment as well as validation of the JSIL axiomatic specifications themselves.

§8 JavaScript Verification. The verification workflow of JaVerT includes: compiling the annotated JavaScript program to JSIL using the JS-2-JSIL compiler; translating JavaScript annotations using the JS-2-JSIL logic translator to equivalent JSIL annotations; and automatically verifying the resulting annotated JSIL program with JSIL Verify, making use of the verified JS-2-JSIL environment.

We formally introduce the JavaScript assertion language for writing JavaScript specifications and formally define the JS-2-JSIL logic translator from JavaScript assertions to JSIL assertions. We validate the JS-2-JSIL logic translator by establishing a full correctness result for the assertion languages, and a partial correctness result for the program logics; the latter result is partial, because it depends on the correctness of the JS-2-JSIL compiler, which we prove for a fragment of ES5 Strict.

To specify JavaScript programs, we design abstractions that capture its key heap structures, allowing the user to write clear and succinct specifications with minimal knowledge of the JavaScript internals.

We demonstrate JaVerT by providing specifications for our JavaScript priority queue library. Our library is written in object-oriented style, use prototype-based inheritance and function closures. This example illustrates the importance of abstractions in specifying JavaScript programs.

Collaboration. Chapter §3 introduces the semantics of a fragment of ES5 Strict, which is an adaptation of a collaborative work, JSCert, published in [9]. We drew inspiration from the JSCert mechanised specification of JavaScript, which provides operational semantics for full ES5, adapting and streamlining the rules of JSCert to ES5 Strict. Chapters §4 - §8 are based on the collaboration with J. Fragoso Santos, P. Gardner, P. Maksimović, and T. Wood. Specifically, Chapter §5 includes joint work with J. Fragoso Santos and P. Maksimović on extending the JS-2-JSIL compiler to its current coverage of ES5 Strict; in particular, for supporting property descriptors and the full implementations of internal and built-in functions. These additions have also impacted the design of the JSIL language (§4). Section §5.3 is based on the work done in collaboration with T. Wood, who created the testing and filtering infrastructure. Chapter §6 includes ideas from the previous work done by P. Gardner and G. Smith [30], such as abstract heaps and the assertions describing non-existent properties of an object. JSIL verification infrastructure (§6) was designed and implemented in collaboration with J. Fragoso Santos and P. Maksimović.

1.3. Publications

JuS: Squeezing the sense out of JavaScript programs. P. Gardner, D. Naudžiūnienė, G. Smith. Second Annual Workshop on Tools for JavaScript Analysis, 2013.

A Trusted Mechanised JavaScript Specification. M. Bodin, A. Charguéraud, D. Filaretti, P. Gardner, S. Maffeis, D. Naudžiūnienė, A. Schmitt, G. Smith. POPL 2014.

Towards Logic-Based Verification of JavaScript Programs. J. Fragoso Santos, P. Gardner, P. Maksimović, D. Naudžiūnienė. CADE 2017.

JaVerT: JavaScript Verification Toolchain. J. Fragoso Santos, P. Maksimović, D. Naudžiūnienė, T. Wood, P. Gardner. POPL 2018.

2. Background Theory

This thesis pulls together a large amount of work on operational semantics, compilers, and separation logic. Much of this was previously developed for static languages. The application of this work to the dynamic and complex language that is JavaScript has not been straightforward.

There is a wide range of literature covering different flavours of static program analysis for JavaScript, which we briefly cover (§2.1). However, we note that there is little work done for logic-based symbolic analysis of JavaScript programs. Since our aim is to develop a logic-based verification tool for JavaScript, we focus our discussion on operational semantics (§2.2), compilers and intermediate representations for JavaScript (§2.3), as well as verification tools based on separation logic (§2.4).

2.1. Static Program Analysis for JavaScript

The existing literature covers a wide range of analysis techniques for JavaScript programs, including: type systems [67, 2, 37, 17, 45, 24, 48, 26, 6, 56] and abstract interpretation [41, 37, 3, 51], among others. In contrast, there has been comparatively little work on logic-based verification tools of JavaScript programs.

Formal Type Systems for Fragments of JavaScript. There has been much research on type safety for JavaScript dating back to the seminal work of Thiemann [67]. Thiemann [67] was the first to propose a type system for a fragment of JavaScript (ECMAScript 3). The work provides a type soundness proof with respect to an operational semantics, but it does not give a type inference algorithm. Around the same time, Anderson et al. [2] designed a type inference algorithm for an idealised version of JavaScript that allows objects to evolve. This work included a theorem that this type inference algorithm is sound with respect to their type system.

Experimental Tools for Type Analysis. Jensen et al. [37] presented the first tool for type analysis in real JavaScript (ECMAScript 3) code, called TAJS. TAJS is a flow- and context-sensitive analysis based on abstract interpretation. It performs points-to analysis as part of the type analysis and is fully automatic. Experiments were done on small and medium size JavaScript programs. One of the main objectives for TAJS, as claimed by the authors, has been soundness. However, no theorems or proofs are given in the paper.

Rich types systems, such as [17, 45] emerged, while sacrificing full automation. Chugh et al. present dependent types for JavaScript in [17]. The work introduces a static type system, based on nested refinements and alias types, for a large subset of JavaScript, called DJS. DJS is desugared to System !D for type checking. The expressiveness of the type system is evaluated on small JavaScript benchmark programs. The authors claim that System !D is sound, but no proof of soundness is given. The correctness of the desugaring is not discussed either. In [45], the authors introduce a framework for building type analyses for JavaScript, called TeJaS, as the authors claim that a single type system

cannot accommodate the broad variety of JavaScript features. The parameterisable type system provides more flexibility in their evaluation, hence their benchmarks are larger compared to [17]. The authors claim that their Base Type system is sound. However, no soundness proof is provided. Moreover, their goal is to support different type systems, including unsound ones.

Fully Fledged Type Analysis Tools in Industry. In industry, the best known examples are Flow [24] from Facebook and TypeScript from Microsoft [48]. TypeScript and Flow have influenced each other and their basic typing mechanisms are very similar. However, Flow has a more expressive type system in general. As far as we are aware, there are almost no publications for Flow. Hence, we concentrate our discussion on TypeScript. The TypeScript programming language [48] was proposed as a flexible language that adds optional types to JavaScript language. TypeScript programs can be trivially compiled to JavaScript programs. In fact, every JavaScript program is also a TypeScript program. The main idea of this language is to harness the flexibility of real JavaScript, while at the same time providing some of the advantages otherwise reserved for statically typed languages, such as informative compiling errors and automatic code completion. Several type systems [6, 26, 56] have been proposed for verifying different flavours of safety properties for TypeScript programs. Bierman et al. [6] were the first to formalise a fragment of TypeScript with the goal of characterising both its safe and unsafe parts, thereby establishing a basis for a principled study of deliberate unsoundness. Almost simultaneously, Feldthaus et al. [26] formalised a safe fragment of TypeScript in order to check the correction of TypeScript declaration files with respect to JavaScript library implementations. Recently, Rastogi et al. [56] designed and implemented a new gradual type system for safely compiling TypeScript to JavaScript. The soundness of the proposed type system is guaranteed by combining strict static checks with residual runtime checks that are inlined into the compiled code. Finally, Vekris et al. [70] study a refinement type system, which enable static verification of TypeScript programs. Their system is able to specify not only refinements, but also value-dependent properties, such as the safety of array accesses. The authors develop a flow-sensitive reasoning by translating input programs to an equivalent intermediate SSA form and prove soundness of the type checking for the intermediate form.

Fully Fledged Type Analysis Tools in Academia. In academia, JSAI [41], TAJS [3], and SAFE [43, 16, 51] are the state-of-the-art type inference tools for JavaScript. They do not require any annotations and are highly automated. However, they are whole-program analyses, and are not suitable for verifying partial programs. These tools incorporate different techniques as part of their analyses. Kashyap et al. describe a formally specified abstract interpreter for JavaScript (ECMAScript 3), called JSAI [41]. It combines pointer analysis, control flow analysis, and different data flow analyses and caters for configurable context, path and heap sensitivities. The authors claim to have soundness proof sketches for some of their analyses. Andersan et al. combine data flow analysis and pointer analysis in their typing analysis tool, TAJS [3], whereas most of the previous tools were keeping them separate. The analysis makes use of selective context and path sensitivity, constant propagation, and branch pruning to obtain better precision. The authors do not provide a soundness proof for TAJS. Park et al. present LSA, Loop-Sensitive Analysis, to improve scalability by providing better precision in loops [51], which they implement in their tool SAFE. The authors formalise LSA using abstract interpretation and prove its soundness in Coq.

A Practical Tool or a Sound Tool. Some of the above tools [3, 51, 44] attempt to analyse

JavaScript libraries, especially jQuery [38]. TAJS [3], SAFE [51], and TeJaS [44] can handle jQuery at different levels. Most of these tools do not come with a proof of soundness and some are even deliberately unsound. Madsen et al. argue that unsoundness is a way to go for a practical tool, especially in the presence of library code [46]. Even though it is difficult to obtain soundness, there exist practical tools that do include sound components. For example, the LSA mechanism of SAFE was formally proven sound. Moreover, Bodin et al. [10] argue that it should be possible to develop certified tools even for large scale projects, such as JSCert [9]. The methodology presented in [10] investigates the development of certified abstract interpreters from operational semantics for a small imperative language, but the authors plan to extend it to JSCert.

Logic-based Verification Tools for JavaScript. There is very little work on logic-based verification tools for JavaScript. Indeed, we are only aware of HOO by Cox et al. [18] and KJS by Ştefănescu et al. [19].

Using abstract interpretation and separation logic, Cox et al. [18] have shown how to specify property iteration, focusing on a simplified version of the JavaScript `for−in` statement.

KJS [52] is a tested executable semantics of JavaScript in the $\mathbb{K}$ framework [58]. It comes with a symbolic execution engine [19] and can thus be used for formal analysis and verification of JavaScript programs, with specifications written in the reachability logic of $\mathbb{K}$ [59]. The authors have used KJS to verify functional correctness properties of operations for manipulating data structures such as binary search trees, AVL trees, and lists. These examples, however, do not address the majority of critical JavaScript-specific features, including dynamic property access, prototype inheritance and function closures. The authors argue that it is impractical: **(1)** to have a different semantics (for example, separation logic) for the language and prove the correspondence; **(2)** to have an intermediate verification language with a translator, as it usually contains errors and cannot be tested.

Our approach is entirely different. First, we argue that it is important to have abstractions for specifying JavaScript programs. We create layers of abstractions, allowing the user to write specifications with only a minimal knowledge of the JavaScript internals. In contrary, as KJS has operational semantics as a basis for verification, their examples contain no JavaScript-specific abstractions. A user thus has to consider all of the internals of JavaScript in order to specify JavaScript code, making the specification difficult and error-prone.

Second, we believe that having an intermediate language is an advantage. Our intermediate language is executable and programs translated from JavaScript can be tested. We also prove that the fragment of our compiler is correct. Moreover, to implement a different verification tool for KJS, one would have to consider all of the operational rules of KJS. A small intermediate language simplifies the implementation of the verification tool, as the underlying logic is usually much simpler.

2.2. Operational Semantics

JavaScript is defined by the international ECMAScript standard [1], with which all major web browsers comply. Maffeis et al. [47] were the first to formalise the semantics of JavaScript, a large subset of ECMAScript 3, developing a hand-written, small-step operational semantics. The goal of the formal semantics was to cover the entire language and serve as a basis for formal proofs of real language properties, which influenced the definition of further JavaScript formalisations.

For the next version of ECMAScript, ES5, more formalisations of operational semantics have been proposed, including the semantics specialised for security analysis [33, 63], and more general mechanised semantics [9, 52], making the formal proofs more manageable. Hedin et al. [33] provide a big-step operational semantics of ES5, instrumented with information flow checks for dynamic security type checking. For ES5, the ECMAScript committee introduced a restricted variant, ES5 Strict, that intentionally has slightly different semantics compared with the full language, and exhibits better behavioural properties, such as lexical scoping and better error checking. Taly et al. [63] propose a small-step semantics for a fragment of ES5 Strict suitable for security analysis that they call SES_{light}. There is an issue of variable scope modelling in their formalisation. In particular, the semantics does not propagate updates to variables that are not in the immediate scope of the function currently executing. This occurs because environment records are not stored in the heap. Having general-purpose analyses in mind, Bodin et al. developed JSCert [9], a Coq mechanised specification of the ES5 standard. Park et al. developed KJS [52], a mechanised specification of JavaScript in the $\mathbb{K}$ framework [58]. JSCert closely follows the ES5 standard and provides executable semantics for testing against the ECMA conformance test suite. Since Bodin et al. separates the operational semantics, JSCert, and the executable semantics, JSRef, the authors also provide the Coq proof, stating that JSRef is correct with respect to JSCert. KJS relates to JSRef as it provides executable semantics for testing.

We focus on ES5 Strict, as described in the ES5 English standard. The strict mode is not confined to a particular chapter, but is described via notes throughout the standard. We draw inspiration from the JSCert mechanised specification of JavaScript, which provides pretty-big-step [15] operational semantics for the full ES5, by adapting the rules of JSCert to reflect ES5 Strict restrictions. In this way, we maintain the same high level of correspondence between our operational semantics and the English specification.

In the recent years, the ECMAScript committee has published new versions of the standard every year. ECMAScript 6 (ES6), released in June 2015, is what the most of the browsers now support.[1] ES6 is based on ES5, released in 2011, mostly extending the language with new features and with some minor changes to the semantics.[2] The latest version of the language is ECMAScript 7, released in June 2016. We are not aware of the work that fully formalises the new versions of the standard.

Pretty-Big-Step Semantics. We justify our JS-2-JSIL compiler in part by proving a correctness result with respect to our operational semantics of ES5 Strict fragment. As we provide pretty-big-step operational semantics, we describe this style of semantics in more detail. We use forward references to our formalisation of ES5 Strict fragment (§3.4.2).

The pretty-big-step operational semantics was developed by Charguéraud [15]. The key difference between the traditional big-step semantics and the pretty-big-step semantics is that we can decompose the evaluation of a single program construct using intermediate forms, which extend the grammar of program statements and can be evaluated just like any other program. This style of semantics allows us to more effectively match the modularity of the ES5 standard, which we illustrate using an assignment $e_1 = e_2$ in §3.4.3. Notice that additionally to an assignment construct $e_1 = e_2$, we use three intermediate forms $o =_1 e$, $w =_2 o$, and $o =_3 v$, where e is a JavaScript expression; o is an

[1] https://kangax.github.io/compat-table/es6/

[2] One such change is that the `length` property of function objects is configurable in ES6 and non-configurable in ES5.

outcome of evaluation, including errors; w and v are an outcome value and a value, respectively.

ES5 use sentences of the form "Let R be the result of evaluating t" (see Figure 3.14). These sentences relate a term directly to its result, just as a big-step judgement would do. Because we want to be close to ES5, we cannot work with a small-step presentation, with rules of the form "To evaluate $e_1 = e_2$, execute one step to reduce e_1 into e_1', and then evaluate $e_1' = e_2$." If we attempt to use traditional big-step semantics, we quickly find that we have to duplicate a significant amount of material across several rules. For an assignment formalisation using big-step rules, we would need additional rules to handle exceptional cases, since e_1, e_2, `GetValue`, and `PutValue` can all evaluate to an exception. The problem is that the big-step semantics makes steps "too big" to correspond to ES5. As suggested by our example, if we attempt to use a big-step presentation for ES5, our repetition of premises will lead to an explosion in the size of our rule set. Whereas using pretty-big-step style, we are able to have a single rule for propagating exceptions.

2.3. Compilers and Intermediate Representations for JavaScript

There exists a rich landscape of intermediate representations (IRs) for JavaScript. We can broadly divide these IRs into two categories: **(1)** those for syntax-directed analyses, following the abstract syntax tree of the program, such as λ_{JS} [32], S5 [55], and notJS [41]; and **(2)** those for analyses based on the control flow graph of the program, such as JSIR [39], IRs of WALA [61] and TAJS [37, 3]. SAFE [43], an analysis framework for JavaScript, provides IRs in both categories. The IRs in **(1)** are normally well-suited for high-level analysis, such as type-checking/inference [32, 55], whereas those in **(2)** are generally the target of separation-logic tools [4, 22, 36, 13, 14], and tools for tractable symbolic evaluation [42, 12]. We believe that an IR for logic-based JavaScript verification should belong to the latter category.

Our goals for JSIL were to: natively support the fundamental dynamic features of JavaScript, namely extensible objects, dynamic property accesses, and dynamic function calls; have JSIL heaps be identical to JavaScript heaps, to keep our correctness proofs simple; and keep JSIL minimal to simplify JSIL logic. For control flow, JSIL has only conditional and unconditional `goto` statements. Having gotos in an IR for JavaScript verification is sensible, for three reasons: first, verification tools, based on separation logic, commonly have goto IRs; second, JavaScript has complex control flow statements with many corner cases (for example, switch or try/catch/finally), which can be naturally decompiled to gotos; third, JavaScript supports a restricted form of goto statements, via labelled statements, breaks, and continues. We have *only* gotos because we have so far not encountered the need for more structured loops: our invariants are always JavaScript assertions; and the JavaScript internal functions implemented in JSIL use only simple loops.

When it comes to the IRs belonging to **(2)**, JSIL is similar to JSIR [39], and the IRs of WALA [27] and TAJS [37, 3]. The limitations of JSIR/WALA are substantial: neither of them comes with either an associated compiler or reference implementations of JavaScript internal functions or a publication. The absence of a compiler makes it impossible for us to discuss the precise nature of the differences between JSIL and JSIR/WALA. On the syntactic level, there exist similarities between JSIL and JSIR: both JSIL and JSIR have built-in support for SSA, and as functions in JSIR come annotated with their respective control flow graphs and lexical scopes, JSIL functions come with associated identifiers and a scope clarification function. However, there is no actual demonstration of how JSIR constructs

would be used to model JavaScript, and any further analysis would involve resorting to guesswork. Moreover, our choices for JSIL come from us wanting to follow closely the standard; the reasons for the choices for JSIR and WALA are not stated. TAJS includes a well-tested compiler, targeted for ES3 (which is substantially different from ES5) but now extended with partial models of the ES5 standard library, the HTML DOM, and the browser API. Since TAJS was designed for type analysis and abstract interpretation, the IR that it uses is slightly more high-level than those typically used for logic-based symbolic verification. The IR of SAFE based on control flow is not documented.

We will now turn our attention to the IRs in **(1)**, which we have considered using as an interim stage during compilation. In [32], the authors introduce λ_{JS}, a lambda calculus extended with objects and prototype-based inheritance that incorporates the essential features of the ECMAScript 3 standard (with the exception of the eval function), that is compact and well-suited for formal reasoning. To demonstrate this, the authors designed a type system for checking a simple confinement property for λ_{JS} programs. Furthermore, they provide a desugaring function, through which they are able to compile JavaScript programs into λ_{JS}. Both λ_{JS} and this desugaring function are automated, and tested against the Mozilla JavaScript test suite. In [55], the authors introduced S5, an adaptation and extension of λ_{JS} that captures the new features and semantics introduced by the ES5 standard, including getters and setters, as well as the strict mode version of eval. S5 places its emphasis on the core features of ES5 Strict, and is tested against the ES5 Test262 test suite. Kashyap et al. [41] proposed notJS, an intermediate language for ES3 for which they design an abstract interpretation analysis, JSAI. notJS keeps most of the control flow constructs of JavaScript, including simplified versions of the for-in loop and the try-catch-finally statement. JSAI is designed to be provably sound with respect to a specific concrete semantics for JavaScript, which has been extensively tested against SpiderMonkey on their own made test suite.

λ_{JS} and notJS were not appropriate as their target is ES3, an older version of JavaScript. The best candidate was S5 developed by Politz et al. [55], which targets the full ES5 standard. The compilation from ES5 to S5 is informally described in the paper, and is validated through testing against the ECMAScript test suite, achieving 70% success on all ES5 tests and 98% on tests designed specifically to test unique features of ES5 Strict. However, the figure critical for us, which is the success rate of S5 on full ES5 Strict tests (those testing its unique features *and* the features common with ES5), was not reported. This may have been due to the numerous errors in ES5 Test262 tests designed for testing common features that render them unrunnable in strict mode. We overcame this issue by moving to ES6 Test262. The only way for us to use S5 would have been to run it on our testing infrastructure and then fix the unfamiliar code in light of failing tests. Also, to prove correctness of our assertion translation and, ultimately, JaVerT, we would have to relate JS Logic and JSIL Logic via S5. This would be a difficult task.

One of our main goals in the development of the JS-2-JSIL compiler was to be fully compliant with ES5 Strict. Thus, a strong connection between the generated JSIL code and the standard was imperative. Our design of the JS-2-JSIL compiler builds on the tradition of compilers that closely follow the operational semantics of the source language, such as the ML Kit Compiler [8]. In that spirit, the JS-2-JSIL compiler mimics ES5 Strict by inlining in the generated JSIL code the internal steps performed by the ES5 Strict semantics, making them explicit.

2.4. Verification Tools Based on Separation Logic

Separation Logic. Hoare logic [35] was introduced to reason formally about the properties of programs. A Hoare triple $\{P\}\, c\, \{Q\}$ describes the behaviour of the program c, where P and Q are assertions for representing the state of the program. A triple $\{P\}\, c\, \{Q\}$ means that if a state satisfies a precondition P, then, if the program c terminates, it does so in a state satisfying the postcondition Q. Hoare logic provides axioms and inference rules for the constructs of the language to derive Hoare triples. In Hoare logic, assertions describe the entire state of the program. This makes it difficult to write program specifications, as we constantly need to describe not only the part of the state that the program changes, but also the parts of the state that stay the same.

Separation logic [57, 50], an extension of Hoare logic, provides modular reasoning about programs which manipulate heap structures. To achieve modularity, separation logic allows us to describe and reason about only a part of a given heap by introducing the separating conjunction $*$. The formula $P * Q$ denotes a heap that can be split into two disjoint parts, where one part satisfies formula P and the other satisfies Q. The precondition P in separation logic only needs to describe the parts of the heap that are necessary for the execution of the program c. Then, using the frame rule, the state can be extended with parts that have not changed.

$$\textsc{Frame Rule}\quad \frac{\{P\}\, c\, \{Q\} \quad \mathsf{mod}(c) \cap \mathsf{fv}(R) = \emptyset}{\{P * R\}\, c\, \{Q * R\}}$$

The frame rule states that if we can prove the specification $\{P\}\, c\, \{Q\}$, we can extend it with R, which describes a part of the heap disjoint from P and Q and which does not mention variables modified by c. Using the frame rule, we can analyse parts of the code independently and join the results together. Modularity is the key to the success of separation logic, since it allows scalable analysis of large codebases.

Separation Logic Tools for Static Programming Languages. Separation logic has been successfully applied to verification tools for static languages. Initially, it was used for reasoning about simple imperative while languages. Smallfoot [4] is the first verification tool based on separation logic. It uses symbolic execution and introduces the frame inference technique. For reasoning about data structures such as lists, Smallfoot provides built-in predicates. To verify programs, a user needs to provide preconditions and postconditions for the functions, as well as loop invariants. Later, the techniques introduced by Smallfoot were transferred to mainstream programming languages. To minimise the burden of annotations, tools infer loop invariants automatically, hence requiring only specifications for functions. Such tools include jStar [22, 11] for Java and Space Invader [72] for C. jStar also supports user-defined predicates. However, it requires of the user to provide logic and abstraction rules for reasoning about such predicates. Abductor [13] is a fully automatic tool that infers not only loop invariants, but also preconditions and postconditions. To achieve this, it uses the bi-abduction technique. Infer [14] is a commercial tool based on bi-abduction and is used in Facebook for reasoning about C, Java, Objective C, and C++. Other tools, instead of aiming at full automation, are supporting richer specifications. For example, Verifast [36, 54] allows users to define custom predicates and specify functional properties for C and Java programs. However, it requires more annotations, such

as fold and unfold directives for inductive predicates on top of the specifications and loop invariants. Verifast has been used to verify safety properties for industrial applications, such as Java Card[3] applets for smart cards, a Linux device driver, and an embedded Linux network management component.

We believe that separation logic is a good fit for JavaScript verification. However, the transposition of the techniques developed for static languages to a highly complex dynamic language, such as JavaScript, is challenging.

Separation Logic for JavaScript. Gardner et al. [30] have developed a separation logic for a tiny fragment of ECMAScript 3, to reason about the variable store emulated in the JavaScript heap. To adapt separation logic for reasoning about JavaScript programs, the authors introduce an assertion to describe negative information about an existence of a property in an object, $(l, p) \mapsto \varnothing$, and a new connective sepish, $\ltimes$, to account for possible sharing. The assertion $(l, p) \mapsto \varnothing$ states that the object l does not have the property p. We draw partial inspiration from this work: our property assertions are similar. However, we do not use sepish, as it complicates automation. Sepish gives us more flexibility in writing specifications, however, at the cost of the ability to prove properties. We will expand on this throughout the thesis, showing how we are still able to write specifications without using sepish.

The logic presented in [30] supports a fragment of JavaScript with simplified semantics. For instance, it does not consider attributes, implicit coercions or any of the intricate control flow commands of the language. Even in such a simplified setting, this proposed logic is very complex due to the (remaining) complexity of JavaScript. We did build a prototype tool, JuS [29], based on the logic, and were able to semi-automatically reason about very simple programs that manipulate the JavaScript variable store.

An extension of this logic to the full language is intractable. For example, the behaviour of the JavaScript assignment is described in the ECMAScript standard in terms of expression evaluation and calls to the internal functions `getValue` and `putValue`. This effectively means that the assignment is described by hundreds of possible pathways through the standard; each of these pathways would have to be a proof rule of the logic, making automation essentially impossible. The same issues would give rise to even greater complexity when applied to the complex control flow given by the `switch` and `try−catch−finally` statements. Direct verification of JavaScript programs using separation logic is, therefore, not feasible. We believe that working directly with JavaScript is too complex in the context of verification and we contend that the correct approach is to first translate JavaScript programs to JSIL and only then to apply automatic reasoning techniques. With such an approach, we move a great part of JavaScript complexity from the logic and into the translation.

Targeting Existing Separation Logic Tools. Instead of developing JSIL and JSIL Verify, we might have attempted to compile ES5 Strict to a language supported by an existing separation logic tool [11, 36, 14]. The main problem is that these tools all target static languages that do not support extensible objects, dynamic properties or dynamic binding of procedure calls. Hence, JavaScript objects cannot be directly encoded using the built-in constructs of these languages. Consequently, at the logical level, one would need to use custom abstractions to reason about JavaScript objects and their associated operations, such as reading from the heap, writing to the heap or deleting a property of an object.

The most closely related tool to ours is Verifast [36]. As in Verifast, we need to provide pre- and postconditions, loops invariants, and fold/unfold statements for user-defined predicates. However,

[3] A Java platform for embedded devices.

Verifast does not allow us to provide a special rule for reasoning about dynamic properties. Instead, we would need to provide abstract predicates, as explained above, to reason about such a fundamental part of the language. In Verifast, a heap cell $(\mathrm{E}_1, f) \mapsto \mathrm{E}_2$ only allows constant field names, but not arbitrary expressions to represent fields. In order to describe a JavaScript heap cell we would need to define an abstract predicate, such as $\mathsf{Cell}(\mathrm{E}_1, \mathrm{E}_2, \mathrm{E}_3)$. Verifast does not provide any means for defining axioms about predicates. Therefore, when we talk about two disjoint cells at the same location $\mathsf{Cell}(\mathrm{E}_1, \mathrm{E}_2, \mathrm{E}_3)$ and $\mathsf{Cell}(\mathrm{E}_1, \mathrm{E}'_2, \mathrm{E}'_3)$, we need to make sure that the fields E_2 and E'_2 are not the same: $\mathsf{Cell}(\mathrm{E}_1, \mathrm{E}_2, \mathrm{E}_3) * \mathsf{Cell}(\mathrm{E}_1, \mathrm{E}'_2, \mathrm{E}'_3) * \mathrm{E}_2 \neq \mathrm{E}'_2$. If we have three cells, we get $\mathsf{Cell}(\mathrm{E}_1, \mathrm{E}_2, \mathrm{E}_3) * \mathsf{Cell}(\mathrm{E}_1, \mathrm{E}'_2, \mathrm{E}'_3) * \mathsf{Cell}(\mathrm{E}_1, \mathrm{E}''_2, \mathrm{E}''_3) * \mathrm{E}_2 \neq \mathrm{E}'_2 * \mathrm{E}_2 \neq \mathrm{E}''_2 * \mathrm{E}'_2 \neq \mathrm{E}''_2$. This encoding becomes quadratic in the number of cells since we need to account for all possible inequalities.

coreStar [11] is a verification framework based on separation logic with its own intermediate language coreStarIL and separation-logic theorem prover. We are not able to use coreStarIL because it only supports static binding of function calls. We did use coreStar's theorem prover to build the prototype tool JuS [29] and the first version of JSIL Verify. The theorem prover provided by coreStar is flexible enough to define our reasoning about extensible objects, dynamic properties and dynamic functions. coreStar provides only an assertion for an empty heap, `emp`, and the separating conjunction, $*$. Everything else needs to be provided by a user in terms of abstract predicates and logic rules to manipulate them. Having a list of logic rules, coreStar then performs proof search. As in the Verifast case, we would define the predicate $\mathsf{Cell}(\mathrm{E}_1, \mathrm{E}_2, \mathrm{E}_3)$. However, in coreStar we are able to provide an axiom $(\mathsf{Cell}(\mathrm{E}_1, \mathrm{E}_2, \mathrm{E}_3) * \mathsf{Cell}(\mathrm{E}_1, \mathrm{E}'_2, \mathrm{E}'_3) * \mathrm{E}_2 = \mathrm{E}'_2) \implies \mathsf{false}$ to avoid quadratic encoding. As our experience shows, it is extremely difficult to provide logic rules without having any control of their application, making the proof search untractable. Hence, we have opted out of using coreStar in the latest version of JSIL Verify.

3. The JavaScript Language

We describe the key features of the JavaScript language focusing on the ECMAScript 5 (ES5) and explain why we choose to work with the strict mode (ES5 Strict) of the language (§3.1). Using a priority queue implementation as our running example, we describe behaviour of ES5 Strict constructs in more detail (§3.2). Finally, we formally define the complete memory model of full ES5 Strict (§3.3), and introduce operational semantics of a fragment of ES5 Strict (§3.4) that we use throughout the thesis.

3.1. JavaScript: ECMAScript 5

JavaScript is sometimes called the most misunderstood language in the world. There is a constant debate on whether it is, in fact, an object-oriented language or a functional language. Some people say that the language itself is simply badly designed. What is JavaScript?

JavaScript is defined by the international ECMAScript standard, with which all major web browsers comply. The 5th edition of the ECMAScript standard (ES5) can broadly be divided into three parts (Figure 3.1): the first part (chapters 1-7) introduces the syntax and details the parser; the second part (chapters 8-14) addresses the core of JavaScript, including language constructs, such as expressions, statements, and internal functions; the third part (chapter 15) describes a number of built-in libraries that provide additional functionalities on top of the core language.

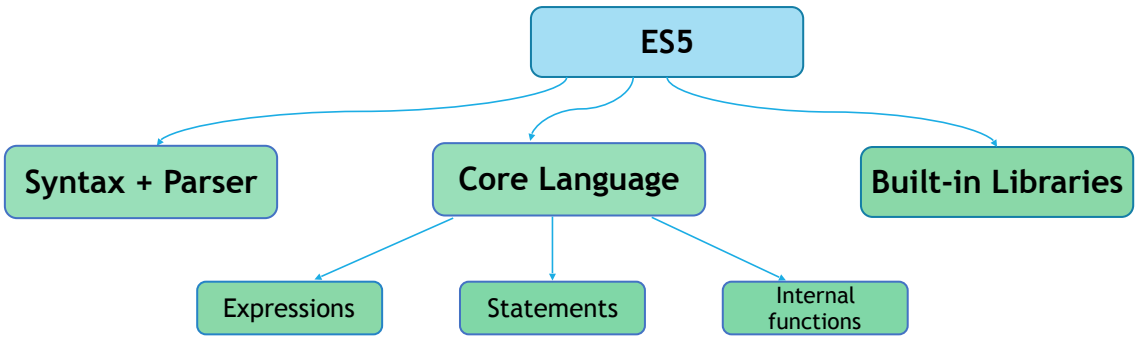


Figure 3.1.: The JavaScript Language described by the ECMAScript 5 standard.

ES5 also introduces a *strict mode* of the language (ES5 Strict), a restricted variant of ES5 that intentionally has slightly different semantics compared with the full language, and exhibits better behavioural properties, such as lexical scoping and better error checking. Strict mode features are mostly addressed via notes interspersed throughout the standard.

Developing and verifying a correct JavaScript parser is out of the scope of our project. Instead, we use an off-the-shelf parser, Esprima [34], which is widely used and is standard-compliant.

3.1.1. The Key Concepts of JavaScript

JavaScript is an object-based language, by which we mean that an object is the main notion of the language and most language features are described in terms of objects. For example, inheritance is supported using *prototype* objects; functions are stored in the JavaScript heap as function objects, which hold the code of the original function together with a representation of the scope in which the function was defined; function scope consists of *environment records*, which are special objects, that contain JavaScript variables as their properties.

In JavaScript, objects are stored in the JavaScript heap, an example of which is shown in Figure 3.2. We give names to the objects in order to refer to them more easily, for example `Object.prototype`, `n1`, and `Node`. Using this heap as an example, we next describe JavaScript objects in general, function objects, and prototype-based inheritance.

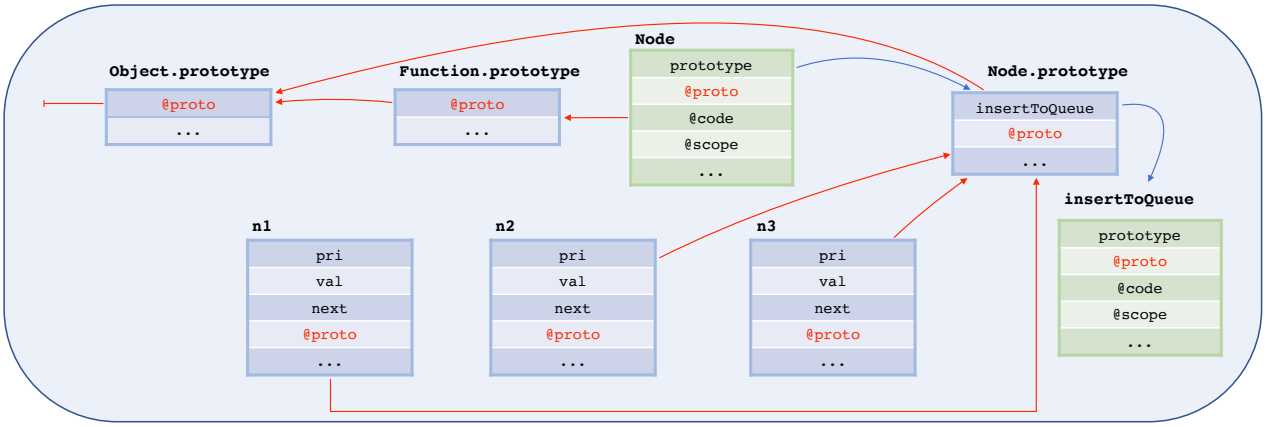


Figure 3.2.: A JavaScript heap illustrating JavaScript objects and prototype-based inheritance.

Objects, Object Properties. A JavaScript object is a collection of properties. JavaScript objects differ from C++ or Java objects in several defining ways. First of all, they are *extensible*, in that properties can be added to/removed from an object after creation. Also, JavaScript objects have two types of properties: *named* and *internal*. Named properties can be thought of as object fields in the style of C++ or Java, except that they are associated with not only a value, but also with a collection of attributes that describe the ways in which a property can be used. For example, the *writable* attribute, which we denote by `[W]`, describes whether or not the property is read-only. This will be discussed in full detail in §3.2. Internal properties, in contrast, are hidden from the user and are critical for the mechanisms underlying JavaScript, such as prototype-based inheritance. We use the prefix `@` to denote internal properties. In Figure 3.2, the object `n1` has three named properties `"pri"`, `"val"`, and `"next"` and one internal property `@proto`, whereas the object `Node` has one named property `"prototype"` and three internal properties `@proto`, `@code`, and `@scope`.

Functions, Function Objects. JavaScript supports the functional programming style. Functions in JavaScript are first-class citizens, which means that they can be passed as arguments to other functions and also returned as outcomes of functions.

Additionally, JavaScript features nested functions. An inner function can use variables defined in an outer function and, hence, needs to be able to access the variables from the outer function. In fact, we can think of JavaScript functions as being *closures*, since they contain both their code and

the representation of scope in which the function was defined. Functions are stored in the JavaScript heap as objects. Each function object has two specific internal properties: `@code`, storing the code of the function; and `@scope`, storing the scope of the function, used for variable resolution. There are two function objects in Figure 3.2, `Node` and `insertToQueue` (we use the green colour to denote function objects). Consider the following JavaScript code snippet, which creates the function object `Node`:

```
1  var Node = function(pri, val) {
2    this.pri = pri; this.val = val; this.next = null;
3  }
```

The ES5 Standard states that the property `@code` stores the ECMAScript code of a function, however, it does not insist on any format. We give a unique identifier to every function and store this identifier in the property `@code`, instead of storing the code itself. We will illustrate the property `@scope` in the later section describing variable binding.

Prototype-based Inheritance. In JavaScript, object inheritance is prototype-based. It is tracked through a dedicated internal property, which we denote by `@proto`. All JavaScript objects have the internal property `@proto`. When a new JavaScript object is created, its property `@proto` is set to an appropriate prototype object. Since new objects in JavaScript are created using functions as constructors, JavaScript function objects also have a dedicated named property `"prototype"`, storing the prototype of objects created using that function. Figure 3.2 illustrates JavaScript's prototype-based inheritance. The object `Node` is a function object that can be invoked as a constructor using the JavaScript expression `new`. It has a named property `"prototype"` that points to another object, namely `Node.prototype` (we use blue arrows for named properties that hold objects). `Node.prototype` is a prototype object for the newly created objects `n1`, `n2`, and `n3` which is expressed by their internal property `@proto` pointing to `Node.prototype` (we use red arrows to denote that one object is a prototype object of another object). Objects `n1`, `n2`, and `n3` have their own named properties `"pri"`, `"val"`, and `"next"`, and they also share a property `"insertToQueue"` from their prototype `Node.prototype`. They also share all of the named properties of the prototype of the `Node.prototype`, which, in our example, is `Object.prototype`. `Object.prototype` is a built-in object that serves as the default prototype object. `Object.prototype` has the property `@proto` with value `null`, which terminates the *prototype chain*.

An object inherits all named properties from the objects in its prototype chain. When we look for a property in an object, we traverse its prototype chain until we find the property in question. For example, in order to determine the value of a property `p` of a given instance of `Node`, say `n1`, we first check whether `n1` has the property `p`, in which case the property lookup yields its value. Otherwise, we check if `p` belongs to the properties of `Node.prototype`, and if it does not, we look for it in `Object.prototype`. If it is not found there either, we declare it to be undefined.

All JavaScript objects have prototype chains. As seen in the example, even the function object `Node` has a prototype, `Function.prototype`, which is another built-in object that is the default prototype object for all JavaScript functions. Also, the prototype of `Function.prototype` is `Object.prototype`.

By having prototype-based inheritance, JavaScript is expressive enough to support a flavour of the object-oriented programming style. Indeed, we can think of objects `Node` and `Node.prototype` being almost equivalent to a class in Java or C++: `Node` plays a role of a constructor while `Node.prototype` holds shared properties and methods. The difference is that the prototype inheritance of JavaScript does not support full encapsulation. Even in the latest version of the standard, which features classes, full encapsulation is still not supported. We elaborate on this point when discussing the running

example in §3.2.

Variable binding. Variables in JavaScript are properties of special objects, called *environment records*. An environment record (ER) is an 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*, which each function object stores in the property `@scope`. Let us illustrate variable binding with nested functions by moving our `Node` function inside immediately-invoked function expression `(function(){...})()` and adding another function `f` that uses `Node` as a constructor:

```
1  (function() {
2     var Node = function(pri, val) {
3        this.pri = pri; this.val = val; this.next = null;
4     }
5
6     var f = function(pri, val) {
7        var n = new Node(pri, val);
8     };
9
10    f(1, "last");
11 })();
```

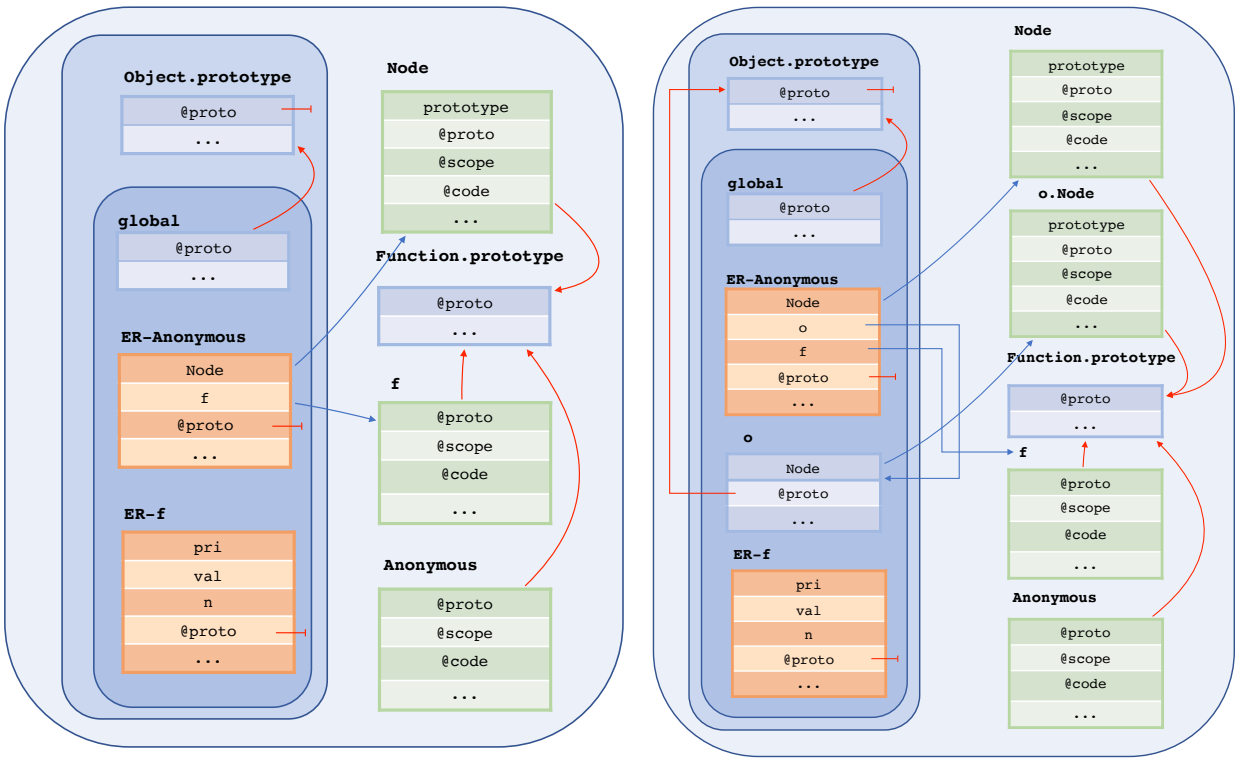


Figure 3.3.: Variable binding in the JavaScript heap

In the execution of the above code, three function objects will be created (Figure 3.3, left). The `@scope` of the function object for the anonymous function will be `[global]` as it is defined in the global code. The *global* object, `global`, is a special built-in object that holds all global variables and is present before executing any JavaScript program. The `@scope` for functions `Node` and `f` will be `[global, ER−Anonymous]` since both functions are created inside the anonymous function. `ER−Anonymous` is an environment record (we show environment records in orange) created upon the invocation of the anonymous function `Anonymous` and it has two properties, `"Node"` and `"f"`, that point to the

corresponding function objects. When we call the function `f`, a new environment record `ER−f` is created, which contains the properties `"pri"`, `"val"`, and `"n"`, corresponding, respectively, to the two formal parameters of the function and a local variable defined inside the function. The body of the function `f` is executed in the scope [`global`, `ER−Anonymous`, `ER−f`], which is highlighted in Figure 3.3 (left). We can think of the scope chain objects together with their prototypes as an emulated variable store. There are four variables in the body of the function `f`. When these variables are resolved, variables `n`, `pri`, and `val` will be found in the last environment record, `ER−f`, while the variable `Node` will be found in the second environment record, `ER−Anonymous`.

Variable dereferencing gets trickier in the presence of the `with` statement. Without the `with` statement, the only object in the scope chain that can have a non-null prototype is the global object. The `with` statement allows any object to be a part of the emulated variable store. Let us define the function `f` inside a `with` statement.

```
1  (function() {
2    var Node = function(pri, val) {
3      this.pri = pri; this.val = val; this.next = null;
4    }
5
6    var o = {Node : function(){}};
7    with (o) {
8      var f = function(pri, val) {
9        var n = new Node(pri, val);
10     };
11   }
12
13   f(1, "last");
14 })();
```

The `with (o){s}` statement changes the current scope inside the execution of the statement `s` by appending the object `o` to the current scope chain. The `@scope` for the function object `f` in this case becomes [`global`, `ER−Anonymous`, `o`]. The function `f` is then executed in the scope [`global`, `ER−Anonymous`, `o`, `ER−f`] (Figure 3.3, right) and the variable `Node` is resolved to the function object `o.Node`, which contains an empty function expression.

The strict mode of the JavaScript language forbids the use of the `with` statement. Without the `with` statement, only environment records and the global object can be a part of a scope chain of a function object. This simplifies variable binding and ensures lexicographic scoping.

3.1.2. The Core Language

JavaScript programs are built from statements, which, in turn, are built from expressions. The behaviour of statements and expressions is described using a variety of internal functions, not available to the developer. In this section, we give a flavour of JavaScript statements, expressions and internal functions, and describe some non-standard aspects of their behaviour.

JavaScript Statements. JavaScript statements (Figure 3.4) include variable definitions, conditional statements (`if then else`, `switch`), a number of iteration statements (`do−while`, `while`, `for`, `for−in`), various control flow statements (`continue`, `break`, `return`, labelled statements), exception handling statements (`throw`, `try−catch−finally`), `with` statements, expression statements and blocks of statements (or sequencing). In §3.4, we formally present a fragment of ES5 Strict, and the statements included in this fragment are highlighted in green.

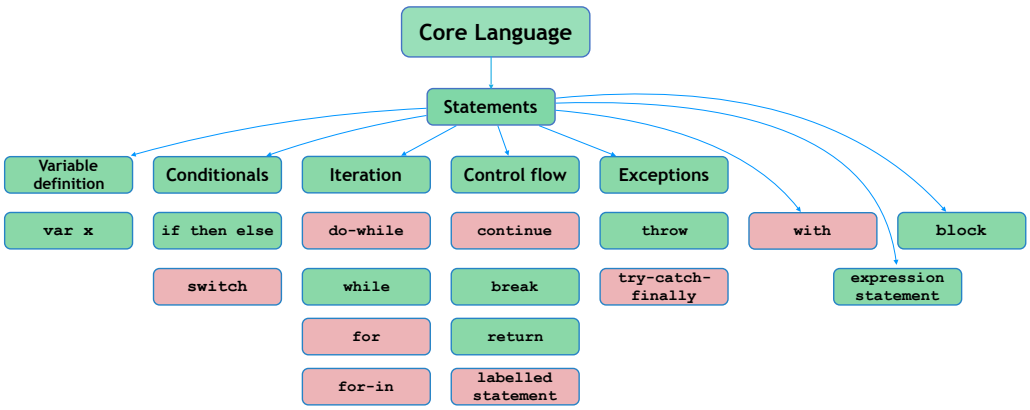


Figure 3.4.: ES5 statements.

A difference with respect to other programming languages is that in JavaScript not only expressions, but also statements, return values which makes the semantics more complicated. For example, sequencing is more complex comparing to other languages. Let us illustrate this with a simple example. The result of evaluating a sequence `var x`; `x` = 3 is 3, while the result of evaluating `var y` is a special value `empty`. We might expect that joining these two statements to `var x`; `x` = 3; `var y` would result in `empty` as this was the result of the last statement. However, the composition evaluates to 3. JavaScript has a special treatment of the `empty` value in sequences. If a statement evaluates to `empty` the result is the last non-empty value of the previous statements. In ES5, the only way to observe a return value of a statement is by using the function `eval`.

JavaScript Expressions. JavaScript expressions (Figure 3.5) include literals (`null`, `undefined`, booleans, numbers, and strings), the `this` keyword, variables, array and object initialisers, function expressions, function and constructor calls, property accessors, assignments, and various operators: unary operators (`delete`, `typeof`, ...), the `instanceof` operator, equality operators (===, ==, ...), binary additive operators (+, −), multiplicative operators (∗, /, %), relational operators (<, >, ...), binary logical operators (&&, ...), and bitwise operators (<<, >>, ...). Similar to JavaScript statements, the green colour represents the fragment formally presented in §3.4. Most of the expressions behave as expected, but there still exists a number of peculiarities that need to be addressed.

JavaScript Expressions: the Confusing `this`. Similar to object-oriented languages, JavaScript has the keyword `this` that is used to denote the current object in a constructor or a method. Recall the code of the function `Node`:

```
1  var Node = function (pri, val) {
2    this.pri = pri; this.val = val; this.next = null;
3  }
```

To create a new node object, we use the JavaScript expression `new Node(...)`. A new object, say `n1`, is created and when the body of the function `Node` is executing, the `this` corresponds to the newly created object `n1`. Also, when we call the method `n1.insertToQueue(...)`, the `this` inside the body of the function `insertToQueue` corresponds to the object `n1`.

The confusion comes from the fact that, differently from other object-oriented programming languages, in JavaScript a constructor or a method is also a function and it can be called in a standard way, for example, `Node(...)` or `f` = `n1.insertToQueue`; `f()`. What happens to the `this` then? In the non-strict mode of the language, the `this` will correspond to the `global` object. In the case of `Node(...)`,

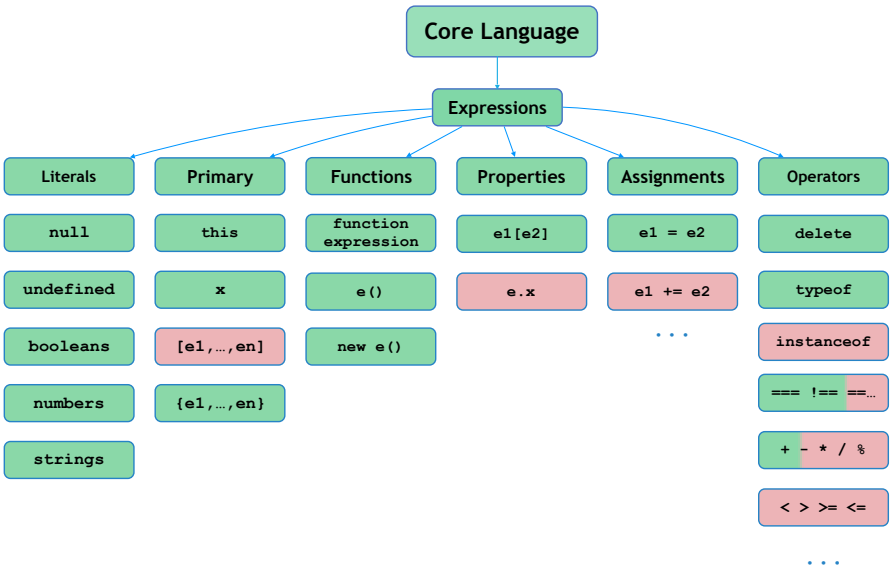


Figure 3.5.: ES5 expressions.

three new global variables, `pri`, `val`, and `next` will be created. This is not what we intended and instead of getting an error because we forgot the `new` keyword, we end up creating global variables. This is a common mistake in JavaScript programs. The strict mode of the language helps finding such errors earlier on in the development. In strict mode, when a function is called in a standard way, the `this` gets the value `undefined` instead of the global object. In the case of `Node(...)`, `this.pri` throws an error, since `undefined` is a primitive value, not an object, and therefore cannot have properties.

JavaScript Expressions: Implicit Type Conversion. JavaScript was designed with the idea of delaying error reporting to the user for as long as possible. To that end, most of JavaScript operators implicitly convert operands to the types that they expect, which makes the understanding of JavaScript programs and the detection of bugs more difficult. For entertainment purposes, little JavaScript code snippets that make use of this behaviour are often shown as puzzles. For example, when we add up an empty string and an empty object `""`+ {}, we might expect a type error. However, JavaScript implicitly converts the empty object to the string `"[object Object]"`, which becomes the result of evaluating the whole addition expression. We might expect the same result when we switch operands, {} + `""`. However, what we get is, in fact, 0. In this case, {} is actually treated as an empty block, not an empty object. Hence, this code would not even be treated as an expression, but in fact as the sequence of statements {}; + `""`, where + is a unary operator on numbers. Therefore, when evaluated, the empty string is converted to the number 0, which is the outcome of the entire evaluation.

The JavaScript abstract equality or *double-equal* operator == also uses implicit type conversion. For example, all these expressions evaluate to true: `true` == 1, `false` == `""`, `undefined` == `null`. It is advisable to use strict equality === instead, which requires both operands to be of the same type in order for the equality to hold.

JavaScript Expressions: Property Accessors. A familiar way of accessing a property `p` of an object `o` in object-oriented languages such as Java is to use the dot notation `o.p`. In JavaScript, there are two ways to access properties: member access `e.x` and computed access `e1[e2]`. ES5 standard states that `e.x` is identical in its behaviour to `e["x"]`. When we use member access, we statically know the name of the property `x`, whereas in computed access the name of property is dynamically evaluated

from the expression e2. This is what we mean by the phrase *dynamic properties*. A property can only be a string, and if e2 evaluates to a value of another type, an implicit type conversion using the internal ToString function occurs. Here are several examples that illustrate computed access:

```
1 var o = { prop: 0 };
2 var r1 = o["pr"+"op"]; // evaluates "pr"+"op" to "prop", then evaluates the property "prop" of object o,
3                        // which has value 0
4
5 var o2 = { 1: 1 };
6 var r2 = o2[4 - 3]; // evaluates 4-3 to 1, implicitly converts the number 1 to the string "1",
7                     // then evaluates the property "1" of object o2, which has value 1
8
9 var o3 = { toString: function(){return "prop"} };
10 var r3 = o[o3]; // evaluates o3 to an object, implicitly converts it to the string "prop",
11                // then evaluates the property "prop" of object o, which has value 0.
12
13 var o4 = {};
14 o4[o] = 2;       // evaluates o to an object and implicitly converts it to the string "[object Object]"
15 var r4 = o4[o2]; // evaluates o2 to an object, implicitly converts it to the string "[object Object]",
16                  // then evaluates the property "[object Object]" of object o4, which has value 2
```

The most confusing lines of the above example are the lines 9-15, where an object is used as a property. In such a case, a given object is implicitly converted to a string by an internal JavaScript function ToString. ToString takes one parameter v as input (either undefined, null, a boolean, a string, a number, or an object), and returns the corresponding string: when v = undefined, it returns "undefined"; when v = null it returns "null"; when v = true, it returns "true"; when v = false it returns "false"; when v is a string, it returns v; and when v is a number, it returns its string representation (for example, when v = 5, it returns "5"). The case in which v is an object is the most complex. There, ToString calls another internal function, ToPrimitive, which converts an object to a primitive value (either undefined, null, a boolean, a number, or a string) by calling yet another internal function, DefaultValue. DefaultValue checks if the given object has a function toString in its prototype chain and if it does, it calls the function and returns the result of that call, which is then propagated back as the return value of the initial call to ToString. The object o3, defined in line 9, has the function toString. Hence, when the implicit conversion using ToString takes place in line 10, o3 is converted to a string "prop", resulting in r3 containing the value 0. In line 13, we create an empty object o4 and set its property o in line 14 to hold value 2. The object o itself does not have the function toString, but its prototype, Object.prototype, does by default. Object.prototype.toString always returns the string "[object Object]" when a given argument is an object of class "Object". Hence, the result of converting the object o to a string is "[object Object]". In line 15, we access the property o2 from the object o3. The same implicit type conversion occurs, resulting in the same property "[object Object]". Hence, the value of r4 is 2.

JavaScript Expressions: Functions. In an object-oriented language such as Java, we statically know most of the times which function will be called given its name. One exception is the dynamic dispatch, where methods are resolved dynamically from a constrained set of defined implementations. In JavaScript, the relationship between the name and the actual function is much looser. Consider the following example:

```
1 var y = function() {...};
2 var x = function() {return 1};
3 y();
4 var r1 = x();
```

We define two functions and assign them to variables x and y. For the moment, we do not know what

the body of function `y` does. We might expect that if we reach line 4 (that is, if function `y` does not throw an exception), then we will call the function defined on line 2 and get value 1 for `r1`. However, nothing is preventing us from reassigning the value of the variable `x` inside the body of the function `y`, for example, `var y = function(){x = function(){return 3}};` In such a case, the value assigned to `r1` will be 3. Similarly, for built-in functions, we cannot be sure that, for instance, `Object.defineProperty` is always the actual function intended by ES5. We can easily re-assign it to anything else, for example, `Object.defineProperty = 5`.

This illustrates what we aim to capture by using the term *dynamic function*: we cannot assume to know statically which function is being called. We need to look into the heap to obtain its code.

Internal Functions. JavaScript internal functions, shown in Figure 3.6, describe the fundamental inner workings of the language, such as prototype chain traversal (`GetProperty`), property definition (`DefineOwnProperty`) and property deletion (`Delete`), as well as type conversions implicitly invoked by JavaScript operators (`ToString`, `ToNumber`, `ToInteger`, `ToBoolean`, `ToPrimitive`, `ToObject`, ...). They also include internal functions on references, which are internal constructs of JavaScript specification to represent resolved property bindings (`GetValue`, `PutValue`). JavaScript internal functions are not accessible by the developer in ES5[1], but are called internally by all JavaScript expressions and statements. Their definitions in the standard are complex, are given operationally, and are often intertwined, making it difficult for the user to fully grasp the control flow and allowed behaviours. To illustrate: `GetValue` calls `Get`, which calls `GetProperty`, which calls `GetOwnProperty`; `PutValue` calls `Put`, which calls `CanPut` and `DefineOwnProperty`, which calls `GetOwnProperty`. Similar to JavaScript statements and expressions, the green colour represents the fragment formally presented in §3.4. We will discuss internal functions in more detail in §7.

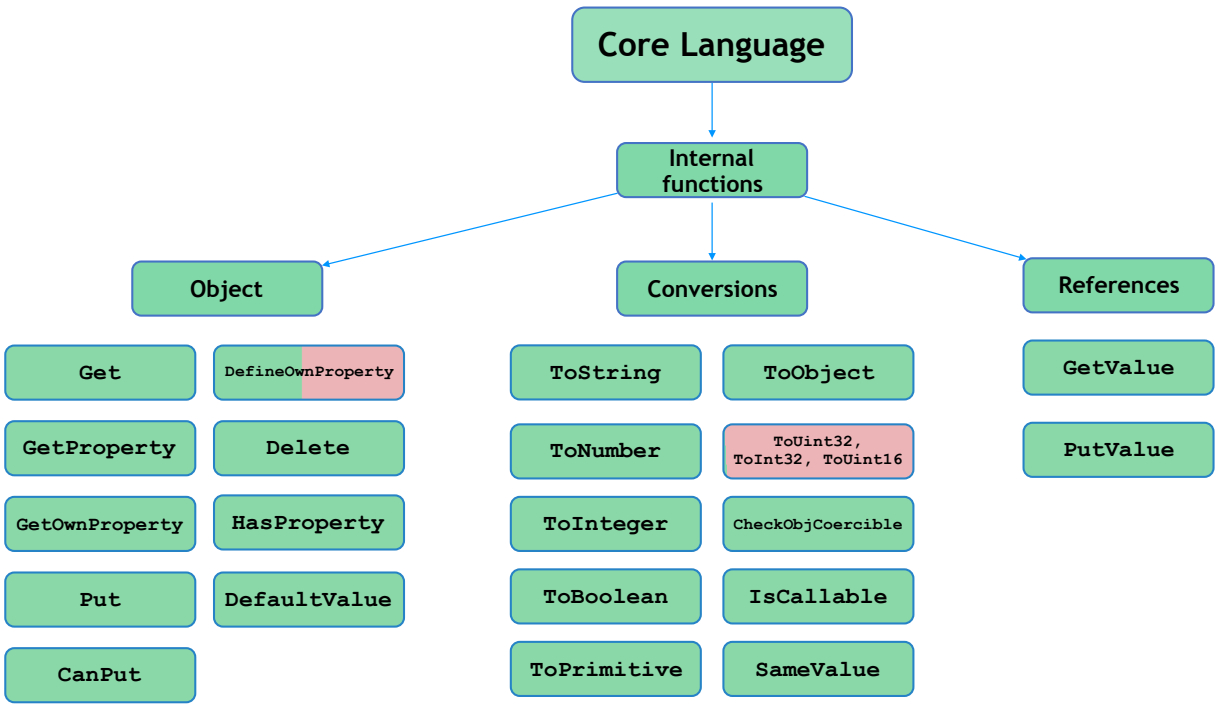


Figure 3.6.: ES5 internal functions.

[1]ES6 has the Reflect library, which effectively exposes the internal functions to the user.

3.1.3. Built-in Libraries and the Initial Heap

JavaScript has a collection of built-in libraries. While most of the built-in libraries provide additional functionality to the core of JavaScript, some of them are strongly intertwined with the core language. Figure 3.7 enumerates all the built-in libraries of ES5.

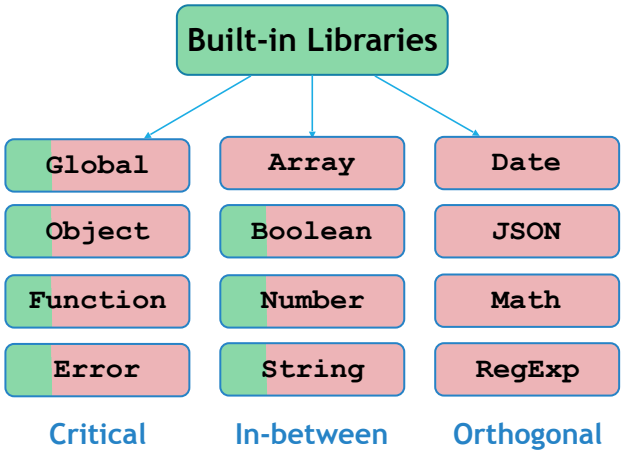


Figure 3.7.: The Built-in Libraries in ES5

The `Global` library, associated with the global object, `Object`, `Function`, and `Error` libraries are intertwined with the core language. We already mentioned some of the built-in objects when we explained the key features of the language. For example, a unique global object holds all global variables, `Object.prototype` is the default prototype object of newly created objects, while `Function.prototype` is the default prototype object of function objects. Basic functionalities of `Array`, `Boolean`, `Number`, and `String` libraries are used to define some of the core features, as well. For example, the type conversion `ToObject` uses `Boolean`, `Number`, and `String` constructors to create objects when converting primitive values booleans, numbers and strings to objects. In contrast, the `Date`, `JSON`, `Math` and `RegExp` libraries are orthogonal to the core language. Similar to other JavaScript constructs, the green colour represents the fragment formally presented in §3.4. We will discuss the libraries in more detail in §5.2.

The built-in libraries are implemented in terms of objects, called built-in objects. The built-in objects are present before the execution of any JavaScript program, and together form what we call the *JavaScript initial heap*.

In Figure 3.8, we illustrate the critical built-in objects and the relationships between them in the JavaScript initial heap. Built-in objects, such as `Object`, `Function`, and `Error` are properties of the `global` object. They are all functions, hence their prototype (`@proto`) is `Function.prototype`, and they all have `"prototype"` properties that point to the `Object.prototype`, `Function.prototype`, and `Error.prototype` objects, respectively. Other (non-critical) built-in objects are also properties of the global object and have their respective `"prototype"` properties.

Built-in objects contain a number of functions associated to them. For example, Figure 3.8 shows two such functions, `Object.defineProperty` and `Object.prototype.toString`, which are properties of `Object` and `Object.prototype`, respectively. `Object.defineProperty` is a proper library function, in the sense that the core language does not depend on it. `Object.prototype.toString`, on the other hand, is used in the property accessor `e1[e2]` in the case if `e2` is an object and does not have the `toString`

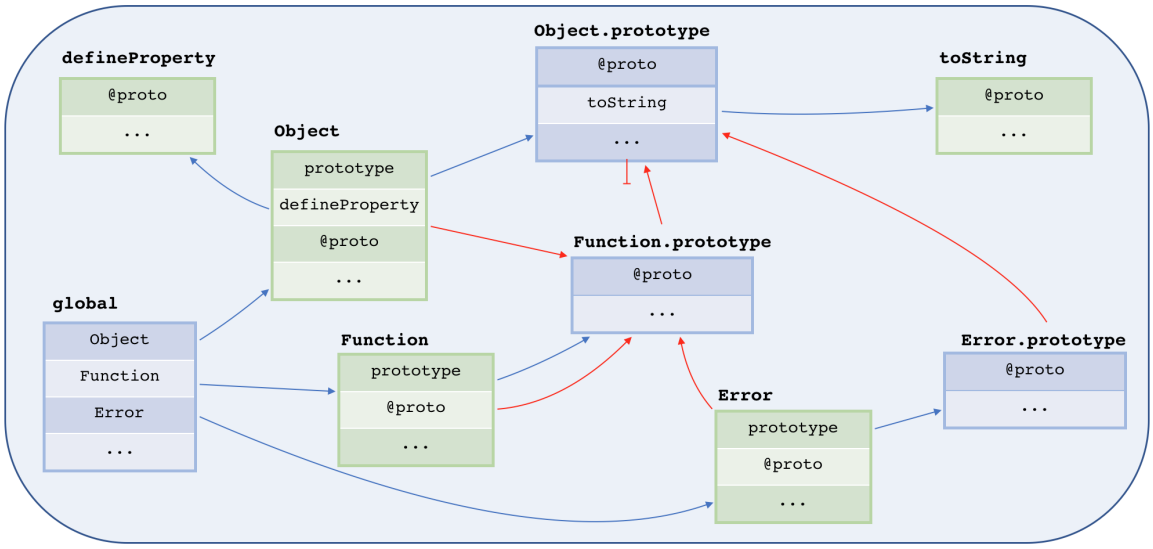


Figure 3.8.: Initial heap of the critical built-in objects

function anywhere else in its prototype chain.

3.1.4. Why ES5 Strict?

At the starting time of the project, ES5 was the current version of the standard, and the publication date of ES6 was too close to the end of the project for it to be properly updated. For this reason, we remain in the scope of ES5.

We have chosen to focus on ES5 Strict, a restricted variant of ES5 that intentionally has slightly different semantics compared with the full language and exhibits better behavioural properties. ES5 Strict has lexicographic scoping, requires explicit declaration of all variables before instantiation, does not allow assignment to certain key properties of the global and function objects, and makes error reporting explicit.

To ensure lexicographic scoping, ES5 Strict forbids the use of the `with` statement. When we use a variable in the strict mode, we statically know in which function it was defined. Without the `with` statement, a variable cannot be a property of an arbitrary object anymore. The only exception is the global object holding global variables.

In non-strict mode, when an error happens, it is silently ignored. However, some unexpected side effects can change the global state or some operations that are supposed to change the state, behave as no-op. This makes it more difficult to find the cause of a faulty program. The strict mode of the language helps finding errors earlier on in the development by throwing an exception. We give some examples below. Recall the `Node` function defined above, and consider the following:

```
1 var f1 = function(pri, val) { n = new Node(pri, val) }; // creates a global variable n in non-strict mode
2                                                         // throws a ReferenceError in strict mode
3 var f2 = function(pri, val) { var n = Node(pri, val) }; // creates global variables pri, val and next
4                                                         // in non-strict mode
5                                                         // throws a TypeError in strict mode
6 Object.defineProperty(Node.prototype, "pri", { value: 0, writable: false } );
7 var f3 = function(pri, val) { var n = new Node(pri, val) }; // creates a Node n without the property pri
8                                                               // in non-strict mode
9                                                               // throws a TypeError in strict mode
```

In non-strict mode, it is very easy to introduce global variables by forgetting to declare a variable

(variable `n`, line 1) or by calling a function as a standard function, hence, making `this` to point to the global object (function `Node`, line 3). In strict mode, a `ReferenceError` is thrown if a variable is used without declaring it first (line 1); and `this` holds the value `undefined` instead of the global object when a function is called as a standard function, resulting in a `TypeError` being thrown when we try to add a property to `undefined` (line 3). Assigning to a non-writable property of an object or to a property that is non-writable in the prototype chain of an object (property `pri`, line 7), silently skips the assignment operation in non-strict mode. In strict mode, a `TypeError` is thrown.

ES5 Strict was developed by the ECMAScript committee, and was recommended for use by the committee itself as well as by professional developers [28], and is also widely used by major industrial players: for example, Google's V8 JavaScript engine [31], and Facebook's React JavaScript library [25].

We believe that ES5 Strict is the correct starting point, since our overall aim is to *demonstrate the feasibility* of logic-based verification for JavaScript. Moreover, the work that we have done remains relevant for ES6 and ES7. Moving from ES5 Strict to ES6 Strict or ES7 Strict does require the addition of new language constructs, which constitutes a sizeable effort, but only very minor changes to the current infrastructure, as ES6 is built on top of ES5, and ES7 on top of ES6.

3.2. The Running Example

We explain the fundamentals of JavaScript by appealing to the example given in Figure 3.9, which is an implementation of a priority queue library in JavaScript, and the heap obtained from its execution, shown in Figure 3.10. We use this example to illustrate the complexity of programming in JavaScript and the behavioural properties that we wish to capture using JaVerT. In §8, we use JaVerT to specify and verify all of the functions associated with the library.

Our priority queue library is given in lines 1-48, with a small client program given in lines 50-54. To use the library, a client program constructs a new priority queue, identified by the variable `q`, by calling

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

Figure 3.9.: Running Example - a priority queue implemented in JavaScript

the `PriorityQueue` constructor (line 50). To manipulate the queue, `enqueue` and `dequeue` functions are called (lines 51-54). The `enqueue` function inserts the given priority and value to the queue, while the `dequeue` function retrieves the value with the highest priority. This is all the client program needs to know in order to use the priority queue library.

The library implements a priority queue as an object with a property `_head` pointing to a singly-linked list of node objects, with the nodes ordered in descending order of priority. When a new priority queue is constructed (line 50), the `PriorityQueue` function (lines 25-28) (we annotate all function literals with unique identifiers) initially sets `_head` to `null` of the new priority queue object. Each node contains a priority (`pri`), a value (`val`), and the pointer to the next node in the queue (`next`). The function `insertToQueue` (lines 9-23) inserts a node into an existing queue. This function is stored in the node prototype object (`Node.prototype`), and is available to all node objects. It should be used in the form `n.insertToQueue(q)`, where `q` is the head of the queue into which we are inserting and `n` is the node to be inserted. It returns the head of the new queue, obtained by correctly inserting `n` into the queue starting with `q`. The `enqueue` function (lines 30-34), for example, `q.enqueue(1, "last")` uses the `Node` function to construct a node object, `n` (line 32), with priority (`pri` equal to 1), value (`val` equal to `"last"`), and a pointer to the next node (initially `next` equal to `null`) (line 6). It then calls the function `n.insertToQueue(this._head)` which inserts `n` into an existing node list pointed by `this._head`, returning the head of the new node list (line 33).

Note that the function `insertToQueue` could be implemented either using a recursive function or a while loop. We chose to implement it using recursion. Doing so simplifies the specification of the function, as we do not need to write down the loop invariant.

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 variable `Node` is declared in an immediately-invoked function expression (lines 2-48). 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. Using the underscore prefix for denoting private property names is a convention broadly used by JavaScript developers. Even though it is possible for someone to abuse this interface, it does clarify the intention of keeping the properties private. In §8, we give specifications of the queue library functions that ensure functionally correct behaviour. 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. As we discussed in §3.1.3, before the execution of any JavaScript program, an initial heap is established. It contains the global object, which will hold all global variables, such as `PriorityQueue`, `q`, and `r` from the running example. The initial heap also contains the constructors and prototypes of all JavaScript built-in libraries, such as `Object`, `Function`, and `Error`. In the running example, we use the constructor of the `Error` built-in object to construct a new error object and throw an exception when trying to dequeue an empty queue (line 39).

Objects, Object Properties. Recall that JavaScript objects have two types of properties: *in-*

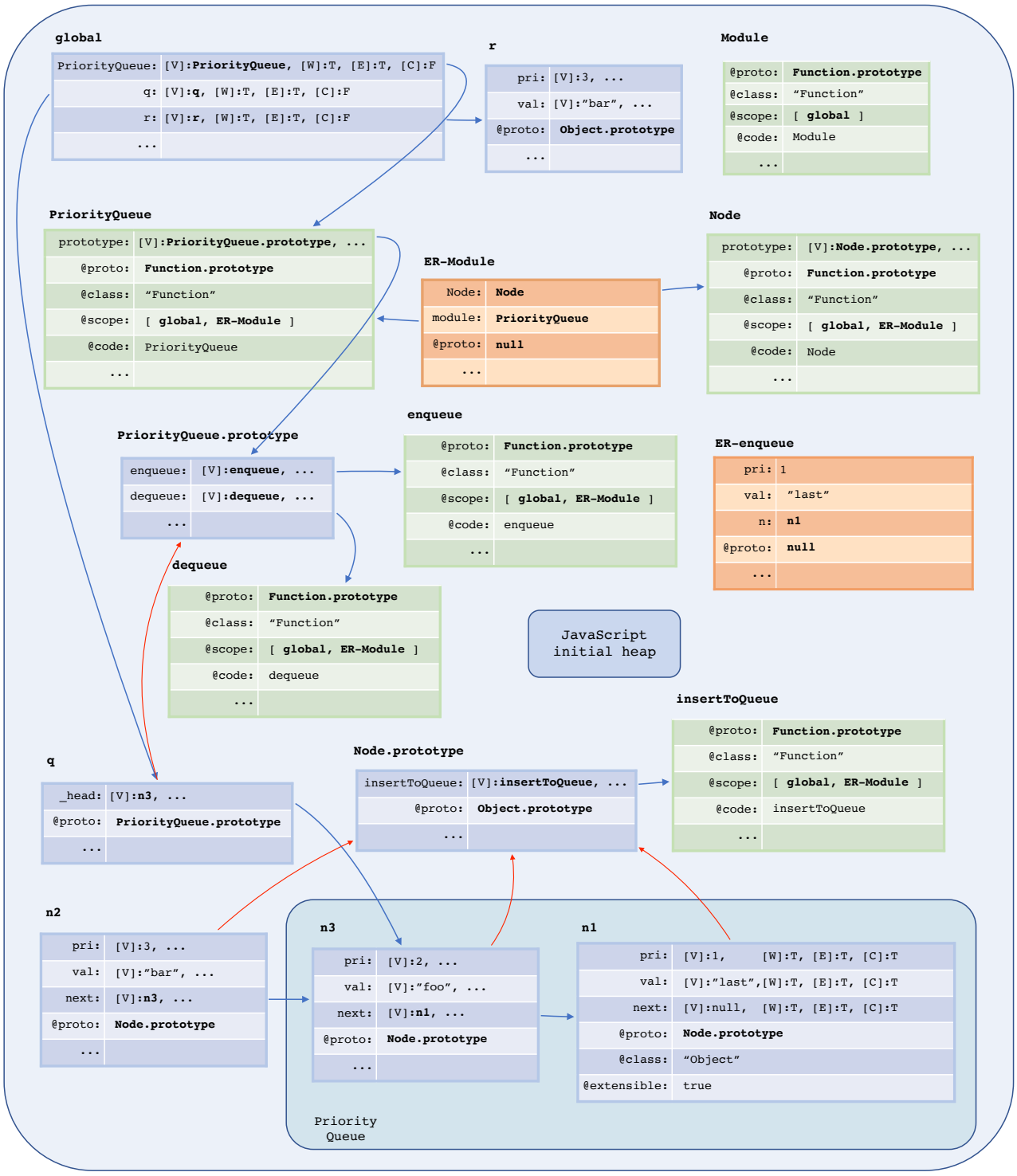


Figure 3.10.: JavaScript heap obtained from the execution of the running example

ternal and *named*. Standard JavaScript objects have three internal properties: `@proto`, `@class`, and `@extensible`. For example, as seen in Figure 3.10, object `n1` is extensible, its prototype is `Node.prototype`, and its class is `"Object"`.

Named properties are not associated with values in the heap, but instead with *property descriptors*. Property descriptors are quadruples of *attributes*, which describe the ways in which a property can be accessed and/or modified. Depending on the attributes they contain, named properties can either be

data properties or *accessor properties*. Data properties contain the value, writable, enumerable, and configurable attributes (denoted by [V], [W], [E], and [C]), whereas accessor properties contain get and set attributes (denoted by [G] and [S]), as well as [E] and [C]. The attributes have the following semantics: [V] holds the actual value of the property; [W] describes whether or not the value of the property can be changed; [E] indicates whether or not the property will be included in a for-in enumeration; [C] allows or disallows property deletion, together with any change to the other attributes (except for value, which it does not affect) and any change in the type of the property (data to accessor and vice versa); [G] and [S] play a role similar to getters and setters of Java and provide property encapsulation. Let us illustrate how JavaScript uses descriptors. If a property of an object was created using a property accessor (for example, `this.pri = pri`), it will be writable, enumerable, and configurable (for example, `"pri"`, `"val"`, and `"next"` in the object `n1`). On the other hand, if a property was declared as a variable, it will not be configurable (for example, `PriorityQueue` and `q` in the global object). That is, JavaScript variables, once declared, cannot be deleted.

Prototype-based inheritance. All node objects constructed using `new Node(...)` share the same prototype, which is the `Node.prototype` object. In order to determine the value of a property `p` of a given instance of `Node`, say `n1`, one needs to traverse its prototype chain.

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

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

The above code creates a non-writable property `"pri"` with value 0 in `Node.prototype`. While this may seem reasonable, as the default value of `"pri"` should not be updated, the JavaScript standard forbids assignment to a property of an object if a non-writable property of the same name exists in the prototype chain of that object. This means that any program executing `new Node(...)` will fail, as the code of `Node` includes an assignment to the property `"pri"`. We will see later how providing the specification for the `PriorityQueue` library enforces us to consider all such corner cases.

Functions, Function objects. Functions are also stored in the JavaScript heap as objects. For instance, the functions defined in the code are represented in the final heap by the objects labelled with their respective identifiers: `Module`, `Node`, `insertToQueue`, `PriorityQueue`, `enqueue`, and `dequeue`. As we have mentioned, each function object has three specific properties: `@code`, `@scope`, and `"prototype"`. All function objects from the running example store their corresponding unique identifiers in the property `@code`. The function `Module` has its `@scope` equal to [`global`], while the remaining five functions have their `@scope` equal to [`global`, `ER−Module`]. `ER−Module` is an environment record created for the immediately-invoked function expression `Module`. `Node.prototype` is the prototype of all instances of `Node`, such as `n1`, `n2` and `n3`, whereas `PriorityQueue.prototype` is the prototype of all instances of `PriorityQueue`, such as `q`. The rest of the functions also have `"prototype"` properties, but since we do not use them as constructors, they are not relevant.

The running example also illustrates a couple of interesting points about functions in JavaScript. First, a function can be returned as an outcome of another function. The function `Module` returns `module` on line 47, which is, in fact, a function object defined on line 26 and corresponds to the function object `PriorityQueue` in the heap. Second, JavaScript supports immediate invocation of function expressions. The function `Module` is being called right away after its definition (line 48) and

is not being assigned to any of the variables. Hence, we cannot access the function object `Module` anymore, as we can see from the JavaScript heap in Figure 3.10.

Variable Binding. The most interesting function to illustrate variable binding is the `enqueue` function, which is in a similar setting as the function `f` from the discussion on **Variable Binding** in in §3.1.1. It uses four variables `pri`, `val`, `n`, and `Node` in its body. Variables `pri` and `val` are formal parameters and `n` is a local variable, hence, they are stored in the new environment record `ER−enqueue`. Figure 3.10 shows the `ER−enqueue`, created upon the invocation of the first call to the function `enqueue` on line 51. However, `Node` is not a property of the `ER−enqueue`, so we have to look for it in the rest of the scope chain associated with `enqueue`, which is [`global`, `ER−Module`], and we find it in `ER−Module`. All other functions use only variables that are either their own formal parameters or local variables and are, therefore, stored in the environment records newly created upon function invocation.

The `this` Keyword. In the running example, we use `this` in functions `Node`, `PriorityQueue`, `insertToQueue`, `enqueue`, and `dequeue`. Its usage should not be confusing, since these functions are either constructors or methods. The functions `Node` and `PriorityQueue` are constructors and are called using the `new` expression, for example, `new Node(...)` on line 32 and `new PriorityQueue()` on line 50. The keyword `this` in the body of these functions will correspond to the newly created objects, `n1`, `n2`, `n3` (for `Node`) and `q` (for `PriorityQueue`). The other functions are called as methods `o.m(...)`, in which case the `this` keyword in their bodies correspond to the objects `o` (lines 20, 33, and 51-54).

3.3. The Memory Model of ES5 Strict

We formally define the memory model of full ES5 Strict. We use the memory model to prove a full correctness result of the translation from the JavaScript assertion language to the JSIL assertion language in §8.

We define JavaScript heaps in Figure 3.11. A JavaScript heap, $h \in \mathcal{H}_{\mathtt{JS}}$, is a partial function mapping pairs of object locations and property names to JS heap values. Object locations are taken from a set of locations $\mathcal{L}$. Property names are taken from a set of strings $\mathcal{P}_{\mathtt{JS}}$. JS literals contain: numbers, n; booleans, b; strings, t; and the special JavaScript values `undefined` and `null`. JS values contain JS literals, λ_{JS}, and object locations, l. JS heap values, $\omega \in \mathcal{V}^h_{\mathtt{JS}}$, contain: JS values, $v \in \mathcal{V}_{\mathtt{JS}}$; lists of JS values, $\overline{v}$; and function identifiers, m.

Lists of JS values, $\overline{v}$, are used to represent scope chains and descriptors. We represent a scope chain as a list of locations of environment records. For the descriptors, recall that there exist two types of descriptors: data and accessor. Data descriptors have four attributes: [`V`] holding JavaScript value, and three boolean attributes, writable [`W`], enumerable [`E`], and configurable [`C`]. Accessor descriptors have get [`G`] and set [`S`] attributes that hold either locations to a function objects or are `undefined`, together with two boolean attributes, enumerable [`E`] and configurable [`C`]. We represent descriptors as five-element lists; the first element describes the descriptor type and the remaining four represent values of appropriate attributes; for example, [`"d"`, `"foo"`, `true`, `false`, `true`] is a writable, non-enumerable, and configurable data descriptor with value `"foo"`, while [`"a"`, `g`, `undefined`, `false`, `true`] is a non-enumerable, and configurable accessor descriptor which has a getter function at location `g` and which setter is `undefined`. In the semantics we use a notation desc d to denote a property descriptor.

Function identifiers, m, are associated with syntactic functions in the JavaScript code and are used

to represent function bodies in the heap uniquely. This choice differs from the approach of [30], where function bodies are also JS heap values. The ECMAScript standard does not prescribe how function bodies should be represented and our choice closely connects JavaScript and JSIL heap models.

$$
\begin{array}{rl}
\textsc{Locations}: & l \in \mathcal{L} \\
\textsc{Property Names}: & p \in \mathcal{P}_{\mathsf{JS}} \subset \mathcal{S}tr \\
 & \\
\textsc{Numbers}: & n \in \mathcal{N}um \\
\textsc{Booleans}: & b \in \mathcal{B}ool \\
\textsc{Strings}: & t \in \mathcal{S}tr \\
 & \\
\textsc{JS literals}: & \lambda_{\mathsf{JS}} \in \mathcal{L}it_{\mathsf{JS}} \quad \triangleq \quad n \mid b \mid t \mid \mathsf{undefined} \mid \mathsf{null} \\
\textsc{JS values}: & v \in \mathcal{V}_{\mathsf{JS}} \quad \triangleq \quad \lambda_{\mathsf{JS}} \mid l \\
\textsc{JS heap values}: & \omega \in \mathcal{V}^h_{\mathsf{JS}} \quad \triangleq \quad v \mid \overline{v} \mid m \\
 & \\
\textsc{JS heaps}: & h \in \mathcal{H}_{\mathsf{JS}} \quad : \mathcal{L} \times \mathcal{P}_{\mathsf{JS}} \rightharpoonup \mathcal{V}^h_{\mathsf{JS}}
\end{array}
$$

Figure 3.11.: The Memory Model of ES5 Strict

Objects can be viewed as sets of heap cells with the same location, but different property names. We denote a heap cell by $(l, p) \mapsto \omega$, the union of two disjoint heaps by $h_1 \uplus h_2$, a heap lookup by $h(l, p)$, an update operation by $h[(l, p) \mapsto \omega]$, a cell deallocation by $h \backslash (l, p)$ and the empty heap by emp. We use $l \mapsto \{p_1 : \omega_1, \ldots, p_n : \omega_n\}$ as shorthand for $(l, p_1) \mapsto \omega_1 \uplus \ldots \uplus (l, p_n) \mapsto \omega_n$. This approach, in contrast to mapping locations to objects with all fields, reduces overhead and gives us precise footprints when writing separation logic specifications.

3.4. A Formal Fragment of ES5 Strict

We formally define a fragment of ES5 Strict (§3.4.1) together with its pretty-big-step operational semantics (§3.4.2). Later, we prove the correctness of the compiler for this fragment (§5.5). Using the pretty-big-step rules for the assignment expression, we illustrate how we closely follow the ES5 English standard (§3.4.3).

3.4.1. Syntax of the ES5 Strict Fragment

Figure 3.12 defines the subset of ES5 Strict that we will be formally considering in this thesis. JavaScript *expressions* include the this keyword, variables, literals, object initialisers, computed field accesses, constructor calls, function calls, function literals annotated with function identifiers, unary operators, binary operators and assignments. We consider two unary operators: the delete operator and the typeof operator, and three binary operators: addition, strict equality and strict inequality. JavaScript *statements* include sequence, variable declaration, the expression statement, the if-then-else statement, the while loop, the break statement, the throw statement and the return statement.

