Imperial College
London

MEng Individual Project

Department of Computing

Towards Automatic Verification of JavaScript Programs

Author:
Beatrix de Wilde

Supervisor:
Prof. Philippa Gardner
Second Marker:
Prof. Alessio R. Lomuscio

June 19, 2017

Abstract

JavaScript, due to its dynamic nature and complex semantics, has fewer verification tools than languages such as C and Java. In order to tackle this challenge, we introduce for the first time a bi-abductive symbolic analysis for JSIL, an intermediate language for JavaScript verification. Our analysis is scalable and can fully automatically infer specifications of JSIL procedures that are non-recursive and do not contain loops.

Bi-Abduction provides the foundation of our analysis; we give the rules of our bi-abductive symbolic execution and prove their soundness. We further present an algorithm for generating procedure specifications using these rules.

We give an implementation of our bi-abductive analysis for JSIL and use it to generate specifications for JSIL programs. Using the tool, we generate and verify specifications for some of the JavaScript internal functions. Finally, we evaluate the quality of the automatically generated specifications by comparing them with their handwritten counterparts.

Acknowledgements

First, I would like to thank my supervisor, Prof. Philippa Gardner, for her wholehearted support throughout this project.

I would also like to thank José Fragoso Santos for all his input and feedback, it has been a privilege to work with him. I am grateful to Petar Maksimović and Azalea Raad, for sharing their insight during the project.

For her inspiration and advice across my time at Imperial, I would like to thank my tutor, Prof. Susan Eisenbach.

Finally, I would like to thank my family and friends for their endless support across these past four years.

Contents

Chapter 1

Introduction

JavaScript is the developer's language of choice for Web applications and is supported by all major browsers. It is an untyped, dynamic language, with complicated semantics and notoriously confusing corner cases. Among other features, it supports extensible objects, prototype inheritance, dynamic property access, dynamic functions calls, and type coercion for delaying error reporting. These features interact in an intricate and sometimes unexpected fashion and can lead to developers unknowingly introducing bugs and vulnerabilities. For example, even simple addition can perplex the seasoned developer: adding an object to an empty string results in the number 0 rather than an error, whereas reversing the arguments leads to a completely different result and type. In particular, the JavaScript program `''+ {}` results in the string `'[object␣Object]'` whereas the program `{} + ''` results in the number 0.

In addition to the substantial economic cost resulting from vulnerabilities, program errors in medical software have led to fatalities in the past. Developers aim to reduce bugs dominantly through testing and code reviews. This requires significant developer effort and is subject to human error. Therefore program verification tools have seen a rise in the recent years. For practical applications, it is vital that these tools can scale to industrial-level codebases. Compared to other languages, such as Java, C, and C++, JavaScript has far fewer tools available for developers.

As part of an endeavour to verify JavaScript programs, the Program Specification and Verification Group at Imperial College London has developed JaVerT [35], which is a semi-automatic verification toolchain for JavaScript based on separation logic [40]. JaVerT can verify functional correctness properties of JavaScript programs annotated with preconditions and postconditions for functions, loop invariants, and folding and unfolding instructions for user-defined predicates, written in an assertion language, JS Logic. JaVerT translates JavaScript programs to the simple intermediate goto language JSIL, translates JS Logic assertions to JSIL Logic assertions, and performs verification at the level of JSIL. Correctness results have been established to prove that verification at the level of JSIL lifts to verification at the level of JavaScript.

As a first step towards fully automated verification of JavaScript programs, we aim to generate automatic specifications for JSIL procedures. Bi-Abduction [12] is a technique for automatically generating specifications through the use of separation logic inference rules. The properties that can be proven using bi-abduction are weaker than those provable semi-automatically, but bi-abduction allows verification tools to scale to industry-grade code bases.

Motivation

There are two key motivations for having a fully automatic tool: to automatically verify general properties and to support the developer in verifying functional properties of JavaScript programs.

Specifications provide the footprint of a program. Therefore, using these specifications it is possible to automatically verify properties of programs. In particular, verification of confinement properties is of interest for JavaScript programs. Confinement involves preventing untrusted code from modifying or accessing security-sensitive resources. As web applications are usually composed of many programs coming from different origins executing in the same global context, it would be beneficial to automatically verify whether sensitive resources get tainted or leaked.

In addition to automatic verification of general properties, generating specifications can support developers in verifying functional properties. As it can be taxing for developers to manually annotate code and write specifications for a large codebase, automatically generating specifications allows the developer to annotate an interesting subset of procedures or adapt generated specifications to verify more expressive functional properties.

Contribution

We formalise the symbolic analysis used for the semi-automatic JaVerT tool in a way that closely follows the implementation of the tool (Section 3). In particular, and in contrast to a previous formalisation of JSIL logic [35], our analysis is specified in an algorithmic fashion, meaning that it can be straightforwardly implemented and serve as a basis for the bi-abductive analysis we are aiming at.

Building upon our symbolic analysis for JSIL, we introduce a bi-abductive symbolic analysis that generates verifiable specifications for JSIL procedures (Section 4). We establish the soundness of both analyses. Furthermore, we present an algorithm, that given a JSIL program, infers the specifications of its procedures. Our analysis is a first step to a fully-fledged bi-abudctive analysis for JSIL. In particular, we do not support the inference of loop invariants and specifications of recursive procedures.

Finally, we introduce the tool, JSIL Abduce, that implements the bi-abductive symbolic analysis for JSIL. The tool is able to produce procedure specifications for a JSIL program (Section 5). We evaluate the tool by using JSIL Abduce to automatically generate specifications for the JSIL implementations of the JavaScript internal functions (Section 6). We check that the generated specifications are correct by checking that they are satisfied by their corresponding implementations, using the existing verification tool for JSIL, JSIL Verify [35]. We analyse the quality of the bi-abductive analysis by comparing the inferred specifications with the prior handwritten specifications.

In the conclusion (Section 7) we discuss how our contributions can be used to achieve fully automatic verification of JavaScript as well as possible applications of such an analysis.

Chapter 2

Background

First, we provide a background in program verification and static analysis techniques (Section 2.1). We give a summary of related work in this area. We then focus on separation logic, bi-abduction and a range of program verification tools based on separation logic (Section 2.2). Additionally, we give high level details of the JavaScript language (Section 2.3).

Finally, we discuss work by the Program Specification and Verification Group at Imperial College London in JavaScript verification. In particular, we describe the JavaScript intermediate language (Section 2.4.2) and the JavaScript verification tool chain (Section 2.4.3).

2.1 Program Verification

2.1.1 Program Correctness

Program correctness is vital, not only to safety-critical code, but to any industrial level system. The impact of avoidable bugs is evident in today's society, with recent vulnerabilities such as a bug in Microsoft Windows [34] which was exploited by the ransomware WannaCry [41]. Affecting over 150 countries, the ransomware encrypted the files of the infected computers requesting a ransom in Bitcoins. The impact of software bugs can be catastrophic and even lead to loss of lives, as shown by a bug in Therac-25, a radiation therapy machine, which led to the death of six patients from radiation overdose [28].

In practice, developers ensure the correctness of code through testing and code reviews. This is at the expense of developer time and is subject to human error. Dijkstra noted "Testing shows the presence, not the absence of bugs" [9]. Therefore, the effectiveness of testing is based on the coverage of the test suite, which, in turn, depends on the manual effort of the developer. Increasingly, developers are turning to automatic verification tools to analyse the correctness of programs and support them in the development process. Program correctness can be split into verifying two types of properties: functional and general.

Verifying functional properties requires checking that a program meets a specification [27]. Program specifications typically consist of a precondition and postcondition. The precondition outlines the state required before the code is run and the postcondition specifies the state which holds after execution. We give an example specification in Figure 2.1. The program increments the i-th element of the array. The precondition states that the i-th element must equal some value `u` and the postcondition states the i-th element must equal to `u+1` after executing the program.

```
\\ Precondition: arr[i] = u
arr[i] += 1;
\\ Postcondition: arr[i] = u + 1
```

Figure 2.1: Specification example

On the other hand, verifying general properties involves proving whether a program will produce a certain type of error, such as an integer overflow or a null pointer exception. In the example in Figure 2.1, a general property which may be checked is whether the index i is in the bounds of the array. General properties are often easier to automatically verify as they require less developer input.

Generally, verification is split into static and dynamic analysis [18]. Static analysis involves evaluating the program without running the code, whereas dynamic analysis requires an environment in which the program is run. Typically, dynamic analysis [2, 36] is scalable as it only explores realistic paths. However, dynamic analysis requires the entire execution environment and often the entire codebase, therefore it is hard to analyse isolated sections of the system. Developer tests, such as unit tests, can be considered a form of dynamic analysis. In comparison, static analysis is able to explore more execution paths, gaining much higher coverage. Additionally, if the analysis is performed in a modular style, then it is able to analyse incomplete sections of code.

2.1.2 Static Verification Techniques

We will now explore the main techniques used in static analysis tools.

SAT and SMT solvers are required for many backends of static analysis tools [15, 3, 8]. Satisfiability (SAT) solvers take a boolean formula and try to solve for a variable assignment where the formula evaluates to true. Satisfiability modulo theories (SMT) solvers are similar to SAT solvers; however, they take a formula in the first-order logic with equality. Competitions are run in order to encourage advancement in SAT and SMT solvers [32, 30].

Model checking is a lightweight technique to check whether the program satisfies a property instead of verifying the entire system [14, 22]. Given a model, M, which is a representation of a system and a temporal logic formula, ϕ, relating to the property to be verified, then we check that the model satisfies the formula. Formally, $M \models \phi$.

Bounded model checking (BMC) takes a program transition system and unrolls the loops to a predefined depth [6, 13]. A SAT solver can then check whether there is a property violation up to that depth. BMC is an under-approximation and can scale for large programs. The developer need only provide a correctness property.

Symbolic execution evaluates program instructions on symbolic values instead of concrete values [25]. Symbolic values represent an entire range of values that a variable can take, and get constrained by conditions resulting from the program path covered so far. Therefore, symbolic execution analysis does not need to perform the infeasible task of exhaustively testing all possible concrete input values. Additionally, symbolic execution has the ability to produce concrete test cases, and is therefore usually the basis of many testing tools.

In order to illustrate symbolic execution of a program, we give a snippet of code for a binary search algorithm in Figure 2.2. The code calculates the middle of the upper and lower bounds, then it checks if the search key is at that index. Otherwise, it checks if the search key is smaller or greater than the middle index and returns.

```
index = (lower_index + upper_index) / 2;

if (arr[index] = key) {
   return index;
} else if (arr[index] < key) {
   lower_index = index + 1;
} else {
   upper_index = index - 1;
}
```

Figure 2.2: Code Snippet for Binary Search

We give the symbolic execution graph for this code snippet in Figure 2.3. Branching of the symbolic execution occurs when an `if` statements is reached, the path conditions are recorded for each branch. For the branch, where the search key is greater than the middle index, the path conditions not (arr[index] = key) and not (arr[index] < key) are combined to the single path condition arr[index] > key. It was discovered that almost all binary searches contain a bug [24], if the code snippet given in Figure 2.2, is written in Java then the first line contains a bug. An integer overflow occurs if the sum of lower_index and upper_index exceeds the maximum positive value for type int.

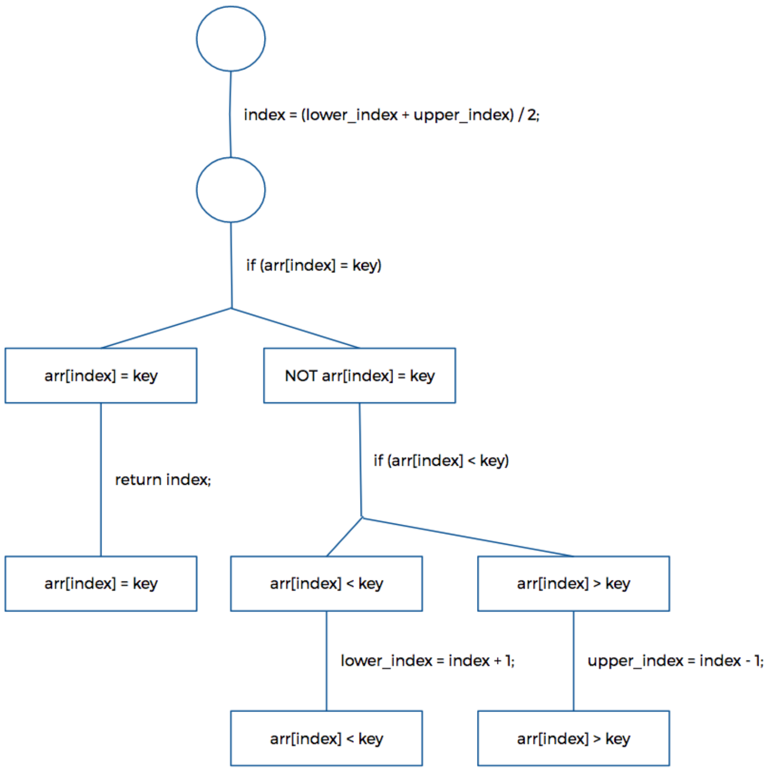

Figure 2.3: Symbolic Execution Graph of Code Snippet for Binary Search

2.1.3 Related Tools

KLEE [10], a symbolic execution tool, automatically generates tests for the intermediate language LLVM. The tool uses environment models written in C in order to stub system calls. The core KLEE engine uses the STP constraint solver [31] to solve branch conditions and search heuristics, random path selection and coverage-optimised search to decide which paths to explore. The focus of KLEE is on general properties and it succeeds in finding many bugs, mostly memory-related errors, in well established-code.

Symbolic PathFinder (SPF) [38, 39] focuses on the verification of Java. The tool symbolically executes the intermediate language Java ByteCode. This symbolic execution is built on top of a model checking tool, Java PathFinder [42]. When branching, the path conditions are checked by constraint solvers. The tool uses three constraint solvers, Choco [1], CVC3 [33] and IASolver [20], for different types of constraints. JUnit test cases can be generated by the tool.

CBMC [26] is a bounded model checker for annotated C and C++ programs. First, it translates the program into an intermediate goto language. It then performs symbolic execution unrolling the loops to a predefined depth. It passes the resulting formulae, corresponding to paths though the program control flow graph, into the SAT solver MiniSat 2.2.0 [17]. CBMC can be run with increasing unrolling depths, in order to reduce false negatives.

2.2 Separation Logic and Tools

2.2.1 Hoare Logic

Hoare logic, a system to reason about the correctness of programs, was developed by Tony Hoare in the 1960s [21]. Hoare Triples describe the connection between a program's precondition P, the code C, and postcondition Q. A Hoare Triple, $\{P\}\ C\ \{Q\}$, can be interpreted as "if the precondition P holds, then after executing the program C, the postcondition Q will hold". Proof of a Hoare Triple $\{P\}\ C\ \{Q\}$ gives partial correctness. In order to establish total correctness, termination of the Hoare Triple must also be proven. Although Hoare Logic is capable of reasoning about programs that alter the variable state, extending it to account for heap state leads to issues with scalability when it comes to reasoning about programs.

2.2.2 Separation Logic

Separation Logic [40] extends Hoare Logic in order to scalably reason about the correctness of programs with heap state. It extends the formulae of predicate calculus with assertions and connectives to describe the heap. Four new constructs are added to describe the heap: the empty heap `emp`; the singleton heap $e_1 \mapsto e_2$; the separating conjunction $a_1 * a_2$; and the separating implication $a_1 -\!\!* a_2$.

The assertion `emp` asserts the heap is empty. The assertion $e_1 \mapsto e_2$ asserts the heap has exactly one cell, with the address given by the value of the expression e_1 and the contents given by the value of the expression e_2. The assertion $a_1 * a_2$ asserts the heap can be split into two disjoint parts, where one part satisfies a_1 and the other satisfies a_2. The assertion $a_1 -\!\!* a_2$ asserts if the heap is extended with a part where a_1 holds, then in the resulting heap a_2 holds.

The Frame Rule, given in Figure 2.4, allows reasoning about portions of the heap that are affected by the program code C, leaving the framed section R of the heap unmodified. This allows for local reasoning about the heap.

$$\textsc{Frame}\quad \frac{\{P * R\}\ C\ \{Q * R\}}{\{P\}\ C\ \{Q\}}$$

where no variable occurring free in R is modified by C.

Figure 2.4: Frame Rule

2.2.3 Bi-Abduction

Abductive inference, described by Charles Peirce [37], finds the most likely explanation for an observation. Notably adopted by the fictional detective Sherlock Holmes in the works of Sir Arthur Conan Doyle, Holmes is able to solve cases by abducing important crime information from clues. Calcagno et al. [12] proposed using abduction in order generate program specifications.

Formally, abductive inference is when, given an assumption A and an observation O, we abduce the missing assumption M. In first-order logic, this is presented by the entailment $A \wedge M \vdash O$. This problem has been adapted for separation logic and is presented by the entailment $A * M \vdash O$. This entailment states that given two spatially disjoint assumptions, A and M, we are able to entail the observation O.

Bi-Abduction [12] generalises the abductive inference problem. It solves the following question: Given an assumption A and an observation O, what is the required missing assumption M and additional untouched conclusion F not captured by the observation O? Formally, the problem can be given by the entailment $A * M \vdash O * F$. We call the missing state M the anti-frame and F the frame axiom.

Upon calling a procedure in a program, we are able to use bi-abduction to solve for both the additional state required before calling the procedure and any additional state not touched by the procedure call. Given the precondition of the procedure being called and the current state we solve the problem $current\ state * M \vdash precondition * F$.

We give an example of a simple addition procedure add(x,y) which adds the two inputs x and y together. The precondition of the procedure is $\{\ x \mapsto a * y \mapsto b\ \}$ and the postcondition is $\{\ x \mapsto a * y \mapsto b * z \mapsto a + b\ \}$. When calling this procedure in a state $\{\ x \mapsto 4 * w \mapsto 2\ \}$ we pose the bi-abduction question:

$$x \mapsto 4 * w \mapsto 2 * \text{ ?M } \vdash x \mapsto a * y \mapsto b * \text{ ?F}$$

A possible solution is $?M = y \mapsto b * a \doteq 4$ and $?F = w \mapsto 2$, where $?M$ is required for the precondition and $?F$ is untouched by the precondition.

2.2.4 Related Tools

Separation logic allows verification tools to reason about programs at a larger scale. We discuss some of these tools.

An initial experiment into whether hand written separation logic proofs could work in an automatic environment was explored with the tool Smallfoot [4]. Smallfoot is a a semi-automatic tool, as annotations for preconditions and postconditions are required. A specially designed input language was developed for the tool. The assertion language provides inductive predicates for trees and lists. They note the potential for symbolic execution in automating separation logic proofs.

Expanding on Smallfoot, Space Invader [43] adds a widening operator to the symbolic execution in order to guarantee termination of fixed-point calculations. These calculations are performed in the semantics of while loops. Therefore, the tool is able to find and then prove while loop invariants.

Similar to Space Invader, SLAyer [5], builds upon Smallfoot's assertion language. The tool verifies C programs and targets memory safety errors. No annotations are required. The tool's approach to loop invariants is similar to Space Invader. The Z3 SMT solver [15] is used to reason about pure formulae. SLAyer is able to verify sizeable code bases, including Windows device drivers.

Abductor [12], goes one step further by generating specifications and is, therefore, fully automatic and able to scale to larger programs. Bi-Abduction is used to generate preconditions for procedures. The specifications Abductor generates are small specifications, as they only contain the footprint of the procedures.

A specification table is gradually built up as the functions are split into partitions where every function is in a lower or the same partition with respect to its caller. The tool starts by inferring the specifications of the functions in the first partition. Then, it moves on to inferring the specifications of the functions in the second partition. The tool continues through all the partitions, where in each partition it uses the specifications already generated from the lower partitions.

Abductor was developed further into the industrial level tool Infer [11]. The compositional nature of the tool allowed it to properly scale. This is shown through the ability for Infer to be integrated into the Facebook development cycle. Initially aimed at C, the tool has also been expanded to Java.

Verifast [23] is a semi-automatic verification tool based on separation logic. The tool requires annotations for preconditions and postconditions, but not for loop invariants. It can take programs written in C and Java and supports folding and unfolding of user defined predicates. The toolchain JaVerT [35] is inspired by Verifast, but targets JavaScript verification and is discussed in Section 2.4.3.

2.3 The JavaScript Language

JavaScript is the predominant language for Web developers. It is conventionally coupled with CSS and HTML for the writing of client-side web applications. Untyped, dynamic-natured and supporting concepts from multiple programming paradigms, JavaScript makes the detection of vulnerabilities difficult. In addition to the nature of the language, most applications include external libraries such as jQuery, React, and Async, as well as other third-party code. Having multiple scripts from different sources running in the same global environment can easily lead to leaking sensitive information.

JavaScript is standardised by the ECMAScript Committee, which provides and regularly updates the ECMAScript Language Standard. The standard is currently in its sixth edition (ES6). The standard also defines a strict mode of the language, which improves error reporting and has better behavioural properties. For this project, we will be working with the strict mode of the fifth edition of the standard (ES5 Strict) [16], as JaVerT is targeting ES5 Strict. There have been several attempts at formalising the semantics of JavaScript. Small-step operational semantics for ECMAScript have been defined by Maffeis et al [29] for ES3. Drawing inspiration from that work, JSCert [7] provides a formalisation of ES5 semantics in the Coq theorem prover.

JavaScript Objects. As defined in the ECMAScript specification, objects in JavaScript are collections of properties. There are two types of object properties: *internal* and *named*. Internal properties capture the inner workings of the language, such as prototype inheritance and object extensibility, and are not available to the programmer. Named properties can be either *data properties* or *accessor properties* and, unlike in C++ or Java, they are not associated with values, but rather with *property descriptors*. Property descriptors are lists of *attributes*, which describe the ways in which a property can be accessed and/or modified. Depending on the attributes they contain, named properties can either be *data properties* or *accessor properties*. Data properties contain the value, writable, enumerable, and configurable attributes (denoted by `[V]`, `[W]`, `[E]`, and `[C]`), whereas accessor properties contain get and set attributes (denoted by `[G]` and `[S]`), as well as `[E]` and `[C]`. The attributes have the following semantics: `[V]` holds the actual value of the property; `[W]` describes whether or not the property's value can be changed; `[E]` indicates whether or not the property will be included in a for-in enumeration; `[C]` allows or disallows any change to the other attributes (except for value, which it does not affect), as well as any change in the type of the property (data to accessor and vice versa); `[G]` and `[S]` play a role similar to getters and setters of Java and provide property encapsulation.

JavaScript Initial Heap. Before the execution of any JavaScript program, an initial heap must be established, as described in Chapter 15 of the ECMAScript standard. It must contain a unique global object as well as the constructors and prototypes of all JavaScript built-in libraries, such as Object, Function, Array, and String, which are widely used by JavaScript programmers.

The global object is critical for the correct functioning of JavaScript programs, as it holds: all global variables; value properties such as `NaN`, `infinity` and `undefined`; and function properties such as `eval`, which executes the supplied string argument containing arbitrary JavaScript code. Some properties of the global object are fixed; it is, for instance, not possible to reassign to `undefined`, `Array`, or `String.prototype`. It is, however, possible to reassign to `eval`, as shown in figure 2.5, which may introduce serious vulnerabilities. It is also possible to perform *prototype poisoning* by, for instance, adding or overriding the properties in the built-in object prototypes, thereby altering their functionality and possibly gaining access to sensitive information.

```
eval('2+2') // returns 4
// alters the global objects eval function
this.eval = new Function('return␣this;')
// which now returns the global object
eval('2+2') // returns {global: ...}
```

Figure 2.5: Altering the `eval` property of the global object

In addition to built-in library functions, JavaScript has internal functions not accessible to developers. These functions provide core functionality, including: type conversion methods such as toPrimitive and toBoolean; prototype chain traversal methods such as getProperty; equality comparison methods; and the methods getValue and putValue.

