Cosette: Compositional Symbolic Execution for JavaScript

ANONYMOUS AUTHOR(S)

We present Cosette, a framework for trusted, compositional symbolic execution of JavaScript programs. Cosette is the first symbolic analysis tool for JavaScript that precisely models the semantics of the language. Its aim is to assist the general programmer in debugging their code; the programmer writes symbolic tests for which Cosette provides concrete counter-models. We prove that the symbolic execution underpinning Cosette is sound and that it does not generate false positives. Moreover, this symbolic execution is compositional, which allows us to use Cosette to find bugs in separation logic specifications of JavaScript programs that are not detectable by standard symbolic execution techniques. We evaluate Cosette on an array of examples, including tests from the official JavaScript Test262 test suite and real-world Node.js libraries. These examples involve JavaScript-specific features, such as prototype inheritance, function closures, the for-in statement, and dynamic dispatch. We highlight the range of Cosette by using it to find bugs in a number of non-trivial separation logic specifications of JavaScript programs.

PM: What's the story?

(1) **Slogan**: Symbolic execution for JavaScript that precisely follows the language standard.
Goal: Symbolic testing, bug-finding, concrete counter-models.
Novelty: The precision wrt semantics, formal correctness guarantees.
Benefits: Trustworthy, sound analysis.
Limitations: No loop invariants, bounded. No eval.

(2) **Slogan**: Compositional execution for dynamic languages in general, and JavaScript in particular.
Novelty: Compositionality.
Benefits: summaries, frame-related bugs.

(3) **Application**: Symbolic testing of JavaScript programs.
Novelty: None?
Evaluation: Tests for the symbolic execution rules; JS programs using prototype inheritance, arrays, function closures, for-in, dynamic dispatch, etc.; test262 tests; node.js libraries for data structures

(4) **Application**: Debugging of separation logic specifications.
Novelty: Counter-models for separation logic assertions.
Evaluation: JaVerT specifications of this and that.

1 INTRODUCTION

JavaScript is the most widespread dynamic language: it is the de facto language for client-side Web applications (used by 94.8% of websites [?]); it is used for server-side scripting via Node.js; and it is even run on small embedded devices with limited memory. It is the most active language in both GitHub [?] and StackOverflow [?]. The dynamic nature of JavaScript and its complex semantics make it a difficult target for symbolic analysis and logic-based verification. This paper presents Cosette, a symbolic execution tool for JavaScript (ECMAScript 5, ES5 [?]). We highlight two relevant use cases for Cosette. First, we show how Cosette can be used as **(i)** a tool for running symbolic tests for JavaScript programs; and **(ii)** a debugging tool for separation logic specifications of JavaScript programs.

Architecture. The core of Cosette consists of a symbolic interpreter for JSIL [?], a simple intermediate goto language. We obtain this symbolic interpreter *for free*, by implementing a concrete JSIL

2018. 2475-1421/2018/1-ART1 $15.00
https://doi.org/

interpreter in Rosette [??], a symbolic virtual machine that facilitates generation of solver-aided languages. We design the concrete interpreter so that all of Rosette's natively supported solver-aided features, such as advanced string and regular-expression reasoning, are lifted to the JSIL symbolic interpreter. In §2, we give a formalisation of the JSIL concrete and symbolic executions, linking them together with a *soundness result*. We also provide insights on how to correctly design the concrete JSIL interpreter in Rosette.

The second component that Cosette uses is JS-2-JSIL [?], a well-tested, standard-compliant compiler from JavaScript to JSIL. We extend JS-2-JSIL with support for the non-strict mode of ES5, as well as regular expressions and the entire `String` built-in library. JS-2-JSIL allows us to lift the JSIL symbolic execution to JavaScript by first compiling JavaScript code to JSIL code, and then symbolically executing the compiled code in the JSIL symbolic interpreter. This process, described in §3.1, involves extending JavaScript syntax and the JS-2-JSIL compiler to support symbolic values and constructs for reasoning about them. These constructs are intuitive and allow the general developer to easily write assertions about the behaviour of their program. Moreover, we adjust the JSIL symbolic interpreter so that the abstraction level of the generated JSIL code precisely matches the abstraction level of Rosette, maximising the use of Rosette's native reasoning capabilities.

Application: Symbolic Testing. A commonly used approach to obtaining trust in JavaScript code is running it against adhoc test batteries—verifying that given concrete inputs, the code produces the expected output. The main drawback of this approach is that tests, in general, cannot guarantee exhaustiveness. In §3.2, we show how to use Cosette for symbolic testing of JavaScript code: instead of tests with concrete inputs, the developer uses symbolic inputs and states the constraints that the output needs to satisfy as simple, intuitive first-order assertions over these inputs. Furthermore, if a test fails, Cosette provides the concrete inputs that cause it to fail, exposing bugs in the tested code. We highlight the capabilities of Cosette through examples that showcase challenging reasoning on strings, regular expressions, and the `eval` statement.