The given fragment is representative of the full core language of ES5 Strict. Recall the discussion about ES5 core language in Figures 3.4 and 3.5, where the statements and expressions included in our fragment are highlighted in green. Even though we exclude some of the syntax, we do not simplify the semantics of the chosen constructs. We leave out the `with` statement as it is forbidden in strict mode.

$$
\begin{array}{rrcl}
\text{JS EXPRESSIONS} : e & \in \mathcal{E}_{\mathsf{JS}} & \triangleq & \mathsf{this} \mid x \mid \lambda_{\mathsf{JS}} \mid \{\,\} \mid e[e] \mid \mathsf{new}\ e(\overline{e}) \mid e(\overline{e}) \mid \mathsf{function}\ (\overline{x})\{s\}^m \mid \\
 & & & \ominus\ e \mid e \oplus e \mid e = e \\
\text{JS UNARY OPERATORS} : \ominus & & \triangleq & \mathsf{delete} \mid \mathsf{typeof} \\
\text{JS BINARY OPERATORS} : \oplus & & \triangleq & + \mid === \mid !== \\
\text{JS STATEMENTS} : s & \in \mathcal{S}_{\mathsf{JS}} & \triangleq & s;\ s \mid \mathsf{var}\ x \mid e \mid \mathsf{if}(e)\,\{s\}\,\mathsf{else}\,\{s\} \mid \mathsf{while}(e)\,\{s\} \mid \\
 & & & \mathsf{break} \mid \mathsf{throw}\ e \mid \mathsf{return}\ e
\end{array}
$$

Figure 3.12.: A Fragment of ES5 Strict

The `switch` and `try−catch−finally` are too lengthy for handwritten formalisation, while `for−in` has non-deterministic semantics. We choose one iteration statement `while`, as `do−while` and `for` are only variations of it. From the expressions we leave out: array literals, as most of `Array` functionality is a part of the `Array` library and we concentrate on the core language; member accesses, as they are simplified version of computed accesses; compound assignments, since they can be desugared to simple assignments and binary operators; and some of unary and binary operators, while including a sample of them in the fragment. To consider the full language, it would only make sense to do a mechanised specification in the spirit of JSCert.

3.4.2. Pretty-Big-Step Semantics of the ES5 Strict Fragment

We introduce the semantic relation for ES5 Strict and give pretty-big-step rules of a selection of expressions, statements, and internal functions. The full operational semantics of our ES5 fragment is given in Appendix A: notation and auxiliary functions in §A.1; JavaScript expressions and statements in §A.2; the property descriptors in §A.3; the JavaScript internal functions in §A.4, §A.5, and §A.6; and the JavaScript built-in libraries in §A.7.

References. Before we introduce the semantic relation for ES5 Strict, we need to define *references*. References are internal constructs of JavaScript that appear, for example, as a result of evaluating a left-hand side of an assignment, and represent resolved property bindings. A reference consists 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*, `"o"`) or that of an environment record (*variable reference*, `"v"`). The global object is the only object that can be part of both object and variable references. To obtain the associated value, the reference needs to be dereferenced, which is performed by the `GetValue` internal function. In the ES5 standard, dereferencing an object reference is different from dereferencing a variable reference. For the former, one has to inspect the entire prototype chain of the object. For the latter, one only has to inspect the object itself. We present the precise dereferencing algorithm shortly. We formalise references as lists of three elements, containing the reference type (`"o"` or `"v"`), the base, and the property name. For example, `["v", o, "p"]` is a variable reference with the base `o` and the property name `"p"`. In the semantics we use a shorter notation $o.\mathsf{v}p$ for representing references.

ES5 Strict Semantic relation. We model the operational semantics of ES5 Strict in pretty-big-step style. The judgement is of the form:

$$\wp, L, v_t \vdash \langle h, s\rangle \Downarrow_m \langle h', o\rangle$$

where: $\wp$ is a JavaScript program and m is a function identifier; L is the current scope chain and $v_t \in \mathcal{V}_{\mathsf{JS}}$ is the this value (most commonly it is a location); h is the initial heap and s the statement (or the expression) to evaluate; h' is the final heap and o the *outcome* of the evaluation. The outcome (Figure 3.13) of an execution can be a normal outcome value w or it can be a returned value $\mathsf{ret}\, w$, an outcome value after a `break` statement $\mathsf{break}\, w$, or an error $\mathsf{error}\, w$. An outcome value can be: a value $v \in \mathcal{V}_{\mathsf{JS}}$; a list of values $\overline{v}$, that represents references and property descriptors; and the empty value. The judgement reads:

If we execute the statement s in the heap h, with the scope chain L and the this value v_t, then we arrive at the heap h', and the outcome of executing s is o.

$$\text{Outcome values} : w \triangleq v \mid \overline{v} \mid \mathsf{empty}$$
$$\text{Outcomes} : o \in \mathcal{O} \triangleq w \mid \mathsf{ret}\, w \mid \mathsf{break}\, w \mid \mathsf{error}\, w$$

Figure 3.13.: Internal Concepts of the Semantics

Prototype-based Inheritance. `GetProperty` is the JavaScript internal function that traverses the prototype chain. We define internal functions using pretty-big-step semantics in the same way as for ES5 Strict expressions and statements. `GetProperty` traverses the prototype chain and returns the property descriptor of the given property p first found in the prototype chain of the current this object (l_t), or undefined if absent. We denote the internal function `GetProperty(p)` by $\mathcal{I}_{gp}(p)$.

$$\textsc{GP-getOwn}\ \frac{\wp, L, l_t \vdash \langle h, \mathcal{I}_{gop}(p)\rangle \Downarrow_m \langle h_1, o\rangle \quad \wp, L, l_t \vdash \langle h_1, \mathcal{I}_{gp}(p, o)_1\rangle \Downarrow_m \langle h_f, o_f\rangle}{\wp, L, l_t \vdash \langle h, \mathcal{I}_{gp}(p)\rangle \Downarrow_m \langle h_f, o_f\rangle}$$

$$\textsc{GP-ownDef}\ \frac{}{\vdash \langle h, \mathcal{I}_{gp}(\_, \mathsf{desc}\ d)_1\rangle \Downarrow_m \langle h, \mathsf{desc}\ d\rangle}$$

$$\textsc{GP-ownUndef-protoNull}\ \frac{h(l_t, @proto) = \mathsf{null}}{\_, \_, l_t \vdash \langle h, \mathcal{I}_{gp}(\_, \mathsf{undefined})_1\rangle \Downarrow_m \langle h, \mathsf{undefined}\rangle}$$

$$\textsc{GP-ownUndef-protoNotNull}\ \frac{h(l_t, @proto) = l'_t \quad \wp, L, l'_t \vdash \langle h, \mathcal{I}_{gp}(p)\rangle \Downarrow_m \langle h_f, o_f\rangle}{\wp, L, l_t \vdash \langle h, \mathcal{I}_{gp}(p, \mathsf{undefined})_1\rangle \Downarrow_m \langle h_f, o_f\rangle}$$

$\mathcal{I}_{gp}(p)$ calls the internal function `GetOwnProperty(p)`[2], denoted by $\mathcal{I}_{gop}(p)$, which returns the property descriptor of the *own* property of the this object, or undefined if absent (rule Gp-getOwn). If the property is found, the corresponding property descriptor is returned (rule Gp-ownDef). Otherwise, the prototype of the object is inspected. If the prototype is null, undefined is returned (rule Gp-ownUndef-protoNull). Otherwise, `GetProperty` is recursively called for the prototype object (rule Gp-ownUndef-protoNotNull).

Function objects. When the function expression is evaluated (rule Function Literal), two new objects are created in the heap, the `prototype` object, l', and the function object itself, l:

$$\textsc{Function Literal}\ \frac{\begin{array}{l} h_f = h \uplus l' \mapsto \{@proto : l_{op},\ @class : \text{“}\texttt{Object}\text{”},\ @extensible : \mathsf{true}\} \uplus \\ l \mapsto \{@proto : l_{fp}, @class : \text{“}\texttt{Function}\text{”}, @extensible : \mathsf{true}, \text{“}\texttt{prototype}\text{”} : l', @scope : L, @code : m\} \end{array}}{\_, L, \_ \vdash \langle h, \mathsf{function}\ (\overline{x})\{s\}^m\rangle \Downarrow_m \langle h_f, l\rangle}$$

[2]The semantics of `GetOwnProperty` is given in Appendix A.4.

Figure 3.10 shows the objects `PriorityQueue`, `PriorityQueue.prototype`, `Node`, and `Node.prototype`, created on evaluation of function expressions of the running example in Figure 3.9, lines 5 and 26.

Variable binding. 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`. To capture this, the ES5 standard uses an auxiliary internal function `GetIdentifierReference(x)`, which we denote by $\mathcal{I}_\sigma(x)$ (rule VARIABLE, below). The result of $\mathcal{I}_\sigma(x)$ is either a location of the environment record within which the variable x is defined or undefined if x cannot be found anywhere in the scope chain. The result of variable resolution is always a variable reference $v.\mathsf{v}x$ (rule VARIABLE-REF).

$$\textsc{Variable}\quad \frac{\wp, L, v_t \vdash \langle h, \mathcal{I}_\sigma(x)\rangle \Downarrow_m \langle h_1, o_1\rangle \quad \wp, L, v_t \vdash \langle h_1, id(x, o_1)\rangle \Downarrow_m \langle h_f, o_f\rangle}{\wp, L, v_t \vdash \langle h, x\rangle \Downarrow_m \langle h_f, o_f\rangle}$$

$$\textsc{Variable-ref}\quad \vdash \langle h, id(x, v)\rangle \Downarrow_m \langle h, v.\mathsf{v}x\rangle$$

Note that all scope chains begin with the global object. Hence, if we do not find a binding for `x` in the rest of the scope, we traverse the prototype chain of the global object. If a variable x is found in the environment record l, then this environment record l is returned (rule GIR-CURRENT). Otherwise, $\mathcal{I}_\sigma(x)$ is evaluated in the rest of the scope chain (rule GIR-NEXT). If x could not be found in the rest of the scope chain, we check if it is defined in the prototype chain of the global object l_g. To do that, we use the internal function `HasProperty(x)`, denoted by $\mathcal{I}_{hp}(x)$, which returns true *iff* the this object, in our case the global object l_g, has the specified property x in its prototype chain (rule GIR-HASPROP). If $\mathcal{I}_{hp}(x)$ returns true, the result of $\mathcal{I}_\sigma(x)$ is l_g (rule GIR-GLOBAL), otherwise the result is undefined (rule GIR-UNDEF).

$$\textsc{Gir-current}\quad \frac{(l, x) \in \mathsf{dom}(h) \quad l \neq l_g}{\_, L \mathbin{@} [l], \_ \vdash \langle h, \mathcal{I}_\sigma(x)\rangle \Downarrow_m \langle h, l\rangle}$$

$$\textsc{Gir-next}\quad \frac{(l, x) \notin \mathsf{dom}(h) \quad l \neq l_g \quad \wp, L, v_t \vdash \langle h, \mathcal{I}_\sigma(x)\rangle \Downarrow_m \langle h_f, o\rangle}{\wp, L \mathbin{@} [l], v_t \vdash \langle h, \mathcal{I}_\sigma(x)\rangle \Downarrow_m \langle h_f, o\rangle}$$

$$\textsc{Gir-hasProp}\quad \frac{\wp, [l_g], l_g \vdash \langle h, \mathcal{I}_{hp}(x)\rangle \Downarrow_m \langle h, o_1\rangle \quad \wp, [l_g], v_t \vdash \langle h, \mathcal{I}_\sigma(o_1)_1\rangle \Downarrow_m \langle h_f, o_f\rangle}{\wp, [l_g], v_t \vdash \langle h, \mathcal{I}_\sigma(x)\rangle \Downarrow_m \langle h_f, o_f\rangle}$$

$$\textsc{Gir-global}\quad \vdash \langle h, \mathcal{I}_\sigma(\mathsf{true})_1\rangle \Downarrow_m \langle h, l_g\rangle$$

$$\textsc{Gir-undef}\quad \vdash \langle h, \mathcal{I}_\sigma(\mathsf{false})_1\rangle \Downarrow_m \langle h, \mathsf{undefined}\rangle$$

In our running example, all variables except for one are resolved in the environment record associated with the function in which they were defined. Only the variable `Node` in the function `enqueue` (line 32) is not defined in the function `enqueue`, but instead in the function `Module`. In Figure 3.10, it is a property of the object `ER−Module`.

Dereferencing. Evaluation of variables and property accesses results in a reference. `GetValue(w)`, denoted by $\mathcal{I}_{gv}(w)$, is the JavaScript internal function that performs dereferencing to obtain the corresponding value. It takes one parameter: the outcome value `w` to be dereferenced. If `w` is not a reference, it is returned immediately (rule GV-NOTREFERENCE[3]). If `w` is a reference whose base is `undefined`, a

[3]Note that $v \in \mathcal{V}_{\text{JS}}$, defined in Figure 3.11, cannot be a reference, hence, we do not add a restriction on v in the GV-NOTREFERENCE rule.

JavaScript reference error, represented by $\mathsf{err}(l, l_{rep})$, is thrown (rule Gv-unresolvable). $\mathsf{err}(l, l_{rep})$ defines a newly created object l, whose prototype is `ReferenceError.prototype`, l_{rep}. If `w` is a reference with a primitive base (that is, whose base is not an object location), a special JavaScript internal `Get` function, which we denote by $\mathcal{I}^i_g(p)$, is called (rule Gv-primitiveBase, more details on $\mathcal{I}^i_g(p)$ can be found in Appendix A.6). Otherwise, `w` $= l.\mathsf{a}p$ and, in that case, `GetValue` returns the value associated with the property p of object l. If `w` is a variable reference whose base is not the global object, this value is obtained by directly inspecting the heap (rule Gv-variableReferenceNotLg). Otherwise, `GetValue` uses the `Get` internal function, $\mathcal{I}_g(p)$ (Appendix A.4), to traverse the prototype chain and obtain the appropriate value (rules Gv-noPrimitiveBase and Gv-variableReferenceLg).

$$\textsc{Gv-notReference}\quad \vdash \langle h, \mathcal{I}_{gv}(v)\rangle \Downarrow_m \langle h, v\rangle$$

$$\textsc{Gv-unresolvable}\quad \frac{h_f = h \uplus \mathsf{err}(l, l_{rep})}{\vdash \langle h, \mathcal{I}_{gv}(\mathsf{undefined}.\mathsf{a}p)\rangle \Downarrow_m \langle h_f, \mathsf{error}\, l\rangle}$$

$$\textsc{Gv-noPrimitiveBase}\quad \frac{\wp, L, l \vdash \langle h, \mathcal{I}_g(p)\rangle \Downarrow_m \langle h_f, o_f\rangle}{\wp, L, v_t \vdash \langle h, \mathcal{I}_{gv}(l.\mathsf{o}p)\rangle \Downarrow_m \langle h_f, o_f\rangle}$$

$$\textsc{Gv-primitiveBase}\quad \frac{v \notin \mathcal{L} \quad \wp, L, v \vdash \langle h, \mathcal{I}^i_g(p)\rangle \Downarrow_m \langle h_f, o_f\rangle}{\wp, L, v_t \vdash \langle h, \mathcal{I}_{gv}(v.\mathsf{o}p)\rangle \Downarrow_m \langle h_f, o_f\rangle}$$

$$\textsc{Gv-variableReferenceLg}\quad \frac{\wp, L, l_g \vdash \langle h, \mathcal{I}_g(p)\rangle \Downarrow_m \langle h_f, o_f\rangle}{\wp, L, v_t \vdash \langle h, \mathcal{I}_{gv}(l_g.\mathsf{v}p)\rangle \Downarrow_m \langle h_f, o_f\rangle}$$

$$\textsc{Gv-variableReferenceNotLg}\quad \frac{l \neq l_g \quad v = h(l, p)}{\vdash \langle h, \mathcal{I}_{gv}(l.\mathsf{v}p)\rangle \Downarrow_m \langle h, v\rangle}$$

`GetValue` is used very frequently in the semantics after evaluating an expression. For this reason, we define the notation $\wp, L, v_t \vdash \langle h, e\rangle \Downarrow^{\gamma}_m \langle h', o\rangle$ to mean that we first evaluate an expression e (rule GetValue) and then call `GetValue` (rule GetValue-ref) on the obtained reference:

$$\textsc{GetValue}\quad \frac{\wp, L, v_t \vdash \langle h, e\rangle \Downarrow_m \langle h_1, o_1\rangle \quad \wp, L, v_t \vdash \langle h_1, \gamma(o_1)\rangle \Downarrow_m \langle h_f, o_f\rangle}{\wp, L, v_t \vdash \langle h, e\rangle \Downarrow^{\gamma}_m \langle h_f, o_f\rangle}$$

$$\textsc{GetValue-ref}\quad \frac{\wp, L, v_t \vdash \langle h, \mathcal{I}_{gv}(w)\rangle \Downarrow_m \langle h_f, o_f\rangle}{\wp, L, v_t \vdash \langle h, \gamma(w)\rangle \Downarrow_m \langle h_f, o_f\rangle}$$

Function, Method and Constructor Calls. In JavaScript, functions can be called in three different ways: as standard functions, as methods, and as constructors. Usually, a function is designed to be called in one way only. In our running example, the function `Module` is called as a standard function, the functions `Node` and `PriorityQueue` are called as constructors, and the remaining functions, `insertToQueue`, `enqueue` and `dequeue`, are called as methods. Below, we give the pretty-big-step rules for these three types of calls. Since function and method calls share the same syntactic structure, we use the same set of rules to describe them.

Given a function or method call $e(\overline{e})$, we first evaluate the function object e (rules Function Call, Function Call - 1), and its arguments $\overline{e}$ (rule Function Call - 2). Instead of using $\Downarrow^{\gamma}_m$, we first evaluate e (rule Function Call) and then call `GetValue` (rule Function Call - 1) separately, since we need both the reference w and its value v. To evaluate arguments, we use a helper notation $\mathsf{iterate}\{\overline{e}\}$, which returns the list of values obtained by evaluating and dereferencing each expression in $\overline{e}$. The formal definition of $\mathsf{iterate}\{\overline{e}\}$ is given in §A.5. If v is not callable, determined by the predicate

$\neg\mathcal{P}_c(h,v)$[4], a `TypeError` occurs, where $\mathsf{err}(l, l_{tep})$ defines a newly created object l, whose prototype is `TypeError.prototype`, l_{tep} (rule Function Call - 3 (Fault)). Otherwise, we gather information about the function to be called: we select the this value; obtain function identifier m', its scope L, and its formal parameters $\overline{x}$; and if there are less arguments provided than the function expects, the remaining values are set to undefined (rule Function Call - 3). The definition of the $\mathsf{SelectThis}(w)$ function that is used in rule Function Call - 3 is:

$$\mathsf{SelectThis}(w) = \begin{cases} l & \text{if } w = l.\mathtt{o}x \\ \mathsf{undefined} & \text{otherwise} \end{cases}$$

It captures the difference between function and method calls, and states that if we are given an object reference (method call), the this value should be the base of that reference, l, and undefined otherwise (function call).

Function Call

$$\frac{\wp, L, v_t \vdash \langle h, e\rangle \Downarrow_m \langle h_1, o_1\rangle \quad \wp, L, v_t \vdash \langle h_1, o_1(\overline{e})_1\rangle \Downarrow_m \langle h_f, o_f\rangle}{\wp, L, v_t \vdash \langle h, e(\overline{e})\rangle \Downarrow_m \langle h_f, o_f\rangle}$$

Function Call - 1

$$\frac{\wp, L, v_t \vdash \langle h, \mathcal{I}_{gv}(w)\rangle \Downarrow_m \langle h_1, o_1\rangle \quad \wp, L, v_t \vdash \langle h_1, (w, o_1)(\overline{e})_2\rangle \Downarrow_m \langle h_f, o_f\rangle}{\wp, L, v_t \vdash \langle h, w(\overline{e})_1\rangle \Downarrow_m \langle h_f, o_f\rangle}$$

Function Call - 2

$$\frac{\wp, L, v_t \vdash \langle h, \mathsf{iterate}\{\overline{e}\}\rangle \Downarrow_m \langle h_1, \overline{v}\rangle \quad \wp, L, v_t \vdash \langle h_1, (w, v)(\overline{v})_3\rangle \Downarrow_m \langle h_f, o_f\rangle}{\wp, L, v_t \vdash \langle h, (w, v)(\overline{e})_2\rangle \Downarrow_m \langle h_f, o_f\rangle}$$

Function Call - 3

$$\frac{\begin{array}{l}\mathcal{P}_c(h,v) \quad v_t = \mathsf{SelectThis}(w) \\ m' = h(v, @code) \quad L = h(v, @scope) \\ \wp(m') = \lambda x_1, ...x_{n_2}.s \\ \forall_{1\le n\le n_1} v'_n = v_n \\ \forall_{n_1<n\le n_2} v'_n = \mathsf{undefined} \\ \wp, L, v_t \vdash \langle h, m'(\overline{x}, \overline{v'})\rangle \Downarrow_{m'} \langle h_f, o_f\rangle\end{array}}{\wp, \_, \_ \vdash \langle h, (w, v)(v_1, ...v_{n_1})_3\rangle \Downarrow_m \langle h_f, o_f\rangle}$$

Function Call - 3 (Fault)

$$\frac{\neg\mathcal{P}_c(h,v) \quad h_f = h \uplus \mathsf{err}(l, l_{tep})}{\vdash \langle h, (w, v)(\overline{v})_3\rangle \Downarrow_m \langle h_f, \mathsf{error}\, l\rangle}$$

Call

$$\frac{\wp(m) = \lambda\overline{x}.s \quad \wp, L @ [l_s], v_t \vdash \langle h \uplus \mathsf{env}_m(l_s, \overline{x}, \overline{v}, s), s\rangle \Downarrow_m \langle h_f, o\rangle}{\wp, L, v_t \vdash \langle h, m(\overline{x}, \overline{v})\rangle \Downarrow_m \langle h_f, \mathsf{FunRet}(o)\rangle}$$

In our running example, for method call `q.enqueue(1, "last")` on line 51, `q.enqueue` evaluates to an object reference, hence this will be given the value `q`. The function call on line 48 evaluates the function expression to an object location, and in that case the this value gets the value `undefined`. What $m(\overline{x}, \overline{v})$ does is that it obtains function m body $\lambda\overline{x}.s$ from the program $\wp$, and executes the function body s in the scope $L @ [l_s]$, obtained by appending fresh location l_s to the end of the scope chain L (rule Call). The environment record contains the function arguments and local variables: $\mathsf{env}_m(l_s, \overline{x}, \overline{v}, s) \triangleq (\uplus_{i=1}^{n}(l_s, \overline{x}_i) \mapsto \overline{v}_i) \uplus (\uplus_{i=1}^{m}(l_s, y_i) \mapsto \mathsf{undefined})$, where $y_1, ..., y_m$ are local variables of s. Finally, the result of function needs to be determined. If the evaluation of the body of the function terminates with a return statement, that is, $o = \mathsf{ret}\, v$, the result of function call is the value v. If

[4]The definition of $\mathcal{P}_c(h,v)$ is given in the Appendix §A.5

the function body throws an exception, that is, $o = \mathsf{error}\, v$, the result of the function call is the same error $\mathsf{error}\, v$. In other cases, the result of the function call is $\mathsf{undefined}$. This behaviour is captured by the auxiliary function $\mathsf{FunRet}(o)$. In our running example, the functions `Module`, `insertToQueue`, and `dequeue` have return statements and calls of them return either a value provided by the return statement or an error if an exception is thrown. The function `enqueue` does not have a return statement, hence the call to this function returns `undefined` or it might result in an exception.

Operational semantics of constructor calls is similar to that of the function and method calls. Given a constructor call $\mathsf{new}\ e(\overline{e})$, we first evaluate the function object e (Constructor Call), and its arguments $\overline{e}$ (Constructor Call - 1). If v is not callable, an error occurs (Constructor Call - 2 (Fault)). Otherwise, a new object l_o is created with its internal property $@proto$ set to the value (l') of the internal property `"prototype"` of the constructor object (Constructor Call - 2). In the constructor case, we do not need to select the this value, as it will always hold the location of the newly created object l_o. $\mathsf{SelectProto}(v)$ makes sure that the prototype of the newly created object is actually an object. If a property `"prototype"` of a constructor object does not hold an object, a newly created object l_o will have `Object.prototype` as its prototype. The result of constructor call usually is the newly created object l_o, except for two cases. First, if the body of the constructor throws an error, that is, $o = \mathsf{error}\, v$, then the result of the constructor is $\mathsf{error}\, v$. Second, if the body of the constructor returns an object, that is, $o = \mathsf{ret}\, l_r$, the result of the constructor call is the object l_r, not the newly created object l_o. This behaviour is captured by $\mathsf{ConsRet}(o, l_o)$. In our running example, both the constructors `Node` and `PriorityQueue` do not have return statements in their bodies. Hence, the result of calling each of them as a constructor will be the newly created object.

Constructor Call

$$\frac{\wp, L, v_t \vdash \langle h, e\rangle \Downarrow_m^{\gamma} \langle h_1, o_1\rangle \quad \wp, L, v_t \vdash \langle h_1, \mathsf{new}_1\ o_1(\overline{e})\rangle \Downarrow_m \langle h_f, o_f\rangle}{\wp, L, v_t \vdash \langle h, \mathsf{new}\ e(\overline{e})\rangle \Downarrow_m \langle h_f, o_f\rangle}$$

Constructor Call - 1

$$\frac{\wp, L, v_t \vdash \langle h, \mathsf{iterate}\{\overline{e}\}\rangle \Downarrow_m \langle h_1, \overline{v}\rangle \quad \wp, L, v_t \vdash \langle h_1, \mathsf{new}_2\ v(\overline{v})\rangle \Downarrow_m \langle h_f, o_f\rangle}{\wp, L, v_t \vdash \langle h, \mathsf{new}_1\ v(\overline{e})\rangle \Downarrow_m \langle h_f, o_f\rangle}$$

Constructor Call - 2 (Fault)

$$\frac{\neg \mathcal{P}_c(h, v) \quad h_f = h \uplus \mathsf{err}(l, l_{tep})}{\vdash \langle h, \mathsf{new}_2\ v(\overline{v})\rangle \Downarrow_m \langle h_f, \mathsf{error}\, l\rangle}$$

Constructor Call - 2

$$\frac{\begin{array}{l}\mathcal{P}_c(h, l) \quad v = h(l, \text{"}\mathsf{prototype}\text{"}) \quad l' = \mathsf{SelectProto}(v)\\ h_1 = h \uplus l_o \mapsto \{@proto\colon l',\ @class\colon \text{“}\texttt{Object}\text{”},\ @extensible\colon \mathsf{true}\}\\ m' = h(l, @code) \quad L = h(l, @scope)\\ \wp(m') = \lambda x_1, ...x_{n_2}.s\\ \forall_{1\le n\le n_1} v'_n = v_n\\ \forall_{n_1<n\le n_2} v'_n = \mathsf{undefined}\\ \wp, L, l_o \vdash \langle h, m'(\overline{x}, \overline{v'})\rangle \Downarrow_{m'} \langle h_f, o\rangle\end{array}}{\wp, \_, \_ \vdash \langle h, \mathsf{new}_2\ l(v_1, ...v_{n_1})\rangle \Downarrow_m \langle h_f, \mathsf{ConsRet}(o, l_o)\rangle}$$

3.4.3. Following the ES5 Standard

Our operational semantics closely follows the ECMAScript 5 English standard and was inspired by the operational semantics of JSCert [9], the recent Coq specification of the ES5 standard. As in JSCert, we follow the ECMAScript standard step-by-step by using the pretty-big-step style of semantics [15]. Let us illustrate this close relationship with the English standard using the assignment expression, whose evaluation, as described by the standard, is given in Figure 3.14.

1. In the first step, we evaluate the left-hand side expression of the assignment. This corresponds to the rule Assignment - 1, shown below.

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*.

Figure 3.14.: An assignment defined in English standard.

2.-3. Next, we do the same for the right-hand side of the assignment. The obtained right-hand side reference is dereferenced and we get the actual value to be assigned. The dereferencing is done by the `GetValue` internal function (§8.7.1 of the ES5 standard). Since we use the special notation $\Downarrow_m^\gamma$ for an expression evaluation followed by dereferencing, we have only one rule for steps 2 and 3, Assignment - 2 and 3.

4. In ES5 Strict, the identifiers `eval` and `arguments` may not appear as the left-hand side of an assignment (for example, `eval` = 42), and this step enforces this restriction. The corresponding rule is Assignment - 4.

5. The actual assignment is performed by calling the internal `PutValue` function (§8.7.2 of the ES5 standard). The corresponding rule is Assignment - 5, where `PutValue` is denoted by $\mathcal{I}_{pv}(w_1, v_2)$.

6. In JavaScript, every expression and statement returns a value. Here we return a value of dereferenced right-hand side expression which corresponds to the rule Assignment - 6.

Assignment - 1

$$\frac{\wp, L, v_t \vdash \langle h, e_1\rangle \Downarrow_m \langle h_1, o_1\rangle \quad \wp, L, v_t \vdash \langle h_1, o_1 =_1 e_2\rangle \Downarrow_m \langle h_f, o_f\rangle}{\wp, L, v_t \vdash \langle h, e_1 = e_2\rangle \Downarrow_m \langle h_f, o_f\rangle}$$

Assignment - 2 and 3

$$\frac{\wp, L, v_t \vdash \langle h, e_2\rangle \Downarrow_m^\gamma \langle h_1, o_1\rangle \quad \wp, L, v_t \vdash \langle h_1, w_1 =_2 o_1\rangle \Downarrow_m \langle h_f, o_f\rangle}{\wp, L, v_t \vdash \langle h, w_1 =_1 e_2\rangle \Downarrow_m \langle h_f, o_f\rangle}$$

Assignment - 4

$$\frac{p \in \{\mathsf{eval}, \mathsf{arguments}\} \quad h_f = h \uplus \mathsf{err}(l', l_{sep})}{\vdash \langle h, l.{}_{\mathsf{v}}p =_2 v\rangle \Downarrow_m \langle h_f, \mathsf{error}\, l'\rangle}$$

Assignment - 5

$$\frac{\begin{array}{l}(w_1 = v_1 \vee w_1 = l.{}_{\mathsf{o}}p \vee \\ (w_1 = l.{}_{\mathsf{v}}p \wedge p \notin \{\mathsf{eval}, \mathsf{arguments}\})) \\ \wp, L, v_t \vdash \langle h, \mathcal{I}_{pv}(w_1, v_2)\rangle \Downarrow_m \langle h_1, o_1\rangle \\ \wp, L, v_t \vdash \langle h_1, o_1 =_3 v_2\rangle \Downarrow_m \langle h_f, o_f\rangle\end{array}}{\wp, L, v_t \vdash \langle h, w_1 =_2 v_2\rangle \Downarrow_m \langle h_f, o_f\rangle}$$

Assignment - 6

$$\vdash \langle h, \_ =_3 v\rangle \Downarrow_m \langle h, v\rangle$$

Summary. We described the JavaScript language and presented a priority queue implementation as our running example. We formally defined the memory model of full ES5 Strict, and introduced a representative fragment of ES5 Strict with its operational semantics, illustrating the complexity of the language. In §5, we use the operational semantics of the fragment to prove the correctness of

the formally defined part of the JS-2-JSIL compiler. In §8, we use the memory model to prove a full correctness result of the translation from the JavaScript assertion language to the JSIL assertion language.

4. The JSIL Language

We introduce JSIL, our intermediate language for JavaScript verification. JSIL is a simple goto language with top-level procedures and commands operating on object heaps. It is dynamic in the sense that it supports extensible objects and dynamic evaluation of properties as well as procedure names. Having gotos in an intermediate representation for JavaScript verification is sensible, for three reasons: verification tools, based on separation logic, commonly have gotos; JavaScript has complex control flow statements with many corner cases (for example, switch, try/catch/finally), which can be naturally decompiled to gotos; JavaScript supports a restricted form of goto statements, via labelled statements, breaks, and continues. With JSIL, we aim at: a minimal language that can capture the assorted control flow commands of ES5 Strict; a language with heaps similar to ES5 Strict heaps, as we are aiming at verifying properties of the heap; a target language for a translation that precisely follows the semantics of ES5 Strict.

We define JSIL syntax (§4.1), semantics (§4.2), and give an example of a JSIL procedure (§4.3).

4.1. The JSIL Syntax

The syntax of JSIL is defined in Figure 4.1. Most syntactic constructs of JSIL either directly emulate those of JavaScript or are required or useful for JavaScript verification. We believe, however, that JSIL does stand as a low-level language of its own and that it can be reused for other types of analysis on other programming languages.

JSIL primitive literals include, first of all, JavaScript primitive values: numbers n, booleans b, strings t and the special values undefined and null. To capture the return values of JavaScript statements, we also include the special value empty. In contrast to JavaScript, we explicitly include object literals l and type literals τ. We use object literals to refer to JavaScript built-in objects, such as the `global` object l_g and `Object.prototype` l_{op}. Type literals τ are needed to reason about the types of literals (every literal has its corresponding type) and expressions. Finally, primitive literals include procedure identifiers m.

JSIL literals include primitive literals and literal lists $\overline{\lambda}_p$, as JavaScript descriptors, references, and scope chains all share a structure that is naturally compiled to lists in JSIL. We also use the notation $[\lambda_p, ..., \lambda_p]$ for a literal list.

JSIL expressions include JSIL literals, JSIL variables `x`, and a variety of unary and binary operators. The unary and binary operators contain the standard arithmetic ($+$, $-$, $*$, $/$, $\%$), comparison ($=$, $<$, $<_s$ (lexicographic comparison of strings)), boolean (not, and, or), list (head, tail, length, :: (list append), @ (list concatenation), nth (n-th element of a list), ord (returns true when applied to a list of lexicographically ordered strings)) and string (length $_s$, @ $_s$ (string concatenation), nth $_s$ (n-th element of a string)) operators. Additionally, due to JavaScript type coercions, we include the conversion operators from numbers to strings and vice versa (toString and toNumber), as well as the typeOf

NUMBERS: $n \in \mathcal{N}um$	BOOLEANS: $b \in \mathcal{B}ool$	STRINGS: $t \in \mathcal{S}tr$ LOCATIONS: $l \in \mathcal{L}$
VARIABLES: $\mathtt{x} \in \mathcal{X}_{\mathtt{JSIL}}$	PRIMITIVE LITERALS: $\lambda_p \triangleq$	$n \mid b \mid t \mid \mathsf{undefined} \mid \mathsf{null} \mid \mathsf{empty} \mid l \mid \tau \mid m$
TYPES :	$\tau \in \mathtt{Types} \triangleq$	$\mathsf{Num} \mid \mathsf{Bool} \mid \mathsf{Str} \mid \mathsf{Undef} \mid \mathsf{Null} \mid \mathsf{Empty} \mid \mathsf{Obj} \mid \mathsf{Type} \mid \mathsf{List}$
LITERALS :	$\lambda \in \mathcal{L}it \triangleq$	$\lambda_p \mid \overline{\lambda_p}$
EXPRESSIONS :	$\mathtt{e} \in \mathcal{E}_{\mathtt{JSIL}} \triangleq$	$\lambda \mid \mathtt{x} \mid \ominus\ \mathtt{e} \mid \mathtt{e} \oplus \mathtt{e}$
UNARY OPERATORS :	$\ominus \triangleq$	$\mathsf{not} \mid \mathsf{head} \mid \mathsf{tail} \mid \mathsf{length} \mid \mathsf{length}_s \mid$ $\mathsf{toString} \mid \mathsf{toNumber} \mid \mathsf{typeOf} \mid \mathsf{ord} \mid ...$
BINARY OPERATORS :	$\oplus \triangleq$	$+ \mid - \mid * \mid / \mid \% \mid = \mid < \mid <_s \mid$ $\mathsf{and} \mid \mathsf{or} \mid :: \mid @ \mid \mathsf{nth} \mid @_s \mid \mathsf{nth}_s \mid ...$
BASIC COMMANDS :	$\mathtt{bc} \in \mathtt{BCmd} \triangleq$	$\mathsf{skip} \mid \mathtt{x} := \mathtt{e} \mid \mathtt{x} := \mathsf{new}() \mid \mathtt{x} := [\mathtt{e}, \mathtt{e}] \mid [\mathtt{e}, \mathtt{e}] := \mathtt{e} \mid \mathsf{delete}(\mathtt{e}, \mathtt{e}) \mid$ $\mathtt{x} := \mathsf{hasProperty}(\mathtt{e}, \mathtt{e}) \mid \mathtt{x} := \mathsf{getProperties}(\mathtt{e})$
COMMANDS :	$\mathtt{c} \in \mathtt{Cmd} \triangleq$	$\mathtt{bc} \mid \mathsf{goto}\ i \mid \mathsf{goto}\ [\mathtt{e}]\ i,\ j \mid \mathtt{x} := \mathtt{e}(\overline{\mathtt{e}})\ \mathsf{with}\ j \mid$ $\mathtt{x}_1, ..., \mathtt{x}_n := \phi(\overline{\mathtt{x}_1}; ...; \overline{\mathtt{x}_n})$
PROCEDURES :	$\mathtt{proc} \in \mathtt{Proc} \triangleq$	$\mathsf{proc}\ m(\overline{\mathtt{x}})\{\overline{\mathtt{c}}\}$

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

Figure 4.1.: Syntax of the JSIL Language.

operator, which returns the type of an expression. There is also a variety of bitwise and mathematical operators needed to support all JavaScript operators; for more details, we refer the reader to [66]. The difference between JavaScript operators and JSIL operators is that JSIL operators only work for operands of specific types (for example, the + operator of JSIL works for numbers only), while JavaScript operators coerce expressions to specific types using implicit type conversion (for example, the + operator of JavaScript has no issues with taking two operands of object type). The advantages of having type-specific operators include simpler semantics of the operators, no side effects, and the ability to use these operators in logical specifications.

JSIL basic commands provide the essential machinery for the management of extensible objects, and do not affect control flow. They include the skip command, variable assignment, object creation, as well as a number of dynamic operations on objects: property access, property assignment, property deletion, membership check, and property collection. Out of these commands, only property collection (getProperties) is non-standard. It obtains the names of all properties of a given object and it is required for the compilation of the JavaScript `for-in` construct.

JSIL commands comprise basic commands and control flow commands. Commands related to control flow include conditional and unconditional gotos, procedure calls with dynamic evaluation of their names, and ϕ-node commands, used for reasoning. The two goto commands are standard: goto i transfers control to the i-th command of the active procedure, and goto $[\mathtt{e}]\ i,\ j$ transfers control to the i-th command if $\mathtt{e}$ evaluates to true, and to the j-th otherwise. The dynamic procedure call $\mathtt{x} := \mathtt{e}(\overline{\mathtt{e}})$ with j first obtains the procedure name by evaluating the expression $\mathtt{e}$, then the arguments by evaluating the list of expressions $\overline{\mathtt{e}}$, then executes the procedure supplying these arguments, and finally assigns its return value to $\mathtt{x}$. If the procedure does not raise an error, the control is transferred to the next command; otherwise, it is transferred to the j-th command. The dynamic nature of procedures is inherited from the dynamic functions of JavaScript. Finally, the most complex command of the

language is the ϕ-node command $\mathtt{x}_1, ..., \mathtt{x}_n := \phi(\mathtt{x}_1^1, ..., \mathtt{x}_1^m; ...; \mathtt{x}_n^1, ..., \mathtt{x}_n^m)$. Intuitively, this command can be interpreted as follows: there exist m paths via which this command can be reached during the execution of the program; the value assigned to $\mathtt{x}_j|_{j=1}^n$ will be $\mathtt{x}_j^i|_{j=1}^n$ *iff* the i-th path was taken. We include ϕ-nodes in JSIL to allow for direct support for Static-Single-Assignment (SSA), which is well-known to simplify analysis and facilitates code optimisations [20]. In addition, ϕ-nodes allow us to produce more streamlined compiled code. Our JS-2-JSIL compiler generates JSIL code directly in SSA form.

A JSIL program $\mathtt{p} \in \mathtt{P}$ is a set of top-level procedures $\mathsf{proc}\ m(\overline{\mathtt{x}})\{\overline{\mathtt{c}}\}$, where m is the identifier of the procedure, $\overline{\mathtt{x}}$ is the list containing its formal parameters, and its body $\overline{\mathtt{c}}$ is a command list, that is, a numbered sequence of JSIL commands. We use $\mathtt{p}_m$ and $\mathtt{p}_m(i)$ to refer, respectively, to procedure m of the program $\mathtt{p}$ and to the i-th command of that procedure. Every JSIL program contains a special procedure main, corresponding to the entry point of the program. In contrast to JavaScript, JSIL does not have nested procedures, for two main reasons. First, we aim at a simple, minimal language. Second, we would like to reason about heap properties using separation logic, which is proven to work well using interprocedural analysis.

JSIL procedures do not explicitly return. Instead, each procedure has two special command indexes, which we denote by `ret` and `err`, 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 `ret`, it returns normally, with the return value `xret`; when it jumps to `err`, it returns an error, with the error value `xerr`, which contains information about the error. Handling returns in this way simplifies the control flow graphs of programs, where there are now only two possible exit points, rather than multiple ones.

4.2. The JSIL Semantics

Figure 4.2 introduces the memory model of JSIL. All JSIL procedures are top-level, meaning that each procedure is executed in its own dedicated variable store. A variable store, $\rho \in \mathcal{S}to$, is a mapping from JSIL variables to JSIL values, and a JSIL heap, $h \in \mathcal{H}_{\mathtt{JSIL}}$, is a mapping from locations and JSIL properties to JSIL values. JSIL is designed so that its memory model subsumes the memory model of JavaScript defined in §3.3.

Property names	$: p \in \mathcal{P}_{\mathtt{JSIL}} \subset \mathcal{S}tr$
Jsil values	$: \mathtt{v} \in \mathcal{V}_{\mathtt{JSIL}} \triangleq \mathcal{L}it$ (Figure 4.1)
Jsil stores	$: \rho \in \mathcal{S}to \ : \ \mathcal{X}_{\mathtt{JSIL}} \rightharpoonup \mathcal{V}_{\mathtt{JSIL}}$
Jsil heaps	$: h \in \mathcal{H}_{\mathtt{JSIL}} \ : \ (\mathcal{L} \times \mathcal{P}_{\mathtt{JSIL}}) \rightharpoonup \mathcal{V}_{\mathtt{JSIL}}$

Figure 4.2.: The JSIL memory model.

JSIL expressions do not have side effects and do not depend on heap values. Given a JSIL expression $\mathtt{e}$ and a store ρ, $[\![\mathtt{e}]\!]_\rho$ denotes the value of $\mathtt{e}$ with respect to ρ. The semantics of JSIL expressions is straightforward (Figure 4.3). The outcome of evaluating a JSIL expression is always a JSIL value.

The semantics of JSIL basic commands is described by a function $[\![.]\!] : \mathtt{BCmd} \times \mathcal{H}_{\mathtt{JSIL}} \times \mathcal{S}to \rightharpoonup \mathcal{H}_{\mathtt{JSIL}} \times \mathcal{S}to \times \mathcal{V}_{\mathtt{JSIL}}$ and is given in Figure 4.4. Informally, the judgement $[\![\mathtt{bc}]\!]_{h,\rho} = (h', \rho', \mathtt{v})$ means that the evaluation of `bc` in the heap h and store ρ results in the heap h' and the store ρ', with the

$$\textsc{Literal}\ \frac{}{[\![\lambda]\!]_\rho \triangleq \lambda} \qquad \textsc{Variable}\ \frac{}{[\![\mathtt{x}]\!]_\rho \triangleq \rho(\mathtt{x})} \qquad \textsc{Unary Operator}\ \frac{}{[\![\ominus\ \mathtt{e}]\!]_\rho \triangleq \ominus([\![\mathtt{e}]\!]_\rho)} \qquad \textsc{Binary Operator}\ \frac{}{[\![\mathtt{e}_1 \oplus \mathtt{e}_2]\!]_\rho \triangleq \oplus([\![\mathtt{e}_1]\!]_\rho, [\![\mathtt{e}_2]\!]_\rho)}$$

Figure 4.3.: Semantics of JSIL Expressions: $[\![\mathtt{e}]\!]_\rho = \mathtt{v}$

outcome $\mathtt{v}$. As basic commands do not affect control flow, this judgement does not need to include the index of the command. The Skip and Assignment rules are standard. The Object Creation rule ensures that when we create a new object, we bind it to a fresh location in the heap and set its prototype to null. The Property Access and the Property Assignment rules read from and write to a property of an object in the heap. The Property Deletion rule removes a property from an object. The two Member Check rules determine whether or not an object has a given own property. The GetProperties rule obtains the names of all properties of a given object. The semantic predicate $\mathcal{O}rd$ holds for lists of lexicographically ordered strings.

$$\textsc{Skip}\ [\![\mathsf{skip}]\!]_{h,\rho} \triangleq (h, \rho, \mathsf{empty}) \qquad \textsc{Assignment}\ \frac{[\![\mathtt{e}]\!]_\rho = \mathtt{v} \quad \rho' = \rho[\mathtt{x} \mapsto \mathtt{v}]}{[\![\mathtt{x} := \mathtt{e}]\!]_{h,\rho} \triangleq (h, \rho', \mathtt{v})}$$

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

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

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

$$\textsc{Property Deletion}\ \frac{h = h' \uplus ([\![\mathtt{e}_1]\!]_\rho, [\![\mathtt{e}_2]\!]_\rho) \mapsto - \quad [\![\mathtt{e}_2]\!]_\rho \neq @proto}{[\![\mathsf{delete}\,(\mathtt{e}_1, \mathtt{e}_2)]\!]_{h,\rho} \triangleq (h', \rho, \mathsf{true})}$$

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

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

$$\textsc{GetProperties}\ \frac{\begin{array}{c}[\![\mathtt{e}]\!]_\rho = l \quad h = (h' \uplus (l, p_1) \mapsto - \uplus \ldots \uplus (l, p_n) \mapsto -) \quad (l, -) \notin \mathsf{dom}(h') \\ [p_1, \ldots, p_n] = \mathtt{v} \quad \mathcal{O}rd([p_1, \ldots, p_n]) \quad \rho' = \rho[\mathtt{x} \mapsto \mathtt{v}]\end{array}}{[\![\mathtt{x} := \mathsf{getProperties}\,(\mathtt{e})]\!]_{h,\rho} \triangleq (h, \rho', \mathtt{v})}$$

Figure 4.4.: Semantics of JSIL basic commands: $[\![\mathtt{bc}]\!]_{h,\rho} = (h', \rho', \mathtt{v})$

The effects of control flow commands are captured in the semantics of JSIL programs, described using the relation $\Downarrow \subseteq (\mathcal{H}_{\mathtt{JSIL}} \times \mathcal{S}to \times \mathbb{N} \times \mathbb{N}) \times \mathcal{S}tr \times (\mathcal{H}_{\mathtt{JSIL}} \times \mathcal{S}to \times \mathcal{O}_{\mathtt{JSIL}})$ in Figure 4.5, where outcomes of executing JSIL commands are either JSIL values, or an error carrying a JSIL value: $\mathtt{o} \in \mathcal{O}_{\mathtt{JSIL}} \triangleq \mathsf{nm}\langle \mathtt{v} \rangle \mid \mathsf{er}\langle \mathtt{v} \rangle$. Informally, the judgement $\mathtt{p} \vdash \langle h, \rho, j, i \rangle \Downarrow_m \langle h', \rho', \mathtt{o} \rangle$ means that the evaluation of procedure m of program $\mathtt{p}$, starting from its i-th command, to which we have arrived from its j-th command, in the heap h and store ρ, generates the heap h', the store ρ', and results in the outcome $\mathtt{o}$. We need to know the index of the previous command for the ϕ-node command. The Basic Command rule illustrates the treatment of basic commands and the fact that they do not disrupt the control flow. In contrast, the three Goto-related rules intuitively describe the control flow jumps of

the unconditional and conditional goto statements. The rules Normal Return and Error Return capture the previously described return behaviour of a procedure. We require that the commands at the `ret` and `err` indexes are both skip. The two Procedure Call rules start by evaluating the expression that denotes the identifier of the procedure to call and the expressions that denote the arguments. If the number of supplied arguments is lower than the number of formal parameters of the procedure, the remaining parameters are set to undefined, which is a behaviour inherited from JavaScript. Then, the body of the procedure is evaluated in a new store. If the procedure returns normally, the control is transferred to the next command (Procedure Call - Normal); otherwise, it is transferred to the command with index j (Procedure Call - Error). Finally, the Phi-assignment selects the variable x_k, where i is the k-th predecessor of j. We say that i is the k-th predecessor of j in the procedure m, written $i \overset{k}{\mapsto}_m j$, if it is the k-th element of the list containing all the predecessors of j in chronological order of numbered sequence of JSIL commands. We give the definition of the successor relation below.

Definition 4.1 (Successor relation)**.** *Given a JSIL program* $\mathtt{p} \in \mathtt{P}$*, and a procedure* m *in* $\mathtt{p}$*, the* successor relation*,* $\mapsto_m \subseteq \mathbb{N} \times \mathbb{N}$*, is defined as follows:*

$$
\begin{aligned}
\mapsto_m \triangleq\ & \{(i, i{+}1) \mid i \notin \{\mathtt{ret}, \mathtt{err}\} \wedge (\mathtt{p}_m(i) = \mathtt{bc} \vee \mathtt{p}_m(i) = \mathtt{x}_1, ..., \mathtt{x}_n := \phi(\overline{\mathtt{x}_1}; ...; \overline{\mathtt{x}_n})\} \\
& \cup \{(i, j) \mid \mathtt{p}_m(i) = \mathit{goto}\ j\} \ \cup\ \{(i, j), (i, k) \mid \mathtt{p}_m(i) = \mathit{goto}\ [\mathtt{e}]\ j,\ k\} \\
& \cup\ \{(i, k), (i, i{+}1) \mid \mathtt{p}_m(i) = \mathtt{x} := \mathtt{e}(\mathtt{e}_1, ..., \mathtt{e}_{n_1})\ \mathit{with}\ k\} \\
& \cup \{(\mathtt{ret}, \mathtt{ret}), (\mathtt{err}, \mathtt{err})\}
\end{aligned}
$$

Basic Command

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

Goto

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

Cond. Goto - True

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

Cond. Goto - False

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

Normal Return

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

Error Return

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

Procedure Call - Normal

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

Procedure Call - Error

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

Phi-Assignment

$$\frac{\mathtt{p}_m(j) = \mathtt{x} := \mathtt{x}_1, ..., \mathtt{x}_n := \phi(\mathtt{x}_1^1, ..., \mathtt{x}_1^r; ...; \mathtt{x}_n^1, ..., \mathtt{x}_n^r) \quad i \overset{k}{\mapsto}_m j \quad \mathtt{p} \vdash \langle h, \rho[\mathtt{x}_t \mapsto \rho(\mathtt{x}_t^k)|_{t=1}^n], j, j+1\rangle \Downarrow_m \langle h', \rho', \mathtt{o}\rangle}{\mathtt{p} \vdash \langle h, \rho, i, j\rangle \Downarrow_m \langle h', \rho', \mathtt{o}\rangle}$$

Figure 4.5.: Semantics of JSIL control flow commands: $\mathtt{p} \vdash \langle h, \rho, i, j\rangle \Downarrow_m \langle h', \rho', \mathtt{o}\rangle$

4.3. An Example of a JSIL Procedure

To get more familiar with the JSIL language, let us look at an example of a JSIL procedure. The `getProperty` procedure, given in Figure 4.6, implements the JavaScript internal function, `GetProperty`, which traverses the prototype chain of a given object and returns the property descriptor of the given named property first found in the prototype chain, or `undefined` if absent. Together with the code of the procedure, we also present its control flow graph. Instead of indexes, we use labels for the `goto` commands, for better code readability.

```
1  proc getProperty (l, prop) {
2
3          own := "getOwnProperty" (l, prop) with perr;
4          goto [own = undefined] next pret;
5
6     next: proto := [l, "@proto"];
7          goto [proto = null] pret call;
8
9     call: chain := "getProperty" (proto, prop) with perr;
10
11    pret: xret := phi(own, own, chain);
12    ret:  skip
13
14    perr: xerr := phi(own, chain);
15    err:  skip
16 }
```

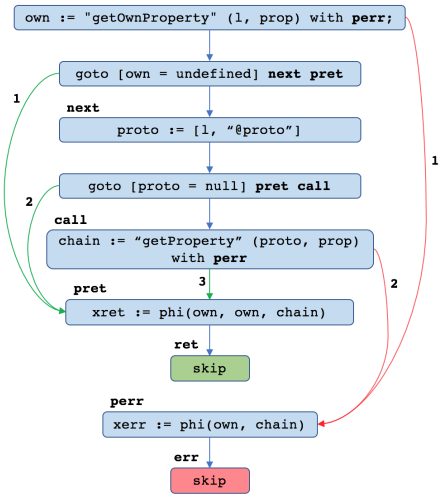


Figure 4.6.: An example of a JSIL procedure and its control flow graph

Given an object `l` and a property `prop`, we first check if the object `l` has its own property `prop` by calling another internal function `GetOwnProperty` and saving its result to a local variable `own` (line 3). If the property is not found (that is, `own` = `undefined`), we need to continue looking in the prototype chain (label `next`), otherwise, we are finished: `own` holds the property descriptor and we jump to the label `pret`, the *return* section of the code (line 4). In the `next` label, we inspect the prototype chain by reading the internal property `"@proto"` of the object `l` and store the result in the local variable `proto` (line 6). If the object `l` does not have a prototype (that is, `proto` = `null`), we are done: `own` has the value `undefined`, which is the required value in the case of an absence of the property in the prototype chain, and we go to the normal return section (line 7). Otherwise, `getProperty` is recursively called for the prototype object `proto` and the result is stored in the local variable `chain` (line 9). In the normal return section, we use `phi` to set the value of the return variable `xret` (line 11) and then we return normally (line 12). Let us look at the control flow graph to explain what `xret` := `phi`(`own`, `own`, `chain`) does. The label `pret` has three predecessors, numbered in chronological order according to the code. Given our description of the code, we want `xret` to hold value of the variable `own` if we come from the first or second predecessor and to hold value of the variable `chain` if we come from the third predecessor. In the *error* section, we again use `phi` to set the value of the error variable `xerr` accordingly (line 14) and then return an error (line 15). It is a common pattern for JSIL procedures to have return (lines 11-12) and error (lines 14-15) sections. Labels `ret` and `err` are the final labels, hence, commands at those labels are not executed. If we need to do preprocessing, for example, to set the value of the return `xret` or the error variable `xerr` before exiting the procedure's body, we create additional labels `pret` and `perr` with `phi`-node commands to accommodate the desired behaviour.

5. The JS-2-JSIL Compiler

The JS-2-JSIL compiler from JavaScript to JSIL, outlined in Figure 5.1, targets ES5 Strict. It closely follows the English standard, which means that the structure of a compiled JSIL program directly reflects the description of the behaviour of the original JavaScript program. We illustrate the compilation process from JavaScript to JSIL by translating an assignment from our running example (§5.1).

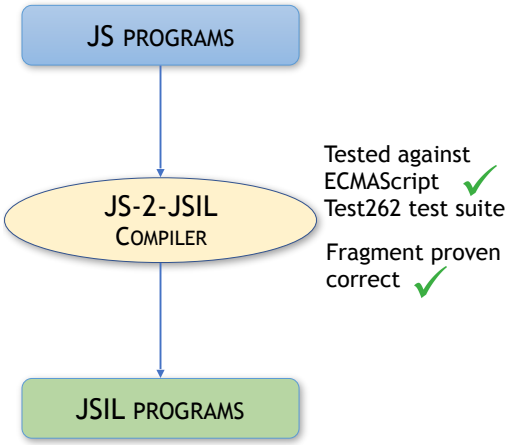


Figure 5.1.: The JS-2-JSIL Compiler

We cover a very large and fully representative fragment of ES5 Strict. In doing so, the memory model is not simplified in any way. We implement the entire core language of ES5 Strict, together with all of the built-in libraries that are strongly intertwined with the core language. We describe the coverage of the JS-2-JSIL compiler in detail in §5.2.

We systematically test the JS-2-JSIL compiler against the new ECMAScript 6 Test262 test suite, which organises tests by feature. This enables us to provide a more fine-grained analysis than was previously possible. Given our substantial but incomplete coverage of ES5 Strict, our aim was to provide an interactive testing framework to enable us to determine precisely which tests are relevant and which are not, which pass and why they pass, and which fail and why they fail. We identify 10469 tests relevant for ES5 Strict and 8797 tests relevant for the coverage of the JS-2-JSIL compiler, of which we pass 100%. We give the details of our testing in §5.3.

We designed the JS-2-JSIL compiler so that there is a simple correspondence between JavaScript and JSIL heaps, and a step-by-step connection to the standard. This allows us to define a straightforward correctness condition for the JS-2-JSIL compiler. We formalise the compiler (§5.4) for the fragment of ES5 Strict defined in §3.4 and give a correctness proof (§5.5) using our formal ES5 Strict operational semantics. The full result would require a substantial mechanised proof development.

5.1. JS-2-JSIL: Compilation by Example

We illustrate the compilation process from JavaScript to JSIL using an assignment from our running example, namely

```
30   /* @id enqueue */
31   module.prototype.enqueue = function(pri, val) {
32      var n = new Node(pri, val);
33      this._head = n.insertToQueue(this._head);
34   };
```

from the function `Module`. With this example, we show the compilation of functions, variables, assignments, property accessors, function expressions, as well as a number of JavaScript internal functions.

First, we explain how functions in JavaScript are translated to JSIL procedures. Next, we describe how JavaScript variable binding is dealt with in JSIL. Then, we go step-by-step through the compilation of the above assignment: the assignment itself, property accessors on the left-hand side and the function expression on the right-hand side of the assignment.

Functions in JSIL. Given an ES5 Strict statement s, as well as translating the statement itself, the JS-2-JSIL compiler also generates a JSIL procedure for each nested function literal in s. We can think of the translation of the above assignment being done in two parts: a translation of the assignment itself (Figure 5.2, left) and a translation of the function body (Figure 5.2, right).

```
1   /* @id = enqueue */
2  module.prototype.enqueue = function(pri, val) {}
```

```
1   /* @id = enqueue */
2   function(pri, val) { /* body of enqueue */ }
```

Figure 5.2.: Compilation by example: the assignment and the body of the nested function.

Having unique function identifiers allows us to separate function expression compilation from function body compilation. A new JSIL procedure is created corresponding to the nested function:

```
1   proc enqueue (x__s, x__this, pri, val) {
2      /* compiled function body */
3   }
```

where its first formal parameter `x__s` contains the scope in which the function was defined; its second formal parameter corresponds to the value of the keyword `this` during the execution of the compiled function body; and its remaining formal parameters match the formal parameters of the original function.

When the body of `enqueue` is executed, the value of its formal parameter `x__s` is a two-element list, [`global`, `ER−Module`], corresponding to the scope in which `enqueue` was defined. At the beginning of the procedure, a new environment record `ER−enqueue` will be created and appended at the end of the given `x__s`. The resulting scope chain [`global`, `ER−Module`, `ER−enqueue`] is stored in the dedicated local variable `x__scope` to denote the current scope chain to be used for variable dereferencing. There, `ER−enqueue` is the environment record created upon invocation of the function call (it is different for each call to `enqueue`), whereas the remainder of the list [`global`, `ER−Module`] is the value of the `@scope` property of the function object `enqueue` (it does not change across different calls to `enqueue`).

JavaScript Variable Binding in JSIL. We illustrate variable binding using the body of the `enqueue` function. We give the translation of the entire body later, in §5.4, after we have formalised the JS-2-JSIL compiler. Here, we explain the compilation of the variables used in the body. Recall the pretty-big-step rules for variable binding:

VARIABLE

$$\frac{\wp, L, v_t \vdash \langle h, \mathcal{I}_\sigma(x)\rangle \Downarrow_m \langle h_1, o_1\rangle \quad \wp, L, v_t \vdash \langle h_1, id(x, o_1)\rangle \Downarrow_m \langle h_f, o_f\rangle}{\wp, L, v_t \vdash \langle h, x\rangle \Downarrow_m \langle h_f, o_f\rangle}$$

VARIABLE-REF

$$\vdash \langle h, id(x, v)\rangle \Downarrow_m \langle h, v._{\mathsf{v}} x\rangle$$

When we need 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`. However, ES5 Strict is syntactically scoped and we can statically determine if a given variable is defined in a given scope chain and if so, in which ER it is defined. This means that we do not need to translate scope inspection as a list traversal $\mathcal{I}_\sigma(x)$. Instead, we first determine the function in which the variable is defined and obtain the index of the corresponding ER in the current scope chain, which we do using a special *scope clarification function* (explained in more detail in §5.4.3). The result of the variable binding is always a reference $v._{\mathsf{v}} x$, where v corresponds to the ER, or is undefined if the variable is not found anywhere in the scope chain.

The `enqueue` function uses four variables: `n`, `pri`, `val`, and `Node` in its body. The local variable `n`, and formal parameters `pri` and `val` are stored in the environment record created upon the invocation of the function which is at the end of the current scope chain. Hence, the compiled code for the variable `n` first reads the last element from `x__scope` (which is 2 for the scope of the `enqueue`), and then computes the corresponding reference:

```
1 x_1 := nth (x__scope, 2);
2 x_2 := ["v", x_1, "n"];
```

We also know syntactically that `Node` is defined inside the function `Module` and that its environment record is the second one in the scope chain. Therefore, the compiled code for the variable `Node` first reads the second element from `x__scope` to obtain the correct ER, `ER~Module`, and then computes the corresponding reference:

```
1 x_1 := nth (x__scope, 1);
2 x_2 := ["v", x_1, "Node"];
```

There is a caveat with the global object. Recall that if we do not find a binding for `x` anywhere else in the current scope chain, we have to traverse the prototype chain of the global object. However, we cannot statically know whether or not a variable is defined in the global object, since we can add properties to the global object not only using variable declarations `var x = v`, but also using property accessors `this.x = v`, where `this` corresponds to the global object in the global code. We discuss this point more in the formalisation of the JS-2-JSIL compiler in §5.4.

Compiling the Assignment. We now go step-by-step through the compilation of the assignment `module.prototype.enqueue = /*@id = enqueue */function(pri, val){/*function body */}`, which is given in Figure 5.3. This statement has non-trivial behaviour and triggers a number of JavaScript internal functions.

1. In the first step, we evaluate the left-hand side expression of the assignment (the property accessor `module.prototype.enqueue`) and obtain the corresponding reference. The evaluation of property accessors is described in §11.2.1 of the ES5 standard, and it is line-by-line reflected in lines 1-14 of the JSIL code. We show the compilation of property accessors in more detail in Figure 5.4, together with the explanation below. Note that the reference resulting from the evaluation of a property accessor is always an object reference. In this case, this is the reference

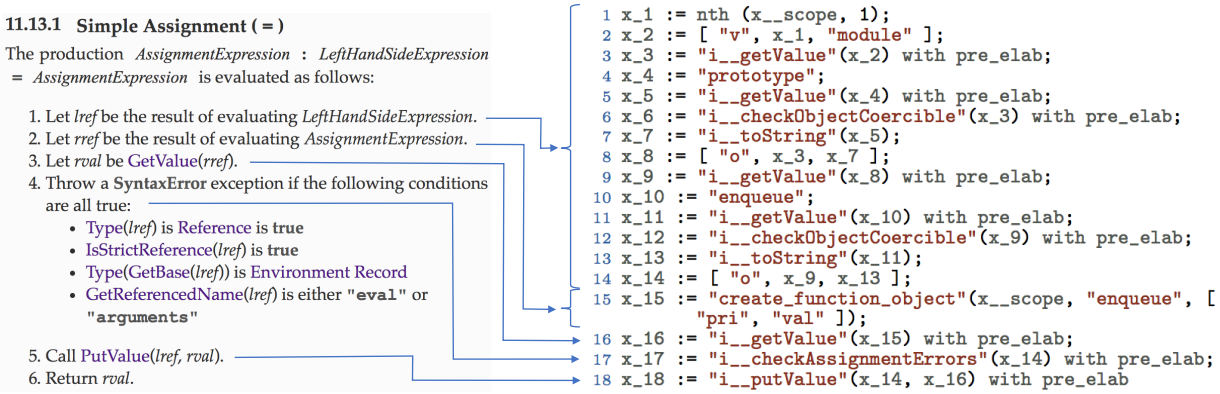


Figure 5.3.: Compiling the JavaScript assignment expression to JSIL

["o", x_9, x_13] which, given the running example, is a reference of the property "enqueue" of the object PriorityQueue.prototype created in the function Module.

2. Next, we do the same for the right-hand side of the assignment, the function literal with identifier enqueue. The evaluation of function expression is described in §13 of the ES5 standard, and it is reflected in line 15 of the JSIL code. We explain function object creation in more detail in Figure 5.4. The result from the evaluation of a function expression is a newly created function object. Given our running example, it corresponds to the function object enqueue.

3. The evaluated right-hand side expression might be a reference and it is dereferenced to get the actual value to be assigned. The dereferencing is done by the GetValue internal function (§8.7.1 of the ES5 standard). Here, "create_function_object" returns an object location, not a reference, in which case, the same object location is the result of GetValue. As we provide reference implementations of all internal functions, any call to an internal function gets translated to JSIL as a procedure call to our corresponding reference implementation, in this case, i__getValue (line 16). We elaborate further on GetValue in §7.2.

4. In ES5 Strict, the identifiers eval and arguments may not appear as the left-hand side of an assignment (e.g. eval = 42), and this step enforces this restriction. We do not inline the conditions every time, but instead call a JSIL procedure i__checkAssignmentErrors (line 17), 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 18). We delay the details of PutValue until §7.2.

6. In JavaScript, every expression and statement returns a value. In JSIL, the compiler, when given an expression or a statement to compile, returns not only the list of corresponding JSIL commands, but also the variable that stores the return value of that statement. In this particular case, the compiler would return, in addition to the presented code, the variable x_16.

Compiling the Property Accessor. Next, we go step-by-step through the compilation of the property accessor (module.prototype)["enqueue"]. Figure 5.4 shows the line-by-line correspondence to the ES5 standard. Recall that the ES5 standard states that e.p is identical in its behaviour to e["p"].

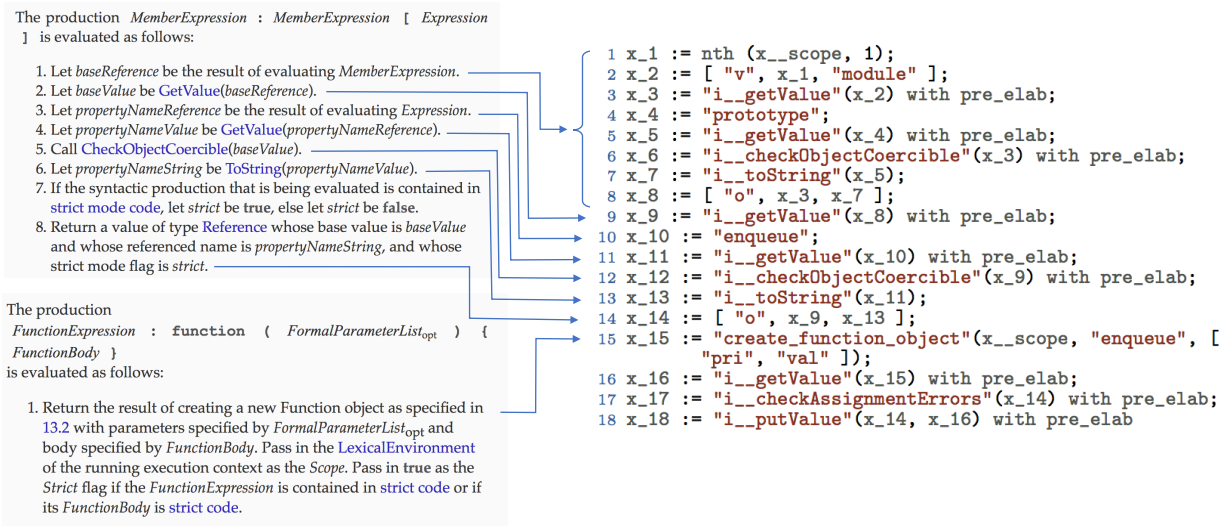


Figure 5.4.: Compiling property accessors and function literals to JSIL

1. In the first step, we evaluate the member expression `e` of the property accessor (which is another property accessor `module.prototype`) and obtain the corresponding reference.

   Compiling the inner property accessor `module["prototype"]`**.** Note that steps in the ES5 Standard are line-by-line reflected in lines 1-8 of the JSIL code for the inner property accessor. We do not go into more detail in the translation of inner property accessor, except that we explain the translation of variable `module`. Here, we need to perform variable resolution and the resulting reference will be a variable reference. For the JSIL translation, given how we emulate JavaScript scope chains, we only need to understand within which ER `module` is defined. As `module` is a local variable of the `Module` function (Figure 3.9), it will be in the last ER in the current scope chain, `ER—Module` (line 1). The appropriate reference, `[ "v", x_1, "module"]`, is then constructed in line 2. This code is automatically generated using the scope clarification function.

   Coming back to the outer property accessor, a reference resulting from the evaluation of the inner property accessor is an object reference `[ "o", x_3, x_7 ]`. Given the running example, it corresponds to the property `"prototype"` of the object `PriorityQueue` created in the function `Module`.

2. Next, the obtained reference is dereferenced to get the actual value. The dereferencing is done by the `GetValue` internal function. Given the running example, the obtained value is the object `PriorityQueue.prototype`.

3. We evaluate *Expression* of the property accessor, `"enqueue"`. It is a string literal, hence, it evaluates to itself (line 10).

4. The obtained *propertyNameReference* might be a reference and it is dereferenced to get the actual value. Again, the dereferencing is done by the `GetValue` internal function (line 11). Here, `"enqueue"` is a string literal, not a reference, in which case, the same string literal is the result of `GetValue`.

5. For the property accessor `o[x]` to be valid, we need to make sure that `o` is actually an object. The internal function `CheckObjectCoercible` (§9.10 of the ES5 standard) throws an error if its argument is a value that cannot be converted to an object. It is translated to JSIL directly, as a procedure call to our reference implementation (line 12).

6. Another requirement for the property accessor `o[x]` is for `x` to be a string. The semantics does not throw an exception otherwise, but performs implicit type conversion instead. Type conversion to a string is performed by calling the internal `ToString` function (§9.8 of the ES5 standard), which is translated to JSIL directly, as a procedure call to our reference implementation `i__toString` (line 13). See §3.1.2 for a detailed description of `ToString` and more examples of it being used in the context of property accessors, and §7.2 for more information on its specification.

7. In ES5 Strict all references are strict, hence, this step is not reflected in the JSIL code.

8. Finally, the result of evaluating property accessor, is the reference [`"o"`, `x_9`, `x_13`] which we already have seen in the assignment translation above.

Compiling the Function Expression. When a function expression is evaluated in JavaScript, a new function object is created with corresponding formal parameters, function identifier and the current scope chain. We call the JSIL procedure `create_function_object` (line 15), which takes as parameters the current scope chain `x__scope`, the function identifier `"enqueue"` and a list of formal parameters [`"pri"`, `"val"`], and creates a new function object. The created function object corresponds to the `enqueue` function object in Figure 3.10.

This example illustrates how close the JS-2-JSIL compiler is to the ES5 standard. Most of the lines of the compiled JSIL code have a direct counterpart in the standard. Variable binding is the only part where an observable difference is expected. This example also reveals part of the complexity behind the JavaScript language. We have seen expression evaluations, syntactic checks, implicit type conversions, and calls to different internal functions. What is not visible yet, and will be shown in §7.2, is the hierarchy of internal functions behind `GetValue` and `PutValue`. This level of complexity is the precise reason why JavaScript analysis is difficult.

5.2. JS-2-JSIL: Compiler Coverage

We cover a very large and fully representative fragment of ES5 Strict, as witnessed by our test suite coverage, detailed in §5.3. In doing so, we do not simplify the memory model of JavaScript in any way, and fully support descriptors and attributes, including getters and setters. The coverage of the JS-2-JSIL compiler is illustrated in Figure 5.5.

As we have already mentioned, we do not target the correctness of the JavaScript parser, and view that as a separate project. We implement the entire core language, except the indirect eval, which by default exits strict mode, falling out of the scope of our project. As for the built-in libraries, we have implemented in JSIL all of the parts that are strongly intertwined with the core language. These include: the core of the `Global` library, associated with the global object; the remaining functionalities target URI processing and number parsing, and are orthogonal; the entire `Object` library, which contains functions for advanced object management, including object creation with a specified prototype

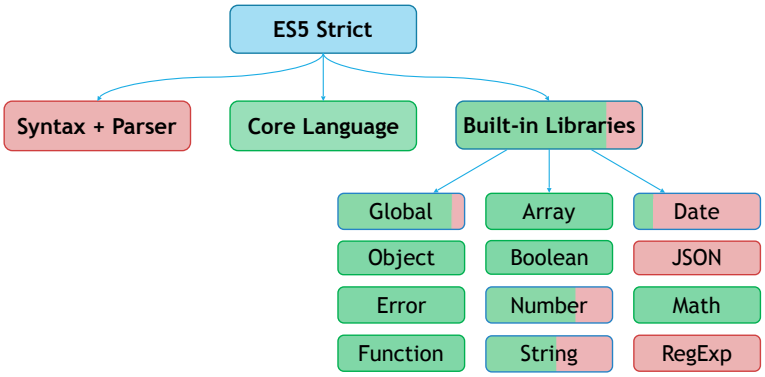


Figure 5.5.: JS-2-JSIL: compiler coverage

(`Object.create`), property definition at the level of descriptors (`Object.defineProperty`), and extensibility management (`Object.preventExtensions`, `Object.seal`, `Object.freeze`); the entire `Function` library, which provides methods for calling (`Function.prototype.call`), applying (`Function.prototype.apply`), and binding (`Function.prototype.bind`) function objects, as well as the `Function` constructor, which creates a function object from a given string that represents a JavaScript function body[1]; and the entire `Error` library, which provides constructors for various JavaScript errors that can be thrown. Without the implementation of these four libraries, running standard JavaScript code would not be feasible. We also implement the entire `Array` library to show the feasibility of tackling comprehensive JavaScript libraries, as well as the entire `Math` library. When it comes to the `Boolean`, `Number`, and `String` libraries, we implement the basic functions used for testing features of the core language, including the mandatory value properties (such as `length` for `String`), internal functions that differ from the default ones presented in chapters 8-14 of the standard (`GetOwnProperty` of `String`), as well as the `toString` and `valueOf` functions, which allow conversion to strings and primitive values. The `Date` library is also used to test some of the core language features and we support some of its basic functionality, including a part of its constructor, and the `valueOf` function. However, there is a part of the basic functionality, such as `toString` function, which we do not yet support, since it requires date parsing, which is implementation-dependent. We do not implement the `RegExp` and `JSON` libraries, as they are fully orthogonal to the core language.

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

We implement the initial heap in full, including stubs for unimplemented functions. This setup requires a fair amount of precision and takes approximately 750 lines of JSIL code. We implement all internal functions (43 of them, approximately 1000 lines of JSIL code), line-by-line following the English standard. We gave an implementation of one of such internal functions, namely `getProperty`, in §4.3, Figure 4.6. Finally, our implementation of the built-in library functions takes approximately

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

3500 lines of JSIL code, again following the standard. We refer to the JSIL implementations of the initial heap, internal functions, and built-in libraries as the *compiler runtime*.

5.3. JS-2-JSIL Validation: Testing

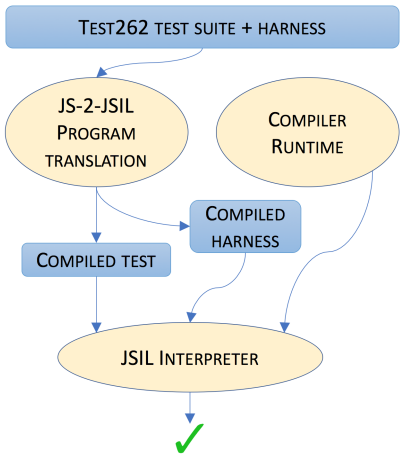
We validate the correctness of the JS-2-JSIL compiler by extensive testing against the official ECMAScript Test262 test suite, achieving a 100% success rate on the 8797 pertinent tests. In addition, we develop a modular and reusable continuous-integration testing infrastructure, greatly simplifying the overall testing process.

ECMAScript Test262 Test Suite. ECMAScript Test262 is the official test suite for JavaScript implementations. Currently, there are two available versions of the suite: an unmaintained version for ES5; and an actively maintained version for the ES6 standard. ES5 Test262 has poor support for ECMAScript implementations that enforce strict mode. Tests are inconsistently flagged with respect to the mode in which they can be run, and a significant number of test cases that should be common to both strict and non-strict modes of the language contain errors preventing them from being executed correctly in strict mode. This renders any kind of systematic effort to target ES5 Strict tests infeasible. All such errors have been fixed in the latest version of the tests for ES6. In addition, a considerable effort has been made by the test suite maintainers to ensure that *all* tests are correctly flagged for strict mode.

On the other hand, there do exist certain disadvantages in using a more recent version of the test suite than the specification was designed for; some test cases will no longer be applicable, and their results will need to be excluded. The ES6 version of the specification was published 6 years after ES5, and during this time, the specification was comprehensively redrafted and gained numerous new features. Luckily, the language committee took great care in minimising the number of backwards incompatible changes between versions of the specification. As a result, a relatively small proportion of the test cases needed to be altered by the test suite maintainers between the versions. It is possible to identify these test cases and exclude them from the results. New features are easy to identify and exclude due to the structure of the test suite.

On the whole, the strong negatives of a poorly maintained ES5 version of the test suite overshadow the minor difficulties of having to track the incompatible changes and new features between versions of the specification. We have thus opted to test the JS-2-JSIL compiler using the latest version of ES6 Test262. While this does mean that we need to do more filtering to arrive at the applicable tests (for example, we have to exclude all of the tests targeting ES6 features), this is a small price to pay for the overall increase in precision and correctness.

Addressing Incompleteness. The Test262 suite assumes a complete implementation of JavaScript on which to run the tests. In many cases, a test for one language feature will make use of unrelated language features, most often built-in libraries, to test its result. While it is clear that it is legitimate to ignore test cases that directly test unimplemented features, several options are available when these features (such as the library functions for `Array` sorting or `String` splitting) are used for the testing of relevant constructs: one could either expand the implementation to cover the missing dependency; filter out the test; or rewrite the test to avoid the unimplemented feature. We have opted for a combination of the first two solutions, and have chosen not to rewrite any tests in the test suite.



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

Figure 5.6.: JS-2-JSIL Validation by testing (left); Detailed testing results (right)

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

Running Tests. We perform the actual testing as shown in Figure 5.6 (left). First, we compile to JSIL the official harness of ES6 Test262. Then, for each test, we: compile the code of the test to JSIL; execute, using our JSIL interpreter, the JSIL code obtained by concatenating the compiled harness, the compiled test, and the compiler runtime; and if the execution terminated normally, we declare that the test has passed. There is never a need to test the return value of the program, because the Test262 tests perform the actual testing within the program and throw an error if any of the tested conditions are not met.

Test Filtering and Testing Results. The breakdown of the testing results is presented in Figure 5.6 (right). The version of the ES6 Test262 test suite used in this study[2] 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 web browsers, 699 tests), and the internationalisation API (189 tests), all of which fall out of the scope of this project. Next, we exclude tests for ES5 non-strict features (890 tests). We note that these include all tests for indirect eval and a number of tests for the `Function` constructor, which both allow the developer to use non-strict code even when explicitly executing the code in strict mode. We obtain 10469 tests that target ES5 Strict.

Next, to filter down to the tests that should reflect the coverage of the JS-2-JSIL compiler, we

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

remove 1297 tests for built-in library functions that have not been implemented, such as those in the `RegExp` and `JSON` libraries. This leaves us with the total of 9172 tests that target the JS-2-JSIL compiler.

Not all of these tests, however, are applicable. ES6, although being mostly backward compatible, has introduced minor changes to the semantics of a few features with respect to ES5, and there are 345 tests that target such features. These changes include: the `length` property of `Function` objects being configurable in ES6, but not configurable in ES5; the prototype of all native errors (for example, `TypeError`, `SyntaxError`, and `ReferenceError`) being `Error` in ES6 and `Function.prototype` in ES5; a slight change in the semantics of the iteration statements `while` and `do−while` in ES6, now returning `undefined` instead of `empty` when the body of the while returns `empty`; `Object.keys(o)` throwing an exception in ES5 if `o` is not an object, while in ES6 an exception is thrown only of `o` is `null` or `undefined`.

Also, 30 tests were testing features covered by the compiler, but in doing so were using non-implemented features. All of these tests, except for one, can be rewritten to still test the same feature, but not use the non-implemented feature. For example, to test the `typeof` operator from the core language, the `RegExp` library is used as a part of the test. Since we do not support `RegExp`, the entire test fails, even though `RegExp` usage forms only a small part of the test. By removing the `RegExp` usage from the test, we would pass it. Another example includes testing the built-in `Object` constructor using the `Date` library. When called with one argument, which is an object, the `Object` constructor does not create a new object but simply returns the provided argument. The test could be easily rewritten using other libraries that we support, such as `String` or `Number`. However, there is one test for the core language construct, namely, the addition operator, that uses the `Date` library and that cannot be rewritten. An addition operator `e1 + e2` calls the internal function `ToPrimitive` on its operands. When `ToPrimitive` is called on an object, another internal function, `DefaultValue`, is executed. `DefaultValue` takes an optional parameter `hint` on which its behaviour depends. If the `hint` is not provided, `DefaultValue` behaves as if the `hint` was provided with the value `"Number"`, except for `Date` objects, where in the absence of the `hint`, the `DefaultValue` behaves as if the `hint` was provided with the value `"String"`. Since such specific behaviour of the addition operator can only be observed using `Date` objects, we are not able to rewrite the test. As we have chosen not to rewrite any tests in the test suite, we excluded these 30 tests from our final applicable tests.

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

5.4. JS-2-JSIL: Compiler Formalisation

We formalise our translation from ES5 Strict to JSIL as a compilation function $\overline{\mathcal{C}} : \mathcal{S}_{\mathtt{JS}} \rightarrow \mathtt{P}$, that receives an ES5 Strict statement $s \in \mathcal{S}_{\mathtt{JS}}$ and produces a JSIL program $\mathtt{p} \in \mathtt{P}$. More concretely, given an ES5 Strict statement s, the translation generates a JSIL procedure for each nested function literal in s and a special procedure `main` for the top-level code in s. In order to simplify the formalisation, we assume that every function literal and lambda abstraction is annotated with a unique identifier m. We define the main compiler $\overline{\mathcal{C}}$ using two auxiliary compilers: $\hat{\mathcal{C}}$, that, given a JavaScript function literal, generates its corresponding JSIL procedure; and $\mathcal{C}$, that is used for compiling arbitrary expressions

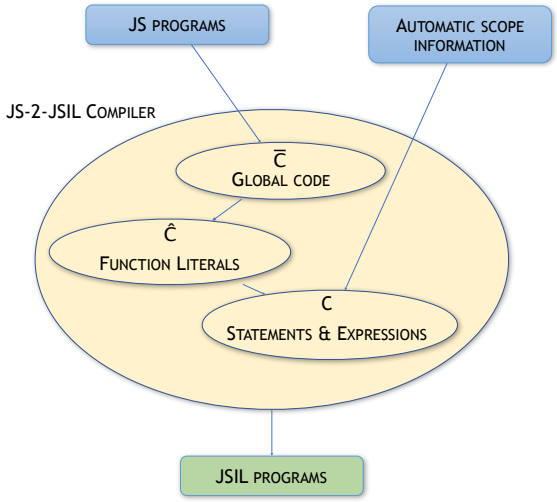


Figure 5.7.: The structure of the JS-2-JSIL Compiler

and statements (Figure 5.7). Both $\hat{\mathcal{C}}$ and $\mathcal{C}$ assume that the function literals that occur in the code to compile have already been separately compiled.

We define the three compilers using OCaml-like notation, which we introduce first, together with other auxiliary functions that we use in the definitions of the compilers. Next, we introduce the compiler $\overline{\mathcal{C}}$ for compiling the global code (§5.4.1). Using the assignment example from §5.1, we then introduce the compiler $\hat{\mathcal{C}}$ for compiling function literals (§5.4.2), and the compiler $\mathcal{C}$ for expressions and statements (§5.4.3).

Notation. In the definition of the compilers, we use OCaml-like notation: **lambda** $\overline{x}.t$ for functions, **let** x **in** t for local variables, **match** x **with** $\mid \ t_1 \Rightarrow t_2 \mid t_3 \Rightarrow t_4 ...$ for pattern matching, and the function $\mathbf{map}(f, X)$ for the application of the function f to each element of the set X.

We also use auxiliary functions to describe JavaScript and JSIL concepts. We denote by $\mathsf{defs}(s)$ and $\mathsf{funlits}(s)$, respectively, the set of variables declared in the JavaScript statement s and the set of function literals in the JavaScript statement s. We use the auxiliary function $\mathsf{fresh}()$ to generate new JSIL variable names and new JSIL labels. We denote concatenation of JSIL command lists and JSIL variable lists with the :: operator.