2.4 The JavaScript Intermediate Language

A separation logic (JS Logic) was developed for a subset of JavaScript, however it is only able to model a simplified JavaScript heap and is difficult for automation purposes. Therefore, an intermediate goto language, JSIL, was developed. JSIL, is able to overcome verification difficulties with JavaScript and JS Logic. The toolchain, JaVerT, then compiles JavaScript programs and JS logic annotations to JSIL and verifies the JSIL programs.

In section 2.4.1 we discuss the complexities involved with JavaScript and the separation logic developed for JavaScript. In section 2.4.2 we discuss the JSIL language and finally in section 2.4.3 we discuss the toolchain, JaVerT.

2.4.1 Motivation for JSIL

A separation logic (JS Logic) has been developed for a subset of JavaScript by Gardner et al. [19], built upon a big-step operational semantics of ES3. JS Logic simplifies the memory model and targets only a fragment of JavaScript. Nonetheless, it is very complex, as illustrated by the function call rule in figure 2.6, which captures the JavaScript function call.

$$
\begin{array}{l}
\text{(Function Call)} \\
\{P\}\mathsf{e1}\{R_1 * \mathbf{r} \doteq F_1\} \\
R_1 = \left(\begin{array}{l} S_1 \uplus \mathsf{This}(F_1, T) \uplus \gamma(Ls_1, F_1, F_2) * \\ (F_2, @body) \mapsto \lambda X.\mathsf{e3} * (F_2, @scope) \mapsto Ls_2 \end{array} \right) \\
\{R_1\}\mathsf{e2}\{R_2 * \mathbf{l} \doteq Ls_3 * \mathbf{r} \doteq V_1\} \quad R_2 = S_2 * \gamma(Ls_4, V_1, V_2) \\
R_3 = \left(\begin{array}{l} R_2 * \exists \mathrm{L}.\, \mathbf{l} \doteq \mathrm{L}{:}Ls_2 * (\mathrm{L}, X) \mapsto V_2 * \\ (\mathrm{L}, @this) \mapsto T * \\ (\mathrm{L}, @proto) \mapsto \mathtt{null} * \mathsf{defs}(X, \mathrm{L}, \mathsf{e3}) * \\ \mathsf{newobj}(\mathrm{L}, @proto, @this, X, \mathsf{decls}(X, \mathrm{L}, \mathsf{e3})) \end{array} \right) \\
\{R_3\}\mathsf{e3}\{\exists \mathrm{L}.\, Q * \mathbf{l} \doteq \mathrm{L}{:}Ls_2\} \qquad \mathbf{l} \notin \mathsf{fv}(Q, R_2) \\
\hline
\{P\}\mathtt{e1(e2)}\{\exists \mathrm{L}.\, Q * \mathbf{l} \doteq Ls_3\}
\end{array}
$$

Figure 2.6: Function Call Rule in JS Logic [19]

JS Logic assertions feature standard boolean assertions and operators on expressions, the separating conjunction $*$ and implication $-\!\!*$, and the heap cell assertion $(E_1, E_2) \mapsto E_3$, which asserts that the object at location E_1 has field E_2 with value E_3 (or has no field E_2, if E_3 equals $\varnothing$). It also features a non-standard separation logic connective, $P \uplus Q$ (read: sepish), which describes a heap that can be split into two parts that do not need to be disjoint. This connective, as well as the $-\!\!*$ separating implication are known to be very difficult for automation. Expanding JS Logic, as formulated in [19] to the full memory model of JavaScript and automating it is, therefore, not feasible.

2.4.2 The JavaScript Intermediate Language

The JSIL language is simpler compared to JavaScript. It does not contain dynamic function calls, also conditional branches and loops are performed by goto commands and commands are explicit with no corner cases. Additionally, the separation logic for JSIL (JSIL Logic) is comparatively simpler than JS Logic: it does not include the $\uplus$ connective or the separating implication $-\!\!*$ and therefore can be easily automated. Additionally, reasoning in JSIL Logic does not require the simplification of the JavaScript memory model.

JSIL Syntax. The syntax of JSIL is given in Figure 2.7. A JSIL program $\mathtt{p} \in \mathtt{P}$ is a set of top-level procedures. Procedures consist of three elements: a procedure name, $\mathtt{m} \in \mathcal{S}tr$; a list of parameters, $\overline{\mathtt{x}}$ and a list of commands, $\overline{\mathtt{c}}$. The list of commands is numbered and we use the notation $\mathtt{p}_{\mathtt{m}}(i)$ to refer to the i-th command of procedure $\mathtt{m}$ in program $\mathtt{p}$. JSIL does not provide an explicit return command; a procedure terminates when it reaches one of two dedicated indexes, i_{nm} and i_{er}. When the i_{nm}-th command is reached, the procedure returns normally and returns the value of the dedicated variable `xret`; when the i_{er}-th command is reached, the procedure returns an error and returns the value of the dedicated variable `xerr`. JSIL commands are divided into basic commands, which do not affect the control flow of programs, and control flow commands, which do affect it.

JSIL basic commands include support for object creation, variable assignment and field manip-

Strings: $m \in \mathcal{S}tr$ Numbers: $n \in \mathcal{N}um$ Booleans: $b \in \mathcal{B}ool$ Locations: $l \in \mathcal{L}$
Variables: $\mathtt{x} \in \mathcal{X}_{\mathtt{JSIL}}$ Literals: $\lambda \in \mathcal{L}it ::= n \mid b \mid m \mid \mathsf{undefined} \mid \mathsf{null}$

$$\text{Types} : \mathtt{t} \in \mathtt{Types} ::= \mathtt{Num} \mid \mathtt{Bool} \mid \mathtt{Str} \mid \mathtt{Undef} \mid \mathtt{Null} \mid \mathtt{Empty} \mid \mathtt{Obj} \mid \mathtt{List} \mid \mathtt{Type}$$
$$\text{Values} : \mathtt{v} \in \mathcal{V}_{\mathtt{JSIL}} ::= \lambda \mid l \mid \mathsf{empty} \mid \mathsf{error} \mid \mathtt{t} \mid \overline{\mathtt{v}}$$
$$\text{Expressions} : \mathtt{e} \in \mathcal{E}_{\mathtt{JSIL}} ::= \mathtt{v} \mid \mathtt{x} \mid \ominus\, \mathtt{e} \mid \mathtt{e} \oplus \mathtt{e} \mid \mathsf{typeOf}\,(\mathtt{e}) \mid \overline{\mathtt{e}} \mid \mathsf{nth}\,(\mathtt{e}, \mathtt{e})$$

Basic Commands:

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

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

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

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

Figure 2.7: JSIL Syntax

ulation including deletion, lookup and assignment. Additionally, there is support to check if an object has a field and to get all the fields of an object. The notation $[\mathtt{e}_1, \mathtt{e}_2]$ denotes the field $\mathtt{e}_2$ of object $\mathtt{e}_1$.

JSIL control flow commands use numbered command labels to transfer control to other commands in the current procedure. The unconditional goto command transfers control to the command labelled i. The conditional goto command transfers control to the command labelled i if the expression $\mathtt{e}$ evaluates to true; otherwise, control shifts to the command labelled j. The procedure call command obtains the procedure name and the arguments by evaluating, respectively, the expression $\mathtt{e}$ and the list of expressions $\overline{\mathtt{e}}$. The result of the procedure call is assigned to the variable $\mathtt{x}$. If the program raises an error, control is transferred to the j-th command otherwise, control is transferred to the following command. The phi-assignment command assigns an element of the list $\overline{\mathtt{x}}$ to the variable $\mathtt{x}$. Each variable in the list $\overline{\mathtt{x}}$ relates to a path taken to the current command. If there are n variables in the list $\overline{\mathtt{x}}$ then there are n paths to the current command. If the i-th path was taken then the i-th element of $\overline{\mathtt{x}}$ is assigned to $\mathtt{x}$.

JSIL expressions include JSIL values, JSIL variables, lists of expressions and operators on expressions. There are various unary and binary operators that can be applied to JSIL expressions as well as the $\mathsf{typeOf}\,(\mathtt{e})$ operator which returns the type of an expression and the $\mathsf{nth}\,(\mathtt{e}_1, \mathtt{e}_2)$ operator which returns the $\mathtt{e}_2$-th element of the list $\mathtt{e}_1$.

In order to illustrate some aspects of JSIL, we give the example below, which demonstrates a JSIL program with one procedure that attempts to retrieve an interval value from a timeout object. In the program, "else", "then" and "rlab" are command labels. The basic command hasField checks if the timeout object has the field interval and assigns the result to variable x. Then, the control flow command goto transfers control to the command labelled "then" if the variable x evaluated to true, i.e. if the field interval exists; otherwise, it transfers control to the command labelled "else".

The basic command at the label "then" assigns the timeout interval to the variable y, control is then transferred to the unconditional goto which jumps to the return label rlab. The program then terminates with the return variable y.

The basic command at the label "else" assigns the default timeout interval to the variable y, control is then transferred to the skip command which again terminates with the return variable y.

```
proc getTimeoutInterval (timeout, defVal) {
          x := hasField(timeout, "interval");
          goto [x] then else;
  then:   y := [timeout, "interval"];
          goto rlab;
  else:   y := defVal;
  rlab:   skip
} with {ret: y, rlab;};
```

JSIL Semantics. The JSIL memory model includes:

- A JSIL store, $\rho \in \mathcal{S}to : \mathcal{X}_{\texttt{JSIL}} \rightharpoonup \mathcal{V}_{\texttt{JSIL}}$, which maps variables to values.
- A JSIL heap, $\texttt{h} \in \mathcal{H}_{\texttt{JSIL}} : \mathcal{L} \times \mathcal{X}_{\texttt{JSIL}} \rightharpoonup \mathcal{V}_{\texttt{JSIL}}$, which maps locations and variables to values.

Each JSIL procedure is executed in an individual store. The semantics of JSIL expressions are denoted by the judgement $[\![\texttt{e}]\!]_\rho = \texttt{v}$, where evaluating the expression e with respect to the store ρ results in the value v.

The semantics of JSIL basic commands are denoted by the judgement $[\![\texttt{bc}]\!]_{h,\rho} = (\texttt{h}', \rho', \texttt{v})$ where evaluating the basic command bc with respect to the store ρ and heap h results in the value v, store ρ' and heap $\texttt{h}'$. The semantics of JSIL control flow commands are denoted by the judgement $\texttt{p} \vdash \langle h, \rho, i, j\rangle \Downarrow_{\texttt{m}} \langle h', \rho', \texttt{v}\rangle$ where starting from the i-th command in procedure m in program p with heap h and store ρ, where we have come from the j-th command, results in the heap $\texttt{h}'$, store ρ' and value v. The full semantics for JSIL are given in Appendix A.0.2.

JSIL Logic Assertions. JSIL logic assertions are described in Figure 2.8. JSIL assertions include existential quantification, heap assertions, and pure boolean assertions, excluding disjunction. The JSIL heap assertions consist of the separating conjunction, the empty heap, the heap cell $(E_1, E_2) \mapsto E_3$ where the field E_2 in object E_1 maps to the value E_3 and the empty fields assertion $\mathsf{emptyFields}(E \mid \overline{E})$ where the object E can only possibly have the fields $\overline{E}$.

$$
\begin{array}{rcl}
\textsc{Logical Values} : V \in \mathcal{V}^L_{\texttt{JSIL}} & \triangleq & \texttt{v} \mid \varnothing \mid \overline{V} \\
\textsc{Logical Expressions} : E \in \mathcal{E}^L_{\texttt{JSIL}} & \triangleq & V \mid \texttt{x} \mid \mathrm{X} \mid \ominus E \mid E \oplus E \mid \overline{E} \\
\\
\text{JSIL } \textsc{Assertions} : P \in \mathcal{AS}_{\texttt{JSIL}} & \triangleq & \mathsf{true} \mid \mathsf{false} \mid \neg P \mid P \wedge P \mid \exists \mathrm{X}.P \quad \textsc{Pure Boolean} \\
 & & \mid E = E \mid E \leq E \quad \textsc{Equalities} \\
 & & \mid \mathsf{emp} \mid (E, E) \mapsto E \quad \textsc{Heap} \\
 & & \mid P * P \mid \mathsf{emptyFields}(E \mid \overline{E})
\end{array}
$$

Figure 2.8: JSIL Logic Assertions

JSIL logical expressions used in JSIL assertions are an extension of JSIL expressions with logical variables and the value $\varnothing$. The special value $\varnothing$ is used with the heap cell assertion $(E_1, E_2) \mapsto \varnothing$ to denote that the object E_1 does not have the field E_2. The semantics of JSIL logical expressions are similar to the semantics of JSIL expressions. The judgement $[\![E]\!]^\epsilon_\rho$ describes the evaluation of the expression E with respect to the store ρ and logical environment ϵ. Logical values consist of JSIL values and the special value $\varnothing$.

In order to illustrate some aspects of JSIL assertions, we give the example below, which is a JSIL assertion describing one object whose location in the heap is l. This object has three fields, interval, function, and @proto, described by the first three heap cell assertions. There, we see that the value of interval is the value of variable time, that the value of function is undefined, and that the value of @proto is null. The emptyFields assertion states that the only fields that the object has are interval, function and @proto. The equality assertion states that the variable time has value 400.

$$\left\{\begin{array}{l}(l, \mathsf{interval}) \mapsto \mathsf{time} \; * \; (l, \mathsf{function}) \mapsto \mathsf{undefined} \; * \; (l, \mathsf{@proto}) \mapsto \mathsf{null} \; * \\ \mathsf{time} = 400 \; * \; \mathsf{emptyFields}(l \mid \{\{\mathsf{interval}, \mathsf{function}, \mathsf{@proto}\}\})\end{array}\right\}$$

JSIL abstract heaps $H \in \mathcal{H}^{\emptyset}_{\texttt{JSIL}} : \mathcal{L} \times \mathcal{S}tr \rightharpoonup \mathcal{V}^{L}_{\texttt{JSIL}}$ are an extension of JSIL heaps with the special value $\varnothing$. An abstract heap is well-formed, if all objects have a non-none $@proto$ field. Formally, $\forall H.\ \mathsf{wf}(H) \Leftrightarrow (\forall l.\ (l, -) \in \mathsf{dom}(H) \Rightarrow \exists \mathrm{V}.\ V \neq \varnothing \ \wedge\ (l, @proto) \mapsto \mathrm{V} \in H)$. A JSIL logical environment $\epsilon \in \mathcal{E}nv_{\texttt{JSIL}} : \mathcal{X}^{L}_{\texttt{JSIL}} \rightharpoonup \mathcal{V}^{L}_{\texttt{JSIL}}$ is a mapping from logical variables to logical values.

A JSIL assertion $P \in \mathcal{AS}_{\texttt{JSIL}}$ is satisfiable if $H, \rho, \epsilon \models P$ where H is an abstract heap, ρ is a JSIL store and ϵ is a logical environment. The satisfiability relation of JSIL assertions is given in Figure 2.9.

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

Figure 2.9: JSIL Logical Assertions (Satisfiability Relation)

2.4.3 The JavaScript Verification Toolchain

The JavaScript Verification Toolchain (JaVerT), shown in Figure 2.10, verifies JavaScript programs annotated with JS Logic specifications. It is based on an infrastructure which has three components: **(1)** JS-2-JSIL, the compiler from JavaScript to JSIL; **(2)** JSIL Verify, a semi-automatic tool for verifying JSIL code annotated with JSIL logic specifications; and **(3)** JSIL implementations and verified specifications of the JavaScript internal functions. JaVerT works in the following way: first, JS-2-JSIL complies JavaScript to JSIL and translates JS logic annotations to JSIL logical annotations. Then the semi-automatic tool JSIL Verify verifies the compiled JSIL programs with respect to the corresponding JSIL logic annotations, using the JSIL logic specifications of the JavaScript internal functions.

Annotations for JSIL programs are JSIL logic assertions. Annotations are given for procedure specifications which include pre- and postconditions. Additionally, annotations for loop invariants, and folding and unfolding user-defined predicates can be written using JSIL logic.

JS-2-JSIL compiles JavaScript code to JSIL code, line-by-line following the ECMAScript English standard. In order to show this correspondence, as well as how the use of the internal functions in the standard is reflected in compiled JSIL code, we give a snippet of JSIL code which is compiled from the JavaScript program `0 == ''`.

As described in the standard, the abstract equality operator first calls `getValue` on both arguments. Then, it follows the abstract equality comparison algorithm. This algorithm is captured by the i__abstractEquality internal function.

The ECMAScript standard notes in the algorithm that "*If Type(x) is Number and Type(y) is String, return the result of the comparison x == ToNumber(y)*". Therefore the result of this program is `true`, as converting the empty string to a number results in the value 0. The equality comparison is the source of much confusion amongst JavaScript developers. It is recommended to use the strict equality operator (===) rather than the equality operator (==).

```
(* The setupInitialHeap function creates the initial heap,
   which is established before the execution of any JavaScript program *)
x_0 := "setupInitialHeap"();
x_sc_0 := {{ $lg }};
(* Set this to the global object location *)
x__this := $lg;
(* Get value of first argument *)
x_1_v := "i__getValue"(0.) with elab;
(* Get value of second argument *)
x_2_v := "i__getValue"("") with elab;
(* Perform abstract equality comparison algorithm *)
x_3 := "i__abstractEquality"(x_1_v, x_2_v) with elab;
(* Get the value of the comparison result *)
x_3_v := "i__getValue"(x_3) with elab
xret := x_3_v;
```

JSIL Verify is a semi-automatic verification tool for JSIL. It is based on a symbolic execution engine and an entailment engine. The entailment engine has two components: one for resolving spatial entailments and the other for pure entailments. Spatial entailments are handled by JSIL Verify, whereas the pure ones are delegated to the Z3 SMT solver. When verifying compiled JavaScript code, the JSIL logic specifications for the internal functions are imported before the symbolic execution begins.

Correctness of JaVert. The correctness of JaVert is given by systematic testing of JS-2-JSIL, the soundness result for JSIL Logic and the correctness result for the translation of assertion between JS Logic and JSIL Logic. JS-2-JSIL is tested against the ECMAScript ES6 Test262 test suite, where it passes all 8797 relevant tests. Additionally, the JavaScript internal functions implemented in JSIL are step-by-step faithful to the standard and their specifications are verified by JSIL Verify.

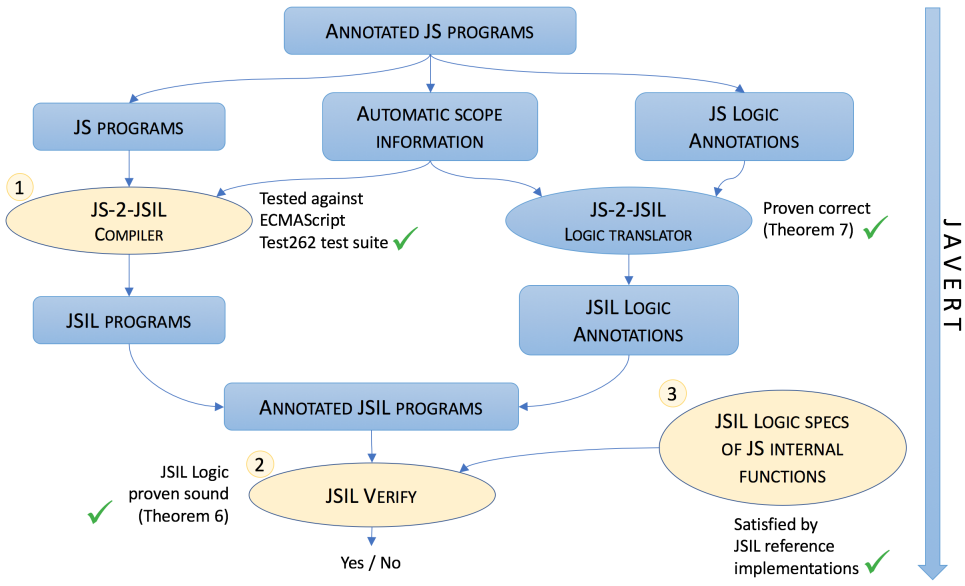
ANNOTATED JS PROGRAMS
JS PROGRAMS
AUTOMATIC SCOPE INFORMATION
JS LOGIC ANNOTATIONS
1
JS-2-JSIL COMPILER
Tested against ECMAScript Test262 test suite ✓
JS-2-JSIL LOGIC TRANSLATOR
Proven correct (Theorem 7) ✓
JSIL PROGRAMS
JSIL LOGIC ANNOTATIONS
ANNOTATED JSIL PROGRAMS
3
JSIL LOGIC SPECS OF JS INTERNAL FUNCTIONS
2
JSIL VERIFY
JSIL Logic proven sound (Theorem 6) ✓
Yes / No
Satisfied by JSIL reference implementations ✓
JAVERT

Figure 2.10: JaVerT [35]

Chapter 3

Symbolic Analysis

The symbolic analysis verifies JSIL procedure specifications. It performs this verification by symbolically executing JSIL basic and control flow commands. We take the symbolic execution rules introduced in [35], and modify them, so they closely relate to the implementation. In particular, the revised rules explicitly show the continuation from the precondition to the final state where the execution of the procedure does not fault. In the symbolic analysis we represent the symbolic states as JSIL assertions. The symbolic analysis is split into two sections, first we define the axioms for basic commands (Section 3.1) then we present the proof rules for the control flow commands (Section 3.2). Additionally, in Section 3.2 we present an example of the symbolic analysis. Finally, we establish the soundness of the symbolic analysis formally in Theorem 1 by appealing to the semantics of JSIL commands.

3.1 Axioms for Basic Commands

The axioms for the JSIL basic commands, given in Figure 3.1, have the form $\{P\}$ `bc` $\{Q\}$ meaning: when in a state where the JSIL assertion P holds, then executing the basic command `bc` will not fault and if the execution terminates then it results in a state satisfying JSIL assertion Q. Throughout the rules the notation $\mathsf{e}_1 \doteq \mathsf{e}_2$ represents $\mathsf{e}_1 = \mathsf{e}_2 \wedge \mathsf{emp}$.

We briefly explain the non-standard axioms.

- [Object Creation] When an object is created it has a single $@proto$ field with a null value. The $\mathsf{emptyFields}(\mathtt{x} \mid @proto)$ assertion, states that the object only has the $@proto$ field.

- [Field Deletion] In the field deletion rule, the object field $@proto$ cannot be deleted.

- [Member Check] In the member check rule, if the field's value is not None ($\varnothing$) then `x` is `true` otherwise `x` is `false`.

Evaluation. Soundness of the hoare triples for basic commands is established by Lemma 1. The lemma connects the hoare triples to the semantics of basic commands. In order to make this connection abstract heaps are related to concrete heaps using the erasure function, $\lfloor . \rfloor : \mathcal{H}^{\emptyset}_{\mathtt{JSIL}} \to \mathcal{H}_{\mathtt{JSIL}} : \lfloor H \rfloor (l, x) = H(l, x) \overset{\text{def}}{\iff} (l, x) \in \mathsf{dom}(H) \ \wedge \ H(l, x) \neq \varnothing$.

Lemma 1 (Frame Property and Soundness for Basic Commands)**.** *For all basic commands* $\mathtt{bc} \in \mathtt{BCmd}$*, abstract heaps* $H, \hat{H}_1, \hat{H}_2 \in \mathcal{H}^{\emptyset}_{\mathtt{JSIL}}$*, stores* $\rho, \rho_f \in \mathcal{S}to$*, logical environments* $\epsilon \in \mathcal{E}nv$*, values*

Basic Commands

Field Assignment
$$\frac{}{\{(\mathsf{e}_1,\mathsf{e}_2)\mapsto -\}\ [\mathsf{e}_1,\mathsf{e}_2] := \mathsf{e}_3\ \{(\mathsf{e}_1,\mathsf{e}_2)\mapsto \mathsf{e}_3\}}$$

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

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

Member Check
$$\frac{P = (\mathsf{e}_1,\mathsf{e}_2)\mapsto \mathrm{X} \qquad Q = P * \mathtt{x} \doteq \mathsf{not}\,(\mathrm{X} = \varnothing)}{\{P\}\ \mathtt{x} := \mathsf{hasField}(\mathsf{e}_1,\mathsf{e}_2)\ \{Q\}}$$

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

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

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

Frame
$$\frac{\{P\}\ \mathtt{bc}\ \{Q\}}{\{P * R\}\ \mathtt{bc}\ \{Q * R\}}$$

Consequence
$$\frac{\{P'\}\ \mathtt{bc}\ \{Q'\} \qquad P \vdash P' \qquad Q' \vdash Q}{\{P\}\ \mathtt{bc}\ \{Q\}}$$

Exists
$$\frac{\{P\}\ \mathtt{bc}\ \{Q\}}{\{\exists \mathrm{X}.\, P\}\ \mathtt{bc}\ \{\exists \mathrm{X}.\, Q\}}$$

Figure 3.1: Hoare Triples for Basic Commands

$\mathtt{v} \in \mathcal{V}_{\text{JSIL}}$, *JSIL heaps* $\hat{\mathtt{h}}_f \in \mathcal{H}_{\text{JSIL}}$, *and assertions* $P, Q \in \mathcal{AS}_{\text{JSIL}}$, *if* $\{P\}\,\mathtt{bc}\,\{Q\}$, $H, \rho, \epsilon \models P$, *and* $[\![\mathtt{bc}]\!]_{\lfloor H \uplus \hat{H}_1 \uplus \hat{H}_2 \rfloor, \rho} = (\hat{\mathtt{h}}_f, \rho_f, \mathtt{v})$*, then there is an abstract heap* H_f *such that* $H_f, \rho_f, \mathtt{v} \models Q$*,* $\hat{\mathtt{h}}_f = \lfloor H_f \uplus \hat{H}_1 \uplus \hat{H}_2 \rfloor$ *and* $[\![\mathtt{bc}]\!]_{\lfloor H \uplus \hat{H}_1 \rfloor, \rho} = (\lfloor H_f \uplus \hat{H}_1 \rfloor, \rho_f, \mathtt{v})$*.*

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

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

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

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

We proceed by induction on the derivation of **H1**.

- [Skip] We have that $\mathtt{bc} = \mathsf{skip}$ and, after applying **H1**, that $P = \mathsf{emp}$ and $Q = \mathsf{emp}$. From the satisfiability of JSIL assertions and **H2**, we obtain that $H = \mathsf{emp}$. From **H3** and the semantics of JSIL basic commands, we obtain that $\hat{\mathtt{h}}_f = \lfloor \hat{H}_1 \uplus \hat{H}_2 \rfloor$, $\rho_f = \rho$, and $\mathtt{v} = \mathsf{empty}$. We choose $H_f = \mathsf{emp}$, therefore the goals 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.

- [Field Assignment] We have that $\mathtt{bc} = [\mathsf{e}_1, \mathsf{e}_2] := \mathsf{e}_3$ and, after applying **H1**, that $P = (\mathsf{e}_1, \mathsf{e}_2) \mapsto \_$ and $Q = (\mathsf{e}_1, \mathsf{e}_2) \mapsto \mathsf{e}_3$. From the satisfiability of JSIL assertions and **H2**, we obtain that $H = (\llbracket \mathsf{e}_1 \rrbracket_\rho, \llbracket \mathsf{e}_2 \rrbracket_\rho) \mapsto V$, for some value V, possibly $\varnothing$. From **H3** and the semantics of JSIL basic commands, we obtain that $\hat{\mathtt{h}}_f = \lfloor (\llbracket \mathsf{e}_1 \rrbracket_\rho, \llbracket \mathsf{e}_2 \rrbracket_\rho) \mapsto \llbracket \mathsf{e}_3 \rrbracket_\rho \uplus \hat{H}_1 \uplus \hat{H}_2 \rfloor$, $\rho_f = \rho$, and $\mathtt{v} = \llbracket \mathsf{e} \rrbracket_{\rho_3}$. We choose $H_f = (\llbracket \mathsf{e}_1 \rrbracket_\rho, \llbracket \mathsf{e}_2 \rrbracket_\rho) \mapsto \llbracket \mathsf{e}_3 \rrbracket_\rho$, therefore the goals become:

  - **G1**: $\llbracket [\mathsf{e}_1, \mathsf{e}_2] := \mathsf{e}_3 \rrbracket_{\lfloor (\llbracket \mathsf{e}_1 \rrbracket_\rho, \llbracket \mathsf{e}_2 \rrbracket_\rho) \mapsto V \uplus \hat{H}_1 \rfloor, \rho} = (\lfloor (\llbracket \mathsf{e}_1 \rrbracket_\rho, \llbracket \mathsf{e}_2 \rrbracket_\rho) \mapsto \llbracket \mathsf{e}_3 \rrbracket_\rho \uplus \hat{H}_1 \rfloor, \rho, \llbracket \mathsf{e} \rrbracket_{\rho_3})$
  - **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 $\mathtt{bc} = \mathtt{x} := \mathsf{e}$ and, after applying **H1**, that $P = \mathsf{emp}$ and $Q = \mathtt{x} \doteq \mathsf{e}$. From the satisfiability of JSIL assertions and **H2**, we obtain that $H = \mathsf{emp}$. From **H3** and the semantics of JSIL basic commands, we obtain that $\hat{\mathtt{h}}_f = \lfloor \hat{H}_1 \uplus \hat{H}_2 \rfloor$, $\rho_f = \rho[\mathtt{x} \mapsto \llbracket \mathsf{e} \rrbracket_\rho]$, and $\mathtt{v} = \llbracket \mathsf{e} \rrbracket_\rho$. We choose $H_f = \mathsf{emp}$ therefore the goals 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.

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

  - **G1**: $\llbracket \mathsf{new}\,() \rrbracket_{\lfloor \hat{H}_1 \rfloor, \rho} = (\lfloor (l, @proto) \mapsto \mathsf{null} \uplus H_1 \rfloor, \rho[\mathtt{x} \mapsto l], l)$
  - **G2**: $H_f, \rho[\mathtt{x} \mapsto l], \epsilon \models (\mathtt{x}, @proto) \mapsto \mathsf{null} * \mathsf{emptyFields}(\mathtt{x} \mid \{\{@proto\}\})$
  - **G3**: $(l, @proto) \mapsto \mathsf{null} \uplus \lfloor \hat{H}_1 \uplus \hat{H}_2 \rfloor = \lfloor H_f \uplus \hat{H}_1 \uplus \hat{H}_2 \rfloor$.

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

- [Field Deletion] We have that $\mathtt{bc} = \mathsf{delete}\,(\mathsf{e}_1, \mathsf{e}_2)$ and, applying **H1**, that $P = (\mathsf{e}_1, \mathsf{e}_2) \mapsto \mathrm{X} * \mathrm{X} \not\doteq \varnothing$ and $Q = (\mathsf{e}_1, \mathsf{e}_2) \mapsto \varnothing$. From the satisfiability of JSIL assertions and **H2**, we obtain that $H = (\llbracket \mathsf{e}_1 \rrbracket_\rho, \llbracket \mathsf{e}_2 \rrbracket_\rho) \mapsto \epsilon(\mathrm{X})$, where $\llbracket \mathsf{e}_2 \rrbracket_\rho \neq @proto$. Note that $\lfloor H \rfloor = H$ since $\mathrm{X} \not\doteq \varnothing$. From **H3** and the semantics of JSIL basic commands, we obtain that $\hat{\mathtt{h}}_f = \lfloor \hat{H}_1 \uplus \hat{H}_2 \rfloor$, $\rho_f = \rho$, and $\mathtt{v} = \mathsf{true}$. We choose $H_f = (\llbracket \mathsf{e}_1 \rrbracket_\rho, \llbracket \mathsf{e}_2 \rrbracket_\rho) \mapsto \varnothing$ therefore noting that $\lfloor H_f \rfloor = \mathsf{emp}$ the goals become:

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

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

- [FIELD ACCESS] We have that $\mathtt{bc} = \mathtt{x} := [\mathsf{e}_1, \mathsf{e}_2]$ and, after applying **H1**, that $P = (\mathsf{e}_1, \mathsf{e}_2) \mapsto \mathrm{X} * \mathrm{X} \not\doteq \varnothing$ and $Q = (\mathsf{e}_1, \mathsf{e}_2) \mapsto \mathrm{X} * \mathrm{X} \not\doteq \varnothing * \mathtt{x} \doteq \mathrm{X}$. From the satisfiability of JSIL assertions and **H2**, we obtain that $H = ([\![\mathsf{e}_1]\!]_\rho, [\![\mathsf{e}_2]\!]_\rho) \mapsto \epsilon(\mathrm{X})$. Note that $\lfloor H \rfloor = H$ since $\mathrm{X} \not\doteq \varnothing$. From **H3** and the semantics of JSIL basic commands, we obtain that $\hat{\mathtt{h}}_f = \lfloor H \uplus \hat{H}_1 \uplus \hat{H}_2 \rfloor$, $\rho_f = \rho[\mathtt{x} \mapsto \epsilon(\mathrm{X})]$, and $\mathtt{v} = \epsilon(\mathrm{X})$. We choose $H_f = H = ([\![\mathsf{e}_1]\!]_\rho, [\![\mathsf{e}_2]\!]_\rho) \mapsto \epsilon(\mathrm{X})$ therefore the goals become:
  - **G1**: $[\![\mathtt{x} := [\mathsf{e}_1, \mathsf{e}_2]]\!]_{\lfloor ([\![\mathsf{e}_1]\!]_\rho, [\![\mathsf{e}_2]\!]_\rho) \mapsto \epsilon(\mathrm{X}) \uplus \hat{H}_1 \rfloor, \rho} = (\lfloor ([\![\mathsf{e}_1]\!]_\rho, [\![\mathsf{e}_2]\!]_\rho) \mapsto \epsilon(\mathrm{X}) \uplus \hat{H}_1 \rfloor, \rho[\mathtt{x} \mapsto \epsilon(\mathrm{X})], \epsilon(\mathrm{X}))$
  - **G2**: $([\![\mathsf{e}_1]\!]_\rho, [\![\mathsf{e}_2]\!]_\rho) \mapsto \epsilon(\mathrm{X}), \rho[\mathtt{x} \mapsto \epsilon(\mathrm{X})], \epsilon \models (\mathsf{e}_1, \mathsf{e}_2) \mapsto \mathrm{X} * \mathrm{X} \not\doteq \varnothing * \mathtt{x} \doteq \mathrm{X}$.
  - **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] We have that $\mathtt{bc} = x := \mathsf{hasField}(\mathsf{e}_1, \mathsf{e}_2)$. After applying **H1**, we obtain $P = (\mathsf{e}_1, \mathsf{e}_2) \mapsto \mathrm{X}$ **(I1)** and $Q = P * \mathtt{x} \doteq \mathsf{not}\,(\mathrm{X} = \varnothing)$ **(I2)** Using the satisfiability of JSIL assertions and **H2**, we obtain $H = ([\![E_1]\!]^\epsilon_\rho, [\![E_2]\!]^\epsilon_\rho) \mapsto \epsilon(\mathrm{X})$ **(I3)**.

  From **I2** we consider two possible cases:
  - $\mathrm{X} = \varnothing$ **(I4)**

    Due to **I4**, we note $\lfloor H \rfloor = H$. From **H3** and the semantics of JSIL basic commands, we obtain that $\hat{\mathtt{h}}_f = \lfloor H \uplus \hat{H}_1 \uplus \hat{H}_2 \rfloor$ and $\rho_f = \rho[\mathtt{x} \mapsto \mathsf{true}]$. We choose $H_f = H = ([\![\mathsf{e}_1]\!]_\rho, [\![\mathsf{e}_2]\!]_\rho) \mapsto \epsilon(\mathrm{X})$ and therefore the goals become:
    * **G1**: $[\![x := \mathsf{hasField}(\mathsf{e}_1, \mathsf{e}_2)]\!]_{\lfloor ([\![E_1]\!]^\epsilon_\rho, [\![E_2]\!]^\epsilon_\rho) \mapsto \epsilon(\mathrm{X}) \uplus \hat{H}_1 \rfloor, \rho} = (\lfloor ([\![E_1]\!]^\epsilon_\rho, [\![E_2]\!]^\epsilon_\rho) \mapsto \epsilon(\mathrm{X}) \uplus \hat{H}_1 \rfloor, \rho[\mathtt{x} \mapsto \mathsf{true}], \mathsf{true})$
    * **G2**: $([\![\mathsf{e}_1]\!]_\rho, [\![\mathsf{e}_2]\!]_\rho) \mapsto \epsilon(\mathrm{X}), \rho[\mathtt{x} \mapsto \mathsf{true}], \epsilon \models P * \mathtt{x} \doteq \mathsf{not}\,(\mathrm{X} = \varnothing)$
    * **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 $\epsilon(\mathrm{X}) \neq \varnothing$.
  - $\mathrm{X} \neq \varnothing$ **(I5)**

    Therefore $H_1 \uplus H_2$ cannot contain the cell $([\![\mathsf{e}_1]\!]_\rho, [\![\mathsf{e}_2]\!]_\rho)$, as the disjoint union $H \uplus \hat{H}_1 \uplus \hat{H}_2$ is well-defined. The goals follow from the $\mathrm{X} = \varnothing$ case, with false instead of true.

- [CONSEQUENCE] We have $\{P\}$ $\mathtt{bc}$ $\{Q\}$. From **H1** we obtain $\{P'\}$ $\mathtt{bc}$ $\{Q'\}$ **(I1)**, $P \vdash P'$ **(I2)** and $Q' \vdash Q$ **(I3)**. From **I2** and **H2** we conclude $H, \rho, \epsilon \models P'$ **(I4)**. Applying the inductive hypothesis to **I4**, **I1** and **H3** we obtain $H_f, \rho_f, \epsilon \models Q'$ **(I5)**, **G1** and **G3**. Finally, we obtain goal **G2** from **I5** and **I3**.

- [FRAME] We have $\{P * R\}$ $\mathtt{bc}$ $\{Q * R\}$. From **H1** we obtain $\{P\}$ $\mathtt{bc}$ $\{Q\}$ **(I1)**. We conclude from the satisfiability of JSIL assertions and **H2**, that there exists a H_1 and H_2 such that $H = H_1 \uplus H_2$ **(I2)**, $H_1, \rho, \epsilon \models P$ **(I3)** and $H_2, \rho, \epsilon \models R$ **(I4)**. From **I2** we have $[\![\mathtt{bc}]\!]_{\lfloor H_1 \uplus (H_2 \uplus \hat{H}_1) \uplus \hat{H}_2 \rfloor, \rho} = (\hat{\mathtt{h}}_f, \rho_f, \mathtt{v})$ **(I5)**. Applying the induction hypothesis to **I1**, **I3** and **I5** we obtain that there exists a JSIL abstract heap H' such that:
  - **G1**: $[\![\mathtt{bc}]\!]_{\lfloor H \uplus H_2 \uplus \hat{H}_1 \rfloor, \rho} = (\lfloor H' \uplus H_2 \uplus \hat{H}_1 \rfloor, \rho', \mathtt{v})$
  - **I6**: $H', \rho', \epsilon \models Q$.
  - **G3**: $\hat{\mathtt{h}}' = \lfloor H' \uplus H_2 \uplus \hat{H}_1 \uplus \hat{H}_2 \rfloor$.

  From **I6**, **I4** and the satisfiability of JSIL assertions, we obtain $H_f, \rho', \epsilon \models Q * R$ **(G2)** where $H_f = H' \uplus H_2$.

- [Exists] We have $\{\exists \mathrm{X}.\, P\}$ $\texttt{bc}$ $\{\exists \mathrm{X}.\, Q\}$. From **H1** we obtain $\{P\}$ $\texttt{bc}$ $\{Q\}$ (**I1**). From the satisfiability of JSIL assertions and **H2**, we obtain $\exists V \in \mathcal{V}^{L}_{\texttt{JSIL}}.\, H, \rho, \epsilon[\mathrm{X} \mapsto V] \models P$ (**I2**). Applying the induction hypothesis to **I1**, **I2** and **H3**, we obtain $H_f, \rho_f, \epsilon[\mathrm{X} \mapsto V] \models Q$ (**I3**), **G1** and **G3**. Using **I3** and the satisfiability of JSIL assertions we are able to achieve **G2**.

□

3.2 Symbolic Execution Rules for Control Flow Commands

We present symbolic execution rules for the control flow commands in Figure 3.2, these rules allow for verification of procedure specifications.

Procedure specifications have the form $\{P\}\, \texttt{m}(\overline{\texttt{x}})\, \{Q\} \in \mathcal{S}pec$ where $\texttt{m}$ is the procedure name, $\overline{\texttt{x}}$ are the procedure parameters, P is the precondition of the procedure and Q is the postcondition. The pre- and postcondition are JSIL logic assertions. Procedures can return in a normal mode or an error mode, this is denoted by a return flag $fl \in \{\mathsf{nm}, \mathsf{er}\}$. A specification environment, $\texttt{S} : \mathcal{S}tr \rightharpoonup \mathcal{F}lag \rightharpoonup \mathcal{S}pec$, maps a procedure name and return flag onto its corresponding procedure specification.

The symbolic execution rules take the form $\texttt{p}, \texttt{S}, \texttt{m} \vdash_{fl} \{P, j, i\} \rightsquigarrow Q$ where:

- $\texttt{m} \in \mathcal{S}tr$ and $\texttt{p} \in \texttt{P}$ respectively denote the JSIL procedure name and program being analysed, and $\texttt{S}$ the specification environment;
- $fl \in \mathcal{F}lag$ denotes the return mode of the procedure currently being executed;
- i denotes the index of the JSIL command to be symbolically executed and j the index of the command that was symbolically executed immediately before i; and
- P is an assertion describing the precondition of the ith command.

The result of the symbolic execution for control flow commands is an assertion Q which describes the final state reached.

We say that a specification environment $\texttt{S}$ is well-formed if all the specifications in the specification environment are provable using the symbolic execution. Formally, for all fl and $\texttt{m}$ where $\texttt{S}(\texttt{m}, fl) = \{P\}$ $\texttt{m}(\texttt{x}_1, ..., \texttt{x}_\texttt{n})$ $\{Q\}$ then $\texttt{p}, \texttt{S}, \texttt{m} \vdash_{fl} \{P, 0, 0\} \rightsquigarrow Q$ and $\mathsf{vars(P)} \cup \mathsf{vars(Q)} \subseteq \{\texttt{x}_1, ..., \texttt{x}_\texttt{n}\}$.

We briefly explain the more complicated rules below.

- [Conditional Goto - Unknown] Upon reaching a conditional goto command, where the conditional expression cannot be evaluated to `false` or `true`, the proof rule branches down both possibilities.

- [Phi-Assignment] We use the notation $i \overset{k}{\mapsto}_{\texttt{m}} j$ to denote that i is the k-th predecessor of j in procedure $\texttt{m}$. The phi-assignment rule assigns the k-th element of the list $(\texttt{x}_1, ..., \texttt{x}_n)$ to the variable $\texttt{x}$, where i is the k-th predecessor of j.

- [Return] The terminating commands, return normal and return error, ensure the final symbolic execution state entails the specified postcondition of the procedure.

- [PROCEDURE CALL] Both procedure call rules state that the current state must entail the precondition of the callee and some disjoint frame state P_F. The continuation then combines this frame state with the postcondition of the callee.

Control Flow Commands

BASIC COMMAND

$$\frac{p_m(i) = \mathtt{bc} \qquad \mathtt{bc} \in BCmd \qquad \{P\}\ \mathtt{bc}\ \{Q\} \qquad \mathtt{p}, \mathtt{S}, \mathtt{m} \vdash_{fl} \{Q, i, i+1\} \rightsquigarrow Q'}{\mathtt{p}, \mathtt{S}, \mathtt{m} \vdash_{fl} \{P, \_, i\} \rightsquigarrow Q'}$$

GOTO

$$\frac{p_m(i) = \mathsf{goto}\quad k \qquad \mathtt{p}, \mathtt{S}, \mathtt{m} \vdash_{fl} \{P, i, k\} \rightsquigarrow Q}{\mathtt{p}, \mathtt{S}, \mathtt{m} \vdash_{fl} \{P, \_, i\} \rightsquigarrow Q}$$

CONDITIONAL GOTO - TRUE

$$\frac{p_m(i) = \mathsf{goto}\quad \mathsf{e}\quad k\quad j \qquad P \vdash \mathtt{true} * \mathsf{e} \doteq \mathtt{true} \qquad \mathtt{p}, \mathtt{S}, \mathtt{m} \vdash_{fl} \{P, i, k\} \rightsquigarrow Q}{\mathtt{p}, \mathtt{S}, \mathtt{m} \vdash_{fl} \{P, \_, i\} \rightsquigarrow Q}$$

CONDITIONAL GOTO - FALSE

$$\frac{p_m(i) = \mathsf{goto}\quad \mathsf{e}\quad k\quad j \qquad P \vdash \mathtt{true} * \mathsf{e} \doteq \mathtt{false} \qquad \mathtt{p}, \mathtt{S}, \mathtt{m} \vdash_{fl} \{P, i, j\} \rightsquigarrow Q}{\mathtt{p}, \mathtt{S}, \mathtt{m} \vdash_{fl} \{P, \_, i\} \rightsquigarrow Q}$$

CONDITIONAL GOTO - UNKNOWN

$$\frac{\begin{array}{c} p_m(i) = \mathsf{goto}\quad \mathsf{e}\quad k\quad j \qquad P \nvdash \mathtt{true} * \mathsf{e} \doteq \mathtt{false} \qquad P \nvdash \mathtt{true} * \mathsf{e} \doteq \mathtt{true} \\ \mathtt{p}, \mathtt{S}, \mathtt{m} \vdash_{fl} \{P * (\mathsf{e} \doteq \mathtt{true}), i, k\} \rightsquigarrow Q \qquad \mathtt{p}, \mathtt{S}, \mathtt{m} \vdash_{fl} \{P * (\mathsf{e} \doteq \mathtt{false}), i, j\} \rightsquigarrow Q \end{array}}{\mathtt{p}, \mathtt{S}, \mathtt{m} \vdash_{fl} \{P, \_, i\} \rightsquigarrow Q}$$

PHI-ASSIGNMENT

$$\frac{p_m(i) = \mathtt{x} := \phi(\mathtt{x}_1, ..., \mathtt{x}_n) \qquad j \overset{k}{\mapsto}_m i \qquad \mathtt{p}, \mathtt{S}, \mathtt{m} \vdash_{fl} \{P * \mathtt{x} \doteq \mathtt{x}_k, i, i+1\} \rightsquigarrow Q}{\mathtt{p}, \mathtt{S}, \mathtt{m} \vdash_{fl} \{P, j, i\} \rightsquigarrow Q}$$

PROCEDURE CALL - NORMAL

$$\frac{\begin{array}{c} \mathtt{p}_\mathtt{m}(i) = \mathtt{x} := \mathsf{e}_0(\mathsf{e}_1, ..., \mathsf{e}_{n_1})\ \mathsf{with}\ k \qquad \mathtt{S}(\mathtt{m}', \mathsf{nm}) = \{P'\}\ \mathtt{m}'(\mathtt{x}_1, ..., \mathtt{x}_{\mathtt{n}_2})\ \{Q' * \mathtt{xret} \doteq \mathsf{e}\} \\ \mathsf{e}_n = \mathsf{undefined}\ |_{n=n_1+1}^{n2} \qquad P \vdash P_F * P'[\mathsf{e}_i/\mathtt{x}_i|_{i=1}^{n_2}] * \mathsf{e}_0 \doteq \mathtt{m}' \\ Q'' = P_F * Q'[\mathsf{e}_i/\mathtt{x}_i|_{i=1}^{n_2}] * \mathsf{e}_0 \doteq \mathtt{m}' * \mathtt{x} \doteq \mathsf{e}[\mathsf{e}_i/\mathtt{x}_i|_{i=1}^{n_2}] \qquad \mathtt{p}, \mathtt{S}, \mathtt{m} \vdash_{fl} \{Q'', i, i+1\} \rightsquigarrow Q \end{array}}{\mathtt{p}, \mathtt{S}, \mathtt{m} \vdash_{fl} \{P, \_, i\} \rightsquigarrow Q}$$