5.4.1. Compiling the Global Code

The main compiler

$$\overline{\mathcal{C}} \triangleq \mathbf{lambda}\, s.\mathbf{map}(\hat{\mathcal{C}}, \mathsf{funlits}(s) \cup \{\ \mathsf{function}\ ()\{s\}^{\mathsf{main}}\ \})$$

given the global code s, collects all nested function literals, $\mathsf{funlits}(s)$, and translates them using the auxiliary compiler $\hat{\mathcal{C}}$. It also uses the same $\hat{\mathcal{C}}$ for compiling the global code itself, which we can think of as a function literal without any parameters, s as its body, and main as its identifier. Given our running example (Figure 3.9), $\mathsf{funlits}(s)$ contains six function literals. Additionally, we have one more function literal for the global code. All of these seven function literals are given in Figure 5.8. Note that function literals do not contain the bodies of their nested functions. However, they do contain their

```
1  function () {
2     /* @id Node */
3     var Node = function (pri, val) {}
4
5     /* @id insertToQueue */
6     Node.prototype.insertToQueue = function (q) {}
7
8     /* @id PriorityQueue */
9     var module = function () {};
10
11    /* @id enqueue */
12    module.prototype.enqueue = function(pri, val) {};
13
14    /* @id dequeue */
15    module.prototype.dequeue = function () {};
16
17    return module;
18 } /* Module */

19 function (pri, val) {
20    this.pri = pri; this.val = val; this.next = null;
21 } /* Node */

22 function (q) {
23    if (q === null) {
24       return this
25    }
26
27    if (this.pri >= q.pri) {
28       this.next = q;
29       return this
30    }
31
32    var tmp = this.insertToQueue (q.next);
33    q.next = tmp;
34    return q
35 } /* insertToQueue */
```

```
36 function () {
37    this._head = null;
38 }; /* PriorityQueue */

39 function(pri, val) {
40    var n = new Node(pri, val);
41    this._head = n.insertToQueue(this._head);
42 } /* enqueue */

43 function () {
44    if (this._head === null) {
45       throw new Error("Queue is empty");
46    }
47
48    var first = this._head;
49    this._head = this._head.next;
50    return {pri: first.pri, val: first.val};
51 }; /* dequeue */


52 function() {
53    /* @id Module */
54    var PriorityQueue = (function () {})();
55
56    var q = new PriorityQueue();
57    q.enqueue(1, "last");
58    q.enqueue(3, "bar");
59    q.enqueue(2, "foo");
60    var r = q.dequeue();
61 } /* main */
```

Figure 5.8.: The function literals from the running example to be compiled

identifiers. We have already discussed this separation in §5.1 using an assignment from the function `Module`, shown on lines 11-12 together with the function `enqueue` on lines 39-42 in Figure 5.8. We have informally discussed the translation of the assignment itself and started discussing the translation of the function `enqueue`. Next, we formally define the compilation of function literals using the `enqueue` function which is compiled to the procedure given in Figure 5.9. Full understanding of the code of the procedure is not required at this moment. We will be explaining most of its parts step-by-step in the upcoming sections.

```
proc enqueue (x__s, x__this, pri, val) {
         x__er := new();
         x__scope = x__s @ [x_er];
         [x__er, "n"] := undefined;
         [x__er, "pri"] := pri;
         [x__er, "val"] := val;
         x_1 := nth (x__scope, 2); /* n */
         x_2 := [ "v", x_1, "n" ];
         x_3 := nth (x__scope, 1);
         x_4 := [ "v", x_3, "Node" ];
         x_4_v := "i__getValue"(x_4) with perr; /* Node */
         x_5 := nth (x__scope, 2);
         x_6 := [ "v", x_5, "pri" ];
         x_6_v := "i__getValue"(x_6) with perr; /* pri */
         x_7 := nth (x__scope, 2);
         x_8 := [ "v", x_7, "val" ];
         x_8_v := "i__getValue"(x_8) with perr; /* val */
         goto [(not (typeOf(x_4_v) = object_type))] terr next_1; /* Error if Node is not an object */
  terr: x_err := "TypeError"();
         goto perr;
next_1: x_9 := hasProperty(x_4_v,"@code"); /* Error if Node cannot be called as constructor */
         goto [x_9] next_2 terr;
next_2: x_10 := new();
         x_11 := [ "o", x_4_v, "prototype" ];
         x_11_v := "i__getValue"(x_11) with perr; /* Retrieving Node.prototype object */
         goto [(typeOf(x_11_v) = object_type)] then_1 else_1;
else_1: x_12 := lobj_proto; /* if Node.prototype is not an object, Object.prototype will be used for @proto*/
then_1: x_13 := PHI(x_11_v, x_12);
         x_14 := "create_default_object"(x_10, x_13); /* newly created object with appropriate @proto */
         x_fid_1 := [x_4_v, "@code"]; /* reading the function identifier from function object */
         x_fscope_1 := [x_4_v, "@scope"]; /* reading the scope from function object */
         x_15 := x_fid_1(x_fscope_1, x_10, x_6_v, x_8_v) with perr; /* calling the function */
         goto [(typeOf(x_15) = object_type)] then_2 else_2;
else_2: skip;
then_2: x_16 := PHI(x_15, x_10);
         x_16_v := "i__getValue"(x_16) with perr
         x_17 := "i__checkAssignmentErrors"(x_2) with perr;
         x_18 := "i__putValue"(x_2, x_16_v) with perr; /* n = new Node(pri, val) */
         x_r_1 := empty;
         x_19 := x__this; /* this */
         x_19_v := "i__getValue"(x_19) with perr;
         x_20 := "i__checkObjectCoercible"(x_19_v) with perr;
         x_21 := [ "o", x_19_v, "_head" ]; /* this._head */
         x_22 := nth (x__scope, 2);
         x_23 := [ "v", x_22, "n" ];
         x_23_v := "i__getValue"(x_23) with perr;
         x_24 := "i__checkObjectCoercible"(x_23_v) with perr;
         x_25 := [ "o", x_23_v, "insertToQueue" ];
         x_25_v := "i__getValue"(x_25) with perr; /*n.insertToQueue*/
         x_26 := x__this;
         x_26_v := "i__getValue"(x_26) with perr;
         x_27 := "i__checkObjectCoercible"(x_26_v) with perr;
         x_28 := [ "o", x_26_v, "_head" ];
         x_28_v := "i__getValue"(x_28) with perr /* this._head */
         goto [(not (typeOf(x_25_v) = object_type))] terr next_3; /* Error if insertToQueue is not an object */
next_3: x_29 := "i__isCallable"(x_25_v);
         goto [x_29] next_4 terr; /* Error if insertToQueue cannot be called */
next_4: goto [((typeOf(x_25) = list_type) and (nth(x_25, 0) = "o"))] then_3 else_3;
then_3: x_this_1 := nth(x_25, 1);
         goto phi_1;
else_3: x_this_2 := undefined;
 phi_1: x_this_3 := PHI(x_this_1, x_this_2); /* Resolving this */
         x_fid_2 := [x_25_v, "@code"];
         x_fscope_2 := [x_25_v, "@scope"];
         x_30 := x_fid_2(x_fscope_2, x_this_3, x_28_v) with perr;
         goto [(x_30 = empty)] then_4 else_4;
then_4: x_31 := undefined;
else_4: x_32 := PHI(x_30, x_31);
         x_32_v := "i__getValue"(x_32) with perr;
         x_33 := "i__checkAssignmentErrors"(x_21) with perr;
         x_34 := "i__putValue"(x_21, x_32_v) with perr; /* this._head = n.insertToQueue(this._head); */
         goto [(x_32_v = empty)] then_5 else_5;
else_5: skip;
then_5: x_35 := PHI(x_r_1, x_32_v);
  pret: xret := undefined;
   ret: skip;
  perr: xerr := PHI(x_4_v, x_6_v, x_8_v, x_err, x_11_v, x_15, x_16_v, x_17, x_18, x_19_v, x_20, x_23_v, x_24,
        x_25_v, x_26_v, x_27, x_28_v, x_30, x_32_v, x_33, x_34);
   err: skip;
}
```

Figure 5.9.: The compiled `enqueue` procedure

$\hat{\mathcal{C}} \triangleq$

lambda function $(x_i|_{i=1}^n)\{s\}^m$.
let $(y_i|_{i=1}^k) = \mathsf{defs}(s)$;
$\mathtt{x} = \mathsf{fresh}()$;
$\overline{\mathtt{c}}, \overline{\mathtt{e}}, \overline{\mathtt{r}}, \_ = \mathcal{C}_m(s, \mathtt{x}, \_, \_)$;
$\mathtt{x}_i = \mathsf{fresh}()|_{i=1}^{n+1}$;
$\mathtt{x}_{er} = \mathsf{fresh}()$;
in proc $m(\mathtt{x}_{sc}, \mathtt{x}_{this}, \mathtt{x}_i|_{i=1}^n)\{$
$\mathtt{x}_{er} :=$ new ()
$\mathtt{xscope} := \mathtt{x}_{sc} @ [\mathtt{x}_{er}]$
$[\mathtt{x}_{er}, y_i] :=$ undefined $|_{i=1}^k$
$[\mathtt{x}_{er}, x_i] := \mathtt{x}_i|_{i=1}^n$
$\overline{\mathtt{c}}$
$\mathtt{x}_{n+1} :=$ undefined
$\mathtt{pret} : \mathtt{xret} := \phi(\overline{\mathtt{r}} :: \mathtt{x}_{n+1})$
$\mathtt{ret} :$ skip
$\mathtt{perr} : \mathtt{xerr} := \phi(\overline{\mathtt{e}})$
$\mathtt{err} :$ skip
}

```
1  proc enqueue (x__s, x__this, pri, val) {
2           x__er := new();
3           x__scope = x__s @ [x_er];
4           [x__er, "n"] := undefined;
5           [x__er, "pri"] := pri;
6           [x__er, "val"] := val;

                c
75     pret: xret := undefined;
76      ret: skip;
77     perr: xerr := PHI(x_4_v, x_6_v, x_8_v, x_err,
              x_11_v, x_15, x_16_v, x_17, x_18, x_19_v,
              x_20, x_23_v, x_24, x_25_v, x_26_v, x_27,
              x_28_v, x_30, x_32_v, x_33, x_34);
78      err: skip;
79 }
```

Figure 5.10.: The auxiliary compiler $\hat{\mathcal{C}}$, compiling JavaScript function literals to JSIL procedures

5.4.2. Compiling Function Literals

The compiler $\hat{\mathcal{C}}$ (Figure 5.10, left) receives as input an ES5 Strict function literal and generates its corresponding JSIL procedure. Each function literal function $(\overline{x})\{s\}^m$ is compiled by $\hat{\mathcal{C}}$ to a JSIL procedure proc $m(\mathtt{x}_{sc}, \mathtt{x}_{this}, \overline{\mathtt{x}})\{\overline{\mathtt{c}}\}$, whose name is the identifier of the compiled function and whose: first formal parameter corresponds to its scope chain in which the function was defined; second formal parameter corresponds to the value of the keyword this during the execution of the compiled function; and its remaining formal parameters match the formal parameters of the original function. This is illustrated in line 1 of the `enqueue` procedure in Figure 5.10, right. The function body s is compiled using the auxiliary compiler $\mathcal{C}$ to a JSIL command list $\overline{\mathtt{c}}$. We discuss the compiler $\mathcal{C}$ shortly. Here, we would like to mention that the compiler $\mathcal{C}$ returns not only the command list $\overline{\mathtt{c}}$, but also a list of error variables, $\overline{\mathtt{e}}$, a list of return variables $\overline{\mathtt{r}}$ (and a list of break variables, which is not used by $\hat{\mathcal{C}}$). The compiler $\hat{\mathcal{C}}$ produces additional JSIL commands before and after the command list $\overline{\mathtt{c}}$. Before the command list $\overline{\mathtt{c}}$, we need to deal with local variables, $\mathsf{defs}(s)$ and the formal parameters $x_i|_{i=1}^n$. They all are properties of the newly created environment record $\mathtt{x}_{er}$. The current scope chain is constructed by appending $\mathtt{x}_{er}$ to the scope $\mathtt{x}_{sc}$ in which the function was defined. All local variables are initialised with undefined, while formal parameters are set to the corresponding values of the arguments (see lines 2-6 of the `enqueue` procedure in Figure 5.10, right). After the command list $\overline{\mathtt{c}}$, we finalise the procedure body by creating the return and error sections. In the return section `pret`, we use a ϕ node to set the value of the return variable `xret`. All possible return values are stored in the list of variables $\overline{\mathtt{r}}$. It might be that a function body does not contain any return statements, in which case, the function should return `undefined`. This behaviour is ensured by an additional JSIL local variable $\mathtt{x}_{n+1}$. In our example, there are no return statements, hence the set $\overline{\mathtt{r}}$ is empty and the return variable `x` is assigned the value `undefined` (lines 75-76). Correspondingly, we deal with the error section, where the list $\overline{\mathtt{e}}$ holds all the possible error variables. See lines 77-78 of the `enqueue` procedure. We will see how the list of break variables are used by the compiler $\mathcal{C}$ itself in the compilation of JavaScript statements.

Next, we present the compiler $\mathcal{C}$.

5.4.3. Compiling Expressions and Statements

We separate the compiler $\mathcal{C}$ in two parts: one for compiling expressions and another one for compiling statements. The *expression* compiler $\mathcal{C}$ takes three parameters: the unique identifier m of the function in which the code to compile occurs; an ES5 Strict expression e; and a JSIL variable $\mathtt{x}$. It outputs a command list $\overline{\mathtt{c}}$ corresponding to the compilation of e, where the return value of e is stored in $\mathtt{x}$ after each terminating execution of $\overline{\mathtt{c}}$, together with a list of error variables, $\overline{\mathtt{e}}$. For simplicity, we write $\mathcal{C}_m(e, \mathtt{x})$ instead of $\mathcal{C}(m, e, \mathtt{x})$. The compiler $\mathcal{C}$ branches on the type of the given expression (Figure 5.11, left). The *statement* compiler $\mathcal{C}$ takes the unique identifier m of the function, an ES5 Strict statement s and, in addition to a JSIL variable $\mathtt{x}$, it takes two more parameters required for the compilation of `break` statements. The additional parameters are: another JSIL variable $\mathtt{x}_{pr}$, which denotes a value of the previously executed statement, and a label b_r to denote the end of the loop which is terminated by the `break` statement. The statement compiler $\mathcal{C}$ outputs the compiled list of statements $\overline{\mathtt{c}}$, a list of errors variables $\overline{\mathtt{e}}$, as well as a list of return variables, $\overline{\mathtt{r}}$, which denotes the results of `return` statements, and a list of break variables, $\overline{\mathtt{b}}$, which denotes the results of `break` statements. The compiler $\mathcal{C}$ branches on the type of the given statement (Figure 5.11, right). Following the `enqueue` example, we describe in detail how to compile variables, constructor call expressions, and sequence statements. We also explain the compilation of `break`, `while`, and `return` statements to illustrate the need for additional parameters $\mathtt{x}_{pr}$ and b_r, and outputs $\overline{\mathtt{r}}$ and $\overline{\mathtt{b}}$ for the statement compiler. The full definition of the compiler $\mathcal{C}$ is given in Appendix §B.

$\mathcal{C}_m \triangleq$ **lambda** $e, \mathtt{x}$.	**Inputs:** an expression e and a JSIL variable $\mathtt{x}$
match e **with**	Branching on the type of expression
$\mid \lambda \Rightarrow$ $\overline{\mathtt{c}}, \overline{\mathtt{e}}$	**Literal value**
$\mid \mathsf{this} \Rightarrow$ $\overline{\mathtt{c}}, \overline{\mathtt{e}}$	**This**
$\mid x \Rightarrow$ $\overline{\mathtt{c}}, \overline{\mathtt{e}}$	**Variable**
$\mid \ldots$	

$\mathcal{C}_m \triangleq$ **lambda** $s, \mathtt{x}, \mathtt{x}_{pr}, b_r$.	**Inputs:** a statement s, a JSIL variable $\mathtt{x}$, that denotes the result of the current statement, a JSIL variable $\mathtt{x}_{pr}$, that denotes the result of the previous statement, and a label b_r for `break` statements
match s **with**	Branching on the type of statement
$\mid \mathsf{var}\, x \Rightarrow$ $\overline{\mathtt{c}}, \overline{\mathtt{e}}, \overline{\mathtt{r}}, \overline{\mathtt{b}}$	**Variable Declaration**
$\mid e \Rightarrow$ $\overline{\mathtt{c}}, \overline{\mathtt{e}}, \overline{\mathtt{r}}, \overline{\mathtt{b}}$	**Expression Statement**
$\mid s_1; s_2 \Rightarrow$ $\overline{\mathtt{c}}, \overline{\mathtt{e}}, \overline{\mathtt{r}}, \overline{\mathtt{b}}$	**Sequence Statement**
$\mid \ldots$	

Figure 5.11.: The structure of the compiler $\mathcal{C}$

The Variables. We have already talked about the compilation of variables in §5.1. Recall that there we use the scope clarification function, which returns an index in the current scope chain from which we can obtain the required ER. Let us now formally define the scope clarification function, by giving the definition of the scope clarification function constructor Φ.

Definition 5.1 (Scope Clarification Construction $\Phi^i_m(s,\psi) = \psi'$)**.**

$$\Phi^i_m(s,\psi) = \begin{cases} \psi & \text{if } s = \lambda_{\mathsf{JS}} \vee s = \mathit{this} \vee s = \{\,\} \vee s \in \mathcal{X}_{\mathsf{JS}} \vee s = \mathit{var}\, x \\ \Phi^i_m(e,\psi) & \text{if } s = \ominus e \vee s = e \vee s = \mathit{return}\, e \vee s = \mathit{throw}\, e \\ \Phi^i_m(e_2, \Phi^i_m(e_1,\psi)) & \text{if } s = e_1 \oplus e_2 \vee s = (e_1 = e_2) \vee s = e_1[e_2] \\ \Phi^i_m(e_n, \Phi^i_m(\ldots, \Phi^i_m(e_0,\psi))) & \text{if } s = e_0(e_1,...,e_n) \vee s = \mathit{new}\ e_0(e_1,...,e_n) \\ \Phi^i_m(s_2, \Phi^i_m(s_1,\psi)) & \text{if } s = s_1;\ s_2 \\ \Phi^i_m(s_2, \Phi^i_m(s_1, \Phi^i_m(e,\psi))) & \text{if } s = \mathit{if}(e)\,\{s_1\}\,\mathit{else}\,\{s_2\} \\ \Phi^i_m(s, \Phi^i_m(e,\psi)) & \text{if } s = \mathit{while}(e)\,\{s\} \\ \Phi^{i+1}_{m'}(s', \hat{\psi}) & \text{if } \begin{cases} s = \mathit{function}\,(\overline{x})\{s'\}^{m'} \\ \hat{\psi} = \psi \left[\begin{array}{l} (m', x_k) \mapsto i \mid_{x_k \in \overline{x} \,\cup\, \mathsf{defs}(s')} \\ (m', y_j) \mapsto \psi(m, y_j) \mid_{y_j \in \mathsf{dom}(\psi(m)) \setminus (\overline{x} \,\cup\, \mathsf{defs}(s'))} \end{array} \right] \end{cases} \end{cases}$$

The scope clarification function constructor $\Phi^i_m(s,\psi)$ takes four arguments: a function identifier m, the depth of the function nesting i, a JavaScript statement s, and a constructed scope clarification function so far, ψ. The result of the constructor is the extended scope clarification function ψ'. Given the global code s, we construct the scope clarification function starting from an empty function: $\Phi^0(\mathsf{function}\,()\{s\}^{\mathsf{main}}, \emptyset)$. To construct the scope clarification function, we traverse the given JavaScript expression or statement. The interesting case is that of the function literal, whereas all other cases simply recursively call the scope clarification constructor for expressions and statements they contain. In the function literal case, $\mathsf{function}\,(\overline{x})\{s'\}^{m'}$, we update the already constructed ψ by saying that: the parameters $\overline{x}$ and local variables $\mathsf{defs}(s')$ of the function m' are defined in the environment record with the index i; the other variables accessible in the enclosed function m, $\mathsf{dom}(\psi(m)) \setminus (\overline{x} \cup \mathsf{defs}(s'))$, are defined in the environment records whose indexes are the same as in the enclosed function m. The result of the function literal case is the recursive call for the function m', increased depth of the nesting $i+1$, the body s' and an updated scope clarification function $\hat{\psi}$. Figure 5.12 shows a part of the scope clarification function for the identifiers `main`, `Module`, and `enqueue` from our running example. For simplicity, we write $\psi_m(x)$ instead of $\psi(m,x)$.

m	x	index
main	PriorityQueue	0
	q	0
	r	0
Module	Node	1
	module	1
	PriorityQueue	0
	q	0
	r	0

m	x	index
enqueue	pri	2
	val	2
	n	2
	Node	1
	module	1
	PriorityQueue	0
	q	0
	r	0

Figure 5.12.: A part of Scope Clarification Function $\psi(m,x)$ for our running example

Currently, our construction of the scope clarification function requires the entire program, but it should be relatively easy to consider the library code and the client code separately. For the library code, we would use the scope clarification function constructor in the same way, starting from an empty function: $\Phi^0(\mathsf{function}\,()\{s\}^{\mathsf{main}}, \emptyset)$. For the client code, we would start from a function ψ^{lib} that already contains some global variables: $\Phi^0(\mathsf{function}\,()\{s\}^{\mathsf{main}}, \psi^{lib})$. In our running example, if we were to split the library code (lines 1-48) and the client code (lines 50-54), ψ^{lib} would correspond to

$\mathcal{C}_m \triangleq$ **lambda** $e, \mathtt{x}$.
match e **with**
| $x \Rightarrow$
let $\mathtt{x}', \mathtt{x}_h, \mathtt{x}'_1, \mathtt{x}'_2 = \mathsf{fresh}()$;
$t, e, n = \mathsf{fresh}()$; **in**
match $\psi_m(x)$ **with**
| $i \Rightarrow$
$\mathtt{x}' := \mathsf{nth}\,(\mathtt{xscope}, i)$
$\mathtt{x} := [\text{"v"}, \mathtt{x}', x]$,
[ ]
| $\bot \Rightarrow$
$\mathtt{x}_h := \mathtt{hasProperty}(l_g, x)$ *with* perr
goto $[\mathtt{x}_h]\ t,\ e$
$t : \mathtt{x}'_1 := l_g$
goto n
$e : \mathtt{x}'_2 := \mathsf{undefined}$
$n : \mathtt{x}' := \phi(\mathtt{x}'_1, \mathtt{x}'_2)$
$\mathtt{x} := [\text{"v"}, \mathtt{x}', x]$,
$[\mathtt{x}_h]$

```
1  proc enqueue (x__s, x__this, pri, val) {
                          ⋮
7           x_1 := nth (x__scope, 2); /* n */
8           x_2 := [ "v", x_1, "n" ];
9           x_3 := nth (x__scope, 1); /* Node */
10          x_4 := [ "v", x_3, "Node" ];
                          ⋮
12          x_5 := nth (x__scope, 2); /* pri */
13          x_6 := [ "v", x_5, "pri" ];
                          ⋮
15          x_7 := nth (x__scope, 2); /* val */
16          x_8 := [ "v", x_7, "val" ];
                          ⋮
44          x_22 := nth (x__scope, 2); /* n */
45          x_23 := [ "v", x_22, "n" ];
                          ⋮
79 }
```

Figure 5.13.: Compiling JavaScript Variables

the function defined only for (`main`, `PriorityQueue`) to hold value 0, that is, ψ(`main`, `PriorityQueue`)$= 0$. This way, we could separately compile and verify the library and the client.

We note that ES5 itself does not natively support modules, but there exists a number of ways for one to write modular JavaScript code: for example, by using script tags in DOM or the module system provided by `Node.js`. Moreover, ES6 provides native support for modules. We would need to revisit the construction of scope clarification function in order to support such module systems and achieve a fully modular translation from JavaScript to JSIL.

We are now ready to formalise the compilation of variables, which is given in Figure 5.13 (left). If the scope clarification function $\psi_m(x)$ returns an index i, we construct the reference and we are done. The list of error variables is empty. The right hand side of Figure 5.13 shows how variables used in the function `enqueue` are compiled. In the case where the scope clarification function is undefined, that is, $\psi_m(x)$ returns $\bot$, we need to check if the global object has the property x. This is done by calling the internal function `HasProperty`, which traverses the prototype chain of the global object and returns true if it has the property and false otherwise. If the global object has the required property, a variable reference with the base l_g is constructed, otherwise a variable reference with the base undefined is constructed. Finally, we use ϕ to construct the appropriate SSA code. Note that the list of error variables includes $\mathtt{x}_h$, to cover the case in which `HasProperty` throws an error, which is then stored in the variable $\mathtt{x}_h$. Also note that if `HasProperty` throws an error, the control flow jumps to the error section `perr`, created in the compilation of the function literals in Figure 5.10. Since our fragment of ES5 Strict does not include `try−catch−finally`, every time an exception is thrown, the control flow jumps to the error section. In the full compiler, we need to know more information about the context when we compile expressions and statements. For example, if we are in the `try` block, we need to jump to the `catch` block when an exception is being thrown.

Constructor Calls. Figure 5.14 (left) formalises constructor call compilation and gives the compilation of `new Node(pri, val)` from the function `enqueue` (right). The compilation closely follows the operational semantics. First, the expression e is compiled to a list of commands $\overline{\mathtt{c}}_e$. In the example, this corresponds to compiling the variable `Node` (lines 9-10), which we just discussed in the compila-

$\mathcal{C}_m \triangleq$ **lambda** $e, \mathtt{x}.$ **match** e **with**
| new $e(e_1, \cdots, e_n) \Rightarrow$
let $\mathtt{x}_e, \mathtt{x}'_e, \mathtt{x}_i|_{i=1}^n, \mathtt{x}'_i|_{i=1}^n, \mathtt{x}_{err}, \mathtt{x}_{hp}, \mathtt{x}_l, \mathtt{x}_r,$
$\mathtt{x}'_p, \mathtt{x}''_p, \mathtt{x}_p, \mathtt{x}'_l, \mathtt{x}_{m'}, \mathtt{x}_{scp}, \mathtt{x}' = \mathsf{fresh}();$
$t_1, t_2, t_3, e_1, e_2, e_3, n = \mathsf{fresh}();$
$\overline{\mathtt{c}}_e, \overline{\mathtt{e}}_e = \mathcal{C}_m(e, \mathtt{x}_e);$
$\overline{\mathtt{c}}_i, \overline{\mathtt{e}}_i = \mathcal{C}_m(e_i, \mathtt{x}_i)|_{i=1}^n;$
$\overline{\mathtt{c}}'_i = \mathtt{x}'_i := \mathtt{getValue}(\mathtt{x}_i)$ *with* $\mathtt{perr}|_{i=1}^n;$
in
$\overline{\mathtt{c}}_e$
$\mathtt{x}'_e := \mathtt{getValue}(\mathtt{x}_e)$ *with* perr
$\{\overline{\mathtt{c}}_i :: \overline{\mathtt{c}}'_i\}|_{i=1}^n$
goto [typeOf$(\mathtt{x}'_e)$!= Obj] t_1, e_1
$t_1 : x_{err} := \mathtt{typeError}()$
goto perr
$e_1 : \mathtt{x}_{hp} :=$ hasProperty$(\mathtt{x}'_e, @code)$
goto [$\mathtt{x}_{hp}$] n, t_1
$n : \mathtt{x}_l :=$ new$()$
$\mathtt{x}_r := [$"o", $\mathtt{x}'_e, prototype]$
$\mathtt{x}'_p := \mathtt{getValue}(\mathtt{x}_r)$ *with* perr
goto [typeOf$(\mathtt{x}'_p)$ = Obj] t_2, e_2
$e_2 : \mathtt{x}''_p := l_{op}$
$t_2 : \mathtt{x}_p := \phi(\mathtt{x}'_p, \mathtt{x}''_p)$
$\mathtt{x}'_l := \mathtt{defaultObj}(\mathtt{x}_l, \mathtt{x}_p)$
$\mathtt{x}_{m'} := [\mathtt{x}'_e, @code]$
$\mathtt{x}_{scp} := [\mathtt{x}'_e, @scope]$
$\mathtt{x}' := \mathtt{x}_{m'}(\mathtt{x}_{scp}, \mathtt{x}_l, \mathtt{x}'_i|_{i=1}^n)$ with perr
goto [typeOf$(\mathtt{x}')$ = Obj] t_3, e_3
$e_3 :$ skip
$t_3 : \mathtt{x} := \phi(\mathtt{x}', \mathtt{x}_l),$
$\overline{\mathtt{e}}_e :: [\mathtt{x}'_e] :: (\overline{\mathtt{e}}_i :: [\mathtt{x}'_i])|_{i=1}^n :: [\mathtt{x}_{err}, \mathtt{x}'_p, \mathtt{x}']$

```
1  proc enqueue (x__s, x__this, pri, val) {
                          ⋮
9           x_3 := nth (x__scope, 1);
10          x_4 := [ "v", x_3, "Node" ];
11          x_4_v := "i__getValue"(x_4) with perr;
12          x_5 := nth (x__scope, 2);
13          x_6 := [ "v", x_5, "pri" ];
14          x_6_v := "i__getValue"(x_6) with perr;
15          x_7 := nth (x__scope, 2);
16          x_8 := [ "v", x_7, "val" ];
17          x_8_v := "i__getValue"(x_8) with perr;
18          goto [(not (typeOf(x_4_v) = object_type))] terr next_1;
19    terr: x_err := "TypeError"();
20          goto perr;
21  next_1: x_9 := hasProperty(x_4_v,"@code");
22          goto [x_9] next_2 terr;
23  next_2: x_10 := new();
24          x_11 := [ "o", x_4_v, "prototype" ];
25          x_11_v := "i__getValue"(x_11) with perr;
26          goto [(typeOf(x_11_v) = object_type)] then_1 else_1;
27  else_1: x_12 := lobj_proto;
28  then_1: x_13 := PHI(x_11_v, x_12);
29          x_14 := "create_default_object"(x_10, x_13);
30          x_fid_1 := [x_4_v, "@code"];
31          x_fscope_1 := [x_4_v, "@scope"];
32          x_15 := x_fid_1(x_fscope_1, x_10, x_6_v, x_8_v) with
                 perr;
33          goto [(typeOf(x_15) = object_type)] then_2 else_2;
34  else_2: skip;
35  then_2: x_16 := PHI(x_15, x_10);
                          ⋮
79 }
```

Figure 5.14.: Compiling Constructor Calls

tion of JavaScript variables. Next, `GetValue` is used to obtain the function object $\mathtt{x}'_e$ (line 11). Then, we compile the arguments and obtain their values using `GetValue`. In our example, this corresponds to lines 12-17 for arguments `pri` and `val`. A type error should occur if $\mathtt{x}'_e$ is not an object or if it does not have an internal property $@code$ (lines 18-22). Otherwise, a new object $\mathtt{x}_l$ is created (line 23). The prototype of the newly created object is set to the value of the property `"prototype"` of the function object $\mathtt{x}'_e$, with an exception that if the property `"prototype"` does not contain an object, `Object.prototype` is used instead (lines 24-29). Note the use of the ϕ node to ensure SSA in line 28. To call the function, we first retrieve its unique identifier m' from the internal property $@code$ and its scope, $\mathtt{x}_{scp}$, from $@scope$ (lines 30-32). Finally, the newly created object $\mathtt{x}_l$ is the result of the compilation of the constructor call, unless the function returns an object $\mathtt{x}'$, which then is the result instead. This, again, is handled by a ϕ function (lines 33-35). We also need to collect all the possible error variables from the compilations of subexpressions: $\overline{\mathtt{e}}_e :: [\mathtt{x}'_e] :: (\overline{\mathtt{e}}_i :: [\mathtt{x}'_i])|_{i=1}^n :: [\mathtt{x}_{err}, \mathtt{x}'_p, \mathtt{x}']$, which in our example correspond to [`x_4_v`, `x_6_v`, `x_8_v`, `x_err`, `x_11_v`, `x_15`].

Sequences. To compile a sequence $s_1; s_2$ (Figure 5.15), we first compile s_1, followed by the compilation of s_2. In our example, the compilation of the first statement `var n = new Node(pri, val)` compiles to the command list in the lines 9-39, while the second statement `this._head = n.insertToQueue(this._head)` compiles to lines 40-71. Recall the semantics of sequence, which states that if a statement evaluates to `empty`, the result is the last non-empty value of the previous statements. This behaviour is reflected in lines 72-74. Note that in the compilation of the statement s_2 we provide the result variable $\mathtt{x}_1$ of the compilation of statement s_1 as a third argument. It denotes the result of the previous statement and as we will see next is important for compilation of `break` statements.

$$
\begin{array}{l}
\mathcal{C}_m \triangleq \textbf{lambda}\, s, \mathtt{x}, \mathtt{x}_{pr}, b_r. \\
\quad \textbf{match}\ s\ \textbf{with} \\
\quad \mid s_1; s_2 \Rightarrow \\
\quad\quad \textbf{let}\ \mathtt{x}_1, \mathtt{x}_2 = \mathsf{fresh}(); \\
\quad\quad\quad t, e = \mathsf{fresh}(); \\
\quad\quad\quad \overline{\mathtt{c}}_1, \overline{\mathtt{e}}_1, \overline{\mathtt{r}}_1, \overline{\mathtt{b}}_1 = \mathcal{C}_m(s_1, \mathtt{x}_1, \mathtt{x}_{pr}, b_r); \\
\quad\quad\quad \overline{\mathtt{c}}_2, \overline{\mathtt{e}}_2, \overline{\mathtt{r}}_2, \overline{\mathtt{b}}_2 = \mathcal{C}_m(s_2, \mathtt{x}_2, \mathtt{x}_1, b_r); \\
\quad\quad\quad \textbf{in} \\
\quad\quad\quad\quad \overline{\mathtt{c}}_1 \\
\quad\quad\quad\quad \overline{\mathtt{c}}_2 \\
\quad\quad\quad\quad \mathsf{goto}\ [\mathtt{x}_2 = \mathsf{empty}]\ t,\ e \\
\quad\quad\quad e : \mathsf{skip} \\
\quad\quad\quad t : \mathtt{x} := \phi(\mathtt{x}_1, \mathtt{x}_2), \\
\quad\quad\quad \overline{\mathtt{e}}_1 :: \overline{\mathtt{e}}_2, \overline{\mathtt{r}}_1 :: \overline{\mathtt{r}}_2, \overline{\mathtt{b}}_1 :: \overline{\mathtt{b}}_2
\end{array}
$$

```
1  proc enqueue (x__s, x__this, pri, val) {
                          ⋮
9           x_3 := nth (x__scope, 1); /* Node */
                          c̄1

39          x_r_1 := empty;
40          x_19 := x__this; /* this */
                          c̄2

71          x_34 := "i__putValue"(x_21, x_32_v) with perr;
72          goto [(x_32_v = empty)] then_5 else_5;
73  else_5: skip;
74  then_5: x_35 := PHI(x_r_1, x_32_v);
                          ⋮
79 }
```

Figure 5.15.: Compiling Sequences

Break, While, Return. We need the parameters $\mathtt{x}_{pr}$ and b_r in the compiler $\mathcal{C}$ for the compilation of the `break` statements. JavaScript `break` statements can be used only inside `while` loops, to terminate their execution. Figure 5.16 shows the formalisation of the `break` and `while` statements. We do not explain all the details of the compilation, as there are descriptions in the figure. However, we do describe the main points. In the compilation of the break statement we need to jump to the end of the while loop, which is given as the parameter b_r (**Break Statement a. line 2**). We provide the label of the end of the loop, b (**While Statement e., line 12**), for the compilation of the body of the `while` (**While Statement d.**). We also need to set the correct result value for the `break` statement, for which the parameter $\mathtt{x}_{pr}$ is used. Let us look at some examples of `break`.

```
1    eval("while(true){ break; }") // undefined
2    eval("x=3; while(true){ break; }") // 3
3    eval("while(true){ x=3; break; }") // 3
4    eval("i=0; while (i++ < 2) {if (i===1) { x=3; } else { break; }}") // 3
```

When `break` terminates the loop and there was no previous statement to evaluate, the result of the statement is the special value `empty`, which is transformed to `undefined` by the `eval` command (line 1). If, on the other hand, there was a previous statement to evaluate (line 2), the overall result will be the result of that previous statement. Similarly, if there was previous statement in the body of the while loop before the `break` happens, the overall result is the result of the previous statement (line 3). Finally, the most complicated case is illustrated on line 4. There will be two iterations of the `while` loop. The first iteration finishes with the result 3, while the second iteration breaks the loop with an `empty` value. The overall expression is the result of the previous iteration of the while loop.

To capture the behaviour of remembering the value of the previous statement, we use the parameter $\mathtt{x}_{pr}$. We have already seen in the compilation of the sequence statement how to correctly provide this argument. Similarly, in the compilation of the while loop, we provide the result of the while loop, which is stored in the variable $\mathtt{x}''$ until the loop has reached its end label. In the end label of the while loop (b) (**While Statement e., line 12**) we use a ϕ node to set the result of the `while` statement, including all possible `break` statements with their values, denoted by variables $\overline{\mathtt{b}}_s$.

Note that the the compilation of the while loop consumes $\overline{\mathtt{b}}_s$, that is, it returns an empty list of break variables, since `break` statements can occur only in the `while` loops. In our fragment of ES5 Strict, we do not consider breaking to labels, `break l`. In full ES5 Strict, `break` statements can have labels which denote which enclosed `while` loop the `break` terminates. To account for such cases, while loops need

$\mathcal{C}_m \triangleq$ **lambda** $s, \mathtt{x}, \mathtt{x}_{pr}, b_r.$

Inputs: a statement s, a JSIL variable $\mathtt{x}$, that denotes the result of the current statement, a JSIL variable $\mathtt{x}_{pr}$, that denotes the result of the previous statement, and a label b_r for break statements

match s **with**

Branching on the type of statement to compile

| break ⇒

$\mathtt{x} := \mathtt{x}_{pr}$
goto b_r,
$[\,], [\,], [\mathtt{x}]$

Break Statement
- **a.** Generated code:
  - **1.** The result is the provided previous value $\mathtt{x}_{pr}$
  - **2.** Go to the provided break label

| while(e) {s} ⇒
let $\mathtt{x}_e, \mathtt{x}_s, \mathtt{x}'_e, \mathtt{x}', \mathtt{x}'', \mathtt{x}''', \mathtt{x}_b = \mathsf{fresh}();$
$t, e, n, h, b = \mathsf{fresh}();$
$\overline{\mathtt{c}}_e, \overline{\mathtt{e}}_e = \mathcal{C}_m(e, \mathtt{x}_e);$
$\overline{\mathtt{c}}_s, \overline{\mathtt{e}}_s, \overline{\mathtt{r}}_s, \overline{\mathtt{b}}_s = \mathcal{C}_m(s, \mathtt{x}_s, \mathtt{x}'', b);$
in
$\mathtt{x}' := \mathsf{empty}$
$h : \mathtt{x}'' := \phi(\mathtt{x}', \mathtt{x}''')$
$\overline{\mathtt{c}}_e$
$\mathtt{x}'_e := \mathtt{getValue}(\mathtt{x}_e)$ *with* perr
$\mathtt{x}_b := \mathtt{toBoolean}(\mathtt{x}'_e)$ *with* perr
goto $[\mathtt{x}_b]$ n, b
$n : \overline{\mathtt{c}}_s$
goto $[\mathtt{x}_s$!= empty$]$ t, e
$t :$ skip
$e : \mathtt{x}''' := \phi(\mathtt{x}'', \mathtt{x}_s)$
goto h
$b : \mathtt{x} := \phi(\mathtt{x}'', \overline{\mathtt{b}}_s),$
$\overline{\mathtt{e}}_e :: [\mathtt{x}'_e, \mathtt{x}_b] :: \overline{\mathtt{e}}_s, \overline{\mathtt{r}}_s, [\,]$

While Statement
- **a.** Fresh vars
- **b.** Fresh labels
- **c.** Compile e
- **d.** Compile s
- **e.** Generated code:
  - **1.** Result of while is empty in case no iterations occur
  - **2.** Joining the branch of no iterations with the branch iterating the while body
  - **3.** Compilation of e
  - **4.** Dereferencing of $\mathtt{x}_e$
  - **5.** Converting the while condition to boolean
  - **6.** Branch on $\mathtt{x}_b$
  - **7.** The while condition holds: compilation of s
  - **8.** Branch on $\mathtt{x}_s$ not being equal to empty
  - **9.** $\mathtt{x}_s$ is not equal to empty: the result is $\mathtt{x}_s$
  - **10.** $\mathtt{x}_s$ is empty: the result is the value of the previous iteration
  - **11.** Proceed to the next iteration
  - **12.** The while condition does not hold: exit the loop

Figure 5.16.: Compilation of `break` and `while`

to contain labels. To capture this, the full compiler takes not just one label b_r as a parameter, but a list of label pairs, denoting the loop itself and the end of the loop, for all enclosing while loops.

To finish up, we describe the compilation of the return statement to illustrate how the return variables $\overline{\mathtt{r}}$ are being set (Figure 5.17). We compile the given expression, dereference it, and jump to the return section `pret`. The output of the compilation includes the list of return variables $[\mathtt{x}]$. Recall the creation of the return section `pret` and the treatment of return variables $\overline{\mathtt{r}}$ in the compilation of function literals in Figure 5.10.

Phi-nodes. Note that, for our formally defined JavaScript fragment, in the formalisation of the compiler $\overline{\mathcal{C}}$ it is sufficient to use the ϕ-assignment for only one variable: $\mathtt{x} := \phi(\overline{\mathtt{x}})$. In the implemented full compiler, however, we require the more general form of ϕ assignment for the translation of the

$\mathcal{C}_m \triangleq$ **lambda** $s, \mathtt{x}, \mathtt{x}_{pr}, b_r.$

Inputs: a statement s, a JSIL variable $\mathtt{x}$, that denotes the result of the current statement, a JSIL variable $\mathtt{x}_{pr}$, that denotes the result of the previous statement, and a label b_r for break statements

match s **with**

Branching on the type of statement to compile

| return e ⇒
let $\mathtt{x}_e = \mathsf{fresh}();$
$\overline{\mathtt{c}}_e, \overline{\mathtt{e}}_e = \mathcal{C}_m(e, \mathtt{x}_e);$
in
$\overline{\mathtt{c}}_e$
$\mathtt{x} := \mathtt{getValue}(\mathtt{x}_e)$ *with* perr
goto pret,
$\overline{\mathtt{e}}_e :: [\mathtt{x}], [\mathtt{x}], [\,]$

Return Statement
- **a.** Fresh vars
- **b.** Compile e
- **c.** Generated code:
  - **1.** Compilation of e
  - **2.** Dereferencing of $\mathtt{x}_e$
  - **3.** Go to the return section

Figure 5.17.: Compilation of `return`

`switch` and `for−in` statements, where we have to keep track of two variables.

5.5. JS-2-JSIL Validation: Compiler Correctness

We say that a JS-2-JSIL compiler is *correct* if: "whenever a ES5 Strict program and its compilation are evaluated in two heaps that are equal, 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 equal".

Recall the ES5 Strict semantic relation $\wp, L, v_t \vdash \langle h, m(\overline{x}, \overline{v})\rangle \Downarrow_m \langle h', o\rangle$ (defined in §3.4), where: $\wp$ is a JavaScript program and m is a function identifier; L is the current scope chain and v_t is the this value; h is the initial heap and $m(\overline{x}, \overline{v})$ states that it is the body of the function with identifier m that we evaluate; and h' is the final heap and o the *outcome* of the evaluation. Also recall the JSIL judgement $\mathtt{p} \vdash \langle h, \rho, \_, 0\rangle \Downarrow_m \langle h', \rho', \mathit{fl}\langle v\rangle\rangle$ (defined in §4.2), which says that the evaluation of procedure m of program $\mathtt{p}$, starting from the beginning, in the heap h and store ρ, generates the heap h', the store ρ', and results in the outcome $\mathit{fl}\langle v\rangle$.

The Compiler Correctness Theorem 5.1 is stated below. It describes the compiler correctness at the level of functions. That is, the behaviour the every function m in the program $\wp$ is equivalent to the behaviour of its corresponding procedure m in JSIL. We relate formal parameters $\overline{x}$ and their values $\overline{v}$ in JavaScript by adding them to the store ρ in JSIL. Moreover, the scope chain L and the this value are stored in the JSIL variables $\mathtt{x}_{sc}$ and $\mathtt{x}_{\mathsf{this}}$ respectively. Finally, the output evaluating function body in JavaScript can be either normal output with value v or an error output $\mathsf{error}\, v$. This is related to the JSIL outputs $\mathit{fl}\langle v\rangle$, for flags nm and er, using the function $\mathsf{out_{JS}}(\mathit{fl}, v)$.

Theorem 5.1 (Compiler Correctness)**.** *We say that a JS-2-JSIL compiler $\overline{\mathcal{C}}$ is correct if compiled programs preserve the behaviour of their original versions.*

$$\wp, L, v_t \vdash \langle h, m(\overline{x}, \overline{v})\rangle \Downarrow_m \langle h_f, \mathsf{out_{JS}}(\mathrm{fl}, v)\rangle \iff \exists \rho_f \,.\, \overline{\mathcal{C}}(\wp) \vdash \langle h, \rho, \_, 0\rangle \Downarrow_m \langle h_f, \rho_f, \mathrm{fl}\langle v\rangle\rangle$$

where $\rho = \emptyset[x_i \mapsto v_i|_{i=1}^{n}, \mathtt{x}_{sc} \mapsto L, \mathtt{x}_{\mathit{this}} \mapsto v_t]$ *and* $\mathsf{out_{JS}}(\mathrm{fl}, v) = \begin{cases} v, & \mathit{if}\ \mathrm{fl} = \mathsf{nm} \\ \mathit{error}\, v, & \mathit{if}\ \mathrm{fl} = \mathsf{er} \end{cases}$

Proof. Given in Appendix §B.2. □

6. JSIL Verification Infrastructure

We introduce JSIL verification infrastructure (Figure 6.1), which revolves around JSIL, presented in §4. We write specifications of JSIL programs using the JSIL Logic assertions, introduced in §6.1. A JSIL program, its specification, loop invariants and fold/unfold directives together form an annotated JSIL program. To be able to verify that a JSIL program satisfies its specification, we design JSIL Logic, a program logic for JSIL, introduced in §6.2. We prove JSIL Logic to be sound with respect to the operational semantics of JSIL in §6.3. Finally, we give a high-level overview of JSIL Verify, a semi-automatic verification tool for JSIL, based on JSIL Logic, in §6.4.

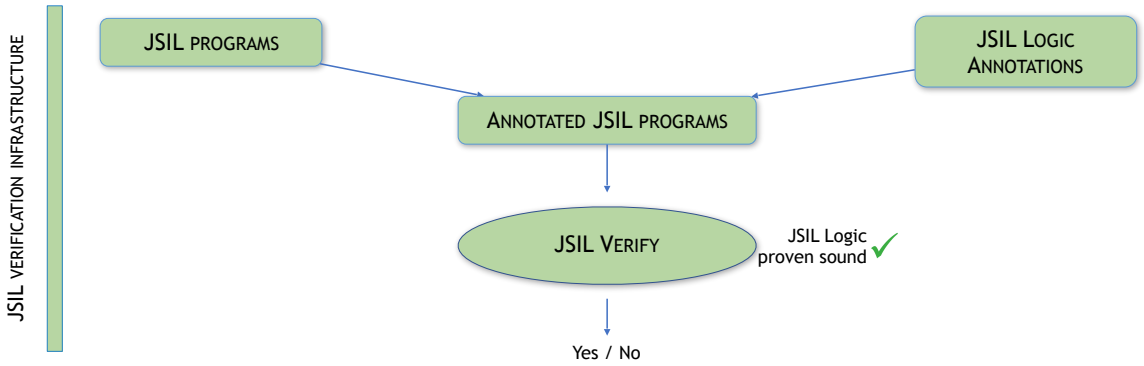


Figure 6.1.: JSIL Verification Infrastructure

6.1. JSIL Logic Assertions

Since our aim is the mechanised verification of JavaScript programs, we take inspiration for the JSIL Logic assertions from the assertion languages provided by tools based on separation logic for other programming languages [4, 22, 72, 13, 11, 14], mostly by keeping the assertion language as standard and as simple as possible. However, JSIL inherits the dynamic features of JavaScript, namely extensible objects and dynamic property access, meaning that the corresponding assertion language has to accommodate for these features and, therefore, be more expressive than those for static languages.

To adapt separation logic for reasoning about JavaScript programs, Gardner et al. [30] introduce an assertion to describe negative information about an existence of a property in an object, $(l, p) \mapsto \varnothing$, and a new connective sepish, $\between$, to account for possible sharing. We draw partial inspiration from this work: our property assertions are similar, however, we do not use sepish. Sepish gives us more flexibility in writing specifications, however, at the cost of the ability to prove properties. In §8, we illustrate the problem of using sepish in specifications and show that we are able to write specifications without sepish.

JSIL Logic assertions are given in Figure 6.2. JSIL logical values, $\mathrm{V} \in \mathcal{V}^{L}_{\mathtt{JSIL}}$, consist of JSIL values,

$$
\begin{array}{rrcll}
\textsc{Logical Values} & \mathrm{V} \in \mathcal{V}^L_{\mathtt{JSIL}} & \triangleq & \mathtt{v} \mid \mathtt{v_{set}} \mid \varnothing & \\
\textsc{Logical Expressions} & \mathrm{E} \in \mathcal{E}^L_{\mathtt{JSIL}} & \triangleq & \mathrm{V} \mid \mathtt{x} \mid \mathrm{X} \mid \ominus \mathrm{E} \mid \mathrm{E} \oplus \mathrm{E} & \\
\\
\textsc{JSIL Assertions} & P \in \mathcal{AS}_{\mathtt{JSIL}} & \triangleq & \mathrm{true} \mid \mathrm{false} \mid P \wedge P \mid \neg P \mid & \textsc{Classical} \\
 & & & \exists \mathrm{X}.P \mid & \textsc{Quantification} \\
 & & & \mathrm{E} = \mathrm{E} \mid \mathrm{E} \leq \mathrm{E} \mid \mathrm{E} < \mathrm{E} \mid & \textsc{Equalities} \\
 & & & \mathtt{emp} \mid P * P \mid & \textsc{Separation Logic} \\
 & & & (\mathrm{E}, \mathrm{E}) \mapsto \mathrm{E} \mid & \textsc{JavaScript Cell} \\
 & & & \mathsf{types}(\overline{\mathrm{E} : \tau}) \mid \mathsf{emptyProps}(\mathrm{E} \mid \mathrm{E}) & \textsc{Predicates}
\end{array}
$$

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

Figure 6.2.: JSIL Logic Assertions, where $\mathtt{v} \in \mathcal{V}_{\mathtt{JSIL}}$ (Figure 4.2) and $\mathtt{x} \in \mathcal{X}_{\mathtt{JSIL}}$ (Figure 4.1)

sets of JSIL values, and the special value $\varnothing$, read *none*, used to denote the absence of a property in an object. This special value is inherited from the program logic for JavaScript presented in [30] and is required as a direct consequence of the extensibility of JSIL objects. More precisely, since properties of JSIL objects can be added and deleted arbitrarily, beside being able to express that a given object has a given property, which is completely standard, we also have to be able to express that a given object *does not have* a given property. JSIL logical expressions, $\mathrm{E} \in \mathcal{E}^L_{\mathtt{JSIL}}$, contain: logical values, V; JSIL program variables, $\mathtt{x}$; JSIL logical variables, X; unary operators, $\ominus$; and binary operators, $\oplus$. JSIL assertions include classical assertions as well as existential quantification over logical variables; equalities and relations on logical expressions; separation logic assertions for describing JSIL heaps, such as an empty heap and a composition of two disjoint heaps; a JavaScript heap cell assertion; and built-in predicates, which include the pure predicate types, which describes types of expressions, and a spatial predicate emptyProps, which denotes non-existent properties of an object. At the level of assertions, the dynamic nature of JSIL is captured by the heap cell assertion $(\mathrm{E}, \mathrm{E}) \mapsto \mathrm{E}$, which is more general than the one required for static languages in that the property name can also be an arbitrary expression and not necessarily a literal string. Also, the emptyProps predicate, whose full meaning we describe shortly, is introduced, similarly to the $\varnothing$ logical value, due to JSIL having extensible objects.

We provide the semantics of JSIL Logic assertions using a satisfiability relation $H, \rho, \epsilon \models P$. An assertion may be satisfied by a triple (H, ρ, ϵ), consisting of an *abstract heap* H, a JSIL store ρ (Figure 4.2), and a logical environment ϵ, mapping logical variables to logical values. Following the definition given in [30], an abstract heap maps pairs of locations and property names to logical values. Recall that logical values include the special value $\varnothing$, which we use to denote the absence of a given property in an object. That is, we write $(l, p) \mapsto \varnothing$ to state that the object at location l has no property named p. To define abstract heaps, we thus extend the range of JSIL heaps (Figure 4.2) with the $\varnothing$ value: $H \in \mathcal{H}^{\varnothing}_{\mathtt{JSIL}} : \mathcal{L} \times \mathcal{P}_{\mathtt{JSIL}} \rightharpoonup \mathcal{V}^L_{\mathtt{JSIL}}$.

The satisfiability relation for JSIL Logic assertions builds on the semantics of JSIL logical expressions. A JSIL logical expression E is interpreted with respect to the store ρ and logical environment ϵ, written $[\![\mathrm{E}]\!]^{\epsilon}_{\rho}$. Both the satisfiability relation and the interpretation of logical expressions are given in Figure 6.3. Most of the assertions are interpreted in the standard way, and here we only discuss those that are interesting. The assertion $(\mathrm{E}_1, \mathrm{E}_2) \mapsto \mathrm{E}_3$ describes an object at the location denoted by E_1 with a property denoted by E_2 mapped onto the value denoted by E_3. The assertion $\mathsf{types}(\overline{\mathrm{E} : \tau})$

provides typing information for JSIL expressions. We use the function TypeOf, which given a JSIL expression, outputs its type. The assertion $\mathsf{emptyProps}(\mathrm{E}_1 \mid \mathrm{E}_2)$ describes an object denoted by E_1 that has no properties other than possibly those included in the set denoted by E_2.

$$[\![\mathrm{V}]\!]^{\epsilon}_{\rho} \triangleq \mathrm{V} \qquad [\![\mathtt{x}]\!]^{\epsilon}_{\rho} \triangleq \rho(\mathtt{x}) \qquad [\![\mathrm{X}]\!]^{\epsilon}_{\rho} \triangleq \epsilon(\mathrm{X}) \qquad [\![\ominus\, \mathrm{E}]\!]^{\epsilon}_{\rho} \triangleq \ominus([\![\mathrm{E}]\!]^{\epsilon}_{\rho}) \qquad [\![\mathrm{E}_1 \oplus \mathrm{E}_2]\!]^{\epsilon}_{\rho} \triangleq \oplus([\![\mathrm{E}_1]\!]^{\epsilon}_{\rho}, [\![\mathrm{E}_2]\!]^{\epsilon}_{\rho})$$

$$\begin{array}{ll}
H, \rho, \epsilon \models \mathrm{true} & \Leftrightarrow \mathrm{always} \\
H, \rho, \epsilon \models \mathrm{false} & \Leftrightarrow \mathrm{never} \\
H, \rho, \epsilon \models P_1 \wedge P_2 & \Leftrightarrow H, \rho, \epsilon \models P_1 \wedge H, \rho, \epsilon \models P_2 \\
H, \rho, \epsilon \models \neg P & \Leftrightarrow H, \rho, \epsilon \not\models P \\
H, \rho, \epsilon \models \mathrm{E}_1 = \mathrm{E}_2 & \Leftrightarrow H = \mathsf{emp} \wedge [\![\mathrm{E}_1]\!]^{\epsilon}_{\rho} = [\![\mathrm{E}_2]\!]^{\epsilon}_{\rho} \\
H, \rho, \epsilon \models \mathrm{E}_1 \leq \mathrm{E}_2 & \Leftrightarrow H = \mathsf{emp} \wedge [\![\mathrm{E}_1]\!]^{\epsilon}_{\rho} \leq [\![\mathrm{E}_2]\!]^{\epsilon}_{\rho} \\
H, \rho, \epsilon \models \mathrm{E}_1 < \mathrm{E}_2 & \Leftrightarrow H = \mathsf{emp} \wedge [\![\mathrm{E}_1]\!]^{\epsilon}_{\rho} < [\![\mathrm{E}_2]\!]^{\epsilon}_{\rho} \\
H, \rho, \epsilon \models \mathsf{emp} & \Leftrightarrow H = \mathsf{emp} \\
H, \rho, \epsilon \models (\mathrm{E}_1, \mathrm{E}_2) \mapsto \mathrm{E}_3 & \Leftrightarrow H = ([\![\mathrm{E}_1]\!]^{\epsilon}_{\rho}, [\![\mathrm{E}_2]\!]^{\epsilon}_{\rho}) \mapsto [\![\mathrm{E}_3]\!]^{\epsilon}_{\rho} \\
H, \rho, \epsilon \models P_1 * P_2 & \Leftrightarrow \exists H_1, H_2.\, H = H_1 \uplus H_2 \ \wedge (H_1, \rho, \epsilon \models P_1) \wedge (H_2, \rho, \epsilon \models P_2) \\
H, \rho, \epsilon \models \exists \mathrm{X}.P & \Leftrightarrow \exists \mathrm{V} \in \mathcal{V}^{L}_{\mathtt{JSIL}}.\, H, \rho, \epsilon[\mathrm{X} \mapsto \mathrm{V}] \models P \\
H, \rho, \epsilon \models \mathsf{types}(\overline{\mathrm{E} : \tau}) & \Leftrightarrow H = \mathsf{emp} \ \wedge \ (\forall (\mathrm{E}, \tau) \in \overline{\mathrm{E} : \tau}.\, \mathsf{TypeOf}([\![\mathrm{E}]\!]^{\epsilon}_{\rho}) = \tau) \\
H, \rho, \epsilon \models \mathsf{emptyProps}(\mathrm{E}_1 \mid \mathrm{E}_2) & \Leftrightarrow H = \biguplus_{p \notin [\![\mathrm{E}_2]\!]^{\epsilon}_{\rho}} (([\![\mathrm{E}_1]\!]^{\epsilon}_{\rho}, p) \mapsto \varnothing)
\end{array}$$

Figure 6.3.: Interpretation of logical expressions and satisfiability of JSIL Logic Assertions

6.2. JSIL Logic

We define JSIL Logic specifications and present JSIL Logic by providing axiomatic semantics for JSIL basic commands and by defining symbolic execution for JSIL control flow commands.

JSIL Logic Specifications. A *procedure specification*, $S \in \mathcal{S}pec$, is of the form $\{P\}\, m(\overline{\mathtt{x}})\, \{Q\}$, where m is the procedure identifier, $\overline{\mathtt{x}}$ denotes a list of formal parameters of the procedure, and P and Q are the pre- and postconditions of the procedure. Furthermore, each specification is associated with a return mode $\mathit{fl} \in \{\mathsf{nm}, \mathsf{er}\}$, stating if the procedure returns in normal (nm) or in error (er) mode. We say that a JSIL specification $\{P\}\, m(\overline{\mathtt{x}})\, \{Q\}$ is *valid* with respect to 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.

Definition 6.1 (Validity of JSIL Logic Specifications)**.** *A JSIL Logic specification* $\{P\}\, m(\overline{x})\, \{Q\}$ *for return mode* fl *is valid with respect to a program* $\mathtt{p}$*, written* $\mathtt{p}, \mathrm{fl} \vDash \{P\}\, m(\overline{x})\, \{Q\}$*, if and only if, for all logical contexts* (H, ρ, ϵ)*, heaps* h_f*, stores* ρ_f*, flags* fl'*, and JSIL values* $\mathtt{v}$*, it holds that:*

$$\begin{array}{c}
H, \rho, \epsilon \models P \ \wedge \ \mathtt{p} \vdash \langle \lfloor H \rfloor, \rho, -, 0 \rangle \Downarrow_m \langle h_f, \rho_f, \mathrm{fl}'\langle \mathtt{v} \rangle \rangle \implies \\
\mathrm{fl}' = \mathrm{fl} \ \wedge \ \exists H_f\,.\, H_f, \rho_f, \epsilon \models Q \ \wedge \ \lfloor H_f \rfloor = h_f
\end{array}$$

We use the notation $\lfloor H \rfloor$ to denote the concrete heap obtained by restricting the abstract heap H to the elements of its domain not mapped to $\varnothing$. $\lfloor . \rfloor : \mathcal{H}^{\varnothing}_{\mathtt{JSIL}} \to \mathcal{H}_{\mathtt{JSIL}}$ transforms an abstract heap to

a concrete heap as follows:

$$\lfloor H \rfloor(l,x) \triangleq H(l,x) \iff (l,x) \in \mathsf{dom}(H) \ \wedge \ H(l,x) \neq \varnothing$$

.

We note the following limitation of our JSIL logic specifications: since the return mode fl is associated with the entire specification, we cannot capture some behaviours of non-deterministic programs. For instance, we cannot capture that a procedure, for the same precondition, has two postconditions with different return modes. To address this, we would need to associate a flag with each of the postconditions, rather than with the entire specification, in the style of $\{P\}\, m(\overline{x})\, \{Q\}_{\mathsf{nm}} \{Q\}_{\mathsf{er}}$.

Axiomatic Semantics of Basic Commands. Our Hoare triples for JSIL basic commands (presented in §4 Figure 4.4) are of the form $\{P\}\mathtt{bc}\{Q\}$, and have the following interpretation: "if the basic command bc is executed in a state satisfying P, then, if it terminates, it will do so in a state satisfying Q". Importantly, we assume that JSIL programs are in SSA form, that is, each variable can be assigned to only once. This takes away the need for standard substitutions in many of the axioms. The axioms for basic commands are given in Figure 6.4. We use the notation $\mathrm{E}_1 \doteq \mathrm{E}_2$ as shorthand for $\mathrm{E}_1 = \mathrm{E}_2 \wedge \mathsf{emp}$.

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

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

Var Assignment
$\{\mathsf{emp}\}\ \mathtt{x} := \mathsf{e}\ \{\mathtt{x} \doteq \mathsf{e}\}$

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

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

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

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

Get Properties
$$\frac{P = (\circledast_{i=1}^{n}(\mathsf{e}, \mathrm{X}_i) \mapsto \mathrm{Y}_i) * \mathsf{emptyProps}(\mathsf{e} \mid \{\, \mathrm{X}_i|_{i=1}^{n} \,\}) * (\circledast_{i=1}^{n} \mathrm{Y}_i \neq \varnothing)}{\{P\}\ \mathtt{x} := \mathsf{getProperties}\,(\mathsf{e})\ \{P * \mathtt{x} \doteq \mathrm{X}_i|_{i=1}^{n} * (\mathsf{ord}\,(\mathtt{x}) \doteq \mathsf{true})\}}$$

Figure 6.4.: Axiomatic Semantics of Basic Commands: $\{P\}\mathtt{bc}\{Q\}$

The Object Creation axiom states that the new object at x only contains the @*proto* property with value null. It is important to have the resource of all other empty properties after the creation of a new object. When we later add a new property p to the object x, we need to have the resource $(\mathtt{x}, p) \mapsto \_$ to perform property assignment. The Property Deletion axiom forbids the deletion of @*proto* properties. The Get Properties axiom states that if the object bound to e only contains the properties denoted by $\mathrm{X}_i|_{i=1}^{n}$, then, after the execution of $\mathtt{x} := \mathsf{getProperties}\,(\mathsf{e})$, x will be bound to a list containing precisely $\mathrm{X}_i|_{i=1}^{n}$ in an order described by the ord predicate. The remaining axioms are straightforward.

Symbolic Execution for JSIL Control Flow Commands. Our goal is to use symbolic execution to prove the specifications of JSIL procedures. As procedures may call other procedures, we group specifications in *specification environments*, $\mathtt{S} : \mathcal{S}tr \times \mathcal{F}lag \mapsto \mathcal{S}pec$, mapping procedure identifiers and return modes onto specifications. To avoid clutter, we assume in the formalisation of the logic that

each procedure has a single specification for each return mode. Hence, if $\mathtt{S}(m, \mathit{fl}) = S \in \mathcal{S}pec$, then S is the specification of m for the return mode fl. The generalisation to multiple pre- and postconditions is straightforward. We use $\mathsf{post}(S)$ to denote the postcondition of specification S.

We define logic rules for relating the precondition of every command to its postcondition, which at the same time is the precondition of the command that immediately follows its execution. The rules have the form $\mathtt{p}, m, \mathtt{S}, \mathit{fl} \vdash \langle P, j, i\rangle \rightsquigarrow \langle Q, n\rangle$, where $\mathtt{S}$ is the specification environment, P and Q are the pre- and postconditions of the i-th command of procedure m in program $\mathtt{p}$, j is the index of the command from which the symbolic execution reaches i, and n is the index of the command to which symbolic execution goes next (Figure 6.5). We need to keep the previous command j for the symbolic execution of the ϕ-node command. The rules are additionally annotated with the return mode fl, associated with the specification that is currently being verified.

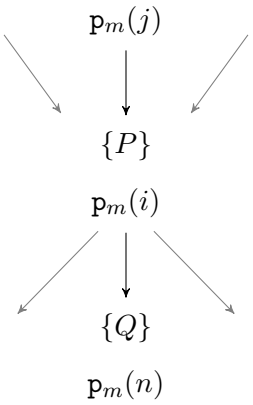


Figure 6.5.: Graphical Representation of the Logic Rules $\mathtt{p}, m, \mathtt{S}, \mathit{fl} \vdash \langle P, j, i\rangle \rightsquigarrow \langle Q, n\rangle$.

We now give the proof rules for symbolically executing control flow commands (Figure 6.6). As $\mathtt{p}$, m, $\mathtt{S}$, and fl do not change during symbolic execution, we leave them implicit.

The Basic Command rule updates the state from P to Q according to its axiom, and transfers control to the following command. The Goto rule jumps to the provided n-th command without changing the state, while the Cond. Goto rules branch to explore both states, where $e = \mathsf{true}$ and $e = \mathsf{false}$. Similarly, the Procedure Call rules branch to consider the case where the procedure returns normally and the case where the procedure throws an exception. The Procedure Call rules update the state from P to Q, where $\{P\}\, m'(\overline{\mathtt{x}})\, \{Q\}$ is the specification of the procedure m'. The parameters which are not provided in the call are bound to $\mathsf{undefined}$. The ϕ-Assignment rule proceeds to the following command after assigning the value corresponding to the k-th predecessor of the current command to the variables $\mathtt{x}_t|_{t=1}^{n}$. Recall the operational semantics of ϕ-node command in §4, Figure 4.5. Frame Rule, Consequence, and Existential Elimination rules are three standard program logic rules that allow the state to be changed while remaining at the same node. The Normal Return rule first checks if the symbolic execution is associated with a nm-mode specification, in which case it further checks if the current symbolic state entails the postcondition of that specification. Note that the Normal Return rule cannot be used during the symbolic execution of an er-mode specification, because the first check would fail. The Error Return rule is analogous.

Basic Command
$$\frac{\mathtt{p}_m(i) = \mathtt{bc} \quad \{P\}\,\mathtt{bc}\,\{Q\} \quad i \notin \{\mathtt{ret}, \mathtt{err}\}}{\langle\, P, j, i \,\rangle \rightsquigarrow \langle\, Q, i+1 \,\rangle}$$

Goto
$$\frac{\mathtt{p}_m(i) = \mathsf{goto}\ n}{\langle\, P, j, i \,\rangle \rightsquigarrow \langle\, P, n \,\rangle}$$

Cond. Goto - True
$$\frac{\mathtt{p}_m(i) = \mathsf{goto}\ [\mathsf{e}]\ n_1,\ n_2}{\langle\, P, j, i \,\rangle \rightsquigarrow \langle\, P * \mathsf{e} \doteq \mathsf{true}, n_1 \,\rangle}$$

Cond. Goto - False
$$\frac{\mathtt{p}_m(i) = \mathsf{goto}\ [\mathsf{e}]\ n_1,\ n_2}{\langle\, P, j, i \,\rangle \rightsquigarrow \langle\, P * \mathsf{e} \doteq \mathsf{false}, n_2 \,\rangle}$$

ϕ-Assignment
$$\frac{\mathtt{p}_m(i) = \mathtt{x}_1, ..., \mathtt{x}_n := \phi(\mathtt{x}_1^1, ..., \mathtt{x}_1^r; ...; \mathtt{x}_n^1, ..., \mathtt{x}_n^r) \quad j \overset{k}{\mapsto}_m i}{\langle\, P, j, i \,\rangle \rightsquigarrow \langle\, P * (\circledast_{t=1}^{n} \mathtt{x}_t \doteq \mathtt{x}_t^k), i+1 \,\rangle}$$

Frame Rule
$$\frac{\langle\, P, j, i \,\rangle \rightsquigarrow \langle\, Q, n \,\rangle \quad i \notin \{\mathtt{ret}, \mathtt{err}\}}{\langle\, P * R, j, i \,\rangle \rightsquigarrow \langle\, Q * R, n \,\rangle}$$

Existential Elimination
$$\frac{\langle\, P, j, i \,\rangle \rightsquigarrow \langle\, Q, n \,\rangle \quad i \notin \{\mathtt{ret}, \mathtt{err}\}}{\langle\, \exists \mathrm{X}.\, P, j, i \,\rangle \rightsquigarrow \langle\, \exists \mathrm{X}.\, Q, n \,\rangle}$$

Consequence
$$\frac{\langle\, P, j, i \,\rangle \rightsquigarrow \langle\, Q, n \,\rangle \quad P' \Rightarrow P \quad Q \Rightarrow Q' \quad i \notin \{\mathtt{ret}, \mathtt{err}\}}{\langle\, P', j, i \,\rangle \rightsquigarrow \langle\, Q', n \,\rangle}$$

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

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

Normal Return
$$\frac{\mathit{fl} = \mathsf{nm} \quad Q \Rightarrow \mathsf{post}(\mathtt{S}(m, \mathsf{nm}))}{\langle\, Q, j, \mathtt{ret} \,\rangle \rightsquigarrow \langle\, Q, \mathtt{ret} \,\rangle}$$

Error Return
$$\frac{\mathit{fl} = \mathsf{er} \quad Q \Rightarrow \mathsf{post}(\mathtt{S}(m, \mathsf{er}))}{\langle\, Q, j, \mathtt{err} \,\rangle \rightsquigarrow \langle\, Q, \mathtt{err} \,\rangle}$$

$j \mapsto_m i$ denotes that j is an immediate predecessor of i.
$j \overset{k}{\mapsto}_m i$ states that j is the k-th element of the list containing all the predecessors of i in chronological order.

Figure 6.6.: Symbolic Execution of Control Flow Commands: $\mathtt{p}, m, \mathtt{S}, \mathit{fl} \vdash \langle\, P, j, i \,\rangle \rightsquigarrow \langle\, Q, n \,\rangle$

6.3. Soundness of JSIL Logic

In this section, we prove the soundness of JSIL Logic. Before we can formulate the soundness theorem, we need to introduce the notion of *proof candidates*.

A proof candidate, $\mathtt{pd} \in \mathcal{D} : \mathcal{S}tr \times \mathcal{F}lag \times \mathbb{N} \rightharpoonup \mathscr{P}(\mathcal{AS}_{\mathtt{JSIL}} \times \mathbb{N})$, maps each command in a procedure to a set of possible preconditions, associating each such precondition with the index of the command that led to it. For instance, if $(P, j) \in \mathtt{pd}(m, \mathit{fl}, i)$, 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 its j-th command. A proof candidate $\mathtt{pd}$ is *well-formed* if and only if: **(1)** the set of preconditions of the first command of every procedure contains only the precondition of the procedure itself; and **(2)** one can symbolically execute every command on all of its possible preconditions.

Definition 6.2 (Well-formed proof candidate)**.** *Given a program* $\mathtt{p} \in \mathtt{P}$ *and a specification environment* $\mathtt{S} \in \mathcal{S}tr \times \mathcal{F}lag \rightharpoonup \mathcal{S}pec$*, a proof candidate* $\mathtt{pd} \in \mathcal{D}$ *is* well-formed *with respect to* $\mathtt{p}$ *and* $\mathtt{S}$*, written* $\mathtt{p}, \mathtt{S} \vdash \mathtt{pd}$*, if and only if, for all procedures* m *in* $\mathtt{p}$*, and index* i*, the following hold:*

1. $\forall \mathrm{fl}, P, Q.\ \mathtt{S}(m, \mathrm{fl}) = \{P\} m(\overline{\mathtt{x}}) \{Q\} \iff \mathtt{pd}(m, \mathrm{fl}, 0) = \{(P, 0)\}$
2. $\forall \mathrm{fl}, P, j.\ (P, j) \in \mathtt{pd}(m, \mathrm{fl}, i) \implies$
$$\big(\forall n.\, i \mapsto_m n \implies \exists Q.\, (Q, i) \in \mathtt{pd}(m, \mathrm{fl}, n) \ \wedge\ \mathtt{p}, m, \mathtt{S}, \mathrm{fl} \vdash \langle\, P, j, i \,\rangle \rightsquigarrow \langle\, Q, n \,\rangle\big)$$

$$\vee \left(i \in \{\texttt{ret}, \texttt{err}\} \implies \texttt{p}, m, \texttt{S}, \texttt{fl} \vdash \langle\, P, j, i \,\rangle \rightsquigarrow \langle\, P, i \,\rangle\right)$$

We prove that JSIL symbolic execution rules are sound with respect to the JSIL operational semantics, that is, we prove that if there is a well-formed proof candidate derivation with respect to a program p and specification environment S, then all specifications in the co-domain of S are valid (Theorem 6.1). To prove this theorem, we prove Frame Property and Soundness for Control Flow Commands (Lemma 2), which uses the Frame Property and Soundness for Basic Commands (Lemma 1).

Both Lemma 1 and Lemma 2 treat Frame Property and Soundness at the same time, in order to avoid duplication, as these proofs are quite similar. On the other hand, this choice requires of us to talk about two frames in the formulation of the lemmas.

Lemma 1 (Frame Property and Soundness for Basic Commands)**.** *For every basic command axiom* $\{P\}\,\texttt{bc}\,\{Q\}$ *it holds that:*

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

We use the notation $\mathcal{L}(h)$ and $\mathcal{L}(H)$ to denote the set of locations present in either concrete or abstract JSIL heaps. Observe the second condition of the lemma, which requires that no locations created during the execution of the command are present in the abstract heaps constituting the frame. This condition is not required for program logics for static languages, but for languages with extensible objects, it is necessary, as we will shortly demonstrate.

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

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

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

- **G1**: $[\![\texttt{bc}]\!]_{\lfloor H \uplus \hat{H}_1 \rfloor, \rho} = (\lfloor H_f \uplus \hat{H}_1 \rfloor, \rho_f, \texttt{v})$
- **G2**: $H_f, \rho_f, \epsilon \models Q$
- **G3**: $\hat{h}_f = \lfloor H_f \uplus \hat{H}_1 \uplus \hat{H}_2 \rfloor$

We proceed by analysing all basic command axioms.

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

  and all hold directly from the definitions and hypotheses, noting that $\llbracket \mathsf{e}_3 \rrbracket_\rho \neq \varnothing$.

- [Var Assignment] We have that $\mathtt{bc} = \mathtt{x} := \mathsf{e}$, $P = \mathsf{emp}$ and $Q = \mathtt{x} \doteq \mathsf{e}$. From the satisfiability of JSIL assertions and **H1**, we obtain that $H = \mathsf{emp}$. From **H3** and the semantics of JSIL basic commands, we obtain that $\hat{h}_f = \lfloor \hat{H}_1 \uplus \hat{H}_2 \rfloor$, $\rho_f = \rho[\mathtt{x} \mapsto \llbracket \mathsf{e} \rrbracket_\rho]$, and $\mathtt{v} = \llbracket \mathsf{e} \rrbracket_\rho$. We choose $H_f = \mathsf{emp}$ as our witness. The goals then become:
  - **G1**: $\llbracket \mathtt{x} := \mathsf{e} \rrbracket_{\lfloor \hat{H}_1 \rfloor, \rho} = (\lfloor \hat{H}_1 \rfloor, \rho[\mathtt{x} \mapsto \llbracket \mathsf{e} \rrbracket_\rho], \llbracket \mathsf{e} \rrbracket_\rho)$
  - **G2**: $\mathsf{emp}, \rho[\mathtt{x} \mapsto \llbracket \mathsf{e} \rrbracket_\rho], \epsilon \models \mathtt{x} \doteq \mathsf{e}$
  - **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, noting that $\mathtt{x}$ is not mentioned in e as we only consider programs in SSA.

- [Object Creation] We have that $\mathtt{bc} = \mathtt{x} := \mathsf{new}\,()$, $P = \mathsf{emp}$ and $Q = (\mathtt{x}, @\mathit{proto}) \mapsto \mathsf{null} * \mathsf{emptyProps}(\mathtt{x} \mid \{ @\mathit{proto} \})$. From the satisfiability of JSIL assertions and **H1**, we obtain that $H = \mathsf{emp}$. From **H3** and the semantics of JSIL basic commands, we obtain that $\hat{h}_f = (l, @\mathit{proto}) \mapsto \mathsf{null} \uplus \lfloor \hat{H}_1 \uplus \hat{H}_2 \rfloor$, $\rho_f = \rho[\mathtt{x} \mapsto l]$, and $\mathtt{v} = l$, for a fresh location l. We know that $l \notin \mathsf{dom}(\hat{H}_1 \uplus \hat{H}_2)$ from **H2**. Here, if we didn't have the hypothesis **H2**, then we would allow $\hat{H}_1 \uplus \hat{H}_2$ to contain empty properties of the object l. However, in that case, since H_f has to capture the entire object at location l in order for **G2** to hold, $H_f \uplus \hat{H}_1 \uplus \hat{H}_2$ would not be well-defined. This is a consequence of having explicit negative information about object properties. We choose $H_f = (l, @\mathit{proto}) \mapsto \mathsf{null} \uplus \left(\biguplus_{p \neq @\mathit{proto}} (l, p) \mapsto \varnothing \right)$ as our witness. Note that then $\lfloor H_f \rfloor = (l, @\mathit{proto}) \mapsto \mathsf{null}$. The goals become:
  - **G1**: $\llbracket \mathsf{new}\,() \rrbracket_{\lfloor \hat{H}_1 \rfloor, \rho} = (\lfloor (l, @\mathit{proto}) \mapsto \mathsf{null} \uplus H_1 \rfloor, \rho[\mathtt{x} \mapsto l], l)$
  - **G2**: $H_f, \rho[\mathtt{x} \mapsto l], \epsilon \models (\mathtt{x}, @\mathit{proto}) \mapsto \mathsf{null} * \mathsf{emptyProps}(\mathtt{x} \mid \{ @\mathit{proto} \})$
  - **G3**: $(l, @\mathit{proto}) \mapsto \mathsf{null} \uplus \lfloor \hat{H}_1 \uplus \hat{H}_2 \rfloor = \lfloor H_f \uplus \hat{H}_1 \uplus \hat{H}_2 \rfloor$

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

- [Property Deletion] We have that $\mathtt{bc} = \mathsf{delete}\,(\mathsf{e}_1, \mathsf{e}_2)$, $P = (\mathsf{e}_1, \mathsf{e}_2) \mapsto \mathrm{X} * \mathrm{X} \not\doteq \varnothing * \mathsf{e}_2 \not\doteq @\mathit{proto}$ and $Q = (\mathsf{e}_1, \mathsf{e}_2) \mapsto \varnothing$. From the satisfiability of JSIL assertions and **H1**, we obtain that $H =$

$(\llbracket \mathtt{e}_1 \rrbracket_\rho, \llbracket \mathtt{e}_2 \rrbracket_\rho) \mapsto \epsilon(\mathrm{X})$, where $\epsilon(\mathrm{X}) \neq \varnothing$ and $\llbracket \mathtt{e}_2 \rrbracket_\rho \neq @proto$. Note that $\lfloor H \rfloor = H$ since $\epsilon(\mathrm{X}) \neq \varnothing$. 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 $\mathtt{v} = \mathsf{true}$. We choose $H_f = (\llbracket \mathtt{e}_1 \rrbracket_\rho, \llbracket \mathtt{e}_2 \rrbracket_\rho) \mapsto \varnothing$ as our witness, noting that $\lfloor H_f \rfloor = \mathsf{emp}$. The goals become:

- **G1**: $\llbracket \mathtt{bc} \rrbracket_{\lfloor (\llbracket \mathtt{e}_1 \rrbracket_\rho, \llbracket \mathtt{e}_2 \rrbracket_\rho) \mapsto \epsilon(\mathrm{X}) \uplus \hat{H}_1 \rfloor, \rho} = (\lfloor \hat{H}_1 \rfloor, \rho, \mathsf{true})$
- **G2**: $(\llbracket \mathtt{e}_1 \rrbracket_\rho, \llbracket \mathtt{e}_2 \rrbracket_\rho) \mapsto \varnothing, \rho, \epsilon \models (\mathtt{e}_1, \mathtt{e}_2) \mapsto \varnothing$
- **G3**: $\lfloor \hat{H}_1 \uplus \hat{H}_2 \rfloor = \lfloor (\llbracket \mathtt{e}_1 \rrbracket_\rho, \llbracket \mathtt{e}_2 \rrbracket_\rho) \mapsto \varnothing \uplus \hat{H}_1 \uplus \hat{H}_2 \rfloor$

and all follow directly from the definitions and hypotheses, noting that the disjoint union in **G3** is well-defined due to **H3**.

- [Property Access] We have that $\mathtt{bc} = \mathtt{x} := [\mathtt{e}_1, \mathtt{e}_2]$, $P = (\mathtt{e}_1, \mathtt{e}_2) \mapsto \mathrm{X} * \mathrm{X} \not\doteq \varnothing$ and $Q = (\mathtt{e}_1, \mathtt{e}_2) \mapsto \mathrm{X} * \mathrm{X} \not\doteq \varnothing * \mathtt{x} \doteq \mathrm{X}$. From the satisfiability of JSIL assertions and **H1**, we obtain that $H = (\llbracket \mathtt{e}_1 \rrbracket_\rho, \llbracket \mathtt{e}_2 \rrbracket_\rho) \mapsto \epsilon(\mathrm{X})$, where $\epsilon(\mathrm{X}) \neq \varnothing$. Note that $\lfloor H \rfloor = H$ since $\epsilon(\mathrm{X}) \neq \varnothing$. From **H3** and the semantics of JSIL basic commands, we obtain that $\hat{h}_f = \lfloor H \uplus \hat{H}_1 \uplus \hat{H}_2 \rfloor$, $\rho_f = \rho[\mathtt{x} \mapsto \epsilon(\mathrm{X})]$, and $\mathtt{v} = \epsilon(\mathrm{X})$. We choose $H_f = H = (\llbracket \mathtt{e}_1 \rrbracket_\rho, \llbracket \mathtt{e}_2 \rrbracket_\rho) \mapsto \epsilon(\mathrm{X})$ as our witness. The goals then become:
  - **G1**: $\llbracket \mathtt{x} := [\mathtt{e}_1, \mathtt{e}_2] \rrbracket_{\lfloor (\llbracket \mathtt{e}_1 \rrbracket_\rho, \llbracket \mathtt{e}_2 \rrbracket_\rho) \mapsto \epsilon(\mathrm{X}) \uplus \hat{H}_1 \rfloor, \rho} = (\lfloor (\llbracket \mathtt{e}_1 \rrbracket_\rho, \llbracket \mathtt{e}_2 \rrbracket_\rho) \mapsto \epsilon(\mathrm{X}) \uplus \hat{H}_1 \rfloor, \rho[\mathtt{x} \mapsto \epsilon(\mathrm{X})], \epsilon(\mathrm{X}))$
  - **G2**: $(\llbracket \mathtt{e}_1 \rrbracket_\rho, \llbracket \mathtt{e}_2 \rrbracket_\rho) \mapsto \epsilon(\mathrm{X}), \rho[\mathtt{x} \mapsto \epsilon(\mathrm{X})], \epsilon \models (\mathtt{e}_1, \mathtt{e}_2) \mapsto \mathrm{X} * \mathrm{X} \not\doteq \varnothing * \mathtt{x} \doteq \mathrm{X}$
  - **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 $\epsilon(\mathrm{X}) \neq \varnothing$.

- [Member Check - True] We have that $\mathtt{bc} = \mathtt{x} := \mathsf{hasProperty}\,(\mathtt{e}_1, \mathtt{e}_2)$, $P = (\mathtt{e}_1, \mathtt{e}_2) \mapsto \mathrm{X} * \mathrm{X} \not\doteq \varnothing$ and $Q = (\mathtt{e}_1, \mathtt{e}_2) \mapsto \mathrm{X} * \mathrm{X} \not\doteq \varnothing * \mathtt{x} \doteq \mathsf{true}$. From the satisfiability of JSIL assertions and **H1**, we obtain that $H = (\llbracket \mathtt{e}_1 \rrbracket_\rho, \llbracket \mathtt{e}_2 \rrbracket_\rho) \mapsto \epsilon(\mathrm{X})$, where $\epsilon(\mathrm{X}) \neq \varnothing$. Note that $\lfloor H \rfloor = H$ since $\epsilon(\mathrm{X}) \neq \varnothing$. From **H3** and the semantics of JSIL basic commands, we obtain that $\hat{h}_f = \lfloor H \uplus \hat{H}_1 \uplus \hat{H}_2 \rfloor$, $\rho_f = \rho[\mathtt{x} \mapsto \mathsf{true}]$, and $\mathtt{v} = \mathsf{true}$. We choose $H_f = H = (\llbracket \mathtt{e}_1 \rrbracket_\rho, \llbracket \mathtt{e}_2 \rrbracket_\rho) \mapsto \epsilon(\mathrm{X})$ as our witness. The goals then become:
  - **G1**: $\llbracket \mathtt{x} := \mathsf{hasProperty}\,(\mathtt{e}_1, \mathtt{e}_2) \rrbracket_{\lfloor (\llbracket \mathtt{e}_1 \rrbracket_\rho, \llbracket \mathtt{e}_2 \rrbracket_\rho) \mapsto \epsilon(\mathrm{X}) \uplus \hat{H}_1 \rfloor, \rho} =$
    $(\lfloor (\llbracket \mathtt{e}_1 \rrbracket_\rho, \llbracket \mathtt{e}_2 \rrbracket_\rho) \mapsto \epsilon(\mathrm{X}) \uplus \hat{H}_1 \rfloor, \rho[\mathtt{x} \mapsto \mathsf{true}], \mathsf{true})$
  - **G2**: $(\llbracket \mathtt{e}_1 \rrbracket_\rho, \llbracket \mathtt{e}_2 \rrbracket_\rho) \mapsto \epsilon(\mathrm{X}), \rho[\mathtt{x} \mapsto \mathsf{true}], \epsilon \models (\mathtt{e}_1, \mathtt{e}_2) \mapsto \mathrm{X} * \mathrm{X} \not\doteq \varnothing * \mathtt{x} \doteq \mathsf{true}$
  - **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 $\epsilon(\mathrm{X}) \neq \varnothing$.

- [Member Check - False] We have that $\mathtt{bc} = \mathtt{x} := \mathsf{hasProperty}\,(\mathtt{e}_1, \mathtt{e}_2)$, $P = (\mathtt{e}_1, \mathtt{e}_2) \mapsto \varnothing$ and $Q = (\mathtt{e}_1, \mathtt{e}_2) \mapsto \varnothing * \mathtt{x} \doteq \mathsf{false}$. From the satisfiability of JSIL assertions and **H1**, we obtain that $H = (\llbracket \mathtt{e}_1 \rrbracket_\rho, \llbracket \mathtt{e}_2 \rrbracket_\rho) \mapsto \varnothing$. Note that $\lfloor H \rfloor = \mathsf{emp}$. From **H3** and the semantics of JSIL basic commands, we obtain that $\hat{h}_f = \lfloor H \uplus \hat{H}_1 \uplus \hat{H}_2 \rfloor$, $\rho_f = \rho[\mathtt{x} \mapsto \mathsf{false}]$, and $\mathtt{v} = \mathsf{false}$, noting that $\hat{H}_1 \uplus \hat{H}_2$ cannot contain the cell $(\llbracket \mathtt{e}_1 \rrbracket_\rho, \llbracket \mathtt{e}_2 \rrbracket_\rho)$, since the disjoint union $H \uplus \hat{H}_1 \uplus \hat{H}_2$ is well-defined. We choose $H_f = H = (\llbracket \mathtt{e}_1 \rrbracket_\rho, \llbracket \mathtt{e}_2 \rrbracket_\rho) \mapsto \varnothing$ as our witness. The goals then become:
  - **G1**: $\llbracket \mathtt{x} := \mathsf{hasProperty}\,(\mathtt{e}_1, \mathtt{e}_2) \rrbracket_{\lfloor \hat{H}_1 \rfloor, \rho} = (\lfloor \hat{H}_1 \rfloor, \rho[\mathtt{x} \mapsto \mathsf{false}], \mathsf{false})$

– **G2**: $([\![\mathsf{e}_1]\!]_\rho, [\![\mathsf{e}_2]\!]_\rho) \mapsto \varnothing, \rho[\mathtt{x} \mapsto \mathsf{false}], \epsilon \models (\mathsf{e}_1, \mathsf{e}_2) \mapsto \varnothing * \mathtt{x} \doteq \mathsf{false}$

– **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 $\hat{H}_1$ cannot contain the cell $([\![\mathsf{e}_1]\!]_\rho, [\![\mathsf{e}_2]\!]_\rho)$.