PROCEDURE CALL - ERROR

$$\frac{\begin{array}{c} \mathtt{p}_\mathtt{m}(i) = \mathtt{x} := \mathsf{e}_0(\mathsf{e}_1, ..., \mathsf{e}_{n_1})\ \mathsf{with}\ k \qquad \mathtt{S}(\mathtt{m}', \mathsf{er}) = \{P'\}\ \mathtt{m}'(\mathtt{x}_1, ..., \mathtt{x}_{\mathtt{n}_2})\ \{Q' * \mathtt{xerr} \doteq \mathsf{e}\} \\ \mathsf{e}_n = \mathsf{undefined}\ |_{n=n_1+1}^{n2} \qquad P \vdash P_F * P'[\mathsf{e}_i/\mathtt{x}_i|_{i=1}^{n_2}] * \mathsf{e}_0 \doteq \mathtt{m}' \\ Q'' = P_F * Q'[\mathsf{e}_i/\mathtt{x}_i|_{i=1}^{n_2}] * \mathsf{e}_0 \doteq \mathtt{m}' * \mathtt{x} \doteq \mathsf{e}[\mathsf{e}_i/\mathtt{x}_i|_{i=1}^{n_2}] \qquad \mathtt{p}, \mathtt{S}, \mathtt{m} \vdash_{fl} \{Q'', i, k\} \rightsquigarrow Q \end{array}}{\mathtt{p}, \mathtt{S}, \mathtt{m} \vdash_{fl} \{P, \_, i\} \rightsquigarrow Q}$$

RETURN - NORMAL

$$\frac{\mathtt{S}(\mathtt{m}, \mathsf{nm}) = \{P'\}\ \mathtt{m}(\mathtt{x}_1, ..., \mathtt{x}_{\mathtt{n}_2})\ \{Q'\} \qquad P \vdash Q'}{\mathtt{p}, \mathtt{S}, \mathtt{m} \vdash_{\mathsf{nm}} \{P, \_, i_{\mathsf{nm}}\} \rightsquigarrow Q'}$$

RETURN - ERROR

$$\frac{\mathtt{S}(\mathtt{m}, \mathsf{er}) = \{P'\}\ \mathtt{m}(\mathtt{x}_1, ..., \mathtt{x}_{\mathtt{n}_2})\ \{Q'\} \qquad P \vdash Q'}{\mathtt{p}, \mathtt{S}, \mathtt{m} \vdash_{\mathsf{er}} \{P, \_, i_{\mathsf{er}}\} \rightsquigarrow Q'}$$

Figure 3.2: Control Flow Proof Rules

Example. In order to illustrate the symbolic execution proof rules, we present a proof sketch for the procedure swap. We prove the specification:

`Precondition:`

$$\left\{ \begin{array}{l} cur\_node \neq \mathsf{null} \ * \ (cur\_node, \text{“}prev\text{”}) \mapsto \#prev \ * \ \#prev \neq \varnothing \\ \ * \ (cur\_node, \text{“}next\text{”}) \mapsto \#next \ * \ \#next \neq \varnothing \end{array} \right\}$$

`Postcondition:`

$$\left\{ \begin{array}{l} cur\_node \dot{\neq} \mathsf{null} \ * \ (cur\_node, \text{“}prev\text{”}) \mapsto \#next \ * \ \#prev \dot{\neq} \varnothing \\ \ * \ (cur\_node, \text{“}next\text{”}) \mapsto \#prev \ * \ \#next \dot{\neq} \varnothing \ * \ ret = \#next \end{array} \right\}$$

```
Flag:
Normal
```

The specification states that the current node cannot be null and it has two none-non fields: previous and next. The procedure swaps these two fields and returns the value of next.

```
proc swap (cur_node) {
   { $cur\_node \dot{\neq} \mathsf{null} \ * \ (cur\_node, “prev”) \mapsto \#prev \ * \ \#prev \dot{\neq} \varnothing$
     $* \ (cur\_node, “next”) \mapsto \#next \ * \ \#next \dot{\neq} \varnothing$ }
   goto [cur_node = null] then else;
   then:     ret := null;
             goto rlab;
   else:     prev := [cur_node, "prev"];
             { $cur\_node \dot{\neq} \mathsf{null} \ * \ (cur\_node, “prev”) \mapsto \#prev \ * \ \#prev \dot{\neq} \varnothing$
               $* \ (cur\_node, “next”) \mapsto \#next \ * \ \#next \dot{\neq} \varnothing \ * \ prev \doteq \#prev$ }
             next := [cur_node, "next"];
             { $cur\_node \dot{\neq} \mathsf{null} \ * \ (cur\_node, “prev”) \mapsto \#prev \ * \ \#prev \dot{\neq} \varnothing$
               $* \ (cur\_node, “next”) \mapsto \#next \ * \ \#next \dot{\neq} \varnothing \ * \ prev \doteq \#prev$
               $* \ next \doteq \#next$ }
             [cur_node, "prev"] := next;
             { $cur\_node \dot{\neq} \mathsf{null} \ * \ (cur\_node, “prev”) \mapsto \#next \ * \ \#prev \dot{\neq} \varnothing$
               $* \ (cur\_node, “next”) \mapsto \#next \ * \ \#next \dot{\neq} \varnothing \ * \ prev \doteq \#prev$
               $* \ next \doteq \#next$ }
             [cur_node, "next"] := prev;
             { $cur\_node \dot{\neq} \mathsf{null} \ * \ (cur\_node, “prev”) \mapsto \#next \ * \ \#prev \dot{\neq} \varnothing$
               $* \ (cur\_node, “next”) \mapsto \#prev \ * \ \#next \dot{\neq} \varnothing \ * \ prev \doteq \#prev$
               $* \ next \doteq \#next$ }
             ret := next;
             { $cur\_node \dot{\neq} \mathsf{null} \ * \ (cur\_node, “prev”) \mapsto \#next \ * \ \#prev \dot{\neq} \varnothing$
               $* \ (cur\_node, “next”) \mapsto \#prev \ * \ \#next \dot{\neq} \varnothing \ * \ prev \doteq \#prev$
               $* \ next \doteq \#next \ * \ ret \doteq \#next$ }
   rlab:     skip
} with {ret: ret, rlab;};
```

Evaluation. The soundness of the symbolic analysis is established in Theorem 1. The proof connects the symbolic execution control flow commands to the JSIL semantics which are given in Appendix A.0.2.

Theorem 1 (Soundness of Symbolic Analysis)**.** *For any abstract heaps $H, \hat{H}_1, \hat{H}_2 \in \mathcal{H}^{\emptyset}_{\texttt{JSIL}}$, store $\rho \in \mathcal{S}to$, logical environment $\epsilon \in \mathcal{E}nv$, program $\texttt{p} \in \texttt{P}$, well-formed specification environment $\texttt{S}$, JSIL assertions $P, Q \in \mathcal{AS}_{\texttt{JSIL}}$, procedure name $\texttt{m} \in \mathcal{S}tr$ and command labels i and j such that:*

- $H, \rho, \epsilon \models P$ **(H1)**,
- $\texttt{p}, \texttt{S}, \texttt{m} \vdash_{fl} \{P, i, j\} \rightsquigarrow Q$ **(H2)**,
- $\texttt{p} \vdash \langle \lfloor H \uplus \hat{H}_1 \uplus \hat{H}_2 \rfloor, \rho, i, j \rangle \Downarrow_{\texttt{m}} \langle \texttt{h}_f, \rho_f, o \rangle$ **(H3)**

then there is an abstract heap H_f such that:

- $H_f, \rho_f, \epsilon \models Q$ **(G1)**,
- $o = fl\langle \texttt{v} \rangle$ *for some value* $\texttt{v}$ **(G2)**,
- $\lfloor H_f \uplus \hat{H}_1 \uplus \hat{H}_2 \rfloor = \texttt{h}_f$ **(G3)**,

- $\mathtt{p} \vdash \langle \lfloor H \uplus \hat{H}_1 \rfloor, \rho, i, j\rangle \Downarrow_{\mathtt{m}} \langle \lfloor H_f \uplus \hat{H}_1 \rfloor, \rho_f, o\rangle$ **(G4)**

Proof. We proceed by induction on the derivation of **H3**.

[Procedure Call - Normal] It follows that $\mathtt{p}_{\mathtt{m}}(j) = \mathtt{x} := \mathtt{e}_0(\mathtt{e}_1, ..., \mathtt{e}_{n_1})$ with k for a given $JSIL$ variable $\mathtt{x}$, n JSIL expressions $\mathtt{e}_0$, $\mathtt{e}_1$, ..., $\mathtt{e}_{n_1}$, and index k. We conclude, using **H3** and the semantics of JSIL, that:

$[\![\mathtt{e}_0]\!]_\rho = \mathtt{m}'$ **(I1)** $\quad \mathtt{p}(\mathtt{m}') = \mathsf{proc}\ \mathtt{m}'(\mathtt{y}_1, ..., \mathtt{y}_{n_2})\{\bar{\mathtt{c}}\}$ **(I2)** $\quad \forall_{1 \le n \le n_1} \mathtt{v}_n = [\![\mathtt{e}_n]\!]_\rho$ **(I3)**
$\forall_{n_1 < n \le n_2} \mathtt{v}_n = \mathsf{undefined}$ **(I4)** $\quad \mathtt{p} \vdash \langle \lfloor H \uplus \hat{H}_1 \uplus \hat{H}_2 \rfloor, \emptyset[\mathtt{y}_i \mapsto \mathtt{v}_i|_{i=1}^{n_2}], 0, 0\rangle \Downarrow_{\mathtt{m}'} \langle \mathtt{h}', \rho', \mathsf{nm}\langle \mathtt{v}'\rangle\rangle$ **(I5)**
$\mathtt{p} \vdash \langle \mathtt{h}', \rho[\mathtt{x} \mapsto \mathtt{v}'], j, j+1\rangle \Downarrow_{\mathtt{m}} \langle \mathtt{h}_f, \rho_f, o\rangle$ **(I6)**

From **H2**, we conclude that:

$\mathtt{S}(\mathtt{m}'', \mathsf{nm}) = \{P'\}\ \mathtt{m}''(\mathtt{x}_1, ..., \mathtt{x}_{n_3})\ \{Q' * \mathtt{xret} \doteq \mathtt{e}\}$ **(I7)**
$\mathtt{e}_n = \mathsf{undefined}\ |_{n=n_1+1}^{n_3}$ **(I8)** $\quad P \vdash P_F * P'[\mathtt{e}_i/\mathtt{x}_i|_{i=1}^{n_3}] * \mathtt{e}_0 \doteq \mathtt{m}''$ **(I9)**
$Q'' = P_F * Q'[\mathtt{e}_i/\mathtt{x}_i|_{i=1}^{n_3}] * \mathtt{e}_0 \doteq \mathtt{m}'' * \mathtt{x} \doteq \mathtt{e}[\mathtt{e}_i/\mathtt{x}_i|_{i=1}^{n_3}]$ **(I10)** $\quad \mathtt{p}, \mathtt{S}, \mathtt{m} \vdash_{fl} \{Q'', j, j+1\} \rightsquigarrow Q$ **(I11)**

From **H1**, **I1**, **I2**,**I7**, and **I9**, we conclude that $\mathtt{m}'' = [\![\mathtt{e}_0]\!]_\rho = \mathtt{m}'$, $n_2 = n_3$, and $(\mathtt{x}_1, ..., \mathtt{x}_{n_3}) = (\mathtt{y}_1, ..., \mathtt{y}_{n_2})$**(I12)**. For convenience, using **(I12)**, we rewrite **I7**-**I10** as follows:

$\mathtt{S}(\mathtt{m}', \mathsf{nm}) = \{P'\}\ \mathtt{m}'(\mathtt{y}_1, ..., \mathtt{y}_{n_2})\ \{Q' * \mathtt{xret} \doteq \mathtt{e}\}$ **(I13)** $\quad \mathtt{e}_n = \mathsf{undefined}\ |_{n=n_1+1}^{n_2}$ **(I14)**
$P \vdash P_F * P'[\mathtt{e}_i/\mathtt{y}_i|_{i=1}^{n_2}] * \mathtt{e}_0 \doteq \mathtt{m}'$ **(I15)** $\quad Q'' = P_F * Q'[\mathtt{e}_i/\mathtt{y}_i|_{i=1}^{n_2}] * \mathtt{e}_0 \doteq \mathtt{m}' * \mathtt{x} \doteq \mathtt{e}[\mathtt{e}_i/\mathtt{y}_i|_{i=1}^{n_2}]$ **(I16)**

Noting that all the specs in $\mathtt{S}$ are well-formed, we conclude, from **I13** that $\mathsf{vars}(\mathsf{P}') \cup \mathsf{vars}(\mathsf{Q}') \cup \mathsf{vars}(\mathsf{e}) \subseteq \{\mathtt{y}_1, ..., \mathtt{y}_{n_2}\}$ **(I17)**. From **H1** and **I15**, we conclude that there are two heaps H_1 and H_2 such that: $H = H_1 \uplus H_2$ **(I18)**, $H_1, \rho, \epsilon \models P_F$ **(I19)**, and $H_2, \rho, \epsilon \models P'[\mathtt{e}_i/\mathtt{y}_i|_{i=1}^{n_2}]$ **(I20)**. Applying the Substitution Lemma for Assertions (Lemma 3, given in the Appendix 3) to **I20** and **I17**, we conclude that $H_2, \emptyset[\mathtt{y}_i \mapsto [\![\mathtt{e}_i]\!]_\rho^\epsilon|_{i=1}^{n_2}], \epsilon \models P'$ **(I21)**. From **I3**, **I4**, and **I21**, we conclude that:

$$H_2, \emptyset[\mathtt{y}_i \mapsto \mathtt{v}_i|_{i=1}^{n_2}], \epsilon \models P' \ \textbf{(I22)}$$

Because the specification environment is well-formed, we conclude that:

$$\mathtt{p}, \mathtt{S}, \mathtt{m}' \vdash_{\mathsf{nm}} \{P', 0, 0\} \rightsquigarrow (Q' * \mathtt{xret} \doteq \mathtt{e}) \ \textbf{(I23)}$$

Applying the induction hypothesis to **I5**, **I18**, **I22**, and **I23**, we conclude that there is an abstract H' such that: $H', \rho', \epsilon \models (Q' * \mathtt{xret} \doteq \mathtt{e})$ **(I24)**, $\lfloor H' \uplus H_1 \uplus \hat{H}_1 \uplus \hat{H}_2 \rfloor = \mathtt{h}'$ **(I25)** , and $\mathtt{p} \vdash \langle \lfloor H_2 \uplus H_1 \uplus \hat{H}_1 \rfloor, \rho, 0, 0\rangle \Downarrow_{\mathtt{m}'} \langle \lfloor H' \uplus H_1 \uplus \hat{H}_1 \rfloor, \rho', \mathsf{nm}\langle \mathtt{v}'\rangle\rangle$ **(I26)**. Since we only consider programs in SSA, we conclude, from **I26**, that $\rho' \ge \emptyset[\mathtt{y}_i \mapsto \mathtt{v}_i|_{i=1}^{n_2}]$ **(I27)**. From **I17**, **I24**, and **I27**, we conclude that:

$$H', \emptyset[\mathtt{y}_i \mapsto \mathtt{v}_i|_{i=1}^{n_2}], \epsilon \models Q' \ \textbf{(I28)} \qquad [\![\mathtt{e}]\!]^\epsilon_{\emptyset[\mathtt{y}_i \mapsto \mathtt{v}_i|_{i=1}^{n_2}]} = \rho'(\mathtt{xret}) = \mathtt{v}' \ \textbf{(I29)}$$

Applying the Substitution Lemma to **I28**, we conclude that $H', \rho, \epsilon \models Q'[\mathtt{e}_i/\mathtt{y}_i|_{i=1}^{n_2}]$ **(I30)**. Applying the Substitution Lemma for Expressions to **I29**, we conclude that: $[\![\mathtt{e}[\mathtt{e}_i/\mathtt{y}_i|_{i=1}^{n_2}]]\!]_\rho^\epsilon = \mathtt{v}'$ **(I31)**. From **H1**, **I15**, **I16**, **I19**, **I30**, and **I31**, we conclude that: $H_1 \uplus H', \rho, \epsilon \models Q''$ **(I32)**. Applying the induction hypothesis to **I6**, **I11**, **I25**, **I32**, we conclude that there is an abstract H_f such that: $H_f, \rho_f, \epsilon \models Q$ **(G1)**, $\lfloor H_f \uplus \hat{H}_1 \uplus \hat{H}_2 \rfloor = \mathtt{h}_f$ **(G3)** , and $\mathtt{p} \vdash \langle \lfloor H_1 \uplus H' \uplus \hat{H}_1 \rfloor, \rho[\mathtt{x} \mapsto \mathtt{v}'], j, j+1\rangle \Downarrow_{\mathtt{m}} \langle \lfloor H_f \uplus \hat{H}_1 \rfloor, \rho_f, o\rangle$ **(I33)** and there is a value $\mathtt{v}$ such that $o = fl\langle \mathtt{v}\rangle$ **(G2)**. **G4** follows from **I1**-**I4**, **I26**, and **I33**.

[BASIC COMMAND] It follows that $\mathtt{p_m}(j) = \mathtt{bc} \in \mathtt{BCmd}$ **(I1)**. We conclude, using **H3** and the semantics of JSIL, that:

$$[\![\mathtt{bc}]\!]_{\lfloor H \uplus \hat{H}_1 \uplus \hat{H}_2 \rfloor, \rho} = (\mathtt{h}', \rho', \mathtt{v}) \ \textbf{(I2)} \qquad \mathtt{p} \vdash \langle \mathtt{h}', \rho', j, j+1 \rangle \Downarrow_{\mathtt{m}} \langle \mathtt{h}_f, \rho_f, \mathtt{o} \rangle \ \textbf{(I3)}$$

From **H2** we conclude that:

$$\{P\} \ \mathtt{bc} \ \{Q\} \ \textbf{(I4)} \qquad \mathtt{p}, \mathtt{S}, \mathtt{m} \vdash_{fl} \{Q, j, j+1\} \rightsquigarrow Q' \ \textbf{(I5)}$$

Applying the Soundness of Basic Commands (Lemma 1) to **H1**, **I2** and **I4**, then there is an abstract heap H' such that:

$$[\![\mathtt{bc}]\!]_{\lfloor H \uplus \hat{H}_1 \rfloor, \rho} = (\lfloor H' \uplus \hat{H}_1 \rfloor, \rho', \mathtt{v}) \ \textbf{(I6)} \qquad H', \rho', \epsilon \models Q \ \textbf{(I7)} \qquad \mathtt{h}' = \lfloor H' \uplus \hat{H}_1 \uplus \hat{H}_2 \rfloor \ \textbf{(I8)}$$

We then apply the induction hypothesis to **I3**, **I5**, **I7** so there exists an abstract heap H_f such that:

$$\mathtt{p} \vdash \langle \lfloor H' \uplus \hat{H}_1 \rfloor, \rho', j, j+1 \rangle \Downarrow_{\mathtt{m}} \langle \lfloor H_f \uplus \hat{H}_1 \rfloor, \rho_f, o \rangle \ \textbf{(I9)} \qquad o = fl\langle \mathtt{v} \rangle \ \textbf{(G2)}$$
$$\mathtt{h}_f = \lfloor H_f \uplus \hat{H}_1 \uplus \hat{H}_2 \rfloor \ \textbf{(G3)} \qquad H_f, \rho_f, \epsilon \models Q' \ \textbf{(G1)}$$

We are then able to apply the semantics to **I6**, **I9** and **I1** in order to obtain $\mathtt{p} \vdash \langle \lfloor H \uplus \hat{H}_1 \rfloor, \rho, i, j \rangle \Downarrow_{\mathtt{m}} \langle \lfloor H_f \uplus \hat{H}_1 \rfloor, \rho_f, o \rangle$ **(G4)**

[GOTO] It follows that $\mathtt{p_m}(j) = \text{goto } k$. We conclude, using **H3** and the semantics of JSIL, that $\mathtt{p} \vdash \langle \lfloor H \uplus \hat{H}_1 \uplus \hat{H}_2 \rfloor, \rho, i, k \rangle \Downarrow_{\mathtt{m}} \langle \mathtt{h}_f, \rho_f, o \rangle$ **(I1)**. From **H2**, we conclude that $\mathtt{p}, \mathtt{S}, \mathtt{m} \vdash_{fl} \{P, i, k\} \rightsquigarrow Q$ **(I2)**.

Applying the induction hypothesis to **I1**, **I2** and **H1** gives goals **G1**-**G3** and $\mathtt{p} \vdash \langle \lfloor H \uplus \hat{H}_1 \rfloor, \rho, i, k \rangle \Downarrow_{\mathtt{m}} \langle \lfloor H_f \uplus \hat{H}_1 \rfloor, \rho_f, o \rangle$ **(I3)**. From the semantics and **I3**, goal **G4** follows.

[CONDITIONAL GOTO - TRUE] It follows that $\mathtt{p_m}(j) = \text{goto} \quad \mathtt{e} \quad k \quad l$. As we are in the conditional goto - true case we also have $[\![\mathtt{e}]\!]_\rho = \mathsf{true}$ **(I1)**. From **H2**, we consider the three possible cases:

- $P \vdash \mathtt{true} * \mathtt{e} \doteq \mathtt{true}$ **(I2)**

  We conclude, using **H3** and the semantics of JSIL, that $\mathtt{p} \vdash \langle \lfloor H \uplus \hat{H}_1 \uplus \hat{H}_2 \rfloor, \rho, j, k \rangle \Downarrow_{\mathtt{m}} \langle \mathtt{h}_f, \rho_f, o \rangle$ **(I3)**

  From **H2** and **I2** we conclude that $\mathtt{p}, \mathtt{S}, \mathtt{m} \vdash_{fl} \{P, j, k\} \rightsquigarrow Q$ **(I4)**. Applying the induction hypothesis to **I1**, **I4** and **H1**, gives $\mathtt{p} \vdash \langle \lfloor H \uplus \hat{H}_1 \rfloor, \rho, j, k \rangle \Downarrow_{\mathtt{m}} \langle \lfloor H_f \uplus \hat{H}_1 \rfloor, \rho_f, o \rangle$ **(I5)** and goals **G1**-**G3**. Using the semantics and **I5** we obtain $\mathtt{p} \vdash \langle \lfloor H \uplus \hat{H}_1 \rfloor, \rho, i, j \rangle \Downarrow_{\mathtt{m}} \langle \lfloor H_f \uplus \hat{H}_1 \rfloor, \rho_f, o \rangle$ **(G4)**.

- $P \vdash \mathtt{true} * \mathtt{e} \doteq \mathtt{false}$

  From **H1** we conclude $[\![\mathtt{e}]\!]_\rho = \mathsf{false}$, this is a contradiction of **I1**.

- $P \nvdash \mathtt{true} * \mathtt{e} \doteq \mathtt{false}$ and $P \nvdash \mathtt{true} * \mathtt{e} \doteq \mathtt{true}$ **(I2)**

  From **H2** and **I2** we conclude that $\mathtt{p}, \mathtt{S}, \mathtt{m} \vdash_{fl} \{P * (\mathtt{e} \doteq \mathtt{true}), j, k\} \rightsquigarrow Q$ **(I3)** and $\mathtt{p}, \mathtt{S}, \mathtt{m} \vdash_{fl} \{P * (\mathtt{e} \doteq \mathtt{false}), i, l\} \rightsquigarrow Q$. Using **H1** and **I1**, then $H, \rho, \epsilon \models P * (\mathtt{e} \doteq \mathtt{true})$ follows **(I4)**.

  By applying the semantics of JSIL and using **H3**, we conclude that $\mathtt{p} \vdash \langle \lfloor H \uplus \hat{H}_1 \uplus \hat{H}_2 \rfloor, \rho, j, k \rangle \Downarrow_{\mathtt{m}} \langle \lfloor H_f \uplus \hat{H}_1 \rfloor, \rho_f, o \rangle$ **(I5)** and goals **G1**-**G3**. We can then apply the inductive hypothesis to **I3**, **I4** and **I5** to obtain $\mathtt{p} \vdash \langle \lfloor H \uplus \hat{H}_1 \rfloor, \rho, i, j \rangle \Downarrow_{\mathtt{m}} \langle \lfloor H_f \uplus \hat{H}_1 \rfloor, \rho_f, o \rangle$ **(G4)**.