- [Get Properties] We have that $\mathtt{bc} = \mathtt{x} := \mathsf{getProperties}\,(\mathsf{e})$,

$$P = (\circledast_{i=1}^{n}(\mathsf{e}, \mathrm{X}_i) \mapsto \mathrm{Y}_i) * \mathsf{emptyProps}(\mathsf{e} \mid \{\, \mathrm{X}_i|_{i=1}^{n} \,\}) * (\circledast_{i=1}^{n} \mathrm{Y}_i \neq \varnothing)$$

and $Q = P * \mathtt{x} \doteq \mathrm{X}_i|_{i=1}^{n} * \mathsf{ord}\,(\mathtt{x}) \doteq \mathsf{true}$. From the satisfiability of JSIL assertions and **H1**, we obtain that

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

where $\epsilon(\mathrm{Y}_i) \neq \varnothing|_{i=1}^{n}$. Therefore, we conclude that:

$$\lfloor H \rfloor = \left(\biguplus_{i=1}^{n} ([\![\mathsf{e}]\!]_\rho, \epsilon(\mathrm{X}_i)) \mapsto \epsilon(\mathrm{Y}_i) \right).$$

From **H3**, the semantics of JSIL basic commands, we obtain that $\hat{h}_f = \lfloor H \uplus \hat{H}_1 \uplus \hat{H}_2 \rfloor$, $\rho_f = \rho[\mathtt{x} \mapsto \epsilon(\mathrm{X}_i|_{i=1}^{n})]$, and $\mathtt{v} = \epsilon(\mathrm{X}_i|_{i=1}^{n})$, where $\mathcal{O}rd(\epsilon(\mathrm{X}_i|_{i=1}^{n}))$. 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**: $[\![\mathtt{x} := \mathsf{getProperties}\,(\mathsf{e})]\!]_{\lfloor H \uplus \hat{H}_1 \rfloor, \rho} = (\lfloor H \uplus \hat{H}_1 \rfloor, \rho[\mathtt{x} \mapsto \epsilon(\mathrm{X}_i|_{i=1}^{n})], \epsilon(\mathrm{X}_i|_{i=1}^{n}))$

– **G2**: $H, \rho[\mathtt{x} \mapsto \epsilon(\mathrm{X}_i|_{i=1}^{n})], \epsilon \models P * \mathtt{x} \doteq \mathrm{X}_i|_{i=1}^{n} * \mathsf{ord}\,(\mathtt{x}) \doteq \mathsf{true}$

– **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.

□

Lemma 2 (Frame Property and Soundness for Control Flow Commands)**.** *For any derivation* $\mathtt{pd} \in \mathcal{D}$*, program* $\mathtt{p} \in \mathtt{P}$*, specification environment* $\mathtt{S} \in \mathcal{S}tr \times \mathcal{F}lag \rightharpoonup \mathcal{S}pec$*, return mode flag* fl*, abstract heaps* $H, \hat{H}_1, \hat{H}_2$*, store* $\rho \in \mathcal{S}to$*, logical environment* ϵ*, procedure identifier* m*, and command labels* i *and* k *such that:*

- ***H1****:* $\mathtt{p}, \mathtt{S} \vdash \mathtt{pd}$
- ***H2****:* $(P, k) \in \mathtt{pd}(m, \mathrm{fl}, i)$
- ***H3****:* $H, \rho, \epsilon \models P$
- ***H4****:* $\mathtt{p} \vdash \langle \lfloor H \uplus \hat{H}_1 \uplus \hat{H}_2 \rfloor, \rho, k, i \rangle \Downarrow_m \langle h_f, \rho_f, o \rangle$
- ***H5****:* $(\mathcal{L}(h_f) \setminus \mathcal{L}(\lfloor H \rfloor)) \cap \mathcal{L}(\hat{H}_1 \uplus \hat{H}_2) = \emptyset$

it follows that there is an abstract heap H_f *and a value* $\mathtt{v}$ *such that:*

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

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

[Basic Command] It follows that $\mathtt{p}_m(i) = \mathtt{bc}$ for a given basic command $\mathtt{bc}$. We conclude, using **H4** and the semantics of JSIL, that there is a heap h', a store ρ', and value $\mathtt{v}'$, such that:

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

Using **H1** and **H2**, we conclude that there is an assertion Q such that: $(Q, i) \in \mathtt{pd}(m, fl, i+1)$ (**I3**), and $\langle\, P, k, i \,\rangle \rightsquigarrow \langle\, Q, i+1 \,\rangle$ (**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 $\lfloor H' \uplus \hat{H}_1 \uplus \hat{H}_2 \rfloor = h'$. We prove that such an abstract heap exists by induction on the derivation of $\langle\, P, k, i \,\rangle \rightsquigarrow \langle\, Q, i+1 \,\rangle$. More concretely, we have to prove that, given $[\![\mathtt{bc}]\!]_{\lfloor H \uplus \hat{H}_1 \uplus \hat{H}_2 \rfloor, \rho} = (h', \rho', \mathtt{v}')$, $H, \rho, \epsilon \models P$, and $\langle\, P, k, i \,\rangle \rightsquigarrow \langle\, Q, i+1 \,\rangle$, there must exist an abstract heap H' such that $\lfloor H' \uplus \hat{H}_1 \uplus \hat{H}_2 \rfloor = h'$ (*goal 1*), $H', \rho', \epsilon \models Q$ (*goal 2*), and $[\![\mathtt{bc}]\!]_{\lfloor H \uplus \hat{H}_1 \rfloor, \rho} = (\lfloor H' \uplus \hat{H}_1 \rfloor, \rho', \mathtt{v}')$ (*goal 3*). In the following, we proceed by case analysis on the last rule applied to obtain $\langle\, P, k, i \,\rangle \rightsquigarrow \langle\, Q, i+1 \,\rangle$.

- [Basic Command] We conclude that: $\{P\}\, \mathtt{bc}\, \{Q\}$ (**C1.1**). Applying the Frame Property and Soundness for Basic Commands (Lemma 1) to **H3**, **H5**, and **I1**, we conclude that there is an abstract heap H' such that $[\![\mathtt{bc}]\!]_{\lfloor H \uplus \hat{H}_1 \rfloor, \rho} = (\lfloor H' \uplus \hat{H}_1 \rfloor, \rho', \mathtt{v}')$, $H', \rho', \epsilon \models Q$, and $h' = \lfloor H' \uplus \hat{H}_1 \uplus \hat{H}_2 \rfloor$ (*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 $\langle\, P', k, i \,\rangle \rightsquigarrow \langle\, Q', i+1 \,\rangle$ (**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: $[\![\mathtt{bc}]\!]_{\lfloor (H'_p \uplus H_r) \uplus \hat{H}_1 \uplus \hat{H}_2 \rfloor, \rho} = (h', \rho', \mathtt{v}')$ (**C2.7**). Using the associativity of $\uplus$, we get from **C2.7**, that $[\![\mathtt{bc}]\!]_{\lfloor H'_p \uplus (H_r \uplus \hat{H}_1) \uplus \hat{H}_2 \rfloor, \rho} = (h', \rho', \mathtt{v}')$ (**C2.8**). Applying the inner induction hypothesis to **C2.8**, **C2.5**, and **C2.3**, we conclude that there is an abstract heap H'_q such that: $\lfloor H'_q \uplus (H_r \uplus \hat{H}_1) \uplus \hat{H}_2 \rfloor = h'$ (**C2.9**), $H'_q, \rho', \epsilon \models Q'$ (**C2.10**), and $[\![\mathtt{bc}]\!]_{\lfloor H'_p \uplus (H_r \uplus \hat{H}_1) \rfloor, \rho} = (\lfloor H'_q \uplus (H_r \uplus \hat{H}_1) \rfloor, \rho', \mathtt{v}')$ (**C2.11**). We now claim that $H'_q \uplus H_r$ is our witness, having to show that $\lfloor (H'_q \uplus H_r) \uplus \hat{H}_1 \uplus \hat{H}_2 \rfloor = h'$, $H'_q \uplus H_r, \rho', \epsilon \models Q$, and $[\![\mathtt{bc}]\!]_{\lfloor H \uplus \hat{H}_1 \rfloor, \rho} = (\lfloor (H'_q \uplus H_r) \uplus \hat{H}_1 \rfloor, \rho', \mathtt{v}')$. Given the associativity of $\uplus$, we conclude, from **C2.9**, that $\lfloor (H'_q \uplus H_r) \uplus \hat{H}_1 \uplus \hat{H}_2 \rfloor = h'$ (*goal 1*) and, from **C2.11**, that $[\![\mathtt{bc}]\!]_{\lfloor (H'_p \uplus H_r) \uplus \hat{H}_1 \rfloor, \rho} = (\lfloor (H'_q \uplus H_r) \uplus \hat{H}_1 \rfloor, \rho', \mathtt{v}')$ (*goal 3*). From **C2.2**, **C2.6**, and **C2.10**, it follows that $H'_q \uplus H_r, \rho', \epsilon \models Q$ (*goal 2*). Notice that $H_r, \rho, \epsilon \models R$, and that R does not talk about the variable modified by the command $\mathtt{p}_m(i)$ because of SSA, hence, $H_r, \rho', \epsilon \models R$.

- [Consequence] We conclude that there are two assertions P' and Q', such that: $P \Rightarrow P'$ (**C3.1**), $Q' \Rightarrow Q$ (**C3.2**), and $\langle\, P', k, i \,\rangle \rightsquigarrow \langle\, Q', i+1 \,\rangle$ (**C3.3**). From **H3** and **C3.1**, we conclude that $H, \rho, \epsilon \models P'$ (**C3.4**). Applying the inner induction hypothesis to **I1**, **C3.4**, and **C3.3**, it follows that there is an abstract heap H' such that: $\lfloor H' \uplus \hat{H}_1 \uplus \hat{H}_2 \rfloor = h'$ (*goal 1*), $H', \rho', \epsilon \models Q'$ (**C3.5**), and $[\![\mathtt{bc}]\!]_{\lfloor H \uplus \hat{H}_1 \rfloor, \rho} = (\lfloor H' \uplus \hat{H}_1 \rfloor, \rho', \mathtt{v}')$ (*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: $P = \exists \mathrm{X}.\, P'$ (**C3.1**), $Q = \exists \mathrm{X}.\, Q'$ (**C3.2**), and $\langle\, P', k, i \,\rangle \rightsquigarrow \langle\, Q', i+1 \,\rangle$ (**C3.3**). From **H3** and **C3.1**, it follows that there is a value V such that $H, \rho, \epsilon[\mathrm{X} \mapsto \mathrm{V}] \models P'$ (**C3.4**). Applying the inner induction hypothesis to **I1**, **C3.4**, and **C3.3**, we conclude that there is an abstract heap H' such that: $\lfloor H' \uplus \hat{H}_1 \uplus \hat{H}_2 \rfloor = h'$ (*goal 1*), $H', \rho', \epsilon[\mathrm{X} \mapsto \mathrm{V}] \models Q'$ (**C3.5**), and $[\![\mathtt{bc}]\!]_{\lfloor H \uplus \hat{H}_1 \rfloor, \rho} = (\lfloor H' \uplus \hat{H}_1 \rfloor, \rho', \mathtt{v}')$ (*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 $\langle\, P, k, i \,\rangle \rightsquigarrow \langle\, Q, i+1 \,\rangle$ because $\mathtt{p}_m(i)$ 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 $\lfloor H' \uplus \hat{H}_1 \uplus \hat{H}_2 \rfloor = h'$ (**I5**), $H', \rho', \epsilon \models Q$ (**I6**), and $[\![\mathtt{bc}]\!]_{\lfloor H \uplus \hat{H}_1 \rfloor, \rho} = (\lfloor H' \uplus \hat{H}_1 \rfloor, \rho', \mathtt{v}')$ (**I7**). Applying the induction hypothesis to **H1**, **I3**, **I6**, and **I2**, we have that there is an abstract heap H_f and value $\mathtt{v}$ such that: $o = \mathit{fl}\langle \mathtt{v} \rangle$ (**G1**), $\mathtt{p} \vdash \langle \lfloor H' \uplus \hat{H}_1 \rfloor, \rho', i, i+1 \rangle \Downarrow_m \langle \lfloor H_f \uplus \hat{H}_1 \rfloor, \rho_f, o \rangle$ (**I8**), $H_f, \rho_f, \epsilon \models \mathsf{post}(\mathtt{S}(m, \mathit{fl}))$ (**G3**), and $h_f = \lfloor H_f \uplus \hat{H}_1 \uplus \hat{H}_2 \rfloor$ (**G4**). Recalling that $\mathtt{p}_m(i) = \mathtt{bc}$, it follows from **I7** and **I8** that $\mathtt{p} \vdash \langle \lfloor H \uplus \hat{H}_1 \rfloor, \rho, k, i \rangle \Downarrow_m \langle \lfloor H_f \uplus \hat{H}_1 \rfloor, \rho_f, o \rangle$ (**G2**).

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

$$\mathtt{p} \vdash \langle \lfloor H \uplus \hat{H}_1 \uplus \hat{H}_2 \rfloor, \rho, i, j \rangle \Downarrow_m \langle h_f, \rho_f, o \rangle \ (\mathbf{I1})$$

Using **H1** and **H2**, we conclude that there is an assertion Q such that: $(Q, i) \in \mathtt{pd}(m, \mathit{fl}, j)$ (**I2**) and $\langle\, P, k, i \,\rangle \rightsquigarrow \langle\, Q, j \,\rangle$ (**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 $\langle\, P, k, i \,\rangle \rightsquigarrow \langle\, Q, j \,\rangle$. More concretely, given that $H, \rho, \epsilon \models P$ and $\langle\, P, k, i \,\rangle \rightsquigarrow \langle\, Q, j \,\rangle$, we need to show that $H, \rho, \epsilon \models Q$ (*goal*).

- [Goto] We conclude that: $\langle\, P, k, i \,\rangle \rightsquigarrow \langle\, P, j \,\rangle$ (**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 $\langle\, P', k, i \,\rangle \rightsquigarrow \langle\, Q', j \,\rangle$ (**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 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 $\langle\, P', k, i \,\rangle \rightsquigarrow \langle\, Q', j \,\rangle$ (**C3.3**). From **H3** and **C3.1**, we conclude that $H, \rho, \epsilon \models P'$ (**C3.4**). Applying the 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: $P = \exists \mathrm{X}.\, P'$ (**C3.1**), $Q = \exists \mathrm{X}.\, Q'$ (**C3.2**), and $\langle\, P', k, i \,\rangle \rightsquigarrow \langle\, Q', j \,\rangle$ (**C3.3**). From **H3** and **C3.1**, it follows that there is a value V such that $H, \rho, \epsilon[\mathrm{X} \mapsto \mathrm{V}] \models P'$ (**C3.4**). Applying the inner induction hypothesis to **C3.4** and **C3.3**, we conclude that $H, \rho, \epsilon[\mathrm{X} \mapsto \mathrm{V}] \models Q'$ (**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 $\langle\, P, k, i \,\rangle \rightsquigarrow \langle\, Q, j \,\rangle$ because $\mathtt{p}_m(i) = \mathsf{goto}\ j$.

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 $\mathtt{v}$ such that: $o = \mathit{fl}\langle \mathtt{v} \rangle$ (**G1**), $\mathtt{p} \vdash \langle \lfloor H \uplus \hat{H}_1 \rfloor, \rho, i, j \rangle \Downarrow_m \langle \lfloor H_f \uplus \hat{H}_1 \rfloor, \rho_f, o \rangle$ (**I5**), $H_f, \rho_f, \epsilon \models \mathsf{post}(\mathtt{S}(m, \mathit{fl}))$ (**G3**), and $h_f = \lfloor H_f \uplus \hat{H}_1 \uplus \hat{H}_2 \rfloor$ (**G4**). Recalling that $\mathtt{p}_m(i) = \mathsf{goto}\ j$, it follows from **I5** that $\mathtt{p} \vdash \langle \lfloor H \uplus \hat{H}_1 \rfloor, \rho, k, i \rangle \Downarrow_m \langle \lfloor H_f \uplus \hat{H}_1 \rfloor, \rho_f, o \rangle$ (**G2**).

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

$$[\![\mathtt{e}]\!]_\rho = \mathsf{true}\ (\mathbf{I1}) \qquad \mathtt{p} \vdash \langle \lfloor H \uplus \hat{H}_1 \uplus \hat{H}_2 \rfloor, \rho, i, j_1 \rangle \Downarrow_m \langle h_f, \rho_f, o \rangle\ (\mathbf{I2})$$

Using **H1** and **H2**, we conclude that there is an assertion Q such that: $(Q, i) \in \mathtt{pd}(m, \mathit{fl}, j_1)$ (**I3**), and $\langle\, P, k, i \,\rangle \rightsquigarrow \langle\, Q, j_1 \,\rangle$ (**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 $\langle\, P, k, i \,\rangle \rightsquigarrow \langle\, Q, j_1 \,\rangle$. More concretely, given that $H, \rho, \epsilon \models P$ and $\langle\, P, k, i \,\rangle \rightsquigarrow \langle\, Q, j_1 \,\rangle$, we need to show that $H, \rho, \epsilon \models Q$ (*goal*).

- [Conditional Goto - True] We conclude that: $\langle\, P, k, i \,\rangle \rightsquigarrow \langle\, P \ \wedge\ \mathtt{e} = \mathsf{true}, j_1 \,\rangle$. Noting that $\mathtt{e}$ does not contain any logical variables, we conclude that: $[\![\mathtt{e}]\!]_\rho = [\![\mathtt{e}]\!]^\epsilon_\rho$ (**C1.1**). From **I1** and **C1.1**, it follows that $[\![\mathtt{e}]\!]^\epsilon_\rho = \mathsf{true}$ (**C1.2**). From **H3** and **C1.2**, we conclude that $H, \rho, \epsilon \models (P \wedge \mathtt{e} = \mathsf{true})$ (*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 $\langle\, P, k, i \,\rangle \rightsquigarrow \langle\, Q, j_1 \,\rangle$ because $\mathtt{p}_m(i) = \mathsf{goto}\ [\mathtt{e}]\ j_1,\ j_2$.

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 $\mathtt{v}$ such that: $o = \mathit{fl}\langle \mathtt{v} \rangle$ (**G1**), $\mathtt{p} \vdash \langle \lfloor H \uplus \hat{H}_1 \rfloor, \rho, i, j_1 \rangle \Downarrow_m \langle \lfloor H_f \uplus \hat{H}_1 \rfloor, \rho_f, o \rangle$ (**I6**), $H_f, \rho_f, \epsilon \models \mathsf{post}(\mathtt{S}(m, \mathit{fl}))$ (**G3**), and $h_f =$

$\lfloor H_f \uplus \hat{H}_1 \uplus \hat{H}_2 \rfloor$ (**G4**). Recalling that $\mathtt{p}_m(i) = \mathsf{goto}\ [\mathtt{e}]\ j_1,\ j_2$, it follows from **I1** and **I6** that $\mathtt{p} \vdash \langle \lfloor H \uplus \hat{H}_1 \rfloor, \rho, k, i \rangle \Downarrow_m \langle \lfloor H_f \uplus \hat{H}_1 \rfloor, \rho_f, o \rangle$ (**G2**).

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

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

Using **H1** and **H2**, we conclude that there is an assertion Q such that: $(Q, i) \in \mathtt{pd}(m, \mathit{fl}, i+1)$ (**I7**) and $\langle\, P, k, i \,\rangle \rightsquigarrow \langle\, Q, i+1 \,\rangle$ (**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: $H', \rho[\mathtt{x} \mapsto \mathtt{v}'], \epsilon \models Q$, and $\lfloor H' \uplus \hat{H}_1 \uplus \hat{H}_2 \rfloor = h'$. We prove that such an abstract heap exists by induction on the derivation of $\langle\, P, k, i \,\rangle \rightsquigarrow \langle\, Q, i+1 \,\rangle$. More concretely, we have to prove that, given $\mathtt{p} \vdash \langle \lfloor H \uplus \hat{H}_1 \uplus \hat{H}_2 \rfloor, \emptyset[\mathtt{y}_i \mapsto \mathtt{v}_i|_{i=1}^{n_2}], 0, 0 \rangle \Downarrow_{m'} \langle h', \rho', \mathtt{nm}\langle \mathtt{v}' \rangle \rangle$, $H, \rho, \epsilon \models P$, and $\langle\, P, k, i \,\rangle \rightsquigarrow \langle\, Q, i+1 \,\rangle$, there must exist an abstract heap H' such that $\lfloor H' \uplus \hat{H}_1 \uplus \hat{H}_2 \rfloor = h'$ (*goal 1*), $H', \rho[\mathtt{x} \mapsto \mathtt{v}'], \epsilon \models Q$ (*goal 2*), and $\mathtt{p} \vdash \langle \lfloor H \uplus \hat{H}_1 \rfloor, \emptyset[\mathtt{y}_i \mapsto \mathtt{v}_i|_{i=1}^{n_2}], 0, 0 \rangle \Downarrow_{m'} \langle \lfloor H' \uplus \hat{H}_1 \rfloor, \rho', \mathtt{nm}\langle \mathtt{v}' \rangle \rangle$ (*goal 3*). In the following, we proceed by case analysis on the last rule applied to obtain $\langle\, P, k, i \,\rangle \rightsquigarrow \langle\, Q, i+1 \,\rangle$.

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

  Noting that all the specs in $\mathtt{S}$ are well-formed, we conclude, from **C1.10**, that $\mathsf{vars}(P') \cup \mathsf{vars}(Q') \cup \mathsf{vars}(\mathtt{e}') \subseteq \{\mathtt{y}_1, ..., \mathtt{y}_{n_2}\}$ (**C1.11**). Applying the Substitution Lemma for Assertions (Lemma 20) to **H3**, **C1.11**, and **C1.8**, we conclude that $H, \emptyset[\mathtt{y}_i \mapsto [\![\mathtt{e}_i]\!]_\rho^\epsilon|_{i=1}^{n_2}], \epsilon \models P'$ (**C1.12**). For $1 \leq i \leq n_1$, it holds that $[\![\mathtt{e}_i]\!]_\rho^\epsilon = [\![\mathtt{e}_i]\!]_\rho$ (**C1.13**), because $\mathtt{e}_i$ does not contain logical variables. For $n_1 < i \leq n_2$, it holds that $[\![\mathtt{e}_i]\!]_\rho^\epsilon = \mathsf{undefined}$ (**C1.14**), because $\mathtt{e}_i = \mathsf{undefined}$. From **I3**, **I4**, **C1.13**, and **C1.14**, it follows that $\emptyset[\mathtt{y}_i \mapsto \mathtt{v}_i|_{i=1}^{n_2}] = \emptyset[\mathtt{y}_i \mapsto [\![\mathtt{e}_i]\!]_\rho^\epsilon|_{i=1}^{n_2}]$ (**C1.15**). From **C1.12** and **C1.15**, we have that $H, \emptyset[\mathtt{y}_i \mapsto \mathtt{v}_i|_{i=1}^{n_2}], \epsilon \models P'$ (**C1.16**). From **H1** and **C1.10**, we conclude that $(P', 0) \in \mathtt{pd}(m', \mathtt{nm}, 0)$ (**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': $\mathtt{p} \vdash \langle \lfloor H \uplus \hat{H}_1 \rfloor, \emptyset[\mathtt{y}_i \mapsto \mathtt{v}_i|_{i=1}^{n_2}], 0, 0 \rangle \Downarrow_{m'} \langle \lfloor H' \uplus \hat{H}_1 \rfloor, \rho', \mathtt{nm}\langle \mathtt{v}' \rangle \rangle$ (*goal 3*), $H', \rho', \epsilon \models \mathsf{post}(\mathtt{S}(m', \mathtt{nm}))$ (**C1.18**), and $h' = \lfloor H' \uplus \hat{H}_1 \uplus \hat{H}_2 \rfloor$ (*goal 1*). From **C1.10** and **C1.18**, it follows that $H', \rho', \epsilon \models Q' * \mathtt{xret} \doteq \mathtt{e}'$ (**C1.19**). Since we only consider programs in SSA, we conclude, from *goal 3*, that $\rho' \geq \emptyset[\mathtt{y}_i \mapsto \mathtt{v}_i|_{i=1}^{n_2}]$ (**C1.20**). From **C1.11**, **C1.15**, **C1.19**, and **C1.20**, we have that $H', \emptyset[\mathtt{y}_i \mapsto [\![\mathtt{e}_i]\!]_\rho^\epsilon|_{i=1}^{n_2}], \epsilon \models Q'$ (**C1.21**). Applying the Substitution Lemma for Assertions (Lemma 20) to **C1.11** and **C1.21**, we conclude that $H', \rho, \epsilon \models Q'[\mathtt{e}_i/\mathtt{y}_i|_{i=1}^{n_2}]$ (**C1.22**). Applying Lemma 21 to **I5**, it follows that $\rho'(\mathtt{xret}) = \mathtt{v}'$,

from which we have, using **C1.19**, that $[\![\mathsf{e}']\!]^{\epsilon}_{\rho'} = \mathtt{v}'$ (**C1.23**). From **C1.11**, **C1.15**, **C1.20**, and **C1.23**, we have that $[\![\mathsf{e}']\!]^{\epsilon}_{\emptyset[\mathtt{y}_i \mapsto [\![\mathsf{e}_i]\!]^{\epsilon}_{\rho}|_{i=1}^{n_2}]} = \mathtt{v}'$ (**C1.24**). Applying the Substitution Lemma for Logical Expressions (Lemma 19) to **C1.24**, we conclude that $[\![\mathsf{e}'[\mathsf{e}_i/\mathtt{y}_i|_{i=1}^{n_2}]]\!]^{\epsilon}_{\rho} = \mathtt{v}'$ (**C1.25**). Since we only consider programs in SSA, we know that $\mathtt{x} \notin \bigcup_{i=1}^{n_2} \mathsf{vars}(\mathsf{e}_i)$, from which it follows that $\mathtt{x} \notin \mathsf{vars}(Q'[\mathsf{e}_i/\mathtt{y}_i|_{i=1}^{n_2}]) \cup \mathsf{vars}(\mathsf{e}'[\mathsf{e}_i/\mathtt{y}_i|_{i=1}^{n_2}])$ (**C1.26**). Combining **C1.22**, **C1.25**, **C1.26**, and **I1** we get that $H', \rho[\mathtt{x} \mapsto \mathtt{v}'], \epsilon \models Q'[\mathsf{e}_i/\mathtt{y}_i|_{i=1}^{n_2}] * \mathsf{e} \doteq m' * \mathtt{x} \doteq \mathsf{e}'[\mathsf{e}_i/\mathtt{y}_i|_{i=1}^{n_2}]$ (*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 $\langle P, k, i \rangle \rightsquigarrow \langle Q, i+1 \rangle$ because $\mathtt{p}_m(i) = \mathtt{x} := \mathsf{e}(\mathsf{e}_1, ..., \mathsf{e}_{n_1})$ with j.

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

[Phi-Assignment] It follows that $\mathtt{p}_m(i) = \mathtt{x}_1, ..., \mathtt{x}_n := \phi(\mathtt{x}_1^1, ..., \mathtt{x}_1^r; ...; \mathtt{x}_n^1, ..., \mathtt{x}_n^r)$ for some JSIL variables $\mathtt{x}_t|_{t=1}^n$, $\mathtt{x}_t^u|_{t=1}^n|_{u=1}^r$. We conclude, using **H4** and the semantics of JSIL, that:

$$j \overset{k}{\mapsto}_m i \text{ (I1)} \qquad \mathtt{p} \vdash \langle \lfloor H \uplus \hat{H}_1 \uplus \hat{H}_2 \rfloor, \rho[\mathtt{x}_t \mapsto \rho(\mathtt{x}_t^k)|_{t=1}^n], i, i+1 \rangle \Downarrow_m \langle h_f, \rho_f, o \rangle \text{ (I2)}$$

for a given index k. Using **H1** and **H2**, we conclude that there is an assertion Q such that: $(Q, i) \in \mathtt{pd}(m, \mathit{fl}, i+1)$ (**I3**) and $\langle P, j, i \rangle \rightsquigarrow \langle Q, i+1 \rangle$ (**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 $H, \rho[\mathtt{x}_t \mapsto \rho(\mathtt{x}_t^k)|_{t=1}^n], \epsilon \models Q$. As in the previous cases, we proceed by induction on the derivation of $\langle P, j, i \rangle \rightsquigarrow \langle Q, i+1 \rangle$. More concretely, given that $H, \rho, \epsilon \models P$ and $\langle P, j, i \rangle \rightsquigarrow \langle Q, i+1 \rangle$, we need to show that $H, \rho[\mathtt{x}_t \mapsto \rho(\mathtt{x}_t^k)|_{t=1}^n], \epsilon \models Q$ (*goal*). In the following, we proceed by case analysis on the last rule applied to obtain $\langle P, j, i \rangle \rightsquigarrow \langle Q, i+1 \rangle$.

- [Phi-Assignment] We conclude that $Q = P * (\circledast_{t=1}^n \mathtt{x}_t \doteq \mathtt{x}_t^k)$ (**C1.1**). Because we assume programs are in SSA form, we conclude that $\mathtt{x}_t \notin \mathsf{vars}(P)|_{t=1}^n$ (**C1.2**). From **H3** and **C1.2**, we have that $H, \rho[\mathtt{x}_t \mapsto \rho(\mathtt{x}_t^k)|_{t=1}^n], \epsilon \models P$ (**C1.3**). Using the satisfiability of JSIL assertions, we get that $H, \rho[\mathtt{x}_t \mapsto \rho(\mathtt{x}_t^k)|_{t=1}^n], \epsilon \models \circledast_{t=1}^n \mathtt{x}_t \doteq \mathtt{x}_t^k$ (**C1.4**). Combining **C1.3** and **C1.4**, we conclude that $H, \rho[\mathtt{x}_t \mapsto \rho(\mathtt{x}_t^k)|_{t=1}^n], \epsilon \models Q$ (*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 $\langle P, j, i \rangle \rightsquigarrow \langle Q, i+1 \rangle$ because $\mathtt{p}_m(i) = \mathtt{x}_1, ..., \mathtt{x}_n := \phi(\overline{\mathtt{x}_1}; ...; \overline{\mathtt{x}_n})$.

Having established $H, \rho[\mathtt{x}_t \mapsto \rho(\mathtt{x}_t^k)|_{t=1}^n], \epsilon \models Q$ (**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 $\mathtt{v}$ such that: $o = \mathit{fl}\langle \mathtt{v} \rangle$ (**G1**), $\mathtt{p} \vdash \langle \lfloor H \uplus \hat{H}_1 \rfloor, \rho[\mathtt{x}_t \mapsto \rho(\mathtt{x}_t^k)|_{t=1}^n], i, i+1 \rangle \Downarrow_m \langle \lfloor H_f \uplus \hat{H}_1 \rfloor, \rho_f, o \rangle$ (**I6**), $H_f, \rho_f, \epsilon \models \mathsf{post}(\mathtt{S}(m, \mathit{fl}))$ (**G3**), $h_f = \lfloor H_f \uplus \hat{H}_1 \uplus \hat{H}_2 \rfloor$ (**G4**). Recalling that $\mathtt{p}_m(i) = \mathtt{x}_1, ..., \mathtt{x}_n := \phi(\mathtt{x}_1^1, ..., \mathtt{x}_1^r; ...; \mathtt{x}_n^1, ..., \mathtt{x}_n^r)$, it follows from **I1** and **I6** that $\mathtt{p} \vdash \langle \lfloor H \uplus \hat{H}_1 \rfloor, \rho, j, i \rangle \Downarrow_m \langle \lfloor H_f \uplus \hat{H}_1 \rfloor, \rho_f, o \rangle$ (**G2**).

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

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

- [All Other Cases] No other rule may have been applied in the derivation of $\langle P, k, \mathtt{ret} \rangle \rightsquigarrow \langle Q, \mathtt{ret} \rangle$ because $\mathtt{ret}$ has no successor but itself.

Having proven that $H_f, \rho_f, \epsilon \models \mathsf{post}(\mathtt{S}(m, \mathtt{nm}))$ (**G3**), we proceed to the remaining goals. We obtain **G1** directly from **I4** and **I6**, while we obtain **G4** directly from **I2**. Using the semantics of JSIL and **I1**, we get that: $\mathtt{p} \vdash \langle \lfloor H \uplus \hat{H}_1 \rfloor, \rho, k, i \rangle \Downarrow_m \langle \lfloor H \uplus \hat{H}_1 \rfloor, \rho, \mathtt{nm}\langle \rho(\mathtt{xret}) \rangle \rangle$ (**I7**). From **I3**, **I4**, and **I7**, we obtain **G2**.

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

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

- [All Other Cases] No other rule may have been applied in the derivation of $\langle P, k, \mathtt{err} \rangle \rightsquigarrow \langle Q, \mathtt{err} \rangle$ because $\mathtt{err}$ has no successor but itself.

Having proven that $H_f, \rho_f, \epsilon \models \mathsf{post}(\mathtt{S}(m, \mathtt{er}))$ (**G3**), we proceed to the remaining goals. We obtain **G1** directly from **I4** and **I6**, while we obtain **G4** directly from **I2**. Using the semantics of JSIL and **I1**, we get that: $\mathtt{p} \vdash \langle \lfloor H \uplus \hat{H}_1 \rfloor, \rho, k, i \rangle \Downarrow_m \langle \lfloor H \uplus \hat{H}_1 \rfloor, \rho, \mathtt{er}\langle \rho(\mathtt{xerr}) \rangle \rangle$ (**I7**). From **I3**, **I4**, and **I7**, we obtain **G2**.

□

Theorem 6.1 (Soundness of Symbolic Execution for JSIL)**.** *For all JSIL programs* $\mathtt{p}$ *and specification environments* $\mathtt{S}$*, if there exists a proof candidate* $\mathtt{pd} \in \mathcal{D}$ *such that* $\mathtt{p}, \mathtt{S} \vdash \mathtt{pd}$*, then:*

$$\forall m, \mathit{fl}, P, Q, \overline{\mathtt{x}}\,.\ \mathtt{S}(m, \mathit{fl}) = \{P\} m(\overline{\mathtt{x}}) \{Q\} \implies \mathtt{p}, \mathit{fl} \vDash \{P\} m(\overline{\mathtt{x}}) \{Q\}$$

Proof. We have to prove that for every procedure identifier m, return modes fl, fl', JSIL abstract heap H, concrete heap h_f, stores ρ and ρ_f, environment ϵ, and JSIL value $\mathtt{v}$ such that $\mathtt{S}(m, \mathit{fl}) = \{P\}\, m(\overline{\mathtt{x}})\, \{Q\}$ (**I1**), $H, \rho, \epsilon \models P$ (**I2**), and $\mathtt{p} \vdash \langle \lfloor H \rfloor, \rho, -, 0 \rangle \Downarrow_m \langle h_f, \rho_f, \mathit{fl}' \langle \mathtt{v} \rangle \rangle$ (**I3**), it holds that there is an abstract heap H_f such that $\lfloor H_f \rfloor = h_f$ (**G1**), $H_f, \rho_f, \epsilon \models Q$ (**G2**), and $\mathit{fl} = \mathit{fl}'$ (**G3**). From $\mathtt{p}, \mathtt{S} \vdash \mathtt{pd}$ and **I1**, we conclude that $\mathtt{pd}(m, \mathit{fl}, 0) = \{(P, 0)\}$ (**I4**). We can now apply the Frame Property and Soundness for Control Flow Commands (Lemma 2) to $\mathtt{p}, \mathtt{S} \vdash \mathtt{pd}$, **I4**, **I2**, and **I3**, by choosing $\hat{H}_1 = \hat{H}_2 = \mathtt{emp}$, concluding that there exists an abstract heap H_f such that **G1-G3** hold. □

6.4. JSIL Verify

We give an overview of JSIL Verify, a semi-automatic verification tool for JSIL. The high-level architecture of JSIL Verify is presented in Figure 6.7. Given a JSIL program annotated with the specifications of its procedures, and the specifications of the JavaScript internal functions, JSIL Verify checks whether or not the program procedures satisfy their specifications. JSIL Verify consists of: a symbolic execution engine that symbolically executes JSIL commands according to the proof rules of JSIL Logic (§6.2); and an entailment engine for resolving frame inference and entailment questions, which is supported by the Z3 SMT solver [21].

We describe the symbolic execution engine and the entailment engine, and explain how we use JavaScript internal functions during symbolic execution. We also discuss how JSIL Verify evolved over time.

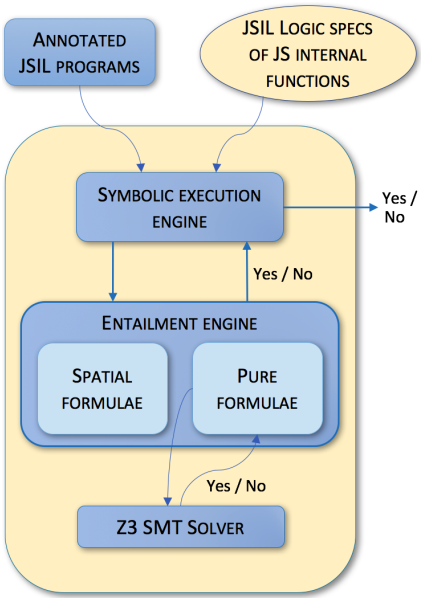


Figure 6.7.: Architecture of JSIL Verify

Symbolic Execution Engine. The symbolic execution engine of JSIL Verify symbolically executes JSIL commands according to the proof rules of JSIL Logic (§6.2). During symbolic execution, we encounter frame inference and entailment questions that need to be resolved. We start with a given precondition P of a procedure m, and follow its control flow graph. Whenever a basic command $\mathtt{bc}$ or a procedure call to a procedure m' is encountered, we take its precondition P', solve the frame inference question $P \vdash P' * [?F]$, and continue to the next command in the state $Q' * ?F$, where Q' is the postcondition of $\mathtt{bc}$ or m'. When we reach the final labels of the procedure m, we check if the final symbolic state Q_f entails the postcondition Q of the procedure m, by solving the entailment $Q_f \vdash Q * \mathit{true}$.

To give a flavour of the symbolic execution, let us symbolically execute `[x__er, "n"] := undefined` from the `enqueue` procedure (Figure 5.9, line 4). Before executing this command, the symbolic state

is $P * \mathtt{x}_{er} \doteq l_{er} * \mathsf{emptyProps}(l_{er} \mid [@proto])$ [1]. To apply the small axiom $\{(\mathtt{x}_{er}, \text{“n”}) \mapsto \_\}$ [x__er, "n"] := undefined $\{(\mathtt{x}_{er}, \text{“n”}) \mapsto \mathsf{undefined}\}$, we need to solve the frame inference problem $P * \mathtt{x}_{er} \doteq l_{er} * \mathsf{emptyProps}(l_{er} \mid [@proto]) \vdash (\mathtt{x}_{er}, \text{“n”}) \mapsto \_ * [?F]$. In this case, we get that the frame is $P * \mathtt{x}_{er} \doteq l_{er} * \mathsf{emptyProps}(l_{er} \mid [@proto, \text{“n”}])$, which we explain in more detail shortly. The symbolic state after executing the command is $(\mathtt{x}_{er}, \text{“n”}) \mapsto \mathsf{undefined} * P * \mathtt{x}_{er} \doteq l_{er} * \mathsf{emptyProps}(l_{er} \mid [@proto, \text{“n”}])$.

Entailment Engine: Frame Inference. The frame inference problem that JSIL Verify has to solve is more complex than those featured in equivalent tools for static languages, such as C and Java. Namely, as JSIL features dynamic property access, the property of a cell assertion is an arbitrary logical expression and not a concrete string. This makes symbolic evaluation of object management commands non-trivial. Consider, for instance, the property assignment $[\mathsf{e}_1, \mathsf{e}_2] := \mathsf{e}_3$. To symbolically execute this command in a symbolic state P, JSIL Verify must solve the following instance of the frame inference problem $P \vdash (\mathsf{e}_1, \mathsf{e}_2) \mapsto \_ * [?F]$, where $?F$ denotes the resources to be framed off. In this case, solving the frame inference problem involves: traversing all the cell assertions $(\mathrm{E}_1, \mathrm{E}_2) \mapsto \_$ in P, checking for each one whether $P \vdash \mathsf{e}_i \doteq \mathrm{E}_i \mid_{i=1,2}$; and traversing all the empty property assertions $\mathsf{emptyProps}(\mathrm{E}_1 \mid \mathrm{E}_2)$ in P, checking for each one whether $P \vdash \mathsf{e}_1 \doteq \mathrm{E}_1$ and $P \vdash \mathsf{e}_2 \dot{\notin} \mathrm{E}_2$.

For the first version of JSIL Verify, we resorted to using the coreStar [11] theorem prover for solving frame inference and entailment questions. To achieve that, we translated JSIL assertions to coreStar assertions, by defining corresponding built-in predicates and logic rules for them in coreStar. We have demonstrated that it is possible to define logic rules for built-in predicates, such as $(\mathrm{E}_1, \mathrm{E}_2) \mapsto \mathrm{E}_3$ or $\mathsf{emptyProps}(\mathrm{E}_1 \mid \mathrm{E}_2)$. To evaluate JSIL Verify, we verified small JavaScript examples that use prototype-based inheritance and a number of examples from Test262 test suite. However, as our experience showed, it was extremely difficult to provide logic rules without having any control of their application, making the proof search untractable. Moreover, the specifications of these examples were exposing the internal details of JavaScript. Even though it would be possible to implement built-in JavaScript abstractions, the coreStar approach would not scale for user defined predicates, as it would be difficult to automatically generate logic rules given a predicate definition. Also, it would not be reasonable to ask of the user to define predicates in terms of logic rules, as that would require knowledge of the internals of coreStar.

As the project evolved, we stepped away from coreStar and implemented our own entailment engine for resolving frame inference and entailment questions. As in [5], given the frame inference problem $P \vdash Q * [?F]$, we first decompose P and Q into pairs of the form (Σ, Π), denoting respectively their spatial and pure parts. Hence, we are left with $(\Sigma_p, \Pi_p) \vdash (\Sigma_q, \Pi_q) * [?F]$, which can be further decomposed into: **(1)** $(\Sigma_p, \Pi_p) \vdash (\Sigma_q, \mathsf{True}) * [?F]$ and the pure entailment **(2)** $\Pi_p \vdash \Pi_q$. In Figure 6.8, we present a proof system for solving **(1)**, which we rewrite as $\Sigma_p \mid \Pi_p \vdash \Sigma_q * [?F]$. This proof system makes use of a pure entailment oracle to check entailments between pure assertions of the form $\Pi_1 \vdash \Pi_2$. In the implementation, this oracle is a function that encodes the pure entailment to be checked into Z3 and then asks of Z3 to actually check it. The main idea behind the proof system is to remove matching spatial parts from both sides of the entailment (rules Cell, EmptyProps - None Cell, EmptyProps - EmptyProps) until the right hand side is empty (rule Emp). What is left on the left hand side is moved to the frame (rule Frame).

[1] We will give the specification of enqueue in §8.4. Here, we treat it as a black box: P.

$$\textsc{Cell}\ \frac{\Pi \vdash \mathrm{E}_i = \mathrm{E}'_i \ |_{i=1,2,3} \quad \Sigma_1 \mid \Pi \vdash \Sigma_2 * [?F]}{\Sigma_1 * (\mathrm{E}_1, \mathrm{E}_2) \mapsto \mathrm{E}_3 \mid \Pi \vdash \Sigma_2 * (\mathrm{E}'_1, \mathrm{E}'_2) \mapsto \mathrm{E}'_3 * [?F]} \qquad \textsc{Frame}\ \frac{\Sigma_1 \mid \Pi \vdash \Sigma_2 * [?F]}{\Sigma_1 * \Sigma \mid \Pi \vdash \Sigma_2 * [?F * \Sigma]}$$

$$\textsc{EmptyProps - None Cell}\ \frac{\Pi \vdash \mathrm{E}_1 = \mathrm{E}'_1 \quad \Pi \vdash \mathrm{E}'_2 \notin \mathrm{E}_2 \quad \Sigma_1 * \mathsf{emptyProps}(\mathrm{E}_1 \mid \mathrm{E}_2 \cup \{\, \mathrm{E}'_2 \,\}) \mid \Pi \vdash \Sigma_2 * [?F]}{\Sigma_1 * \mathsf{emptyProps}(\mathrm{E}_1 \mid \mathrm{E}_2) \mid \Pi \vdash \Sigma_2 * (\mathrm{E}'_1, \mathrm{E}'_2) \mapsto \varnothing * [?F]} \qquad \textsc{Emp}\ \ \mathsf{emp} \mid \Pi \vdash \mathsf{emp} * [\mathsf{emp}]$$

$$\textsc{EmptyProps - EmptyProps}\ \frac{\Pi \vdash \mathrm{E}_0 = \mathrm{E}'_0 \quad \Pi \vdash \mathrm{E} \backslash \{\mathrm{E}_i \mid_{i=1}^{k}\} = \mathrm{E}' \quad \Sigma_1 \mid \Pi \vdash \Sigma_2 * [?F]}{\Sigma_1 * \circledast_{1 \le i \le k}(\mathrm{E}_0, \mathrm{E}_i) \mapsto \varnothing * \mathsf{emptyProps}(\mathrm{E}_0 \mid \mathrm{E}) \mid \Pi \vdash \Sigma_2 * \mathsf{emptyProps}(\mathrm{E}'_0 \mid \mathrm{E}') * [?F]}$$

Figure 6.8.: Proof System for Frame Inference - $\Sigma_1 \mid \Pi \vdash \Sigma_2 * [?F]$

$$\dfrac{\dfrac{\Pi \wedge \mathtt{x}_{er} = l_{er} \vdash \text{“n”} \notin [@\mathit{proto}] \qquad \dfrac{\mathsf{emp} \mid \Pi \vdash \mathsf{emp} * [\mathsf{emp}]\ \textsc{Emp}}{\Sigma_F \mid \Pi \vdash \mathsf{emp} * [\Sigma_F]}\ \textsc{Frame}}{\mathsf{emptyProps}(l_{er} \mid [@\mathit{proto}]) \mid \Pi_p \wedge \mathtt{x}_{er} = l_{er} \vdash (\mathtt{x}_{er}, \text{“n”}) \mapsto \varnothing * [\Sigma_F]}\ \textsc{EF - None}}{\mathsf{emptyProps}(l_{er} \mid [@\mathit{proto}]) * \Sigma_p \mid \Pi_p \wedge \mathtt{x}_{er} = l_{er} \vdash (\mathtt{x}_{er}, \text{“n”}) \mapsto \varnothing * [\Sigma_F * \Sigma_p]}\ \textsc{Frame}$$

$$\Sigma_F = \mathsf{emptyProps}(l_{er} \mid [@\mathit{proto}, \text{“n”}])$$

Figure 6.9.: Example Derivation of the Proof System for Frame Inference

To illustrate the use of this proof system, we demonstrate the derivation of $P * \mathtt{x}_{er} \doteq l_{er} * \mathsf{emptyProps}(l_{er} \mid [@\mathit{proto}]) \vdash (\mathtt{x}_{er}, \text{“n”}) \mapsto \_ * [?F]$ in Figure 6.9, where $P = (\Sigma_p, \Pi_p)$. The computed frame $?F$ coincides with the spatial part of the original symbolic state except that the property `n` is removed from the infinite footprint of the emptyProps assertion.

Entailment Engine: Pure Entailment. JSIL Verify discharges pure entailments of the form $\Pi_1 \vdash \Pi_2$ to the Z3 SMT solver [21]. To this end, it encodes JSIL Logic pure assertions as Z3 formulae. Z3 gives native support for arithmetic, bit-vectors, arrays, and uninterpreted functions. It additionally supports the definition of new algebraic data-types. We encoded JSIL Logic values as a Z3 algebraic data type taking advantage of Z3 native types when possible, and specified the operations for the JSIL value types not natively supported using uninterpreted functions. For instance, Z3 does not have native support for list reasoning; therefore, we had to encode lists and list theory axioms manually. On the other hand, we were able to leverage on Z3 native set support and reasoning.

JavaScript Internal Functions. Initially, we did not have JSIL reference implementations of JavaScript internal functions. Instead of calling a JavaScript internal function from generated JSIL code, as we demonstrated in §5.1, we inlined the body of the JavaScript internal function. This was not feasible as a long-term solution because of a blow-up in the size and readability of generated JSIL code.

To solve the problem of the blow-up, we provide JSIL reference implementations for JavaScript internal functions and use procedure calls in the general JSIL code. To use our JSIL Logic rules for procedure calls during symbolic execution, we provide JSIL specifications for reference implementations of JavaScript internal functions. One aspect of future work could be to investigate how verification time would be impacted if, instead of using the specifications of JavaScript internal functions, we were to symbolically execute their bodies upon some/all procedure calls.

JavaScript internal functions were an important use case and source of validation for JSIL Verify itself. Using JSIL Verify, we prove that reference implementations of JavaScript internal functions

satisfy their axiomatic specifications. These functions consistently exercise the dynamic behaviour that underpins JSIL Verify, and we found the obtained verification time (186 specifications that we currently have for the internal functions are verified in under six seconds) encouraging. More details about JavaScript internal functions, their specifications, and their applications are given in the next chapter (§7).

Summary. We presented the JSIL verification infrastructure, which, given an annotated JSIL program, checks whether or not the program procedures satisfy their specifications. In the following chapter, we illustrate the specification and verification of the JSIL implementations of the JavaScript internal functions and show how these specifications are used in the verification of compiled JavaScript code.

7. The JS-2-JSIL Environment

JavaScript internal functions are used to describe the concepts of the language, including prototype chain traversal, object management, and type conversions. They are called extensively by all JavaScript commands. Therefore, in order to reason about JavaScript code, we have to first be able to reason efficiently about the internal functions. However, their definitions in the standard are complex, are given operationally, and are often intertwined, making it difficult for the user to fully grasp the control flow and allowed behaviours.

As we provide reference implementations of all internal functions, any call to an internal function gets translated to JSIL as a procedure call to our corresponding reference implementation, which we demonstrated in §5. For example, the compiled JavaScript function `enqueue` in Figure 5.9, features a number of the internal functions being called in the JSIL code: `GetValue`, `PutValue`, `CheckObjectCoercible`, and `IsCallable`.

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

We believe that our JSIL axiomatic specifications of the internal functions are an important contribution. They directly benefit JaVerT, since the verification of JavaScript code only needs to use the specifications, not the underlying implementations. We envisage that these specifications will be useful beyond JaVerT. For example, starting from our axiomatic specifications, we could create executable specifications of the internal functions, that could then be used for different types of symbolic analysis for JavaScript. They would also provide a mechanism for restricting the semantics of JavaScript in a principled way. If, for instance, we would like to perform an analysis that wishes to abstract a semantic feature of JavaScript, say type coercion, we would generate executable specifications of the internal functions without taking into account the axiomatic specifications that describe type coercion. This would be much more robust than altering the code of the internal functions manually.

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

work on higher-order separation logics [60, 7, 62, 36] and the verification tool VeriFast [36] which is based on one such logic.

We illustrate our specifications of JavaScript internal functions using `GetValue` and `PutValue`, two internal functions that deal with references. But first, we present our `Pi` predicate, an abstraction to precisely capture the prototype chains of JavaScript, and without which the specifications of internal functions would be impossible.

7.1. Capturing JavaScript prototype chains: the Pi predicate

For our Pi predicate, we take inspiration from the prototype-chain predicate of Gardner et al. [30]. Their predicate is much simpler as it describes prototype chains of standard objects with simple values, whereas ours describes prototype chains for property descriptors and accounts for the subtle combination of standard objects and string objects, capturing the full prototype inheritance of JavaScript.

To design the Pi predicate correctly, we need to understand the resources required for a property lookup `l[p]` in the setting of prototype inheritance. Even though we do not consider higher-order logic, there is no reason to limit ourselves to only data descriptors when it comes to the Pi predicate. The Pi predicate basically describes the axiomatic semantics of the JavaScript internal function `GetProperty`, the implementation of which we already have seen in §4.3. `GetProperty` calls another internal function, `GetOwnProperty`, which determines if the given object has the given property. If the property `p` is defined in the object `l` and is associated with a descriptor d, then the descriptor d is the result of `GetProperty`. Otherwise, the prototype chain of the object `l` is recursively inspected. First, let us consider only standard JavaScript objects.

For the first attempt at defining the prototype-chain predicate, Pi_1, we recognise that its parameters have to include: the object l, the property p, and the result of the lookup v. Next, we need to capture the actual prototype chain. For this, we need a list ls containing locations of the objects in the chain up to and including the one in which the property is found (or all of them if it is not found):

$$
\begin{array}{rcl}
\mathsf{Pi}_1(l, p, \mathsf{undefined}, [l]) & \triangleq & (l, p) \mapsto \varnothing \ * \ (l, @\mathit{proto}) \mapsto \mathsf{null} \\
\mathsf{Pi}_1(l, p, d, [l]) & \triangleq & (l, p) \mapsto d \ * \ d \dot{\neq} \varnothing \\
\mathsf{Pi}_1(l, p, d, l :: l_p :: ls) & \triangleq & (l, p) \mapsto \varnothing \ * \ (l, @\mathit{proto}) \mapsto l_p \ * \ l_p \dot{\neq} \mathsf{null} \ * \ \mathsf{Pi}_1(l_p, p, d, l_p :: ls)
\end{array}
$$

The first base case reads as follows: the lookup of a property p of the object l yields the undefined value, the property is not defined in the object itself, and the object is the last one in its prototype chain. The second base case states: the lookup of a property p of the object l yields the descriptor d, the property is defined in the object itself, and its associated value is the descriptor d. Finally, the recursive case reads: the lookup of a property p in the object l yields the value d, the property is not defined in the object itself, the prototype of the object is l_p, and the lookup of p in the rest of the prototype chain yields the data descriptor d.

Such a definition is not sufficient, since the `GetOwnProperty` function is different for `String` objects, and the Pi predicate needs to account for that. For a lookup `l[p]`, if the object `l` is a `String` object, we would similarly start by looking for the property `p`. However, if `p` is not found in the object `l`, before

inspecting its prototype, an additional check is done. This additional check converts `p` to a number which represents an index `i`. If the index `i` is a valid index for a primitive value `pv` associated with the `String` object `l`, then the $\texttt{i}^{th}$ character of the `pv` is the result of `GetOwnProperty`. For example, consider a string object l with a primitive value "foo", that is, $(l, @primval) \mapsto$ "foo" and two properties "bar" and "9", that is, $(l,$ "bar"$) \mapsto v_1$ and $(l,$ "9"$) \mapsto v_2$. Then, the result of `GetOwnProperty` for the object l: for the property "bar" would be v_1; for the property "9" would be v_2; for the property "1" would be "o"; and for the properties "test" and "3" would be undefined. Interestingly, the properties that correspond to valid indexes of a `String` object do not exist in the heap and cannot be changed. If we tried to add a property "0" to the object l, the JavaScript internal function `DefineOwnProperty` would throw an exception, because `GetOwnProperty` of "0" would return the data descriptor with the attribute `writable` being false.

To account for such behaviour, we define a cell abstraction for `String` objects as follows:

$$
\begin{array}{rcl}
\mathsf{stringIndex}(s,p,i) & \triangleq & p \doteq \mathsf{toString}\,(i) \;*\; 0 \dot{\leq} i \;*\; i \dot{<} \mathsf{length}_s(s) \\
(l,p) \mapsto^s_{pv} \varnothing & \triangleq & \exists\, i.\ (l,p) \mapsto \varnothing \;*\; (l, @primval) \mapsto pv \;* \\
 & & i \doteq \mathsf{toString}\,(\mathsf{toNumber}\,(p)) \;*\; \neg\mathsf{stringIndex}(pv,p,i) \\
(l,p) \mapsto^s_{pv} d & \triangleq & \exists\, i, v.\ (l,p) \mapsto \varnothing \;*\; (l, @primval) \mapsto pv \;* \\
 & & d \doteq [\text{“d”}, v, \mathsf{false}, \mathsf{true}, \mathsf{false}] \;*\; v \doteq \mathsf{nth}_s(pv,i) \;* \\
 & & i \doteq \mathsf{toString}\,(\mathsf{toNumber}\,(p)) \;*\; \mathsf{stringIndex}(pv,p,i) \\
(l,p) \mapsto^s_{pv} d & \triangleq & \exists\, i.\ (l,p) \mapsto d \;*\; d \dot{\neq} \varnothing \;*\; (l, @primval) \mapsto pv \;* \\
 & & i \doteq \mathsf{toString}\,(\mathsf{toNumber}\,(p)) \;*\; \neg\mathsf{stringIndex}(pv,p,i)
\end{array}
$$

First, we define what it means for a property p to be a valid string index i for a string s, denoted by $\mathsf{stringIndex}(s,p,i)$. Given a string s, a property p, which is a string, and an index i, which is a number, $\mathsf{stringIndex}(s,p,i)$ is a valid string index if the index i, corresponding to the property p successfully converted to an integer, is greater or equal than zero and is smaller than the length of the string s.

Next, the abstraction $(l,p) \mapsto^s_{pv} d$, given an object l, a property p, a primitive value pv of the object l, and d, which can be either a descriptor or $\varnothing$, describes that either: the object l neither contains the property p in the heap, nor p is a valid string index for the primitive value of the object; the object l does not contain the property p in the heap, but the property p is a valid string index for the primitive value of the object and the descriptor d is associated with its i-th character; or the object l contains the property p associated with the descriptor d and p is not a valid string index for the primitive value of the object.

To account for `String` objects, we require two additional parameters for our Pi predicate: the list vs_{cls} of values of the `@class` internal property, to be able to distinguish between `String` object and all other objects; and the list vs_{pv} of values of the `@primval` internal property, for all objects in the list ls, to be able to use primitive value in the cell abstraction for `String` objects. The precise abstraction needed in order to reason about JavaScript prototype chains is $\mathsf{Pi}(l,p,v,ls,vs_{cls},vs_{pv})$. We show the full definition of Pi below. First, we show the base cases and the recursive case for the objects that are not `String` objects, followed by the cases for the `String` objects:

$$
\begin{array}{rcl}
\mathsf{Pi}(l, p, \mathsf{undefined}, [l], [cls], \_) & \triangleq & (l, @class) \mapsto cls * cls \dot{\neq} \text{“String”} * \\
& & (l, p) \mapsto \varnothing \ * \ (l, @proto) \mapsto \mathsf{null} \\
\mathsf{Pi}(l, p, d, [l], [cls], \_) & \triangleq & (l, @class) \mapsto cls * cls \dot{\neq} \text{“String”} * \\
& & (l, p) \mapsto d \ * \ d \dot{\neq} \varnothing \\
\mathsf{Pi}(l, p, d, l :: l_p :: ls, cls :: vs_{cls}, \_ :: vs_{pv}) & \triangleq & (l, @class) \mapsto cls * cls \dot{\neq} \text{“String”} \ * \ (l, p) \mapsto \varnothing * \\
& & (l, @proto) \mapsto l_p * l_p \dot{\neq} \mathsf{null} \ * \ \mathsf{Pi}(l_p, p, d, l_p :: ls, vs_{cls}, vs_{pv}) \\
\mathsf{Pi}(l, p, \mathsf{undefined}, [l], [cls], [pv]) & \triangleq & (l, @class) \mapsto cls * cls \doteq \text{“String”} * \\
& & (l, p) \mapsto^{s}_{pv} \varnothing \ * \ (l, @proto) \mapsto \mathsf{null} \\
\mathsf{Pi}(l, p, d, [l], [cls], [pv]) & \triangleq & (l, @class) \mapsto cls * cls \doteq \text{“String”} * \\
& & (l, p) \mapsto^{s}_{pv} d \ * \ d \dot{\neq} \varnothing \\
\mathsf{Pi}(l, p, d, l :: l_p :: ls, cls :: vs_{cls}, pv :: vs_{pv}) & \triangleq & (l, @class) \mapsto cls * cls \doteq \text{“String”} \ * \ (l, p) \mapsto^{s}_{pv} \varnothing * \\
& & (l, @proto) \mapsto l_p * l_p \dot{\neq} \mathsf{null} \ * \ \mathsf{Pi}(l_p, p, d, l_p :: ls, vs_{cls}, vs_{pv})
\end{array}
$$

The first three cases correspond directly to the cases of the Pi_1 predicate with the additional information of describing that the object l is not a `String` object. The remaining cases for `String` objects use the abstraction $\mapsto^{s}_{pv}$ instead of $\mapsto$, and are otherwise analogous to the cases for standard objects.

Recall our running example (Figure 3.9) and the heap obtain from its execution (Figure 3.10). When we add the first element to the queue, `q.enqueue(1, "last")`, a new node object `n1` is created (see Figure 3.10). Its prototype chain contains `n1`, `Node.prototype`, and `Object.prototype`. We illustrate the Pi predicate using the object `n1` and the property lookups `n1.pri`, yielding the data descriptor (1, `true`, `true`, `true`) and `n1.foo`, yielding `undefined`. The Pi predicate for the former lookup, using the second base case is:

$$\mathsf{Pi}(\texttt{n1}, \text{“pri”}, [\text{“d”}, 1, \mathsf{true}, \mathsf{true}, \mathsf{true}], [\texttt{n1}], [\text{“Object”}], \_).$$

The Pi predicate for the latter lookup, on the other hand, requires two unfoldings of the recursive case and the first base case:

$$\mathsf{Pi}(\texttt{n1}, \text{“foo”}, \mathsf{undefined}, [\texttt{n1}, \texttt{Node.prototoype}, \texttt{Object.prototype}], [\text{“Object”}, \text{“Object”}, \text{“Object”}], \_).$$

7.2. Specifying Internal Functions

The definitions of the JavaScript internal functions in the ECMAScript standard are complex and often intertwined, making it difficult to fully grasp the control flow and allowed behaviours. To illustrate: `GetValue` calls `Get`, which calls `GetProperty`, which calls `GetOwnProperty`; `PutValue` calls `Put`, which calls `CanPut` and `DefineOwnProperty`, which calls `GetOwnProperty`. The precise call graph of `GetValue` and `PutValue` is given in Figure 7.1. Specifying such dependencies axiomatically involves the joining of the specifications of all nested functions at the top level, which is highly non-trivial and results in numerous branchings. The resulting specifications, however, are much more readable than the operational definitions of the standard.

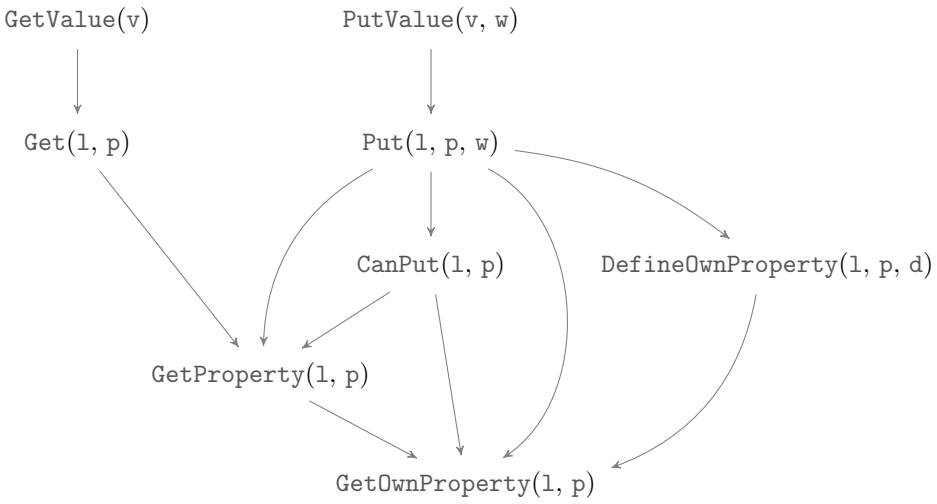


Figure 7.1.: Call graph for `GetValue` and `PutValue`

We illustrate how specifications of internal functions `GetValue` and `PutValue` are constructed, using the specifications of the internal functions they depend on. We present one specification for `GetValue`, two specifications for `PutValue`, and a number of specifications for the underlying internal functions.

The `GetOwnProperty` internal function. In the previous section, we introduced all the ingredients required to specify `GetOwnProperty`. The specification of the `GetOwnProperty`, given below, contains two cases for `Non−String` objects: either the property `p` is not defined in the object `l`, yielding undefined as the result of the function; or the property `p` of the object `l` contains the data descriptor d, which is the result of the function (for readability, we write Pre instead of repeating the entire precondition).

$$\left\{ \quad (\mathsf{l}, \mathsf{p}) \mapsto \varnothing \ * \ (\mathsf{l}, @class) \mapsto cls \ * \ cls \dot{\neq} \text{“String”} \quad \right\}$$
$$\texttt{GetOwnProperty(l, p)}$$
$$\left\{ \quad Pre \ * \ \mathsf{ret} \doteq \mathsf{undefined} \quad \right\}$$

$$\left\{ \quad (\mathsf{l}, \mathsf{p}) \mapsto d \ * \ d \dot{\neq} \varnothing \ * \ (\mathsf{l}, @class) \mapsto cls \ * \ cls \dot{\neq} \text{“String”} \quad \right\}$$
$$\texttt{GetOwnProperty(l, p)}$$
$$\left\{ \quad Pre \ * \ \mathsf{ret} \doteq d \quad \right\}$$

Similarly, there are two more cases for `GetOwnProperty` considering `String` objects, and using the abstraction $(\mathsf{l}, \mathsf{p}) \mapsto^{s}_{pv} d$ instead of $(\mathsf{l}, \mathsf{p}) \mapsto d$.

The `GetProperty` internal function. The Pi predicate describes prototype chains in JavaScript, while the internal JavaScript function `GetProperty` implements the traversal of prototype chains. Hence, the specification of `GetProperty` simply uses the Pi predicate:

$$\left\{ \quad \mathsf{Pi}(\mathsf{l}, \mathsf{p}, v, ls, vs_{cls}, vs_{pv}) \quad \right\}$$
$$\texttt{GetProperty(l, p)}$$
$$\left\{ \quad Pre \ * \ \mathsf{ret} \doteq v \quad \right\}$$

In the precondition, we have that the result of looking for the property `p` in the prototype chain of

the object l yields a value v. The postcondition states that GetProperty does not affect any resources and returns v.

Recall the reference implementation of GetProperty given in §4.3. To verify that the reference implementation satisfies the given specification, we need to guide JSIL Verify by providing fold/unfold annotations for the inductive predicate Pi. In Figure 7.2 we show the annotated body of GetProperty. We unfold Pi in line 3 before calling GetOwnProperty, since it requires the resource of a single cell. We fold Pi in line 11 to obtain the postcondition of GetProperty.

```
1  proc getProperty (l, prop) {
2
3          [* unfold Pi (l, prop, _v, _ls, _vscls, _vspn) *]
4          own := "getOwnProperty" (l, prop) with perr;
5          goto [own = undefined] next pret;
6
7    next: proto := [l, "@proto"];
8          goto [proto = null] pret call;
9
10   call: chain := "getProperty" (proto, prop) with perr;
11
12         [* fold Pi (l, prop, _v, _ls, _vscls, _vspn) *]
13   pret: xret := phi(own, own, chain);
14   ret:  skip
15
16   perr: xerr := phi(own, chain);
17   err:  skip
18 }
```

Figure 7.2.: An annotated JSIL implementation of GetProperty.

The Get internal function. The Get(l, p) internal function calls GetOwnProperty to traverse the prototype chain and retrieve associated value which can be undefined, a data descriptor, or an accessor descriptor. If the result of GetOwnProperty is undefined, so is the result of Get. If the result of GetOwnProperty is a data descriptor, then the value of an attribute [V] is the result of Get. Otherwise, the result of GetOwnProperty is an accessor descriptor, in which case the getter associated with the accessor descriptor is being called. Here, we show the specification of Get when the result of GetProperty is a data descriptor:

$$\left\{ \quad \mathsf{Pi}(\mathsf{l}, \mathsf{p}, d, ls, vs_{cls}, vs_{pv}) \ * \ \mathsf{DescVal}(d, w) \quad \right\}$$
$$\mathtt{Get(l, p)}$$
$$\left\{ \quad Pre \ * \ \mathsf{ret} \doteq w \quad \right\}$$

In the precondition, we have that p is defined in the prototype chain of l and that the corresponding data descriptor d has value w. We use the abstraction $\mathsf{DescVal}(d, w) \triangleq d \doteq [\text{“d”}, w, \_, \_, \_]$ to denote a data descriptor d which has the value w of the attribute [V]. The postcondition states that, in this case, Get does not affect any resources and returns w. Notice how the precondition of Get is being built by using the precondition of GetProperty, which is highlighted in a different colour.

The GetValue internal function. GetValue(v) is the JavaScript internal function that performs dereferencing. It takes one parameter: the value v to be dereferenced. If v is not a reference, it is returned immediately. If v is a reference with the base undefined, a JavaScript reference error is thrown. If v is a reference with the base being a primitive value, such as string, number or boolean, then the base is converted to an object and a special internal function Get is called. Otherwise, v = ["o"/"v", l, p] and, in that case, GetValue returns the value associated with the property p of

object `l`. If `v` is a variable reference whose base is not the global object, this value is obtained by directly inspecting the heap. Otherwise, `GetValue` uses the `Get` internal function to traverse the prototype chain and obtain the appropriate value. Here, we show the specification of `GetValue` for the case in which `v` is an object reference and the corresponding property is defined as a data descriptor.

$$\left\{ \ \mathsf{v} \doteq [\text{“o”}, l, p] \ * \ \mathsf{Pi}(l, p, d, ls, vs_{cls}, vs_{pv}) \ * \ \mathsf{DescVal}(d, w) \ \right\}$$
$$\texttt{GetValue(v)}$$
$$\left\{ \ Pre \ * \ \mathsf{ret} \doteq w \ \right\}$$

In the precondition, we require an object reference `v`. Building on top of the precondition of `Get`, we have that p is defined in the prototype chain of l and that the corresponding data descriptor d has value w. The postcondition states that `GetValue` does not affect any resources and returns w.

The `DefineOwnProperty` internal function. Given an object at location `l`, a property `p`, and a descriptor `d`, `DefineOwnProperty(l, p, d)` performs the actual heap update. It is one of the most complicated JavaScript internal functions as it needs to perform various validations before updating the heap. For example, if the object `l` is not extensible and does not contain the property `p`, the heap update is not allowed. Also, if the property `p` is already defined in the object `l`, and contains a descriptor d_{old} with the configurable attribute being `false`, it is not allowed to update the property with a descriptor which would change the value of the configurable attribute to `true`. As a result of all such validations, `DefineOwnProperty` has over fifteen specifications. Moreover, `Array` objects have a different behaviour, since the `length` property needs to be updated every time an update happens to an index property of an `Array` object. Here, we give just one specification of `DefineOwnProperty` for objects that are not `Array` or `String` objects, where the object `l` is extensible and does not contain the property `p`, and the descriptor `d` is a data descriptor:

$$\left\{ \begin{array}{c} (\mathsf{l}, @class) \mapsto cls \ * \ cls \dot{\neq} \text{“String”} * \ cls \dot{\neq} \text{“Array”} \ * \\ (\mathsf{l}, @extensible) \mapsto \mathsf{true} \ * \ (\mathsf{l}, \mathsf{p}) \mapsto \varnothing \ * \ \mathsf{DataDescriptor(d)} \end{array} \right\}$$
$$\texttt{DefineOwnProperty(l, p, d)}$$
$$\left\{ \begin{array}{c} (\mathsf{l}, @class) \mapsto cls \ * \ cls \dot{\neq} \text{“String”} * \ cls \dot{\neq} \text{“Array”} \ * \\ (\mathsf{l}, @extensible) \mapsto \mathsf{true} \ * \ (\mathsf{l}, \mathsf{p}) \mapsto \mathsf{d} \ * \ \mathsf{DataDescriptor(d)} \ * \ \mathsf{ret} \doteq \mathsf{true} \end{array} \right\}$$

Given that the object `l` is not a `String` nor an `Array` object, is extensible, and does not contain the property `p`, the heap update is successful provided that `d` is a data descriptor. We use the abstraction $\mathsf{DataDescriptor}(d) \triangleq d \doteq [\text{“d”}, \_, \_, \_, \_]$ to denote a data descriptor d. The result of the `DefineOwnProperty` is `true`, as required by the standard. `DefineOwnProperty` calls `GetOwnProperty` and we highlighted the precondition of `GetOwnProperty` for this specification.

The `CanPut` internal function. `CanPut(l, p)` tells us if assigning to the property `p` of object `l` is allowed, which mostly depends on the extensibility of objects or the value of the writable attribute. We present two cases of `CanPut`:

$$\left\{ \ \mathsf{Pi}(\mathsf{l}, \mathsf{p}, \mathsf{undefined}, ls, vs_{cls}, vs_{pv}) \ * \ (\mathsf{l}, @extensible) \mapsto b \ \right\}$$
$$\texttt{CanPut(l, p)}$$
$$\left\{ \ Pre \ * \ \mathsf{ret} \doteq b \ \right\}$$

$$\left\{ \quad \mathsf{Pi}(\mathsf{l}, \mathsf{p}, d, \mathsf{l} :: l_p :: ls, vs_{cls}, vs_{pv}) \ * \ (\mathsf{l}, @extensible) \mapsto \mathsf{true} \ * \ \mathsf{DescW}(d, b) \quad \right\}$$
$$\texttt{CanPut(l, p)}$$
$$\left\{ \quad Pre \ * \ \mathsf{ret} \doteq b \quad \right\}$$

The first specification of `CanPut` states that if the property `p` is not defined in the prototype of the object `l`, then the result of `CanPut` is the value of the `"@extensible"` property of `l`. The second specification describes a case where the property `p` is defined in the prototype chain of the object `l`, but not the object itself, containing a data descriptor d, and the object `l` is extensible. The result in such a case is the writable attribute of the descriptor d. We use the abstraction $\mathsf{DescW}(d, b) \triangleq d \doteq [\text{“d”}, \_, b, \_, \_]$ to denote a data descriptor d which has the value b of the attribute `[W]`. The highlighted part of the preconditions in both cases corresponds to the precondition of `GetProperty`.

The `Put` internal function. Given an object `l`, a property `p`, and a value `w`, the `Put` internal function assigns the value `w` to the property `p` of the object `l`. In order to do that it relies on `CanPut`, `GetOwnProperty`, `GetProperty`, and `DefineOwnProperty`. Here, we present two specifications of `Put`, both relevant to the running example. We first describe the case in which we try to assign a value to a property of an object that has not been previously defined in the prototype chain of that object. As this case involves adding a new property to an object, we will succeed only if the object is extensible.

$$\left\{ \quad \mathsf{Pi}(\mathsf{l}, \mathsf{p}, \mathsf{undefined}, \mathsf{l} :: l_p :: ls, cls :: vs_{cls}, pv :: vs_{pv}) \ * \ cls \neq \text{“Array”} \ * \ (\mathsf{l}, @extensible) \mapsto \mathsf{true} \quad \right\}$$
$$\texttt{Put(l, p, v)}$$
$$\left\{ \begin{array}{c} cls \neq \text{“Array”} \ * \ (\mathsf{l}, @extensible) \mapsto \mathsf{true} \ * \\ \mathsf{Pi}(\mathsf{l}, \mathsf{p}, [\text{“d”}, \mathsf{v}, \mathsf{true}, \mathsf{true}, \mathsf{true}], [\mathsf{l}], [cls], [pv]) \ * \ (\mathsf{l}, @proto) \mapsto l_p \ * \\ \mathsf{Pi}(l_p, \mathsf{p}, \mathsf{undefined}, l_p :: ls, vs_{cls}, vs_{pv}) \ * \ \mathsf{ret} \doteq \mathsf{empty} \end{array} \right\}$$

The precondition states that the property `p` of object `l` is not defined in the prototype chain of `l`, and that `l` is extensible. This is the precondition of the first specification of `CanPut` given above. We also require `l` not to be an `Array` object, which comes from the specification of `DefineOwnProperty`. As the `DefineOwnProperty` internal function for JavaScript arrays differs from that for standard objects, `Put` for arrays has a different specification. The postcondition illustrates the subtlety of the Pi predicate. First, all assertions from the precondition except the Pi still hold. The return value is stated to be the `empty` value, as required by the standard. As for the Pi, by adding the property `p` to `l`, we break the prototype chain of the precondition into two: a single-element chain containing only `l`, where the property is now defined with the appropriate descriptor, and the rest of the original chain, in which the property is still undefined. This separation leaves a hanging resource $(\mathsf{l}, @proto) \mapsto l_p$, hidden in the original Pi but now stated explicitly.

We next give the specification of `Put` for the error case described in §3.2, when we are attempting to assign a value to a property of an object that is not yet defined in the object itself, but it is defined in its prototype chain and there it is not writable:

$$\{ \quad \mathsf{Pi}(\mathsf{l}, \mathsf{p}, d, \mathsf{l} :: l_p :: ls, vs_{cls}, vs_{pv}) \ * \ \mathsf{DescW}(d, \mathsf{false}) \ * \ (\mathsf{l}, @extensible) \mapsto \mathsf{true} \quad \}$$
$$\texttt{Put(l, p, w)}$$
$$\{ \quad Pre \ * \ \mathsf{isTypeError(err)} \quad \}$$

In the precondition, we have that `p` is present in the prototype chain of `l`, not in `l` itself, and we have that the associated data descriptor d is not writable. We also require of the object to be extensible.

This is precisely the precondition of the second specification of `CanPut` given above. The postcondition states that we have not affected any of the previously existing resources, and that we are throwing a JavaScript `TypeError`.

The `PutValue` internal function. We conclude by giving two specifications for `PutValue(w, v)`, which is, in a sense, the dual of `GetValue(v)`. It takes two parameters: values `w` and `v`. The value `w` is expected to be a reference whose base is not `undefined` or a primitive value, and an error is thrown otherwise. When `w` = [`"o"`/`"v"`, `l`, `p`] is a reference, the `Put` internal function is called.

Below, we present two specifications of `PutValue`. Both specifications consider the case where `w` is an object reference. The first specification describes the case in which we assign a value to a property of an extensible object that has not been previously defined in the prototype chain of that object. The second specification describes an error case, when we are attempting to assign a value to a property of an object that is not yet defined in the object itself, but it is defined in its prototype chain and there it is not writable. Both specifications are direct consequence of calling the `Put` internal function, the specifications of which are given above.

$$\left\{ \begin{array}{c} \mathsf{w} \doteq [\text{“o”}, l, p] \ * \\ \mathsf{Pi}(l, p, \mathsf{undefined}, l :: l_p :: ls, cls :: vs_{cls}, pv :: vs_{pv}) \ * \ cls \dot{\neq} \text{“Array”} \ * \ (l, @extensible) \mapsto \mathsf{true} \end{array} \right\}$$
$$\texttt{PutValue(w, v)}$$
$$\left\{ \begin{array}{c} \mathsf{w} \doteq [\text{“o”}, l, p] \ * \ cls \dot{\neq} \text{“Array”} \ * \ (l, @extensible) \mapsto \mathsf{true} \ * \\ \mathsf{Pi}(l, p, [\text{“d”}, \mathsf{v}, \mathsf{true}, \mathsf{true}, \mathsf{true}], [l], [cls], [pv]) \ * \ (l, @proto) \mapsto l_p \ * \\ \mathsf{Pi}(l_p, p, \mathsf{undefined}, l_p :: ls, vs_{cls}, vs_{pv}) \ * \ \mathsf{ret} \doteq \mathsf{empty} \end{array} \right\}$$

$$\left\{ \begin{array}{c} \mathsf{w} \doteq [\text{“o”}, l, p] \ * \\ \mathsf{Pi}(l, p, d, l :: l_p :: ls, vs_{cls}, vs_{pv}) \ * \ \mathsf{DescW}(d, \mathsf{false}) \ * \ (l, @extensible) \mapsto \mathsf{true} \end{array} \right\}$$
$$\texttt{PutValue(w, v)}$$
$$\{ \quad Pre \ * \ \mathsf{isTypeError(err)} \quad \}$$

Higher-Order Internal Functions. Since JSIL logic does not yet support higher-order reasoning, we cannot specify internal functions that use higher order. In such a case, we symbolically execute the body of an internal function, instead of using its specification. An example of an internal function that has higher-order cases is the `ToString` function (see §3.1.2), whose call graph is shown in Figure 7.3.

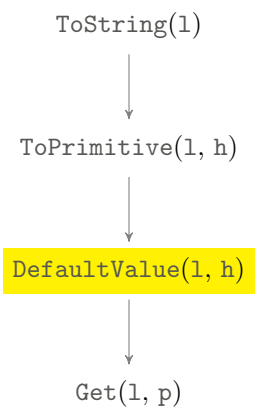


Figure 7.3.: Call graph for `ToString`

When `ToString` is given an object `l`, it calls another internal function, `ToPrimitive`, passing as parameters the object `l` and a hint `"String"`. `ToPrimitive` then calls the `DefaultValue` internal function with the same parameters. `DefaultValue`, when given the hint `"String"`, first checks if the given object `l` has the function `toString` in its prototype chain, using the internal function `Get`. If it does, the result of `DefaultValue` is the result of calling that `toString` function, as long as that result is a primitive value. Otherwise, if a given object `l` has the function `valueOf` in its prototype chain, the result of `DefaultValue` is the result of calling that `valueOf` function, as long as that result is a primitive value. If none of the previous holds, a type error is thrown. `DefaultValue` is a higher-order function in the sense that its result depends on other functions that are not known in advance. Therefore, we are not able to specify it. By extension, we are not able to specify the cases of the `ToString` internal function in which it is called on an object.

Summary. We provide reference implementations of all JavaScript internal functions. These implementations are substantially tested via our testing of the JS-2-JSIL compiler against Test262, discussed in §5.3. We give axiomatic specifications for all JavaScript internal functions and annotate their reference implementations with fold/unfold directives, as demonstrated in Figure 7.2. Here, we illustrated our specifications of JavaScript internal functions using `GetValue` and `PutValue`, as well as a number of other internal functions that their implementations depend on. In total, we have 186 specifications. These specifications are non-trivial and the underlying code makes extensive use of the dynamic features of JSIL, as the internal functions are written in a general way in the standard. Using JSIL Verify, we verify that our JSIL implementations of JavaScript internal functions satisfy their axiomatic specifications. These results can be interpreted in two ways: they provide validation of the JSIL axiomatic specifications, as the implementations closely follow the standard and are well tested; and, at the same time, they provide further validation of the implementations of the internal functions. JSIL Verify verifies all 186 specifications of the JavaScript internal functions in 5.1 seconds.[1]

[1]For verification, we use a machine with an Intel Core i7-4980HQ CPU 2.80 GHz and DDR3 RAM 16GB.

8. JavaScript Verification

Given a program whose functions are annotated with specifications in the form of pre- and post-conditions written in JS Logic, JaVerT verifies whether or not the code of each function satisfies its specification. To specify JavaScript programs, we need to provide assertions that fully capture the key heap structures of JavaScript, such as property descriptors, prototype chains for modelling inheritance, the variable store emulated in the heap using scope chains, and function closures. We first introduce JS Logic assertions and specifications (§8.1). As JavaScript heaps are identical to JSIL heaps, JS assertions has many similarities to JSIL assertions. Next, we provide a translation from JS Logic to JSIL Logic, and prove correct the translation of assertions and specifications (§8.2), allowing us to lift JSIL verification to JavaScript verification. We would like the user of JaVerT to be able to specify JavaScript programs clearly and concisely, with only minimal knowledge of JavaScript internals. To that end, we build a number of predicates on top of JS Logic that capture the common JavaScript heap structures (§8.3). We illustrate how to use these predicates by specifying the priority queue library given in the running example (§8.4). For such libraries, we want to give specifications that ensure *prototype safety* of library operations, in that they describe the conditions under which these operations exhibit the desired behaviour.

JaVerT is available online at [65], where the user can verify the running example of the thesis as well as other simple JavaScript programs.

8.1. JS Logic

We present JS Logic, the JavaScript assertion language that targets full ES5 Strict heaps, and formally introduce JavaScript specifications.

JS Logic Assertions. JS Logic assertions are mostly standard and are given in Figure 8.1. The difference with respect to [30] is that we do not use the sepish connective, introduced to describe overlapping.

JS logical values, $\mathrm{V} \in \mathcal{V}^L_{\mathtt{JS}}$, contain: JS heap values, ω; sets of JavaScript heap values, $\omega_{\mathtt{set}}$; and the special value $\varnothing$. JS logical values is a subset of JSIL logic values. JS logical expressions, $\mathrm{E} \in \mathcal{E}^L_{\mathtt{JS}}$, include JSIL logical expressions and have two additional special logical expressions, `this` and `sc`, referring respectively to the current scope chain and the current `this` object. We note that program variables `x` only include the formal parameters of the functions. Finally, as ES5 Strict heaps are by design a proper subset of JSIL heaps, we have that JS Logic assertions, $P, Q \in \mathcal{AS}_{\mathtt{JS}}$, coincide with JSIL Logic assertions.