[CONDITIONAL GOTO - FALSE] Follows from conditional goto true case.

[PHI-ASSIGNMENT] It follows that $\mathtt{p}_{\mathtt{m}}(j) = \mathtt{x} := \phi(\mathtt{x}_1, ..., \mathtt{x}_n)$. We conclude, using **H3** and the semantics of JSIL, that:

$$i \overset{k}{\mapsto}_{\mathtt{m}} j \textbf{ (I1)} \qquad \mathtt{p} \vdash \langle \lfloor H \uplus \hat{H}_1 \uplus \hat{H}_2 \rfloor, \rho[\mathtt{x} \mapsto \rho(\mathtt{x}_k)], j, j+1 \rangle \Downarrow_{\mathtt{m}} \langle \mathtt{h}_f, \rho_f, o \rangle \textbf{ (I2)}$$

From **H2** we conclude that $\mathtt{p}, \mathtt{S}, \mathtt{m} \vdash_{fl} \{P * \mathtt{x} \doteq \mathtt{x}_k, j, j+1\} \rightsquigarrow Q$ **(I3)**. Noting that we assume programs are only written in SSA form, we can obtain from **H1** $H, \rho[\mathtt{x} \mapsto \rho(\mathtt{x}_k)], \epsilon \models P * \mathtt{x} \doteq \mathtt{x}_k$ **(I4)**.

Applying the induction hypothesis to **I3**, **I2** and **I4** we obtain goals **G1**-**G3** and $\mathtt{p} \vdash \langle \lfloor H \uplus \hat{H}_1 \rfloor, \rho[\mathtt{x} \mapsto \rho(\mathtt{x}_k)], j, j+1 \rangle \Downarrow_{\mathtt{m}} \langle \lfloor H_f \uplus \hat{H}_1 \rfloor, \rho_f, o \rangle$ **(I4)**. We can conclude from the semantics, **I4** and **I1** that $\mathtt{p} \vdash \langle \lfloor H \uplus \hat{H}_1 \rfloor, \rho, i, j \rangle \Downarrow_{\mathtt{m}} \langle \lfloor H_f \uplus \hat{H}_1 \rfloor, \rho_f, o \rangle$ **(G4)**.

[RETURN - NORMAL] It follows that $j = i_{\mathsf{nm}}$, $fl = \mathsf{nm}$ and $Q = Q'$. From the semantics and **H3** we conclude that $\lfloor H \uplus \hat{H}_1 \uplus \hat{H}_2 \rfloor = \mathtt{h}_f$ **(G3)** and $o = \mathsf{nm}\langle \rho(\mathtt{xret}) \rangle$ **(G2)**. From the semantics we obtain $\vdash \langle \lfloor H \uplus \hat{H}_1 \rfloor, \rho, \_, i_{\mathsf{nm}} \rangle \Downarrow_{\mathtt{m}} \langle \lfloor H \uplus \hat{H}_1 \rfloor, \rho, \mathsf{nm}\langle \rho(\mathtt{xret}) \rangle \rangle$ **(G4)**. From **H2**, we obtain $P \vdash Q'$ **(I1)**. We then obtain **G1** from **I1** and **H1**.

[RETURN - ERROR] Follows from the return normal case, with $j = i_{\mathsf{er}}$ and $fl = \mathsf{er}$. □

Chapter 4

Bi-Abductive Symbolic Analysis

Given a JSIL procedure, the bi-abductive symbolic analysis infers a set of specifications under which the execution of the given procedure does not fault. The analysis presented here is a first step towards a fully-fledged symbolic execution analysis. In particular, we do not provide the means for automatically inferring loop invariants and we cannot deal with recursive procedures. In this section, we first give an intuitive account of the bi-abductive analysis (Section 4.1), then we define the bi-abductive axioms for basic commands (Section 4.2) and the bi-abductive proof rules for control flow commands (Section 4.2), and we conclude with a description of the algorithm used to generate specifications for JSIL programs (Section 4.4).

The bi-abductive analysis builds on the symbolic execution given in Chapter 3. In particular, we establish the soundness of the bi-abductive analysis by appealing to the symbolic execution analysis. In a nutshell, we prove that all the specifications inferred using the bi-abductive symbolic analysis are verifiable using the standard symbolic analysis. This claim is made formally in Theorem 2.

4.1 Bi-Abductive Analysis: High Level Description

Here we give an intuitive account of the bi-abductive symbolic analysis. In contrast to the symbolic analysis, for which the initial precondition holds the entire resource which the program needs in order not to fault, in the bi-abductive symbolic analysis we have to compute the missing resource. Hence, in the bi-abductive symbolic analysis, we associate each program point with:

- an assertion P, describing the state (heap and store) when reaching that program point; and
- an assertion M, describing the resources that need to be added to the initial precondition so that the program can reach the current program point without faulting.

We refer to (P, M) as the bi-abductive symbolic state.

Suppose that (P, M) holds before the bi-abductive symbolic execution of a given command c. After analysing c, we obtain a new bi-abductive symbolic state (P', M'). The assertion P' describes the heap and store after executing c and M' extends M with the resources needed for the execution of c not to fault.

In order to better illustrate this, suppose we use the bi-abductive analysis to find the pre- and postconditions of a program containing the following JSIL code:

```
x := [timeout, interval]
y := convertToSeconds(x)
[timeout, interval] := y
```

Figure 4.1 describes how the bi-abductive symbolic analysis handles this code, comparing it with the normal symbolic execution. In the normal symbolic execution, we start with an assertion P, describing all the resources needed for the execution of these three commands not to fault. After symbolically executing the first command, we obtain P_1 (which holds all the resources for the execution of the last two commands not to fault). After execution the last two commands, we obtain P_2 and P_3, respectively. In contrast, in the bi-abductive symbolic analysis, we also need to update the assertion describing the missing initial resources. It is important to note that M is monotonically increasing. Therefore, in this example, $M_2 \vdash M_1 * \hat{M}$ where $\hat{M}$ describes the resource required to execute the command y = convertToSeconds(x). Ultimately, we use the final missing assertion M_3 to generate the precondition for the procedure and we use the final symbolic state P_3 for the postcondition.

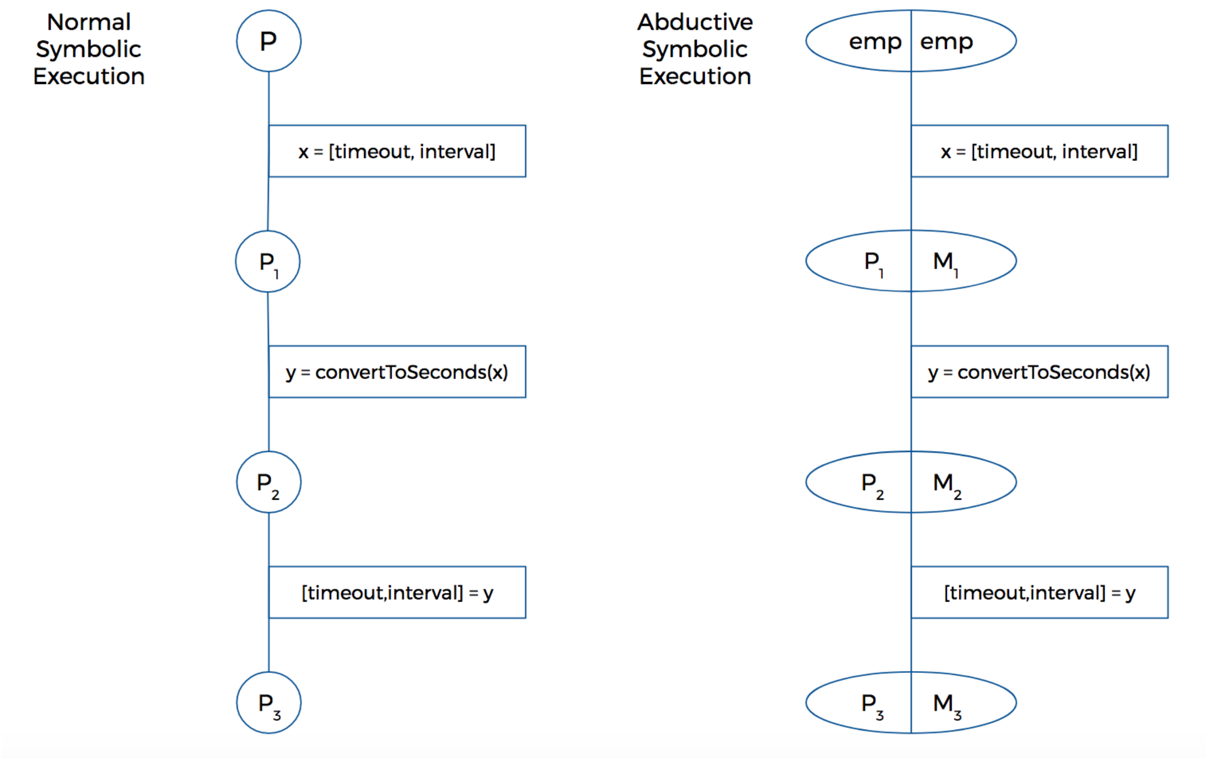

Figure 4.1: Comparison of normal and bi-abductive symbolic execution

In Figure 4.2, we show how the bi-abductive symbolic execution deals with JSIL branching. The execution branches when it is unable to symbolically evaluate the guard of a conditional goto to true or false. When this happens, the bi-abductive symbolic execution branches, obtaining two separate bi-abductive symbolic states: (P_3^T, M_3^T) (for the case in which we assume the guard to evaluate to true) and (P_3^E, M_3^E) (for the false case).

4.2 Bi-Abductive Axioms for Basic Commands

The hoare triples given in Figure 4.3 infer missing resources of the JSIL assertion P needed to execute the command bc. The triples take the form $\{P, M\}$ bc $\{Q, M'\}$, where:

- P is the precondition of bc and M the missing resources computed so far; and

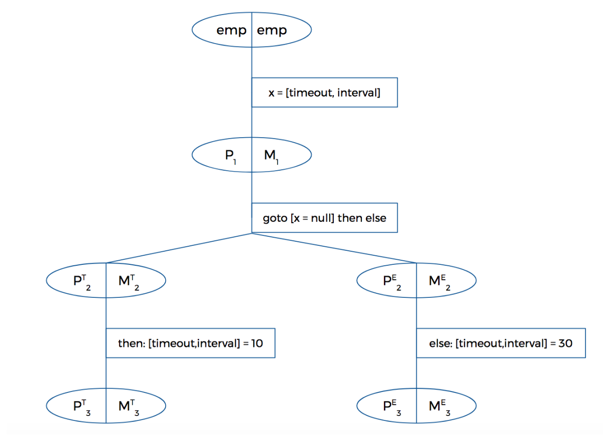
emp
emp
x = [timeout, interval]
P1
M1
goto [x = null] then else
PT2
MT2
PE2
ME2
then: [timeout,interval] = 10
else: [timeout,interval] = 30
PT3
MT3
PE3
ME3

Figure 4.2: Bi-Abductive symbolic execution with branching

- Q is the postcondition of `bc` and M' the extension of the missing resources for `bc` not to fault.

If $\{P, M\}$ `bc` $\{Q, M'\}$ holds, then there is an assertion $\hat{M}$ such that: $M' \vdash M * \hat{M}$ and $\left\{P * \hat{M}\right\}$ `bc` $\{Q\}$ where $\hat{M}$ describes the resources that have to be added to P for the execution of `bc` not to fault.

Basic Commands

$$\textsc{Field Assignment - Missing}\quad \frac{P \nvdash \texttt{true} * (\mathsf{e}_1, \mathsf{e}_2) \mapsto - \qquad Q = P * (\mathsf{e}_1, \mathsf{e}_2) \mapsto \mathsf{e}_3 \qquad M' = M * (\mathsf{e}_1, \mathsf{e}_2) \mapsto -}{\{P, M\}\ [\mathsf{e}_1, \mathsf{e}_2] := \mathsf{e}_3\ \{Q, M'\}}$$

$$\textsc{Field Deletion - Missing}\quad \frac{\begin{array}{c} P \vdash \texttt{true} * \mathsf{e}_2 \mathrel{\dot{\neq}} @\mathit{proto} \\ P \nvdash \texttt{true} * (\mathsf{e}_1, \mathsf{e}_2) \mapsto \mathrm{X} * \mathrm{X} \mathrel{\dot{\neq}} \varnothing \qquad Q = P * (\mathsf{e}_1, \mathsf{e}_2) \mapsto \varnothing \qquad M' = M * (\mathsf{e}_1, \mathsf{e}_2) \mapsto Y * Y \mathrel{\dot{\neq}} \varnothing \end{array}}{\{P, M\}\ \mathsf{delete}(\mathsf{e}_1, \mathsf{e}_2)\ \{Q, M'\}}$$

$$\textsc{Field Access - Missing}\quad \frac{\begin{array}{c} P \nvdash \texttt{true} * (\mathsf{e}_1, \mathsf{e}_2) \mapsto \mathrm{X} * \mathrm{X} \mathrel{\dot{\neq}} \varnothing \\ Q = P * \mathtt{x} \doteq \mathrm{X} * (\mathsf{e}_1, \mathsf{e}_2) \mapsto \mathrm{X} * \mathrm{X} \mathrel{\dot{\neq}} \varnothing \qquad M' = M * (\mathsf{e}_1, \mathsf{e}_2) \mapsto \mathrm{X} * \mathrm{X} \mathrel{\dot{\neq}} \varnothing \end{array}}{\{P, M\}\ \mathtt{x} := [\mathsf{e}_1, \mathsf{e}_2]\ \{Q, M'\}}$$

$$\textsc{Member Check - Missing}\quad \frac{P \nvdash \texttt{true} * (\mathsf{e}_1, \mathsf{e}_2) \mapsto - \qquad Q = P * (\mathsf{e}_1, \mathsf{e}_2) \mapsto Z * \mathtt{x} \doteq \mathsf{not}\,(Z = \varnothing) \qquad M' = M * (\mathsf{e}_1, \mathsf{e}_2) \mapsto Z}{\{P, M\}\ \mathtt{x} := \mathsf{hasField}(\mathsf{e}_1, \mathsf{e}_2)\ \{Q, M'\}}$$

$$\textsc{Basic Command}\quad \frac{\{P\}\ \mathtt{bc}\ \{Q\}}{\{P, M\}\ \mathtt{bc}\ \{Q, M\}}$$

Figure 4.3: Bi-Abductive Hoare Triples for Basic Commands

In order to illustrate the mechanics of the bi-abductive Hoare triples for basic commands, we give the following example:

$$\frac{\begin{array}{c} P = (\mathsf{timeout}, \mathsf{id}) \mapsto 17 \qquad P \nvdash \texttt{true} * (\mathsf{timeout}, \mathsf{interval}) \mapsto - \\ Q = P * (\mathsf{timeout}, \mathsf{interval}) \mapsto 400 \\ M' = (\mathsf{timeout}, \mathsf{id}) \mapsto - * (\mathsf{timeout}, \mathsf{interval}) \mapsto - \end{array}}{\{(\mathsf{timeout}, \mathsf{id}) \mapsto 17, (\mathsf{timeout}, \mathsf{id}) \mapsto -\} \ [\mathsf{timeout}, \mathsf{interval}] := 400 \ \{Q, M'\}} \textit{ Field Assignment - Missing}$$

We start in a state with the object timeout which has one field id and we have already inferred a part of the precondition, $(\mathsf{timeout}, \mathsf{id}) \mapsto -$. In order to perform a field assignment operation, we require that the object timeout has the field interval. Therefore the resulting assertion, $M' = (\mathsf{timeout}, \mathsf{id}) \mapsto - * (\mathsf{timeout}, \mathsf{interval}) \mapsto -$, consists of the previously inferred assertion plus the newly required heap cell. The resulting state after performing the field assignment is $Q = (\mathsf{timeout}, \mathsf{id}) \mapsto 17 * (\mathsf{timeout}, \mathsf{interval}) \mapsto 400$.

Evaluation. Lemma 2 establishes the soundness of the bi-abductive Hoare triples for basic commands. The lemma connects the triples to the non bi-abductive Hoare triples for basic commands.

Lemma 2 (Soundness of Bi-Abductive Hoare Triples for Basic Commands)**.** *For any assertions $P, M, Q, M' \in \mathcal{AS}_{\texttt{JSIL}}$ and basic commands* $\texttt{bc} \in BCmd$ *such that:*

- **H1***:* $\{P, M\}$ $\texttt{bc}$ $\{Q, M'\}$

It follows that there exists an $\hat{M} \in \mathcal{AS}_{\texttt{JSIL}}$ such that:

- **G1***:* $M' \vdash M * \hat{M}$
- **G2***:* $\left\{P * \hat{M}\right\}$ $\texttt{bc}$ $\{Q\}$

Proof. We proceed by case analysis on the structure of $\{P, M\}$ $\texttt{bc}$ $\{Q, M'\}$.

[Field Assignment - Missing] Given $\texttt{bc} = [\mathsf{e}_1, \mathsf{e}_2] := \mathsf{e}_3$ and $P \nvdash \texttt{true} * (\mathsf{e}_1, \mathsf{e}_2) \mapsto -$ from the bi-abductive field assignment missing basic command rule and **H1** we obtain: $Q = P*(\mathsf{e}_1, \mathsf{e}_2) \mapsto \mathsf{e}_3$ **(I1)** and $M' = M * (\mathsf{e}_1, \mathsf{e}_2) \mapsto -$ **(I2)**. From **I2** it follows that $\hat{M} = (\mathsf{e}_1, \mathsf{e}_2) \mapsto -$ where $M' \vdash M * \hat{M}$ **(I3)(G1)**. Using **I3** we obtain $\{P * (\mathsf{e}_1, \mathsf{e}_2) \mapsto -\}$ $\texttt{bc}$ $\{P * (\mathsf{e}_1, \mathsf{e}_2) \mapsto \mathsf{e}_3\}$ **(G2)** as shown in the proof derivation tree:

$$\frac{\dfrac{}{\{(\mathsf{e}_1, \mathsf{e}_2) \mapsto -\} \ [\mathsf{e}_1, \mathsf{e}_2] := \mathsf{e}_3 \ \{(\mathsf{e}_1, \mathsf{e}_2) \mapsto \mathsf{e}_3\}} \textit{ Field Assignment}}{\{P * (\mathsf{e}_1, \mathsf{e}_2) \mapsto -\} \ [\mathsf{e}_1, \mathsf{e}_2] := \mathsf{e}_3 \ \{P * (\mathsf{e}_1, \mathsf{e}_2) \mapsto \mathsf{e}_3\}} \textit{ Frame}$$

[Field Deletion - Missing] Given $\texttt{bc} = \mathsf{delete}(\mathsf{e}_1, \mathsf{e}_2)$ and $P \nvdash \texttt{true} * (\mathsf{e}_1, \mathsf{e}_2) \mapsto \mathrm{X} * \mathrm{X} \neq \varnothing$ from the bi-abductive field deletion missing basic command rule and **H1** we obtain $P \vdash \texttt{true} * (\mathsf{e}_2 \neq @proto)$ **(I1)**, $Q = P * (\mathsf{e}_1, \mathsf{e}_2) \mapsto \varnothing$ **(I2)** and $M' = M * ((\mathsf{e}_1, \mathsf{e}_2) \mapsto Y) * (Y \neq \varnothing)$ **(I3)**. From **I3** it follows that $\hat{M} = ((\mathsf{e}_1, \mathsf{e}_2) \mapsto Y) * (Y \neq \varnothing)$ where $M' \vdash M * \hat{M}$ **(I4)(G1)**. Using **I4** and **I1**, we obtain goal **G2** as shown in the proof derivation tree:

$$\frac{\dfrac{}{\{(\mathsf{e}_2 \neq @proto) * ((\mathsf{e}_1, \mathsf{e}_2) \mapsto Y) * (Y \neq \varnothing)\} \ \mathsf{delete}(\mathsf{e}_1, \mathsf{e}_2) \ \{(\mathsf{e}_1, \mathsf{e}_2) \mapsto \varnothing\}} \textit{ Field Deletion}}{\left\{P' * (\mathsf{e}_2 \neq @proto) * ((\mathsf{e}_1, \mathsf{e}_2) \mapsto Y) * (Y \neq \varnothing)\right\} \ \mathsf{delete}(\mathsf{e}_1, \mathsf{e}_2) \ \left\{P' * (\mathsf{e}_1, \mathsf{e}_2) \mapsto \varnothing\right\}} \textit{ Frame}$$

where $P \vdash P' * (\mathsf{e}_2 \neq @proto)$

[Field Access - Missing] Given $\mathtt{bc} = \mathtt{x} := [\mathsf{e}_1, \mathsf{e}_2]$ and $P \nvdash \mathtt{true} * (\mathsf{e}_1, \mathsf{e}_2) \mapsto \mathrm{X} * \mathrm{X} \neq \varnothing$ from the bi-abductive field access missing basic command rule and **H1** we obtain $Q = P * \mathtt{x} \doteq \mathrm{X} * (\mathsf{e}_1, \mathsf{e}_2) \mapsto \mathrm{X} * \mathrm{X} \neq \varnothing$ **(I1)** and $M' = M * (\mathsf{e}_1, \mathsf{e}_2) \mapsto \mathrm{X} * \mathrm{X} \neq \varnothing$ **(I2)** From **I2** it follows that $\hat{M} = (\mathsf{e}_1, \mathsf{e}_2) \mapsto \mathrm{X} * \mathrm{X} \neq \varnothing$ where $M' \vdash M * \hat{M}$ **(I3)(G1)**. Using **I3** and **I1**, we obtain goal **G2** as shown in the proof derivation tree:

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

[Member Check - Missing] Given $\mathtt{bc} = \mathtt{x} := \mathsf{hasField}(\mathsf{e}_1, \mathsf{e}_2)$ and $P \nvdash \mathtt{true} * (\mathsf{e}_1, \mathsf{e}_2) \mapsto -$ from the bi-abductive member check missing basic command rule and **H1** we obtain $Q = P * (\mathsf{e}_1, \mathsf{e}_2) \mapsto Z * \mathtt{x} \doteq \mathsf{not}\,(Z = \varnothing)$ **(I1)** and $M' = M * (\mathsf{e}_1, \mathsf{e}_2) \mapsto -$ **(I2)**. From **I2** it follows that $\hat{M} = (\mathsf{e}_1, \mathsf{e}_2) \mapsto Z$ where $M' \vdash M * \hat{M}$ **(I3)(G1)**. Using **I3** and **I1**, we obtain goal **G2** as shown in the proof derivation tree:

$$\dfrac{\dfrac{}{\{(\mathsf{e}_1, \mathsf{e}_2) \mapsto Z\}\ \mathtt{x} := \mathsf{hasField}(\mathsf{e}_1, \mathsf{e}_2)\ \{(\mathsf{e}_1, \mathsf{e}_2) \mapsto Z * \mathtt{x} \doteq \mathsf{not}\,(Z = \varnothing)\}}\ \textit{Member Check}}{\{P * (\mathsf{e}_1, \mathsf{e}_2) \mapsto Z\}\ \mathtt{x} := \mathsf{hasField}(\mathsf{e}_1, \mathsf{e}_2)\ \{P * (\mathsf{e}_1, \mathsf{e}_2) \mapsto Z * \mathtt{x} \doteq \mathsf{not}\,(Z = \varnothing)\}}\ \textit{Frame}$$

[Basic Command] Given $\{P, M\}\ \mathtt{bc}\ \{Q, M\}$ by applying the basic command rule we obtain $\{P\}\ \mathtt{bc}\ \{Q\}$ **(I1)**. Therefore, from **I1**, $\hat{M} = \mathsf{emp}$ where $M \vdash M * \mathsf{emp}$ **(G1)**. By applying the consequences rule we conclude $\{P * \mathsf{emp}\}\ \mathtt{bc}\ \{Q\}$ **(G2)**.

□

4.3 Bi-Abductive Proof Rules for Control Flow Commands

The bi-abductive proof rules for the symbolic execution of the control flow commands are given in Figure 4.4. The rules take the form $\mathtt{p}, \mathtt{S}, \mathtt{m} \vdash \{P, M, j, i\} \rightsquigarrow_{ab} \Pi$, where:

- $\mathtt{m} \in \mathcal{S}tr$ and $\mathtt{p} \in \mathtt{P}$ respectively denote the JSIL procedure name and program being analysed, and $\mathtt{S}$ the specification environment;
- i denotes the index of the JSIL command to be symbolically executed and j the index of the command that was symbolically executed immediately before i; and
- P is an assertion describing the precondition of the ith command, whereas M is an assertion describing the missing resources computed so far.

The result of the bi-abductive symbolic execution for control flow commands is a set Π consisting of triples of the form (P, M, fl) where: P is a possible postcondition, M is the anti-frame that needs be added to the precondition so that the execution of the procedure does not fault, and fl is the return flag for this execution. We briefly explain the more complicated rules below.

- [Conditional Goto - Unknown], captures the case when the expression e cannot be evaluated to `false` or `true`. Therefore, the symbolic execution explores both paths, adding the condition to both the current state and the missing state. However, the condition cannot be added to the missing state if it contains a variable not included in either the collected missing state so far or the procedure arguments. Adding such a variable would entail the generation of a non-well-formed specification. The final result is the union of the results from each path.

- [Return] normal and error, these rules return a tuple containing the current state, the collected missing state and the return flag. The return flag is dependant on the index of the final command.

- [Procedure call] rule includes the solving of the bi-abductive problem. More concretely, given the current state P and the callee's precondition P', the system needs to find the assertions P_M and P_F, such that $P * P_M \vdash P' * P_F$. The required missing state is then added to the collected anti-frame and the untouched frame state is combined with the callee's postcondition.

Example. We present a swap procedure example in order to illustrate the bi-abductive proof rules. In particular, if the current node argument is `null` the procedure returns `null`. Otherwise, the procedure swaps the next and previous fields of the current node. In the example we show how the current state is transformed and how the missing state is collected. The result contains two tuples, one with the final state and missing state of the `null` case and the other with the swap case. Both cases have normal return flags.

```
proc swap (cur_node) {
  State: { emp }
  Missing: { emp }
  goto [cur_node = null] then else;
  State: { $cur\_node \doteq$ null }
  Missing: { $cur\_node \doteq$ null }
  then:     ret := null;
             State: { $cur\_node \doteq$ null $*$ $ret \doteq null$ }
             Missing: { $cur\_node \doteq$ null }
            goto rlab;
  State: { $cur\_node \not\doteq$ null }
  Missing: { $cur\_node \not\doteq$ null }
  else:     prev := [cur_node, "prev"];
           State: { $cur\_node \not\doteq$ null $*$ $(cur\_node, \text{“}prev\text{”}) \mapsto \#prev$ $*$ $\#prev \not\doteq \varnothing$
                    $*$ $prev \doteq \#prev$ }
           Missing: { $cur\_node \not\doteq$ null $*$ $(cur\_node, \text{“}prev\text{”}) \mapsto \#prev$ $*$ $\#prev \not\doteq \varnothing$ }
            next := [cur_node, "next"];
            State: { $cur\_node \not\doteq$ null $*$ $(cur\_node, \text{“}prev\text{”}) \mapsto \#prev$ $*$ $\#prev \not\doteq \varnothing$
                     $*$ $prev \doteq \#prev$ $*$ $(cur\_node, \text{“}next\text{”}) \mapsto \#next$ $*$ $\#next \not\doteq \varnothing$
                     $*$ $next \doteq \#next$ }
            Missing: { $cur\_node \not\doteq$ null $*$ $(cur\_node, \text{“}prev\text{”}) \mapsto \#prev$ $*$ $\#prev \not\doteq \varnothing$
                       $*$ $(cur\_node, \text{“}next\text{”}) \mapsto \#next$ $*$ $\#next \not\doteq \varnothing$ }
            [cur_node, "prev"] := next;
            State:
              { $cur_node \not\doteq$ null $*$ $(cur\_node, \text{“}prev\text{”}) \mapsto \#next$ $*$ $\#prev \not\doteq \varnothing$
                $*$ $prev \doteq \#prev$ $*$ $(cur\_node, \text{“}next\text{”}) \mapsto \#next$ $*$ $\#next \not\doteq \varnothing$ $*$
                $next \doteq \#next$ }
            Missing: { $cur\_node \not\doteq$ null $*$ $(cur\_node, \text{“}prev\text{”}) \mapsto \#prev$ $*$ $\#prev \not\doteq \varnothing$
                       $*$ $(cur\_node, \text{“}next\text{”}) \mapsto \#next$ $*$ $\#next \not\doteq \varnothing$ }
            [cur_node, "next"] := prev;
            State: { $cur\_node \not\doteq$ null $*$ $(cur\_node, \text{“}prev\text{”}) \mapsto \#next$ $*$ $\#prev \not\doteq \varnothing$
                     $*$ $prev \doteq \#prev$ $*$ $(cur\_node, \text{“}next\text{”}) \mapsto \#prev$ $*$ $\#next \not\doteq \varnothing$ $*$
                     $next \doteq \#next$ }
            Missing: $\left\{ \begin{array}{l} cur\_node \not\doteq \mathsf{null} \;*\; (cur\_node, \text{“}prev\text{”}) \mapsto \#prev \;*\; \#prev \not\doteq \varnothing \\ \;*\; (cur\_node, \text{“}next\text{”}) \mapsto \#next \;*\; \#next \not\doteq \varnothing \end{array} \right\}$
            ret := next;
            State: $\left\{ \begin{array}{l} cur\_node \not\doteq \mathsf{null} \;*\; (cur\_node, \text{“}prev\text{”}) \mapsto \#next \;*\; \#prev \not\doteq \varnothing \\ \;*\; prev \doteq \#prev \;*\; (cur\_node, \text{“}next\text{”}) \mapsto \#prev \;*\; \#next \not\doteq \varnothing \;* \\ next \doteq \#next \;*\; ret = \#next \end{array} \right\}$
            Missing: $\left\{ \begin{array}{l} cur\_node \not\doteq \mathsf{null} \;*\; (cur\_node, \text{“}prev\text{”}) \mapsto \#prev \;*\; \#prev \not\doteq \varnothing \\ \;*\; (cur\_node, \text{“}next\text{”}) \mapsto \#next \;*\; \#next \not\doteq \varnothing \end{array} \right\}$
    rlab:   skip
} with {ret: ret, rlab;};
```

This bi-abductive symbolic execution generates two specifications, given below.

We give the specification for the case where the current node is `null`:

`Precondition:`
$\{\; cur\_node \doteq \mathsf{null} \;\}$

`Postcondition:`
$\{\; cur\_node \doteq \mathsf{null} \;*\; ret \doteq \mathsf{null} \;\}$

`Flag:`
`Normal`

We give the specification for the case where the current node is not `null`, and therefore the values of previous and next have been swapped.

`Precondition:`
$$\left\{ \begin{array}{l} cur\_node \not\doteq \mathsf{null} \;*\; (cur\_node, \text{“}prev\text{”}) \mapsto \#prev \;*\; \#prev \not\doteq \varnothing \\ \;*\; (cur\_node, \text{“}next\text{”}) \mapsto \#next \;*\; \#next \not\doteq \varnothing \end{array} \right\}$$

`Postcondition:`
$$\left\{ \begin{array}{l} cur\_node \not\doteq \mathsf{null} \;*\; (cur\_node, \text{“}prev\text{”}) \mapsto \#next \;*\; \#prev \not\doteq \varnothing \\ \;*\; (cur\_node, \text{“}next\text{”}) \mapsto \#prev \;*\; \#next \not\doteq \varnothing \;*\; ret = \#next \end{array} \right\}$$

`Flag:`
`Normal`

Evaluation. We establish the soundness of the bi-abductive control flow rules in Theorem 2. We assume the specifications of all callee's a procedure will use are in the specification table and there is no recursion in the program. In this proof we connect the bi-abductive control flow rules to the normal control flow rules. We must also update the specification environment in order to ensure a successful termination of the normal symbolic execution.

Theorem 2 (Soundness of Bi-Abductive Control Flow Proof Rules). *For all programs* $\mathtt{p} \in \mathtt{P}$*, specifications* $\mathtt{S} \in \mathcal{S}pec$*, procedure names* $\mathtt{m} \in \mathcal{S}tr$*, assertions* $P, M \in \mathcal{AS}_{\mathrm{JSIL}}$ *and command labels* j *and* i *such that:*

- **H1**: $\mathtt{p}, \mathtt{S}, \mathtt{m} \vdash \{P, M, j, i\} \rightsquigarrow_{ab} \Pi$

It follows that for all $(Q_f, M_f, fl_f) \in \Pi$ *there exists a JSIL assertions* $\hat{M}'', \hat{P} \in \mathcal{AS}_{\mathrm{JSIL}}$ *such that:*

- **G1**: $M_f \vdash M * \hat{M}''$
- **G2**: $\mathtt{S}' = \mathtt{S}[(\mathtt{m}, fl_f) \mapsto \left\{\hat{P}\right\}\; \mathtt{m}(\overline{\mathtt{x}})\; \{Q_f\}]$
- **G3**: $\mathtt{p}, \mathtt{S}', \mathtt{m} \vdash_{fl_f} \{P * \hat{M}'', j, i\} \rightsquigarrow Q_f$

Proof. We proceed by induction on the structure of bi-abductive control flow proof rules.

[BASIC COMMAND] Given $p_m(i) = \texttt{bc}$ where $\texttt{bc} \in BCmd$ **(I1)**, using the control flow rules and **H1** we obtain $\{P, M\}\ \texttt{bc}\ \{Q, M'\}$ **(I2)** and $\texttt{p}, \texttt{S}, \texttt{m} \vdash \{Q, M', i, i+1\} \rightsquigarrow_{ab} \Pi$ **(I3)** Applying the induction hypothesis to **I3**, we obtain that there exists some $\hat{M}'$ and P' such that $\texttt{p}, \texttt{S}', \texttt{m} \vdash_{fl_f} \{Q * \hat{M}', i, i+1\} \rightsquigarrow Q_f$ **(I4)** where $M_f \vdash M' * \hat{M}'$ **(I5)** and $\texttt{S}' = \texttt{S}[(\texttt{m}, fl_f) \mapsto \{P'\}\ \texttt{m}(\overline{\texttt{x}})\ \{Q_f\}]$ **(G2)**. From **I2** and Lemma 2, it follows that $\left\{P * \hat{M}\right\}\ \texttt{bc}\ \{Q\}$ **(I6)** where $M' \vdash M * \hat{M}$ **(I7)**. Hence using **I5** and **I7** we are able to obtain, $M_f \vdash M * \hat{M} * \hat{M}'$ where $\hat{M}'' = \hat{M}' * \hat{M}$ **(G1)**. We apply the frame rule to **I6**, which results in $\left\{P * \hat{M} * \hat{M}'\right\}\ \texttt{bc}\ \left\{Q * \hat{M}'\right\}$ **(I7)**. Using **I1**, **I3** and **I7**, we obtain goal **G3** as shown in the proof derivation:

$$\frac{\begin{array}{c} p_m(i) = \texttt{bc} \qquad \texttt{bc} \in BCmd \\ \left\{P * \hat{M}''\right\}\ \texttt{bc}\ \left\{Q * \hat{M}'\right\} \qquad \texttt{p}, \texttt{S}', \texttt{m} \vdash_{fl_f} \{Q * \hat{M}', i, i+1\} \rightsquigarrow Q_f \end{array}}{\texttt{p}, \texttt{S}', \texttt{m} \vdash_{fl_f} \{P * \hat{M}'', j, i\} \rightsquigarrow Q_f}\ \textit{Basic Command}$$

[CONDITIONAL GOTO - UNKNOWN] Given $p_m(i) = \text{goto} \quad e \quad k \quad l$ **(I1)**, $P \nvdash \texttt{true} * (\texttt{e} \doteq \texttt{false})$ **(I2)** and $P \nvdash \texttt{true} * (\texttt{e} \doteq \texttt{true})$ **(I3)**, we conclude using the control flow rules and **H1**:

$$\begin{array}{l} \texttt{p}, \texttt{S}, \texttt{m} \vdash \{P * (\texttt{e} \doteq \texttt{true}), M * (\texttt{e} \doteq \texttt{true}), i, k\} \rightsquigarrow_{ab} \Pi_t \ \textbf{(I4)} \\ \texttt{p}, \texttt{S}, \texttt{m} \vdash \{P * (\texttt{e} \doteq \texttt{false}), M * (\texttt{e} \doteq \texttt{false}), i, l\} \rightsquigarrow_{ab} \Pi_e \ \textbf{(I5)} \\ vars(\texttt{e}) \subseteq vars(M) \cup vars(\texttt{m}) \ \textbf{(I6)} \qquad \Pi = \Pi_t \cup \Pi_e \ \textbf{(I7)} \end{array}$$

From **I7** there are two possible cases, either $(Q_f, M_f, fl_f) \in \Pi_t$ or $(Q_f, M_f, fl_f) \in \Pi_e$. We take the case where $(Q_f, M_f, fl_f) \in \Pi_t$ **(I8)**. Applying the induction hypothesis to **I4**, we obtain that there exists some $\hat{M}$ and P' such that $\texttt{S}' = \texttt{S}[(\texttt{m}, fl_f) \mapsto \{P'\}\ \texttt{m}(\overline{\texttt{x}})\ \{Q_f\}]$ **(I9)** and $\texttt{p}, \texttt{S}', \texttt{m} \vdash_{fl_f} \{P * (\texttt{e} \doteq \texttt{true}) * \hat{M}, i, k\} \rightsquigarrow Q_f$ **(I10)** where $M_f \vdash M * (\texttt{e} \doteq \texttt{true}) * \hat{M}$ **(I11)**. Using **I1** and **I10**, we obtain goal **G3** as shown in the proof derivation:

$$\frac{\begin{array}{c} p_m(i) = \text{goto} \quad e \quad k \quad l \qquad P * (\texttt{e} \doteq \texttt{true}) * \hat{M} \vdash \texttt{true} * (\texttt{e} \doteq \texttt{true}) \\ \texttt{p}, \texttt{S}', \texttt{m} \vdash_{fl_f} \{P * (\texttt{e} \doteq \texttt{true}) * \hat{M}, i, k\} \rightsquigarrow Q_f \end{array}}{\texttt{p}, \texttt{S}', \texttt{m} \vdash_{fl_f} \{P * (\texttt{e} \doteq \texttt{true}) * \hat{M}, j, i\} \rightsquigarrow Q_f}\ \textit{Conditional Goto - True}$$

The case $(Q_f, M_f, fl_f) \in \Pi_e$ follows from the previous case.

[PROCEDURE CALL - NORMAL] Given $\texttt{p}_\texttt{m}(i) = \texttt{x} := \texttt{e}_0(\texttt{e}_1, ..., \texttt{e}_{n_1})$ with k **(I1)**, then we obtain using the control flow rules and **H1**:

$$\begin{array}{l} \texttt{S}(\texttt{m}', \texttt{nm}) = \{P'\}\ \texttt{m}'(\texttt{x}_1, ..., \texttt{x}_{\texttt{n}_2})\ \{Q' * \texttt{xret} \doteq \texttt{e}\} \ \textbf{(I2)} \\ \texttt{e}_n = \text{undefined}\ |_{n=n_1+1}^{n2} \ \textbf{(I3)} \\ P * P_M \vdash P_F * P'[\texttt{e}_i/\texttt{x}_i|_{i=1}^{n_2}] * \texttt{e}_0 \doteq \texttt{m}' \ \textbf{(I4)} \\ Q'' = P_F * Q'[\texttt{e}_i/\texttt{x}_i|_{i=1}^{n_2}] * \texttt{e}_0 \doteq \texttt{m}' * \texttt{x} \doteq \texttt{e}[\texttt{e}_i/\texttt{x}_i|_{i=1}^{n_2}] \ \textbf{(I5)} \\ \texttt{p}, \texttt{S}, \texttt{m} \vdash \{Q'', M * P_M, i, i+1\} \rightsquigarrow_{ab} \Pi \ \textbf{(I6)} \end{array}$$

Applying the induction hypothesis to **I6**, we obtain that there exists some $\hat{M}$ and $\hat{P}$ such that $\texttt{p}, \texttt{S}', \texttt{m} \vdash_{fl_f} \{Q'' * \hat{M}, i, i+1\} \rightsquigarrow Q_f$ **(I7)** where $M_f \vdash M * P_M * \hat{M}$ **(I8)**(**G1**) and $\texttt{S}' = \texttt{S}[(\texttt{m}, fl_f) \mapsto \left\{\hat{P}\right\}\ \texttt{m}(\overline{\texttt{x}})\ \{Q_f\}]$ **(G2)**. From **I5**, $Q'' * \hat{M} = P_F * Q'[\texttt{e}_i/\texttt{x}_i|_{i=1}^{n_2}] * \texttt{e}_0 \doteq \texttt{m}' * \texttt{x} \doteq \texttt{e}[\texttt{e}_i/\texttt{x}_i|_{i=1}^{n_2}] * \hat{M}$ **(I9)**. Additionally, from **I4** it follows that $P * P_M * \hat{M} \vdash P_F * P'[\texttt{e}_i/\texttt{x}_i|_{i=1}^{n_2}] * \texttt{e}_0 \doteq \texttt{m}' * \hat{M}$ **(I10)**.

Using **I1**,**I2**,**I3**,**I7**,**I9** and **I10**, we obtain goal **G3** as shown in the proof derivation:

$$\dfrac{\begin{array}{c} \mathtt{p_m}(i) = \mathtt{x} := \mathtt{e}_0(\mathtt{e}_1, ..., \mathtt{e}_{n_1}) \text{ with } k \\ \mathtt{S}(\mathtt{m}', \mathsf{nm}) = \{P'\}\ \mathtt{m}'(\mathtt{x}_1, ..., \mathtt{x}_{\mathtt{n}_2})\ \{Q' * \mathtt{xret} \doteq \mathtt{e}\} \\ \mathtt{e}_n = \mathsf{undefined}\ |_{n=n_1+1}^{n2} \\ P * P_M * \hat{M} \vdash P_F * P'[\mathtt{e}_i/\mathtt{x}_i|_{i=1}^{n_2}] * \mathtt{e}_0 \doteq \mathtt{m}' * \hat{M} \\ Q'' * \hat{M} = P_F * Q'[\mathtt{e}_i/\mathtt{x}_i|_{i=1}^{n_2}] * \mathtt{e}_0 \doteq \mathtt{m}' * \mathtt{x} \doteq \mathtt{e}[\mathtt{e}_i/\mathtt{x}_i|_{i=1}^{n_2}] * \hat{M} \\ \mathtt{p}, \mathtt{S}', \mathtt{m} \vdash_{fl_f} \{Q'' * \hat{M}, i, i+1\} \rightsquigarrow Q_f \end{array}}{\mathtt{p}, \mathtt{S}', \mathtt{m} \vdash_{fl_f} \{P * P_M * \hat{M}, j, i\} \rightsquigarrow Q_f} \textit{ Procedure Call - Normal}$$

[PROCEDURE CALL - ERROR] Follows from the procedure call normal case.

[RETURN - NORMAL] Given $i = i_{\mathsf{nm}}$, then we obtain using the control flow rules and **H1** that $\Pi = \{(P, M, \mathsf{nm})\}$ (**I1**). Therefore $M_f = M$ and $\hat{M}'' = \mathsf{emp}$, where $M_f \vdash M * \mathsf{emp}$ (**G1**). We set the new specification environment as follows $\mathtt{S}' = \mathtt{S}[(\mathtt{m}, \mathsf{nm}) \mapsto \{P'\}\ \mathtt{m}(\overline{\mathtt{x}})\ \{P\}]$ (**G2**). Therefore when we lookup the specification in the non bi-abductive normal return rule we get $\mathtt{S}'(\mathtt{m}, \mathsf{nm}) = \{P''\}\ \mathtt{m}(\mathtt{x}_1, ..., \mathtt{x}_{\mathtt{n}_2})\ \{Q''\}$ (**I2**) where $P'' = P' * M$ and $Q'' = P$ (**I3**). Due to **I3** we obtain $P \vdash Q''$ (**I4**). Using **I2** and **I4**, we obtain goal **G3** as shown in the proof derivation tree:

$$\dfrac{\mathtt{S}'(\mathtt{m}, \mathsf{nm}) = \{P''\}\ \mathtt{m}(\mathtt{x}_1, ..., \mathtt{x}_{\mathtt{n}_2})\ \{Q''\} \qquad P \vdash Q''}{\mathtt{p}, \mathtt{S}', \mathtt{m} \vdash_{\mathsf{nm}} \{P * \mathsf{emp}, j, i_{\mathsf{nm}}\} \rightsquigarrow P} \textit{ Return - Normal}$$

[RETURN - ERROR] Follows from the return normal case.

[ALL OTHER CASES] The phi-assignment, goto, conditional goto true and conditional goto false follow as $\hat{M} = \mathsf{emp}$.

□

4.4 Bi-Abductive Algorithm for JSIL Programs

We provide a procedure level algorithm which converts the result of the symbolic execution proof rules into a specification table. We infer specifications for non recursive procedures. In particular, we start by inferring the specifications of procedures with no procedure calls. We then proceed to infer the specifications of the procedures which only call procedures with already inferred specifications and so on, until all specifications have been inferred. For this algorithm, we assume the procedures in the program p given as input are ordered as such.

Algorithm 1 presents the algorithm to infer specifications of a JSIL program p. The specification environment, S, contains partial specifications. These specifications are optional for each procedure and are not assumed to be correct.

First, the algorithm initialises the new specification environment S′, which will contain the inferred specifications. Next, it loops through all procedures in order. If a partial specification is provided, the initial state is the partial precondition. Otherwise, the initial state is empty. Additionally, the symbolic execution always starts with an empty missing state and the 0-th command.

The algorithm then takes the result of the symbolic execution of the procedure, and loops through all the resulting tuples. It checks if every final state Q_f entails the partial postcondition if one is provided. Similar to the procedure call rule, this involves solving the bi-abductive problem. More concretely, given a partial postcondition Q and the computed postcondition Q_f, we have to find Q_M such that $Q_f * Q_M \vdash Q * \texttt{true}$. Intuitively, Q_M describes the missing resources that are required by the provided partial postcondition.

The new specification environment is then extended with the inferred specification. The partial precondition is combined with both the missing resources required for the procedure not to fault as well as the missing resources required by the partial postcondition. This becomes the new precondition. The final state combined with missing resources required by the partial postcondition becomes the new postcondition.