In Figure 8.2, we give the satisfiability relation for JavaScript assertions. The satisfiability relation for JS assertions has the form: $H, \rho, L, v_t, \epsilon \models P$, where: H is an abstract JS heap; ρ is a JS variable store; L is the current scope chain; v_t is the binding of the `this` value; and ϵ is a logical environment (a mapping from logical variables to logical values).

$$
\begin{array}{rll}
\textsc{Logical Values} : \mathrm{V} \in \mathcal{V}^L_{\mathtt{JS}} & \triangleq & \omega \mid \omega_{\mathtt{set}} \mid \varnothing \\
\textsc{Logical Expressions} : \mathrm{E} \in \mathcal{E}^L_{\mathtt{JS}} & \triangleq & \mathrm{V} \mid \mathtt{x} \mid \mathrm{X} \mid \ominus\, \mathrm{E} \mid \mathrm{E} \oplus \mathrm{E} \mid \mathsf{this} \mid \mathtt{sc}
\end{array}
$$

$$
\begin{array}{rll}
\textsc{JS Assertions} : P \in \mathcal{AS}_{\mathtt{JS}} & \triangleq & \mathrm{true} \mid \mathrm{false} \mid P \wedge P \mid \neg P \mid \qquad \textsc{Classical} \\
& & \exists \mathrm{X}.P \mid \qquad \textsc{Quantification} \\
& & \mathrm{E} = \mathrm{E} \mid \mathrm{E} \leq \mathrm{E} \mid \mathrm{E} < \mathrm{E} \mid \qquad \textsc{Equalities} \\
& & \mathsf{emp} \mid P * P \mid \qquad \textsc{Separation Logic} \\
& & (\mathrm{E}, \mathrm{E}) \mapsto \mathrm{E} \mid \qquad \textsc{JavaScript Cell} \\
& & \mathsf{types}(\overline{\mathrm{E} : \tau}) \mid \mathsf{emptyProps}(\mathrm{E} \mid \mathrm{E}) \qquad \textsc{Predicates}
\end{array}
$$

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

Figure 8.1.: JS Logic Assertions, where $\omega \in \mathcal{V}^h_{\mathtt{JS}}$ (Figure 3.11).

Logical Expressions:

$$
\begin{array}{lcl}
[\![\mathrm{V}]\!]^{\epsilon}_{\rho,v_t,L} & \triangleq & \mathrm{V} \\
[\![\mathtt{x}]\!]^{\epsilon}_{\rho,v_t,L} & \triangleq & \rho(\mathtt{x}) \\
[\![\mathrm{X}]\!]^{\epsilon}_{\rho,v_t,L} & \triangleq & \epsilon(\mathrm{X}) \\
[\![\ominus\, \mathrm{E}]\!]^{\epsilon}_{\rho,v_t,L} & \triangleq & \ominus([\![\mathrm{E}]\!]^{\epsilon}_{\rho,v_t,L}) \\
[\![\mathrm{E}_1 \oplus \mathrm{E}_2]\!]^{\epsilon}_{\rho,v_t,L} & \triangleq & \oplus([\![\mathrm{E}_1]\!]^{\epsilon}_{\rho,v_t,L}, [\![\mathrm{E}_2]\!]^{\epsilon}_{\rho,v_t,L}) \\
[\![\mathsf{this}]\!]^{\epsilon}_{\rho,v_t,L} & \triangleq & v_t \\
[\![\mathtt{sc}]\!]^{\epsilon}_{\rho,v_t,L} & \triangleq & L
\end{array}
$$

Assertions:

$$
\begin{array}{ll}
H, \rho, L, v_t, \epsilon \models \mathrm{true} & \Leftrightarrow \text{always} \\
H, \rho, L, v_t, \epsilon \models \mathrm{false} & \Leftrightarrow \text{never} \\
H, \rho, L, v_t, \epsilon \models P_1 \wedge P_2 & \Leftrightarrow H, \rho, L, v_t, \epsilon \models P_1 \wedge H, \rho, L, v_t, \epsilon \models P_2 \\
H, \rho, L, v_t, \epsilon \models \neg P & \Leftrightarrow H, \rho, L, v_t, \epsilon \not\models P \\
H, \rho, L, v_t, \epsilon \models \mathrm{E}_1 = \mathrm{E}_2 & \Leftrightarrow H = \mathsf{emp} \wedge [\![\mathrm{E}_1]\!]^{\epsilon}_{\rho,v_t,L} = [\![\mathrm{E}_2]\!]^{\epsilon}_{\rho,v_t,L} \\
H, \rho, L, v_t, \epsilon \models \mathrm{E}_1 \leq \mathrm{E}_2 & \Leftrightarrow H = \mathsf{emp} \wedge [\![\mathrm{E}_1]\!]^{\epsilon}_{\rho,v_t,L} \leq [\![\mathrm{E}_2]\!]^{\epsilon}_{\rho,v_t,L} \\
H, \rho, L, v_t, \epsilon \models \mathrm{E}_1 < \mathrm{E}_2 & \Leftrightarrow H = \mathsf{emp} \wedge [\![\mathrm{E}_1]\!]^{\epsilon}_{\rho,v_t,L} < [\![\mathrm{E}_2]\!]^{\epsilon}_{\rho,v_t,L} \\
H, \rho, L, v_t, \epsilon \models \mathsf{emp} & \Leftrightarrow H = \mathsf{emp} \\
H, \rho, L, v_t, \epsilon \models (\mathrm{E}_1, \mathrm{E}_2) \mapsto \mathrm{E}_3 & \Leftrightarrow H = ([\![\mathrm{E}_1]\!]^{\epsilon}_{\rho,v_t,L}, [\![\mathrm{E}_2]\!]^{\epsilon}_{\rho,v_t,L}) \mapsto [\![\mathrm{E}_3]\!]^{\epsilon}_{\rho,v_t,L} \\
H, \rho, L, v_t, \epsilon \models P_1 * P_2 & \Leftrightarrow \exists H_1, H_2.\, H = H_1 \uplus H_2 \wedge \\
& \quad (H_1, \rho, L, v_t, \epsilon \models P_1) \wedge (H_2, \rho, L, v_t, \epsilon \models P_2) \\
H, \rho, L, v_t, \epsilon \models \exists \mathrm{X}.P & \Leftrightarrow \exists \mathrm{V} \in \mathcal{V}^L_{\mathtt{JS}}.\, H, \rho, L, v_t, \epsilon[\mathrm{X} \mapsto \mathrm{V}] \models P \\
H, \rho, L, v_t, \epsilon \models \mathsf{types}(\overline{\mathrm{E} : \tau}) & \Leftrightarrow H = \mathsf{emp} \wedge \forall (\mathrm{E}, \tau) \in \overline{\mathrm{E} : \tau}.\, \mathsf{TypeOf}([\![\mathrm{E}]\!]^{\epsilon}_{\rho,v_t,L}) = \tau \\
H, \rho, L, v_t, \epsilon \models \mathsf{emptyProps}(\mathrm{E}_1 \mid \mathrm{E}_2) & \Leftrightarrow H = \biguplus_{p \notin \{[\![\mathrm{E}_2]\!]^{\epsilon}_{\rho,v_t,L}\}} (([\![\mathrm{E}_1]\!]^{\epsilon}_{\rho,v_t,L}, p) \mapsto \varnothing)
\end{array}
$$

Figure 8.2.: Semantics of JS Logical Expressions and Assertions

JS Specifications. JaVerT specifications have the form $\{P\}\, m(\overline{x})\, \{Q\}$, where P and Q are the pre- and postconditions of the function with identifier m, and $\overline{x}$ its list of formal parameters. We think of global code as a function with identifier `main`. Each specification is associated with a return mode $\mathit{fl} \in \{\mathsf{nm}, \mathsf{er}\}$, indicating if the function returns normally or with an error. If it returns normally, then its return value can be accessed via a dedicated variable `xret`, and `xerr` otherwise. Intuitively, a specification $\{P\}\, m(\overline{x})\, \{Q\}$ for mode fl is valid for a given JavaScript program $\wp$, if $\wp$ contains a function with identifier m and "whenever m is executed in a state satisfying P, then, if it terminates, it does so in a state satisfying Q, with return mode fl".

Definition 8.1 (Validity of JS Logic Specifications)**.** *A JS Logic specification* $\{P\}\, m(\overline{x})\, \{Q\}$ *for return mode* fl *is valid w.r.t. a JavaScript program* $\wp$*, written* $\wp, \mathrm{fl} \vDash \{P\}\, m(\overline{x})\, \{Q\}$*, if and only if, for all logical contexts* $(H, L, v_t, \rho, \epsilon)$*, heaps* h_f*, flags* fl'*, and JS values* v*, where* $\rho(\overline{x}) = \overline{v}$*, it holds that:*

$$H, \rho, L, v_t, \epsilon \models P \ \wedge \ \wp, L, v_t \vdash \langle \lfloor H \rfloor, m(\overline{x}, \overline{v}) \rangle \Downarrow_m \langle h_f, \mathsf{out_{JS}}(\mathrm{fl}', v) \rangle \implies$$
$$\mathrm{fl} = \mathrm{fl}' \ \wedge \ \exists H_f \,.\, H_f, \rho, L, v_t, \epsilon \models Q \ \wedge \ \lfloor H_f \rfloor = h_f$$

Recall that we use the notation $\lfloor H \rfloor$ to denote the concrete heap obtained by restricting the abstract heap H to the elements of its domain not mapped to $\varnothing$.

8.2. JS-2-JSIL: Logic Translator

JaVerT verifies programs annotated with JS Logic annotations. The JSIL Logic Translator translates these annotations to equivalent annotations in JSIL Logic, and then integrates them into the compiled JSIL code. It also automatically inserts additional fold/unfold annotations for the Pi predicate, as they are required by some of the internal functions.

We provide a translation from JS Logic assertions to JSIL logic assertions, and prove that translation correct (Figure 8.3). Given a correct compiler from ES5 Strict to JSIL, we prove JS-2-JSIL Logic correspondence, stating that a JavaScript specification is valid if and only if its translated JSIL specification is valid. This allows us to lift JSIL verification to JavaScript verification.

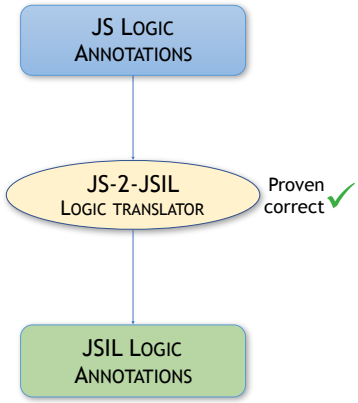


Figure 8.3.: The JS-2-JSIL Logic Translator

JS-2-JSIL Translation of Assertions. There is a strong correspondence between JavaScript and JSIL at the level of the logics. JSIL logical values subsumes JS logical values. JSIL logical expressions coincide with JS logical expressions, except that they do not contain the special logical values `sc` and this. Finally, as ES5 Strict heaps are by design a proper subset of JSIL heaps, we have that JSIL Logic assertions coincide with JS Logic assertions.

Translating JS Logic assertions to JSIL Logic assertions amounts to replacing the occurrences of the `sc` and this special logical values of JS Logic with the variables $\mathtt{x}_{sc}$ and $\mathtt{x}_{\mathsf{this}}$ of JSIL logic, which hold their associated values at the JSIL level. The translation of a JS Logic assertion P to a JSIL Logic assertion is denoted by $\mathcal{T}_a(P)$. In Figure 8.4, we give the complete translation from JS Logic assertions to JSIL Logic assertions.

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

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

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

$$\begin{array}{lcl}
\mathcal{T}_e(\mathrm{V}) & \triangleq & \mathrm{V} \\
\mathcal{T}_e(\mathtt{x}) & \triangleq & \mathtt{x} \\
\mathcal{T}_e(\mathrm{X}) & \triangleq & \mathrm{X} \\
\mathcal{T}_e(\ominus \mathrm{E}) & \triangleq & \ominus \mathcal{T}_e(\mathrm{E}) \\
\mathcal{T}_e(\mathrm{E}_1 \oplus \mathrm{E}_2) & \triangleq & \mathcal{T}_e(\mathrm{E}_1) \oplus \mathcal{T}_e(\mathrm{E}_2) \\
\mathcal{T}_e(\mathsf{this}) & \triangleq & \mathtt{x}_{\mathsf{this}} \\
\mathcal{T}_e(\mathtt{sc}) & \triangleq & \mathtt{x}_{sc}
\end{array}$$

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

$$\begin{array}{lcl}
\mathcal{T}_a(\mathrm{true}) & \triangleq & \mathrm{true} \\
\mathcal{T}_a(\mathrm{false}) & \triangleq & \mathrm{false} \\
\mathcal{T}_a(\neg P) & \triangleq & \neg \mathcal{T}_a(P) \\
\mathcal{T}_a(P_1 \wedge P_2) & \triangleq & \mathcal{T}_a(P_1) \wedge \mathcal{T}_a(P_2) \\
\mathcal{T}_a(\exists X.P) & \triangleq & \exists X. \mathcal{T}_a(P) \\
\mathcal{T}_a(\mathrm{E}_1 = \mathrm{E}_2) & \triangleq & \mathcal{T}_a(\mathrm{E}_1) = \mathcal{T}_a(\mathrm{E}_2) \\
\mathcal{T}_a(\mathrm{E}_1 \leq \mathrm{E}_2) & \triangleq & \mathcal{T}_a(\mathrm{E}_1) \leq \mathcal{T}_a(\mathrm{E}_2) \\
\mathcal{T}_a(\mathrm{E}_1 < \mathrm{E}_2) & \triangleq & \mathcal{T}_a(\mathrm{E}_1) < \mathcal{T}_a(\mathrm{E}_2) \\
\mathcal{T}_a(\mathsf{emp}) & \triangleq & \mathsf{emp} \\
\mathcal{T}_a(P_1 * P_2) & \triangleq & \mathcal{T}_a(P_1) * \mathcal{T}_a(P_2) \\
\mathcal{T}_a((E_1, E_2) \mapsto E_3) & \triangleq & (\mathcal{T}_e(E_1), \mathcal{T}_e(E_2)) \mapsto \mathcal{T}_e(E_3) \\
\mathcal{T}_a(\mathsf{emptyProps}(E_1 \mid E_2)) & \triangleq & \mathsf{emptyProps}(\mathcal{T}_e(E_1) \mid \mathcal{T}_e(E_2))
\end{array}$$

Figure 8.4.: Translation from JS Logical Assertions to JSIL Logical Assertions. $\mathcal{E}nv_{\texttt{JS}}$, $\mathcal{E}^L_{\texttt{JS}}$, $\mathcal{AS}_{\texttt{JS}}$ are logical environments, logical expressions, and assertions of JavaScript (Figure 8.1). $\mathcal{E}nv_{\texttt{JSIL}}$, $\mathcal{E}^L_{\texttt{JSIL}}$, $\mathcal{AS}_{\texttt{JSIL}}$ are logical environments, logical expressions, and assertions of JSIL (Figure 6.2).

Given how close the semantics of JS and JSIL assertions are, it immediately follows that:

Theorem 8.1 (Assertion translation correctness)**.** *For any assertion P, abstract heap H, variable store ρ, logical environment ϵ, value v_t, and scope chain L, it holds that:*

$$H, \rho, L, v_t, \epsilon \models P \iff H, \rho[\mathtt{x}_{sc} \mapsto L, \mathtt{x}_{\mathit{this}} \mapsto v_t], \epsilon \models \mathcal{T}_a(P)$$

.

Proof. Given in Appendix D. □

JS-2-JSIL Logic Correspondence. To be able to state the correspondence theorem, we lift the translation of assertions to specifications: $\mathcal{T}(\{P\}\, m(\overline{x})\, \{Q\}) = \{\mathcal{T}_a(P)\}\, m(\mathtt{x}_{sc}, \mathtt{x}_{\mathsf{this}}, \overline{x})\, \{\mathcal{T}_a(Q)\}$. Theorem 8.2 states that under the assumption of a correct compiler, a JavaScript specification is valid if and only if its translated JSIL specification is valid.

Theorem 8.2 (JS-2-JSIL Logic correspondence)**.** *Given a* correct *JS-2-JSIL compiler, $\overline{\mathcal{C}}$ (Theorem 5.1), for any JavaScript program $\wp$, return mode* fl*, and JS specification $\{P\}\, m(\overline{x})\, \{Q\}$, it holds that:*

$$\wp, \mathrm{fl} \vDash \{P\}\, m(\overline{x})\, \{Q\} \iff \overline{\mathcal{C}}(\wp), \mathrm{fl} \vDash \mathcal{T}(\{P\}\, m(\overline{x})\, \{Q\})$$

Proof. [$\Longrightarrow$] Assuming that $\{P\}\, m(\overline{x})\, \{Q\}$ is valid for the given return mode fl, we need to prove that $\{\mathcal{T}_a(P)\}\, m(\mathtt{x}_{sc}, \mathtt{x}_{this}, \overline{x})\, \{\mathcal{T}_a(Q)\}$ is also valid. To this end, we need to show that for every JSIL logical context H, ρ, ϵ such that $H, \rho, \epsilon \models \mathcal{T}_a(P)$ and $\overline{\mathcal{C}}(\wp) \vdash \langle \lfloor H \rfloor, \rho, -, 0\rangle \Downarrow_m \langle h_f, \rho_f, \mathit{fl}'\langle \mathtt{v}\rangle\rangle$, for some $h_f, \rho_f, \mathtt{v}, \overline{v}, \rho'$, where $\rho' = \emptyset[x_i \mapsto v_i|_{i=1}^n], \rho = \rho'[\mathtt{x}_{sc} \mapsto L, \mathtt{x}_{\mathsf{this}} \mapsto v_t]$, it holds that $\mathit{fl}' = \mathit{fl}$ and there exists an abstract heap H_f such that $\lfloor H_f \rfloor = h_f$ and $H_f, \rho_f, \epsilon \models \mathcal{T}_a(Q)$. That is, we assume:

- $\wp, \mathit{fl} \vDash \{P\}\, m(\overline{x})\, \{Q\}$ (**H1**)
- $H, \rho, \epsilon \models \mathcal{T}_a(P)$ (**H2**)
- $\overline{\mathcal{C}}(\wp) \vdash \langle \lfloor H \rfloor, \rho, -, 0\rangle \Downarrow_m \langle h_f, \rho_f, \mathit{fl}'\langle \mathtt{v}\rangle\rangle$, for some $h_f, \rho_f, \mathtt{v}$ (**H3**)

Our goal is to show that there is an abstract heap H_f such that:

- $\mathit{fl} = \mathit{fl}'$ (**G1**)
- $H_f, \rho_f, \epsilon \models \mathcal{T}_a(Q)$ (**G2**)
- $\lfloor H_f \rfloor = h_f$ (**G3**)

1. From **H2**, we conclude, using Theorem 8.1, $H, \rho', L, v_t, \epsilon \models P$ (**I1**).
2. From **H3**, we conclude, using Theorem 5.1 (*compiler correctness*), that $\wp, L, v_t \vdash \langle \lfloor H \rfloor, m(\overline{x}, \overline{v})\rangle \Downarrow_m \langle h_f, \mathsf{out}_{\mathtt{JS}}(\mathit{fl}', \mathtt{v})\rangle$ (**I2**).
3. From **H1** (recall Definition 8.1), **I1**, **I2**, we conclude, that $\mathit{fl} = \mathit{fl}'$ (**G1**) and there exists an abstract heap $\hat{H}_f$ such that $\hat{H}_f, \rho', L, v_t, \epsilon \models Q$ (**I3**) and $\lfloor \hat{H}_f \rfloor = h_f$ (**I4**).
4. We take $H_f = \hat{H}_f$ (**I5**).

5. From **I3** and **I5**, we conclude, using Theorem 8.1, $H_f, \rho, \epsilon \models \mathcal{T}_a(Q)$. Noting that $\rho_f \geq \rho$, it follows that $H_f, \rho_f, \epsilon \models \mathcal{T}_a(Q)$ (**G2**).

6. From **I4** and **I5** we get $\lfloor H_f \rfloor = h_f$ (**G3**).

[$\Longleftarrow$] Assuming that $\{\mathcal{T}_a(P)\}\, m(\mathtt{x}_{sc}, \mathtt{x}_{this}, \overline{x})\, \{\mathcal{T}_a(Q)\}$ is valid for the given return mode fl, we need to prove that $\{P\}\, m(\overline{x})\, \{Q\}$ is also valid. To this end, we need to show that for every JS logical context $H, \rho', L, v_t, \epsilon$ such that $H, \rho', L, v_t, \epsilon \models P$ and $\wp, L, v_t \vdash \langle \lfloor H \rfloor, m(\overline{x}, \overline{v}) \rangle \Downarrow_m \langle h_f, \mathsf{out}_{\mathtt{JS}}(\mathit{fl}', \mathtt{v}) \rangle$, for some $h_f, \mathtt{v}, \overline{v}$, where $\rho' = \emptyset[x_i \mapsto v_i|_{i=1}^n]$, it holds that $\mathit{fl}' = \mathit{fl}$ and there exists an abstract heap H_f such that $\lfloor H_f \rfloor = h_f$ and $H_f, \rho', L, v_t, \epsilon \models Q$. That is, we assume:

- $\overline{\mathcal{C}}(\wp), \mathit{fl} \vDash \{\mathcal{T}_a(P)\}\, m(\mathtt{x}_{sc}, \mathtt{x}_{this}, \overline{x})\, \{\mathcal{T}_a(Q)\}$ (**H1**)
- $H, \rho', L, v_t, \epsilon \models P$ (**H2**)
- $\wp, L, v_t \vdash \langle \lfloor H \rfloor, m(\overline{x}, \overline{v}) \rangle \Downarrow_m \langle h_f, \mathsf{out}_{\mathtt{JS}}(\mathit{fl}', \mathtt{v}) \rangle$, for some $h_f, \mathtt{v}, \overline{v}$ (**H3**)

Our goal is to show that there is an abstract heap H_f such that:

- $\mathit{fl} = \mathit{fl}'$ (**G1**)
- $H_f, \rho', L, v_t, \epsilon \models Q$ (**G2**)
- $\lfloor H_f \rfloor = h_f$ (**G3**)

1. From **H2**, we conclude, using Theorem 8.1, $H, \rho, \epsilon \models \mathcal{T}_a(P)$, where $\rho = \rho'[\mathtt{x}_{sc} \mapsto L, \mathtt{x}_{\mathsf{this}} \mapsto v_t]$ (**I1**).

2. From **H3**, we conclude, using Theorem 5.1 (*compiler correctness*), that there exists ρ_f, such that $\overline{\mathcal{C}}(\wp) \vdash \langle \lfloor H \rfloor, \rho, -, 0 \rangle \Downarrow_m \langle h_f, \rho_f, \mathit{fl}'\langle \mathtt{v} \rangle \rangle$ (**I2**).

3. From **H1** (recall Definition 6.1), **I1**, **I2**, we conclude, that $\mathit{fl} = \mathit{fl}'$ (**G1**) and there exists an abstract heap $\hat{H}_f$ such that $\hat{H}_f, \rho_f, \epsilon \models \mathcal{T}_a(Q)$ (**I3**) and $\lfloor \hat{H}_f \rfloor = h_f$ (**I4**).

4. We take $H_f = \hat{H}_f$ (**I5**).

5. From **I3** and **I5**, we conclude, using Theorem 8.1, $H_f, \rho'_f, L, v_t, \epsilon \models Q$, where $\rho_f = \rho'_f[\mathtt{x}_{sc} \mapsto L, \mathtt{x}_{\mathsf{this}} \mapsto v_t]$. Noting that $\rho_f \geq \rho > \rho'$, and the fact that JS assertions can only mention the formal parameters of the functions, it follows that $H_f, \rho', L, v_t, \epsilon \models Q$ (**G2**).

6. From **I4** and **I5** we get $\lfloor H_f \rfloor = h_f$ (**G3**).

□

Lifting JSIL Verification to JavaScript Verification. An immediate consequence of all of the obtained theoretical results is that we can lift JSIL verification back to JavaScript verification. Let us expand on what that means. To verify a JavaScript program $\wp$, we need to show the validity of its specifications for all functions m: $\wp, \mathit{fl} \vDash \{P\}\, m(\overline{x})\, \{Q\}$. To do that, we: compile $\wp$ to a JSIL program $\overline{\mathcal{C}}(\wp)$ using the JS-2-JSIL compiler (§5); and translate its specifications $\{P\}\, m(\overline{x})\, \{Q\}$ using the JS-2-JSIL logic translator to obtain a specification environment $\mathtt{S}$, where each specification has

been translated to the form $\mathcal{T}(\{P\}\, m(\overline{x})\, \{Q\})$. Next, we move to JSIL-land. In §6, we presented JSIL verification. Given the JSIL program $\overline{\mathcal{C}}(\wp)$ and the specification environment $\mathtt{S}$, we construct a well-formed proof candidate $\overline{\mathcal{C}}(\wp), \mathtt{S} \vdash \mathtt{pd}$. From the soundness of JSIL logic (Theorem 6.1), we get that for all specifications in the specification environment: $\overline{\mathcal{C}}(\wp), \mathit{fl} \vDash \mathcal{T}(\{P\}\, m(\overline{x})\, \{Q\})$. Finally, we can go back to JavaScript using the JS-2-JSIL logic correspondence (Theorem 8.2), obtaining the desired $\wp, \mathit{fl} \vDash \{P\}\, m(\overline{x})\, \{Q\}$.

Additional Annotations for JavaScript Internal Functions. Compiled JSIL code contains procedure calls to the reference implementations of JavaScript internal functions. As we have seen in §7.2, some of the internal functions, for example, `GetValue` and `PutValue`, use the Pi predicate in their specifications. During symbolic execution of the JSIL code, the Pi predicate needs to be folded for the precondition to hold. To account for this, JS-2-JSIL automatically inserts annotations for folding the appropriate Pi prior to such calls and for unfolding it afterwards. This is illustrated in Figure 8.5 for one `PutValue` call in the compiled JSIL code of the `enqueue` in Figure 5.9. This way, we ensure that prototype chains are always unfolded and, therefore, we do not require the sepish connective of [30].

```
[*fold Pi (x_21, x_32_v, _, _, _)*]
71  x_34 := "i__putValue"(x_21, x_32_v) with perr; /* this._head = n.insertToQueue(this._head); */
[*unfold Pi *]
```

Figure 8.5.: Automatic Fold/Unfold Annotations

8.3. Basic JS Logic Predicates

We start by introducing the basic predicates for describing JavaScript object properties, function objects, and the JS initial heap. These predicates constitute the building blocks of our specifications and will be used in the specification of the running example in the following sections.

Objects and Object Properties. Standard JavaScript objects always have the three internal properties, `@proto`, `@class`, and `@extensible`, which respectively denote the prototype of the object, the class of the object, and whether the object can be extended with new properties. When JavaScript developers create and manipulate objects, they normally care only about their prototypes, and do not think of the internal properties `@class` and `@extensible`. Hence, JaVerT has a built-in predicate $\mathsf{JSObject}(o, p)$ which states that object o has prototype p, and its internal properties `@class` and `@extensible` have their default values, `"Object"` and `true`. JaVerT also provides a general version for objects, the $\mathsf{JSObjectGen}(o, p, c, e)$ predicate, which allows the user to specify the values of `@class` and `@extensible` as c and e.

$$\mathsf{JSObjectGen}(o, p, c, e) \triangleq (o, @\mathit{proto}) \mapsto p \ * \ (o, @\mathit{class}) \mapsto c \ * \ (o, @\mathit{extensible}) \mapsto e$$

$$\mathsf{JSObject}(o, p) \triangleq \mathsf{JSObjectGen}(o, p, \text{“Object”}, \mathsf{true})$$

JaVerT also has built-in predicates for describing named object properties. Named properties are associated with descriptors. Most of the time, JavaScript developers do not need to think in terms of descriptors. Usually, it is enough to think about data properties as having values and accessor properties as having setters and getters. Hence, JaVerT provides the $\mathsf{DataProp}(o, p, v)$ predicate

which states that the property p of object o holds a data descriptor with value v and all other attributes set to `true`. Similarly, the $\mathsf{AccessorProp}(o, p, g, s)$ predicate states that the property p of object o holds an accessor descriptor with getter g, setter s and all other attributes set to `true`. If a user needs to reason about other attributes of the descriptors, JaVerT has more general predicates: $\mathsf{DataPropGen}(o, p, v, w, e, c)$ allows the user to specify the values of the remaining attributes, writable, enumerable and configurable; $\mathsf{AccessorPropGen}(o, p, g, s, e, c)$ allows the user to specify the values of the remaining attributes, enumerable and configurable.

$$\begin{aligned}
\mathsf{DataPropGen}(o, p, v, w, e, c) &\triangleq (o, p) \mapsto [\text{“d”}, v, w, e, c] \\
\mathsf{DataProp}(o, p, v) &\triangleq \mathsf{DataPropGen}(o, p, v, \mathsf{true}, \mathsf{true}, \mathsf{true}) \\
\mathsf{AccessorPropGen}(o, p, g, s, e, c) &\triangleq (o, p) \mapsto [\text{“a”}, g, s, e, c] \\
\mathsf{AccessorProp}(o, p, g, s) &\triangleq \mathsf{AccessorPropGen}(o, p, g, s, \mathsf{true}, \mathsf{true})
\end{aligned}$$

Function Objects. When we define a JavaScript function, a new extensible function object is created: its prototype is the built-in `Function.prototype` object, denoted by l_{fp}; and its class is `"Function"`. The function object also stores its unique identifier and the scope in which it was defined. When we reason about a function object, it is enough to know its unique identifier and its scope. JaVerT offers the $\mathsf{FunctionObject}(o, m, sc)$ predicate, which describes the function object o, whose internal properties @*code* and @*scope* have values given by the function identifier, m, and the scope chain, sc, respectively. All other properties are the default ones.

$$\mathsf{FunctionObject}(o, m, sc) \triangleq \mathsf{JSObjectGen}(o, l_{fp}, \text{“Function”}, \mathsf{true}) * (o, @code) \mapsto m * (o, @scope) \mapsto sc$$

We have to expose the scope parameter sc, if we need to reason about variables defined in enclosing functions. However, the user never needs to explicitly describe the scope chain, and it is always enough for them to use a logical variable. We will see how it is used in the following section.

JS Initial Heap. JaVerT provides predicates that describe the built-in library objects. These predicates come in two flavours: frozen, where changes to the target object are not allowed; and open, where changes are allowed. For instance, $\mathsf{ObjProtoF}()$ and $\mathsf{ObjProto}()$ describe the frozen and open `Object.prototype`, respectively.

8.4. Specification of the Running Example

JavaScript developers rely on prototype-based inheritance to emulate the standard class-based inheritance mechanism of static OO languages when implementing JavaScript libraries. However, as JavaScript objects are extensible, it is possible to break the functionality of such libraries by adding properties either to the constructed objects or to their prototype chains. This makes the specifications of these libraries challenging as they not only need to capture the resources that must be present in the heap, but also the resources that *must not be* present in the heap if the library code is to run as intended. Moreover, JavaScript does not provide full encapsulation, forcing developers to use ad

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

Figure 8.6.: A Reminder of the Running Example

hoc features, such as the underscore prefix, to denote properties that are intended to be private. We would like the specifications to ensure that private properties are not used outside library code.

We highlight a general methodology for specifying that JavaScript libraries behave as intended. We discuss two important aspects of specifying JavaScript libraries: capturing *prototype safety* and enforcing *encapsulation*. In this section, we first show some examples of how a user can misuse the library. Next, we provide the specification of the Priority Queue Library that ensures that the library behaves as intended. Finally, we show that the given specification does not allow us to verify client code that misuses the library.

8.4.1. Client Code Misusing the Library

In order to guarantee that the Priority Queue library (Figure 8.6) works as intended, we must make sure that: **(1)** every time one calls `enqueue` or `dequeue` on a priority queue object, one reaches the appropriate functions defined within its prototype; **(2)** one can always successfully call `enqueue` or `dequeue` on a priority queue object: `enqueue` inserts an element in the queue in the right place, while `dequeue` retrieves the element with the highest priority or throws an error if the queue is empty; **(3)** one can always successfully construct a priority queue using the `PriorityQueue` constructor; and **(4)** one can always retrieve the value of a highest priority previously inserted into a priority queue. In Figure 8.7, we show how a user can misuse the library, effectively breaking **(1)**-**(4)**. To break **(1)**, one simply has to override `enqueue` or `dequeue` on the constructed priority queue object (Client 1). To break **(2)**, it suffices to assign an arbitrary *non-writable* value to `"pri"` in `Node.prototype` (Client 2). By doing that, a call to `enqueue` will fail, as a construction of a new node to insert into the queue will fail (recall the discussion in §3.2). To break **(3)**, one can define a property `"_head"` containing an arbitrary *non-writable* value in `Object.prototype` (Client 3). Notice that one does not need to modify the library itself to break it, it is enough to modify the initial heap. To break **(4)**, one can modify the property `"_head"` of a priority queue object, which is expected to be private (Client 4). After assigning `null` to `"_head"` (line 5), all previously inserted value-priority pairs disappear from the

priority queue.

CLIENT 1:

```
var q = new PriorityQueue();
q.dequeue = function() {};
q.enqueue(1, "foo");
var r = q.dequeue();
```

CLIENT 2:

```
var q = new PriorityQueue();
q.enqueue(3, "bar");
var np = Object.getPrototypeOf(q._head);
var desc = { value: 0, writable: false };
Object.defineProperty(np, "pri", desc);
q.enqueue(1, "foo");
```

CLIENT 3:

```
var op = Object.prototype;
var desc = { value: null, writable: false };
Object.defineProperty(op, "_head", desc);
var q = new PriorityQueue();
```

CLIENT 4:

```
var q = new PriorityQueue();
q.enqueue(1, "last");
q.enqueue(3, "bar");
q.enqueue(2, "foo");
q._head = null;
var r = q.dequeue();
```

Figure 8.7.: Example clients that misuse the priority queue library.

In general, we want to ensure that all prototype chains are consistent with correct library behaviour. We can express this in the specification of a given library by stating which resources must not be present for its code to run correctly. In particular, constructed objects cannot redefine properties that are to be found in their prototypes; and prototypes cannot define as *non-writable* those properties that are to be present in their instances. We refer to these two criteria as *prototype safety*. CLIENTS 1-3 do not respect prototype safety of the library.

Commonly, a library has a private state, which is expected not to be accessed and manipulated by client programs. Since JavaScript does not provide full encapsulation, it is difficult for a library code to be agnostic to clients modifying library's internal state. CLIENT 4 does not respect the internal state of the library.

Next, we specify the priority queue library and discuss what it means to capture prototype safety and enforce encapsulation.

8.4.2. Specification of the Priority Queue Library

On top of providing basic JS Logic predicates, JaVerT also allows us to define our own predicates and use them to reason about more complex JavaScript structures. Here, we illustrate how to use JaVerT by specifying the functions from the Priority Queue library.

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

$$\begin{aligned}\mathsf{Node}(n, pri, val, next, nproto) \triangleq\ & \mathsf{DataProp}(n, \text{“pri”}, pri) * 0 \dot{<} pri * \\ & \mathsf{DataProp}(n, \text{“val”}, val) * \mathsf{DataProp}(n, \text{“next”}, next) * \\ & \mathsf{JSObject}(n, nproto) * (n, \text{“insertToQueue”}) \mapsto \varnothing * \\ & \mathsf{types}(pri : \mathsf{Num}, val : \mathsf{Str}, nproto : \mathsf{Obj})\end{aligned}$$

A node n is an object which has three data properties: `"pri"`, `"val"`, and `"next"`, with values pri, val, and $next$. The value pri has to be a number, greater than zero, as it represents the priority, whereas, for this example, we chose the value val to be a string. JavaScript prototype-based inheritance also requires to state that the node n is a JavaScript object whose prototype is $nproto$, which is an object; and the node n does not contain property named `"insertToQeueue"`. We need to make

sure that `"insertToQeueue"` does not exist in the node itself, as we want the node objects to inherit `"insertToQeueue"` from their prototype. This is an example of a *prototype safety* property.

Notice how the usage of JS Logic built-in predicates $\mathsf{DataProp}(o, p, v)$ and $\mathsf{JSObject}(o, p)$ makes the definition of nodes simple and concise.

The NodeProto predicate. The NodeProto 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 `insertToQueue` function shown in the example. The node prototype object also needs to satisfy the prototype safety property stating that it cannot contain non-writable `"pri"`, `"val"`, or `"next"` properties. We choose to go with a stronger specification, where these properties are not allowed at all:

$$\begin{aligned}\mathsf{NodeProto}(np) \triangleq\ & \exists\, l_i, sc_i.\ \mathsf{DataProp}(np, \text{“insertToQueue”}, l_i) \ *\ \mathsf{FunctionObject}(\mathsf{insertToQueue}, l_i, sc_i) \ * \\ & \mathsf{JSObject}(np, l_{op}) \ *\ (np, \text{“pri”}) \mapsto \varnothing \ *\ (np, \text{“val”}) \mapsto \varnothing \ *\ (np, \text{“next”}) \mapsto \varnothing\end{aligned}$$

This predicate states that a node prototype np: has a property `"insertToQueue"` bound to the location l_i of the function object representing in memory the function labelled with the identifier `insertToQueue`; has prototype `Object.prototype` (l_{op} denotes the location of the built-in `Object.prototype` object); and does not have the properties `"pri"`, `"val"`, and `"next"`.

There is one more detail that needs to be expanded on, and it has to do with the interplay between separation logic and the prototype inheritance of JavaScript. As `Node.prototype` is shared between all node objects, we cannot simply inline the definition of NodeProto in the definition of the Node predicate. Were we to do that, we could no longer write a satisfiable assertion describing two distinct nodes using the standard separating conjunction. One possible solution would be to use the overlapping conjunction ⋈. We will discuss shortly, in the specification of `insertToQueue`, the complications of describing two nodes using ⋈.

Specification of the `Node` function. The `Node` function is to be used as the constructor of node objects, that is, `var n = new Node(pri, val)`. A constructor in JavaScript is simply a function. Note that functions can be called as normal functions, as methods, and as constructors. Hence, if we intend to use a function as a constructor, but not as a normal function, we can state this in our specification. This would not allow us to prove programs that use our constructor function as a normal function, as the given precondition would not hold. To ensure that the function `Node` is not used as a normal function, we state that 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"`, `"val"`, and `"next"`, setting their values to `pri`, `val`, and `null`. The specification of `Node` is:

$$\left\{ \begin{array}{c} 0 \dot{<} \mathsf{pri} \ * \ \mathsf{types}(\mathsf{pri} : \mathsf{Num}, \mathsf{val} : \mathsf{Str}) \ * \ (\mathsf{this}, \text{“pri”}) \mapsto \varnothing \ * \ (\mathsf{this}, \text{“val”}) \mapsto \varnothing \ * \ (\mathsf{this}, \text{“next”}) \mapsto \varnothing \ * \\ \mathsf{JSObject}(\mathsf{this}, nproto) \ * \ \mathsf{NodeProto}(nproto) \ * \ (\mathsf{this}, \text{“insertToQueue”}) \mapsto \varnothing \ * \\ (l_{op}, \text{“pri”}) \mapsto \varnothing \ * \ (l_{op}, \text{“val”}) \mapsto \varnothing \ * \ (l_{op}, \text{“next”}) \mapsto \varnothing \end{array} \right\}$$

$$\texttt{Node(pri, val)}$$

$$\left\{ \begin{array}{c} \mathsf{Node}(\mathsf{this}, \mathsf{pri}, \mathsf{val}, \mathsf{null}, nproto) * \mathsf{NodeProto}(nproto) \ * \\ (l_{op}, \text{“pri”}) \mapsto \varnothing \ * \ (l_{op}, \text{“val”}) \mapsto \varnothing \ * \ (l_{op}, \text{“next”}) \mapsto \varnothing \end{array} \right\}$$

The precondition of `Node` states: restrictions on its parameters, that is, the priority `pri` is greater than zero and is of number type, whereas the node value `val` is of string type; restrictions on the function being used as a constructor, that is, the keyword `this` must be initially bound to an object that does not have the properties `"pri"`, `"val"`, and `"next"` and has prototype *nproto*, which is a valid node prototype; a prototype safety property, stating that the `this` object does not have the property `"insertToQueue"`; and the prototype safety requirements for `Object.prototype`, stating that `Object.prototype` does not have properties `"pri"`, `"val"`, and `"next"`.

The postcondition states that after the execution of the body of `Node`: the keyword `this` is bound to a Node object with priority `pri`, value `val`, no next node, and prototype *nproto*; *nproto* is still a valid node prototype; and we still have the prototype safety requirements for `Object.prototype`.

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

$$\begin{array}{rcl} \mathsf{NodeList}(\mathsf{null}, nproto, 0, 0) & \triangleq & \mathtt{emp} \\ \mathsf{NodeList}(nl, nproto, pri_{max}, len) & \triangleq & \exists\, pri, val, next, len_{rest}.\ 0 \dot{<} pri_{max} \ * \\ & & \mathsf{Node}(nl, pri_{max}, val, next, nproto) \ * \ pri \dot{\leq} pri_{max} \ * \\ & & \mathsf{NodeList}(next, nproto, pri, len_{rest}) \ * \ len \doteq len_{rest} + 1 \ * \\ & & \mathsf{types}(nl, nproto : \mathsf{Obj}, pri, pri_{max}, len, len_{rest} : \mathsf{Num}) \end{array}$$

A node list $\mathsf{NodeList}(nl, nproto, pri_{max}, len)$ is a `null`-terminated list of length *len* of Node objects singly linked via their `"next"` properties. All of the nodes in the node list share the same prototype *nproto* and have priority not greater than pri_{max}. Given our choice of the underlying data structure, the NodeList predicate needs to be recursive. In the base case, we have an empty node list, meaning that *nl* has to equal `null`, the priority and the length are equal to zero. In the recursive case, the node list starts with a node that has priority pri_{max}, value *val*, and points to the next node in the queue *next*. The tail of the node list is also a node list, starting with the node *next* and maximum priority *pri*, which has to be not greater than pri_{max}.

Note that we are not able to inline the definition of NodeProto in the definition of Node and use $*$ in the the recursive case of the definition of NodeList. We could use $\bowtie$, which allows partial separation between heaps:

$$\begin{array}{l} H, \rho, L, v_t, \epsilon \models P_1 \bowtie P_2 \Leftrightarrow \\ \exists H_1, H_2, H_3.\, H = H_1 \uplus H_2 \uplus H_3 \ \wedge \ (H_1 \uplus H_3, \rho, L, v_t, \epsilon \models P_1) \wedge (H_2 \uplus H_3, \rho, L, v_t, \epsilon \models P_2). \end{array}$$

$\bowtie$ would allow us to account for the sharing of `Node.prototype`. However, by using $\bowtie$, we would lose information that only `Node.prototype` is shared between the two nodes. We discuss the impact of

losing information after giving the specification for `insertToQueue`.

Specification of `insertToQueue`. Next, we show the specification of `Node.prototype.insertToQueue`. The function `insertToQueue` is used for inserting a new node object into a list of node objects. For example, `q = n.insertToQueue(q)` adds the node `n` to the queue whose head is `q` and returns the head of the extended priority queue which is then assigned to `q`. The formal specification of `insertToQueue` is given below:

$$\left\{ \begin{array}{c} \mathsf{NodeList}(\mathsf{q}, nproto, pri_q, len) \ * \ \mathsf{Node}(\mathsf{this}, pri_n, val, \mathsf{null}, nproto) \ * \ \mathsf{NodeProto}(nproto) \ * \\ \mathsf{types}(pri_q, pri_n : \mathsf{Num}) \end{array} \right\}$$

$$\texttt{insertToQueue(q)}$$

$$\left\{ \ \mathsf{NodeList}(\mathsf{ret}, nproto, \mathsf{max}(pri_q, pri_n), len + 1) * \mathsf{NodeProto}(nproto) * \ \mathsf{types}(\mathsf{ret} : \mathsf{Obj}) \ \right\}$$

The precondition states that `q` is bound to the head of a node list with max priority pri_q, whose node elements all have the valid node prototype *nproto*. It also states that the keyword `this` is bound to a node object with priority pri_n, value *val*, no next element, and prototype *nproto*. The postcondition states that `insertToQueue` returns the head of a node list with max priority $\mathsf{max}(pri_q, pri_n)$ (the keyword `ret` refers to the return value in the postcondition), that all node elements of this node list have the (still valid) node prototype *nproto*.

Even though JaVerT supports user-defined predicates, it does not reason automatically about user-defined recursive predicates, which means that programs need to be annotated with special logical commands that tell JaVerT when to fold or unfold a given user-defined predicate. Let us now consider the annotated code of `insertToQueue` (Figure 8.8).

```
1  Node.prototype.insertToQueue = function (q) {
2
3      /** @unfold NodeList(q, #nproto, #pri_q, #len) */
4      if (q === null) {
5          /** @fold NodeList(this, #nproto, #pri_n, #len+1) */
6          return this
7       }
8
9       if (this.pri >= q.pri) {
10         this.next = q;
11         /** @fold NodeList(this, #nproto, #pri_n, #len+1) */
12         return this
13      }
14
15      var tmp = this.insertToQueue (q.next);
16      q.next = tmp;
17      /** @fold NodeList(q, #nproto, #pri_q, #len+1) */
18      return q
19 }
```

Figure 8.8.: Running Example - annotated code of `insertToQueue`

Because the program starts by branching on the value of `q`, we first have to unfold the NodeList predicate in the precondition. Then, there are three possible cases:

1. If `q` is `null`, we simply have to fold the node bound to `this` as a node list and return it;

2. If the priority of the node bound to `this` is greater than or equal to the priority of the first node of `q`, we have to set the property `"next"` of `this` to `q`, fold the node list now bound to `this`, and return it;

3. If the priority of the node bound to `this` is less than the priority of the first node of `q`, we have to call `insertToQueue` recursively on the rest of the node list to obtain a new head. The head of the node list remains equal to `q`, since the new node was inserted in the middle of the node list. Hence, we fold the node list bound to `q`.

The specification of `insertToQueue` explicitly mentions NodeProto. If we were to inline the definition of NodeProto in the definition of Node, the precondition of `insertToQueue` would look like this: $\mathsf{NodeList}(\mathsf{q}, ...) \bowtie \mathsf{Node}(\mathsf{this}, ...)$. We would encounter a problem in line 10 (Figure 8.8) when proving the body of `insertToQueue`. In line 10, we attempt to assign to `this.next`, where $\mathsf{Node}(\mathsf{this}, ...)$ has the required resource for the heap update. However, we do not know if this update would not have an effect to $\mathsf{NodeList}(\mathsf{q}, ...)$ as $\bowtie$ loses the information of what is separate and what is shared.

The Queue Predicate. In order to specify the `PriorityQueue` function, we need to create the abstraction for a queue:

$$\begin{aligned}\mathsf{Queue}(q, qproto, nproto, pri, len) \triangleq\ & \exists\, head.\ \mathsf{DataProp}(q, \text{“\_head”}, head) * \\ & \mathsf{NodeList}(head, nproto, pri, len) * \\ & \mathsf{JSObject}(q, qproto) * \\ & (q, \text{“enqueue”}) \mapsto \varnothing * \ (q, \text{“dequeue”}) \mapsto \varnothing * \\ & \mathsf{types}(pri, len : \mathsf{Num}, qproto : \mathsf{Obj})\end{aligned}$$

A queue q has a data property `"_head"` holding a value *head*, which corresponds to the head node in a node list with the maximum priority *pri* and the length *len*. The queue's prototype is *qproto* which is described shortly. To satisfy prototype safety, a queue should not have properties `"enqueue"` and `"dequeue"`, as these properties will be inherited from *qproto*. The values *pri* and *len* have to be numbers, while *qproto* must be an object. Observe here that we do not expose *head* as a parameter of the Queue predicate; this will be relevant once we have reached the topic of encapsulation.

The QueueProto predicate. Just as NodeProto describes a valid node prototype, the QueueProto predicate describes what it means for an object to be a valid queue prototype. First, a valid queue prototype needs to capture all of the properties of the queue prototype object, such as the `enqueue` and `dequeue` functions. Second, a valid queue prototype cannot contain a non-writable `"_head"` property, where we choose, analogously to the NodeProto case, a stronger specification of not allowing the property at all. Finally, we would like to abstract our queue module over the implementation details of using nodes. Hence, we include the function object `Node` and its prototype in the QueueProto predicate:

$$\begin{aligned}\mathsf{QueueProto}(qp, nproto, sc_e) \triangleq\ & \exists\, l_e, l_d, n.\ \mathsf{DataProp}(qp, \text{“enqueue”}, l_e) * \mathsf{FunctionObject}(\mathsf{enqueue}, l_e, sc_e) * \\ & \mathsf{DataProp}(qp, \text{“dequeue”}, l_d) * \mathsf{FunctionObject}(\mathsf{dequeue}, l_d, \_) * \\ & \mathsf{JSObject}(qp, l_{op}) * (qp, \text{“\_head”}) \mapsto \varnothing * \\ & \mathsf{FunctionObject}(\mathsf{Node}, n, \_) * \mathsf{DataProp}(n, \text{“prototype”}, nproto) * \\ & \mathsf{NodeProto}(nproto) * \mathsf{Scope}(\mathsf{Node} : n, sc_e, \mathsf{enqueue})\end{aligned}$$

This predicate states that a queue prototype *qp*: has properties `"enqueue"` and `"dequeue"` bound to the locations l_e and l_d of the function objects representing in memory the functions labelled with the identifiers `enqueue` and `dequeue`, respectively; has `Object.prototype` as its prototype; does not have

the property `"_head"`, to capture prototype safety; contains the function object n, representing the function with the identifier `Node`, with its property `"prototype"`, which is a valid node prototype. We also need to state that the variable `Node` is accessible inside `enqueue`. To describe that, we use the JS built-in predicate Scope.

JS Logic Predicate: Scope. To capture variable scoping, we introduce the Scope predicate. The $\mathsf{Scope}(x : v, sc_f, m)$ predicate states that the variable x has value v in the scope chain denoted by sc_f of the function literal with identifier m:

$$\mathsf{Scope}(x : v, sc_f, m) \quad \triangleq \quad (\mathsf{nth}\,(sc_f, n), x) \mapsto v, \text{ where } n = \psi(m, x)$$

In the general case, this predicate corresponds to the JS Logic assertion $(\mathsf{nth}\,(sc_f, n), x) \mapsto v$, where `nth` is the binary list indexing operator and $n = \psi(m, x)$. For instance, the predicate $\mathsf{Scope}(\mathsf{Node} : n, sc_e, \mathsf{enqueue})$ unfolds to $(\mathsf{nth}\,(sc_e, 1), \text{“Node”}) \mapsto n$ as $\psi(\mathsf{enqueue}, \mathsf{Node}) = 1$. We can also use $\mathsf{Scope}(x : v)$ as syntactic sugar for $\mathsf{Scope}(x : v, sc, m)$, where sc is the special logical expression denoting the current scope chain and m is the identifier of the current function.

Specification of the `PriorityQueue` function. The `PriorityQueue` function is to be used as a constructor of queue objects, that is, `var q = new PriorityQueue()`. Therefore, at the beginning of every valid execution of `PriorityQueue`, the keyword `this` is bound to a new object whose prototype is the `PriorityQueue.prototype` object. The function `PriorityQueue` then extends the `this` object with the property `"_head"` setting its initial value to be `null`. The specification of `PriorityQueue` is:

$$\left\{ \begin{array}{c} \mathsf{JSObject}(\mathsf{this}, qproto) \;*\; (\mathsf{this}, \text{“\_head”}) \mapsto \varnothing \;*\; (\mathsf{this}, \text{“enqueue”}) \mapsto \varnothing \;*\; (\mathsf{this}, \text{“dequeue”}) \mapsto \varnothing \;* \\ \mathsf{QueueProto}(qproto, nproto, sc_e) \;*\; \mathsf{ObjProtoF}() \end{array} \right\}$$

$$\texttt{PriorityQueue}()$$

$$\left\{ \mathsf{Queue}(\mathsf{this}, qproto, nproto, 0, 0) * \mathsf{QueueProto}(qproto, nproto, sc_e) \;*\; \mathsf{ObjProtoF}() \right\}$$

The precondition of `PriorityQueue` states that: the keyword `this` must be initially bound to a JavaScript object whose prototype is *qproto*; the `this` object must not have properties `"_head"`, `"enqueue"`, and `"dequeue"`, to capture prototype safety; and *qproto* is a valid queue prototype. The postcondition states that, after the execution of the body of `PriorityQueue`: the keyword `this` is bound to an empty Queue; and *qproto* is still a valid queue prototype. The prototype safety requirements of the library extend to `Object.prototype`. This resource is captured by the built-in ObjProtoF() predicate, describing the frozen `Object.prototype` object, presented in §8.3. Here, the user can instead choose not to use the frozen version of the predicate. In that case, they would have to manually specify the prototype safety requirements, as we have done for `Node` function.

Specification of `enqueue`. Next, we show the specification of `PriorityQueue.prototype.enqueue`. The function `enqueue` is used for creating a new node and inserting it to the queue. For example, `q.enqueue(1, "last")` creates a new node with the priority 1 and the value `"last"` and inserts it to the queue `q`. The specification of `enqueue` is given below:

$$\left\{ \begin{array}{c} 0 \dot{<} \mathsf{pri} \ * \ \mathsf{Queue}(\mathsf{this}, qproto, nproto, pri_q, len) \ * \ \mathsf{QueueProto}(qproto, nproto, sc_e) \ * \\ \mathsf{OChains}(\mathsf{enqueue} : sc_e) * \ \mathsf{types}(\mathsf{pri} : \mathsf{Num}\,, \mathsf{val} : \mathsf{Str}\,) \ * \ \mathsf{ObjProtoF}() \end{array} \right\}$$

$$\texttt{enqueue(pri, val)}$$

$$\left\{ \ \mathsf{Queue}(\mathsf{this}, qproto, nproto, \mathsf{max}(pri_q, \mathsf{pri}), len + 1) \ * \mathsf{QueueProto}(qproto, nproto, sc_e) \ * \ \mathsf{ObjProtoF}() \ \right\}$$

The precondition states that the keyword `this` is bound to the queue with max priority pri_q and length len, whose prototype is a valid queue prototype $qproto$. It also states that `pri` is a number, greater that zero, while `val` is a string. The postcondition states that after executing `enqueue`, the queue bound by `this` has the max priority $\mathsf{max}(pri_q, \mathsf{pri})$, contains one more element and whose prototype is the (still valid) queue prototype $qproto$. Similarly to the `PriorityQueue` specification, we use $\mathsf{ObjProtoF}()$ to capture prototype safety for `Object.prototype`. Additionally, we need to make sure that the variable `Node` is found in the current scope of the function `enqueue`. We could try to capture this with the assertion $\mathsf{Scope}(\mathsf{Node} : n)$, but this is duplicated resource already existing in $\mathsf{QueueProto}(qproto, nproto, sc_e)$. We need a predicate that captures the scope chain overlap between two functions.

JS Logic Predicate: OChains. To capture the scope chain overlap between two functions, we introduce the OChains predicate:

$$\mathsf{OChains}(\mathsf{f} : sc_f, \mathsf{g} : sc_g) \quad \triangleq \quad \circledast_{0 \leq i < n}(\mathsf{nth}\,(sc_f, i) = \mathsf{nth}\,(sc_g, i)), \text{ where } n = \psi^o(\mathsf{f}, \mathsf{g})$$

The *overlapping scope function*, $\psi^o : \mathcal{S}tr \times \mathcal{S}tr \rightharpoonup \mathbb{N}$, takes two function identifiers and returns the length of the overlap of their scope chains. We can also use $\mathsf{OChains}(\mathsf{f} : sc_f)$ as syntactic sugar for $\mathsf{OChains}(\mathsf{f} : sc_f, m : sc)$, where sc is the special logical expression denoting the current scope chain and m is the identifier of the current function.

Specification of `dequeue`. Next, we show the specification of `PriorityQueue.prototype.dequeue`. The function `dequeue` is used for removing the element with the highest priority from the queue. If the queue `q` is empty, `q.dequeue()` throws an exception, otherwise it returns an element from the queue `q` with the highest priority represented as an object with two properties, `"pri"` and `"val"`.

The formal specification of the exceptional case of `dequeue` is given below:

$$\left\{ \quad \mathsf{Queue}(\mathsf{this}, qproto, nproto, 0, 0) \ * \ \mathsf{QueueProto}(qproto, nproto, sc_e) \ * \ \mathsf{ObjProtoF}() \quad \right\}$$

$$\texttt{dequeue()}$$

$$\left\{ \begin{array}{c} \mathsf{Queue}(\mathsf{this}, qproto, nproto, 0, 0) \ * \mathsf{QueueProto}(qproto, nproto, sc_e) \ * \\ \mathsf{ErrorWithMsg}(\mathsf{err}, \text{“Queue is empty”}) \ * \ \mathsf{ObjProtoF}() \end{array} \right\}$$

The precondition states that the keyword `this` is bound to an empty queue. The postcondition states that the queue is still empty and the result (represented by the special variable `err`) of executing the function `dequeue` is an error object with a message `"Queue is empty"`.

The formal specification of the normal case of `dequeue` is:

$$\{ \quad \mathsf{Queue}(\mathsf{this}, qproto, nproto, pri, len) \ * \ \mathsf{QueueProto}(qproto, nproto, sc_e) \ * \ 0 \dot{<} len \ * \ \mathsf{ObjProtoF}() \quad \}$$

$$\texttt{dequeue}()$$

$$\left\{ \begin{array}{c} \mathsf{Queue}(\mathsf{this}, qproto, nproto, pri_r, len_r) \ * \mathsf{QueueProto}(qproto, nproto, sc_e) \ * \ len \doteq len_r + 1 \ * \\ pri_r \dot{\leq} pri \ * \ \mathsf{JSObject}(\mathsf{ret}, l_{op}) \ * \ \mathsf{DataProp}(\mathsf{ret}, \text{“pri”}, pri) \ * \ \mathsf{DataProp}(\mathsf{ret}, \text{“val”}, val) \ * \ \mathsf{ObjProtoF}() \end{array} \right\}$$

The precondition states that the keyword `this` is bound to a queue, which is not empty and whose prototype is a valid queue prototype. The postcondition states that: the queue has one less elements and its priority is not greater than *pri*; and the result of the `dequeue` is a JavaScript object with two named properties `"pri"` and `"val"`. The named property `"pri"` contains the value *pri*, which was the maximum priority of the given queue in the precondition.

Specification of `Module`. Finally, we show the specification of `Module`. The function `Module` is used for implementing the priority queue functionality, using immediately invoked function expression. The formal specification of `Module` is given below:

$$\left\{ \ \mathsf{ObjProtoF}() \ \right\}$$

$$\texttt{Module}()$$

$$\left\{ \begin{array}{c} \mathsf{FunctionObject}(\mathsf{PriorityQueue}, \mathsf{ret}, \_) \ * \ \mathsf{DataProp}(\mathsf{ret}, \text{“prototype”}, qproto) \ * \\ \mathsf{QueueProto}(qproto, nproto, \_) \ * \ \mathsf{ObjProtoF}() \end{array} \right\}$$

The precondition requires a frozen `Object.prototype` object. The postcondition states that the result of `Module`, is a function object with the identifier `PriorityQueue`, which named property `"prototype"` is a valid queue prototype. By choosing the QueueProto abstraction we are able to have such a succinct specification for the `Module` function.

The specifications of the priority queue library show that it is possible to successfully abstract over JavaScript internals, allowing both the library developer and the client developer to write specifications that are as free as possible from JavaScript-specific clutter. Next we demonstrate the verification of the correct client code and explain why the clients, that misuse the library, cannot be verified.

8.4.3. Verification of Client Code

We discuss two important aspects of specifying JavaScript libraries: capturing prototype safety and enforcing encapsulation. Given the specifications of the priority queue library, we demonstrate the verification of the valid client program from our running example. Also, we show that it is not possible to verify client code (Client 1 - Client 3, Figure 8.7) if it compromises prototype safety. The situation for encapsulation is more subtle. There are ways of breaking encapsulation that we could choose to allow. The client could, for instance, add more functionalities to `Queue.prototype` or add more properties to queue objects, and this would not break the existing functionalities. However, there are ways of breaking encapsulation that we should certainly disallow, as Client 4 demonstrates. We discuss a solution for ensuring that such client code cannot be verified.

Valid Client Code: Running Example. We show a proof sketch below[1]. Recall the postcondition of the `Module` function, which creates our new priority queue library: it is highlighted in the starting

[1]Note that in JaVerT we symbolically execute JSIL programs compiled from JavaScript programs. For illustration purposes, in the verification of example client programs we give JavaScript code.

symbolic state below. We first create an empty queue with maximum priority 0. Next we create three nodes, obtaining a queue with three nodes and maximum priority 3. Then, we dequeue the head of the queue (which we can do, as we know that the queue has 3 nodes), obtaining a queue with 2 nodes and existentially quantified priority pri not greater than 3. Moreover, in the end, the variable `r` is bound to an object with two properties: `"pri"`, with value 3; and `"val"`, with value val which is existentially quantified.

$$\left\{\begin{array}{c}\mathsf{Scope}(\mathsf{PriorityQueue} : pq) \ * \ \mathsf{FunctionObject}(\mathsf{PriorityQueue}, pq, \_) \ * \ \mathsf{DataProp}(pq, \text{“prototype”}, qproto) \ * \\ \mathsf{QueueProto}(qproto, nproto, sc_e) \ * \ \mathsf{ObjProtoF}() \ * \ \mathsf{Scope}(\mathsf{r} : \mathsf{undefined}) \ * \ \mathsf{Scope}(\mathsf{q} : \mathsf{undefined})\end{array}\right\}$$

```
var q = new PriorityQueue();
```

$$\left\{\begin{array}{c}\mathsf{Scope}(\mathsf{PriorityQueue} : pq) \ * \ \mathsf{FunctionObject}(\mathsf{PriorityQueue}, pq, \_) \ * \ \mathsf{DataProp}(pq, \text{“prototype”}, qproto) \ * \\ \mathsf{Scope}(\mathsf{q} : q) \ * \ \mathsf{Queue}(q, qproto, nproto, 0, 0) * \mathsf{QueueProto}(qproto, nproto, sc_e) \ * \ \mathsf{ObjProtoF}() \ * \\ \mathsf{Scope}(\mathsf{r} : \mathsf{undefined})\end{array}\right\}$$

```
q.enqueue(1, "last"); q.enqueue(3, "bar"); q.enqueue(2, "foo");
```

$$\left\{\begin{array}{c}\mathsf{Scope}(\mathsf{PriorityQueue} : pq) \ * \ \mathsf{FunctionObject}(\mathsf{PriorityQueue}, pq, \_) \ * \ \mathsf{DataProp}(pq, \text{“prototype”}, qproto) \ * \\ \mathsf{Scope}(\mathsf{q} : q) \ * \ \mathsf{Queue}(q, qproto, nproto, 3, 3) * \mathsf{QueueProto}(qproto, nproto, sc_e) \ * \ \mathsf{ObjProtoF}() \ * \\ \mathsf{Scope}(\mathsf{r} : \mathsf{undefined})\end{array}\right\}$$

```
var r = q.dequeue();
```

$$\left\{\begin{array}{c}\mathsf{Scope}(\mathsf{PriorityQueue} : pq) \ * \ \mathsf{FunctionObject}(\mathsf{PriorityQueue}, pq, \_) \ * \ \mathsf{DataProp}(pq, \text{“prototype”}, qproto) \ * \\ \mathsf{Scope}(\mathsf{q} : q) \ * \ \mathsf{Queue}(q, qproto, nproto, pri, 2) * \mathsf{QueueProto}(qproto, nproto, sc_e) \ * \ \mathsf{ObjProtoF}() \ * \\ pri \dot{\leq} 3 \ * \ \mathsf{Scope}(\mathsf{r} : r) \ * \ \mathsf{JSObject}(r, l_{op}) \ * \ \mathsf{DataProp}(r, \text{“pri”}, 3) * \mathsf{DataProp}(r, \text{“val”}, val)\end{array}\right\}$$

Misusing the Library: Client 1. The client code overrides `dequeue` on the constructed priority queue object. This code requires a resource $(q, \text{“dequeue”}) \mapsto \varnothing$ from inside $\mathsf{Queue}(q, qproto, nproto, 0, 0)$ predicate and updates that resource to $\mathsf{DataProp}(q, \text{“dequeue”}, l_f)$, where l_f is a location of a newly created function object. After this point we cannot fold back $\mathsf{Queue}(q, ...)$ predicate anymore, since by definition it requires $(q, \text{“dequeue”}) \mapsto \varnothing$, and hence we cannot apply the specification of the `enqueue` function to continue.

$$\left\{\begin{array}{c}\mathsf{Scope}(\mathsf{PriorityQueue} : pq) \ * \ \mathsf{FunctionObject}(\mathsf{PriorityQueue}, pq, \_) \ * \ \mathsf{DataProp}(pq, \text{“prototype”}, qproto) \ * \\ \mathsf{QueueProto}(qproto, nproto, sc_e) \ * \ \mathsf{ObjProtoF}() \ * \ \mathsf{Scope}(\mathsf{r} : \mathsf{undefined}) \ * \ \mathsf{Scope}(\mathsf{q} : \mathsf{undefined})\end{array}\right\}$$

```
var q = new PriorityQueue();
```

$$\left\{\begin{array}{c}\mathsf{Scope}(\mathsf{PriorityQueue} : pq) \ * \ \mathsf{FunctionObject}(\mathsf{PriorityQueue}, pq, \_) \ * \ \mathsf{DataProp}(pq, \text{“prototype”}, qproto) \ * \\ \mathsf{Scope}(\mathsf{q} : q) \ * \ \mathsf{Queue}(q, qproto, nproto, 0, 0) * \mathsf{QueueProto}(qproto, nproto, sc_e) \ * \ \mathsf{ObjProtoF}() \ * \\ \mathsf{Scope}(\mathsf{r} : \mathsf{undefined})\end{array}\right\}$$

```
q.dequeue = function(){};
```

$$\{ \quad ... * \ \mathsf{DataProp}(q, \text{“dequeue”}, l_f) \ * ... \quad \}$$

```
q.enqueue(1, "foo");
```

×

Misusing the Library: Client 2. The client code assigns a *non-writable* value 0 to the property `"pri"` of `Node.prototype`. This code requires a resource $(nproto, \text{“pri”}) \mapsto \varnothing$ from $\mathsf{NodeProto}(nproto)$ predicate, which is a part of $\mathsf{QueueProto}(qproto, nproto, sc_e)$ predicate. After the assignment, the symbolic state contains $\mathsf{DataPropGen}(nproto, \text{“pri”}, 0, \mathsf{false}, \mathsf{true}, \mathsf{true})$. Similarly as in Client 1 case, we cannot fold $\mathsf{NodeProto}(nproto)$ and $\mathsf{QueueProto}(qproto, ...)$ predicates anymore, since by definition $\mathsf{NodeProto}(nproto)$ requires $(nproto, \text{“pri”}) \mapsto \varnothing$. Consequently, we cannot apply the specification of the `enqueue` function to continue.

$$\left\{\begin{array}{c}\mathsf{Scope}(\mathsf{PriorityQueue} : pq) \ * \ \mathsf{FunctionObject}(\mathsf{PriorityQueue}, pq, \_) \ * \\ \mathsf{DataProp}(pq, \text{“prototype”}, qproto) \ * \ \mathsf{QueueProto}(qproto, nproto, sc_e) \ * \ \mathsf{ObjProtoF}() \ * \\ \mathsf{Scope}(\mathsf{q} : \mathsf{undefined}) \ * \ \mathsf{Scope}(\mathsf{np} : \mathsf{undefined}) \ * \ \mathsf{Scope}(\mathsf{desc} : \mathsf{undefined})\end{array}\right\}$$

```
var q = new PriorityQueue();
```

$$\left\{\begin{array}{c}\mathsf{Scope}(\mathsf{PriorityQueue} : pq) \ * \ \mathsf{FunctionObject}(\mathsf{PriorityQueue}, pq, \_) \ * \ \mathsf{DataProp}(pq, \text{“prototype”}, qproto) \ * \\ \mathsf{Scope}(\mathsf{q} : q) \ * \ \mathsf{Queue}(q, qproto, nproto, 0, 0) * \mathsf{QueueProto}(qproto, nproto, sc_e) \ * \ \mathsf{ObjProtoF}() \ * \\ \mathsf{Scope}(\mathsf{np} : \mathsf{undefined}) \ * \ \mathsf{Scope}(\mathsf{desc} : \mathsf{undefined})\end{array}\right\}$$

```
q.enqueue(3, "bar");
```

$$\left\{\begin{array}{c}\mathsf{Scope}(\mathsf{PriorityQueue} : pq) \ * \ \mathsf{FunctionObject}(\mathsf{PriorityQueue}, pq, \_) \ * \ \mathsf{DataProp}(pq, \text{“prototype”}, qproto) \ * \\ \mathsf{Scope}(\mathsf{q} : q) \ * \ \mathsf{Queue}(q, qproto, nproto, 3, 1) * \mathsf{QueueProto}(qproto, nproto, sc_e) \ * \ \mathsf{ObjProtoF}() \ * \\ \mathsf{Scope}(\mathsf{np} : \mathsf{undefined}) \ * \ \mathsf{Scope}(\mathsf{desc} : \mathsf{undefined})\end{array}\right\}$$

```
var np = Object.getPrototypeOf(q._head);
```

$$\left\{\begin{array}{c}\mathsf{Scope}(\mathsf{PriorityQueue} : pq) \ * \ \mathsf{FunctionObject}(\mathsf{PriorityQueue}, pq, \_) \ * \ \mathsf{DataProp}(pq, \text{“prototype”}, qproto) \ * \\ \mathsf{Scope}(\mathsf{q} : q) \ * \ \mathsf{Queue}(q, qproto, nproto, 3, 1) * \mathsf{QueueProto}(qproto, nproto, sc_e) \ * \ \mathsf{ObjProtoF}() \ * \\ \mathsf{Scope}(\mathsf{np} : nproto) \ * \ \mathsf{Scope}(\mathsf{desc} : \mathsf{undefined})\end{array}\right\}$$

```
var desc = { value: 0, writable: false }; Object.defineProperty(np, "pri", desc);
```

$$\{ \quad ... \ * \ \mathsf{DataPropGen}(nproto, \text{“pri”}, 0, \mathsf{false}, \mathsf{true}, \mathsf{true}) \ * \ ... \quad \}$$

```
q.enqueue(1, "foo");
```

×

Misusing the Library: Client 3. The client code defines a property `"_head"` containing a *non-writable* value `null` of `Object.prototype`. In this case, differently than Client 1 and Client 2, the code does not modify the library itself. However, since the symbolic state contains a frozen `Object.prototype`, required by the priority queue library, an the update of `Object.prototype` is not allowed.

$$\left\{\begin{array}{c}\mathsf{Scope}(\mathsf{PriorityQueue} : pq) \ * \ \mathsf{FunctionObject}(\mathsf{PriorityQueue}, pq, \_) \ * \\ \mathsf{DataProp}(pq, \text{“prototype”}, qproto) \ * \ \mathsf{QueueProto}(qproto, nproto, sc_e) \ * \ \mathsf{ObjProtoF}() \ * \\ \mathsf{Scope}(\mathsf{q} : \mathsf{undefined}) \ * \ \mathsf{Scope}(\mathsf{op} : \mathsf{undefined}) \ * \ \mathsf{Scope}(\mathsf{desc} : \mathsf{undefined})\end{array}\right\}$$

```
var op = Object.prototype;
var desc = { value: null, writable: false };
Object.defineProperty(op, "_head", desc);
```

×

Misusing the Library: Client 4. We demonstrated that it is not possible to verify a specification of client code if it compromises prototype safety. CLIENT 4 does not compromise prototype safety, and we can symbolically execute the code using our library specification. However, we end up with a symbolic state that does not imply the postcondition expected by the client code.

$$\left\{\begin{array}{c}\mathsf{Scope}(\mathsf{PriorityQueue} : pq) \ * \ \mathsf{FunctionObject}(\mathsf{PriorityQueue}, pq, \_) \ * \ \mathsf{DataProp}(pq, \text{“prototype”}, qproto) \ * \\ \mathsf{QueueProto}(qproto, nproto, sc_e) \ * \ \mathsf{ObjProtoF}() \ * \ \mathsf{Scope}(\mathsf{r} : \mathsf{undefined}) \ * \ \mathsf{Scope}(\mathsf{q} : \mathsf{undefined})\end{array}\right\}$$

```
var q = new PriorityQueue();
```

$$\left\{\begin{array}{c}\mathsf{Scope}(\mathsf{PriorityQueue} : pq) \ * \ \mathsf{FunctionObject}(\mathsf{PriorityQueue}, pq, \_) \ * \ \mathsf{DataProp}(pq, \text{“prototype”}, qproto) \ * \\ \mathsf{Scope}(\mathsf{q} : q) \ * \ \mathsf{Queue}(q, qproto, nproto, 0, 0) * \mathsf{QueueProto}(qproto, nproto, sc_e) \ * \ \mathsf{ObjProtoF}() \ * \\ \mathsf{Scope}(\mathsf{r} : \mathsf{undefined})\end{array}\right\}$$

```
q.enqueue(1, "last"); q.enqueue(3, "bar"); q.enqueue(2, "foo");
```

$$\left\{\begin{array}{c}\mathsf{Scope}(\mathsf{PriorityQueue} : pq) \ * \ \mathsf{FunctionObject}(\mathsf{PriorityQueue}, pq, \_) \ * \ \mathsf{DataProp}(pq, \text{“prototype”}, qproto) \ * \\ \mathsf{Scope}(\mathsf{q} : q) \ * \ \mathsf{Queue}(q, qproto, nproto, 3, 3) * \mathsf{QueueProto}(qproto, nproto, sc_e) \ * \ \mathsf{ObjProtoF}() \ * \\ \mathsf{Scope}(\mathsf{r} : \mathsf{undefined})\end{array}\right\}$$

```
q._head = null;
```

$$\left\{\begin{array}{c}\mathsf{Scope}(\mathsf{PriorityQueue} : pq) \ * \ \mathsf{FunctionObject}(\mathsf{PriorityQueue}, pq, \_) \ * \ \mathsf{DataProp}(pq, \text{“prototype”}, qproto) \ * \\ \mathsf{Scope}(\mathsf{q} : q) \ * \ \mathsf{Queue}(q, qproto, nproto, 0, 0) * \mathsf{QueueProto}(qproto, nproto, sc_e) \ * \ \mathsf{ObjProtoF}() \ * \\ \mathsf{Scope}(\mathsf{r} : \mathsf{undefined}) \ * \ \mathsf{NodeList}(old, nproto, 3, 3)\end{array}\right\}$$

```
var r = q.dequeue();
```

$$\left\{\begin{array}{c}\mathsf{Scope}(\mathsf{PriorityQueue} : pq) \ * \ \mathsf{FunctionObject}(\mathsf{PriorityQueue}, pq, \_) \ * \ \mathsf{DataProp}(pq, \text{“prototype”}, qproto) \ * \\ \mathsf{Scope}(\mathsf{q} : q) \ * \ \mathsf{Queue}(q, qproto, nproto, 0, 0) * \mathsf{QueueProto}(qproto, nproto, sc_e) \ * \ \mathsf{ObjProtoF}() \ * \\ * \ \mathsf{Scope}(\mathsf{r} : \mathsf{undefined}) \ * \mathsf{NodeList}(old, nproto, 3, 3) \ * \ \mathsf{ErrorWithMsg}(\mathsf{err}, \text{“Queue is empty”})\end{array}\right\}$$

Assigning `null` to the `q._head` empties the queue. Even though the assignment does not break the library, however, it exploits the private state of the library. Instead of having a queue of two elements after `var r = q.dequeue()`, the client code throws an exception, as the queue is empty.

One way to ensure full encapsulation would be to keep the Queue predicate partially opaque to the client code. We cannot have the whole predicate Queue opaque, as the client still needs to be able to call the functions from the library. For example, we can define the Queue predicate using QueuePrivate as follows:

$$
\begin{array}{rcl}
\mathsf{Queue}(q, qproto, nproto, pri, len) & \triangleq & \mathsf{QueuePrivate}(q, nproto, pri, len) * \\
& & \mathsf{JSObject}(q, qproto) * \\
& & (q, \text{“enqueue”}) \mapsto \varnothing * \ (q, \text{“dequeue”}) \mapsto \varnothing * \\
& & \mathsf{types}(pri, len : \mathsf{Num}, qproto : \mathsf{Obj}) \\
\mathsf{QueuePrivate}(q, nproto, pri, len) & \triangleq & \exists\, head.\ \mathsf{DataProp}(q, \text{“\_head”}, head) * \\
& & \mathsf{NodeList}(head, nproto, pri, len)
\end{array}
$$

The predicate QueuePrivate is fully opaque to the client. By keeping library predicates partially opaque, we can make sure that client code cannot exploit the private state of the library. Verification of the assignment `q._head = null` would not work as the required resource for it is hidden inside the private part of the queue predicate. Abstract predicates introduced by Parkinson et al. [53] can be applied to implement partially opaque predicates. However, they are not supported by JaVerT at the moment.

Summary. We have demonstrated JaVerT using our running example. We showed how to develop natural JavaScript abstractions, such as Node and Queue, that make reasoning using JaVerT nearly as simple as reasoning about Java programs using a semi-automatic verification tool. Using such abstractions, we provided specifications that ensure prototype safety. The verification workflow of JaVerT includes compiling the annotated JavaScript (§3) program to a JSIL (§4) program using the JS-2-JSIL compiler (§5); translating JavaScript annotations using the JS-2-JSIL logic translator to equivalent JSIL annotations; and automatically verifying the resulting annotated JSIL program by JSIL Verify (§6), making use of the verified JS-2-JSIL environment (§7).

9. Conclusion

9.1. Summary of Thesis Achievements

JaVerT. We believe that JaVerT constitutes an important step towards scalable verification of real-world JavaScript programs. It successfully tackles a number of challenges that are critical for tractable reasoning about JavaScript. JaVerT provides the *JavaScript Assertion Language*, which includes key abstractions to allow the user to capture fundamental JavaScript concepts without exposing the internal features of the language. JaVerT contains the complexity of reasoning about JavaScript programs by providing a *JavaScript Frontend JS-2-JSIL* to the *JSIL verification infrastructure*. We demonstrated how a priority queue library can be specified to ensure *prototype safety* and explained how JaVerT uses JSIL verification infrastructure through the JavaScript Frontend to verify that JavaScript programs satisfy their specifications.

JavaScript Assertion Language. Our first challenge **(C1)** was to design assertions that capture common heap structures of JavaScript. We provide key abstractions that allow the user to capture fundamental JavaScript concepts: Scope and OChains to reason about full variable scoping and Pi to capture the prototype inheritance of JavaScript. Pi and OChains are carefully designed to resolve the tension between the overlapping of prototype and scope chains and the heap separation inherent to separation logic. We specified a priority queue library, written in a typical OO-style. We have demonstrated that a user can write JavaScript specifications with a minimal knowledge of JavaScript internals. We have illustrated how to specify the library to ensure *prototype safety* and verify its clients that do not compromise prototype safety. We also discussed a possible way to treat the lack of encapsulation in JavaScript using opaque predicates.

JSIL Verification Infrastructure. JavaScript verification requires a dedicated low-level control-flow-based intermediate representation. We have developed a simple JavaScript intermediate representation for our verification toolchain, called JSIL. It comprises only the most basic control flow commands (unconditional and conditional gotos), the object management commands needed to support extensible objects and dynamic property accesses, and top-level procedures. Our third verification challenge **(C3)** was to handle the dynamic behaviour associated with extensible object, dynamic property accesses and dynamic function calls, which introduce an additional level of complexity compared with the static features in the IRs underlying the familiar separation-logic tools. We have developed a sound program logic for JSIL, which is the basis for JSIL Verify, the first verification tool based on separation logic to natively support these fundamental dynamic features of JavaScript.

The JS-2-JSIL Compiler. Our second challenge **(C2)** was to support JavaScript statements with all of their complicated control flows. We have presented the JS-2-JSIL compiler from JavaScript to JSIL. We designed the JS-2-JSIL compiler so that the compiled code, and oftentimes the compiler itself, follows the ECMAScript standard line-by-line. This semantics-driven compilation is feasible,

because the ECMAScript standard is given operationally, in an almost pseudo-code format. Given the complexity of JavaScript, this approach, albeit quite informal in nature, can give some confidence to compiler correctness. Ultimately, however, it is not formal enough to be sufficient on its own.

We gave a pen-and-paper correctness proof for a representative fragment of the language. It required formalising the semantics and memory model of JavaScript, formalising the semantics and memory model of JSIL, and proving that the semantics of the JavaScript and compiled JSIL code match. We have given thought to providing a Coq proof of correctness, leveraging on our previous JSCert mechanised specification of JavaScript [9]. However, the process of formalising JSIL and JS-2-JSIL, and then proving the correctness was beyond our manpower.

We believe that testing is an indispensable part of establishing compiler correctness for JavaScript. Regardless of how precise proof of correctness may be, there still is plenty of room for discrepancies to arise: for example, the implementation of the compiler might inadvertently deviate from its formalisation; or the formalised JavaScript semantics might deviate from the standard. We have substantially tested the JS-2-JSIL compiler using the ECMAScript Test262 [23].

We note that the construction of scope clarification function currently requires the entire program. To achieve a more modular translation from JavaScript to JSIL, we would need to revisit the construction of the scope clarification function. This will be required for supporting the module system of the language, which is not addressed explicitly in the ES5 standard but is part of the ES6 standard.

The JS-2-JSIL Logic Translator. We have presented the JS-2-JSIL logic translator from JS Logic to JSIL Logic to make use of the JSIL verification infrastructure. To validate the JS-2-JSIL logic translator, we needed to formally connect JavaScript verification with JSIL verification. To achieve that, a fundamental decision was to make the JavaScript and JSIL memory models identical to each other as possible. To be able to formally lift JSIL verification to JavaScript verification, we gave a strong correspondence between JavaScript and JSIL assertions, relating the semantics of JavaScript triples with the semantics of the JSIL triples, and used the soundness result for the JSIL proof rules from the JSIL verification infrastructure. We validated the JS-2-JSIL logic translator by establishing a full correctness result for the assertion languages, and a partial correctness result for the triples. Relating the semantics of JavaScript triples with the semantics of the JSIL triples requires the correctness of the JS-2-JSIL compiler, which we established for a fragment of the language.

The JS-2-JSIL Environment. We have introduced reference implementations and axiomatic specifications for JavaScript internal functions to solve our fourth challenge **(C4)**. There were two options on how to use reference implementations in verification: inlining and axiomatic specification.

Inlining the bodies of the internal functions was not a viable option. Given the sheer number of calls to the internal functions and their intertwined nature, the size of the compiled code would quickly spiral out of control. We would also entirely lose the visual correspondence between the compiled code and the standard.

Without inlining, on the other hand, the calls to internal functions are featured in the compiled code as procedure calls to their JSIL implementations. In that sense, the compiled code reflects the English standard. In such a situation, we provide axiomatic specifications to the internal functions. During verification, the only check that has to be made is that the current symbolic state entails a precondition of the specification.

Creating axiomatic specifications does not come without its challenges. The definitions of the

internal functions are often intertwined, making it difficult to fully grasp the control flow and allowed behaviours. Specifying such dependencies axiomatically involved the joining of the specifications of all nested function calls at the top level, which resulted in numerous branchings.

The JSIL reference implementations of JavaScript internal functions and built-in libraries are step-by-step faithful to the standard, tested together with JS-2-JSIL compiler using Test262, and verified with respect to their axiomatic specifications using the JSIL verification infrastructure.

9.2. Open Problems

Higher-order reasoning. One of the main challenges related to JavaScript verification is reasoning about higher-order functions of arbitrary complexity. JavaScript has full support for higher-order functions, meaning that a function can take another function as an argument, or that a function can return another function as a result. This behaviour is not easily captured, particularly in a program logic setting, but is often used in practice and verification of JavaScript programs should ultimately be able to tackle it. One possible direction would be to extend JSIL Logic with higher-order reasoning by encoding JSIL Logic in Iris [40], to reason about JavaScript getters/setters and arbitrary functions passed as parameters.

Bi-abduction. We believe that the semi-automatic JaVerT toolchain has a role to play in the development of functionally correct specifications of critical libraries. However, writing specifications for non-critical JavaScript code is infeasible. For verifying properties of large JavaScript codebases, an automated tool based on bi-abduction [13] is necessary.

Other Symbolic Execution Tools. We believe that the JSIL language together with the JS-2-JSIL compiler can be used by different static analysis tools, such as Infer [14], CBMC [42], Viper [49], and Rosette [69, 68], to develop a static analysis tool for JavaScript. It is much simpler to develop a JSIL frontend compared to a JavaScript frontend.

We have started building a prototype JSIL frontend to CBMC [64], with the aim of finding cross-site scripting vulnerabilities. Also, a JSIL frontend is being developed to Rosette [69, 68], where the aim is to use the symbolic execution of Rosette to obtain a bug-finding tool for JavaScript.

ECMAScript 6. Moving JaVerT to ES6 Strict would essentially require extending the current JS-2-JSIL compiler with new ES6 language constructs. The existing specifications of the internal functions would remain the same and our predicate abstractions would be directly relevant.

It would also be possible to move to ES6. This would require modelling scope lookup using an inductive predicate for capturing the footprint of a dynamic scope chain traversal, similarly to [30].

Bibliography

[1] ECMA 262. ECMAScript Language Specification. Technical report, ECMA.

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

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

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

[5] J. Berdine, C. Calcagno, and P. O'Hearn. Symbolic execution with separation logic. In *APLAS*, 2005.

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

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

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

[9] Martin Bodin, Arthur Charguéraud, Daniele Filaretti, Philippa Gardner, Sergio Maffeis, Daiva Naudziuniene, Alan Schmitt, and Gareth Smith. A Trusted Mechanised JavaScript Specification. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL'14, pages 87–100. ACM Press, 2014.

[10] Martin Bodin, Thomas Jensen, and Alan Schmitt. Certified abstract interpretation with pretty-big-step semantics. In *Proceedings of the 2015 Conference on Certified Programs and Proofs*, pages 29–40. ACM, 2015.

[11] Matko Botinčan, Dino Distefano, Mike Dodds, Radu Griore, Daiva Naudžiūnienė, and Matthew J. Parkinson. coreStar: The core of jstar. In *Boogie*, 2011.

[12] Cristian Cadar, Daniel Dunbar, and Dawson R. Engler. KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs. In Richard Draves and Robbert

van Renesse, editors, *8th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2008, December 8-10, 2008, San Diego, California, USA, Proceedings*, pages 209–224. USENIX Association, 2008.

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

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

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

[16] Junhee Cho and Sukyoung Ryu. Javascript module system: exploring the design space. In *Proceedings of the 13th international conference on Modularity*, pages 229–240. ACM, 2014.

[17] Ravi Chugh, David Herman, and Ranjit Jhala. Dependent types for javascript. *SIGPLAN Not.*, 47(10):587–606, October 2012.

[18] Arlen Cox, Bor-Yuh Evan Chang, and Xavier Rival. Automatic analysis of open objects in dynamic language programs. In *Static Analysis - 21st International Symposium, SAS 2014, Munich, Germany, September 11-13, 2014. Proceedings*, pages 134–150, 2014.

[19] Andrei Ştefănescu, Daejun Park, Shijiao Yuwen, Yilong Li, and Grigore Roşu. Semantics-based program verifiers for all languages. In *Proceedings of the 31th Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'16)*, pages 74–91. ACM, Nov 2016.

[20] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. An efficient method of computing static single assignment form. In *Conference Record of the Sixteenth Annual ACM Symposium on Principles of Programming Languages, Austin, Texas, USA, January 11-13, 1989*, pages 25–35. ACM Press, 1989.

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

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

[23] ECMAScript Committee. Test262 test suite. https://github.com/tc39/test262, 2017.

[24] Facebook. Flow: a static type checker for javascript. https://flowtype.org/.

[25] Facebook. react.js. https://facebook.github.io/react/.

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

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

[28] David Flanagan. *JavaScript - The Definitive Guide*. O'Reilly, 2011.

[29] P. Gardner, D. Naudžiūnienė, and G. Smith. JuS: Squeezing the sense out of JavaScript programs. In *Second Annual Workshop on Tools for JavaScript Analysis*, 2013.

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

[31] Google. v8. http://v8project.blogspot.co.uk.

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

[33] Daniel Hedin and Andrei Sabelfeld. Information-flow security for a core of JavaScript. In *Proceedings of the 25th IEEE Computer Security Foundations Symposium*, CSF'12, pages 3–18. IEEE Computer Society, 2012.

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

[35] Charles Antony Richard Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 1969.

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

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

[38] jQuery: a fast, small, and feature-rich JavaScript library. `https://jquery.com/`.

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

[40] Ralf Jung, David Swasey, Filip Sieczkowski, Kasper Svendsen, Aaron Turon, Lars Birkedal, and Derek Dreyer. Iris: Monoids and invariants as an orthogonal basis for concurrent reasoning. In *Proceedings of the 42Nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '15, pages 637–650, New York, NY, USA, 2015. ACM.

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

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

[43] Hongki Lee, Sooncheol Won, Joonho Jin, Junhee Cho, and Sukyoung Ryu. Safe: Formal specification and implementation of a scalable analysis framework for ecmascript. In *Proceedings of the 2012 International Workshop on Foundations of Object-Oriented Languages*, 2012.

[44] Benjamin S Lerner, Liam Elberty, Jincheng Li, and Shriram Krishnamurthi. Combining form and function: Static types for jquery programs. In *European Conference on Object-Oriented Programming*, pages 79–103. Springer, 2013.

[45] Benjamin S Lerner, Joe Gibbs Politz, Arjun Guha, and Shriram Krishnamurthi. Tejas: retrofitting type systems for javascript. In *ACM SIGPLAN Notices*, volume 49, pages 1–16. ACM, 2013.

[46] Magnus Madsen, Benjamin Livshits, and Michael Fanning. Practical static analysis of javascript applications in the presence of frameworks and libraries. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, pages 499–509. ACM, 2013.

[47] Sergio Maffeis, John C. Mitchell, and Ankur Taly. An operational semantics for JavaScript. In *Proceedings of the 6th Asian Symposium on Programming Languages and Systems*, volume 5356 of *Lecture Notes in Computer Science*, pages 307–325. Springer, 2008.

[48] Microsoft. TypeScript language specification. Technical report, Microsoft, 2014.

[49] P. Müller, M. Schwerhoff, and A. J. Summers. Viper: A verification infrastructure for permission-based reasoning. In B. Jobstmann and K. R. M. Leino, editors, *Verification, Model Checking, and Abstract Interpretation (VMCAI)*, volume 9583 of *LNCS*, pages 41–62. Springer-Verlag, 2016.

[50] Peter OHearn, John Reynolds, and Hongseok Yang. Local reasoning about programs that alter data structures. In *International Workshop on Computer Science Logic*, pages 1–19. Springer, 2001.

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

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

[53] Matthew Parkinson and Gavin Bierman. Separation logic and abstraction. In *ACM SIGPLAN Notices*, volume 40, pages 247–258. ACM, 2005.

[54] Pieter Philippaerts, Jan Tobias Mühlberg, Willem Penninckx, Jan Smans, Bart Jacobs, and Frank Piessens. Software verification with verifast: Industrial case studies. *Science of Computer Programming*, 82:77–97, 2014.

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

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

[57] John C. Reynolds. Separation logic: A logic for shared mutable data structures. In *LICS*, 2002.

[58] Grigore Rosu and Traian Florin Serbănută. An overview of the k semantic framework. *The Journal of Logic and Algebraic Programming*, 79(6):397–434, 2010.

[59] Grigore Rosu and Andrei Stefanescu. Checking reachability using matching logic. In *OOPSLA*, 2012.

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

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

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

[63] Ankur Taly, Úlfar Erlingsson, John C. Mitchell, Mark S. Miller, and Jasvir Nagra. Automated analysis of security-critical javascript apis. In *Proceedings of the 2011 IEEE Symposium on Security and Privacy*, SP '11, pages 363–378, Washington, DC, USA, 2011. IEEE Computer Society.

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

[65] JSIL Team. JSIL as a Service. http://goo.gl/au69SV, 2016.

[66] JSIL Team. The source code of JSIL. https://github.com/resource-reasoning/JavaScriptVerification, 2017.

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

[68] Emina Torlak and Rastislav Bodík. Growing solver-aided languages with rosette. In Antony L. Hosking, Patrick Th. Eugster, and Robert Hirschfeld, editors, *ACM Symposium on New Ideas in Programming and Reflections on Software, Onward! 2013, part of SPLASH '13, Indianapolis, IN, USA, October 26-31, 2013*, pages 135–152. ACM, 2013.

[69] Emina Torlak and Rastislav Bodík. A lightweight symbolic virtual machine for solver-aided host languages. In Michael F. P. O'Boyle and Keshav Pingali, editors, *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14, Edinburgh, United Kingdom - June 09 - 11, 2014*, page 54. ACM, 2014.

[70] Panagiotis Vekris, Benjamin Cosman, and Ranjit Jhala. Refinement types for typescript. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 310–325. ACM, 2016.

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

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

A. Pretty-Big-Step Semantics of a Fragment of ES5 Strict

A.1. Notation

Auxiliary Functions.

$$l \mapsto \{p_1 : v_1, \ldots, p_n : v_n\} \triangleq (l, p_1) \mapsto v_1 \uplus \cdots \uplus (l, p_n) \mapsto v_n$$

$$\mathsf{ER}(l.\mathsf{a}x) \triangleq \mathsf{a} = \mathsf{v}$$

$$\mathsf{fun}(l, L, l', m) \triangleq l \mapsto \{@proto : l_{fp}, @class : \text{“}\texttt{Function}\text{”}, @extensible : \mathsf{true}, \text{“}\texttt{prototype}\text{”} : l', @scope : L, @code : m\}$$

$$\mathsf{env}(l_s, \overline{x}, \overline{v}, s) \triangleq \left(\uplus_{i=1}^{n}(l_s, \overline{x}_i) \mapsto \overline{v}_i\right) \uplus \left(\uplus_{i=1}^{m}(l_s, y_i) \mapsto \mathsf{undefined}\right), \text{ where: } \mathsf{defs}(s) = \{y_1, ..., y_m\}$$

$$\mathsf{err}(l, l_p) \triangleq l \mapsto \{@proto : l_p, @class : \text{“}\texttt{Error}\text{”}, @extensible : \mathsf{true}\}$$

$$\mathcal{I}_{tb}(v) \triangleq \begin{cases} \mathsf{false} & \text{if } v \in \{\mathsf{null}, \mathsf{undefined}, \mathsf{false}, +0, -0, \mathsf{NaN}, \text{“”}\} \\ \mathsf{true} & \text{otherwise} \end{cases}$$

$$\mathcal{I}^{prim}_{ts}(v) \triangleq \begin{cases} \text{“undefined”} & \text{if } v = \mathsf{undefined} \\ \text{“null”} & \text{if } v = \mathsf{null} \\ \text{“true”} & \text{if } v = \mathsf{true} \\ \text{“false”} & \text{if } v = \mathsf{false} \\ \texttt{NumberToString}(v) & \text{if } v \in \mathcal{N}um \\ v & \text{if } v \in \mathcal{S}tr \end{cases}$$

$$\mathcal{I}^{prim}_{tn}(v) \triangleq \begin{cases} \mathsf{NaN} & \text{if } v = \mathsf{undefined} \\ +0 & \text{if } v = \mathsf{null} \vee v = \mathsf{false} \\ 1 & \text{if } v = \mathsf{true} \\ v & \text{if } v \in \mathcal{N}um \\ \texttt{StringToNumber}(v) & \text{if } v \in \mathcal{S}tr \end{cases}$$

$$\mathcal{I}^{prim}_{ti}(v) \triangleq \begin{cases} +0 & \text{if } n = \mathsf{NaN} \\ v & \text{if } n \in \{+0, -0, +\infty, -\infty)\} \\ \texttt{sign}(n) \times \texttt{floor}(\texttt{abs}(n)) & \text{otherwise, where } n = \mathcal{I}^{prim}_{tn}(v) \end{cases}$$

$$\mathcal{I}_{type}(v) \triangleq \begin{cases} \text{Undefined} & \text{if } v = \mathsf{undefined} \\ \text{Null} & \text{if } v = \mathsf{null} \\ \text{Boolean} & \text{if } v \in \mathcal{B}ool \\ \text{Number} & \text{if } v \in \mathcal{N}um \\ \text{String} & \text{if } v \in \mathcal{S}tr \\ \text{Object} & \text{if } v \in \mathcal{L} \end{cases}$$

$$\mathcal{I}_{typeof}(h, v) \triangleq \begin{cases} \text{“undefined”} & \text{if } v = \mathsf{undefined} \\ \text{“null”} & \text{if } v = \mathsf{null} \\ \text{“boolean”} & \text{if } v \in \mathcal{B}ool \\ \text{“number”} & \text{if } v \in \mathcal{N}um \\ \text{“string”} & \text{if } v \in \mathcal{S}tr \\ \text{“object”} & \text{if } v \in \mathcal{L} \ \wedge (v, @code) \notin \mathsf{dom}(h) \\ \text{“function”} & \text{if } v \in \mathcal{L} \ \wedge (v, @code) \in \mathsf{dom}(h) \end{cases}$$

$$\mathcal{I}_{===}(v_1, v_2) \triangleq \begin{cases} \mathsf{false} & \text{if } \mathcal{I}_{type}(v_1) \neq \mathcal{I}_{type}(v_2) \\ \mathsf{true} & \text{if } (v_1 = -0 \wedge v_2 = +0) \vee (v_1 = +0 \wedge v_2 = -0) \\ \mathsf{false} & \text{if } \mathcal{I}_{type}(v_1) = \mathcal{I}_{type}(v_2) = \text{Number} \wedge (v_1 = \mathsf{NaN} \vee v_2 = \mathsf{NaN}) \\ v_1 = v_2 & \text{otherwise} \end{cases}$$

$$\mathsf{SelectProto}(v) \triangleq \begin{cases} l & \text{if } v \in \mathcal{L} \\ l_{op} & \text{otherwise} \end{cases}$$

$$\mathsf{SelectThis}(w) \triangleq \begin{cases} l & \text{if } w = l.{}_{\mathsf{o}}x \\ \mathsf{undefined} & \text{otherwise} \end{cases}$$

$$\mathsf{FunRet}(o) \triangleq \begin{cases} \mathsf{undefined} & \text{if } o = v \text{ or } o = \mathsf{empty} \\ v & \text{if } o = \mathsf{ret}\, v \\ \mathsf{error}\, v & \text{if } o = \mathsf{error}\, v \end{cases}$$

$$\mathsf{ConsRet}(o, l) \triangleq \begin{cases} \mathsf{error}\, v & \text{if } o = \mathsf{error}\, v \\ l' & \text{if } o = \mathsf{ret}\, l' \\ l & \text{otherwise} \end{cases}$$

$$\mathsf{defs}(s) \triangleq \begin{cases} \emptyset & \text{if } s \in \mathcal{E}_{\mathsf{JS}} \\ \{x\} & \text{if } s = \mathsf{var}\, x \\ \mathsf{defs}(s_1) \cup \mathsf{defs}(s_2) & \text{if } s = s_1; s_2 \\ \mathsf{defs}(s_1) \cup \mathsf{defs}(s_2) & \text{if } s = \mathsf{if}(e)\, \{s_1\}\, \mathsf{else}\, \{s_2\} \\ \mathsf{defs}(s) & \text{if } s = \mathsf{while}(e)\, \{s\} \\ \emptyset & \text{if } s = \mathsf{break} \\ \emptyset & \text{if } s = \mathsf{throw}\, e \\ \emptyset & \text{if } s = \mathsf{return}\, e \end{cases}$$

Heap update and cell deallocation: $h[(l, x) \mapsto \omega]$ **and** $h \backslash l.{}_{\mathsf{a}}x$**.**

$$\frac{(l, x) \notin \mathsf{dom}(h)}{h[(l, x) \mapsto \omega] \triangleq h \uplus (l, x) \mapsto \omega} \qquad \frac{h = h' \uplus (l, x) \mapsto \omega'}{h[(l, x) \mapsto \omega] \triangleq h \uplus (l, x) \mapsto \omega}$$

$$\frac{(l, x) \notin \mathsf{dom}(h)}{h \backslash l.{}_{\mathsf{a}}x \triangleq h} \qquad \frac{h = h' \uplus (l, x) \mapsto v}{h \backslash l.{}_{\mathsf{a}}x \triangleq h'}$$

A.2. Expressions and Statements

PBS Semantics for Expressions: $\wp, L, v_t \vdash \langle h, e\rangle \Downarrow_m \langle h, o\rangle$**.**

This

$$\wp, \_, v_t \vdash \langle h, \mathsf{this}\rangle \Downarrow_m \langle h, v_t\rangle$$

Variable

$$\frac{\wp, L, v_t \vdash \langle h, \mathcal{I}_\sigma(x)\rangle \Downarrow_m \langle h_1, o_1\rangle \quad \wp, L, v_t \vdash \langle h_1, id(x, o_1)\rangle \Downarrow_m \langle h_f, o_f\rangle}{\wp, L, v_t \vdash \langle h, x\rangle \Downarrow_m \langle h_f, o_f\rangle}$$

Variable-ref

$$\vdash \langle h, id(x, v)\rangle \Downarrow_m \langle h, v.{}_{\mathsf{v}}x\rangle$$

Literal

$$\vdash \langle h, \lambda\rangle \Downarrow_m \langle h, \lambda\rangle$$

Object Literal

$$\frac{h_f = h \uplus l \mapsto \{@\mathit{proto}\colon l_{op},\ @\mathit{class}\colon \text{“}\texttt{Object}\text{”},\ @\mathit{extensible}\colon \mathsf{true}\}}{\vdash \langle h, \{\,\}\rangle \Downarrow_m \langle h_f, l\rangle}$$

Computed Access

$$\frac{\wp, L, v_t \vdash \langle h, e_1\rangle \Downarrow_m^{\gamma} \langle h_1, o_1\rangle \quad \wp, L, v_t \vdash \langle h_1, o_1[e_2]_1\rangle \Downarrow_m \langle h_f, o_f\rangle}{\wp, L, v_t \vdash \langle h, e_1[e_2]\rangle \Downarrow_m \langle h_f, o_f\rangle}$$

Computed Access - 1

$$\frac{\wp, L, v_t \vdash \langle h, e_2\rangle \Downarrow_m^{\gamma} \langle h_1, o_1\rangle \quad \wp, L, v_t \vdash \langle h_1, v_1[o_1]_2\rangle \Downarrow_m \langle h_f, o_f\rangle}{\wp, L, v_t \vdash \langle h, v_1[e_2]_1\rangle \Downarrow_m \langle h_f, o_f\rangle}$$

Computed Access - 2

$$\frac{\wp, L, v_t \vdash \langle h, \mathcal{I}_{coc}(v_1)\rangle \Downarrow_m \langle h_1, o_1\rangle \quad \wp, L, v_t \vdash \langle h_1, v_1[v_2](o_1)_3\rangle \Downarrow_m \langle h_f, o_f\rangle}{\wp, L, v_t \vdash \langle h, v_1[v_2]_2\rangle \Downarrow_m \langle h_f, o_f\rangle}$$

Computed Access - 3

$$\frac{\wp, L, v_t \vdash \langle h, \mathcal{I}_{ts}(v_2)\rangle \Downarrow_m \langle h_1, o_1\rangle \quad \wp, L, v_t \vdash \langle h_1, v_1[o_1]_4\rangle \Downarrow_m \langle h_f, o_f\rangle}{\wp, L, v_t \vdash \langle h, v_1[v_2](\mathsf{empty})_3\rangle \Downarrow_m \langle h_f, o_f\rangle}$$

Computed Access - 4

$$\vdash \langle h, v[p]_4\rangle \Downarrow_m \langle h, v.{}_{\mathsf{o}}p\rangle$$

Constructor Call

$$\frac{\wp, L, v_t \vdash \langle h, e\rangle \Downarrow^{\gamma}_{m} \langle h_1, o_1\rangle \quad \wp, L, v_t \vdash \langle h_1, \mathsf{new}_1\ o_1(\overline{e})\rangle \Downarrow_m \langle h_f, o_f\rangle}{\wp, L, v_t \vdash \langle h, \mathsf{new}\ e(\overline{e})\rangle \Downarrow_m \langle h_f, o_f\rangle}$$

Constructor Call - 1

$$\frac{\wp, L, v_t \vdash \langle h, \mathsf{iterate}\{\overline{e}\}\rangle \Downarrow_m \langle h_1, \overline{v}\rangle \quad \wp, L, v_t \vdash \langle h_1, \mathsf{new}_2\ v(\overline{v})\rangle \Downarrow_m \langle h_f, o_f\rangle}{\wp, L, v_t \vdash \langle h, \mathsf{new}_1\ v(\overline{e})\rangle \Downarrow_m \langle h_f, o_f\rangle}$$

Constructor Call - 2 (Fault)

$$\frac{\neg\mathcal{P}_c(h, v) \quad h_f = h \uplus \mathsf{err}(l, l_{tep})}{\vdash \langle h, \mathsf{new}_2\ v(\overline{v})\rangle \Downarrow_m \langle h_f, \mathsf{error}\, l\rangle}$$

Constructor Call - 2

$$\frac{\begin{array}{l}\mathcal{P}_c(h, l) \quad v = h(l, \text{"}\mathsf{prototype}\text{"}) \quad l' = \mathsf{SelectProto}(v)\\ h_1 = h \uplus l_o \mapsto \{@proto\colon l',\ @class\colon \text{“}\texttt{Object}\text{”},\ @extensible\colon \mathsf{true}\}\\ m' = h(l, @code) \quad L = h(l, @scope)\\ \wp(m') = \lambda x_1, ...x_{n_2}.s\\ \forall_{1\le n\le n_1} v'_n = v_n\\ \forall_{n_1<n\le n_2} v'_n = \mathsf{undefined}\\ \wp, L, l_o \vdash \langle h, m'(\overline{x}, \overline{v'})\rangle \Downarrow_{m'} \langle h_f, o\rangle\end{array}}{\wp, \_, \_ \vdash \langle h, \mathsf{new}_2\ l(v_1, ...v_{n_1})\rangle \Downarrow_m \langle h_f, \mathsf{ConsRet}(o, l_o)\rangle}$$

Function Call

$$\frac{\wp, L, v_t \vdash \langle h, e\rangle \Downarrow_m \langle h_1, o_1\rangle \quad \wp, L, v_t \vdash \langle h_1, o_1(\overline{e})_1\rangle \Downarrow_m \langle h_f, o_f\rangle}{\wp, L, v_t \vdash \langle h, e(\overline{e})\rangle \Downarrow_m \langle h_f, o_f\rangle}$$

Function Call - 1

$$\frac{\wp, L, v_t \vdash \langle h, \mathcal{I}_{gv}(w)\rangle \Downarrow_m \langle h_1, o_1\rangle \quad \wp, L, v_t \vdash \langle h_1, (w, o_1)(\overline{e})_2\rangle \Downarrow_m \langle h_f, o_f\rangle}{\wp, L, v_t \vdash \langle h, w(\overline{e})_1\rangle \Downarrow_m \langle h_f, o_f\rangle}$$

Function Call - 2

$$\frac{\wp, L, v_t \vdash \langle h, \mathsf{iterate}\{\overline{e}\}\rangle \Downarrow_m \langle h_1, \overline{v}\rangle \quad \wp, L, v_t \vdash \langle h_1, (w, v)(\overline{v})_3\rangle \Downarrow_m \langle h_f, o_f\rangle}{\wp, L, v_t \vdash \langle h, (w, v)(\overline{e})_2\rangle \Downarrow_m \langle h_f, o_f\rangle}$$

Function Call - 3

$$\frac{\begin{array}{l}\mathcal{P}_c(h, v) \quad v_t = \mathsf{SelectThis}(w)\\ m' = h(v, @code) \quad L = h(v, @scope)\\ \wp(m') = \lambda x_1, ...x_{n_2}.s\\ \forall_{1\le n\le n_1} v'_n = v_n\\ \forall_{n_1<n\le n_2} v'_n = \mathsf{undefined}\\ \wp, L, v_t \vdash \langle h, m'(\overline{x}, \overline{v'})\rangle \Downarrow_{m'} \langle h_f, o_f\rangle\end{array}}{\wp, \_, \_ \vdash \langle h, (w, v)(v_1, ...v_{n_1})_3\rangle \Downarrow_m \langle h_f, o_f\rangle}$$

Function Call - 3 (Fault)

$$\frac{\neg\mathcal{P}_c(h, v) \quad h_f = h \uplus \mathsf{err}(l, l_{tep})}{\vdash \langle h, (w, v)(\overline{v})_3\rangle \Downarrow_m \langle h_f, \mathsf{error}\, l\rangle}$$

Function Literal

$$\frac{h_f = h \uplus l' \mapsto \{@proto\colon l_{op},\ @class\colon \text{“}\texttt{Object}\text{”},\ @extensible\colon \mathsf{true}\} \uplus \mathsf{fun}(l, L, l', m)}{\wp, L, \_ \vdash \langle h, \mathsf{function}\,(\overline{x})\{s\}^m\rangle \Downarrow_m \langle h_f, l\rangle}$$

Delete

$$\frac{\wp, L, v_t \vdash \langle h, e\rangle \Downarrow_m \langle h_1, o_1\rangle \quad \wp, L, v_t \vdash \langle h_1, \mathsf{del}_1\ o_1\rangle \Downarrow_m \langle h_f, o_f\rangle}{\wp, L, v_t \vdash \langle h, \mathsf{delete}\, e\rangle \Downarrow_m \langle h_f, o_f\rangle}$$

Delete - 1 (Not A Ref)

$$\vdash \langle h_1, \mathsf{del}_1\ v\rangle \Downarrow_m \langle h, \mathsf{true}\rangle$$

Delete - 1 (Fault)

$$\frac{v = \mathsf{undefined} \vee \mathsf{ER}(v._{\mathsf{a}}p) \quad h_f = h \uplus \mathsf{err}(l, l_{sep})}{\vdash \langle h, \mathsf{del}_1\ v._{\mathsf{a}}p\rangle \Downarrow_m \langle h_f, \mathsf{error}\, l\rangle}$$

Delete - 1 (Prop Ref)

$$\frac{\wp, L, v_t \vdash \langle h, \mathcal{I}_{to}(v)\rangle \Downarrow_m \langle h_1, o_1\rangle \quad \wp, L, v_t \vdash \langle h_1, \mathsf{del}_2\ o_1.{}_{\mathsf{o}}p\rangle \Downarrow_m \langle h_f, o_f\rangle}{\wp, L, v_t \vdash \langle h, \mathsf{del}_1\ v.{}_{\mathsf{o}}p\rangle \Downarrow_m \langle h_f, o_f\rangle}$$

Delete - 2

$$\frac{\wp, L, l \vdash \langle h, \mathcal{I}_d(p, \mathsf{true})\rangle \Downarrow_m \langle h_f, o_f\rangle}{\wp, L, v_t \vdash \langle h, \mathsf{del}_2\ l.{}_{\mathsf{o}}p\rangle \Downarrow_m \langle h_f, o_f\rangle}$$

TYPEOF

$$\frac{\wp, L, v_t \vdash \langle h, e\rangle \Downarrow_m \langle h_1, o\rangle \quad \wp, L, v_t \vdash \langle h_1, \mathsf{typeof}_1\ o\rangle \Downarrow_m \langle h_f, o_f\rangle}{\wp, L, v_t \vdash \langle h, \mathsf{typeof}\, e\rangle \Downarrow_m \langle h_f, o_f\rangle}$$

TYPEOF - 1 (UNRESOLVED)

$$\vdash \langle h, \mathsf{typeof}_1\ \mathsf{undefined}.{}_{\mathsf{a}}p\rangle \Downarrow_m \langle h, \text{“undefined”}\rangle$$

TYPEOF -1 (REF)

$$\frac{v \neq \mathsf{undefined} \quad \wp, L, v_t \vdash \langle h, \mathcal{I}_{gv}(v.{}_{\mathsf{a}}p)\rangle \Downarrow_m \langle h_1, o_1\rangle \quad \wp, L, v_t \vdash \langle h_1, \mathsf{typeof}_1\ o\rangle \Downarrow_m \langle h_f, o_f\rangle}{\wp, L, v_t \vdash \langle h, \mathsf{typeof}_1\ v.{}_{\mathsf{a}}p\rangle \Downarrow_m \langle h_f, o_f\rangle}$$

TYPEOF - 1 (VAL)

$$\vdash \langle h, \mathsf{typeof}_1\ v\rangle \Downarrow_m \langle h, \mathcal{I}_{typeof}(h, v)\rangle$$

ADDITION OPERATOR

$$\frac{\wp, L, v_t \vdash \langle h, e_1\rangle \Downarrow_m^{\gamma} \langle h_1, o_1\rangle \quad \wp, L, v_t \vdash \langle h_1, o_1\ +_1\ e_2\rangle \Downarrow_m \langle h_f, o_f\rangle}{\wp, L, v_t \vdash \langle h, e_1\ +\ e_2\rangle \Downarrow_m \langle h_f, o_f\rangle}$$

ADDITION OPERATOR - 1

$$\frac{\wp, L, v_t \vdash \langle h, e_2\rangle \Downarrow_m^{\gamma} \langle h_1, o_1\rangle \quad \wp, L, v_t \vdash \langle h_1, v_1\ +_2\ o_1\rangle \Downarrow_m \langle h_f, o_f\rangle}{\wp, L, v_t \vdash \langle h, v_1\ +_1\ e_2\rangle \Downarrow_m \langle h_f, o_f\rangle}$$

ADDITION OPERATOR - 2

$$\frac{\wp, L, v_t \vdash \langle h, \mathcal{I}_{tp}(v_1)\rangle \Downarrow_m \langle h_1, o\rangle \quad \wp, L, v_t \vdash \langle h_1, o\ +_3\ v_2\rangle \Downarrow_m \langle h_f, o_f\rangle}{\wp, L, v_t \vdash \langle h, v_1\ +_2\ v_2\rangle \Downarrow_m \langle h_f, o_f\rangle}$$

ADDITION OPERATOR - 3

$$\frac{\wp, L, v_t \vdash \langle h, \mathcal{I}_{tp}(v_2)\rangle \Downarrow_m \langle h_1, o\rangle \quad \wp, L, v_t \vdash \langle h_1, v_1\ +_4\ o\rangle \Downarrow_m \langle h_f, o_f\rangle}{\wp, L, v_t \vdash \langle h, v_1\ +_3\ v_2\rangle \Downarrow_m \langle h_f, o_f\rangle}$$

ADDITION OPERATOR - CONCAT

$$\frac{v_1 \in \mathcal{S}tr \vee v_2 \in \mathcal{S}tr \quad v = \mathcal{I}_{ts}^{prim}(v_1)\ \cdot\ \mathcal{I}_{ts}^{prim}(v_2)}{\vdash \langle h, v_1\ +_4\ v_2\rangle \Downarrow_m \langle h, v\rangle}$$

ADDITION OPERATOR - PLUS

$$\frac{v_1 \notin \mathcal{S}tr \wedge v_2 \notin \mathcal{S}tr \quad v = \mathcal{I}_{tn}^{prim}(v_1)\ +\ \mathcal{I}_{tn}^{prim}(v_2)}{\vdash \langle h, v_1\ +_4\ v_2\rangle \Downarrow_m \langle h, v\rangle}$$

STRICT EQUALITY

$$\frac{\wp, L, v_t \vdash \langle h, e_1\rangle \Downarrow_m^{\gamma} \langle h_1, o_1\rangle \quad \wp, L, v_t \vdash \langle h_1, o_1\ ===_1\ e_2\rangle \Downarrow_m \langle h_f, o_f\rangle}{\wp, L, v_t \vdash \langle h, e_1\ ===\ e_2\rangle \Downarrow_m \langle h_f, o_f\rangle}$$

STRICT EQUALITY - 1

$$\frac{\wp, L, v_t \vdash \langle h, e_2\rangle \Downarrow_m^{\gamma} \langle h_1, o_1\rangle \quad \wp, L, v_t \vdash \langle h_1, v_1\ ===_2\ o_1\rangle \Downarrow_m \langle h_f, o_f\rangle}{\wp, L, v_t \vdash \langle h, v_1\ ===_1\ e_2\rangle \Downarrow_m \langle h_f, o_f\rangle}$$

STRICT EQUALITY - 2

$$\vdash \langle h, v_1\ ===_2\ v_2\rangle \Downarrow_m \langle h, \mathcal{I}_{===}(v_1, v_2)\rangle$$

STRICT INEQUALITY

$$\frac{\wp, L, v_t \vdash \langle h, e_1\rangle \Downarrow_m^{\gamma} \langle h_1, o_1\rangle \quad \wp, L, v_t \vdash \langle h_1, o_1\,!==_1\ e_2\rangle \Downarrow_m \langle h_f, o_f\rangle}{\wp, L, v_t \vdash \langle h, e_1\,!==\ e_2\rangle \Downarrow_m \langle h_f, o_f\rangle}$$

STRICT INEQUALITY - 1

$$\frac{\wp, L, v_t \vdash \langle h, e_2\rangle \Downarrow_m^{\gamma} \langle h_1, o_1\rangle \quad \wp, L, v_t \vdash \langle h_1, v_1\,!==_2\ o_1\rangle \Downarrow_m \langle h_f, o_f\rangle}{\wp, L, v_t \vdash \langle h, v_1\,!==_1\ e_2\rangle \Downarrow_m \langle h_f, o_f\rangle}$$

STRICT INEQUALITY - 2 (TRUE)

$$\frac{\mathcal{I}_{===}(v_1, v_2) = \mathsf{true}}{\vdash \langle h, v_1\,!==_2\ v_2\rangle \Downarrow_m \langle h, \mathsf{false}\rangle}$$

STRICT INEQUALITY - 2 (FALSE)

$$\frac{\mathcal{I}_{===}(v_1, v_2) = \mathsf{false}}{\vdash \langle h, v_1\,!==_2\ v_2\rangle \Downarrow_m \langle h, \mathsf{true}\rangle}$$

ASSIGNMENT - 1

$$\frac{\wp, L, v_t \vdash \langle h, e_1\rangle \Downarrow_m \langle h_1, o_1\rangle \quad \wp, L, v_t \vdash \langle h_1, o_1 =_1 e_2\rangle \Downarrow_m \langle h_f, o_f\rangle}{\wp, L, v_t \vdash \langle h, e_1 = e_2\rangle \Downarrow_m \langle h_f, o_f\rangle}$$

ASSIGNMENT - 2 AND 3

$$\frac{\wp, L, v_t \vdash \langle h, e_2\rangle \Downarrow_m^{\gamma} \langle h_1, o_1\rangle \quad \wp, L, v_t \vdash \langle h_1, w_1 =_2 o_1\rangle \Downarrow_m \langle h_f, o_f\rangle}{\wp, L, v_t \vdash \langle h, w_1 =_1 e_2\rangle \Downarrow_m \langle h_f, o_f\rangle}$$

Assignment - 4

$$\frac{p \in \{\mathsf{eval}, \mathsf{arguments}\} \quad h_f = h \uplus \mathsf{err}(l', l_{sep})}{\vdash \langle h, l.\!\!\vee p =_2 v\rangle \Downarrow_m \langle h_f, \mathsf{error}\, l'\rangle}$$

Assignment - 5

$$\frac{(w_1 = v_1 \vee w_1 = l.\!\!\circ p \vee (w_1 = l.\!\!\vee p \wedge p \notin \{\mathsf{eval}, \mathsf{arguments}\})) \quad \wp, L, v_t \vdash \langle h, \mathcal{I}_{pv}(w_1, v_2)\rangle \Downarrow_m \langle h_1, o_1\rangle \quad \wp, L, v_t \vdash \langle h_1, o_1 =_3 v_2\rangle \Downarrow_m \langle h_f, o_f\rangle}{\wp, L, v_t \vdash \langle h, w_1 =_2 v_2\rangle \Downarrow_m \langle h_f, o_f\rangle}$$

Assignment - 6

$$\vdash \langle h, \_ =_3 v\rangle \Downarrow_m \langle h, v\rangle$$

PBS Semantics for Statements: $\wp, L, v_t \vdash \langle h, s\rangle \Downarrow_m \langle h, o\rangle$**.**

Sequence

$$\frac{\wp, L, v_t \vdash \langle h, s_1\rangle \Downarrow_m \langle h_1, o_1\rangle \quad \wp, L, v_t \vdash \langle h_1, \mathsf{seq}_1(o_1, s_2)\rangle \Downarrow_m \langle h_f, o_f\rangle}{\wp, L, v_t \vdash \langle h, s_1; s_2\rangle \Downarrow_m \langle h_f, o_f\rangle}$$

Sequence - 1

$$\frac{o_1 \neq \mathsf{error}\, v \quad o_1 \neq \mathsf{ret}\, v \quad o_1 \neq \mathsf{break}\, w \quad \wp, L, v_t \vdash \langle h, s_2\rangle \Downarrow_m \langle h_1, o_2\rangle \quad \wp, L, v_t \vdash \langle h_1, \mathsf{seq}_2(o_1, o_2)\rangle \Downarrow_m \langle h_f, o_f\rangle}{\wp, L, v_t \vdash \langle h, \mathsf{seq}_1(o_1, s_2)\rangle \Downarrow_m \langle h_f, o_f\rangle}$$

Sequence - 1 (return)

$$\vdash \langle h, \mathsf{seq}_1(\mathsf{ret}\, v, \_)\rangle \Downarrow_m \langle h, \mathsf{ret}\, v\rangle$$

Sequence - 1 (break)

$$\vdash \langle h, \mathsf{seq}_1(\mathsf{break}\, w, \_)\rangle \Downarrow_m \langle h, \mathsf{break}\, w\rangle$$

Sequence - 2 (non-empty)

$$\frac{o \neq \mathsf{empty} \quad o \neq \mathsf{break\ empty}}{\vdash \langle h, \mathsf{seq}_2(\_, o)\rangle \Downarrow_m \langle h, o\rangle}$$

Sequence - 2 (empty)

$$\vdash \langle h, \mathsf{seq}_2(o, \mathsf{empty})\rangle \Downarrow_m \langle h, o\rangle$$

Sequence - 2 (break)

$$\vdash \langle h, \mathsf{seq}_2(w, \mathsf{break\ empty})\rangle \Downarrow_m \langle h, \mathsf{break}\, w\rangle$$

Var Decl

$$\vdash \langle h, \mathsf{var}\, x\rangle \Downarrow_m \langle h, \mathsf{empty}\rangle$$

Expr

$$\frac{\wp, L, v_t \vdash \langle h, e\rangle \Downarrow_m^{\gamma} \langle h_f, o_f\rangle}{\wp, L, v_t \vdash \langle h, e\rangle \Downarrow_m \langle h_f, o_f\rangle}$$

if

$$\frac{\wp, L, v_t \vdash \langle h, e\rangle \Downarrow_m^{\gamma} \langle h_1, o_1\rangle \quad \wp, L, v_t \vdash \langle h_1, \mathsf{if}_1(o_1)\,\{s_1\}\,\mathsf{else}\,\{s_2\}\rangle \Downarrow_m \langle h_f, o_f\rangle}{\wp, L, v_t \vdash \langle h, \mathsf{if}(e)\,\{s_1\}\,\mathsf{else}\,\{s_2\}\rangle \Downarrow_m \langle h_f, o_f\rangle}$$

if - 1 (true)

$$\frac{\mathcal{I}_{tb}(v) = \mathsf{true} \quad \wp, L, v_t \vdash \langle h, s_1\rangle \Downarrow_m \langle h_f, o_f\rangle}{\wp, L, v_t \vdash \langle h, \mathsf{if}_1(v)\,\{s_1\}\,\mathsf{else}\,\{s_2\}\rangle \Downarrow_m \langle h_f, o_f\rangle}$$

if - 1 (false)

$$\frac{\mathcal{I}_{tb}(v) = \mathsf{false} \quad \wp, L, v_t \vdash \langle h, s_2\rangle \Downarrow_m \langle h_f, o_f\rangle}{\wp, L, v_t \vdash \langle h, \mathsf{if}_1(v)\,\{s_1\}\,\mathsf{else}\,\{s_2\}\rangle \Downarrow_m \langle h_f, o_f\rangle}$$

While

$$\frac{\wp, L, v_t \vdash \langle h, \mathsf{while}_1(e)\{s, \mathsf{empty}\}\rangle \Downarrow_m \langle h_f, o_f\rangle}{\wp, L, v_t \vdash \langle h, \mathsf{while}(e)\,\{s\}\rangle \Downarrow_m \langle h_f, o_f\rangle}$$

While - 1

$$\frac{\wp, L, v_t \vdash \langle h, e\rangle \Downarrow_m^{\gamma} \langle h_1, o_1\rangle \quad \wp, L, v_t \vdash \langle h_1, \mathsf{while}_2(o_1, e)\{s, o\}\rangle \Downarrow_m \langle h_f, o_f\rangle}{\wp, L, v_t \vdash \langle h, \mathsf{while}_1(e)\{s, o\}\rangle \Downarrow_m \langle h_f, o_f\rangle}$$

While - 2 (true)

$$\frac{\mathcal{I}_{tb}(v) = \mathsf{true} \quad \wp, L, v_t \vdash \langle h, s\rangle \Downarrow_m^{\gamma} \langle h_1, o_1\rangle \quad \wp, L, v_t \vdash \langle h_1, \mathsf{while}_3(e)\{s, o, o_1\}\rangle \Downarrow_m \langle h_f, o_f\rangle}{\wp, L, v_t \vdash \langle h, \mathsf{while}_2(v, e)\{s, o\}\rangle \Downarrow_m \langle h_f, o_f\rangle}$$

While - 2 (false)

$$\frac{\mathcal{I}_{tb}(v) = \mathsf{false}}{\vdash \langle h, \mathsf{while}_2(v, \_)\{\_, o\}\rangle \Downarrow_m \langle h, o\rangle}$$

While - 3 (value)

$$\frac{\wp, L, v_t \vdash \langle h, \mathsf{while}_1(e)\{s, v\}\rangle \Downarrow_m \langle h_f, o_f\rangle}{\wp, L, v_t \vdash \langle h, \mathsf{while}_3(e)\{s, o, v\}\rangle \Downarrow_m \langle h_f, o_f\rangle}$$

While - 3 (return)

$$\vdash \langle h, \mathsf{while}_3(e)\{s, o, \mathsf{ret}\, v\}\rangle \Downarrow_m \langle h, \mathsf{ret}\, v\rangle$$

While - 3 (empty)

$$\frac{\wp, L, v_t \vdash \langle h, \mathsf{while}_1(e)\{s, o\}\rangle \Downarrow_m \langle h_f, o_f\rangle}{\wp, L, v_t \vdash \langle h, \mathsf{while}_3(e)\{s, o, \mathsf{empty}\}\rangle \Downarrow_m \langle h_f, o_f\rangle}$$

While - 3 (break)

$$\vdash \langle h, \mathsf{while}_3(e)\{s, o, \mathsf{break}\, v\}\rangle \Downarrow_m \langle h, v\rangle$$

Break

$$\vdash \langle h, \mathsf{break}\ \rangle \Downarrow_m \langle h, \mathsf{break\ empty} \rangle$$

Return

$$\frac{\wp, L, v_t \vdash \langle h, e\rangle \ \Downarrow^{\gamma}_m \langle h_1, o_1\rangle \qquad \wp, L, v_t \vdash \langle h_1, \mathsf{return}_1\ o_1\rangle \Downarrow_m \langle h_f, o_f\rangle}{\wp, L, v_t \vdash \langle h, \mathsf{return}\ e\rangle \Downarrow_m \langle h_f, o_f\rangle}$$

Return - 1

$$\vdash \langle h, \mathsf{return}_1\ v\rangle \Downarrow_m \langle h, \mathsf{ret}\ v\rangle$$

Throw

$$\frac{\wp, L, v_t \vdash \langle h, e\rangle \ \Downarrow^{\gamma}_m \langle h_1, o_1\rangle \qquad \wp, L, v_t \vdash \langle h_1, \mathsf{throw}_1\ o_1\rangle \Downarrow_m \langle h_f, o_f\rangle}{\wp, L, v_t \vdash \langle h, \mathsf{throw}\ e\rangle \Downarrow_m \langle h_f, o_f\rangle}$$

Throw - 1

$$\vdash \langle h, \mathsf{throw}_1\ v\rangle \Downarrow_m \langle h, \mathsf{error}\ v\rangle$$

Error Propagation

$$\vdash \langle h, s_{\lightning}\rangle \Downarrow_m \langle h, \mathsf{error}\ v\rangle$$

$$\begin{aligned} s_{\lightning} \triangleq\ & (\mathsf{error}\ v)[e]_1 \mid v_1[\mathsf{error}\ v]_2 \mid v_1[v_2](\mathsf{error}\ v)_3 \mid v_1[\mathsf{error}\ v]_4 \\ & \mid \mathsf{new}_1\ (\mathsf{error}\ v)(\overline{e}) \mid \mathsf{new}_2\ v(\mathsf{error}\ v) \mid \mathsf{iterate}_1\{\overline{e}, \overline{v} :: (\mathsf{error}\ v)\} \\ & \mid (\mathsf{error}\ v)(\overline{e})_1 \mid (w, \mathsf{error}\ v)(\overline{e})_2 \mid (w, v)(\mathsf{error}\ v)_3 \\ & \mid \mathsf{del}_1\ (\mathsf{error}\ v) \mid \mathsf{del}_2\ (\mathsf{error}\ v).{}_o p \mid \mathsf{typeof}_1\ \mathsf{error}\ v \\ & \mid (\mathsf{error}\ v)\ +_1\ e \mid v\ +_2\ (\mathsf{error}\ v) \mid (\mathsf{error}\ v)\ +_3\ e \mid v\ +_4\ (\mathsf{error}\ v) \\ & \mid (\mathsf{error}\ v)\ ===_1\ e \mid v\ ===_2\ (\mathsf{error}\ v) \mid (\mathsf{error}\ v)\,!==_1\ e \mid v\,!==_2\ (\mathsf{error}\ v) \\ & \mid (\mathsf{error}\ v)\ =_1 e \mid w =_2 (\mathsf{error}\ v) \mid (\mathsf{error}\ v) =_3 v \\ & \mid \mathsf{seq}_1(\mathsf{error}\ v, s) \mid \mathsf{seq}_2(o, \mathsf{error}\ v) \\ & \mid \mathsf{if}_1(\mathsf{error}\ v)\,\{s_1\}\,\mathsf{else}\,\{s_2\} \mid \mathsf{while}_2(\mathsf{error}\ v, e)\{s, o\} \mid \mathsf{while}_3(e)\{s, o, \mathsf{error}\ v\} \\ & \mid \mathsf{return}_1\ (\mathsf{error}\ v) \mid \mathsf{throw}_1\ (\mathsf{error}\ v) \end{aligned}$$

A.3. Property Descriptors

1. `makeDataDesc`(d) : create a (fully populated) data descriptor based on the appropriate fields of d.

$$\texttt{makeDataDesc}(d) = \begin{cases} [[\mathrm{V}]] : \text{if } d.[[V]] \text{ exists then } d.[[V]] \text{ else } \mathsf{undefined} \\ [[\mathrm{W}]] : \text{if } d.[[W]] \text{ exists then } d.[[W]] \text{ else } \mathsf{false} \\ [[\mathrm{C}]] : \text{if } d.[[C]] \text{ exists then } d.[[C]] \text{ else } \mathsf{false} \\ [[\mathrm{E}]] : \text{if } d.[[E]] \text{ exists then } d.[[E]] \text{ else } \mathsf{false} \end{cases}$$

2. `makeAccessorDesc`(d) : create a (fully populated) accessor descriptor based on the appropriate fields of d.

$$\texttt{makeAccessorDesc}(d) = \begin{cases} [[\mathrm{G}]] : \text{if } d.[[G]] \text{ exists then } d.[[G]] \text{ else } \mathsf{undefined} \\ [[\mathrm{S}]] : \text{if } d.[[S]] \text{ exists then } d.[[S]] \text{ else } \mathsf{undefined} \\ [[\mathrm{C}]] : \text{if } d.[[C]] \text{ exists then } d.[[C]] \text{ else } \mathsf{false} \\ [[\mathrm{E}]] : \text{if } d.[[E]] \text{ exists then } d.[[E]] \text{ else } \mathsf{false} \end{cases}$$

3. `containsDesc`$(d_c, d) = \begin{cases} \mathsf{true} : \text{if all fields that exist in } d \text{ also exist in } d_c, \\ \qquad\quad \text{and for each such field } X, \texttt{SameValue}(d_c.[[X]], d.[[X]]) \\ \mathsf{false} : \text{otherwise} \end{cases}$

4. $\texttt{changeEnumOnNotConf}(d_c, d) = \begin{cases} \mathsf{true} : d_c = \mathsf{false} \wedge \\ \qquad (d.[[C]] = \mathsf{true} \vee (d.[[E]] \text{ exists} \wedge d_c.[[E]] = \neg d.[[E]])) \\ \mathsf{false} : \text{otherwise} \end{cases}$

5. $\texttt{changeDataOnNotConf}(d_c, d) = \begin{cases} \mathsf{true} : d_c.[[C]] = \mathsf{false} \wedge d_c.[[W]] = \mathsf{false} \wedge \\ \qquad (d.[[W]] = \mathsf{true} \vee (d.[[V]] \text{ exists} \wedge \\ \qquad \neg\texttt{SameValue}(d_c.[[V]], d.[[V]]))) \\ \mathsf{false} : \text{otherwise} \end{cases}$

6. $\texttt{changeAccOnNotConf}(d_c, d) = \begin{cases} \mathsf{true} : d_c.[[C]] = \mathsf{false} \wedge \\ \qquad ((d.[[G]] \text{ exists} \wedge \neg\texttt{SameValue}(d_c.[[G]], d.[[G]])) \vee \\ \qquad (d.[[S]] \text{ exists} \wedge \neg\texttt{SameValue}(d_c.[[S]], d.[[S]]))) \\ \mathsf{false} : \text{otherwise} \end{cases}$

7. $\texttt{updateDesc}(d_c, d) = \begin{cases} [[X]] : \text{if } d.[[X]] \text{ exists then } d.[[X]] \\ \qquad \text{else if } d_c.[[X]] \text{ exists then } d_c.[[X]] \text{ else not defined} \\ \qquad\quad \text{for } X \in \{V, W, G, S, C, E\} \end{cases}$

8. IsDataDescriptor(d) : true, iff d is a data descriptor.

$$\text{IDD-UNDEFINED}\ \frac{}{\neg\mathcal{P}_{dd}(\mathsf{undefined})} \qquad \text{IDD-TRUE}\ \frac{(d, [[D]]) \in h \vee (d, [[W]]) \in h}{\mathcal{P}_{dd}(d)} \qquad \text{IDD-FALSE}\ \frac{(d, [[V]]) \notin h \wedge (d, [[W]]) \notin h}{\neg\mathcal{P}_{dd}(d)}$$

9. IsAccessorDescriptor(d) : true, iff d is an accessor descriptor.

$$\text{IAD-UNDEFINED}\ \frac{}{\neg\mathcal{P}_{ad}(\mathsf{undefined})} \qquad \text{IAD-TRUE}\ \frac{(d, [[G]]) \in h \vee (d, [[S]]) \in h}{\mathcal{P}_{ad}(d)} \qquad \text{IAD-FALSE}\ \frac{(d, [[G]]) \notin h \wedge (d, [[S]]) \notin h}{\neg\mathcal{P}_{ad}(d)}$$

10. IsGenericDescriptor(d) : true, iff d is neither a data nor an accessor descriptor.

$$\text{IPD-UNDEFINED}\ \frac{}{\neg\mathcal{P}_{gd}(\mathsf{undefined})} \qquad \text{IPD-TRUE}\ \frac{\neg\mathcal{P}_{dd}(d) \wedge \neg\mathcal{P}_{ad}(d)}{\mathcal{P}_{gd}(d)} \qquad \text{IPD-FALSE}\ \frac{\mathcal{P}_{dd}(d) \vee \mathcal{P}_{ad}(d)}{\neg\mathcal{P}_{gd}(d)}$$

A.4. Internal Properties

1. General `GetOwnProperty`(P) — $\mathcal{I}_{gop}(x)$: : returns the property descriptor of the named *own* property of the this object, or undefined if absent. Does not traverse the prototype chain. String objects have different behaviour than all other objects.

$$\text{GOP-DEFAULT}\ \frac{h(l_t, @class) \neq \text{“}\texttt{String}\text{”} \quad \wp, L, l_t \vdash \langle h, \mathcal{I}^o_{gop}(x)\rangle \Downarrow_m \langle h_f, o_f\rangle}{\wp, L, l_t \vdash \langle h, \mathcal{I}_{gop}(x)\rangle \Downarrow_m \langle h_f, o_f\rangle} \qquad \text{GOP-STRING}\ \frac{h(l_t, @class) = \text{“}\texttt{String}\text{”} \quad \wp, L, l_t \vdash \langle h, \mathcal{I}^s_{gop}(x)\rangle \Downarrow_m \langle h_f, o_f\rangle}{\wp, L, l_t \vdash \langle h, \mathcal{I}_{gop}(x)\rangle \Downarrow_m \langle h_f, o_f\rangle}$$

2. `GetOwnProperty(P)` — $\mathcal{I}^{o}_{gop}(p)$: returns the property descriptor of the named *own* property of the this object, or undefined if absent. Does not traverse the prototype chain.

OGOP-UNDEF
$$\frac{(l_t, p) \notin \mathsf{dom}(h)}{\wp, \_, l_t \vdash \langle h, \mathcal{I}^{o}_{gop}(p)\rangle \Downarrow_m \langle h, \mathsf{undefined}\rangle}$$

OGOP-DEF
$$\frac{h(l_t, p) = d}{\wp, \_, l_t \vdash \langle h, \mathcal{I}^{o}_{gop}(p)\rangle \Downarrow_m \langle h, \mathsf{desc}\ d\rangle}$$

3. `GetOwnProperty(P)` — $\mathcal{I}^{s}_{gop}(x)$: provides access to named properties corresponding to the individual characters of String objects.

SGOP-GENERAL
$$\frac{\wp, L, l_t \vdash \langle h, \mathcal{I}_{gop}(p)\rangle \Downarrow_m \langle h_1, o_1\rangle \quad \wp, L, l_t \vdash \langle h_1, \mathcal{I}^{s}_{gop}(p, o_1)_1\rangle \Downarrow_m \langle h_f, o_f\rangle}{\wp, L, l_t \vdash \langle h, \mathcal{I}^{s}_{gop}(p)\rangle \Downarrow_m \langle h_f, o_f\rangle}$$

SGOP-EXISTS
$$\vdash \langle h, \mathcal{I}^{s}_{gop}(\_, \mathsf{desc}\ d)_1\rangle \Downarrow_m \langle h, \mathsf{desc}\ d\rangle$$

SGOP-NOTAPOSINT
$$\frac{\mathcal{I}^{prim}_{ts}(\mathtt{abs}(\mathcal{I}^{prim}_{ti}(x))) \neq p}{\vdash \langle h, \mathcal{I}^{s}_{gop}(p, \mathsf{undefined})_1\rangle \Downarrow_m \langle h, \mathsf{undefined}\rangle}$$

SGOP-POSINT
$$\frac{\mathcal{I}^{prim}_{ts}(\mathtt{abs}(\mathcal{I}^{prim}_{ti}(p))) = p \quad h(l_t, @primval) = s \quad \mathtt{length}(s) \leq \mathcal{I}^{prim}_{ti}(p)}{\wp, \_, l_t \vdash \langle h, \mathcal{I}^{s}_{gop}(p, \mathsf{undefined})_1\rangle \Downarrow_m \langle h, \mathsf{undefined}\rangle}$$

SGOP-INDEX
$$\frac{\mathcal{I}^{prim}_{ts}(\mathtt{abs}(\mathcal{I}^{prim}_{ti}(p))) = p \quad h(l_t, @primval) = s \quad \mathtt{length}(s) > \mathcal{I}^{prim}_{ti}(p) \quad \mathtt{charAt}(s, \mathcal{I}^{prim}_{ti}(p)) = v \quad d = \{[[V]] : v, [[W]] : \mathsf{false}, [[C]] : \mathsf{false}, [[E]] : \mathsf{true}\}}{\wp, \_, l_t \vdash \langle h, \mathcal{I}^{s}_{gop}(p, \mathsf{undefined})_1\rangle \Downarrow_m \langle h, \mathsf{desc}\ d\rangle}$$

4. `GetProperty(P)` — $\mathcal{I}_{gp}(p)$: returns the property descriptor of the named property of the this object, or undefined if absent. Traverses the prototype chain.

GP-GETOWN
$$\frac{\wp, L, l_t \vdash \langle h, \mathcal{I}_{gop}(p)\rangle \Downarrow_m \langle h_1, o\rangle \quad \wp, L, l_t \vdash \langle h_1, \mathcal{I}_{gp}(p, o)_1\rangle \Downarrow_m \langle h_f, o_f\rangle}{\wp, L, l_t \vdash \langle h, \mathcal{I}_{gp}(p)\rangle \Downarrow_m \langle h_f, o_f\rangle}$$

GP-OWNDEF
$$\vdash \langle h, \mathcal{I}_{gp}(\_, \mathsf{desc}\ d)_1\rangle \Downarrow_m \langle h, \mathsf{desc}\ d\rangle$$

GP-OWNUNDEF-PROTONULL
$$\frac{h(l_t, @proto) = \mathsf{null}}{\wp, \_, l_t \vdash \langle h, \mathcal{I}_{gp}(\_, \mathsf{undefined})_1\rangle \Downarrow_m \langle h, \mathsf{undefined}\rangle}$$

GP-OWNUNDEF-PROTONOTNULL
$$\frac{h(l_t, @proto) = l'_t \quad \wp, L, l'_t \vdash \langle h, \mathcal{I}_{gp}(p)\rangle \Downarrow_m \langle h_f, o_f\rangle}{\wp, L, l_t \vdash \langle h, \mathcal{I}_{gp}(p, \mathsf{undefined})_1\rangle \Downarrow_m \langle h_f, o_f\rangle}$$

5. `Get(P)` — $\mathcal{I}_{g}(p)$: returns the value of the named property of the this object.

G-getProp

$$\frac{\wp, L, l_t \vdash \langle h, \mathcal{I}_{gp}(p)\rangle \Downarrow_m \langle h_1, o\rangle \quad \wp, L, l_t \vdash \langle h_1, \mathcal{I}_g(o)_1\rangle \Downarrow_m \langle h_f, o_f\rangle}{\wp, L, l_t \vdash \langle h, \mathcal{I}_g(p)\rangle \Downarrow_m \langle h_f, o_f\rangle}$$

G-propUndef

$$\vdash \langle h, \mathcal{I}_g(\mathsf{undefined})_1\rangle \Downarrow_m \langle h, \mathsf{undefined}\rangle$$

G-propDefData

$$\frac{\mathcal{P}_{dd}(d) \quad v = d.[[V]]}{\vdash \langle h, \mathcal{I}_g(\mathsf{desc}\ d)_1\rangle \Downarrow_m \langle h, v\rangle}$$

G-propDefAccGetUndef

$$\frac{\mathcal{P}_{ad}(d) \quad d.[[G]] = \mathsf{undefined}}{\vdash \langle h, \mathcal{I}_g(\mathsf{desc}\ d)_1\rangle \Downarrow_m \langle h, \mathsf{undefined}\rangle}$$

G-propDefAccGetDef

$$\frac{\mathcal{P}_{ad}(d) \quad a_g = d.[[G]] \neq \mathsf{undefined} \quad \wp, L, a_g \vdash \langle h, \mathcal{I}_{call}(a_g, [\,])\rangle \Downarrow_m \langle h_f, o_f\rangle}{\wp, L, l_t \vdash \langle h, \mathcal{I}_g(\mathsf{desc}\ d)_1\rangle \Downarrow_m \langle h_f, o_f\rangle}$$

6. `CanPut(P)` — $\mathcal{I}_{cp}(p)$ - returns true iff a `Put` operation can be performed on a given property of the this object.

CP-getOwnProp

$$\frac{\wp, L, l_t \vdash \langle h, \mathcal{I}_{gop}(p)\rangle \Downarrow_m \langle h, o\rangle \quad \wp, L, l_t \vdash \langle h, \mathcal{I}_{cp}(p, o)_1\rangle \Downarrow_m \langle h_f, o_f\rangle}{\wp, L, l_t \vdash \langle h, \mathcal{I}_{cp}(p)\rangle \Downarrow_m \langle h_f, o_f\rangle}$$

CP-ownPropData

$$\frac{\mathcal{P}_{dd}(d) \quad b_w = d.[[W]]}{\vdash \langle h, \mathcal{I}_{cp}(\_, \mathsf{desc}\ d)_1\rangle \Downarrow_m \langle h, b_w\rangle}$$

CP-getProp

$$\frac{\neg\mathcal{P}_{dd}(d) \quad \wp, L, l_t \vdash \langle h, \mathcal{I}_{gp}(p)\rangle \Downarrow_m \langle h_1, o\rangle \quad \wp, L, l_t \vdash \langle h_1, \mathcal{I}_{cp}(o)_2\rangle \Downarrow_m \langle h_f, o_f\rangle}{\wp, L, l_t \vdash \langle h, \mathcal{I}_{cp}(p, \mathsf{desc}\ d)_1\rangle \Downarrow_m \langle h_f, o_f\rangle}$$

CP-propNotFound

$$\frac{v = h(l_t, @extensible)}{\wp, \_, l_t \vdash \langle h, \mathcal{I}_{cp}(\mathsf{undefined})_2\rangle \Downarrow_m \langle h, v\rangle}$$

CP-propAccSetUndef

$$\frac{\mathcal{P}_{ad}(d) \quad d.[[S]] = \mathsf{undefined}}{\vdash \langle h, \mathcal{I}_{cp}(\mathsf{desc}\ d)_2\rangle \Downarrow_m \langle h, \mathsf{false}\rangle}$$

CP-propAccSetDef

$$\frac{\mathcal{P}_{ad}(d) \quad d.[[S]] \neq \mathsf{undefined}}{\vdash \langle h, \mathcal{I}_{cp}(\mathsf{desc}\ d)_2\rangle \Downarrow_m \langle h, \mathsf{true}\rangle}$$

CP-propDataExtens

$$\frac{\mathcal{P}_{dd}(d) \quad b_w = d.[[W]] \quad h(l_t, @extensible) = \mathsf{true}}{\wp, \_, l_t \vdash \langle h, \mathcal{I}_{cp}(\mathsf{desc}\ d)_2\rangle \Downarrow_m \langle h, b_w\rangle}$$

CP-propDataNotExtens

$$\frac{\mathcal{P}_{dd}(d) \quad h(l_t, @extensible) = \mathsf{false}}{\wp, \_, l_t \vdash \langle h, \mathcal{I}_{cp}(\mathsf{desc}\ d)_2\rangle \Downarrow_m \langle h, \mathsf{false}\rangle}$$

7. `Put(P, V, Throw)` — $\mathcal{I}_p(p, v, b_t)$: sets the specified named property of the this object to the specified value.

P-canPut

$$\frac{\wp, L, l_t \vdash \langle h, \mathcal{I}_{cp}(p)\rangle \Downarrow_m \langle h_1, o\rangle \quad \wp, L, l_t \vdash \langle h_1, \mathcal{I}_p(p, v, b_t, o)_1\rangle \Downarrow_m \langle h_f, o_f\rangle}{\wp, L, l_t \vdash \langle h, \mathcal{I}_p(p, v, b_t)\rangle \Downarrow_m \langle h_f, o_f\rangle}$$

P-cannotPutThrow

$$\frac{h_f = h \uplus \mathsf{err}(l, l_{tep})}{\vdash \langle h, \mathcal{I}_p(\_, \_, \mathsf{true}, \mathsf{false})_1\rangle \Downarrow_m \langle h_f, \mathsf{error}\, l\rangle}$$

P-cannotPutNoThrow

$$\vdash \langle h, \mathcal{I}_p(\_, \_, \mathsf{false}, \mathsf{false})_1\rangle \Downarrow_m \langle h, \mathsf{empty}\rangle$$

P-canPutGetOwnProp

$$\frac{\wp, L, l_t \vdash \langle h, \mathcal{I}_{gop}(p)\rangle \Downarrow_m \langle h_1, o\rangle \quad \wp, L, l_t \vdash \langle h_1, \mathcal{I}_p(p, v, b_t, o)_2\rangle \Downarrow_m \langle h_f, o_f\rangle}{\wp, L, l_t \vdash \langle h, \mathcal{I}_p(p, v, b_t, \mathsf{true})_1\rangle \Downarrow_m \langle h_f, o_f\rangle}$$

P-canPutOwnPropData

$$\frac{\mathcal{P}_{dd}(d) \quad d' = \{[[V]] : v\} \quad \wp, L, l_t \vdash \langle h, \mathcal{I}^o_{dop}(p, d', b_t)\rangle \Downarrow_m \langle h_1, o\rangle \quad \wp, L, l_t \vdash \langle h_1, \mathcal{I}_p(o)_4\rangle \Downarrow_m \langle h_f, o_f\rangle}{\wp, L, l_t \vdash \langle h, \mathcal{I}_p(p, v, b_t, \mathsf{desc}\ d)_2\rangle \Downarrow_m \langle h_f, o_f\rangle}$$

P-getProp

$$\frac{\neg\mathcal{P}_{dd}(d) \quad \wp, L, l_t \vdash \langle h, \mathcal{I}_{gp}(p)\rangle \Downarrow_m \langle h_1, o\rangle \quad \wp, L, l_t \vdash \langle h_1, \mathcal{I}_p(p, v, b_t, o)_3\rangle \Downarrow_m \langle h_f, o_f\rangle}{\wp, L, l_t \vdash \langle h, \mathcal{I}_p(p, v, b_t, \mathsf{desc}\ d)_2\rangle \Downarrow_m \langle h_f, o_f\rangle}$$

P-propAccSet

$$\frac{\mathcal{P}_{ad}(d) \quad a_s = d.[[S]] \quad \wp, L, a_s \vdash \langle h, \mathcal{I}_{call}(a_s, [v])\rangle \Downarrow_m \langle h_1, o\rangle \quad \wp, L, l_t \vdash \langle h_1, \mathcal{I}_p(o)_4\rangle \Downarrow_m \langle h_f, o_f\rangle}{\wp, L, l_t \vdash \langle h, \mathcal{I}_{cp}(\_, v, \_, \mathsf{desc}\ d)_3\rangle \Downarrow_m \langle h_f, o_f\rangle}$$

P-propDOP

$$\frac{\neg\mathcal{P}_{ad}(d) \quad d' = \{[[V]] : v, [[W]] : \mathsf{true}, [[C]] : \mathsf{true}\} \quad \wp, L, l_t \vdash \langle h, \mathcal{I}^o_{dop}(p, d', b_t)\rangle \Downarrow_m \langle h_1, o\rangle \quad \wp, L, l_t \vdash \langle h_1, \mathcal{I}_p(o)_4\rangle \Downarrow_m \langle h_f, o_f\rangle}{\wp, L, l_t \vdash \langle h, \mathcal{I}_{cp}(p, v, b_t, \mathsf{desc}\ d)_3\rangle \Downarrow_m \langle h_f, o_f\rangle}$$

P-PutReturn

$$\vdash \langle h, \mathcal{I}_p(v)_4\rangle \Downarrow_m \langle h, \mathsf{empty}\rangle$$

8. `HasProperty(P)` — $\mathcal{I}_{hp}(p)$: returns true iff the this object has the specified property in its prototype chain.

HP-getProp

$$\frac{\wp, L, l_t \vdash \langle h, \mathcal{I}_{gp}(p)\rangle \Downarrow_m \langle h_1, o_1\rangle \quad \wp, L, l_t \vdash \langle h_1, \mathcal{I}_{hp}(o_1)_1\rangle \Downarrow_m \langle h_f, o_f\rangle}{\wp, L, l_t \vdash \langle h, \mathcal{I}_{hp}(p)\rangle \Downarrow_m \langle h_f, o_f\rangle}$$

HP-propUndef

$$\vdash \langle h, \mathcal{I}_{hp}(\mathsf{undefined})_1\rangle \Downarrow_m \langle h, \mathsf{false}\rangle$$

HP-propDef

$$\vdash \langle h, \mathcal{I}_{hp}(\mathsf{desc}\ d)_1\rangle \Downarrow_m \langle h, \mathsf{true}\rangle$$

9. `Delete(P, Throw)` — $\mathcal{I}_d(p, b_t)$: removes the specified property from the this object

D-getOwnProp

$$\frac{\wp, L, l_t \vdash \langle h, \mathcal{I}_{gop}(p)\rangle \Downarrow_m \langle h_1, o\rangle \quad \wp, L, l_t \vdash \langle h_1, \mathcal{I}_d(p, b_t, o)_1\rangle \Downarrow_m \langle h_f, o_f\rangle}{\wp, L, l_t \vdash \langle h, \mathcal{I}_d(p, b_t)\rangle \Downarrow_m \langle h_f, o_f\rangle}$$

D-ownPropUndef

$$\vdash \langle h, \mathcal{I}_d(\_, \_, \mathsf{undefined})_1\rangle \Downarrow_m \langle h, \mathsf{true}\rangle$$

D-ownPropDef

$$\frac{b_c = d.[[C]] \quad \wp, L, l_t \vdash \langle h, \mathcal{I}_d(p, b_t, b_c)_2\rangle \Downarrow_m \langle h_f, o_f\rangle}{\wp, L, l_t \vdash \langle h, \mathcal{I}_d(p, b_t, \mathsf{desc}\ d)_1\rangle \Downarrow_m \langle h_f, o_f\rangle}$$

D-ownPropDefNotConfThrow
$$\frac{h_f = h \uplus \mathsf{err}(l, l_{tep})}{\vdash \langle h, \mathcal{I}_{hp}(\_, \mathsf{true}, \mathsf{false})_2\rangle \Downarrow_m \langle h_f, \mathsf{error}\, l\rangle}$$

D-ownPropDefNotConfNoThrow
$$\vdash \langle h, \mathcal{I}_{hp}(\_, \mathsf{false}, \mathsf{false})_2\rangle \Downarrow_m \langle h, \mathsf{false}\rangle$$

D-ownPropDefConf
$$\frac{h_f = h \backslash l_t.x}{\wp, \_, l_t \vdash \langle h, \mathcal{I}_{hp}(x, \_, \mathsf{true})_2\rangle \Downarrow_m \langle h_f, \mathsf{true}\rangle}$$

10. `DefaultValue(hint)` — $\mathcal{I}_{dv}(x)$: returns the default value for the this object.

DV-defNum
$$\frac{\wp, L, l_t \vdash \langle h, \mathcal{I}_{dv}(\text{“}\texttt{valueOf}\text{”}, \text{“}\texttt{toString}\text{”})_1\rangle \Downarrow_m \langle h_f, o_f\rangle}{\wp, L, l_t \vdash \langle h, \mathcal{I}_{dv}()\rangle \Downarrow_m \langle h_f, o_f\rangle}$$

DV-hintNum
$$\frac{\wp, L, l_t \vdash \langle h, \mathcal{I}_{dv}(\text{“}\texttt{valueOf}\text{”}, \text{“}\texttt{toString}\text{”})_1\rangle \Downarrow_m \langle h_f, o_f\rangle}{\wp, L, l_t \vdash \langle h, \mathcal{I}_{dv}(\text{“}\texttt{Number}\text{”})\rangle \Downarrow_m \langle h_f, o_f\rangle}$$

DV-hintStr
$$\frac{\wp, L, l_t \vdash \langle h, \mathcal{I}_{dv}(\text{“}\texttt{toString}\text{”}, \text{“}\texttt{valueOf}\text{”})_1\rangle \Downarrow_m \langle h_f, o_f\rangle}{\wp, L, l_t \vdash \langle h, \mathcal{I}_{dv}(\text{“}\texttt{String}\text{”})\rangle \Downarrow_m \langle h_f, o_f\rangle}$$

DV-firstPass
$$\frac{\wp, L, l_t \vdash \langle h, \mathcal{I}_{dv}(x_1, \mathcal{I}_{dv}(x_2)_2)_4\rangle \Downarrow_m \langle h_f, o_f\rangle}{\wp, L, l_t \vdash \langle h, \mathcal{I}_{dv}(x_1, x_2)_1\rangle \Downarrow_m \langle h_f, o_f\rangle}$$

DV-secondPass
$$\frac{\wp, L, l_t \vdash \langle h, \mathcal{I}_{dv}(x, \mathcal{I}_{dv}()_3)_4\rangle \Downarrow_m \langle h_f, o_f\rangle}{\wp, L, l_t \vdash \langle h, \mathcal{I}_{dv}(x)_2\rangle \Downarrow_m \langle h_f, o_f\rangle}$$

DV-error
$$\frac{h_f = h \uplus \mathsf{err}(l, l_{tep})}{\vdash \langle h, \mathcal{I}_{dv}()_3\rangle \Downarrow_m \langle h_f, \mathsf{error}\, l\rangle}$$

DV-getMethod
$$\frac{\wp, L, l_t \vdash \langle h, \mathcal{I}_g(x)\rangle \Downarrow_m \langle h_1, o\rangle \quad \wp, L, l_t \vdash \langle h_1, \mathcal{I}_{dv}(\mathcal{I}, o)_5\rangle \Downarrow_m \langle h_f, o_f\rangle}{\wp, L, l_t \vdash \langle h, \mathcal{I}_{dv}(x, \mathcal{I})_4\rangle \Downarrow_m \langle h_f, o_f\rangle}$$

DV-NotCallable
$$\frac{\neg\mathcal{P}_c(h, v) \quad \wp, L, l_t \vdash \langle h, \mathcal{I}\rangle \Downarrow_m \langle h_f, o_f\rangle}{\wp, L, l_t \vdash \langle h, \mathcal{I}_{dv}(\mathcal{I}, v)_5\rangle \Downarrow_m \langle h_f, o_f\rangle}$$

DV-Callable
$$\frac{\mathcal{P}_c(h, v) \quad \wp, L, l_t \vdash \langle h, v()\rangle \Downarrow_m \langle h_1, o\rangle \quad \wp, L, l_t \vdash \langle h_1, \mathcal{I}_{dv}(\mathcal{I}, o)_6\rangle \Downarrow_m \langle h_f, o_f\rangle}{\wp, L, l_t \vdash \langle h, \mathcal{I}_{dv}(\mathcal{I}, v)_5\rangle \Downarrow_m \langle h_f, o_f\rangle}$$

DV-CallablePrimitive
$$\frac{\mathcal{P}_{pv}(v)}{\vdash \langle h, \mathcal{I}_{dv}(\_, v)_6\rangle \Downarrow_m \langle h, v\rangle}$$

DV-CallableNotPrimitive
$$\frac{\neg\mathcal{P}_{pv}(v) \quad \wp, L, l_t \vdash \langle h, \mathcal{I}\rangle \Downarrow_m \langle h_f, o_f\rangle}{\wp, L, l_t \vdash \langle h, \mathcal{I}_{dv}(\mathcal{I}, v)_6\rangle \Downarrow_m \langle h_f, o_f\rangle}$$

11. General `DefineOwnProperty(P, Desc, Throw)` — $\mathcal{I}_{dop}(p, d, b_t)$: creates or modifies the specified named own property of the this object using the specified property descriptor. Array objects have different behaviour than all other objects. We do not give operational semantics for the Array objects.

DOP-DEFAULT

$$\frac{h(l_t, @class) \neq \text{“}\texttt{Array}\text{”} \quad \wp, L, l_t \vdash \langle h, \mathcal{I}^o_{dop}(p, d, b_t)\rangle \Downarrow_m \langle h_f, o_f\rangle}{\wp, L, l_t \vdash \langle h, \mathcal{I}_{dop}(p, d, b_t)\rangle \Downarrow_m \langle h_f, o_f\rangle}$$

DOP-ARRAY

$$\frac{h(l_t, @class) = \text{“}\texttt{Array}\text{”} \quad \wp, L, l_t \vdash \langle h, \mathcal{I}^a_{dop}(p, d, b_t)\rangle \Downarrow_m \langle h_f, o_f\rangle}{\wp, L, l_t \vdash \langle h, \mathcal{I}_{dop}(p, d, b_t)\rangle \Downarrow_m \langle h_f, o_f\rangle}$$

12. Default `DefineOwnProperty(P, Desc, Throw)` — $\mathcal{I}^o_{dop}(p, d, b_t)$: creates or modifies the specified named own property of the this object using the specified property descriptor.

DOP-GETOWNPROPANDEXTENS

$$\frac{b = h(l_t, @extensible) \quad \wp, L, l_t \vdash \langle h, \mathcal{I}_{gop}(p)\rangle \Downarrow_m \langle h_1, o\rangle \quad \wp, L, l_t \vdash \langle h_1, \mathcal{I}^o_{dop}(p, d, b_t, o, b)_1\rangle \Downarrow_m \langle h_f, o_f\rangle}{\wp, L, l_t \vdash \langle h, \mathcal{I}^o_{dop}(p, d, b_t)\rangle \Downarrow_m \langle h_f, o_f\rangle}$$

DOP-UNDEFINEDNOTEXTENS

$$\frac{\wp, L, l_t \vdash \langle h, \mathcal{I}^o_{dop}(b_t)_r\rangle \Downarrow_m \langle h_f, o_f\rangle}{\wp, L, l_t \vdash \langle h, \mathcal{I}^o_{dop}(p, d, b_t, \mathsf{undefined}, \mathsf{false})_1\rangle \Downarrow_m \langle h_f, o_f\rangle}$$

DOP-UNDEFINEDEXTENSGENDATA

$$\frac{\mathcal{P}_{gd}(d) \vee \mathcal{P}_{dd}(d) \quad d' = \texttt{makeDataDesc}(d) \quad h_f = h[(l_t, p) \mapsto d']}{\wp, \_, l_t \vdash \langle h, \mathcal{I}^o_{dop}(p, d, \_, \mathsf{undefined}, \mathsf{true})_1\rangle \Downarrow_m \langle h_f, \mathsf{true}\rangle}$$

DOP-UNDEFINEDEXTENSACC

$$\frac{\mathcal{P}_{ad}(d) \quad d' = \texttt{makeAccessorDesc}(d) \quad h_f = h[(l_t, p) \mapsto d']}{\wp, \_, l_t \vdash \langle h, \mathcal{I}^o_{dop}(p, d, \_, \mathsf{undefined}, \mathsf{true})_1\rangle \Downarrow_m \langle h', \mathsf{true}\rangle}$$

DOP-NOUPDATES

$$\frac{\texttt{containsDesc}(d_c, d) = \mathsf{true}}{\vdash \langle h, \mathcal{I}^o_{dop}(p, d, \_, \mathsf{desc}\ d_c, \_)_1\rangle \Downarrow_m \langle h, \mathsf{true}\rangle}$$

DOP-UPDATES

$$\frac{\texttt{containsDesc}(d_c, d) = \mathsf{false} \quad \wp, L, l_t \vdash \langle h, \mathcal{I}^o_{dop}(p, d, b_t, d_c)_2\rangle \Downarrow_m \langle h_f, o_f\rangle}{\wp, L, l_t \vdash \langle h, \mathcal{I}^o_{dop}(p, d, b_t, \mathsf{desc}\ d_c, \_)_1\rangle \Downarrow_m \langle h_f, o_f\rangle}$$

DOP-CHANGEENUMNOTCONFIG

$$\frac{\texttt{changeEnumOnNotConf}(d_c, d) = \mathsf{true} \quad \wp, L, l_t \vdash \langle h, \mathcal{I}^o_{dop}(b_t)_r\rangle \Downarrow_m \langle h_f, o_f\rangle}{\wp, L, l_t \vdash \langle h, \mathcal{I}^o_{dop}(\_, d, b_t, d_c)_2\rangle \Downarrow_m \langle h_f, o_f\rangle}$$

DOP-UPDATES

$$\frac{\texttt{changeEnumOnNotConf}(d_c, d) = \mathsf{false} \quad \wp, L, l_t \vdash \langle h, \mathcal{I}^o_{dop}(p, d, b_t, d_c)_3\rangle \Downarrow_m \langle h_f, o_f\rangle}{\wp, L, l_t \vdash \langle h, \mathcal{I}^o_{dop}(p, d, b_t, d_c)_2\rangle \Downarrow_m \langle h_f, o_f\rangle}$$

DOP-ISGENERIC

$$\frac{\mathcal{P}_{gd}(d) \quad \wp, L, l_t \vdash \langle h, \mathcal{I}^o_{dop}(p, d, d_c)_w\rangle \Downarrow_m \langle h_f, o_f\rangle}{\wp, L, l_t \vdash \langle h, \mathcal{I}^o_{dop}(p, d, \_, d_c)_3\rangle \Downarrow_m \langle h_f, o_f\rangle}$$

DOP-NOTBOTHDATA

$$\frac{\neg\mathcal{P}_{gd}(d) \quad \mathcal{P}_{dd}(d) \neq \mathcal{P}_{dd}(d_c) \quad \wp, L, l_t \vdash \langle h, \mathcal{I}^o_{dop}(p, d, b_t, d_c)_4\rangle \Downarrow_m \langle h_f, o_f\rangle}{\wp, L, l_t \vdash \langle h, \mathcal{I}^o_{dop}(p, d, b_t, d_c)_3\rangle \Downarrow_m \langle h_f, o_f\rangle}$$

DOP-BOTHDATA

$$\frac{\neg\mathcal{P}_{gd}(d) \quad \mathcal{P}_{dd}(d) \quad \mathcal{P}_{dd}(d_c) \qquad \wp, L, l_t \vdash \langle h, \mathcal{I}^o_{dop}(p, d, b_t, d_c)_5\rangle \Downarrow_m \langle h_f, o_f\rangle}{\wp, L, l_t \vdash \langle h, \mathcal{I}^o_{dop}(p, d, b_t, d_c)_3\rangle \Downarrow_m \langle h_f, o_f\rangle}$$

DOP-BOTHACCESSOR

$$\frac{\neg\mathcal{P}_{gd}(d) \quad \mathcal{P}_{ad}(d) \quad \mathcal{P}_{ad}(d_c) \qquad \wp, L, l_t \vdash \langle h, \mathcal{I}^o_{dop}(p, d, b_t, d_c)_6\rangle \Downarrow_m \langle h_f, o_f\rangle}{\wp, L, l_t \vdash \langle h, \mathcal{I}^o_{dop}(p, d, b_t, d_c)_3\rangle \Downarrow_m \langle h_f, o_f\rangle}$$

DOP-DIFFERENTNOTCONF

$$\frac{d_c.[[C]] = \mathsf{false} \qquad \wp, L, l_t \vdash \langle h, \mathcal{I}^o_{dop}(b_t)_r\rangle \Downarrow_m \langle h_f, o_f\rangle}{\wp, L, l_t \vdash \langle h, \mathcal{I}^o_{dop}(\_, \_, b_t, d_c)_4\rangle \Downarrow_m \langle h_f, o_f\rangle}$$

DOP-DATANOTCONF

$$\frac{\texttt{changeDataOnNotConf}(d_c, d) = \mathsf{true} \qquad \wp, L, l_t \vdash \langle h, \mathcal{I}^o_{dop}(b_t)_r\rangle \Downarrow_m \langle h_f, o_f\rangle}{\wp, L, l_t \vdash \langle h, \mathcal{I}^o_{dop}(\_, d, b_t, d_c)_5\rangle \Downarrow_m \langle h_f, o_f\rangle}$$

DOP-ACCESSORNOTCONF

$$\frac{\texttt{changeAccOnNotConf}(d_c, d) = \mathsf{true} \qquad \wp, L, l_t \vdash \langle h, \mathcal{I}^o_{dop}(b_t)_r\rangle \Downarrow_m \langle h_f, o_f\rangle}{\wp, L, l_t \vdash \langle h, \mathcal{I}^o_{dop}(\_, d, b_t, d_c)_6\rangle \Downarrow_m \langle h_f, o_f\rangle}$$

DOP-DIFFERENTCONF

$$\frac{d_c.[[C]] = \mathsf{true} \qquad \wp, L, l_t \vdash \langle h, \mathcal{I}^o_{dop}(p, d, d_c)_7\rangle \Downarrow_m \langle h_f, o_f\rangle}{\wp, L, l_t \vdash \langle h, \mathcal{I}^o_{dop}(p, d, \_, d_c)_4\rangle \Downarrow_m \langle h_f, o_f\rangle}$$

DOP-DATAOK

$$\frac{\texttt{changeDataOnNotConf}(d_c, d) = \mathsf{false} \qquad \wp, L, l_t \vdash \langle h, \mathcal{I}^o_{dop}(p, d, d_c)_w\rangle \Downarrow_m \langle h_f, o_f\rangle}{\wp, L, l_t \vdash \langle h, \mathcal{I}^o_{dop}(p, d, \_, d_c)_5\rangle \Downarrow_m \langle h_f, o_f\rangle}$$

DOP-ACCESSOROK

$$\frac{\texttt{changeAccOnNotConf}(d_c, d) = \mathsf{false} \qquad \wp, L, l_t \vdash \langle h, \mathcal{I}^o_{dop}(p, d, d_c)_w\rangle \Downarrow_m \langle h_f, o_f\rangle}{\wp, L, l_t \vdash \langle h, \mathcal{I}^o_{dop}(p, d, \_, d_c)_6\rangle \Downarrow_m \langle h_f, o_f\rangle}$$

DOP-TOACCESSOR

$$\frac{\begin{array}{l}\mathcal{P}_{dd}(d_c) \\ d' = \{[[G]] : \mathsf{undefined}, [[S]] : \mathsf{undefined}, [[C]] : d_c.[[C]], [[E]] : d_c.[[E]]\} \\ h' = h[(l_t, p) \mapsto d'] \\ \wp, L, l_t \vdash \langle h', \mathcal{I}^o_{dop}(p, d, d')_w\rangle \Downarrow_m \langle h_f, o_f\rangle\end{array}}{\wp, L, l_t \vdash \langle h, \mathcal{I}^o_{dop}(p, d, d_c)_7\rangle \Downarrow_m \langle h_f, o_f\rangle}$$

DOP-WRITE

$$\frac{d' = \texttt{updateDesc}(d_c, d) \qquad h_f = h[(l_t, p) \mapsto d']}{\wp, \_, l_t \vdash \langle h, \mathcal{I}^o_{dop}(p, d, d_c)_w\rangle \Downarrow_m \langle h_f, \mathsf{true}\rangle}$$

DOP-TODATA

$$\frac{\begin{array}{l}\neg\mathcal{P}_{dd}(d_c) \\ d' = \{[[V]] : \mathsf{undefined}, [[W]] : \mathsf{false}, [[C]] : d_c.[[C]], [[E]] : d_c.[[E]]\} \\ h' = h[(l_t, p) \mapsto d'] \\ \wp, L, l_t \vdash \langle h', \mathcal{I}^o_{dop}(p, d, d')_w\rangle \Downarrow_m \langle h_f, o_f\rangle\end{array}}{\wp, L, l_t \vdash \langle h, \mathcal{I}^o_{dop}(p, d, d_c)_7\rangle \Downarrow_m \langle h_f, o_f\rangle}$$

DOP-REJECTTHROW

$$\frac{h_f = h \uplus \mathsf{err}(l, l_{tep})}{\vdash \langle h, \mathcal{I}^o_{dop}(\mathsf{true})_r\rangle \Downarrow_m \langle h_f, \mathsf{error}\, l\rangle}$$

DOP-REJECTNOTHROW

$$\vdash \langle h, \mathcal{I}^o_{dop}(\mathsf{false})_r\rangle \Downarrow_m \langle h, \mathsf{false}\rangle$$

A.5. Auxiliary Internal Functions

1. `GetIdentifierReference(x)` — $\mathcal{I}_\sigma(x)$: scope chain traversal.

GIR-CURRENT

$$\frac{(l,x) \in \mathsf{dom}(h) \quad l \neq l_g}{\_, L \mathbin{@} [l], \_ \vdash \langle h, \mathcal{I}_\sigma(x)\rangle \Downarrow_m \langle h, l\rangle}$$

GIR-NEXT

$$\frac{(l,x) \notin \mathsf{dom}(h) \quad l \neq l_g \quad \wp, L, v_t \vdash \langle h, \mathcal{I}_\sigma(x)\rangle \Downarrow_m \langle h_f, o\rangle}{\wp, L \mathbin{@} [l], v_t \vdash \langle h, \mathcal{I}_\sigma(x)\rangle \Downarrow_m \langle h_f, o\rangle}$$

GIR-HASPROP

$$\frac{\wp, [l_g], l_g \vdash \langle h, \mathcal{I}_{hp}(x)\rangle \Downarrow_m \langle h, o_1\rangle \quad \wp, [l_g], v_t \vdash \langle h, \mathcal{I}_\sigma(o_1)_1\rangle \Downarrow_m \langle h_f, o_f\rangle}{\wp, [l_g], v_t \vdash \langle h, \mathcal{I}_\sigma(x)\rangle \Downarrow_m \langle h_f, o_f\rangle}$$

GIR-GLOBAL

$$\vdash \langle h, \mathcal{I}_\sigma(\mathsf{true})_1\rangle \Downarrow_m \langle h, l_g\rangle$$

GIR-UNDEF

$$\vdash \langle h, \mathcal{I}_\sigma(\mathsf{false})_1\rangle \Downarrow_m \langle h, \mathsf{undefined}\rangle$$

2. $\mathsf{iterate}\{\overline{e}\}$: returns the list of values obtained by evaluating and dereferencing each expression in $\overline{e}$.