The $InferSpec$ function is defined as follows:

$$\Pi := \mathit{InferSpec}(\texttt{p}, \texttt{S}, \texttt{m}, P) \overset{\text{def}}{\Leftrightarrow} \texttt{p}, \texttt{S}, \texttt{m} \vdash \{P, \textsf{emp}, \_, 0\} \rightsquigarrow_{ab} \Pi$$

Algorithm 1 Procedure Level Algorithm

```
 1: $\texttt{S}' := \varnothing$
 2: for all proc $\texttt{m}(\overline{\texttt{x}})\{\overline{\texttt{c}}\} \in \texttt{p}$ do
 3:     if $(\texttt{m}, \_) \in \texttt{S}$ then
 4:         $\{P\}\, \texttt{m}(\overline{\texttt{x}})\, \{Q\} = \texttt{S}(\texttt{m}, \_)$
 5:     else
 6:         $P, Q := \textsf{emp}$
 7:     end if
 8:     $\Pi := \mathit{InferSpec}(\texttt{p}, \texttt{S}', \texttt{m}, P)$
 9:     for all $(Q_f, M_f, fl) \in \Pi$ do
10:         if $Q_f * Q_M \vdash Q * \texttt{true}$ then
11:             $\texttt{S}' := \texttt{S}' \cup \{(\texttt{m}, fl) \mapsto \{P * M_f * Q_M\}\ \texttt{m}(\overline{\texttt{x}})\ \{Q_f * Q_M\}\}$
12:         end if
13:     end for
14: end for
```

Control Flow Commands

Basic Command
$$\frac{p_m(i) = \texttt{bc} \quad \texttt{bc} \in BCmd \quad \{P, M\}\ \texttt{bc}\ \{Q, M'\} \quad \texttt{p}, \texttt{S}, \texttt{m} \vdash \{Q, M', i, i+1\} \rightsquigarrow_{ab} \Pi}{\texttt{p}, \texttt{S}, \texttt{m} \vdash \{P, M, \_, i\} \rightsquigarrow_{ab} \Pi}$$

Goto
$$\frac{p_m(i) = \text{goto} \quad k \quad \texttt{p}, \texttt{S}, \texttt{m} \vdash \{P, M, i, k\} \rightsquigarrow_{ab} \Pi}{\texttt{p}, \texttt{S}, \texttt{m} \vdash \{P, M, \_, i\} \rightsquigarrow_{ab} \Pi}$$

Conditional Goto - True
$$\frac{p_m(i) = \text{goto} \quad \texttt{e} \quad k \quad j \quad P \vdash \texttt{true} * \texttt{e} \doteq \texttt{true} \quad \texttt{p}, \texttt{S}, \texttt{m} \vdash \{P, M, i, k\} \rightsquigarrow_{ab} \Pi}{\texttt{p}, \texttt{S}, \texttt{m} \vdash \{P, M, \_, i\} \rightsquigarrow_{ab} \Pi}$$

Conditional Goto - False
$$\frac{p_m(i) = \text{goto} \quad \texttt{e} \quad k \quad j \quad P \vdash \texttt{true} * \texttt{e} \doteq \texttt{false} \quad \texttt{p}, \texttt{S}, \texttt{m} \vdash \{P, M, i, j\} \rightsquigarrow_{ab} \Pi}{\texttt{p}, \texttt{S}, \texttt{m} \vdash \{P, M, \_, i\} \rightsquigarrow_{ab} \Pi}$$

Conditional Goto - Unknown
$$\frac{\begin{array}{c} p_m(i) = \text{goto} \quad \texttt{e} \quad k \quad j \quad vars(\texttt{e}) \subseteq vars(M) \cup vars(\texttt{m}) \quad P \nvdash \texttt{true} * \texttt{e} \doteq \texttt{false} \\ P \nvdash \texttt{true} * \texttt{e} \doteq \texttt{true} \quad \texttt{p}, \texttt{S}, \texttt{m} \vdash \{P * (\texttt{e} \doteq \texttt{true}), M * (\texttt{e} \doteq \texttt{true}), i, k\} \rightsquigarrow_{ab} \Pi \\ \texttt{p}, \texttt{S}, \texttt{m} \vdash \{P * (\texttt{e} \doteq \texttt{false}), M * (\texttt{e} \doteq \texttt{false}), i, j\} \rightsquigarrow_{ab} \Pi' \end{array}}{\texttt{p}, \texttt{S}, \texttt{m} \vdash \{P, M, \_, i\} \rightsquigarrow_{ab} \Pi \cup \Pi'}$$

Phi-Assignment
$$\frac{p_m(i) = \texttt{x} := \phi(\texttt{x}_1, ..., \texttt{x}_n) \quad j \overset{k}{\mapsto}_m i \quad \texttt{p}, \texttt{S}, \texttt{m} \vdash \{P * \texttt{x} \doteq \texttt{x}_k, M, i, i+1\} \rightsquigarrow_{ab} \Pi}{\texttt{p}, \texttt{S}, \texttt{m} \vdash \{P, M, j, i\} \rightsquigarrow_{ab} \Pi}$$

Procedure Call - Normal
$$\frac{\begin{array}{c} \texttt{p}_\texttt{m}(i) = \texttt{x} := \texttt{e}_0(\texttt{e}_1, ..., \texttt{e}_{n_1}) \text{ with } k \quad \texttt{S}(\texttt{m}', \text{nm}) = \{P'\}\ \texttt{m}'(\texttt{x}_1, ..., \texttt{x}_{\texttt{n}_2})\ \{Q' * \texttt{xret} \doteq \texttt{e}\} \\ \texttt{e}_n = \text{undefined } |_{n=n_1+1}^{n2} \quad P * P_M \vdash P_F * P'[\texttt{e}_i/\texttt{x}_i|_{i=1}^{n_2}] * \texttt{e}_0 \doteq \texttt{m}' \\ Q'' = P_F * Q'[\texttt{e}_i/\texttt{x}_i|_{i=1}^{n_2}] * \texttt{e}_0 \doteq \texttt{m}' * \texttt{x} \doteq \texttt{e}[\texttt{e}_i/\texttt{x}_i|_{i=1}^{n_2}] \quad \texttt{p}, \texttt{S}, \texttt{m} \vdash \{Q'', M * P_M, i, i+1\} \rightsquigarrow_{ab} \Pi \end{array}}{\texttt{p}, \texttt{S}, \texttt{m} \vdash \{P, M, \_, i\} \rightsquigarrow_{ab} \Pi}$$

Procedure Call - Error
$$\frac{\begin{array}{c} \texttt{p}_\texttt{m}(i) = \texttt{x} := \texttt{e}_0(\texttt{e}_1, ..., \texttt{e}_{n_1}) \text{ with } k \quad \texttt{S}(\texttt{m}', \text{er}) = \{P'\}\ \texttt{m}'(\texttt{x}_1, ..., \texttt{x}_{\texttt{n}_2})\ \{Q' * \texttt{xerr} \doteq \texttt{e}\} \\ \texttt{e}_n = \text{undefined } |_{n=n_1+1}^{n2} \quad P * P_M \vdash P_F * P'[\texttt{e}_i/\texttt{x}_i|_{i=1}^{n_2}] * \texttt{e}_0 \doteq \texttt{m}' \\ Q'' = P_F * Q'[\texttt{e}_i/\texttt{x}_i|_{i=1}^{n_2}] * \texttt{e}_0 \doteq \texttt{m}' * \texttt{x} \doteq \texttt{e}[\texttt{e}_i/\texttt{x}_i|_{i=1}^{n_2}] \quad \texttt{p}, \texttt{S}, \texttt{m} \vdash \{Q'', M * P_M, i, k\} \rightsquigarrow_{ab} \Pi \end{array}}{\texttt{p}, \texttt{S}, \texttt{m} \vdash \{P, M, \_, i\} \rightsquigarrow_{ab} \Pi}$$

Return - Normal
$$\frac{}{\texttt{p}, \texttt{S}, \texttt{m} \vdash \{P, M, \_, i_{\text{nm}}\} \rightsquigarrow_{ab} \{(P, M, \text{nm})\}}$$

Return - Error
$$\frac{}{\texttt{p}, \texttt{S}, \texttt{m} \vdash \{P, M, \_, i_{\text{er}}\} \rightsquigarrow_{ab} \{(P, M, \text{er})\}}$$

Figure 4.4: Abductive Control Flow Command Proof Rules

Chapter 5

Implementation for Bi-Abductive Symbolic Analysis

We give an implementation of the bi-abductive symbolic analysis. Given a JSIL Program, the tool, called JSIL Abduce, infers specifications for the JSIL procedures in the program. We discuss the architecture and limitations of the tool (Section 5.1) and present some of the implementation challenges of the tool (Section 5.2).

5.1 Architecture

JSIL Abduce generates specifications for JSIL procedures, its architecture is given in Figure 5.1. The tool implements the bi-abductive symbolic execution rules and is entirely written in OCaml. JSIL Abduce has two main components: a bi-abductive symbolic execution engine and a structural entailment engine. As input the tools takes JSIL programs and optional partial specifications. The tool verifies each procedure independently, this modular style allows the tool to scale to large programs.

Before symbolic execution begins, a call graph is generated in order to determine the order in which procedures should be analysed. Procedures which call no other procedures are analysed first. Then all procedures which only call procedures with generated specifications are analysed and so on, until all procedures have been analysed. After the symbolic execution, the result of multiple branches are combined. If the tool fails to generate a specification or a branch contains recursion the symbolic execution for that branch ends. Specifications are still generated for the remaining branches.

Symbolic States. The symbolic execution engine transforms symbolic states $(h, \rho, \pi, \Gamma, \delta)$. A symbolic state contains:

- A symbolic heap $h : \mathcal{L} \times \mathcal{X}_{\texttt{JSIL}} \rightarrow \mathcal{V}_{\texttt{JSIL}}$ which is a mapping from locations and JSIL variables to JSIL values. Where the locations and JSIL variables represent objects and their fields respectively.
- A variable store $\rho : \mathcal{X}_{\texttt{JSIL}} \rightarrow \mathcal{V}_{\texttt{JSIL}}$ which is a mapping from JSIL variables to JSIL values.
- A set of pure formulae $\pi : \mathcal{AS}_{\texttt{JSIL}}$ which is a set of JSIL assertions.
- A typing environment $\Gamma : \mathcal{X}_{\texttt{JSIL}} \rightarrow \texttt{Types}$ which is a map from JSIL variables to JSIL types.

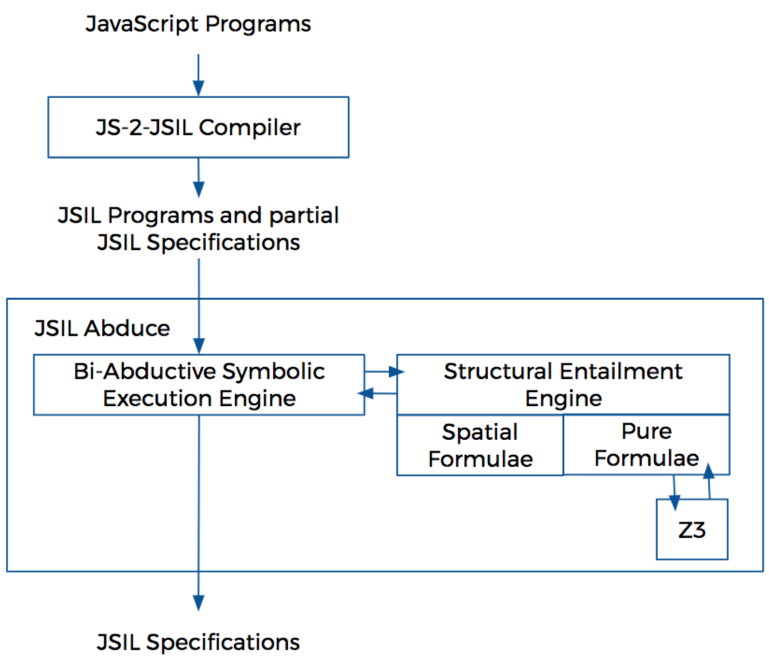
JavaScript Programs
JS-2-JSIL Compiler
JSIL Programs and partial JSIL Specifications
JSIL Abduce
Bi-Abductive Symbolic Execution Engine
Structural Entailment Engine
Spatial Formulae
Pure Formulae
Z3
JSIL Specifications

Figure 5.1: Architecture

- A set of predicate assertions δ.

Bi-Abductive Symbolic Execution. The symbolic execution engine implements Algorithm 1, which is explained in Section 4.4. It passes the entailment problem in the procedure call rules and in the final step of the algorithm to the structural entailment engine.

Anti-Frame Inference. We infer missing resources in two stages of the symbolic execution: when calling a procedure and when terminating the symbolic execution.

When the symbolic execution reaches a procedure call command, initially the tool finds all specifications for the procedure being called. The symbolic execution engine then passes the bi-abductive problem $current\ state\ *\ missing\ state \vdash precondition\ *\ frame$ to the structural entailment engine for each specification. If the current state is able to entail the precondition without the need for missing symbolic state then the symbolic execution transforms the current state into the callee's postcondition plus any framed state. Otherwise, the tool branches and for each specification the missing footprint is added to anti-frame symbolic state and the current state is transformed into the callee's postcondition plus any framed state.

When the symbolic execution for a procedure terminates, as outlined in Algorithm 1, we check that the final state entails any partial postcondition. In particular, the symbolic execution engine passes the bi-abductive problem $final\ state\ *\ missing\ state \vdash postcondition\ *\ frame$ to the structural entailment engine.

Any symbolic state added to the missing footprint must relate to the procedure's arguments or return value. Otherwise, the generated specification will not be well-formed. Additionally, only type information, heap cells and pure formulae are added to the missing footprint $?M$. Entailments involving pure formulae are checked using the constraint solver Z3, whereas those involving spatial formulae are checked in the structural entailment engine.

5.2 Implementation Challenges

We give examples in order to illustrate some interesting aspects of the JSIL Abduce tool.

Typing and Constraints Complexities. When evaluating expressions the tool infers missing type information and constraints. First, the tool attempts to type an expression with the given type information. If this fails the tool then attempts to infer the types needed to evaluate an expression. This type information is added to the type environment in both the current symbolic state and the missing symbolic state. In addition to type information, the tool infers constraints required to evaluate an expression. For example, when evaluating the expression, $\mathsf{nth}\,(\mathsf{list}, 3)$ the type `List` of list and the constraint that the length of list must be greater than 3 is generated.

We give an example to show the additional type information and pure formulae when evaluating expressions. The getTailElement procedure is a simple combination of basic commands in order to retrieve the last element of a list. It does not require any objects; instead, it only requires y to be list. The precondition has the constraint that the list must have at least one element and y is of type `List`. These are added when the expressions l−len(y) and l−nth(y,len) are evaluated.

```
proc getTailElement(y) {
            len := l-len(y) - 1;
            ret := l-nth(y,len);
    rlab:   skip
} with {ret: ret, rlab;};
```

```
Generated precondition:
```
$\{\ (y = \#y)\ \ *\ \ (!((l\text{-}len(\#y) - 1) < 0))\ \ *\ \ \mathsf{types}(\#y : \$\$list\_type)\ \}$

```
Generated postcondition:
```
$$\left\{\begin{array}{l}(y = \#y)\ \ *\ \ (len = (l\text{-}len(\#y) - 1))\ \ *\ \ (ret = l\text{-}nth(\#y, (l\text{-}len(\#y) - 1))\ \ *\ \ (!((l\text{-}len(\#y) - 1) < 0))\\ \ *\ \mathsf{types}(\#y : \$\$list\_type, len : \$\$number\_type)\end{array}\right\}$$

```
Return mode:
Normal
```

Branching. When the tool cannot decide which branch of a goto command to explore, it explores both commands. The conditional expression can only be added to the missing symbolic state if the variables are contained within the previously inferred missing state and the parameters. Otherwise the symbolic execution would generate an invalid specification.

In order to demonstrate branching within the tool, we give the following example. The procedure simply sets the interval of the timeout object if the interval is not undefined. If the interval is undefined, the timeout object interval is set to the default value 0. As no partial specification is given the tool initiates the symbolic execution of the procedure with the empty symbolic state. As no information is known about the variable interval and it is contained within the procedures parameters, the symbolic execution explores both possible paths. A specification is generated for each path.

```
proc getTimeoutInterval (timeout, interval) {
            goto [(interval = undefined)] then1 else1;
    then:   [timeout, "interval"] := 0.;
            goto rlab;
    else:   [timeout, "interval"] := interval;
            ret := timeout;
    rlab:   skip
} with {ret: ret, rlab;};
```

The following specification is generated for the undefined case.

```
Generated precondition:
```
$\{\ (timeout, \text{``}interval\text{''}) \mapsto -\ \ *\ \ (interval = undefined)\ \ *\ \ \mathsf{types}(timeout : \$\$object\_type)\ \}$

```
Generated postcondition:
```
$$\left\{ \begin{array}{l} (timeout, \text{“}interval\text{”}) \mapsto 0 \ * \ (interval = undefined) \ * \ (ret = timeout) \\ * \ \mathsf{types}(timeout : \$\$object\_type) \end{array} \right\}$$

```
Return mode:
Normal
```

The following specification is generated when the given interval is not undefined.

```
Generated precondition:
```
$$\{ \ (timeout, \text{“}interval\text{”}) \mapsto - \ * \ (not \ (interval = undefined)) \ * \ \mathsf{types}(timeout : \$\$object\_type) \ \}$$

```
Generated postcondition:
```
$$\left\{ \begin{array}{l} (timeout, \text{“}interval\text{”}) \mapsto interval \ * \ (not \ (interval = undefined)) \ * \ (ret = timeout) \\ * \ \mathsf{types}(timeout : \$\$object\_type) \end{array} \right\}$$

```
Return mode:
Normal
```

Procedure Calls. The following example demonstrates the procedure call rule. First, we infer the specifications of the procedure getTimeoutInterval as it calls no other functions. This procedure requires an object timeout with a field interval. We infer the pre- and postcondition given below. The specification is added to the new specification environment as shown in Algorithm 1 and then the new specification environment is used for symbolic execution of getTimeoutIntervalInSeconds.

```
proc getTimeoutInterval (timeout) {
            ret := [timeout, "interval"];
    rlab:   skip
} with {ret: ret, rlab;};
```

```
Generated precondition:
```
$$\{ \ (timeout, \text{“}interval\text{”}) \mapsto \#t \ * \ (not(\#t = None)) \ * \ \mathsf{types}(timeout : \$\$object\_type) \ \}$$

```
Generated postcondition:
```
$$\{ \ (timeout, \text{“}interval\text{”}) \mapsto \#t \ * \ (ret = \#t) \ * \ (not(\#t = None)) \ * \ \mathsf{types}(timeout : \$\$object\_type) \ \}$$

```
Return mode:
Normal
```

Next, we symbolically execute getTimeoutIntervalInSeconds which calls the procedure getTimeoutInterval . The symbolic execution of getTimeoutIntervalInSeconds starts with a state and anti-frame emp. First, the field assignment basic command requires the heap cell assertion $(\text{func}, \text{“timeout”}) \mapsto -$. The assertion is added to the anti-frame. The procedure call rule then solves the problem $\text{emp} * P_M \vdash P_F * P$ where P is the precondition of getTimeoutInterval. This is solved by the entailment engine. As none of the precondition P is contained in the current state, P_M is the precondition of getTimeoutInterval. This results in $P_F = (\text{func}, \text{“timeout”}) \mapsto timeout$.

The resulting state is then the adapted postcondition of getTimeoutInterval combined with the frame P_F. The resulting anti-frame is $P_M * (\text{func}, \text{”timeout”}) \mapsto -$. After symbolically executing the final basic command we obtain the specification below.

```
proc getTimeoutIntervalInSeconds (func) {
            t := [func, "timeout"];
            x := "getTimeoutInterval" (t);
            ret := x / 1000;
    rlab:   skip
} with {ret: ret, rlab;};
```

```
Generated precondition:
```
$$\left\{ \begin{array}{l} (func, \text{“}timeout\text{”}) \mapsto \#t \ * \ (\#t, \text{“”}interval\text{””}) \mapsto \#i \\ * \ types(func : object_type, \#t : object_type, \#i : number_type) \end{array} \right\}$$

```
Generated postcondition:
```

$$\left\{ \begin{array}{l} (func, \text{“}timeout\text{”}) \mapsto \#t \;\; * \;\; (\#t, \text{“”}interval\text{””}) \mapsto \#i \;\; * \;\; (x = \#i) \;\; * \;\; (ret = (\#i/1000)) \\ * \;\; (t = \#t) \;\; * \;\; \texttt{types}(func : \$\$object\_type, ret : \$\$number\_type, \#i : \$\$number\_type, t : \$\$object\_type) \end{array} \right\}$$

```
Return mode:
Normal
```

Chapter 6

Evaluation

The evaluation of the tool was twofold; that of the tools ability to generate specifications and that of the quality of the specifications generated. Initially these two qualities were evaluated by a test suite of JSIL programs. The test suite contains a number of representative examples in order to test the correctness of the symbolic execution rules. In total, the test suite has 47 procedures, producing 60 specifications.

This evaluation was extended by inferring the specifications of the internal and built-in functions. We were able to find a bug in the implementation of two of the JavaScript internal functions. We evaluate the tools ability to generate specifications for these functions (Section 6.2). Additionally, we evaluate the quality of the specifications generated, by comparing hand-written specifications to automatically generated specifications. First, we discuss the infrastructure required to carry out this evaluation (Section 6.1).

6.1 Evaluation Infrastructure

The infrastructure required to verify the specifications generated by JSIL Abduce is given in Figure 6.1. The JSIL programs to be evaluated are given to JSIL Abduce which produces a new specification environment contains the JSIL specifications. The JSIL programs and the new specification environment is then passed into JSIL Verify, to verify the generated specifications. JSIL Abduce, generates a call graph before symbolic execution, to calculate the order procedures are analysed. Procedures must be analysed before any procedure which calls it is analysed.

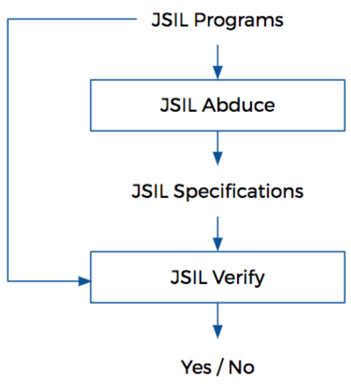

Figure 6.1: Evaluation Infrastructure

Limitations. Currently the basic command getFields is not supported by both JSIL Verify and JSIL Abduce, therefore for the evaluation we need to filter out all internal functions containing getFields . We perform the filtering while the call graph is generated. Additionally, JSIL Abduce does not support recursive, mutually recursive, or looping branches in procedures. Hence, we detect and halt symbolic execution of these branches, but we continue execution for the remaining branches in the procedure.

6.2 Generating Specifications for the Internal Functions

JSIL implementations for JavaScript internal and built-in functions were written for JSIL Verify. We evaluate the tool's ability to generate specifications for these functions, which involve approximately 3500 lines of JSIL code. JSIL specifications for the internal functions were also written, and can be used to verify JavaScript programs compiled to JSIL. We use these specifications to evaluate the quality of the automatically generated specifications. We give the generated call graph for a subset of these procedures in Figure 6.2.

During the analysis of the internal functions, we found a bug in the implementation of two internal functions, i__toDataDescriptor and i__toAccessorDescriptor.

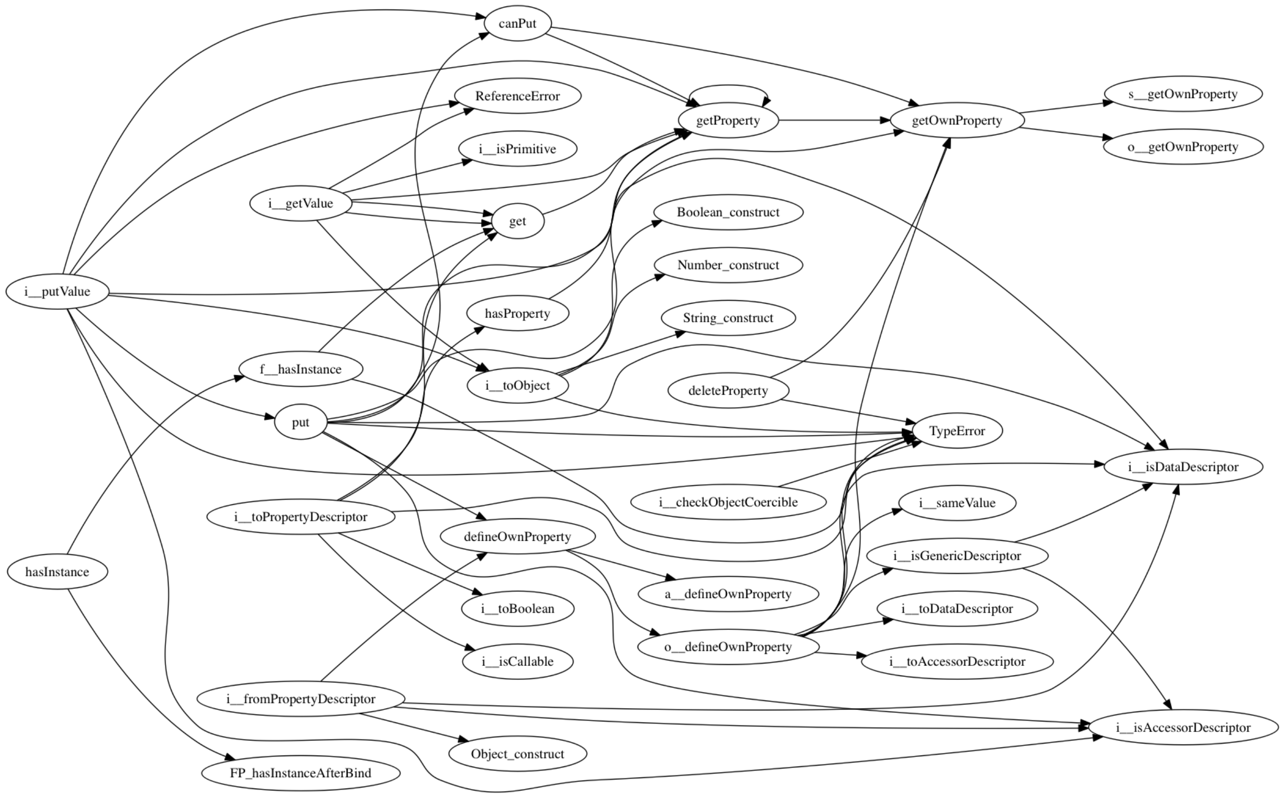

Figure 6.2: Call Graph for a subset of the internal functions.