ITERATE

$$\frac{\wp, L, v_t \vdash \langle h, \mathsf{iterate}_1\{\overline{e}, [\,]\}\rangle \Downarrow_m \langle h_f, \overline{v_f}\rangle}{\wp, L, v_t \vdash \langle h, \mathsf{iterate}\{\overline{e}\}\rangle \Downarrow_m \langle h_f, \overline{v_f}\rangle}$$

ITERATE - 1 (NON-EMPTY)

$$\frac{\wp, L, v_t \vdash \langle h, e\rangle \Downarrow_m^\gamma \langle h_1, v\rangle \quad \wp, L, v_t \vdash \langle h_1, \mathsf{iterate}_1\{\overline{e}, \overline{v} :: v\}\rangle \Downarrow_m \langle h_f, \overline{v_f}\rangle}{\wp, L, v_t \vdash \langle h, \mathsf{iterate}_1\{e :: \overline{e}, \overline{v}\}\rangle \Downarrow_m \langle h_f, \overline{v_f}\rangle}$$

ITERATE - 1 (EMPTY)

$$\vdash \langle h, \mathsf{iterate}_1\{[\,], \overline{v}\}\rangle \Downarrow_m \langle h, \overline{v}\rangle$$

3. Call(params, args) — $m(\overline{x}, \overline{v})$: evaluates the body of the function m.

CALL

$$\frac{\wp(m) = \lambda \overline{x}.s \quad \wp, L \mathbin{@} [l_s], l_t \vdash \langle h \uplus \mathsf{env}_m(l_s, \overline{x}, \overline{v}, s), s\rangle \Downarrow_m \langle h_f, o\rangle}{\wp, L, l_t \vdash \langle h, m(\overline{x}, \overline{v})\rangle \Downarrow_m \langle h_f, \mathsf{FunRet}(o)\rangle}$$

4. `ToPrimitive`$(v,\ x)$ — $\mathcal{I}_{tp}(v, x)$: converts v to a primitive value if it is an object.

TOPRIMITIVE - NOT AN OBJECT

$$\frac{\mathcal{P}_{pv}(v)}{\vdash \langle h, \mathcal{I}_{tp}(v, \_)\rangle \Downarrow_m \langle h, v\rangle}$$

TOPRIMITIVE - OBJECT

$$\frac{\wp, L, l \vdash \langle h, \mathcal{I}_{dv}(x)\rangle \Downarrow_m \langle h_f, o_f\rangle}{\wp, L, v_t \vdash \langle h, \mathcal{I}_{tp}(l, x)\rangle \Downarrow_m \langle h_f, o_f\rangle}$$

5. `ToNumber`(v) — $\mathcal{I}_{tn}(v)$: converts v to a number.

TONUMBER - PRIM

$$\frac{\mathcal{P}_{pv}(v)}{\vdash \langle h, \mathcal{I}_{tn}(v)\rangle \Downarrow_m \langle h, \mathcal{I}_{tn}^{prim}(v)\rangle}$$

TONUMBER - OBJECT

$$\frac{\wp, L, v_t \vdash \langle h, \mathcal{I}_{tp}(l, \text{“}\texttt{Number}\text{”})\rangle \Downarrow_m \langle h_1, o_1\rangle \quad \wp, L, v_t \vdash \langle h_1, \mathcal{I}_{tn}(o_1)_1\rangle \Downarrow_m \langle h_f, o_f\rangle}{\wp, L, v_t \vdash \langle h, \mathcal{I}_{tn}(l)\rangle \Downarrow_m \langle h_f, o_f\rangle}$$

ToNumber - 1

$$\vdash \langle h, \mathcal{I}_{tn}(v)_1 \rangle \Downarrow_m \langle h, \mathcal{I}_{tn}^{prim}(v) \rangle$$

6. `ToInteger`(v) — $\mathcal{I}_{ti}(v)$: converts v to an integer.

ToInteger - Prim

$$\frac{\mathcal{P}_{pv}(v)}{\vdash \langle h, \mathcal{I}_{ti}(v) \rangle \Downarrow_m \langle h, \mathcal{I}_{ti}^{prim}(v) \rangle}$$

ToInteger - Object

$$\frac{\wp, L, v_t \vdash \langle h, \mathcal{I}_{tn}(l) \rangle \Downarrow_m \langle h_1, o_1 \rangle \quad \wp, L, v_t \vdash \langle h_1, \mathcal{I}_{ti}(o_1)_1 \rangle \Downarrow_m \langle h_f, o_f \rangle}{\wp, L, v_t \vdash \langle h, \mathcal{I}_{ti}(l) \rangle \Downarrow_m \langle h_f, o_f \rangle}$$

ToInteger - 1

$$\vdash \langle h, \mathcal{I}_{ti}(v)_1 \rangle \Downarrow_m \langle h, \mathcal{I}_{ti}^{prim}(v) \rangle$$

7. `ToString`(v) — $\mathcal{I}_{ts}(v)$: converts v to a string.

ToString - Prim

$$\frac{\mathcal{P}_{pv}(v)}{\vdash \langle h, \mathcal{I}_{ts}(v) \rangle \Downarrow_m \langle h, \mathcal{I}_{ts}^{prim}(v) \rangle}$$

ToString - Object

$$\frac{\wp, L, v_t \vdash \langle h, \mathcal{I}_{tp}(l) \rangle \Downarrow_m \langle h_1, o_1 \rangle \quad \wp, L, v_t \vdash \langle h_1, \mathcal{I}_{ts}(o_1)_1 \rangle \Downarrow_m \langle h_f, o_f \rangle}{\wp, L, v_t \vdash \langle h, \mathcal{I}_{ts}(l) \rangle \Downarrow_m \langle h_f, o_f \rangle}$$

ToString - 1

$$\vdash \langle h, \mathcal{I}_{ts}(v)_1 \rangle \Downarrow_m \langle h, \mathcal{I}_{ts}^{prim}(v) \rangle$$

8. `ToObject`(v) — $\mathcal{I}_{to}(v)$: converts v to an object.

ToObject-throw

$$\frac{v = \mathsf{null} \vee v = \mathsf{undefined} \quad h_f = h \uplus \mathsf{err}(l, l_{tep})}{\vdash \langle h, \mathcal{I}_{to}(v) \rangle \Downarrow_m \langle h_f, \mathsf{error}\, l \rangle}$$

ToObject-boolean

$$\frac{\wp, L, l_t \vdash \langle h, \mathcal{I}_b^c(b) \rangle \Downarrow_m \langle h_f, o_f \rangle}{\wp, L, l_t \vdash \langle h, \mathcal{I}_{to}(b) \rangle \Downarrow_m \langle h_f, o_f \rangle}$$

ToObject-numbers

$$\frac{\wp, L, l_t \vdash \langle h, \mathcal{I}_n^c(n) \rangle \Downarrow_m \langle h_f, o_f \rangle}{\wp, L, l_t \vdash \langle h, \mathcal{I}_{to}(n) \rangle \Downarrow_m \langle h_f, o_f \rangle}$$

ToObject-strings

$$\frac{\wp, L, l_t \vdash \langle h, \mathcal{I}_s^c(m) \rangle \Downarrow_m \langle h_f, o_f \rangle}{\wp, L, l_t \vdash \langle h, \mathcal{I}_{to}(m) \rangle \Downarrow_m \langle h_f, o_f \rangle}$$

ToObject-object

$$\vdash \langle h, \mathcal{I}_{to}(l) \rangle \Downarrow_m \langle h, l \rangle$$

9. `CheckObjectCoercible`(v) — $\mathcal{I}_{coc}(v)$: throws an exception if v is null or $\mathsf{undefined}$.

COC

$$\frac{v \neq \mathsf{null} \wedge v \neq \mathsf{undefined}}{\vdash \langle h, \mathcal{I}_{coc}(v) \rangle \Downarrow_m \langle h, \mathsf{empty} \rangle}$$

COC-throw

$$\frac{v = \mathsf{null} \vee v = \mathsf{undefined} \quad h_f = h \uplus \mathsf{err}(l, l_{tep})}{\vdash \langle h, \mathcal{I}_{coc}(v) \rangle \Downarrow_m \langle h_f, \mathsf{error}\, l \rangle}$$

10. `IsCallable`$(h,\ v)$ — $\mathcal{P}_c(h, v)$: in order for a value $v \in \mathcal{V}_{\mathsf{JS}}$ to be callable, it must be an object that has the internal `@code` property.

IC-true

$$\frac{(l, @code) \in \mathsf{dom}(h)}{\mathcal{P}_c(h, l)}$$

IC-false

$$\frac{v \neq l \vee (l, @code) \notin \mathsf{dom}(h)}{\neg \mathcal{P}_c(h, v)}$$

11. IsPrimitiveValue(v) — $\mathcal{P}_{pv}(v)$: a value is primitive if it is not an object location.

$$\text{IPV-FALSE}\ \frac{}{\neg\mathcal{P}_{pv}(l)} \qquad \text{IPV-TRUE}\ \frac{v \neq l}{\mathcal{P}_{pv}(v)}$$

A.6. Operations on References

1. `Get(P)` — $\mathcal{I}^i_g(p)$ internal method for `GetValue`.

$$\text{IG-TOOBJECT}\ \frac{\wp, L, v_t \vdash \langle h, \mathcal{I}_{to}(v_t)\rangle \Downarrow_m \langle h_1, o\rangle \quad \wp, L, v_t \vdash \langle h_1, \mathcal{I}^i_g(p, o)_1\rangle \Downarrow_m \langle h_f, o_f\rangle}{\wp, L, v_t \vdash \langle h, \mathcal{I}^i_g(p)\rangle \Downarrow_m \langle h_f, o_f\rangle}$$

$$\text{IG-GETPROPERTY}\ \frac{\wp, L, l \vdash \langle h, \mathcal{I}_{gp}(p)\rangle \Downarrow_m \langle h_1, o\rangle \quad \wp, L, v_t \vdash \langle h_1, \mathcal{I}^i_g(o)_2\rangle \Downarrow_m \langle h_f, o_f\rangle}{\wp, L, v_t \vdash \langle h, \mathcal{I}^i_g(p, l)_1\rangle \Downarrow_m \langle h_f, o_f\rangle}$$

$$\text{IG-PROPUNDEFINED}\ \frac{}{\vdash \langle h, \mathcal{I}^i_g(\mathsf{undefined})_2\rangle \Downarrow_m \langle h, \mathsf{undefined}\rangle}$$

$$\text{IG-PROPDATA}\ \frac{\mathcal{P}_{dd}(d) \quad v = d.[[V]]}{\vdash \langle h, \mathcal{I}^i_g(\mathsf{desc}\ d)_2\rangle \Downarrow_m \langle h, v\rangle}$$

$$\text{IG-PROPACCESSORNOGET}\ \frac{\mathcal{P}_{ad}(d) \quad d.[[G]] = \mathsf{undefined}}{\vdash \langle h, \mathcal{I}^i_g(\mathsf{desc}\ d)_2\rangle \Downarrow_m \langle h, \mathsf{undefined}\rangle}$$

$$\text{IG-PROPACCESSORGET}\ \frac{\mathcal{P}_{ad}(d) \quad a_g = d.[[G]] \neq \mathsf{undefined} \quad \wp, L, a_g \vdash \langle h, \mathcal{I}_{call}(a_g, [\,])\rangle \Downarrow_m \langle h_f, o_f\rangle}{\wp, L, v_t \vdash \langle h, \mathcal{I}^i_g(\mathsf{desc}\ d)_2\rangle \Downarrow_m \langle h_f, o_f\rangle}$$

2. `Put(P,V)` — $\mathcal{I}^i_p(p, v)$ internal method for `PutValue`.

$$\text{IG-TOOBJECT}\ \frac{\wp, L, v_t \vdash \langle h, \mathcal{I}_{to}(v_t)\rangle \Downarrow_m \langle h_1, o\rangle \quad \wp, L, v_t \vdash \langle h_1, \mathcal{I}^i_p(p, v, o)_1\rangle \Downarrow_m \langle h_f, o_f\rangle}{\wp, L, v_t \vdash \langle h, \mathcal{I}^i_p(p, v)\rangle \Downarrow_m \langle h_f, o_f\rangle}$$

$$\text{IG-CANPUT}\ \frac{\wp, L, l \vdash \langle h, \mathcal{I}_{cp}(p)\rangle \Downarrow_m \langle h_1, o\rangle \quad \wp, L, v_t \vdash \langle h_1, \mathcal{I}^i_g(p, v, l, o)_2\rangle \Downarrow_m \langle h_f, o_f\rangle}{\wp, L, v_t \vdash \langle h, \mathcal{I}^i_g(p, v, l)_1\rangle \Downarrow_m \langle h_f, o_f\rangle}$$

$$\text{IG-CANNOTPUT}\ \frac{h_f = h \uplus \mathsf{err}(l, l_{tep})}{\vdash \langle h, \mathcal{I}^i_p(\_, \_, \_), \mathsf{false})_2\rangle \Downarrow_m \langle h_f, \mathsf{error}\, l\rangle}$$

$$\text{IG-CANPUTGETOWNPROP}\ \frac{\wp, L, l \vdash \langle h, \mathcal{I}_{gop}(p)\rangle \Downarrow_m \langle h_1, o\rangle \quad \wp, L, v_t \vdash \langle h_1, \mathcal{I}^i_p(p, v, l, o)_3\rangle \Downarrow_m \langle h_f, o_f\rangle}{\wp, L, v_t \vdash \langle h, \mathcal{I}^i_p(p, v, l, \mathsf{true})_2\rangle \Downarrow_m \langle h_f, o_f\rangle}$$

$$\text{IG-PROPDATA}\ \frac{\mathcal{P}_{dd}(d) \quad h_f = h \uplus \mathsf{err}(l, l_{tep})}{\vdash \langle h, \mathcal{I}^i_p(\_, \_, \_, \mathsf{desc}\ d)_3\rangle \Downarrow_m \langle h_f, \mathsf{error}\, l\rangle}$$

$$\text{IG-GETPROP}\ \frac{\neg\mathcal{P}_{dd}(d) \quad \wp, L, l \vdash \langle h, \mathcal{I}_{gp}(p)\rangle \Downarrow_m \langle h_1, o\rangle \quad \wp, L, v_t \vdash \langle h_1, \mathcal{I}^i_p(v, l, o)_4\rangle \Downarrow_m \langle h_f, o_f\rangle}{\wp, L, v_t \vdash \langle h, \mathcal{I}^i_p(p, v, l, \mathsf{desc}\ d)_3\rangle \Downarrow_m \langle h_f, o_f\rangle}$$

$$\text{IG-PROPACCESSORGET}\ \frac{\mathcal{P}_{ad}(d) \quad a_s = d.[[S]] \quad \wp, L, a_s \vdash \langle h, \mathcal{I}_{call}(a_s, [v])\rangle \Downarrow_m \langle h_1, o\rangle \quad \wp, L, v_t \vdash \langle h_1, \mathcal{I}^i_p(o)_5\rangle \Downarrow_m \langle h_f, o_f\rangle}{\wp, L, v_t \vdash \langle h, \mathcal{I}^i_p(v, l, \mathsf{desc}\ d)_4\rangle \Downarrow_m \langle h_f, o_f\rangle}$$

IG-PROPUNDEFINED

$$\frac{\neg\mathcal{P}_{ad}(d) \quad h_f = h \uplus \mathsf{err}(l, l_{tep})}{\vdash \langle h, \mathcal{I}^i_p(\_, \_, \mathsf{desc}\ d)_4\rangle \Downarrow_m \langle h_f, \mathsf{error}\, l\rangle}$$

IG-RETURN

$$\vdash \langle h, \mathcal{I}^i_p(v)_5\rangle \Downarrow_m \langle h, \mathsf{empty}\rangle$$

3. `GetValue(W)` — $\mathcal{I}_{gv}(w)$: obtaining the value from a reference.

GV-NOTREFERENCE

$$\vdash \langle h, \mathcal{I}_{gv}(v)\rangle \Downarrow_m \langle h, v\rangle$$

GV-UNRESOLVABLE

$$\frac{h_f = h \uplus \mathsf{err}(l, l_{rep})}{\vdash \langle h, \mathcal{I}_{gv}(\mathsf{undefined}.{}_{\mathsf{a}}p)\rangle \Downarrow_m \langle h_f, \mathsf{error}\, l\rangle}$$

GV-NOPRIMITIVEBASE

$$\frac{\wp, L, l \vdash \langle h, \mathcal{I}_g(p)\rangle \Downarrow_m \langle h_f, o_f\rangle}{\wp, L, v_t \vdash \langle h, \mathcal{I}_{gv}(l.{}_{\mathsf{o}}p)\rangle \Downarrow_m \langle h_f, o_f\rangle}$$

GV-PRIMITIVEBASE

$$\frac{v \notin \mathcal{L} \quad \wp, L, v \vdash \langle h, \mathcal{I}^i_g(p)\rangle \Downarrow_m \langle h_f, o_f\rangle}{\wp, L, v_t \vdash \langle h, \mathcal{I}_{gv}(v.{}_{\mathsf{o}}p)\rangle \Downarrow_m \langle h_f, o_f\rangle}$$

GV-VARIABLEREFERENCELG

$$\frac{\wp, L, l_g \vdash \langle h, \mathcal{I}_g(p)\rangle \Downarrow_m \langle h_f, o_f\rangle}{\wp, L, v_t \vdash \langle h, \mathcal{I}_{gv}(l_g.{}_{\mathsf{v}}p)\rangle \Downarrow_m \langle h_f, o_f\rangle}$$

GV-VARIABLEREFERENCENOTLG

$$\frac{l \neq l_g \quad v = h(l, p)}{\vdash \langle h, \mathcal{I}_{gv}(l.{}_{\mathsf{v}}p)\rangle \Downarrow_m \langle h, v\rangle}$$

4. `PutValue(W, V)` — $\mathcal{I}_{pv}(w, v)$: setting the value of a reference.

PV-NOTREFERENCE

$$\frac{h_f = h \uplus \mathsf{err}(l, l_{rep})}{\vdash \langle h, \mathcal{I}_{pv}(v, \_)\rangle \Downarrow_m \langle h_f, \mathsf{error}\, l\rangle}$$

PV-UNRESOLVABLE

$$\frac{h_f = h \uplus \mathsf{err}(l, l_{rep})}{\vdash \langle h, \mathcal{I}_{pv}(\mathsf{undefined}.{}_{\mathsf{a}}p, \_)\rangle \Downarrow_m \langle h_f, \mathsf{error}\, l\rangle}$$

PV-NOPRIMITIVEBASE

$$\frac{\begin{array}{l}\wp, L, l \vdash \langle h, \mathcal{I}_p(p, v, \mathsf{true})\rangle \Downarrow_m \langle h_1, o\rangle \\ \wp, L, v_t \vdash \langle h_1, \mathcal{I}_{pv}(o)_1\rangle \Downarrow_m \langle h_f, o_f\rangle\end{array}}{\wp, L, v_t \vdash \langle h, \mathcal{I}_{pv}(l.{}_{\mathsf{o}}p, v)\rangle \Downarrow_m \langle h_f, o_f\rangle}$$

PV-PRIMITIVEBASE

$$\frac{\begin{array}{l}\mathcal{P}_{pv}(v) \quad \wp, L, v \vdash \langle h, \mathcal{I}^i_p(p, v')\rangle \Downarrow_m \langle h_1, o\rangle \\ \wp, L, v_t \vdash \langle h_1, \mathcal{I}_{pv}(o)_1\rangle \Downarrow_m \langle h_f, o_f\rangle\end{array}}{\wp, L, v_t \vdash \langle h, \mathcal{I}_{pv}(v, p, v')_o\rangle \Downarrow_m \langle h_f, o_f\rangle}$$

PV-VARIABLEREFERENCELG

$$\frac{\begin{array}{l}\wp, L, l_g \vdash \langle h, \mathcal{I}_p(p, v, \mathsf{true})\rangle \Downarrow_m \langle h_1, o\rangle \\ \wp, L, v_t \vdash \langle h_1, \mathcal{I}_{pv}(o)_1\rangle \Downarrow_m \langle h_f, o_f\rangle\end{array}}{\wp, L, v_t \vdash \langle h, \mathcal{I}_{pv}(l_g.{}_{\mathsf{v}}p, v)\rangle \Downarrow_m \langle h_f, o_f\rangle}$$

PV-VARIABLEREFERENCENOTLG

$$\frac{\begin{array}{l}l \neq l_g \\ h_f = h[(l, p) \mapsto v]\end{array}}{\vdash \langle h, \mathcal{I}_{pv}(l.{}_{\mathsf{v}}p, v)\rangle \Downarrow_m \langle h_f, \mathsf{empty}\rangle}$$

PV-RETURN

$$\vdash \langle h, \mathcal{I}_{pv}(v)_1\rangle \Downarrow_m \langle h, \mathsf{empty}\rangle$$

A.7. Libraries

1. `new Boolean` (v) : constructs new boolean object with primitive value v.

BOOLEAN CONSTRUCTOR

$$\frac{h_f = h \uplus l \mapsto \{@\mathit{proto}: l_{bp},\ @\mathit{class}: \text{“}\texttt{Boolean}\text{”},\ @\mathit{extensible}: \mathsf{true},\ @\mathit{primval}: \mathcal{I}_{tb}(v)\}}{\vdash \langle h, \mathcal{I}^c_b(v)\rangle \Downarrow_m \langle h_f, l\rangle}$$

2. `new Number` ([v]) : constructs new number object with primitive value v.

NUMBER CONSTRUCTOR - NO VALUE

$$\frac{\wp, L, v_t \vdash \langle h, \mathcal{I}^c_n(+0)\rangle \Downarrow_m \langle h_f, o_f\rangle}{\wp, L, v_t \vdash \langle h, \mathcal{I}^c_n()\rangle \Downarrow_m \langle h_f, o_f\rangle}$$

NUMBER CONSTRUCTOR - VALUE

$$\frac{\wp, L, v_t \vdash \langle h, \mathcal{I}_{tn}(v)\rangle \Downarrow_m \langle h_1, o_1\rangle \quad \wp, L, v_t \vdash \langle h_1, \mathcal{I}^c_n(o_1)_1\rangle \Downarrow_m \langle h_f, o_f\rangle}{\wp, L, v_t \vdash \langle h, \mathcal{I}^c_n(v)\rangle \Downarrow_m \langle h_f, o_f\rangle}$$

NUMBER CONSTRUCTOR

$$\frac{h_f = h \uplus l \mapsto \{@proto\colon l_{np},\ @class\colon \text{“}\texttt{Number}\text{”},\ @extensible\colon \texttt{true},\ @primval\colon n\}}{\vdash \langle h, \mathcal{I}^c_n(n)_1\rangle \Downarrow_m \langle h_f, l\rangle}$$

3. `new String` ([v]) : constructs new string object with primitive value v.

STRING CONSTRUCTOR - NO VALUE

$$\frac{\wp, L, v_t \vdash \langle h, \mathcal{I}^c_s(\text{“”})\rangle \Downarrow_m \langle h_f, o_f\rangle}{\wp, L, v_t \vdash \langle h, \mathcal{I}^c_s()\rangle \Downarrow_m \langle h_f, o_f\rangle}$$

STRING CONSTRUCTOR - VALUE

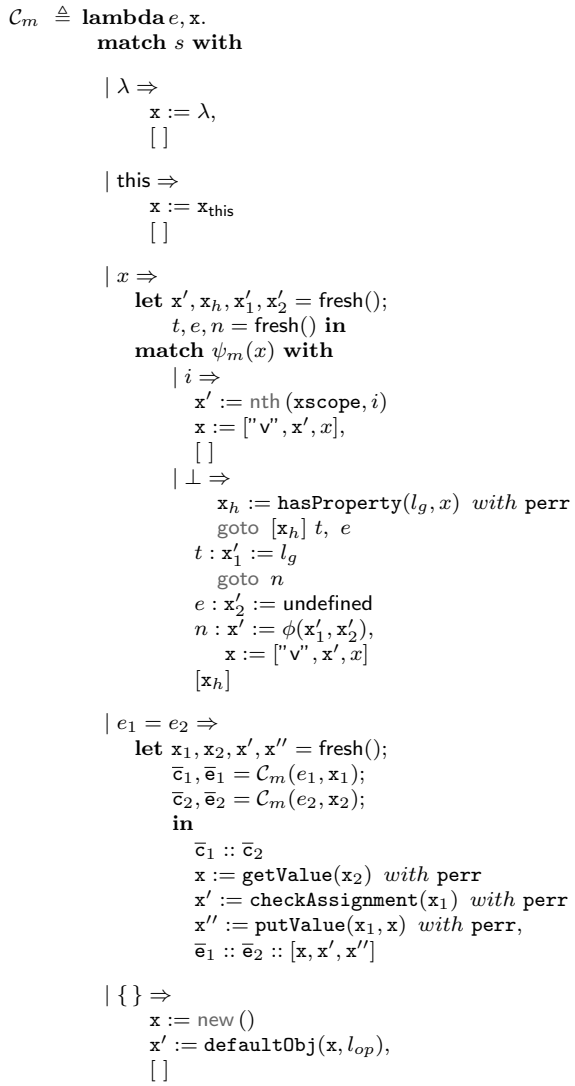
$$\frac{\wp, L, v_t \vdash \langle h, \mathcal{I}_{ts}(v)\rangle \Downarrow_m \langle h_1, o_1\rangle \quad \wp, L, v_t \vdash \langle h_1, \mathcal{I}^c_s(o_1)_1\rangle \Downarrow_m \langle h_f, o_f\rangle}{\wp, L, v_t \vdash \langle h, \mathcal{I}^c_s(v)\rangle \Downarrow_m \langle h_f, o_f\rangle}$$

STRING CONSTRUCTOR

$$\frac{h_f = h \uplus l \mapsto \{@proto\colon l_{sp},\ @class\colon \text{“}\texttt{String}\text{”},\ @extensible\colon \texttt{true},\ @primval\colon m\}}{\vdash \langle h, \mathcal{I}^c_s(m)_1\rangle \Downarrow_m \langle h_f, l\rangle}$$

B. Correctness Proof

B.1. The JS-2-JSIL Compiler



$\mathcal{C}_m \triangleq$ **lambda** $e, \mathtt{x}$.
match s **with**

| $\lambda \Rightarrow$
$\mathtt{x} := \lambda,$
$[\,]$

| this $\Rightarrow$
$\mathtt{x} := \mathtt{x}_{\mathsf{this}}$
$[\,]$

| $x \Rightarrow$
let $\mathtt{x}', \mathtt{x}_h, \mathtt{x}'_1, \mathtt{x}'_2 = \mathsf{fresh}();$
$t, e, n = \mathsf{fresh}()$ **in**
match $\psi_m(x)$ **with**
| $i \Rightarrow$
$\mathtt{x}' := \mathsf{nth}\,(\mathtt{xscope}, i)$
$\mathtt{x} := [\text{"v"}, \mathtt{x}', x],$
$[\,]$
| $\bot \Rightarrow$
$\mathtt{x}_h := \mathtt{hasProperty}(l_g, x)$ *with* perr
goto $[\mathtt{x}_h]$ t, e
$t : \mathtt{x}'_1 := l_g$
goto n
$e : \mathtt{x}'_2 :=$ undefined
$n : \mathtt{x}' := \phi(\mathtt{x}'_1, \mathtt{x}'_2),$
$\mathtt{x} := [\text{"v"}, \mathtt{x}', x]$
$[\mathtt{x}_h]$

| $e_1 = e_2 \Rightarrow$
let $\mathtt{x}_1, \mathtt{x}_2, \mathtt{x}', \mathtt{x}'' = \mathsf{fresh}();$
$\overline{\mathtt{c}}_1, \overline{\mathtt{e}}_1 = \mathcal{C}_m(e_1, \mathtt{x}_1);$
$\overline{\mathtt{c}}_2, \overline{\mathtt{e}}_2 = \mathcal{C}_m(e_2, \mathtt{x}_2);$
in
$\overline{\mathtt{c}}_1 :: \overline{\mathtt{c}}_2$
$\mathtt{x} := \mathtt{getValue}(\mathtt{x}_2)$ *with* perr
$\mathtt{x}' := \mathtt{checkAssignment}(\mathtt{x}_1)$ *with* perr
$\mathtt{x}'' := \mathtt{putValue}(\mathtt{x}_1, \mathtt{x})$ *with* perr,
$\overline{\mathtt{e}}_1 :: \overline{\mathtt{e}}_2 :: [\mathtt{x}, \mathtt{x}', \mathtt{x}'']$

| $\{\,\} \Rightarrow$
$\mathtt{x} :=$ new $()$
$\mathtt{x}' := \mathtt{defaultObj}(\mathtt{x}, l_{op}),$
$[\,]$

Inputs: an expression e and a JSIL variable $\mathtt{x}$
Branching on the type of expression to compile

Literal value
1. Output var is assigned to λ

This
1. Set output var to value of this

Variable

a. Obtain the index of the ER in which x is stored
b. x is statically declared in the original statement
1. Obtain the index of ER in which x is defined
2. Create a new var reference denoting the variable x

c. x is **not** statically declared in the original statement
1. Check if x is in the prototype chain of the global object
2. Branch depending on the global object containing x
3. l_g contains x: the base of reference is l_g
4. Go to the end of the reference creation.
5. l_g does not contain x: the base of reference is undefined
6. Joining of the two branches
7. Create the resulting reference

Assignment
a. Fresh vars
b. Compile e_1
c. Compile e_2
d. Generated code:
1. Compilation of e_1 and e_2
2. Obtain the value denoted by $\mathtt{x}_2$
3. Check if assignment to $\mathtt{x}_1$ is legal
4. Assign $\mathtt{x}$ to the $\mathtt{x}_1$

Object literal
1. Create new object and assign it to output var
2. Set prototype of new object to l_{op}

Figure B.1.: Compilation of Expressions, Part 1

$\mathcal{C}_m \triangleq$ **lambda** $e, \mathtt{x}$.
match e **with**

| delete $e \Rightarrow$
let $\mathtt{x}_1, \mathtt{x}_2, \mathtt{x}_3, \mathtt{x}_4 = \mathsf{fresh}()$;
$t_1, t_2, e_1, e_2, n_1, n_2 = \mathsf{fresh}()$;
$\overline{\mathsf{c}}, \overline{\mathsf{e}} = \mathcal{C}_m(e, \mathtt{x}_1)$;
in
$\overline{\mathsf{c}}$
goto $[\mathsf{typeOf}(\mathtt{x}_1) = \mathsf{List} \wedge (\mathsf{nth}(\mathtt{x}_1, 0) = \mathsf{v} \vee \mathsf{nth}(\mathtt{x}_1, 0) = \mathsf{o})]\ t_1,\ e_1$
t_1 : goto $[(\mathsf{nth}(\mathtt{x}_1, 1) = \mathsf{undefined})]\ t_2,\ e_2$
t_2 : $x_{err} := \mathtt{syntaxError}()$
goto **perr**
e_2 : goto $[\mathsf{nth}(\mathtt{x}_1, 0) = \mathsf{v}]\ t_2,\ n_1$
n_1 : $\mathtt{x}_2 := \mathtt{toObject}(\mathsf{nth}(\mathtt{x}_1, 1))$ *with* **perr**
$\mathtt{x}_3 := \mathtt{delete}(\mathtt{x}_2, \mathsf{nth}(\mathtt{x}_1, 2), \mathsf{true})$ *with* **perr**
goto n_2
e_1 : $\mathtt{x}_4 := \mathsf{true}$
n_2 : $\mathtt{x} := \phi(\mathtt{x}_3, \mathtt{x}_4)$,
$\overline{\mathsf{e}} :: [\mathtt{x}_{err}, \mathtt{x}_2, \mathtt{x}_3]$

Inputs: an expression e and a JSIL variable $\mathtt{x}$
Branching on the type of expression to compile

Field Deletion
a. Fresh vars
b. Fresh labels
c. Compile e
d. Generated code:
1. Compilation of e
2. Branch on $\mathtt{x}_1$ being a reference
3. Branch on $\mathtt{x}_1$ having undefined as its base
4. Creating a new syntax error
5. Jumping to the error section
6. Syntax error if $\mathtt{x}_1$ is a variable reference
7. Converting the base of $\mathtt{x}_1$ to an object
8. Calling internal Delete function
9. Jumping to the end of the generated code
10. If e does not evaluate to a reference, true is returned
11. Joining of the two branches

| function $(\overline{x})\{s\}^{m'} \Rightarrow$
let $\mathtt{x}' = \mathsf{fresh}()$;
in
$\mathtt{x}' :=$ new ()
$\mathtt{x}' := \mathtt{defaultObj}(\mathtt{x}, l_{op})$
$\mathtt{x} :=$ new ()
$[\mathtt{x}, @proto] := l_{fp}$
$[\mathtt{x}, @class] :=$ "Function"
$[\mathtt{x}, @extensible] := \mathsf{true}$
$[\mathtt{x}, @scope] := \mathtt{x}_{sc}$
$[\mathtt{x}, @code] := m'$
$[\mathtt{x}, prototype] := \mathtt{x}'$,
[ ]

Function Literal
a. Fresh var
b. Generated code:
1. Create prototype object
2. Set @*proto* of $\mathtt{x}'$ to default prototype
3. Create function object
4. Set property @*proto* of function object to the default function prototype
5. Set property @*class* of function object to "Function"
6. Set property @*extensible* of function object to true
7. Set property @*scope* of function object to the current scope chain
8. Set property @*code* of function object to the the identifier of the function literal
9. Set property prototype of function object to $\mathtt{x}'$

| $e_1 === e_2 \Rightarrow$
let $\mathtt{x}_1, \mathtt{x}_2, \mathtt{x}'_1, \mathtt{x}'_2 = \mathsf{fresh}()$;
$\overline{\mathsf{c}}_1, \overline{\mathsf{e}}_1 = \mathcal{C}_m(e_1, \mathtt{x}_1)$;
$\overline{\mathsf{c}}_2, \overline{\mathsf{e}}_2 = \mathcal{C}_m(e_2, \mathtt{x}_2)$;
in
$\overline{\mathsf{c}}_1$
$\mathtt{x}'_1 := \mathtt{getValue}(\mathtt{x}_1)$ *with* **perr**
$\overline{\mathsf{c}}_2$
$\mathtt{x}'_2 := \mathtt{getValue}(\mathtt{x}_2)$ *with* **perr**
$\mathtt{x} := \mathtt{strictEq}(\mathtt{x}'_1, \mathtt{x}'_2)$ *with* **perr**
$\overline{\mathsf{e}}_1 :: [\mathtt{x}'_1] :: \overline{\mathsf{e}}_2 :: [\mathtt{x}'_2, \mathtt{x}]$

Strict Equality
a. Fresh vars
b. Compile e_1
c. Compile e_2
d. Generated code:
1. Compilation of e_1
2. Obtain the value denoted by $\mathtt{x}_1$
3. Compilation of e_2
4. Obtain the value denoted by $\mathtt{x}_2$
5. Checking if strict equality holds

| $e_1 ! == e_2 \Rightarrow$
let $\mathtt{x}_1, \mathtt{x}_2, \mathtt{x}'_1, \mathtt{x}'_2, \mathtt{x}_3 = \mathsf{fresh}()$;
$\overline{\mathsf{c}}_1, \overline{\mathsf{e}}_1 = \mathcal{C}_m(e_1, \mathtt{x}_1)$;
$\overline{\mathsf{c}}_2, \overline{\mathsf{e}}_2 = \mathcal{C}_m(e_2, \mathtt{x}_2)$;
in
$\overline{\mathsf{c}}_1$
$\mathtt{x}'_1 := \mathtt{getValue}(\mathtt{x}_1)$ *with* **perr**
$\overline{\mathsf{c}}_2$
$\mathtt{x}'_2 := \mathtt{getValue}(\mathtt{x}_2)$ *with* **perr**
$\mathtt{x}_3 := \mathtt{strictEq}(\mathtt{x}'_1, \mathtt{x}'_2)$ *with* **perr**
$\mathtt{x} := \neg \mathtt{x}_3$
$\overline{\mathsf{e}}_1 :: [\mathtt{x}'_1] :: \overline{\mathsf{e}}_2 :: [\mathtt{x}'_2, \mathtt{x}_3]$

Strict Inequality
a. Fresh vars
b. Compile e_1
c. Compile e_2
d. Generated code:
1. Compilation of e_1
2. Obtain the value denoted by $\mathtt{x}_1$
3. Compilation of e_2
4. Obtain the value denoted by $\mathtt{x}_2$
5. Checking if strict equality holds
6. Negating the result of strict equality

Figure B.2.: Compilation of Expressions, Part 2

$\mathcal{C}_m \triangleq$ **lambda** $e, \mathtt{x}$.
match e **with**

$| \ e_1 + e_2 \Rightarrow$
let $\mathtt{x}_i|_{i=1}^{10}, \mathtt{x}'_1, \mathtt{x}'_2 = \mathsf{fresh}();$
$t_1, e_1, n = \mathsf{fresh}();$
$\overline{\mathtt{c}}_1, \overline{\mathtt{e}}_1 = \mathcal{C}_m(e_1, \mathtt{x}_1);$
$\overline{\mathtt{c}}_2, \overline{\mathtt{e}}_2 = \mathcal{C}_m(e_2, \mathtt{x}_2);$
in
$\overline{\mathtt{c}}_1$
$\mathtt{x}'_1 := \mathtt{getValue}(\mathtt{x}_1)$ *with* `perr`
$\overline{\mathtt{c}}_2$
$\mathtt{x}'_2 := \mathtt{getValue}(\mathtt{x}_2)$ *with* `perr`
$\mathtt{x}_3 := \mathtt{toPrimitive}(\mathtt{x}'_1)$ *with* `perr`
$\mathtt{x}_4 := \mathtt{toPrimitive}(\mathtt{x}'_2)$ *with* `perr`
goto $[\mathsf{typeOf}(\mathtt{x}_3) = \mathsf{Str} \vee \mathsf{typeOf}(\mathtt{x}_4) = \mathsf{Str}] \ t_1, \ e_1$
$t_1 : \mathtt{x}_5 := \mathtt{toString}(\mathtt{x}_3)$ *with* `perr`
$\mathtt{x}_6 := \mathtt{toString}(\mathtt{x}_4)$ *with* `perr`
$\mathtt{x}_7 := \mathtt{x}_5 \ @_s \ \mathtt{x}_6$
goto n
$e_1 : \mathtt{x}_8 := \mathtt{toNumber}(\mathtt{x}_3)$ *with* `perr`
$\mathtt{x}_9 := \mathtt{toNumber}(\mathtt{x}_4)$ *with* `perr`
$\mathtt{x}_{10} := \mathtt{x}_8 + \mathtt{x}_9$
$n : \mathtt{x} := \phi(\mathtt{x}_7, \mathtt{x}_{10}),$
$\overline{\mathtt{e}}_1 :: [\mathtt{x}'_1] :: \overline{\mathtt{e}}_2 :: [\mathtt{x}'_2, \mathtt{x}_3, \mathtt{x}_4, \mathtt{x}_5, \mathtt{x}_6, \mathtt{x}_8, \mathtt{x}_9]$

$| \ e(e_1, \cdots, e_n) \Rightarrow$
let $\mathtt{x}_e, \mathtt{x}_i|_{i=1}^n, \mathtt{x}'_e, \mathtt{x}'_i|_{i=1}^n, \mathtt{x}_{err}, \mathtt{x}_c, \mathtt{x}_{t1}, \mathtt{x}_{t2}, \mathtt{x}_t,$
$\mathtt{x}_{scp}, \mathtt{x}_{m'} = \mathsf{fresh}();$
$t_1, t_2, e_1, e_2, n_1, n_2 = \mathsf{fresh}();$
$\overline{\mathtt{c}}_e, \overline{\mathtt{e}}_e = \mathcal{C}_m(e, \mathtt{x}_e);$
$\overline{\mathtt{c}}_i, \overline{\mathtt{e}}_i = \mathcal{C}_m(e_i, \mathtt{x}_i)|_{i=1}^n;$
$\overline{\mathtt{c}}'_i = \mathtt{x}'_i := \mathtt{getValue}(\mathtt{x}_i)$ *with* $\mathtt{perr}|_{i=1}^n;$
in
$\overline{\mathtt{c}}_e$
$\mathtt{x}'_e := \mathtt{getValue}(\mathtt{x}_e)$ *with* `perr`
$\{\overline{\mathtt{c}}_i :: \overline{\mathtt{c}}'_i\}|_{i=1}^n$
goto $[\mathsf{typeOf}(\mathtt{x}'_e) \ != \mathsf{Obj}] \ t_1, \ e_1$
$t_1 : x_{err} := \mathtt{typeError}()$
goto `perr`
$e_1 : \mathtt{x}_c := \mathtt{isCallable}(\mathtt{x}'_e)$
goto $[\mathtt{x}_c] \ n_1, \ t_1$
$n_1 :$ goto $[\mathsf{typeOf}(\mathtt{x}_e) = \mathsf{List} \wedge (\mathsf{nth}(\mathtt{x}_e, 0) = \mathsf{o}] \ t_2, \ e_2$
$t_2 : \mathtt{x}_{t1} := \mathsf{nth}(\mathtt{x}_e, 1)$
goto n_2
$e_2 : \mathtt{x}_{t2} :=$ `undefined`
$n_2 : \mathtt{x}_t := \phi(\mathtt{x}_{t1}, \mathtt{x}_{t2})$
$\mathtt{x}_{scp} := [\mathtt{x}'_e, @scope]$
$\mathtt{x}_{m'} := [\mathtt{x}'_e, @code]$
$\mathtt{x} := \mathtt{x}_{m'}(\mathtt{x}_{scp}, \mathtt{x}_t, \mathtt{x}'_1, ..., \mathtt{x}'_n)$ with `perr`
$\overline{\mathtt{e}}_e :: [\mathtt{x}'_e] :: (\overline{\mathtt{e}}_i :: [\mathtt{x}'_i])|_{i=1}^n :: [\mathtt{x}_{err}, \mathtt{x}_1, \mathtt{x}]$

Inputs: an expression e and a JSIL variable $\mathtt{x}$
Branching on the type of expression to compile

Addition Operator
a. Fresh vars
b. Fresh labels
c. Compile e_1
d. Compile e_2
e. Generated code:
1. Compilation of e_1
2. Obtain the value denoted by $\mathtt{x}_1$
3. Compilation of e_2
4. Obtain the value denoted by $\mathtt{x}_2$
5. Converting $\mathtt{x}'_1$ to primitive
6. Converting $\mathtt{x}'_2$ to primitive
7. Branch on either $\mathtt{x}'_1$ or $\mathtt{x}'_2$ being a string
8. Converting the first operand to string
9. Converting the second operand to string
10. String concatenation
11. Go to the end of the generated code.
12. Converting the first operand to number
13. Converting the second operand to number
14. Addition operation
15. Joining of the two branches

Function/Method Call
a. Fresh vars
b. Fresh labels
c. Compile e
d. Compile arguments $e_i|_{i=1}^n$
e. Generate the code to dereference $\mathtt{x}_i|_{i=1}^n$
f. Generated code:
1. Compilation of e
2. Dereferencing of $\mathtt{x}_e$
3. Compilation of arguments and their dereferencing
4. Branch on $\mathtt{x}'_e$ being an object
5. Type error if $\mathtt{x}'_e$ in not a callable object
6. Go to the error section
7. Check if $\mathtt{x}'_e$ is callable
8. If $\mathtt{x}'_e$ is not callable throw type error
9. Branch on $\mathtt{x}_e$ being an object reference
10. Set this to the base of an object reference $\mathtt{x}_e$
11. Go to the next joining command
12. Set this to undefined
13. Joining of the two branches
14. Set $\mathtt{x}_{scp}$ to the $@scope$ property of $\mathtt{x}'_e$
15. Set $\mathtt{x}_{m'}$ to the $@code$ property of $\mathtt{x}'_e$
16. Call the procedure with identifier $\mathtt{x}_{m'}$

Figure B.3.: Compilation of Expressions, Part 3

$\mathcal{C}_m \triangleq$ **lambda** $e, \mathtt{x}$.
match e **with**

| new $e(e_1, \cdots, e_n) \Rightarrow$
let $\mathtt{x}_e, \mathtt{x}'_e, \mathtt{x}_i|_{i=1}^n, \mathtt{x}'_i|_{i=1}^n, \mathtt{x}_{err}, \mathtt{x}_{hp}, \mathtt{x}_l, \mathtt{x}_r,$
$\mathtt{x}'_p, \mathtt{x}''_p, \mathtt{x}_p, \mathtt{x}'_l, \mathtt{x}_{m'}, \mathtt{x}_{scp}, \mathtt{x}' = \mathsf{fresh}();$
$t_1, t_2, t_3, e_1, e_2, e_3, n = \mathsf{fresh}();$
$\overline{\mathtt{c}}_e, \overline{\mathtt{e}}_e = \mathcal{C}_m(e, \mathtt{x}_e);$
$\overline{\mathtt{c}}_i, \overline{\mathtt{e}}_i = \mathcal{C}_m(e_i, \mathtt{x}_i)|_{i=1}^n;$
$\overline{\mathtt{c}}'_i = \mathtt{x}'_i := \mathtt{getValue}(\mathtt{x}_i)$ *with* $\mathtt{perr}|_{i=1}^n;$
in
$\overline{\mathtt{c}}_e$
$\mathtt{x}'_e := \mathtt{getValue}(\mathtt{x}_e)$ *with* perr
$\{\overline{\mathtt{c}}_i :: \overline{\mathtt{c}}'_i\}|_{i=1}^n$
goto [typeOf $(\mathtt{x}'_e)$!= Obj] t_1, e_1
$t_1 : x_{err} := \mathtt{typeError}()$
goto perr
$e_1 : \mathtt{x}_{hp} :=$ hasProperty $(\mathtt{x}'_e, @code)$
goto $[\mathtt{x}_{hp}]$ n, t_1
$n : \mathtt{x}_l :=$ new $()$
$\mathtt{x}_r := ["\mathtt{o}", \mathtt{x}'_e, prototype]$
$\mathtt{x}'_p := \mathtt{getValue}(\mathtt{x}_r)$ *with* perr
goto [typeOf $(\mathtt{x}_p)$ = Obj] t_2, e_2
$t_2 : \mathtt{x}''_p := l_{op}$

$e_2 : \mathtt{x}_p := \phi(\mathtt{x}'_p, \mathtt{x}''_p)$
$\mathtt{x}'_l := \mathtt{defaultObj}(\mathtt{x}_l, \mathtt{x}_p)$

$\mathtt{x}_{m'} := [\mathtt{x}'_e, @code]$
$\mathtt{x}_{scp} := [\mathtt{x}'_e, @scope]$
$\mathtt{x}' := \mathtt{x}_{m'}(\mathtt{x}_{scp}, \mathtt{x}_l, \mathtt{x}'_i|_{i=1}^n)$ with perr
goto [typeOf $(\mathtt{x}')$ = Obj] t_3, e_3
$e_3 :$ skip
$t_3 : \mathtt{x} := \phi(\mathtt{x}', \mathtt{x}_l),$
$\overline{\mathtt{e}}_e :: [\mathtt{x}'_e] :: (\overline{\mathtt{e}}_i :: [\mathtt{x}'_i])|_{i=1}^n :: [\mathtt{x}_{err}, \mathtt{x}'_p, \mathtt{x}']$

| typeof $e \Rightarrow$
let $\mathtt{x}_1, \mathtt{x}_2, \mathtt{x}_3, \mathtt{x}_4 = \mathsf{fresh}();$
$t_1, e_1, n_1, n_2 = \mathsf{fresh}();$
$\overline{\mathtt{c}}, \overline{\mathtt{e}} = \mathcal{C}_m(e, \mathtt{x}_1);$
in
$\overline{\mathtt{c}}$
goto [typeOf $(\mathtt{x}_1)$ = List $\wedge$ (nth $(\mathtt{x}_1, 0)$ = v
$\vee$nth $(\mathtt{x}_1, 0)$ = o)] t_1, e_1
$t_1 :$ goto [(nth $(\mathtt{x}_1, 1)$ = undefined)] n_1, e_1
$n_1 : \mathtt{x}_2 :=$ "undefined"
goto n_2
$e_1 : \mathtt{x}_3 := \mathtt{getValue}(\mathtt{x}_1)$ *with* perr
$\mathtt{x}_4 := \mathtt{typeof}(\mathtt{x}_3)$ *with* perr
$n_2 : \mathtt{x} := \phi(\mathtt{x}_2, \mathtt{x}_4),$
$\overline{\mathtt{e}} :: [\mathtt{x}_3, \mathtt{x}_4]$

Inputs: an expression e and a JSIL variable x
Branching on the type of expression to compile

Constructor Call
a. Fresh vars
b. Fresh labels
c. Compile e
d. Compile arguments $e_i|_{i=1}^n$
e. Generate the code to dereference arguments
f. Generated code:
1. Compilation of e
2. Dereferencing $\mathtt{x}_e$
3. Compilation of arguments and their dereferencing
4. Branch on $\mathtt{x}'_e$ being an object
5. Type error if $\mathtt{x}'_e$ in not a constructor object
6. Go to the error section
7. Check if $\mathtt{x}'_e$ has property $@code$
8. If $\mathtt{x}'_e$ does not have $@code$ throw type error
9. Create the this object
10. Obtaining the property *prototype* of the function object $\mathtt{x}'_e$
11. Branch on the property *prototype* of $\mathtt{x}'_e$ being an object
12. If the property *prototype* of $\mathtt{x}'_e$ is not an object: use l_{op}
13. Joining of the two branches
14. Set prototype of newly created object $\mathtt{x}_l$ to selected prototype $\mathtt{x}_p$
15. Set $\mathtt{x}_{m'}$ to the $@code$ property of $\mathtt{x}'_e$
16. Set $\mathtt{x}_{scp}$ to the $@scope$ property of $\mathtt{x}'_e$
17. Call the procedure with identifier m'
18. Branch depending on whether or not $\mathtt{x}'$ is an object
19. Joining of the two branches

Typeof Operator
a. Fresh vars
b. Fresh labels
c. Compile e
d. Generated code:
1. Compilation of e
2. Branch on $\mathtt{x}_1$ being a reference
3. Branch on $\mathtt{x}_1$ having undefined as its base
4. The type of unresolved reference is "undefined"
5. Go to the end of the generated code
6. Dereferencing $\mathtt{x}_1$
7. Getting the type
8. Joining of the two branches

Figure B.4.: Compilation of Expressions, Part 4

$\mathcal{C}_m \triangleq$ **lambda** $s, \mathtt{x}, \mathtt{x}_{pr}, b_r$.

Inputs: a statement s, a JSIL variable $\mathtt{x}$, that denotes the result of the current statement, a JSIL variable $\mathtt{x}_{pr}$, that denotes the result of the previous statement, and a label b_r for break statements

match s **with**

Branching on the type of statement to compile

| var $x \Rightarrow$
  $\mathtt{x} = \mathsf{empty}$,
  $[\,], [\,], [\,]$

Variable Declaration
- **a.** Variable declarations evaluate to empty

| $e \Rightarrow$
  let $\mathtt{x}_e = \mathsf{fresh}()$;
  $\overline{\mathtt{c}}_e, \overline{\mathtt{e}}_e = \mathcal{C}_m(e, \mathtt{x}_e)$;
  in
  $\overline{\mathtt{c}}_e$
  $\mathtt{x} := \mathtt{getValue}(\mathtt{x}_e)$ *with* perr
  $\overline{\mathtt{e}}_e :: [\mathtt{x}], [\,], [\,]$

Expression Statement
- **a.** Fresh vars
- **b.** Compile e
- **c.** Generated code:
  - **1.** Compilation of e
  - **2.** Dereferencing of $\mathtt{x}_e$

| $s_1; s_2 \Rightarrow$
  let $\mathtt{x}_1, \mathtt{x}_2 = \mathsf{fresh}()$;
  $t, e = \mathsf{fresh}()$;
  $\overline{\mathtt{c}}_1, \overline{\mathtt{e}}_1, \overline{\mathtt{r}}_1, \overline{\mathtt{b}}_1 = \mathcal{C}_m(s_1, \mathtt{x}_1, \mathtt{x}_{pr}, b_r)$;
  $\overline{\mathtt{c}}_2, \overline{\mathtt{e}}_2, \overline{\mathtt{r}}_2, \overline{\mathtt{b}}_2 = \mathcal{C}_m(s_2, \mathtt{x}_2, \mathtt{x}_1, b_r)$;
  in
  $\overline{\mathtt{c}}_1$
  $\overline{\mathtt{c}}_2$
  goto $[\mathtt{x}_2 = \mathsf{empty}]\ t,\ e$
  e : skip
  $t : \mathtt{x} := \phi(\mathtt{x}_1, \mathtt{x}_2)$,
  $\overline{\mathtt{e}}_1 :: \overline{\mathtt{e}}_2, \overline{\mathtt{r}}_1 :: \overline{\mathtt{r}}_2, \overline{\mathtt{b}}_1 :: \overline{\mathtt{b}}_2$

Sequence Statement
- **a.** Fresh vars to hold the return values of s_1 and s_2
- **b.** Fresh labels
- **c.** Compile s_1
- **d.** Compile s_2
- **e.** Generated code:
  - **1.** Compilation of s_1
  - **2.** Compilation of s_2
  - **3.** Branch on $\mathtt{x}_2$ being equal to empty
  - **4.** Joining of the two branches

| if$(e)\,\{s_1\}$ else $\{s_2\} \Rightarrow$
  let $\mathtt{x}_e, \mathtt{x}'_e, \mathtt{x}_1, \mathtt{x}_2, \mathtt{x}_b = \mathsf{fresh}()$;
  $t, e, n = \mathsf{fresh}()$;
  $\overline{\mathtt{c}}_e, \overline{\mathtt{e}}_e = \mathcal{C}_m(e, \mathtt{x}_e)$;
  $\overline{\mathtt{c}}_1, \overline{\mathtt{e}}_1, \overline{\mathtt{r}}_1, \overline{\mathtt{b}}_1 = \mathcal{C}_m(s_1, \mathtt{x}_1, \mathtt{x}_{pr}, b_r)$;
  $\overline{\mathtt{c}}_2, \overline{\mathtt{e}}_2, \overline{\mathtt{r}}_2, \overline{\mathtt{b}}_2 = \mathcal{C}_m(s_2, \mathtt{x}_2, \mathtt{x}_{pr}, b_r)$;
  in
  $\overline{\mathtt{c}}_e$
  $\mathtt{x}'_e := \mathtt{getValue}(\mathtt{x}_e)$ *with* perr
  $\mathtt{x}_b := \mathtt{toBoolean}(\mathtt{x}'_e)$ *with* perr
  goto $[\mathtt{x}_b]\ t,\ e$
  $t : \overline{\mathtt{c}}_1$
  goto n
  $e : \overline{\mathtt{c}}_2$
  $n : \mathtt{x} := \phi(\mathtt{x}_1, \mathtt{x}_2)$,
  $\overline{\mathtt{e}}_e :: [\mathtt{x}'_e, \mathtt{x}_b] :: \overline{\mathtt{e}}_1 :: \overline{\mathtt{e}}_2$,
  $\overline{\mathtt{r}}_1 :: \overline{\mathtt{r}}_2, \overline{\mathtt{b}}_1 :: \overline{\mathtt{b}}_2$

If Statement
- **a.** Fresh vars
- **b.** Fresh labels
- **c.** Compile e
- **d.** Compile s_1
- **e.** Compile s_2
- **f.** Generated code:
  - **1.** Compilation of e
  - **2.** Dereferencing of $\mathtt{x}_e$
  - **3.** Converting the if condition to boolean
  - **4.** Branch on $\mathtt{x}_b$
  - **5.** The if condition holds: compilation of s_1
  - **6.** Go to the end of the generated code
  - **7.** The if condition does not hold: compilation of s_2
  - **8.** Joining of the two branches

| while$(e)\,\{s\} \Rightarrow$
  let $\mathtt{x}_e, \mathtt{x}_s, \mathtt{x}'_e, \mathtt{x}', \mathtt{x}'', \mathtt{x}''', \mathtt{x}_b = \mathsf{fresh}()$;
  $t, e, n, h, b = \mathsf{fresh}()$;
  $\overline{\mathtt{c}}_e, \overline{\mathtt{e}}_e = \mathcal{C}_m(e, \mathtt{x}_e)$;
  $\overline{\mathtt{c}}_s, \overline{\mathtt{e}}_s, \overline{\mathtt{r}}_s, \overline{\mathtt{b}}_s = \mathcal{C}_m(s, \mathtt{x}_s, \mathtt{x}'', b)$;
  in
  $\mathtt{x}' := \mathsf{empty}$
  $h : \mathtt{x}'' := \phi(\mathtt{x}', \mathtt{x}''')$
  $\overline{\mathtt{c}}_e$
  $\mathtt{x}'_e := \mathtt{getValue}(\mathtt{x}_e)$ *with* perr
  $\mathtt{x}_b := \mathtt{toBoolean}(\mathtt{x}'_e)$ *with* perr
  goto $[\mathtt{x}_b]\ n,\ b$
  $n : \overline{\mathtt{c}}_s$
  goto $[\mathtt{x}_s\ \mathsf{!=}\ \mathsf{empty}]\ t,\ e$
  t : skip
  $e : \mathtt{x}''' := \phi(\mathtt{x}'', \mathtt{x}_s)$
  goto h
  $b : \mathtt{x} := \phi(\mathtt{x}'', \overline{\mathtt{b}}_s)$,
  $\overline{\mathtt{e}}_e :: [\mathtt{x}'_e, \mathtt{x}_b] :: \overline{\mathtt{e}}_s, \overline{\mathtt{r}}_s, [\,]$

While Statement
- **a.** Fresh vars
- **b.** Fresh labels
- **c.** Compile e
- **d.** Compile s
- **e.** Generated code:
  - **1.** Result of while is empty in case no iterations occur
  - **2.** Joining the branch of no iterations with the branch iterating the while body
  - **3.** Compilation of e
  - **4.** Dereferencing of $\mathtt{x}_e$
  - **5.** Converting the while condition to boolean
  - **6.** Branch on $\mathtt{x}_b$
  - **7.** The while condition holds: compilation of s
  - **8.** Branch on $\mathtt{x}_s$ not being equal to empty
  - **9.** $\mathtt{x}_s$ is not equal to empty: the result is $\mathtt{x}_s$
  - **10.** $\mathtt{x}_s$ is empty: the result is the value of the previous iteration
  - **11.** Proceed to the next iteration
  - **12.** The while condition does not hold: exit the loop

Figure B.5.: Compilation of Statements, Part 1

$\mathcal{C}_m \triangleq$ **lambda** $s, \mathtt{x}, \mathtt{x}_{pr}, b_r$.

Inputs: a statement s, a JSIL variable $\mathtt{x}$, that denotes the result of the current statement, a JSIL variable $\mathtt{x}_{pr}$, that denotes the result of the previous statement, and a label b_r for break statements

match s **with**

Branching on the type of statement to compile

| return $e \Rightarrow$
  let $\mathtt{x}_e = \mathsf{fresh}()$;
  $\overline{\mathtt{c}}_e, \overline{\mathtt{e}}_e = \mathcal{C}_m(e, \mathtt{x}_e)$;
  in
  $\overline{\mathtt{c}}_e$
  $\mathtt{x} := \mathtt{getValue}(\mathtt{x}_e)$ *with* perr
  goto pret,
  $\overline{\mathtt{e}}_e :: [\mathtt{x}], [\mathtt{x}], [\,]$

Return Statement
- **a.** Fresh vars
- **b.** Compile e
- **c.** Generated code:
  1. Compilation of e
  2. Dereferencing of $\mathtt{x}_e$
  3. Go to the return section

| throw $e \Rightarrow$
  let $\mathtt{x}_e = \mathsf{fresh}()$;
  $\overline{\mathtt{c}}_e, \overline{\mathtt{e}}_e = \mathcal{C}_m(e, \mathtt{x}_e)$;
  in
  $\overline{\mathtt{c}}_e$
  $\mathtt{x} := \mathtt{getValue}(\mathtt{x}_e)$ *with* perr
  goto perr,
  $\overline{\mathtt{e}}_e :: [\mathtt{x}, \mathtt{x}], [\,], [\,]$

Throw Statement
- **a.** Fresh vars
- **b.** Compile e
- **c.** Generated code:
  1. Compilation of e
  2. Dereferencing of $\mathtt{x}_e$
  3. Go to the error section

| break $\Rightarrow$
  $\mathtt{x} := \mathtt{x}_{pr}$
  goto b_r,
  $[\,], [\,], [\mathtt{x}]$

Break Statement
- **a.** Generated code:
  1. The result is the provided previous value $\mathtt{x}_{pr}$
  2. Go to the provided break label

Figure B.6.: Compilation of Statements, Part 2

B.2. Compiler Correctness

JSIL semantics defined in §4 Figure 4.5 describes evaluation of JSIL procedures starting at some label i and ending at the final label. To prove the compiler correctness we need to define an alternative JSIL semantics, which allows us to describe evaluation of JSIL procedure from some label i up to some label j. In §B.2.1, we define the alternative JSIL semantics $\mathtt{p} \vdash \langle h, \rho, k, i\rangle \rightarrow_m^* \langle h', \rho', j, l\rangle$ (Definition B.1) and state the equivalence to the JSIL semantics defined in §4 Figure 4.5 (Lemma 3). In §B.2.2, we prove the compiler correctness theorem B.1 stated in terms of the alternative JSIL semantics. To prove the theorem we also need to describe what does it mean for the translation of JavaScript expression e (Lemma 4) and the translation of JavaScript statement s (Lemma 5) to be correct. In §B.2.3, we state a list of helper lemmas that we use in the compiler correctness proof.

B.2.1. Alternative JSIL semantics

Definition B.1 (Alternative JSIL Semantics)**.** *A state transition relation* $\mathtt{p} \vdash \langle h, \rho, k, i\rangle \rightarrow_m^* \langle h', \rho', j, l\rangle$ *relates the state before starting to execute* i^{th} *command of JSIL procedure* m *of program* $\mathtt{p}$ *and the state after executing* j^{th} *command. Labels* k *and* l *point to the previous and the following commands, respectively. The relation is defined by the following rules:*

$$\textsc{Basic Command}\quad \frac{\mathtt{p}_m(i) = \mathtt{bc} \in \mathtt{BCmd} \quad [\![\mathtt{bc}]\!]_{h,\rho} = (h', \rho', -)}{\mathtt{p} \vdash \langle h, \rho, \_, i\rangle \rightarrow_m^* \langle h', \rho', i, i+1\rangle}$$

$$\textsc{Goto}\quad \frac{\mathtt{p}_m(i) = \mathit{goto}\ j}{\mathtt{p} \vdash \langle h, \rho, \_, i\rangle \rightarrow_m^* \langle h, \rho, i, j\rangle}$$

$$\textsc{Cond. Goto - True}\quad \frac{\mathtt{p}_m(i) = \mathit{goto}\ [\mathtt{e}]\ j,\ k \quad [\![\mathtt{e}]\!]_\rho = \mathit{true}}{\mathtt{p} \vdash \langle h, \rho, \_, i\rangle \rightarrow_m^* \langle h, \rho, i, j\rangle}$$

$$\textsc{Cond. Goto - False}\quad \frac{\mathtt{p}_m(i) = \mathit{goto}\ [\mathtt{e}]\ j,\ k \quad [\![\mathtt{e}]\!]_\rho = \mathit{false}}{\mathtt{p} \vdash \langle h, \rho, \_, i\rangle \rightarrow_m^* \langle h, \rho, i, k\rangle}$$

$$\textsc{Procedure Call - Normal}\quad \frac{\begin{array}{l}\mathtt{p}_m(i) = \mathtt{x} := \mathtt{e}(\mathtt{e}_1, ..., \mathtt{e}_{n_1})\ \mathit{with}\ j \quad [\![\mathtt{e}]\!]_\rho = m' \\ \mathtt{p}(m') = \mathit{proc}\ \ m'(\mathtt{y}_1, ..., \mathtt{y}_{n_2})\{\overline{\mathtt{c}}\} \\ \forall_{1\le n\le n_1} \mathtt{v}_n = [\![\mathtt{e}_n]\!]_\rho \quad \forall_{n_1<n\le n_2} \mathtt{v}_n = \mathit{undefined} \\ \mathtt{p} \vdash \langle h, \emptyset[\mathtt{y}_i \mapsto \mathtt{v}_i|_{i=1}^{n_2}], 0, 0\rangle \rightarrow_{m'}^* \langle h', \rho', \_, \mathtt{ret}\rangle \\ \rho'(\mathtt{xret}) = \mathtt{v}\end{array}}{\mathtt{p} \vdash \langle h, \rho, \_, i\rangle \rightarrow_m^* \langle h', \rho[\mathtt{x} \mapsto \mathtt{v}], i, i+1\rangle}$$

$$\textsc{Procedure Call - Error}\quad \frac{\begin{array}{l}\mathtt{p}_m(i) = \mathtt{x} := \mathtt{e}(\mathtt{e}_1, ..., \mathtt{e}_{n_1})\ \mathit{with}\ j \quad [\![\mathtt{e}]\!]_\rho = m' \\ \mathtt{p}(m') = \mathit{proc}\ \ m'(\mathtt{y}_1, ..., \mathtt{y}_{n_2})\{\overline{\mathtt{c}}\} \\ \forall_{1\le n\le n_1} \mathtt{v}_n = [\![\mathtt{e}_n]\!]_\rho \quad \forall_{n_1<n\le n_2} \mathtt{v}_n = \mathit{undefined} \\ \mathtt{p} \vdash \langle h, \emptyset[\mathtt{y}_i \mapsto \mathtt{v}_i|_{i=1}^{n_2}], 0, 0\rangle \rightarrow_{m'}^* \langle h', \rho', \_, \mathtt{err}\rangle \\ \rho'(\mathtt{xerr}) = \mathtt{v}\end{array}}{\mathtt{p} \vdash \langle h, \rho, \_, i\rangle \rightarrow_m^* \langle h', \rho[\mathtt{x} \mapsto \mathtt{v}], i, j\rangle}$$

$$\textsc{Phi-Assignment}\quad \frac{\mathtt{p}_m(j) = \mathtt{x}_1, ..., \mathtt{x}_n := \phi(\mathtt{x}_1^1, ..., \mathtt{x}_1^r; ...; \mathtt{x}_n^1, ..., \mathtt{x}_n^r) \quad i \overset{k}{\mapsto}_m j}{\mathtt{p} \vdash \langle h, \rho, i, j\rangle \rightarrow_m^* \langle h, \rho[\mathtt{x}_t \mapsto \rho(\mathtt{x}_t^k)|_{t=1}^n], j, j+1\rangle}$$

$$\textsc{Transitivity}\quad \frac{\mathtt{p} \vdash \langle h, \rho, i, j\rangle \rightarrow_m^* \langle h', \rho', j, k\rangle \quad \mathtt{p} \vdash \langle h', \rho', k, i'\rangle \rightarrow_m^* \langle h'', \rho'', i', j'\rangle}{\mathtt{p} \vdash \langle h, \rho, i, j\rangle \rightarrow_m^* \langle h'', \rho'', i', j'\rangle}$$

Lemma 3 (Semantics Equivalence)**.** *The following relationship binds the main and the alternative semantics for JSIL.*

$$\mathtt{p} \vdash \langle h, \rho, i, j\rangle \Downarrow_m \langle h', \rho', \mathtt{nm}\langle \mathtt{v}\rangle\rangle \iff \mathtt{p} \vdash \langle h, \rho, i, j\rangle \rightarrow_m^* \langle h', \rho', \_, \mathtt{ret}\rangle \wedge \rho'(\mathtt{xret}) = \mathtt{v}$$
$$\mathtt{p} \vdash \langle h, \rho, i, j\rangle \Downarrow_m \langle h', \rho', \mathtt{er}\langle \mathtt{v}\rangle\rangle \iff \mathtt{p} \vdash \langle h, \rho, i, j\rangle \rightarrow_m^* \langle h', \rho', \_, \mathtt{err}\rangle \wedge \rho'(\mathtt{xerr}) = \mathtt{v}$$

Proof. By induction on $\Downarrow_m$ and $\rightarrow_m^*$. □

B.2.2. The Proof of the Compiler Correctness

Lemma 4 (Correctness of Expression Translation)**.** *For a given JavaScript expression e and JSIL variable $\mathtt{x}$, the translation $C_m(e, \mathtt{x}) = \overline{\mathtt{c}}_{(i,j)}, \overline{\mathtt{e}}$ is correct, meaning that:*

$$\wp, L, v_t \vdash \langle h, e\rangle \Downarrow_m \langle h', w\rangle \iff \exists \rho' \,.\, \overline{\mathtt{c}}_{(i,j)} \vdash \langle h, \rho, \_, i\rangle \rightarrow_m^* \langle h', \rho', \_, j\rangle \wedge \rho'(\mathtt{x}) = w$$
$$\wp, L, v_t \vdash \langle h, e\rangle \Downarrow_m \langle h', \mathit{error}\, w\rangle \iff \exists \rho' \,.\, \overline{\mathtt{c}}_{(i,j)} \vdash \langle h, \rho, \_, i\rangle \rightarrow_m^* \langle h', \rho', i', \mathtt{perr}\rangle \wedge i' \overset{k}{\mapsto} \mathtt{perr} \wedge \rho'(\overline{\mathtt{e}}_k) = w$$

where $\rho \geq \emptyset[\mathtt{xscope} \mapsto L, \mathtt{x}_{\mathit{this}} \mapsto v_t]$.

Lemma 5 (Correctness of Statement Translation)**.** *For a given JavaScript statement s, JSIL variables $\mathtt{x}$, $\mathtt{x}_{pr}$ and break label b_r, the translation $C_m(s, \mathtt{x}, \mathtt{x}_{pr}, b_r) = \overline{\mathtt{c}}_{(i,j)}, \overline{\mathtt{e}}, \overline{\mathtt{r}}, \overline{\mathtt{b}}$ is correct, meaning that:*

$$\wp, L, v_t \vdash \langle h, s\rangle \Downarrow_m \langle h', w\rangle \iff \exists \rho' \,.\, \overline{\mathtt{c}}_{(i,j)} \vdash \langle h, \rho, \_, i\rangle \rightarrow_m^* \langle h', \rho', \_, j\rangle \wedge \rho'(\mathtt{x}) = w$$
$$\wp, L, v_t \vdash \langle h, s\rangle \Downarrow_m \langle h', \mathit{error}\, w\rangle \iff \exists \rho' \,.\, \overline{\mathtt{c}}_{(i,j)} \vdash \langle h, \rho, \_, i\rangle \rightarrow_m^* \langle h', \rho', i', \mathtt{perr}\rangle \wedge i' \overset{k}{\mapsto} \mathtt{perr} \wedge \rho'(\overline{\mathtt{e}}_k) = w$$
$$\wp, L, v_t \vdash \langle h, s\rangle \Downarrow_m \langle h', \mathit{break}\, w\rangle \iff \exists \rho' \,.\, \overline{\mathtt{c}}_{(i,j)} \vdash \langle h, \rho, \_, i\rangle \rightarrow_m^* \langle h', \rho', i', b_r\rangle \wedge i' \overset{k}{\mapsto} b_r \wedge \rho'(\overline{\mathtt{b}}_k) = (w \neq \mathit{empty}\ ?\ w : \rho'(\mathtt{x}_{pr}))$$
$$\wp, L, v_t \vdash \langle h, s\rangle \Downarrow_m \langle h', \mathit{ret}\, w\rangle \iff \exists \rho' \,.\, \overline{\mathtt{c}}_{(i,j)} \vdash \langle h, \rho, \_, i\rangle \rightarrow_m^* \langle h', \rho', i', \mathtt{pret}\rangle \wedge i' \overset{k}{\mapsto} \mathtt{pret} \wedge \rho'(\overline{\mathtt{r}}_k) = w$$

where $\rho \geq \emptyset[\mathtt{xscope} \mapsto L, \mathtt{x}_{\mathit{this}} \mapsto v_t]$.

Theorem B.1 (Compiler Correctness)**.** *The JS-2-JSIL compiler $\mathcal{C}$ is correct, meaning that compiled programs preserve the behaviour of their original versions.*

$$\wp, L, v_t \vdash \langle h, m(\overline{x}, \overline{v})\rangle \Downarrow_m \langle h_f, v\rangle \iff \exists \rho_f \,.\, \overline{\mathcal{C}}(s) \vdash \langle h, \rho, \_, 0\rangle \rightarrow_m^* \langle h_f, \rho_f, \_, \mathtt{ret}\rangle \wedge \rho_f(\mathtt{xret}) = v$$
$$\wp, L, v_t \vdash \langle h, m(\overline{x}, \overline{v})\rangle \Downarrow_m \langle h_f, \mathit{error}\, v\rangle \iff \exists \rho_f \,.\, \overline{\mathcal{C}}(s) \vdash \langle h, \rho, \_, 0\rangle \rightarrow_m^* \langle h_f, \rho_f, \_, \mathtt{err}\rangle \wedge \rho_f(\mathtt{xerr}) = v$$

where $\rho = \emptyset[x_i \mapsto v_i|_{i=1}^n, \mathtt{x}_{sc} \mapsto L, \mathtt{x}_{\mathit{this}} \mapsto v_t]$.

Proof: $\Longrightarrow$. We prove the theorem and the two above lemmas together, by mutual induction on the derivation of expressions, statements, and the full function body. The statements that we are proving are:

$$\wp, L, v_t \vdash \langle h, e\rangle \Downarrow_m \langle h', w\rangle \Rightarrow \exists \rho' \,.\, \overline{\mathtt{c}}_{(i,j)} \vdash \langle h, \rho, \_, i\rangle \rightarrow_m^* \langle h', \rho', \_, j\rangle \wedge \rho'(\mathtt{x}) = w \quad \textbf{(G1)}$$
$$\wp, L, v_t \vdash \langle h, e\rangle \Downarrow_m \langle h', \mathsf{error}\, w\rangle \Rightarrow \exists \rho' \,.\, \overline{\mathtt{c}}_{(i,j)} \vdash \langle h, \rho, \_, i\rangle \rightarrow_m^* \langle h', \rho', i', \mathtt{perr}\rangle \wedge i' \overset{k}{\mapsto} \mathtt{perr} \wedge \rho'(\overline{\mathtt{e}}_k) = w \quad \textbf{(G2)}$$
$$\wp, L, v_t \vdash \langle h, s\rangle \Downarrow_m \langle h', w\rangle \Rightarrow \exists \rho' \,.\, \overline{\mathtt{c}}_{(i,j)} \vdash \langle h, \rho, \_, i\rangle \rightarrow_m^* \langle h', \rho', \_, j\rangle \wedge \rho'(\mathtt{x}) = w \quad \textbf{(G3)}$$
$$\wp, L, v_t \vdash \langle h, s\rangle \Downarrow_m \langle h', \mathsf{error}\, w\rangle \Rightarrow \exists \rho' \,.\, \overline{\mathtt{c}}_{(i,j)} \vdash \langle h, \rho, \_, i\rangle \rightarrow_m^* \langle h', \rho', i', \mathtt{perr}\rangle \wedge i' \overset{k}{\mapsto} \mathtt{perr} \wedge \rho'(\overline{\mathtt{e}}_k) = w \quad \textbf{(G4)}$$
$$\wp, L, v_t \vdash \langle h, s\rangle \Downarrow_m \langle h', \mathsf{break}\, w\rangle \Rightarrow \exists \rho' \,.\, \overline{\mathtt{c}}_{(i,j)} \vdash \langle h, \rho, \_, i\rangle \rightarrow_m^* \langle h', \rho', i', b_r\rangle \wedge i' \overset{k}{\mapsto} b_r \wedge \rho'(\overline{\mathtt{b}}_k) = (w \neq \mathsf{empty}\ ?\ w : \rho'(\mathtt{x}_{pr})) \quad \textbf{(G5)}$$
$$\wp, L, v_t \vdash \langle h, s\rangle \Downarrow_m \langle h', \mathsf{ret}\, w\rangle \Rightarrow \exists \rho' \,.\, \overline{\mathtt{c}}_{(i,j)} \vdash \langle h, \rho, \_, i\rangle \rightarrow_m^* \langle h', \rho', i', \mathtt{pret}\rangle \wedge i' \overset{k}{\mapsto} \mathtt{pret} \wedge \rho'(\overline{\mathtt{r}}_k) = w \quad \textbf{(G6)}$$
$$\wp, L, v_t \vdash \langle h, m(\overline{x}, \overline{v})\rangle \Downarrow_m \langle h_f, w\rangle \Rightarrow \exists \rho_f \,.\, \overline{\mathcal{C}}(s) \vdash \langle h, \rho, \_, 0\rangle \rightarrow_m^* \langle h_f, \rho_f, \_, \mathtt{ret}\rangle \wedge \rho_f(\mathtt{xret}) = w \quad \textbf{(G7)}$$
$$\wp, L, v_t \vdash \langle h, m(\overline{x}, \overline{v})\rangle \Downarrow_m \langle h_f, \mathsf{error}\, w\rangle \Rightarrow \exists \rho_f \,.\, \overline{\mathcal{C}}(s) \vdash \langle h, \rho, \_, 0\rangle \rightarrow_m^* \langle h_f, \rho_f, \_, \mathtt{err}\rangle \wedge \rho_f(\mathtt{xerr}) = w \quad \textbf{(G8)}$$

where, for the last two cases, $\rho = \emptyset[x_i \mapsto v_i|_{i=1}^n, \mathtt{x}_{sc} \mapsto L, \mathtt{x}_{\mathsf{this}} \mapsto v_t]$, and for the other cases,

$$\text{(A)}\ \frac{\dots}{\vdash \langle h, e_1\rangle \Downarrow_m \langle h_1, w_1\rangle} \qquad \text{(B)}\ \frac{\dots}{\vdash \langle h_1, e_2\rangle \Downarrow_m \langle h_2, w_2\rangle} \qquad \text{(C)}\ \frac{\dots}{\vdash \langle h_2, \mathcal{I}_{gv}(w_2)\rangle \Downarrow_m \langle h_3, v\rangle}$$

$$\frac{\vdash \langle h_2, \gamma(w_2)\rangle \Downarrow_m \langle h_3, v\rangle}{\vdash \langle h_1, e_2\rangle \Downarrow_m^{\gamma} \langle h_3, v\rangle} \qquad (*)$$

$$\frac{\vdash \langle h, e_1\rangle \Downarrow_m \langle h_1, w_1\rangle \qquad \vdash \langle h_1, w_1 =_1 e_2\rangle \Downarrow_m \langle h_4, v\rangle}{\vdash \langle h, e_1 = e_2\rangle \Downarrow_m \langle h_4, v\rangle}$$

where $(*)$ is

$$\text{(D)}\ \frac{w_1 = v_1 \vee w_1 = l.{}_{\circ}p \ \vee\ (w_1 = l.{}_{\vee}p \wedge p \notin \{\mathsf{eval}, \mathsf{arguments}\})}{\vdash \langle h_3, w_1 =_2 v\rangle \Downarrow_m \langle h_4, v\rangle} \qquad \frac{\dots}{\vdash \langle h_3, \mathcal{I}_{pv}(w_1, v)\rangle \Downarrow_m \langle h_4, v\rangle} \qquad \text{(E)}\ \frac{}{\vdash \langle h_4, w_1 =_3 v\rangle \Downarrow_m \langle h_4, v\rangle}$$

Figure B.7.: JavaScript operational semantics proof tree of an assignment for the normal execution.

$\rho \geq \emptyset[\texttt{xscope} \mapsto L, \mathtt{x}_{\mathsf{this}} \mapsto v_t]$.

We select the representative cases for the expressions and statements. The proof for other cases follow the same pattern. For JavaScript expressions we prove assignment $e_1 = e_2$ (**G1 - G2**), function call $e_0(\overline{e})$ (**G1**) and variable x (**G1**). For JavaScript statements we prove sequence $s_1; s_2$ (**G3-G6**). Finally we prove normal and error cases for the full function body $m(\overline{x}, \overline{v})$ (**G7-G8**).

[Expressions: Assignment Expression: $e \equiv e_1 = e_2$]
Let

$$C_m(e_1, \mathtt{x}_1) = \overline{\mathtt{c}}_{1(l_1,l_2)}, \overline{\mathtt{e}}'$$
$$C_m(e_2, \mathtt{x}_2) = \overline{\mathtt{c}}_{2(l_2,l_3)}, \overline{\mathtt{e}}''$$
$$C_m(e_1 = e_2, \mathtt{x}) = \overline{\mathtt{c}}_{a(l_1,l_6)}, \overline{\mathtt{e}}$$

where $\overline{\mathtt{e}} = \overline{\mathtt{e}}' :: \overline{\mathtt{e}}'' :: [\mathtt{x}, \mathtt{x}', \mathtt{x}'']$
We need to prove **G1**, **G2** where $(i, j) = (l_1, l_6)$. The translated code is

$$\overline{\mathtt{c}}_a = \begin{cases} l_1: & \overline{\mathtt{c}}_1 \\ l_2: & \overline{\mathtt{c}}_2 \\ l_3: & \mathtt{x} := \mathtt{getValue}(\mathtt{x}_2)\ \mathit{with}\ \mathtt{perr} \\ l_4: & \mathtt{x}' := \mathtt{checkAssignment}(\mathtt{x}_1)\ \mathit{with}\ \mathtt{perr} \\ l_5: & \mathtt{x}'' := \mathtt{putValue}(\mathtt{x}_1, \mathtt{x})\ \mathit{with}\ \mathtt{perr} \end{cases}$$

For the moment, let us focus on the success case and prove

$$\wp, L, v_t \vdash \langle h, e_1 = e_2\rangle \Downarrow_m \langle h', v\rangle \Rightarrow \exists \rho' \,.\, \overline{\mathtt{c}}_{a(l_1,l_6)} \vdash \langle h, \rho, \_, l_1\rangle \rightarrow_m^* \langle h', \rho', \_, l_6\rangle \wedge \rho'(\mathtt{x}) = v.$$

From the proof tree in Fig. B.7 we can apply the induction hypothesis (1) to A and B, Lemma 9 (`getValue`) to C, Lemma 11 (`checkAssignment`) to D, and Lemma 10 (`putValue`) to E to obtain the

following:

$$\begin{aligned}
\overline{\mathtt{c}}_{1(l_1,l_2)} \vdash \langle h, \rho, _, l_1\rangle \rightarrow_m^* \langle h_1, \rho_1, _, l_2\rangle \wedge \rho_1(\mathtt{x}_1) = w_1 \\
\overline{\mathtt{c}}_{2(l_2,l_3)} \vdash \langle h, \rho_1, _, l_2\rangle \rightarrow_m^* \langle h_1, \rho_2, _, l_3\rangle \wedge \rho_2(\mathtt{x}_2) = w_2 \\
\mathtt{x} := \mathtt{getValue}(\mathtt{x}_2) \;\; \textit{with}\; \mathtt{perr} \vdash \langle h_2, \rho_2, _, l_3\rangle \rightarrow_m^* \langle h_3, \rho_3, l_3, l_4\rangle \wedge \rho_3(\mathtt{x}) = v \\
\mathtt{x}' := \mathtt{checkAssignment}(\mathtt{x}_1) \;\; \textit{with}\; \mathtt{perr} \vdash \langle h_3, \rho_3, _, l_4\rangle \rightarrow_m^* \langle h_3, \rho_4, l_4, l_5\rangle \\
\mathtt{x}'' := \mathtt{putValue}(\mathtt{x}_1, \mathtt{x}) \;\; \textit{with}\; \mathtt{perr} \vdash \langle h_3, \rho_4, l_4, l_5\rangle \rightarrow_m^* \langle h_4, \rho_5, l_5, l_6\rangle \wedge \rho_5(\mathtt{x}'') = \mathsf{empty}
\end{aligned}$$

Combining the above transitions using Lemma 6 (concatenation), we obtain that

$$\overline{\mathtt{c}}_{a(l_1,l_6)} \vdash \langle h, \rho, \_, l_1\rangle \rightarrow_m^* \langle h_4, \rho_5, \_, l_6\rangle \wedge \rho_5(\mathtt{x}) = v.$$

Next, we prove the error case.

$$\wp, L, v_t \vdash \langle h, e_1 = e_2\rangle \Downarrow_m \langle h', \mathsf{error}\, w\rangle \Rightarrow \exists \rho' \,.\, \overline{\mathtt{c}}_{a(l_1,l_6)} \vdash \langle h, \rho, \_, l_1\rangle \rightarrow_m^* \langle h', \rho', i, \mathtt{perr}\rangle \wedge i \overset{k}{\mapsto} \mathtt{perr} \wedge \rho'(\overline{\mathtt{e}}_k) = w$$

Fig. B.8 and B.9 show the five proof trees that produce error for assignment statement. Consider the first proof tree, where e_1 throws an exception. Using induction hypothesis (2) on A we get

$$\overline{\mathtt{c}}_{1(l_1,l_2)} \vdash \langle h, \rho, \_, l_1\rangle \rightarrow_m^* \langle h', \rho', i, \mathtt{perr}\rangle \wedge i \overset{k}{\mapsto} \mathtt{perr} \wedge \rho'(\overline{\mathtt{e}}'_k) = w$$

We have that $\overline{\mathtt{e}}_k = \overline{\mathtt{e}}'_k$, since $\overline{\mathtt{c}}_1$ is at the start of $\overline{\mathtt{c}}_a$. Hence $\rho'(\overline{\mathtt{e}}_k) = w$. From Lemma 7 (concatenation) we immediately get **G2**.

For the second proof tree, where e_2 throws an exception, we apply induction hypothesis (1, 2) to B and C to obtain

$$\begin{aligned}
\overline{\mathtt{c}}_{1(l_1,l_2)} \vdash \langle h, \rho, _, l_1\rangle \rightarrow_m^* \langle h_1, \rho_1, _, l_2\rangle \wedge \rho_1(\mathtt{x}_1) = w_1 \\
\overline{\mathtt{c}}_{2(l_2,l_3)} \vdash \langle h_1, \rho_1, _, l_2\rangle \rightarrow_m^* \langle h', \rho', i, \mathtt{perr}\rangle \wedge i \overset{k'}{\mapsto} \mathtt{perr} \wedge \rho'(\overline{\mathtt{e}}''_{k'}) = w
\end{aligned}$$

Since $\overline{\mathtt{c}}_1$ comes before $\overline{\mathtt{c}}_2$ in $\overline{\mathtt{c}}_a$, we get that $\overline{\mathtt{e}}_k = \overline{\mathtt{e}}''_{k'}$ where $k = k' + d$ and d is the length of $\overline{\mathtt{e}}'$. Again, using Lemma 7 (concatenation) we get **G2**.