Results. Table 6.2 gives the results of running JSIL Abduce on the internal functions and the separate built-in function libraries. This includes the functions relating to the setup of the JavaScript initial heap contained in the Init library. The number of halted execution branches, relates to the number of branches halted due to recursion. Most of the internal and built-in JSIL implementations contain sizeable procedures with many branches. As a result a significant number of specifications are created per procedure.

Internal and Built-In functions	Lines of code per library	Total number of procedures	Number of procedures containing unsupported operations	Number of halted execution branches	Number of specifica-tions generated	Time taken (in seconds)
Internal	735	30	0	16	287	40.73
Array	467	16	14	8	7	2.92
Boolean	30	4	0	0	28	1.29
Date	94	10	1	231	0	179.17
Errors	87	17	0	107	1	4.37
Functions	47	6	2	2	8	0.45
Global	14	2	0	0	12	0.44
Init	486	6	0	2	11	9.85
Math	73	18	2	0	145	7.27
Number	23	4	2	0	8	0.43
Object	291	19	1	20	31	2.86
String	194	6	3	0	68	5.06
Total	**2541**	**138**	**25**	**49**	**943**	**254.84**

Table 6.1: Results of inferring specifications of internal functions.

6.2.1 Specification Comparison

The semi-automatic JSIL Verify tool requires the JSIL logic specifications of the JavaScript internal functions. These hand-written specifications are verified with respect to the JSIL implementations. We compare these specifications with the automatically generated specifications.

Create Default Object Procedure. The create default object procedure, shown below, creates an object with the default values as defined in the ECMAScript documentation. It has four manually written specifications and four automatically generated specifications.

We evaluate the specifications for the case where both cl and ext are undefined. Generally the specifications contain similar information about the procedure. The automatically generated specifications contain additional type and spatial information compared to the manually written specifications. This is due to the type inference when evaluating expressions.

```
proc create_default_object (l, pr, cl, ext) {

                goto [cl = undefined] scl text;
        scl:    cl := "Object";
        text:   goto [ext = undefined] sext setall;
        sext:   ext := $$t;

        setall: [l, "@proto"] := pr;
                [l, "@class"] := cl;
                [l, "@extensible"] := ext;

        rlab:   xret := l
}
with
{
        ret:    xret, rlab;
};
```

Hand written specification:

```
Precondition:
```

$$\left\{ \begin{array}{l} (l = \#l) * (pr = \#pr) * (cl = \#cl) * (ext = \#ext) * \mathsf{types}(\#l : \$\$object\_type) \\ *(\#cl = undefined) * (\#ext = undefined) * \mathsf{emptyFields}(\#l :) \end{array} \right\}$$

Postcondition:

$$\left\{ \begin{array}{l} (\#l, \text{“}@proto\text{”}) \mapsto \#pr * (\#l, \text{“}@class\text{”}) \mapsto Object * (\#l, \text{“}@extensible\text{”}) \mapsto \$\$t \\ *\mathsf{emptyFields}(\#l : \text{“}@proto\text{”}, \text{“}@class\text{”}, \text{“}@extensible\text{”}) * (ret = \#l) \end{array} \right\}$$

Return mode:
Normal

Specification automatically generated by JSIL Abduce:

Precondition:

$$\left\{ \begin{array}{l} (l, \text{“}@proto\text{”}) \mapsto \#s\_11 * (l, \text{“}@class\text{”}) \mapsto \#s\_12 * (l, \text{“}@extensible\text{”}) \mapsto \#s\_13 \\ *(cl = undefined) * (ext = undefined) * (pr = \#s\_14) * (ext = undefined) \\ *(cl = undefined) * \mathsf{types}(l : \$\$object\_type, cl : \$\$undefined\_type, ext : \$\$undefined\_type) \end{array} \right\}$$

Postcondition:

$$\left\{ \begin{array}{l} (l, \text{“}@proto\text{”}) \mapsto \#s\_14 * (l, \text{“}@class\text{”}) \mapsto \text{“}Object\text{”} * (l, \text{“}@extensible\text{”}) \mapsto t \\ *(cl = \text{“}Object\text{”}) * (xret = l) * (ext = t) * (pr = \#s\_14) \\ *\mathsf{types}(l : \$\$object\_type, cl : \$\$string\_type, xret : \$\$object\_type, ext : \$\$boolean\_type) \end{array} \right\}$$

Return mode:
Normal

Error Construct Procedure. The error constructor, shown below, constructs an object with the default fields for an error object. Two specifications were manually written for the procedure whereas seven specifications were automatically generated. We generate more specifications because are tool does have abstraction. Therefore every time a program branches we generate a new specification. In particular, the hand written specifications generalise the result of calling the procedure i__toString which converts v to string type. However, the JSIL Abduce tool produces a specification for each possible case of the procedure i__toString. The procedure i__toString has seven specifications, resulting in significantly more cases.

```
proc Error_construct (xsc, vthis, v) {
                xret := vthis;

                [vthis, "@class"] := "Error";
                [vthis, "@extensible"] := t;

                goto [v = undefined] rlab mess;

        mess:   xerr := "i__toString" (v) with elab;
                [xret, "message"] := {{ "d", xerr, t, f, t }};

        rlab:   skip;
        elab:   skip
}
with
{
        ret:    xret, rlab;
        err:    xerr, elab;
};
```

Is Data Descriptor Procedure. The isDataDescriptor procedure, shown below, returns whether a given descriptor is a data descriptor. The five hand written specifications, use the DataDescriptor (d) and GenericDescriptor (d) predicates. Predicates can summarise portions of specifications and can lead to more concise specifications. As are tool does not support predicates or abstraction, the specifications generated are less succinct.

```
pred DataDescriptor (d) :
    types (d : $$list_type, #dwrit : $$boolean_type, #denum : $$boolean_type, #
        dconf : $$boolean_type) *
    (! (#dval == $$empty)) * (d == {{ "d", #dval, #dwrit, #denum, #dconf }});

pred GenericDescriptor (d) :
    types (d : $$list_type) * (d == {{ "g", #genum, #gconf, #gval, #gwrit, #gget
        , #gset }});
```

With these predicates the hand specification, where the descriptor is a data descriptor, is:

```
Precondition:
```
$\{ \ \texttt{DataDescriptor}(desc) \ \}$

```
Postcondition:
```
$\{ \ ret == t \ \}$

```
Return mode:
Normal
```

Even though the automatically generated specification is correct for this case, it is less concise than the hand-written specifications as the details of the case are not summarised with predicates.

```
Precondition:
```
$$\left\{ \begin{array}{l} (desc = \#d) \ * \ (!(\#d = undefined)) \ * \ (0. < (l - len \#d)) \ * \ (l - nth(\#d, 0.) = "d") \\ \ * \ \texttt{types}(\#d : list\_type) \end{array} \right\}$$

```
Postcondition:
```
$$\left\{ \begin{array}{l} (d = l - nth(\#d, 0.)) \ * \ (desc = \#d) \ * \ (xret = t) \ * \ (!(\#d = undefined)) \ * \ (0. < (l - len \#d)) \\ \ * \ (l - nth(\#d, 0.) = "d") \ * \ \texttt{types}(\#d : list\_type, \ desc : list\_type, \ xret : boolean\_type) \end{array} \right\}$$

```
Return mode:
Normal
```

Setup Initial Heap Procedure. The initial heap setup procedure, is required for any JSIL program compiled from JavaScript. The JavaScript heap requires a significant amount of set up. Therefore the setupInitialHeap function contained within the Init library consists of 435 lines of code. The automatically generated specifications for this procedure contain approximately 2674 separated assertions in both the precondition and postcondition.

As the first command in a JSIL program compiled from JavaScript is setupInitialHeap, unless the initial heap is given in a partial precondition, the entire precondition of setupInitialHeap is added to the anti-frame of the main procedure.

Chapter 7

Conclusion

We have progressed from a semi-automatic symbolic analysis to a fully automatic analysis for a subset of JSIL. First, we have successfully modified the symbolic analysis for a more explicit execution. Secondly, we have introduced a bi-abductive symbolic analysis. Finally, we have implemented a bi-abductive tool to infer procedure specifications. We summarise the objectives which have been achieved towards fully automatic verification of JavaScript programs (Section 7.1), and we discuss possible future work (Section 7.2).

7.1 Objectives Achieved

We have succeed in modifying the JSIL logic symbolic rules, in order to describe a symbolic analysis in a style that is closer to the implementation. We have proven this analysis sound with respect to the JSIL semantics.

We have then introduced a bi-abductive symbolic analysis for generating JSIL specifications. This analysis is able to infer the missing resources required to run a given procedure. We have established the soundness of the bi-abductive symbolic analysis in Theorem 2 by appealing to the standard symbolic execution analysis. Additionally, we have presented an algorithm to generate specifications for JSIL programs. This algorithm allows for partial specifications to be given as input to the verification.

Finally, we have developed a tool, JSIL Abduce, which automatically generates specifications for JSIL programs. The tool has generated over 1000 specifications, which have been verified by the semi-automatic tool JSIL Verify. We have evaluated the quality of the generated specifications in Section 6 by comparing them to their handwritten counterparts.

7.2 Future Work

There are two main motivations for generating specifications: to automatically verify properties of JavaScript programs and to support the developer in the verification of functional properties of JavaScript programs. We discuss future work in both of these endeavours. Additionally, future work is required to expand the scope of JSIL programs which the tool can analyse.

Inferring predicate assertions. Currently the tool does not support the inference of logical annotations needed for folding or unfolding user-defined predicates. This is important for partial specifications containing predicate assertions, as well as using the hand-written specifications for the internal functions. For now, due to unsupported operations, in order to infer specifications for JSIL programs compiled from JavaScript, we must use the hand-written internal function specifications. However, issues arise as the hand-written specifications contain a significant number of predicates. Expanding the tool to infer missing annotations needed to fold/unfold predicate assertions, would allow us to generate specifications for a greater range of JSIL programs compiled from JavaScript.

Automatically inferring loop invariants [12]. At present, the tool does not handle looping JSIL procedures. Additionally, the tool does not explore recursive or mutually branches. Future work in automatically generating loop invariants and handling recursion is possible for a pre-established set of abstractions.

Verify properties of JavaScript programs. Future work could explore properties which can be verified by the generated specifications. For example, the specifications are able to record the footprint of each procedure. This can be helpful in detecting confinement violations. For example, detecting whether a program modifies any JavaScript built-in functionality.

Abstraction. As discussed in the evaluation, the quantity of specifications generated for procedures, with multiple branches and procedure calls, is larger than that of the hand-written specifications. This is because the hand-written specifications are able to abstract the result of a procedure call and branching. Expanding the tool to include abstraction would allow us to have the same specification for describing the behaviour of multiple program paths.

Bibliography

[1] *Proceedings of the 3rd International Constraint Solver Competition (2008)*. pas d'éditeur commercial, 2008.

[2] T. Ball. The concept of dynamic analysis. *SIGSOFT Softw. Eng. Notes*, 24(6):216–234, Oct. 1999.

[3] C. Barrett, P. Fontaine, and C. Tinelli. The Satisfiability Modulo Theories Library (SMT-LIB). `www.SMT-LIB.org`, 2016.

[4] J. Berdine, C. Calcagno, and P. W. Hearn. Smallfoot: Modular automatic assertion checking with separation logic. In *Proceedings of the 4th International Conference on Formal Methods for Components and Objects*, FMCO'05, pages 115–137, Berlin, Heidelberg, 2006. Springer-Verlag.

[5] J. Berdine, B. Cook, and S. Ishtiaq. Slayer: Memory safety for systems-level code. In *Proceedings of the 23rd International Conference on Computer Aided Verification*, CAV'11, pages 178–183, Berlin, Heidelberg, 2011. Springer-Verlag.

[6] A. Biere, A. Cimatti, E. M. Clarke, and Y. Zhu. Symbolic model checking without bdds. In *Proceedings of the 5th International Conference on Tools and Algorithms for Construction and Analysis of Systems*, TACAS '99, pages 193–207, London, UK, UK, 1999. Springer-Verlag.

[7] M. Bodin, A. Chargueraud, D. Filaretti, P. Gardner, S. Maffeis, D. Naudziuniene, A. Schmitt, and G. 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, New York, NY, USA, 2014. ACM.

[8] R. Brummayer and A. Biere. Boolector: An efficient smt solver for bit-vectors and arrays. In *Proceedings of the 15th International Conference on Tools and Algorithms for the Construction and Analysis of Systems: Held As Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009,*, TACAS '09, pages 174–177, Berlin, Heidelberg, 2009. Springer-Verlag.

[9] J. N. Buxton and B. Randell, editors. *Software Engineering Techniques: Report of a Conference Sponsored by the NATO Science Committee, Rome, Italy, 27-31 Oct. 1969, Brussels, Scientific Affairs Division, NATO*. 1970.

[10] C. Cadar, D. Dunbar, and D. Engler. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI'08, pages 209–224, Berkeley, CA, USA, 2008. USENIX Association.

[11] C. Calcagno, D. Distefano, J. Dubreil, D. Gabi, P. Hooimeijer, M. Luca, P. O'Hearn, I. Papakonstantinou, J. Purbrick, and D. Rodriguez. *Moving Fast with Software Verification*, pages 3–11. Springer International Publishing, Cham, 2015.

[12] C. Calcagno, D. Distefano, P. W. O'Hearn, and H. Yang. Compositional shape analysis by means of bi-abduction. *J. ACM*, 58(6):26:1–26:66, Dec. 2011.

[13] E. Clarke, A. Biere, R. Raimi, and Y. Zhu. Bounded model checking using satisfiability solving. *Form. Methods Syst. Des.*, 19(1):7–34, July 2001.

[14] E. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, 1999.

[15] L. De Moura and N. 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.

[16] ECMA. 5th Edition of ECMA 262. ECMAScript Language Specification. Technical report, 2011.

[17] N. Eén and N. Sörensson. *An Extensible SAT-solver*, pages 502–518. Springer Berlin Heidelberg, Berlin, Heidelberg, 2004.

[18] R. E. Fairley. Tutorial: Static analysis and dynamic testing of computer software. *Computer*, 11(4):14–23, April 1978.

[19] P. A. Gardner, S. Maffeis, and G. D. Smith. Towards a program logic for javascript. In *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '12, pages 31–44, New York, NY, USA, 2012. ACM.

[20] T. J. Hickey. Iasolver (2010). `http://www.cs.brandeis.edu/~tim/Applets/IAsolver.html`, 2017 (accessed June 15, 2017).

[21] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, Oct. 1969.

[22] M. Huth and M. Ryan. *Logic in Computer Science: modelling and reasoning about systems (second edition)*. Cambridge University Press, 2004. Mass-printing license for Indian sub-continent operative, Portuguese language edition and online e-publishing licenses in preparation, 440pp. http://www.cs.bham.ac.uk/research/lics/.

[23] B. Jacobs, J. Smans, P. Philippaerts, F. Vogels, W. Penninckx, and F. Piessens. Verifast: A powerful, sound, predictable, fast verifier for c and java. In *Proceedings of the Third International Conference on NASA Formal Methods*, NFM'11, pages 41–55, Berlin, Heidelberg, 2011. Springer-Verlag.

[24] G. R. Joshua Bloch. Extra, extra - read all about it: Nearly all binary searches and mergesorts are broken. `https://research.googleblog.com/2006/06/extra-extra-read-all-about-it-nearly.html`, 2017 (accessed June 15, 2017).

[25] J. C. King. Symbolic execution and program testing. *Commun. ACM*, 19(7):385–394, July 1976.

[26] D. Kroening and M. Tautschnig. *CBMC – C Bounded Model Checker*, pages 389–391. Springer Berlin Heidelberg, Berlin, Heidelberg, 2014.

[27] K. R. M. Leino. Dafny: An automatic program verifier for functional correctness. In *Proceedings of the 16th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning*, LPAR'10, pages 348–370, Berlin, Heidelberg, 2010. Springer-Verlag.

[28] N. G. Leveson. An investigation of the therac-25 accidents. *IEEE Computer*, 26:18–41, 1993.

[29] S. Maffeis, J. C. Mitchell, and A. Taly. An operational semantics for javascript. In *Proceedings of the 6th Asian Symposium on Programming Languages and Systems*, APLAS '08, pages 307–325, Berlin, Heidelberg, 2008. Springer-Verlag.

[30] M. J. Marijn Heule and T. Balyo. Sat competition 2017. `http://baldur.iti.kit.edu/sat-competition-2017/index.php`, 2017 (accessed June 15, 2017).

[31] D. L. Mate Soos and R. Govostes. Stp constraint solver. `http://stp.github.io/`, 2017 (accessed June 15, 2017).

[32] G. R. Matthias Heizmann and T. Weber. The 12th international satisfiability modulo theories competition. `http://smtcomp.sourceforge.net/2017/`, 2017 (accessed June 15, 2017).

[33] S. McLaughlin, C. Barrett, and Y. Ge. Cooperating theorem provers: A case study combining hol-light and cvc lite. *Electron. Notes Theor. Comput. Sci.*, 144(2):43–51, Jan. 2006.

[34] Microsoft. Microsoft security bulletin ms17-010 - critical. `https://technet.microsoft.com/en-us/library/security/ms17-010.aspx`, 2017 (accessed June 15, 2017).

[35] D. Naudžiūnienė, J. Fragoso Santos, P. Maksimović, T. Wood, and P. Gardner. An Infrastructure for Tractable Verification of JavaScript Programs. Preprint - in submission to ECOOP'17, 2017.

[36] N. Nethercote and J. Seward. Valgrind: A framework for heavyweight dynamic binary instrumentation. *SIGPLAN Not.*, 42(6):89–100, June 2007.

[37] C. Peirce, C. Hartshorne, and P. Weiss. *Collected Papers of Charles Sanders Peirce*. Number v. 5-6 in Collected Papers of Charles Sanders Peirce. Belknap Press of Harvard University Press, 1974.

[38] C. S. Păsăreanu and N. Rungta. Symbolic pathfinder: Symbolic execution of java bytecode. In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering*, ASE '10, pages 179–180, New York, NY, USA, 2010. ACM.

[39] C. S. Păsăreanu, P. C. Mehlitz, D. H. Bushnell, K. Gundy-Burlet, M. Lowry, S. Person, and M. Pape. Combining unit-level symbolic execution and system-level concrete execution for testing nasa software. In *Proceedings of the 2008 International Symposium on Software Testing and Analysis*, ISSTA '08, pages 15–26, New York, NY, USA, 2008. ACM.

[40] J. C. Reynolds. Separation logic: A logic for shared mutable data structures. In *Proceedings of the 17th Annual IEEE Symposium on Logic in Computer Science*, LICS '02, pages 55–74, Washington, DC, USA, 2002. IEEE Computer Society.

[41] U. S. C. E. R. Team. Indicators associated with wannacry ransomware. `https://www.us-cert.gov/ncas/alerts/TA17-132A`, 2017 (accessed June 15, 2017).

[42] W. Visser, K. Havelund, G. Brat, S. Park, and F. Lerda. Model checking programs. *Automated Software Engg.*, 10(2):203–232, Apr. 2003.

[43] H. Yang, O. Lee, J. Berdine, C. Calcagno, B. Cook, D. Distefano, and P. O'Hearn. Scalable shape analysis for systems code. In *Proceedings of the 20th International Conference on Computer Aided Verification*, CAV '08, pages 385–398, Berlin, Heidelberg, 2008. Springer-Verlag.

Appendix A

JSIL Semantics and Substitution Lemma

A.0.1 JSIL Substitution Lemma

Lemma 3 (Substitution Lemma, Assertions)**.** *Let* $\mathsf{vars}(\mathsf{P}) \subseteq \{\mathtt{x_i}\mid_{\mathsf{i}=1}^{\mathsf{n}}\}$*. Then:*

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

A.0.2 JSIL Semantics

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

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

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

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

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

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

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

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

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

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

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

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

$$\textsc{Get Fields}\quad \frac{\begin{array}{c}[\![\mathtt{e}]\!]_\rho = l \quad \mathtt{h} = (\mathtt{h}' \uplus (l, pn_1) \mapsto - \uplus \ldots \uplus (l, pn_n) \mapsto -) \quad (l, -) \notin \mathsf{dom}(\mathtt{h}') \\ \{\{pn_1, ..., pn_n\}\} = \mathtt{v} \quad \mathcal{O}rd(\{\{pn_1, ..., pn_n\}\}) \quad \rho' = \rho[\mathtt{x} \mapsto \mathtt{v}]\end{array}}{[\![\mathtt{x} := \mathsf{getFields}\,(\mathtt{e})]\!]_{\mathtt{h},\rho} \triangleq (\mathtt{h}, \rho', \mathtt{v})}$$

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

Basic Command

$$\frac{\mathtt{p_m}(i) = \mathtt{bc} \in \mathtt{BCmd} \quad \llbracket \mathtt{bc} \rrbracket_{\mathtt{h},\rho} = (\mathtt{h}', \rho', -) \\ \mathtt{p} \vdash \langle \mathtt{h}', \rho', i, i+1 \rangle \Downarrow_{\mathtt{m}} \langle \mathtt{h}'', \rho'', \mathsf{o} \rangle}{\mathtt{p} \vdash \langle \mathtt{h}, \rho, \_, i \rangle \Downarrow_{\mathtt{m}} \langle \mathtt{h}'', \rho'', \mathsf{o} \rangle}$$

Goto

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

Cond. Goto - True

$$\frac{\mathtt{p_m}(i) = \mathsf{goto}\ [\mathsf{e}]\ j,\ k \quad \llbracket \mathsf{e} \rrbracket_{\rho} = \mathsf{true} \\ \mathtt{p} \vdash \langle \mathtt{h}, \rho, i, j \rangle \Downarrow_{\mathtt{m}} \langle \mathtt{h}', \rho', \mathsf{o} \rangle}{\mathtt{p} \vdash \langle \mathtt{h}, \rho, \_, i \rangle \Downarrow_{\mathtt{m}} \langle \mathtt{h}', \rho', \mathsf{o} \rangle}$$

Cond. Goto - False

$$\frac{\mathtt{p_m}(i) = \mathsf{goto}\ [\mathsf{e}]\ j,\ k \quad \llbracket \mathsf{e} \rrbracket_{\rho} = \mathsf{false} \\ \mathtt{p} \vdash \langle \mathtt{h}, \rho, i, k \rangle \Downarrow_{\mathtt{m}} \langle \mathtt{h}', \rho', \mathsf{o} \rangle}{\mathtt{p} \vdash \langle \mathtt{h}, \rho, \_, i \rangle \Downarrow_{\mathtt{m}} \langle \mathtt{h}', \rho', \mathsf{o} \rangle}$$

Normal Return

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

Error Return

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

Procedure Call - Normal

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

Procedure Call - Error

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

Phi-Assignment

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