For the third proof tree, where `GetValue` throws an exception, we apply induction hypothesis (1) to D, E and Lemma 9 (`getValue`) to F, and obtain

$$\begin{aligned}
\overline{\mathtt{c}}_{1(l_1,l_2)} \vdash \langle h, \rho, _, l_1\rangle \rightarrow_m^* \langle h_1, \rho_1, _, l_2\rangle \wedge \rho_1(\mathtt{x}_1) = w_1 \\
\overline{\mathtt{c}}_{2(l_2,l_3)} \vdash \langle h_1, \rho_1, _, l_2\rangle \rightarrow_m^* \langle h_2, \rho_2, _, l_3\rangle \wedge \rho_2(\mathtt{x}_2) = w_2 \\
\mathtt{x} := \mathtt{getValue}(\mathtt{x}_2) \;\; \textit{with}\; \mathtt{perr} \vdash \langle h_2, \rho_2, _, l_3\rangle \rightarrow_m^* \langle h', \rho', l_3, \mathtt{perr}\rangle \wedge \rho'(\mathtt{x}) = w
\end{aligned}$$

Since $\overline{\mathtt{c}}_1$, $\overline{\mathtt{c}}_2$ comes before `GetValue` in $\overline{\mathtt{c}}_a$, we get that $\overline{\mathtt{e}}_k = \mathtt{x}$ where $k = d' + d'' + 1$ and d' is the length of $\overline{\mathtt{e}}'$, and d'' is the length of $\overline{\mathtt{e}}''$. Again, using Lemma 7 (concatenation) we get **G2**:

$$\overline{\mathtt{c}}_{a(l_1,l_6)} \vdash \langle h, \rho, \_, l_1\rangle \rightarrow_m^* \langle h', \rho', l_3, \mathtt{perr}\rangle \wedge l_3 \overset{k}{\mapsto} \mathtt{perr} \wedge \rho'(\overline{\mathtt{e}}_k) = w$$

The proof trees in Fig. B.9 correspond to the last two paths where the execution of the assignment

$$\text{(A)}\ \frac{\dfrac{\cdots}{\vdash \langle h, e_1\rangle \Downarrow_m \langle h', \mathsf{error}\, w\rangle} \qquad \dfrac{}{\vdash \langle h', \mathsf{error}\, w =_1 e_2\rangle \Downarrow_m \langle h', \mathsf{error}\, w\rangle}}{\vdash \langle h, e_1 = e_2\rangle \Downarrow_m \langle h', \mathsf{error}\, w\rangle}$$

$$\text{(B)}\ \frac{\dfrac{\cdots}{\vdash \langle h, e_1\rangle \Downarrow_m \langle h_1, w_1\rangle} \qquad \dfrac{\dfrac{\text{(C)}\ \dfrac{\dfrac{\cdots}{\vdash \langle h_1, e_2\rangle \Downarrow_m \langle h', \mathsf{error}\, w\rangle} \qquad \dfrac{}{\vdash \langle h', \gamma(\mathsf{error}\, w)\rangle \Downarrow_m \langle h', \mathsf{error}\, w\rangle}}{\vdash \langle h_1, e_2\rangle \Downarrow^{\gamma}_m \langle h', \mathsf{error}\, w\rangle}}{\vdash \langle h_1, w_1 =_1 e_2\rangle \Downarrow_m \langle h', \mathsf{error}\, w\rangle}\ (*)}{\vdash \langle h, e_1 = e_2\rangle \Downarrow_m \langle h', \mathsf{error}\, w\rangle}$$

$$\text{(D)}\ \frac{\dfrac{\cdots}{\vdash \langle h, e_1\rangle \Downarrow_m \langle h_1, w_1\rangle} \qquad \dfrac{\dfrac{\text{(E)}\ \dfrac{\dfrac{\cdots}{\vdash \langle h_1, e_2\rangle \Downarrow_m \langle h_2, w_2\rangle} \qquad \dfrac{\text{(F)}\ \dfrac{\cdots}{\vdash \langle h_2, \mathcal{I}_{gv}(w_2)\rangle \Downarrow_m \langle h', \mathsf{error}\, w\rangle}}{\vdash \langle h_2, \gamma(w_2)\rangle \Downarrow_m \langle h', \mathsf{error}\, w\rangle}}{\vdash \langle h_1, e_2\rangle \Downarrow^{\gamma}_m \langle h', \mathsf{error}\, w\rangle}}{\vdash \langle h_1, w_1 =_1 e_2\rangle \Downarrow_m \langle h', \mathsf{error}\, w\rangle}\ (*)}{\vdash \langle h, e_1 = e_2\rangle \Downarrow_m \langle h', \mathsf{error}\, w\rangle}$$

where (*) is

$$\frac{}{\vdash \langle h', w_1 =_2 \mathsf{error}\, w\rangle \Downarrow_m \langle h', \mathsf{error}\, w\rangle}$$

Figure B.8.: JavaScript operational semantics proof trees of an assignment for the error case. Part I

terminates with an error. First, we apply induction hypothesis (1) to G and H, and Lemma 9 (`getValue`) to I, for both proof trees, as they share the beginning part, to obtain the following :

$$\begin{aligned}
\overline{\mathtt{c}}_{1(l_1,l_2)} \vdash \langle h, \rho, _, l_1\rangle &\rightarrow^*_m \langle h_1, \rho_1, _, l_2\rangle \wedge \rho_1(\mathtt{x}_1) = w_1\\
\overline{\mathtt{c}}_{2(l_2,l_3)} \vdash \langle h_1, \rho_1, _, l_2\rangle &\rightarrow^*_m \langle h_2, \rho_2, _, l_3\rangle \wedge \rho_2(\mathtt{x}_2) = w_2\\
\mathtt{x} := \mathtt{getValue}(\mathtt{x}_2)\ \ \mathit{with}\ \mathtt{perr} \vdash \langle h_2, \rho_2, _, l_3\rangle &\rightarrow^*_m \langle h_3, \rho_3, _, l_4\rangle \wedge \rho_3(\mathtt{x}) = v
\end{aligned}$$

If we, then, take the first proof tree, we can apply Lemma 11 (`checkAssignment`) to J to obtain:

$$\mathtt{x}' := \mathtt{checkAssignment}(\mathtt{x}_1)\ \ \mathit{with}\ \mathtt{perr} \vdash \langle h_3, \rho_3, \_, l_4\rangle \rightarrow^*_m \langle h', \rho', l_4, \mathtt{perr}\rangle \wedge \rho'(\mathtt{x}') = w$$

Since $\overline{\mathtt{c}}_1$, $\overline{\mathtt{c}}_2$ and `GetValue` come before `CheckAssignment` in $\overline{\mathtt{c}}_a$, we get that $\overline{\mathtt{e}}_k = \mathtt{x}'$ where $k = d'+d''+2$ and d' is the length of $\overline{\mathtt{e}}'$, and d'' is the length of $\overline{\mathtt{e}}''$. Again, using Lemma 7 (concatenation) we get **G2**.

Similarly for the last proof tree, where `PutValue` throws an exception, we can apply Lemma 11 (`checkAssignment`) to K and Lemma 10 (`putValue`) to L to obtain:

$$\begin{aligned}
\mathtt{x}' := \mathtt{checkAssignment}(\mathtt{x}_1)\ \ \mathit{with}\ \mathtt{perr} \vdash \langle h_3, \rho_3, _, l_4\rangle &\rightarrow^*_m \langle h_3, \rho_4, l_4, l_5\rangle\\
\mathtt{x}'' := \mathtt{putValue}(\mathtt{x}_1, \mathtt{x})\ \ \mathit{with}\ \mathtt{perr} \vdash \langle h_3, \rho_4, l_4, l_5\rangle &\rightarrow^*_m \langle h', \rho', l_5, \mathtt{perr}\rangle \wedge \rho_5(\mathtt{x}'') = w
\end{aligned}$$

Since $\overline{\mathtt{c}}_1$, $\overline{\mathtt{c}}_2$, `GetValue` and `CheckAssignment` come before `PutValue` in $\overline{\mathtt{c}}_a$, we get that $\overline{\mathtt{e}}_k = \mathtt{x}'$ where $k = d' + d'' + 3$ and d' is the length of $\overline{\mathtt{e}}'$, and d'' is the length of $\overline{\mathtt{e}}''$. Again, using Lemma 7

$$\dfrac{\text{(G)}\ \dfrac{\ldots}{\vdash \langle h, e_1\rangle \Downarrow_m \langle h_1, w_1\rangle} \qquad \dfrac{\text{(H)}\ \dfrac{\ldots}{\vdash \langle h_1, e_2\rangle \Downarrow_m \langle h_2, w_2\rangle} \qquad \text{(I)}\ \dfrac{\dfrac{\ldots}{\vdash \langle h_2, \mathcal{I}_{gv}(w_2)\rangle \Downarrow_m \langle h_3, v\rangle}}{\vdash \langle h_2, \gamma(w_2)\rangle \Downarrow_m \langle h_3, v\rangle}}{\vdash \langle h_1, e_2\rangle \Downarrow_m^{\gamma} \langle h_3, v\rangle}}{\vdash \langle h_1, w_1 =_1 e_2\rangle \Downarrow_m \langle h', \mathsf{error}\, w\rangle}\ (1-2)}{\vdash \langle h, e_1 = e_2\rangle \Downarrow_m \langle h', \mathsf{error}\, w\rangle}$$

where (1) is

$$\dfrac{\begin{array}{l}\text{(J)}\\ w_1 = l.{}_{\mathsf{v}}p \qquad p \in \{\mathsf{eval}, \mathsf{arguments}\}) \end{array} \qquad h' = h \uplus \mathsf{err}(w, l_{sep})}{\vdash \langle h_3, w_1 =_2 v\rangle \Downarrow_m \langle h', \mathsf{error}\, w\rangle}$$

where (2) is

$$\dfrac{\begin{array}{l}\text{(K)}\\ w_1 = v_1 \vee w_1 = l.{}_{\mathsf{o}}p \ \vee \\ (w_1 = l.{}_{\mathsf{v}}p \wedge p \notin \{\mathsf{eval}, \mathsf{arguments}\}) \end{array} \qquad \dfrac{\ldots}{\vdash \langle h_3, \mathcal{I}_{pv}(w_1, v)\rangle \Downarrow_m \langle h', \mathsf{error}\, w\rangle}\ \text{(L)}\ \dfrac{}{\vdash \langle h', w_1 =_3 v\rangle \Downarrow_m \langle h', \mathsf{error}\, w\rangle}}{\vdash \langle h_3, w_1 =_2 v\rangle \Downarrow_m \langle h', \mathsf{error}\, w\rangle}$$

Figure B.9.: JavaScript operational semantics proof trees of an assignment for the error case. Part II

(concatenation) we get **G2**.

[Expressions: Function call: $e \equiv e_0(\overline{e})$]
Let

$$\begin{aligned} C_m(e, \mathtt{x}) &= \overline{\mathsf{c}}_{a(l_1,l_5)}, \overline{\mathsf{e}}^{a} \\ C_m(e_0, \mathtt{x}_e) &= \overline{\mathsf{c}}_{e(l_1,l_2)}, \overline{\mathsf{e}}^{0} \\ C_m(e_i, \mathtt{x}_i) &= \overline{\mathsf{c}}_{i(k_{2i-1},k_{2i})}, \overline{\mathsf{e}}^{i}|_{i=1}^{n} \\ \overline{\mathsf{c}}'_i &= \mathtt{x}'_i := \mathtt{getValue}(\mathtt{x}_i)\ \ \mathit{with}\ \mathtt{perr}|_{i=1}^{n} \end{aligned}$$

We need to prove **G1** - **G2** where $(i, j) = (l_1, l_6)$. The translated code is shown in Fig. B.10. For the moment, let us focus on the success case and prove

$$\wp, L, v_t \vdash \langle h, e_0(\overline{e})\rangle \Downarrow_m \langle h', v_o\rangle \Rightarrow \exists \rho' \,.\, \overline{\mathsf{c}}_{a(l_1,l_6)} \vdash \langle h, \rho, \_, l_1\rangle \rightarrow_m^* \langle h', \rho', \_, l_6\rangle \wedge \rho'(\mathtt{x}) = v_o$$

From the proof tree in Fig. B.11 we can apply the induction hypothesis (1) to A, Lemma 9 (`getValue`) to C, the induction hypothesis (1) to B_i, Lemma 12 (`isCallable`) to $\mathcal{P}_c(h_3, v_1)$, Lemma 13 (`selectThis`) to $v_t = \mathsf{SelectThis}(w_1)$, and JSIL operational semantics for property access to $m' =$

$$
\overline{\mathtt{c}}_a = \begin{cases}
l_1: & \overline{\mathtt{c}}_e \\
l_2: & \mathtt{x}'_e := \mathtt{getValue}(\mathtt{x}_e)\ \mathit{with}\ \mathtt{perr} \\
k_{2i-1}: & \{\overline{\mathtt{c}}_i :: \overline{\mathtt{c}}'_i\}|_{i=1}^{n} \\
k_{2n+1}: & \mathsf{goto}\ [\mathsf{typeOf}\,(\mathtt{x}'_e)\ \mathtt{!=}\ \mathsf{Obj}]\ t_1,\ e_1 \\
t_1: & x_{err} := \mathtt{typeError}() \\
 & \mathsf{goto}\ \mathtt{perr} \\
e_1: & \mathtt{x}_c := \mathtt{isCallable}(\mathtt{x}'_e) \\
 & \mathsf{goto}\ [\mathtt{x}_c]\ n_1,\ t_1 \\
n_1: & \mathsf{goto}\ [\mathsf{typeOf}\,(\mathtt{x}_e) = \mathsf{List} \wedge (\mathsf{nth}\,(\mathtt{x}_e, 0) = \mathtt{o}]\ t_2,\ e_2 \\
t_2: & \mathtt{x}_{t1} := \mathsf{nth}\,(\mathtt{x}_e, 1) \\
 & \mathsf{goto}\ n_2 \\
e_2: & \mathtt{x}_{t2} := \mathsf{undefined} \\
n_2: & \mathtt{x}_t := \phi(\mathtt{x}_{t1}, \mathtt{x}_{t2}) \\
l_3: & \mathtt{x}_{sc'} := [\mathtt{x}'_e, @\mathit{scope}] \\
l_4: & \mathtt{x}_{m'} := [\mathtt{x}'_e, @\mathit{code}] \\
l_5: & \mathtt{x} := \mathtt{x}_{m'}(\mathtt{x}_{sc'}, \mathtt{x}_t, \mathtt{x}'_1, ..., \mathtt{x}'_n)\ \text{with}\ \mathtt{perr}
\end{cases}
$$

Figure B.10.: Translated code of a function call expression $e_0(\overline{e})$.

$h_3(v_1, @\mathit{code})$ and $L' = h_3(v_1, @\mathit{scope})$ to obtain the following:

$$
\begin{aligned}
\overline{\mathtt{c}}_{e(l_1,l_2)} \vdash \langle h, \rho, \_, l_1\rangle \rightarrow_m^* \langle h_1, \rho_1, \_, l_2\rangle \wedge \rho_1(\mathtt{x}_e) = w_1 \\
\mathtt{x}'_e := \mathtt{getValue}(\mathtt{x}_e)\ \mathit{with}\ \mathtt{perr} \vdash \langle h_1, \rho_1, \_, l_2\rangle \rightarrow_m^* \langle h_2, \rho_2, \_, k_1\rangle \wedge \rho_2(\mathtt{x}'_e) = v_0 \\
\overline{\mathtt{c}}_{i(k_{2i-1},k_{2i})} \vdash \langle h'_{2i-1}, \rho'_{2i-1}, \_, k_{2i-1}\rangle \rightarrow_m^* \langle h'_{2i}, \rho'_{2i}, \_, k_{2i}\rangle \wedge \rho'_{2i}(\mathtt{x}_i) = \overline{w}_i \\
\mathtt{x}'_i := \mathtt{getValue}(\mathtt{x}_i)\ \mathit{with}\ \mathtt{perr} \vdash \langle h'_{2i}, \rho'_{2i}, \_, k_{2i}\rangle \rightarrow_m^* \langle h'_{2i+1}, \rho'_{2i+1}, \_, k_{2i+1}\rangle \wedge \rho'_{2i+1}(\mathtt{x}'_i) = \overline{v}_i \\
\overline{c_{ic}}_{(k_{2n+1},n_1)} \vdash \langle h_{2n+1}, \rho_{2n+1}, \_, k_{2n+1}\rangle \rightarrow_m^* \langle h_3, \rho_3, \_, n_1\rangle \wedge \rho_3(\mathtt{x}_c) = \mathsf{true} \\
\overline{c_{st}}_{(n_1,l_3)} \vdash \langle h_3, \rho_3, \_, n_1\rangle \rightarrow_m^* \langle h_3, \rho_4, \_, l_3\rangle \wedge \rho_4(\mathtt{x}_t) = v_t \\
\mathtt{x}_{sc'} := [\mathtt{x}'_e, @\mathit{scope}] \vdash \langle h_3, \rho_4, \_, l_3\rangle \rightarrow_m^* \langle h_3, \rho_5, \_, l_4\rangle \wedge \rho_5(\mathtt{x}_{sc'}) = L' \\
\mathtt{x}_{m'} := [\mathtt{x}'_e, @\mathit{code}] \vdash \langle h_3, \rho_5, \_, l_4\rangle \rightarrow_m^* \langle h_3, \rho_6, \_, l_5\rangle \wedge \rho_6(\mathtt{x}_{m'}) = m'
\end{aligned}
$$

To obtain

$$
\mathtt{x} := \mathtt{x}_{m'}(\mathtt{x}_{sc'}, \mathtt{x}_t, \mathtt{x}'_1, ..., \mathtt{x}'_n)\ \mathit{with}\ \mathtt{perr} \vdash \langle h_4, \rho_6, \_, l_5\rangle \rightarrow_m^* \langle h', \rho', \_, l_6\rangle \wedge \rho'(\mathtt{x}) = v_o
$$

we use the induction hypothesis (7) to M and the Procedure normal rule of JSIL semantics. Combining all the above transitions using Lemma 6 (concatenation), we obtain that

$$
\overline{\mathtt{c}}_{a(l_1,l_6)} \vdash \langle h, \rho, \_, l_1\rangle \rightarrow_m^* \langle h', \rho', \_, l_6\rangle \wedge \rho'(\mathtt{x}) = v_o.
$$

[Expressions: Variable: $e \equiv x$]
Let $C_m(x, \mathtt{x}) = \overline{\mathtt{c}}_{a(l_1,l_3)}, \overline{\mathtt{e}}^a$. We need to prove **G1** where $(i, j) = (l_1, l_3)$ and $w = v.\mathtt{x}$. We consider two cases: $\psi_m(x) = n$ and $\psi_m(x) = \bot$.

$$\text{(A)}\ \frac{\dots}{\vdash \langle h, e\rangle \Downarrow_m \langle h_1, w_1\rangle} \quad \text{(C)}\ \frac{\dots}{\vdash \langle h_1, \mathcal{I}_{gv}(w_1)\rangle \Downarrow_m \langle h_2, v_0\rangle} \quad \text{(B}_i\text{)}\ \frac{\dots}{\vdash \langle h_2, \mathsf{iterate}\{\overline{e}\}\rangle \Downarrow_m^{\gamma} \langle h_3, \overline{v}\rangle} \quad (*)$$
$$\frac{\vdash \langle h_2, (w_1, v_0)(\overline{e})_2\rangle \Downarrow_m \langle h', v_o\rangle}{\vdash \langle h_1, w_1(\overline{e})_1\rangle \Downarrow_m \langle h', v_o\rangle}$$
$$\vdash \langle h, e_0(\overline{e})\rangle \Downarrow_m \langle h', v_o\rangle$$

where $(*)$ is

$$\frac{\begin{array}{l}\mathcal{P}_c(h_3, v_0)\\ v_t = \mathsf{SelectThis}(w_1)\\ m' = h_3(v_0, @code)\\ L' = h_3(v_0, @scope)\\ \wp(m') = \lambda x_1, ..., x_{n_2}.s\\ \forall_{1\le n\le n_1} v'_n = v_n\\ \forall_{n_1<n\le n_2} v'_n = \mathsf{undefined}\end{array} \qquad \frac{\dots}{\wp, L', v_t \vdash \langle h_3, m'(\overline{x}, \overline{v'})\rangle \Downarrow_{m'} \langle h', v_o\rangle}\ \text{(M)}}{\wp, \_, \_ \vdash \langle h_3, (w_1, v_0)(v_1, ...v_{n_1})_3\rangle \Downarrow_m \langle h', v_o\rangle}$$

Figure B.11.: JavaScript operational semantics proof tree of a function call for the normal execution.

$$\text{(A)}\ \frac{\dots}{\vdash \langle h, \mathcal{I}_\sigma(x)\rangle \Downarrow_m \langle h', v\rangle} \qquad \frac{}{\vdash \langle h', id(x, v)\rangle \Downarrow_m \langle h', v.\mathtt{v}x\rangle}$$
$$\vdash \langle h, x\rangle \Downarrow_m \langle h', v.\mathtt{v}x\rangle$$

Figure B.12.: JavaScript operational semantics proof tree of a variable.

When $\psi_m(x) = n$, the translated code is

$$\overline{\mathtt{c}}_a = \begin{cases} l_1: & \mathtt{x}' := \mathsf{nth}\,(\mathtt{xscope}, n)\\ l_2: & \mathtt{x} := [\text{“v”}, \mathtt{x}', x]\end{cases}$$

When $\psi_m(x) = \bot$, the translated code is

$$\overline{\mathtt{c}}_a = \begin{cases} l_1: & \mathtt{x}_h := \mathtt{hasProperty}(l_g, x)\ \ with\ \mathtt{perr}\\ & \mathsf{goto}\ [\mathtt{x}_h]\ t,\ e\\ t: & \mathtt{x}'_1 := l_g\\ & \mathsf{goto}\ n\\ & \mathtt{x}'_2 := \mathsf{undefined}\\ n: & \mathtt{x}' := \phi(\mathtt{x}'_1, \mathtt{x}'_2)\\ l_2: & \mathtt{x} := [\text{“v”}, \mathtt{x}', x]\end{cases}$$

From the proof tree in Fig. B.12 and Lemma 15 (variable) we obtain the following:

$$\overline{\mathtt{c}}_{a(l_1,l_2)} \vdash \langle h, \rho, \_, l_1\rangle \to_m^* \langle h', \rho_1, \_, l_2\rangle \wedge \rho_1(\mathtt{x}') = v$$
$$\mathtt{x} := [\text{“v”}, \mathtt{x}', x] \vdash \langle h', \rho_1, \_, l_2\rangle \to_m^* \langle h', \rho', l_2, l_3\rangle \wedge \rho'(\mathtt{x}) = v.\mathtt{v}x$$

$$\text{(A)}\ \frac{\dfrac{\dots}{\wp, L, v_t \vdash \langle h, s_1\rangle \Downarrow_m \langle h_1, w\rangle} \quad \text{(B)}\ \dfrac{\dfrac{\dots}{\vdash \langle h_1, s_2\rangle \Downarrow_m \langle h', \mathsf{empty}\rangle} \quad \dfrac{}{\vdash \langle h', \mathsf{seq}_2(w, \mathsf{empty})\rangle \Downarrow_m \langle h', w\rangle}}{\vdash \langle h_1, \mathsf{seq}_1(w, s_2)\rangle \Downarrow_m \langle h', w\rangle}}{\vdash \langle h, s_1; s_2\rangle \Downarrow_m \langle h', w\rangle}$$

$$\text{(C)}\ \frac{\dfrac{\dots}{\vdash \langle h, s_1\rangle \Downarrow_m \langle h_1, w_1\rangle} \quad \text{(D)}\ \dfrac{\dfrac{\dots}{\vdash \langle h_1, s_2\rangle \Downarrow_m \langle h', w\rangle} \quad \dfrac{w \neq \mathsf{empty}}{\vdash \langle h', \mathsf{seq}_2(w_1, w)\rangle \Downarrow_m \langle h', w\rangle}}{\vdash \langle h_1, \mathsf{seq}_1(w_1, s_2)\rangle \Downarrow_m \langle h', w\rangle}}{\vdash \langle h, s_1; s_2\rangle \Downarrow_m \langle h', w\rangle}$$

Figure B.13.: JavaScript operational semantics proof trees of a sequence for the normal execution.

Combining the above transitions using Lemma 6 (concatenation), we obtain that

$$\overline{\mathtt{c}}_{a(l_1,l_3)} \vdash \langle h, \rho, \_, l_1\rangle \rightarrow_m^* \langle h', \rho', l_2, l_3\rangle \wedge \rho'(\mathtt{x}) = v.\mathtt{v}x.$$

[Statements: Sequence: $s \equiv s_1; s_2$]
Let

$$C_m(s_1, \mathtt{x}_1, \mathtt{x}_{pr}, b_r) = \overline{\mathtt{c}}_{1(l_1,l_2)}, \overline{\mathtt{e}}', \overline{\mathtt{r}}', \overline{\mathtt{b}}'$$
$$C_m(s_2, \mathtt{x}_2, \mathtt{x}_1, b_r) = \overline{\mathtt{c}}_{2(l_2,l_3)}, \overline{\mathtt{e}}'', \overline{\mathtt{r}}'', \overline{\mathtt{b}}''$$
$$C_m(s_1; s_2, \mathtt{x}, \mathtt{x}_{pr}, b_r) = \overline{\mathtt{c}}_{a(l_1,l_4)}, \overline{\mathtt{e}}, \overline{\mathtt{r}}, \overline{\mathtt{b}}$$

where $\overline{\mathtt{e}} = \overline{\mathtt{e}}' :: \overline{\mathtt{e}}''$, $\overline{\mathtt{r}} = \overline{\mathtt{r}}' :: \overline{\mathtt{r}}''$, and $\overline{\mathtt{b}} = \overline{\mathtt{b}}' :: \overline{\mathtt{b}}''$. We need to prove **G3**-**G6** where $(i, j) = (l_1, l_4)$. The translated code is

$$\overline{\mathtt{c}}_a = \begin{cases} l_1: & \overline{\mathtt{c}}_1 \\ l_2: & \overline{\mathtt{c}}_2 \\ l_3: & \mathsf{goto}\ [\mathtt{x}_2 = \mathsf{empty}]\ t,\ e \\ e: & \mathsf{skip} \\ t: & \mathtt{x} := \phi(\mathtt{x}_1, \mathtt{x}_2) \end{cases}$$

For the moment, let us focus on the success case **G3** and prove

$$\wp, L, v_t \vdash \langle h, s\rangle \Downarrow_m \langle h', w\rangle \Rightarrow \exists \rho' \,.\, \overline{\mathtt{c}}_{(l_1,l_4)} \vdash \langle h, \rho, \_, l_1\rangle \rightarrow_m^* \langle h', \rho', \_, l_4\rangle \wedge \rho'(\mathtt{x}) = w$$

The two proof trees in Fig. B.13 correspond to the two paths where the execution of the second statement returns empty or not. If we take the first tree, we can apply the induction hypothesis (3) to A and B to obtain the following:

$$\overline{\mathtt{c}}_{1(l_1,l_2)} \vdash \langle h, \rho, \_, l_1\rangle \rightarrow_m^* \langle h_1, \rho_1, \_, l_2\rangle \wedge \rho_1(\mathtt{x}_1) = w$$
$$\overline{\mathtt{c}}_{2(l_2,l_3)} \vdash \langle h_1, \rho, \_, l_2\rangle \rightarrow_m^* \langle h', \rho_2, \_, l_3\rangle \wedge \rho_2(\mathtt{x}_2) = \mathsf{empty}$$

$$\text{(A)}\ \frac{\dfrac{\dots}{\vdash \langle h, s_1\rangle \Downarrow_m \langle h_1, \mathsf{break}\, w\rangle} \qquad \dfrac{}{\vdash \langle h_1, \mathsf{seq}_1(\mathsf{break}\, w, s_2)\rangle \Downarrow_m \langle h', \mathsf{break}\, w\rangle}}{\vdash \langle h, s_1; s_2\rangle \Downarrow_m \langle h', \mathsf{break}\, w\rangle}$$

$$\text{(B)}\ \frac{\dfrac{\dots}{\vdash \langle h, s_1\rangle \Downarrow_m \langle h_1, w_1\rangle} \qquad \text{(C)}\ \dfrac{\dfrac{\dots}{\vdash \langle h_1, s_2\rangle \Downarrow_m \langle h', \mathsf{break}\, w\rangle} \qquad \dfrac{w \neq \mathsf{empty}}{\vdash \langle h', \mathsf{seq}_2(w_1, \mathsf{break}\, w)\rangle \Downarrow_m \langle h', \mathsf{break}\, w\rangle}}{\vdash \langle h_1, \mathsf{seq}_1(w_1, s_2)\rangle \Downarrow_m \langle h', \mathsf{break}\, w\rangle}}{\vdash \langle h, s_1; s_2\rangle \Downarrow_m \langle h', \mathsf{break}\, w\rangle}$$

$$\text{(D)}\ \frac{\dfrac{\dots}{\vdash \langle h, s_1\rangle \Downarrow_m \langle h_1, w\rangle} \qquad \text{(E)}\ \dfrac{\dfrac{\dots}{\vdash \langle h_1, s_2\rangle \Downarrow_m \langle h', \mathsf{break}\, \mathsf{empty}\rangle} \qquad \dfrac{}{\vdash \langle h', \mathsf{seq}_2(w, \mathsf{break}\, \mathsf{empty})\rangle \Downarrow_m \langle h', \mathsf{break}\, w\rangle}}{\vdash \langle h_1, \mathsf{seq}_1(w, s_2)\rangle \Downarrow_m \langle h', \mathsf{break}\, w\rangle}}{\vdash \langle h, s_1; s_2\rangle \Downarrow_m \langle h', \mathsf{break}\, w\rangle}$$

Figure B.14.: JavaScript operational semantics proof trees of a sequence with break.

From the operational semantics of JSIL we get the following for the rest of the statements:

$$\overline{\mathsf{c}}_{a(l_3,l_4)} \vdash \langle h', \rho_2, \_, l_3\rangle \rightarrow_m^* \langle h', \rho', \_, l_4\rangle \wedge \rho'(\mathtt{x}) = w.$$

Combining the above transitions using Lemma 6 (concatenation), we obtain that

$$\overline{\mathsf{c}}_{a(l_1,l_4)} \vdash \langle h, \rho, \_, l_1\rangle \rightarrow_m^* \langle h', \rho', \_, l_4\rangle \wedge \rho'(\mathtt{x}) = w.$$

If we take the second proof tree from Fig. B.13 using hypothesis (3) to C and D we get

$$\overline{\mathsf{c}}_{1(l_1,l_2)} \vdash \langle h, \rho, \_, l_1\rangle \rightarrow_m^* \langle h_1, \rho_1, \_, l_2\rangle \wedge \rho_1(\mathtt{x}_1) = w_1$$
$$\overline{\mathsf{c}}_{2(l_2,l_3)} \vdash \langle h_1, \rho_1, \_, l_2\rangle \rightarrow_m^* \langle h', \rho_2, \_, l_3\rangle \wedge \rho_2(\mathtt{x}_2) = w$$

From the proof tree we also get that $w \neq \mathsf{empty}$. In the same way as in the first case, using operational semantics of JSIL and combining the transition relations using Lemma 6 (concatenation) we conclude that

$$\overline{\mathsf{c}}_{a(l_1,l_4)} \vdash \langle h, \rho, \_, l_1\rangle \rightarrow_m^* \langle h', \rho', \_, l_4\rangle \wedge \rho'(\mathtt{x}) = w.$$

Sequence rules are related to break rules. Hence we explicitly investigate **G5** and prove

$$\wp, L, v_t \vdash \langle h, s_1; s_2\rangle \Downarrow_m \langle h', \mathsf{break}\, w\rangle \Rightarrow \exists \rho' \,.\, \overline{\mathsf{c}}_{(l_1,l_4)} \vdash \langle h, \rho, \_, l_1\rangle \rightarrow_m^* \langle h', \rho', i', b_r\rangle \wedge i' \overset{k}{\mapsto} b_r$$
$$\wedge\, \rho'(\overline{\mathsf{b}}_k) = (w \neq \mathsf{empty}\ ?\ w : \rho'(\mathtt{x}_{pr}))$$

We need to consider three proof trees of $\wp, L, v_t \vdash \langle h, s\rangle \Downarrow_m \langle h', \mathsf{break}\, w\rangle$ that are shown in Fig. B.14. They correspond to either s_1 returning $\mathsf{break}\, w$ or s_2 returning either $\mathsf{break}\, \mathsf{empty}$ or $\mathsf{break}\, w'$, where $w' \neq \mathsf{empty}$. Let us take the first proof tree. We can apply induction hypothesis (5) to A to obtain

$$\overline{\mathsf{c}}_{(l_1,l_2)} \vdash \langle h, \rho, \_, l_1\rangle \rightarrow_m^* \langle h', \rho', i', b_r\rangle \wedge i' \overset{k}{\mapsto} b_r \wedge \rho'(\overline{b'}_k) = (w \neq \mathsf{empty}\ ?\ w : \rho'(\mathtt{x}_{pr}))$$

Since $\overline{\mathsf{c}}_1$ is at the start of $\overline{\mathsf{c}}$, $\rho'(\overline{\mathsf{b}}_k) = \rho'(\overline{\mathsf{b}}'_k)$. Using Lemma 7 (concatenation), we get **G5**.

$$\text{(A)}\ \frac{\dfrac{\dots}{\vdash \langle h, s_1\rangle \Downarrow_m \langle h_1, \mathsf{error}\, w\rangle} \qquad \dfrac{}{\vdash \langle h_1, \mathsf{seq}_1(\mathsf{error}\, w, s_2)\rangle \Downarrow_m \langle h', \mathsf{error}\, w\rangle}}{\vdash \langle h, s_1; s_2\rangle \Downarrow_m \langle h', \mathsf{error}\, w\rangle}$$

$$\text{(B)}\ \frac{\dfrac{\dots}{\vdash \langle h, s_1\rangle \Downarrow_m \langle h_1, w_1\rangle} \qquad \text{(C)}\ \dfrac{\dfrac{\dots}{\vdash \langle h_1, s_2\rangle \Downarrow_m \langle h', \mathsf{error}\, w\rangle} \qquad \dfrac{}{\vdash \langle h', \mathsf{seq}_2(w_1, \mathsf{error}\, w)\rangle \Downarrow_m \langle h', \mathsf{error}\, w\rangle}}{\vdash \langle h_1, \mathsf{seq}_1(w_1, s_2)\rangle \Downarrow_m \langle h', \mathsf{error}\, w\rangle}}{\vdash \langle h, s_1; s_2\rangle \Downarrow_m \langle h', \mathsf{error}\, w\rangle}$$

Figure B.15.: JavaScript operational semantics proof trees of a sequence with error.

For the second proof tree we apply induction hypothesis (3, 5) to B and C to obtain

$$\overline{\mathsf{c}}_{1(l_1,l_2)} \vdash \langle h, \rho, \_, l_1\rangle \rightarrow_m^* \langle h_1, \rho_1, \_, l_2\rangle \wedge \rho_1(\mathtt{x}_1) = w_1$$
$$\overline{\mathsf{c}}_{2(l_2,l_3)} \vdash \langle h_1, \rho_1, \_, l_2\rangle \rightarrow_m^* \langle h', \rho', i', b_r\rangle \wedge i' \overset{k'}{\mapsto} b_r \wedge \rho'(\overline{\mathsf{b}}''_{k'}) = w$$

From the proof tree we know that $w \neq \mathsf{empty}$. Since $\overline{\mathsf{c}}_1$ comes before $\overline{\mathsf{c}}_2$ in $\overline{\mathsf{c}}_a$, we get that $\overline{\mathsf{b}}_k = \overline{\mathsf{b}}''_{k'}$ where $k = k' + d$ and d is the length of $\overline{\mathsf{b}}'$. Using Lemma 7 (concatenation), we get **G5**.

For the third proof tree we apply induction hypothesis (3, 5) to D and E to obtain

$$\overline{\mathsf{c}}_{1(l_1,l_2)} \vdash \langle h, \rho, \_, l_1\rangle \rightarrow_m^* \langle h_1, \rho_1, \_, l_2\rangle \wedge \rho_1(\mathtt{x}_1) = w$$
$$\overline{\mathsf{c}}_{2(l_2,l_3)} \vdash \langle h_1, \rho_1, \_, l_2\rangle \rightarrow_m^* \langle h', \rho', i', b_r\rangle \wedge \rho'(\overline{\mathsf{b}}_k) = \rho'(\mathtt{x}_1)$$

Again $\overline{\mathsf{b}}_k = \overline{\mathsf{b}}''_{k'}$ where $k = k' + d$ and d is the length of $\overline{\mathsf{b}}'$. Hence $\rho'(\overline{\mathsf{b}}_k) = w$ and, using Lemma 7 (concatenation), we get **G5**.

To illustrate how errors propagate through the translated code we investigate **G4** for sequence and prove

$$\wp, L, v_t \vdash \langle h, s_1; s_2\rangle \Downarrow_m \langle h', \mathsf{error}\, w\rangle \Rightarrow \exists \rho' \,.\, \overline{\mathsf{c}}_{(l_1,l_4)} \vdash \langle h, \rho, \_, i\rangle \rightarrow_m^* \langle h', \rho', i', \mathtt{perr}\rangle \wedge i' \overset{k}{\mapsto} \mathtt{perr} \wedge \rho'(\overline{\mathsf{e}}_k) = w$$

Fig. B.15 shows the two proof trees that produce error for sequence statement. Consider the first proof tree. Using induction hypothesis (4) on A we get

$$\overline{\mathsf{c}}_{1(l_1,l_2)} \vdash \langle h, \rho, \_, i\rangle \rightarrow_m^* \langle h', \rho', i', \mathtt{perr}\rangle \wedge i' \overset{k}{\mapsto} \mathtt{perr} \wedge \rho'(\overline{\mathsf{e}}'_k) = w$$

Now $\overline{\mathsf{e}}_k = \overline{\mathsf{e}}'_k$, since $\overline{\mathsf{c}}_1$ is at the start of $\overline{\mathsf{c}}_a$. Hence $\rho'(\overline{\mathsf{e}}_k) = w$. From Lemma 7 (concatenation) we immediately get **G4**.

For the second proof tree we apply induction hypothesis (3, 4) to B and C to obtain

$$\overline{\mathsf{c}}_{1(l_1,l_2)} \vdash \langle h, \rho, \_, l_1\rangle \rightarrow_m^* \langle h_1, \rho_1, \_, l_2\rangle \wedge \rho_1(\mathtt{x}_1) = w_1$$
$$\overline{\mathsf{c}}_{2(l_2,l_3)} \vdash \langle h_1, \rho_1, \_, l_2\rangle \rightarrow_m^* \langle h', \rho', i', \mathtt{perr}\rangle \wedge i' \overset{k'}{\mapsto} \mathtt{perr} \wedge \rho'(\overline{\mathsf{e}}''_{k'}) = w$$

Since $\overline{\mathsf{c}}_1$ comes before $\overline{\mathsf{c}}_2$ in $\overline{\mathsf{c}}_a$, we get that $\overline{\mathsf{e}}_k = \overline{\mathsf{e}}''_{k'}$ where $k = k' + d$ and d is the length of $\overline{\mathsf{e}}'$. Again, using Lemma 7 (concatenation) we get **G4**.

$$\text{(A)}\ \frac{\dfrac{\dots}{\vdash \langle h, s_1\rangle \Downarrow_m \langle h_1, \mathsf{ret}\, w\rangle} \qquad \dfrac{}{\vdash \langle h_1, \mathsf{seq}_1(\mathsf{ret}\, w, s_2)\rangle \Downarrow_m \langle h', \mathsf{ret}\, w\rangle}}{\vdash \langle h, s_1; s_2\rangle \Downarrow_m \langle h', \mathsf{ret}\, w\rangle}$$

$$\text{(B)}\ \frac{\dfrac{\dots}{\vdash \langle h, s_1\rangle \Downarrow_m \langle h_1, w_1\rangle} \qquad \text{(C)}\ \dfrac{\dfrac{\dots}{\vdash \langle h_1, s_2\rangle \Downarrow_m \langle h', \mathsf{ret}\, w\rangle} \qquad \dfrac{}{\vdash \langle h', \mathsf{seq}_2(w_1, \mathsf{ret}\, w)\rangle \Downarrow_m \langle h', \mathsf{ret}\, w\rangle}}{\vdash \langle h_1, \mathsf{seq}_1(w_1, s_2)\rangle \Downarrow_m \langle h', \mathsf{ret}\, w\rangle}}{\vdash \langle h, s_1; s_2\rangle \Downarrow_m \langle h', \mathsf{ret}\, w\rangle}$$

Figure B.16.: JavaScript operational semantics proof trees of a sequence with ret.

The final case left to prove for the sequence is **G6**:

$$\wp, L, v_t \vdash \langle h, s_1; s_2\rangle \Downarrow_m \langle h', \mathsf{ret}\, w\rangle \Rightarrow \exists \rho' \,.\, \overline{\mathtt{c}}_{(l_1,l_4)} \vdash \langle h, \rho, \_, i\rangle \rightarrow_m^* \langle h', \rho', i', \mathtt{pret}\rangle \wedge i' \overset{k}{\mapsto} \mathtt{pret} \wedge \rho'(\overline{\mathtt{r}}_k) = w$$

Fig. B.16 shows the two proof trees that contains outcome ret for sequence statement. Consider the first proof tree. Using induction hypothesis (6) on A we get

$$\overline{\mathtt{c}}_{1(l_1,l_2)} \vdash \langle h, \rho, \_, i\rangle \rightarrow_m^* \langle h', \rho', i', \mathtt{pret}\rangle \wedge i' \overset{k}{\mapsto} \mathtt{pret} \wedge \rho'(\overline{\mathtt{r}}'_k) = w$$

Now $\overline{\mathtt{r}}_k = \overline{\mathtt{r}}'_k$, since $\overline{\mathtt{c}}_1$ is at the start of $\overline{\mathtt{c}}_a$. Hence $\rho'(\overline{\mathtt{r}}_k) = w$. From Lemma 7 (concatenation) we immediately get **G6**.

For the second proof tree we apply induction hypothesis (3, 6) to B and C to obtain

$$\overline{\mathtt{c}}_{1(l_1,l_2)} \vdash \langle h, \rho, \_, l_1\rangle \rightarrow_m^* \langle h_1, \rho_1, \_, l_2\rangle \wedge \rho_1(\mathtt{x}_1) = w_1$$
$$\overline{\mathtt{c}}_{2(l_2,l_3)} \vdash \langle h_1, \rho_1, \_, l_2\rangle \rightarrow_m^* \langle h', \rho', i', \mathtt{pret}\rangle \wedge i' \overset{k'}{\mapsto} \mathtt{pret} \wedge \rho'(\overline{\mathtt{r}}''_{k'}) = w$$

Since $\overline{\mathtt{c}}_1$ comes before $\overline{\mathtt{c}}_2$ in $\overline{\mathtt{c}}_a$, we get that $\overline{\mathtt{r}}_k = \overline{\mathtt{r}}''_{k'}$ where $k = k' + d$ and d is the length of $\overline{\mathtt{r}}'$. Again, using Lemma 7 (concatenation) we get **G6**.

[Functions: $m(\overline{x}, \overline{v})$]
Let

$$\mathcal{C}_m(s, \mathtt{x}, \_, \_) = \overline{\mathtt{c}}, \overline{\mathtt{e}}, \overline{\mathtt{r}}, \_$$

where $\wp(m) = \lambda \overline{x}.s$ and $l_0 = 0$. We need to prove **G7-G8**. The translated code is the body of the

$$\frac{\wp(m) = \lambda\overline{x}.s \qquad \dfrac{\cdots}{\wp, L', v_t \vdash \langle h', s\rangle \Downarrow_m \langle h_f, w'\rangle}}{\wp, L, v_t \vdash \langle h, m(\overline{x}, \overline{v})\rangle \Downarrow_m \langle h_f, \mathsf{undefined}\rangle}\ (\mathrm{A})$$

$$\frac{\wp(m) = \lambda\overline{x}.s \qquad \dfrac{\cdots}{\wp, L', v_t \vdash \langle h', s\rangle \Downarrow_m \langle h_f, \mathsf{ret}\, w\rangle}}{\wp, L, v_t \vdash \langle h, m(\overline{x}, \overline{v})\rangle \Downarrow_m \langle h_f, w\rangle}\ (\mathrm{B})$$

where $L' = L \mathbin{@} [l_s]$, $h' = h \uplus \mathsf{env}_m(l_s, \overline{x}, \overline{v}, s)$

Figure B.17.: JavaScript operational semantics proof tree of a full function body for the normal execution.

procedure $\mathsf{proc}\, m(\mathtt{x}_{sc}, \mathtt{x}_{\mathsf{this}}, \mathtt{x}_i|_{i=1}^n)$:

$$\overline{\mathtt{c}}_a = \begin{cases} l_0: & \mathtt{x}_{er} := \mathsf{new}\,() \\ l_1: & \mathtt{xscope} := \mathtt{x}_{sc} \mathbin{@} [\mathtt{x}_{er}] \\ l_2: & [\mathtt{x}_{er}, y_i] := \mathsf{undefined}\,|_{i=1}^k \\ l_3: & [\mathtt{x}_{er}, x_i] := \mathtt{x}_i|_{i=1}^n \\ l_4: & \overline{\mathtt{c}} \\ l_5: & \mathtt{x}_{n+1} := \mathsf{undefined} \\ \mathtt{pret}: & \mathtt{xret} := \phi(\overline{\mathtt{r}} :: \mathtt{x}_{n+1}) \\ \mathtt{ret}: & \mathsf{skip} \\ \mathtt{perr}: & \mathtt{xerr} := \phi(\overline{\mathtt{e}}) \\ \mathtt{err}: & \mathsf{skip} \end{cases}$$

where $y_i|_{i=1}^k = \mathsf{defs}(s)$.

For the moment, let us focus on the success case **G7** and prove

$$\wp, L, v_t \vdash \langle h, m(\overline{x}, \overline{v})\rangle \Downarrow_m \langle h_f, w\rangle \Rightarrow \exists \rho_f\,.\, \overline{\mathtt{c}}_a \vdash \langle h, \rho, \_, 0\rangle \rightarrow_m^* \langle h_f, \rho_f, \_, \mathtt{ret}\rangle \wedge \rho_f(\mathtt{xret}) = w$$

where $\rho = \emptyset[x_i \mapsto v_i|_{i=1}^n, \mathtt{x}_{sc} \mapsto L, \mathtt{x}_{\mathsf{this}} \mapsto v_t]$.

The two proof trees in Fig. B.17 correspond to the two paths where the execution of the statement s terminates with normally or with the ret flag. First, we apply Lemma 14 (ER) to $\mathsf{env}_m(l_s, \overline{x}, \overline{v}, s)$, to both proof trees to obtain the following:

$$\overline{\mathtt{c}}_{a(l_0,l_4)} \vdash \langle h, \rho, \_, 0\rangle \rightarrow_m^* \langle h', \rho_1, \_, l_4\rangle \wedge \rho_1(\mathtt{xscope}) = L'$$

If we, then, take the first proof tree, we can apply the induction hypothesis (3) to A to obtain:

$$\overline{\mathtt{c}}_{a(l_4,l_5)} \vdash \langle h', \rho_1, \_, l_4\rangle \rightarrow_m^* \langle h_f, \rho_2, \_, l_5\rangle \wedge \rho_2(\mathtt{x}) = w'$$

$$\frac{\wp(m) = \lambda\overline{x}.s \qquad \dfrac{\cdots}{\wp, L', v_t \vdash \langle h', s\rangle \Downarrow_m \langle h_f, \mathsf{error}\, w\rangle}}{\wp, L, v_t \vdash \langle h, m(\overline{x}, \overline{v})\rangle \Downarrow_m \langle h_f, \mathsf{error}\, w\rangle}\ (\mathrm{A})$$

where $L' = L \mathbin{@} [l_s]$, $h' = h \uplus \mathsf{env}_m(l_s, \overline{x}, \overline{v}, s)$

Figure B.18.: JavaScript operational semantics proof tree of a full function body for the error case.

From the operational semantics of JSIL we get the following for the rest of the statements:

$$\overline{\mathtt{c}}_{a(l_5,\mathtt{ret})} \vdash \langle h_f, \rho_2, \_, l_5\rangle \rightarrow_m^* \langle h_f, \rho_f, \_, \mathtt{ret}\rangle \wedge \rho_f(\mathtt{xret}) = \mathsf{undefined}$$

Combining the above transitions using Lemma 6 (concatenation), we obtain that

$$\overline{\mathtt{c}}_a \vdash \langle h, \rho, \_, 0\rangle \rightarrow_m^* \langle h_f, \rho_f, \_, \mathtt{ret}\rangle \wedge \rho_f(\mathtt{xret}) = \mathsf{undefined}$$

If we take the second proof tree from Fig. B.17 using hypothesis (6) to B we get

$$\overline{\mathtt{c}}_{1(l_4,\mathtt{pret})} \vdash \langle h', \rho', \_, l_4\rangle \rightarrow_m^* \langle h_f, \rho_2, i', \mathtt{pret}\rangle \wedge i' \overset{k}{\mapsto} \mathtt{pret} \wedge \rho_2(\overline{\mathtt{r}}_k) = w$$

From the operational semantics of JSIL we get the following for the last statement:

$$\overline{\mathtt{c}}_{a(\mathtt{pret},\mathtt{ret})} \vdash \langle h_f, \rho_2, i', \mathtt{pret}\rangle \rightarrow_m^* \langle h_f, \rho_f, \_, \mathtt{ret}\rangle \wedge \rho_f(\mathtt{xret}) = w$$

In the same way as in the first case, combining the transition relations using Lemma 6 (concatenation), we conclude that

$$\overline{\mathtt{c}}_a \vdash \langle h, \rho, \_, 0\rangle \rightarrow_m^* \langle h_f, \rho_f, \_, \mathtt{ret}\rangle \wedge \rho_f(\mathtt{xret}) = w$$

Next we prove the error case **G8**:

$$\wp, L, v_t \vdash \langle h, m(\overline{x}, \overline{v})\rangle \Downarrow_m \langle h_f, \mathsf{error}\, w\rangle \Rightarrow \exists \rho_f \,.\, \overline{\mathtt{c}}_a \vdash \langle h, \rho, \_, 0\rangle \rightarrow_m^* \langle h_f, \rho_f, \_, \mathtt{err}\rangle \wedge \rho_f(\mathtt{xerr}) = w$$

where $\rho = \emptyset[x_i \mapsto v_i|_{i=1}^n, \mathtt{x}_{sc} \mapsto L, \mathtt{x}_{\mathsf{this}} \mapsto v_t]$.

The proof tree in Fig. B.18 corresponds to the execution of the statement s which terminates with an error. First, we apply Lemma 14 (ER) to $\mathsf{env}_m(l_s, \overline{x}, \overline{v}, s)$, to obtain the following:

$$\overline{\mathtt{c}}_{a(l_0,l_4)} \vdash \langle h, \rho, \_, 0\rangle \rightarrow_m^* \langle h', \rho_1, \_, l_4\rangle \wedge \rho_1(\mathtt{xscope}) = L'$$

Then, we can apply the induction hypothesis (4) to A to obtain:

$$\overline{\mathtt{c}}_{1(l_4,\mathtt{perr})} \vdash \langle h', \rho_1, \_, l_4\rangle \rightarrow_m^* \langle h_f, \rho_2, i, \mathtt{perr}\rangle \wedge i \overset{k}{\mapsto} \mathtt{perr} \wedge \rho_2(\overline{\mathtt{r}}_k) = w$$

From the operational semantics of JSIL we get the following for the last statement:

$$\overline{\mathtt{c}}_{a(\mathtt{perr},\mathtt{err})} \vdash \langle h_f, \rho_2, i, \mathtt{perr}\rangle \rightarrow_m^* \langle h_f, \rho_f, \_, \mathtt{err}\rangle \wedge \rho_f(\mathtt{xerr}) = w$$

Combining the transition relations using Lemma 6 (concatenation), we conclude that

$$\overline{\mathsf{c}}_a \vdash \langle h, \rho, \_, 0\rangle \rightarrow_m^* \langle h_f, \rho_f, \_, \mathtt{err}\rangle \wedge \rho_f(\mathtt{xerr}) = w$$

□

Proof: $\Longleftarrow$. We prove the theorem and the two lemmas together, by rule induction $\rightarrow_m^*$ on cases where $\overline{\mathsf{c}}_{(i,j)}$ are valid translations of expression e, statement s or function body $m(\overline{x}, \overline{v})$. The statements that we are proving are:

$$\wp, L, v_t \vdash \langle h, e\rangle \Downarrow_m \langle h', w\rangle \Leftarrow \exists \rho' \,.\, \overline{\mathsf{c}}_{(i,j)} \vdash \langle h, \rho, \_, i\rangle \rightarrow_m^* \langle h', \rho', \_, j\rangle \wedge \rho'(\mathtt{x}) = w \quad (\mathbf{G1})$$

$$\wp, L, v_t \vdash \langle h, e\rangle \Downarrow_m \langle h', \mathsf{error}\, w\rangle \Leftarrow \exists \rho' \,.\, \overline{\mathsf{c}}_{(i,j)} \vdash \langle h, \rho, \_, i\rangle \rightarrow_m^* \langle h', \rho', i', \mathtt{perr}\rangle \wedge i' \overset{k}{\mapsto} \mathtt{perr} \wedge \rho'(\overline{\mathsf{e}}_k) = w \quad (\mathbf{G2})$$

$$\wp, L, v_t \vdash \langle h, s\rangle \Downarrow_m \langle h', w\rangle \Leftarrow \exists \rho' \,.\, \overline{\mathsf{c}}_{(i,j)} \vdash \langle h, \rho, \_, i\rangle \rightarrow_m^* \langle h', \rho', \_, j\rangle \wedge \rho'(\mathtt{x}) = w \quad (\mathbf{G3})$$

$$\wp, L, v_t \vdash \langle h, s\rangle \Downarrow_m \langle h', \mathsf{error}\, w\rangle \Leftarrow \exists \rho' \,.\, \overline{\mathsf{c}}_{(i,j)} \vdash \langle h, \rho, \_, i\rangle \rightarrow_m^* \langle h', \rho', i', \mathtt{perr}\rangle \wedge i' \overset{k}{\mapsto} \mathtt{perr} \wedge \rho'(\overline{\mathsf{e}}_k) = w \quad (\mathbf{G4})$$

$$\wp, L, v_t \vdash \langle h, s\rangle \Downarrow_m \langle h', \mathsf{break}\, w\rangle \Leftarrow \exists \rho' \,.\, \overline{\mathsf{c}}_{(i,j)} \vdash \langle h, \rho, \_, i\rangle \rightarrow_m^* \langle h', \rho', i', b_r\rangle \wedge i' \overset{k}{\mapsto} b_r \quad (\mathbf{G5})$$
$$\wedge\, \rho'(\overline{\mathsf{b}}_k) = (w \neq \mathsf{empty}\ ?\ w : \rho'(\mathtt{x}_{pr}))$$

$$\wp, L, v_t \vdash \langle h, s\rangle \Downarrow_m \langle h', \mathsf{ret}\, w\rangle \Leftarrow \exists \rho' \,.\, \overline{\mathsf{c}}_{(i,j)} \vdash \langle h, \rho, \_, i\rangle \rightarrow_m^* \langle h', \rho', i', \mathtt{pret}\rangle \wedge i' \overset{k}{\mapsto} \mathtt{pret} \wedge \rho'(\overline{\mathtt{r}}_k) = w \quad (\mathbf{G6})$$

$$\wp, L, v_t \vdash \langle h, m(\overline{x}, \overline{v})\rangle \Downarrow_m \langle h_f, w\rangle \Leftarrow \exists \rho_f \,.\, \hat{\mathcal{C}}(s) \vdash \langle h, \rho, \_, 0\rangle \rightarrow_m^* \langle h_f, \rho_f, \_, \mathtt{ret}\rangle \wedge \rho_f(\mathtt{xret}) = w \quad (\mathbf{G7})$$

$$\wp, L, v_t \vdash \langle h, m(\overline{x}, \overline{v})\rangle \Downarrow_m \langle h_f, \mathsf{error}\, w\rangle \Leftarrow \exists \rho_f \,.\, \hat{\mathcal{C}}(s) \vdash \langle h, \rho, \_, 0\rangle \rightarrow_m^* \langle h_f, \rho_f, \_, \mathtt{err}\rangle \wedge \rho_f(\mathtt{xerr}) = w \quad (\mathbf{G8})$$

where, for the last two cases, $\rho = \emptyset[x_i \mapsto v_i|_{i=1}^n, \mathtt{x}_{sc} \mapsto L, \mathtt{x}_{\mathsf{this}} \mapsto v_t]$, and for the other cases, $\rho \geq \emptyset[\mathtt{xscope} \mapsto L, \mathtt{x}_{\mathsf{this}} \mapsto v_t]$.

The proof cases are essentially the same as in $\Longrightarrow$ proof. We will only consider the assignment case to show the equivalence. Other cases follow the same pattern.

[Expressions: Assignment Expression: $e \equiv e_1 = e_2$]
Let $C_m(e_1, \mathtt{x}_1) = \overline{\mathsf{c}}_{1(l_1,l_2)}, \overline{\mathsf{e}}'$, $C_m(e_2, \mathtt{x}_2) = \overline{\mathsf{c}}_{2(l_2,l_3)}, \overline{\mathsf{e}}''$, and $C_m(e_1 = e_2, \mathtt{x}) = \overline{\mathsf{c}}_{a(l_1,l_6)}, \overline{\mathsf{e}}$. We need to prove the following:

$$\wp, L, v_t \vdash \langle h, e_1 = e_2\rangle \Downarrow_m \langle h', w\rangle \Leftarrow \exists \rho' \,.\, \overline{\mathsf{c}}_{a(l_1,l_6)} \vdash \langle h, \rho, \_, l_1\rangle \rightarrow_m^* \langle h', \rho', \_, l_6\rangle \wedge \rho'(\mathtt{x}) = w$$
$$\wp, L, v_t \vdash \langle h, e_1 = e_2\rangle \Downarrow_m \langle h', \mathsf{error}\, w\rangle \Leftarrow \exists \rho' \,.\, \overline{\mathsf{c}}_{a(l_1,l_6)} \vdash \langle h, \rho, \_, l_1\rangle \rightarrow_m^* \langle h', \rho', i', \mathtt{perr}\rangle \wedge i' \overset{k}{\mapsto} \mathtt{perr} \wedge \rho'(\overline{\mathsf{e}}_k) = w$$

where $\rho \geq \emptyset[\mathtt{xscope} \mapsto L, \mathtt{x}_{\mathsf{this}} \mapsto v_t]$.

For the moment, let us again focus on the success case and prove

$$\wp, L, v_t \vdash \langle h, e_1 = e_2\rangle \Downarrow_m \langle h', w\rangle \Leftarrow \exists \rho' \,.\, \overline{\mathsf{c}}_{a(l_1,l_6)} \vdash \langle h, \rho, \_, l_1\rangle \rightarrow_m^* \langle h', \rho', \_, l_6\rangle \wedge \rho'(\mathtt{x}) = w.$$

Assume that we have ρ' and $\overline{\mathsf{c}}_{a(l_1,l_6)} \vdash \langle h, \rho, \_, l_1\rangle \rightarrow_m^* \langle h', \rho', \_, l_6\rangle \wedge \rho'(\mathtt{x}) = w$. We need to construct

$\wp, L, v_t \vdash \langle h, e_1 = e_2\rangle \Downarrow_m \langle h', w\rangle$. The translated code is

$$\overline{\mathtt{c}}_a = \begin{cases} l_1: & \overline{\mathtt{c}}_1 \\ l_2: & \overline{\mathtt{c}}_2 \\ l_3: & \mathtt{x} := \mathtt{getValue}(\mathtt{x}_2)\ \mathit{with}\ \mathtt{perr} \\ l_4: & \mathtt{x}' := \mathtt{checkAssignment}(\mathtt{x}_1)\ \mathit{with}\ \mathtt{perr} \\ l_5: & \mathtt{x}'' := \mathtt{putValue}(\mathtt{x}_1, \mathtt{x})\ \mathit{with}\ \mathtt{perr} \end{cases}$$

In the case of the normal execution path, we can decompose the above derivation into the following using Lemma 6 (concatenation):

$$\begin{aligned} \overline{\mathtt{c}}_{1(l_1,l_2)} \vdash \langle h, \rho, \_, l_1\rangle &\rightarrow_m^* \langle h_1, \rho_1, \_, l_2\rangle \wedge \rho_1(\mathtt{x}_1) = w_1 \\ \overline{\mathtt{c}}_{2(l_2,l_3)} \vdash \langle h, \rho, \_, l_2\rangle &\rightarrow_m^* \langle h_1, \rho_2, \_, l_3\rangle \wedge \rho_2(\mathtt{x}_2) = w_2 \\ \mathtt{x} := \mathtt{getValue}(\mathtt{x}_2)\ \mathit{with}\ \mathtt{perr} \vdash \langle h_2, \rho_2, \_, l_3\rangle &\rightarrow_m^* \langle h_3, \rho_3, l_3, l_4\rangle \wedge \rho_3(\mathtt{x}) = v \\ \mathtt{x}' := \mathtt{checkAssignment}(\mathtt{x}_1)\ \mathit{with}\ \mathtt{perr} \vdash \langle h_3, \rho_3, \_, l_4\rangle &\rightarrow_m^* \langle h_3, \rho_4, l_3, l_5\rangle \\ \mathtt{x}'' := \mathtt{putValue}(\mathtt{x}_1, \mathtt{x})\ \mathit{with}\ \mathtt{perr} \vdash \langle h_3, \rho_4, l_3, l_5\rangle &\rightarrow_m^* \langle h', \rho', l_5, l_6\rangle \wedge \rho'(\mathtt{x}'') = \mathsf{empty} \end{aligned}$$

Applying the induction hypothesis (1) Lemma 9 (`getValue`), Lemma 11 (`checkAssignment`) and Lemma 10 (`putValue`), we get the proof trees for A, B, C, D, E and we can construct the proof tree of $\wp, L, v_t \vdash \langle h, e_1 = e_2\rangle \Downarrow_m \langle h_4, v\rangle$ as shown in Fig. B.7.

□

B.2.3. Helper Lemmas

Lemma 6 (Concatenation 1)**.**

$$\begin{aligned} &\forall h', \rho'.\overline{\mathtt{c}}_1 \vdash \langle h, \rho, \_, \_\rangle \rightarrow_m^* \langle h', \rho', \_, i\rangle \Longrightarrow \\ &(\overline{\mathtt{c}}_1 :: \overline{\mathtt{c}}_2 \vdash \langle h, \rho, \_, i\rangle \rightarrow_m^* \langle h'', \rho'', \_, j\rangle \iff \overline{\mathtt{c}}_2 \vdash \langle h', \rho', i, \_\rangle \rightarrow_m^* \langle h'', \rho'', \_, j\rangle) \end{aligned}$$

Lemma 7 (Concatenation 2)**.**

$$\forall h', \rho'.\overline{\mathtt{c}}_1 \vdash \langle h, \rho, i, j\rangle \rightarrow_m^* \langle h', \rho', i', j'\rangle \Longrightarrow \overline{\mathtt{c}}_1 :: \overline{\mathtt{c}}_2 \vdash \langle h, \rho, i, j\rangle \rightarrow_m^* \langle h', \rho', i', j'\rangle$$

Lemma 8 (Store Expansion)**.**

$$\overline{\mathtt{c}} \vdash \langle h, \rho\rangle \rightarrow_m^* \langle h', \rho'\rangle \Longrightarrow \rho \leq \rho'$$

Lemma 9 (Correctness of `getValue`)**.** *Let* $\mathtt{c} = \mathtt{x} := \mathtt{getValue}(\mathtt{x}')$ *with* `perr`*. Then, for any* ρ*, such that* $\rho(\mathtt{x}') = w$*:*

$$\begin{aligned} \wp, L, v_t \vdash \langle h, \mathcal{I}_{gv}(w)\rangle \Downarrow_m \langle h', v\rangle &\iff \mathtt{c}_{(i,i+1)} \vdash \langle h, \rho, \_, i\rangle \rightarrow_m^* \langle h', \rho[\mathtt{x} \mapsto v], i, i+1\rangle \\ \wp, L, v_t \vdash \langle h, \mathcal{I}_{gv}(w)\rangle \Downarrow_m \langle h', \mathit{error}\, v\rangle &\iff \mathtt{c}_{(i,i+1)} \vdash \langle h, \rho, \_, i\rangle \rightarrow_m^* \langle h', \rho[\mathtt{x} \mapsto v], i, \mathtt{perr}\rangle \end{aligned}$$

Lemma 10 (Correctness of `putValue`)**.** *Let* $\mathtt{c} = \mathtt{x} := \mathtt{putValue}(\mathtt{x}', \mathtt{x}'')$ *with* `perr`*. For any* ρ*, such*

that $\rho(\mathtt{x}') = w$ *and* $\rho(\mathtt{x}'') = v$*:*

$$\wp, L, v_t \vdash \langle h, \mathcal{I}_{pv}(w, v)\rangle \Downarrow_m \langle h', \mathit{empty}\rangle \iff \mathtt{c}_{(i,i+1)} \vdash \langle h, \rho, \_, i\rangle \rightarrow_m^* \langle h', \rho[\mathtt{x} \mapsto \mathit{empty}], i, i+1\rangle$$
$$\wp, L, v_t \vdash \langle h, \mathcal{I}_{pv}(w, v)\rangle \Downarrow_m \langle h', \mathit{error}\, v'\rangle \iff \mathtt{c}_{(i,i+1)} \vdash \langle h, \rho, \_, i\rangle \rightarrow_m^* \langle h', \rho[\mathtt{x} \mapsto v'], i, \mathtt{perr}\rangle$$

Lemma 11 (Correctness of `checkAssignment`)**.** *Let* $\mathtt{c} = \mathtt{x} := \mathtt{checkAssignment}(\mathtt{x}')$ *with . For any* h *and* ρ*, such that* $\rho(\mathtt{x}') = w$*:*

$$(w = v \vee w = l.{}_o p \ \vee (w = l.{}_v p \wedge p \notin \{\mathit{eval}, \mathit{arguments}\})) \iff \mathtt{c}_{(i,i+1)} \vdash \langle h, \rho, \_, i\rangle \rightarrow_m^* \langle h, \rho[\mathtt{x} \mapsto \mathit{empty}], i, i+1\rangle$$
$$w = l.{}_v p \wedge p \in \{\mathit{eval}, \mathit{arguments}\}) \iff \mathtt{c}_{(i,i+1)} \vdash \langle h, \rho, \_, i\rangle \rightarrow_m^* \langle h', \rho[\mathtt{x} \mapsto l'], i, \mathtt{perr}\rangle$$

where $h' = h \uplus \mathsf{err}(l', l_{sep})$*.*

Lemma 12 (Correctness of `isCallable`)**.** *Let* $\mathtt{c} = \mathtt{x} := \mathtt{isCallable}(\mathtt{x}')$*. For any* h *and* ρ*, such that* $\rho(\mathtt{x}') = v$*:*

$$\mathcal{P}_c(h, v) \iff \mathtt{c}_{(i,i+1)} \vdash \langle h, \rho, \_, i\rangle \rightarrow_m^* \langle h, \rho[\mathtt{x} \mapsto \mathit{true}], i, i+1\rangle$$
$$\neg\mathcal{P}_c(h, v) \iff \mathtt{c}_{(i,i+1)} \vdash \langle h, \rho, \_, i\rangle \rightarrow_m^* \langle h, \rho[\mathtt{x} \mapsto \mathit{false}], i, i+1\rangle$$

Lemma 13 (Correctness of `SelectThis`)**.** *For any* h *and* ρ*, such that* $\rho(\mathtt{x}') = w$*:*

$$\mathsf{SelectThis}(w) = v \iff \exists \rho_f \,.\, \overline{\mathtt{c}}_{(n_1,n_3)} \vdash \langle h, \rho, \_, n_1\rangle \rightarrow_m^* \langle h, \rho_f, n_2, n_3\rangle \wedge \rho_f(\mathtt{x}_t) = v$$

where

$$\overline{\mathtt{c}} = \begin{cases} n_1: & \mathit{goto}\ [\mathit{typeOf}(\mathtt{x}') = \mathit{List} \wedge (\mathit{nth}(\mathtt{x}', 0) = o]\ t_2,\ e_2 \\ t_2: & \mathtt{x}_{t1} := \mathit{nth}(\mathtt{x}', 1) \\ & \mathit{goto}\ n_2 \\ e_2: & \mathtt{x}_{t2} := \mathit{undefined} \\ n_2: & \mathtt{x}_t := \phi(\mathtt{x}_{t1}, \mathtt{x}_{t2}) \end{cases}$$

Lemma 14 (Correctness of Enviroment Record Translation)**.** *For any* h *and* ρ*, such that* $\rho = \emptyset[x_i \mapsto v_i|_{i=1}^n, \mathtt{x}_{sc} \mapsto L, \mathtt{x}_{\mathit{this}} \mapsto v_t]$*:*

$$h' = h \uplus \mathsf{env}_m(l_s, \overline{x}, \overline{v}, s) \iff \exists \rho_f \,.\, \overline{\mathtt{c}}_{(l_1,l_5)} \vdash \langle h, \rho, \_, l_1\rangle \rightarrow_m^* \langle h', \rho_f, l_4, l_5\rangle \wedge \rho_f(\mathtt{xscope}) = L@[l_s]$$

where

$$\overline{\mathtt{c}} = \begin{cases} l_1: & \mathtt{x}_{er} := \mathit{new}() \\ l_2: & \mathtt{xscope} := \mathtt{x}_{sc}@[\mathtt{x}_{er}] \\ l_3: & [\mathtt{x}_{er}, y_i] := \mathit{undefined}|_{i=1}^k \\ l_4: & [\mathtt{x}_{er}, x_i] := \mathtt{x}_i|_{i=1}^n \end{cases}$$

and $y_i|_{i=1}^k = \mathsf{defs}(s)$*.*

Lemma 15 (Correctness of variable translation)**.** *Let* $\psi_m(x) = n \wedge \overline{c}_a = \mathtt{x} := \mathit{nth}(\mathtt{xscope}, n)$*. For*

any ρ*, such that* $\rho \geq \emptyset[\texttt{xscope} \mapsto L, \texttt{x}_{\mathit{this}} \mapsto v_t]$*:*

$$\wp, L, v_t \vdash \langle h, \mathcal{I}_\sigma(x)\rangle \Downarrow_m \langle h, v\rangle \iff \overline{\mathsf{c}}_{a(i,i+1)} \vdash \langle h, \rho, \_, i\rangle \rightarrow_m^* \langle h, \rho[\texttt{x} \mapsto v], i, i+1\rangle$$

Let $\psi_m(x) = \perp$ *and*

$$\overline{\mathsf{c}}_a = \begin{cases} i: & \texttt{x}' := \texttt{hasProperty}(l_g, x) \ \mathit{with}\ \texttt{perr} \\ & \mathit{goto}\ [\texttt{x}']\ t,\ e \\ t: & \texttt{x}'_1 := l_g \\ & \mathit{goto}\ n \\ & \texttt{x}'_2 := \mathit{undefined} \\ n: & \texttt{x} := \phi(\texttt{x}'_1, \texttt{x}'_2) \end{cases}$$

For any ρ*, such as* $\rho \geq \emptyset[\texttt{xscope} \mapsto L, \texttt{x}_{\mathit{this}} \mapsto v_t]$*:*

$$\wp, L, v_t \vdash \langle h, \mathcal{I}_\sigma(x)\rangle \Downarrow_m \langle h, v\rangle \iff \exists \rho' \,.\, \overline{\mathsf{c}}_{a(i,j)} \vdash \langle h, \rho, \_, i\rangle \rightarrow_m^* \langle h, \rho'[\texttt{x} \mapsto v], n, j\rangle$$

Proof. We use the notation L_i to denote the i^{th} element of the list L and the notation $L_{(i,j)}$ to de note the elements i^{th} up to j^{th} of the list L. First we prove the case when $\psi_m(x) = n$. To prove the equivalence we need to show that the values v on the both side match. From JSIL operational semantics we get that

$$\texttt{x} := \mathsf{nth}\,(\texttt{xscope}, n) \vdash \langle h, \rho, \_, i\rangle \rightarrow_m^* \langle h, \rho[\texttt{x} \mapsto L_n], i, i+1\rangle$$

Hence, we are left to show that $v = L_n$ on the left hand side. From $\psi_m(x) = n$ Lemma 17 (ψ_m correctness), we get that

$$(L_n, x) \in \mathsf{dom}(h) \wedge \forall i.\ i > n \Longrightarrow (L_i, x) \notin \mathsf{dom}(h)$$

Consequently, we get two unfolded proof trees for $\wp, L, v_t \vdash \langle h, \mathcal{I}_\sigma(x)\rangle \Downarrow_m \langle h, v\rangle$ as shown in Fig. B.19 depending on whether $\psi_m(x) = 0$ or not. In case when $\psi_m(x) > 0$, we get $v = L_n$ directly from the tree. When $\psi_m(x) = 0$, the proof tree (A) returns true, since $l_g = L_0$ and $(L_0, x) \in \mathsf{dom}(h)$. Hence, we get $v = L_0$.

Now we prove the case when $\psi_m(x) = \perp$. From Lemma 17 (ψ_m correctness) we get $\forall i, i > 1 \Longrightarrow (L_i, x) \notin \mathsf{dom}(h)$. Hence, the proof tree of $\wp, L, v_t \vdash \langle h, \mathcal{I}_\sigma(x)\rangle \Downarrow_m \langle h, v\rangle$ must unfold as shown in Fig. B.20. To prove implication to the right, we need to show

$$\overline{\mathsf{c}}_{a(i,j)} \vdash \langle h, \rho, \_, i\rangle \rightarrow_m^* \langle h, \rho'[\texttt{x} \mapsto v], n, j\rangle$$

Using `hasProperty` lemma on the proof tree (A) we get

$$\texttt{x}' := \texttt{hasProperty}(l_g, x)\ \mathit{with}\ \texttt{perr} \vdash \langle h, \rho, \_, i\rangle \rightarrow_m^* \langle h, \rho[\texttt{x}' \mapsto o], i, i+1\rangle$$

If $o = \mathsf{true}$, from rule GIL-GLOBAL, we get that $v = l_g$. Using JSIL operational semantics we get

$$\dfrac{\begin{array}{l}(L_d, x) \notin \mathsf{dom}(h)\\ L_d \neq l_g\end{array} \qquad \dfrac{\begin{array}{l}(L_{n+1}, x) \notin \mathsf{dom}(h)\\ L_{n+1} \neq l_g\end{array} \qquad \dfrac{(L_n, x) \in \mathsf{dom}(h) \qquad L_n \neq l_g}{\wp, L_{(0,n)}, v_t \vdash \langle h, \mathcal{I}_\sigma(x)\rangle \Downarrow_m \langle h, L_n\rangle}}{\cdots}}{\wp, L_{(0,d)}, v_t \vdash \langle h, \mathcal{I}_\sigma(x)\rangle \Downarrow_m \langle h, L_n\rangle}$$

$$\dfrac{\begin{array}{l}(L_d, x) \notin \mathsf{dom}(h)\\ L_d \neq l_g\end{array} \qquad \dfrac{\begin{array}{l}(L_1, x) \notin \mathsf{dom}(h)\\ L_1 \neq l_g\end{array} \qquad (*)}{\cdots}}{\wp, L_{(0,d)}, v_t \vdash \langle h, \mathcal{I}_\sigma(x)\rangle \Downarrow_m \langle h, l_g\rangle}$$

where $(*)$ is

$$\text{(A)}\ \dfrac{\dfrac{\cdots}{\wp, [l_g], l_g \vdash \langle h, \mathcal{I}_{hp}(x)\rangle \Downarrow_m \langle h, \mathsf{true}\rangle} \qquad \wp, [l_g], v_t \vdash \langle h, \mathcal{I}_\sigma(\mathsf{true})_1\rangle \Downarrow_m \langle h, l_g\rangle}{\wp, [l_g], v_t \vdash \langle h, \mathcal{I}_\sigma(x)\rangle \Downarrow_m \langle h, l_g\rangle}$$

Figure B.19.: JavaScript operational semantics proof trees of a variable dereferencing when $\psi_m(x) = n > 0$ and $\psi_m(x) = 0$ respectively, where $d + 1$ is the length of the current scope chain L.

$$\dfrac{\begin{array}{l}(L_d, x) \notin \mathsf{dom}(h)\\ L_d \neq l_g\end{array} \qquad \dfrac{\begin{array}{l}(L_1, x) \notin \mathsf{dom}(h)\\ L_1 \neq l_g\end{array} \qquad (*)}{\cdots}}{\wp, L_{(0,d)}, v_t \vdash \langle h, \mathcal{I}_\sigma(x)\rangle \Downarrow_m \langle h, v\rangle}$$

where $(*)$ is

$$\text{(A)}\ \dfrac{\dfrac{\cdots}{\wp, [l_g], l_g \vdash \langle h, \mathcal{I}_{hp}(x)\rangle \Downarrow_m \langle h, o\rangle} \qquad \wp, [l_g], v_t \vdash \langle h, \mathcal{I}_\sigma(o)_1\rangle \Downarrow_m \langle h, v\rangle}{\wp, [l_g], v_t \vdash \langle h, \mathcal{I}_\sigma(x)\rangle \Downarrow_m \langle h, v\rangle}$$

Figure B.20.: JavaScript operational semantics proof tree of a variable dereferencing when $\psi_m(x) = \bot$.

that $\overline{\mathsf{c}}_{a(i,j)} \vdash \langle h, \rho, \_, i\rangle \rightarrow_m^* \langle h, \rho'[\mathtt{x} \mapsto l_g], n, j\rangle$ as required. If $o = \mathsf{false}$, from rule GIR-UNDEF, we get that $v = \mathsf{undefined}$. Again, using JSIL operational semantics, we get $\overline{\mathsf{c}}_{a(i,j)} \vdash \langle h, \rho, \_, i\rangle \rightarrow_m^* \langle h, \rho'[\mathtt{x} \mapsto \mathsf{undefined}], n, j\rangle$ as required.

To prove the implication to the right, from $\overline{\mathsf{c}}_{a(i,j)} \vdash \langle h, \rho, \_, i\rangle \rightarrow_m^* \langle h, \rho'[\mathtt{x} \mapsto v], n, j\rangle$ we get

$$\mathtt{x}' := \mathtt{hasProperty}(l_g, x)\ \ \textit{with}\ \mathtt{perr} \vdash \langle h, \rho, \_, i\rangle \rightarrow_m^* \langle h, \rho[\mathtt{x}' \mapsto o], i, i+1\rangle$$

Using Lemma 16 (`hasProperty`), we obtain the proof tree for (A). Depending on whether $o = \mathsf{true}$ or $o = \mathsf{false}$, $\overline{\mathsf{c}}_a$ evaluates to $\rho'(\mathtt{x}) = l_g$ or $\rho'(\mathtt{x}) = \mathsf{undefined}$ respectively. Now we can construct the proof tree for $\wp, L, v_t \vdash \langle h, \mathcal{I}_\sigma(x)\rangle \Downarrow_m \langle h, v\rangle$ as shown in Fig. B.20 using (A) and either GIR-GLOBAL or GIR-UNDEF rule for both cases.

□

Lemma 16 (Correctness of `hasProperty`). *Let* $\mathtt{c} = \mathtt{x} := \mathtt{hasProperty}(v_t, x)$ *with* `perr`. *Then, for*

any ρ, such that $\rho \geq \emptyset[\texttt{xscope} \mapsto L, \texttt{x}_{\mathit{this}} \mapsto v_t]$:

$$\wp, L, v_t \vdash \langle h, \mathcal{I}_{hp}(x)\rangle \Downarrow_m \langle h', v\rangle \iff \mathsf{c} \vdash \langle h, \rho, \_, i\rangle \rightarrow_m^* \langle h, \rho[\texttt{x} \mapsto v], i, i+1\rangle.$$

Definition B.2 (Validity of ψ_m)**.** *We say that ψ_m is valid with respect to the current scope chain L and the heap h if the following hold*

$$\psi_m(x) = n \iff \forall i.i > n, (L_i, x) \notin \mathsf{dom}(h) \wedge (L_n, x) \in \mathsf{dom}(h)$$
$$\psi_m(x) = \bot \iff \forall i.\ i > 1, (L_i, x) \notin \mathsf{dom}(h)$$

Lemma 17 (Correctness of ψ_m)**.** *The operational semantics of JavaScript and the operational semantics of JSIL preserve the validity of ψ_m.*

C. JSIL Logic

Lemma 18 (Substitution Lemma, Expressions). *Let* $\mathit{vars}(\mathtt{e}) \subseteq \{\mathtt{x}_i \mid_{i=1}^{n}\}$. *Then:*

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

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

$$[\![\mathtt{e}_i]\!]_\rho \;=\; [\![\mathtt{x}_i]\!]_{\emptyset[\mathtt{x}_i \mapsto [\![\mathtt{e}_i]\!]_\rho \mid_{i=1}^{n}]}$$

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

Lemma 19 (Substitution Lemma, Logical Expressions). *Let* $\mathit{vars}(E) \subseteq \{\mathtt{x}_i \mid_{i=1}^{n}\}$. *Then:*

$$[\![E[\mathtt{e}_i/\mathtt{x}_i \mid_{i=1}^{n}]]\!]^\epsilon_\rho \;=\; [\![E]\!]^\epsilon_{\emptyset[\mathtt{x}_i \mapsto [\![\mathtt{e}_i]\!]_\rho \mid_{i=1}^{n}]}.$$

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

Lemma 20 (Substitution Lemma, Assertions). *Let* $\mathit{vars}(P) \subseteq \{\mathtt{x}_i \mid_{i=1}^{n}\}$. *Then:*

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

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 19, we have:

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

- Let $P \equiv (E_1, E_2) \mapsto E_3$. Then, using Lemma 19, we have:

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

□

Lemma 21 (Return Values)**.** *For any JSIL program* $\mathtt{p}$*, heaps* h *and* h_f*, stores* ρ *and* ρ_f*, identifiers* i *and* j*, value* $\mathtt{v}$*, procedure identifier* m*, the following implications hold:*

$$
\begin{array}{l}
\mathtt{p} \vdash \langle h, \rho, i, j \rangle \Downarrow_m \langle h_f, \rho_f, \mathsf{nm}\langle \mathtt{v} \rangle \rangle \implies \rho_f(\mathtt{xret}) = \mathtt{v} \\
\mathtt{p} \vdash \langle h, \rho, i, j \rangle \Downarrow_m \langle h_f, \rho_f, \mathsf{er}\langle \mathtt{v} \rangle \rangle \implies \rho_f(\mathtt{xerr}) = \mathtt{v}
\end{array}
$$

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

D. JS-2-JSIL Logic Translator

Lemma 22 (Translation of Logical Expressions - Correctness)**.** *For any variable store ρ, logical environment ϵ, value v_t, and scope chain L, it holds that:*

$$[\![\mathrm{E}]\!]^{\epsilon}_{\rho,v_t,L} = [\![\mathcal{T}_e(\mathrm{E})]\!]^{\epsilon}_{\rho[\mathtt{x}_{sc}\mapsto L,\mathtt{x}_{this}\mapsto v_t]}$$

Theorem D.1 (Assertion translation correctness)**.** *For any assertion P, abstract heap H, variable store ρ, logical environment ϵ, value v_t, and scope chain L, it holds that: $H,\rho,L,v_t,\epsilon \models P$ iff $H,\rho[\mathtt{x}_{sc}\mapsto L,\mathtt{x}_{this}\mapsto v_t],\epsilon \models \mathcal{T}_a(P)$.*

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

1. $P = \text{false}$, $\mathcal{T}_a(P) = \text{false}$. It can never be the case that either $H,\rho,L,v_t,\epsilon \models \text{false}$ or $H,\rho,\epsilon \models \text{false}$ holds. Hence, the result holds.

2. $P = \text{true}$. $\mathcal{T}_a(P) = \text{true}$. It is always the case that both $H,\rho,L,v_t,\epsilon \models \text{true}$ and $H,\rho,\epsilon \models \text{true}$ hold, from which the result follows.

3. $P = \exists X.P'$. $\mathcal{T}_a(P) \triangleq \exists X.\mathcal{T}_a(P')$. Suppose $H,\rho,L,v_t,\epsilon \models P$. We conclude (using satisfiability for JS assertions) that there is a value V, such that $H,\rho,L,v_t,\epsilon[\mathrm{X}\mapsto \mathrm{V}] \models P'$. Applying the induction hypothesis, we conclude that $H,\rho[\mathtt{x}_{sc}\mapsto L,\mathtt{x}_{\mathsf{this}}\mapsto v_t],\epsilon[\mathrm{X}\mapsto \mathrm{V}] \models \mathcal{T}_a(P')$, from which it follows that $H,\rho[\mathtt{x}_{sc}\mapsto L,\mathtt{x}_{\mathsf{this}}\mapsto v_t],\epsilon \models \mathcal{T}_a(P)$. The converse direction of the equivalence is proven analogously.

4. $P = \mathsf{emp}$. $\mathcal{T}_a(P) = \mathsf{emp}$. Suppose $H,\rho,L,v_t,\epsilon \models P$. We conclude (using the definition of satisfiability for JS assertions) that $H = \mathtt{emp}$. Using the satisfiability relation for JSIL assertions, we conclude that: $H,\rho[\mathtt{x}_{sc}\mapsto L,\mathtt{x}_{\mathsf{this}}\mapsto v_t],\epsilon \models \mathcal{T}_a(P)$. 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 $H,\rho,L,v_t,\epsilon \models P$. We conclude (using satisfiability for JS assertions) that $H = ([\![E_1]\!]^{\epsilon}_{\rho,v_t,L},[\![E_2]\!]^{\epsilon}_{\rho,v_t,L}) \mapsto [\![E_3]\!]^{\epsilon}_{\rho,v_t,L}$. Using Lemma 22, we conclude that: $H = ([\![\mathcal{T}_e(E_1)]\!]^{\epsilon}_{\rho'},[\![\mathcal{T}_e(E_2)]\!]^{\epsilon}_{\rho'}) \mapsto [\![\mathcal{T}_e(E_3)]\!]^{\epsilon}_{\rho'}$ for $\rho' = \rho[\mathtt{x}_{sc}\mapsto L,\mathtt{x}_{\mathsf{this}}\mapsto v_t]$, from which it follows that $H,\rho',\epsilon \models \mathcal{T}_a(P)$. The converse direction of the equivalence is proven analogously.

6. $P = \mathsf{emptyProps}(E_0 \mid E_1)$. $\mathcal{T}_a(P) = \mathsf{emptyProps}(\mathcal{T}_e(E_0) \mid \mathcal{T}_e(E_1))$. Suppose $H,\rho,L,v_t,\epsilon \models P$. We conclude (using satisfiability for JS assertions) that $H = \biguplus_{p\notin[\![E_1]\!]^{\epsilon}_{\rho,v_t,L}}(([\![E_0]\!]^{\epsilon}_{\rho,v_t,L},p)\mapsto\varnothing)$. Using Lemma 22, we conclude that $H = \biguplus_{p\notin[\![\mathcal{T}_e(E_1)]\!]^{\epsilon}_{\rho'}}(([\![\mathcal{T}_e(E_0)]\!]^{\epsilon}_{\rho'},p)\mapsto\varnothing)$, for $\rho' = \rho[\mathtt{x}_{sc}\mapsto L,\mathtt{x}_{\mathsf{this}}\mapsto v_t]$, from which it follows that $H,\rho',\epsilon \models \mathcal{T}_a(P)$. The converse direction of the equivalence is proven analogously.

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