Application: Debugging Separation Logic Specifications. Due to the complexity of JavaScript semantics, functional correctness specifications of JS programs are highly intricate. There are only a few tools (for example, JaVerT [?] and KJS [??]) that support such expressivity. They target the specialist developer wanting rich, mechanically verified specifications of critical JavaScript code. However, when these tools cannot prove that a given function satisfies a specification, to discover the error, the developer needs to understand in detail a complicated proof trace (JaVerT), or even act with almost no feedback (KJS).

In §4, we show how Cosette can be used as an auxiliary mechanism for debugging separation logic specifications of JavaScript programs in JaVerT. Our approach consists of: translating the separation logic specifications into symbolic tests and running these tests using Cosette. Then, if a symbolic test generated from a given specification fails, we can be sure that the code to be verified does not satisfy its specification. More importantly, Cosette then generates a concrete witness that invalidates the specification. This information greatly simplifies the debugging of both specifications and code.

Why Cosette? Cosette is *useful*: it has tangible applications. It can report bugs in JavaScript programs, producing concrete witnesses triggering the bugs. It can also be used as a helper tool for developers of logic-based functional correctness specifications of JavaScript code. Cosette is *approachable*: it can easily be used by a general JavaScript developer. The annotation burden of Cosette is minimal and the assertion language is simple and intuitive. Sweet spot? Cosette is *trustworthy*: its components come with correctness guarantees. The correctness of the JS-2-JSIL compiler ensures full adherence to the real semantics of JavaScript. The Cosette symbolic execution engine is based on a sound symbolic analysis for JSIL, guaranteeing the absence of false positives.

$$
\begin{array}{l}
\text{Numbers: } n \in \mathcal{N} \quad \text{Booleans: } b \in \mathcal{B} \quad \text{Strings: } s \in \mathcal{S} \quad \text{Locs: } l \in \mathcal{L} \quad \text{Vars: } x \in \mathcal{X} \\
\text{Types: } \tau \in \mathsf{Types} \quad \text{Values: } v \in \mathcal{V}_{\mathsf{JSIL}} ::= n \mid b \mid s \mid \mathsf{undefined} \mid \mathsf{null} \mid l \mid \tau \mid \mathit{fid} \mid \mathsf{empty} \\
\text{Expressions: } e \in \mathcal{E}_{\mathsf{JSIL}} ::= v \mid x \mid \ominus\, e \mid e \oplus e \\
\text{Basic Commands: } bc \in \mathcal{B}cmd ::= \mathsf{skip} \mid x := e \mid x := \mathsf{new}\,() \mid x := [e, e] \mid [e, e] := e \mid \\
\qquad\qquad \mathsf{delete}\,(e, e) \mid x := \mathsf{hasField}\,(e, e) \mid x := \mathsf{getFields}\,(e) \mid \mathsf{assume}\,(e) \mid \mathsf{assert}\,(e) \\
\text{Commands: } \mathsf{c} \in \mathit{Cmd} ::= bc \mid \mathsf{goto}\ i \mid \mathsf{goto}\ [e]\ i,\ j \mid x := e(\overline{e})\ \mathsf{with}\ j \\
\text{Procedures : } \mathit{proc} \in \mathsf{Proc} ::= \mathsf{proc}\ \mathit{fid}(\overline{x})\{\overline{\mathsf{c}}\}
\end{array}
$$

Fig. 1. Syntax of the JSIL Language

Sentence about unification. Finally, Cosette is *extensible*: its coverage can easily be extended in a modular way. This gives us the mechanism for supporting built-in libraries not covered by JS-2-JSIL, or adding support for standard-external runtime libraries, such as the DOM.

2 SYMBOLIC EXECUTION FOR JSIL

We present our symbolic execution engine for JSIL [?].

JFS:

- The concrete and symbolic semantics of JSIL are defined by means of an abstract semantics that can be parameterised with the appropriate state for obtaining both the concrete semantics and the symbolic semantics. This approach has a great many number of benefits: **(1)** it avoids redundancy, **(2)** it makes the choices in the design of the symbolic semantics more explicit, and **(3)** it leads to more modular implementations, avoiding code duplication.
- We present the theoretical underpinnings of our symbolic analysis, including a soundness result and a guarantee of absence of false positives for bug-finding.

2.1 Formalisation

JSIL: Syntax. JSIL is a simple goto language featuring top-level procedures and commands operating on object heaps. It was purposefully designed to natively support the main dynamic features of JavaScript: extensible objects; dynamic property access; and dynamic procedure calls. The syntax of JSIL is shown in Figure 1.

JSIL *values*, $v \in \mathcal{V}_{\mathsf{JSIL}}$, include numbers, booleans, strings, the special values undefined and null, as well as types τ, procedure identifiers fid, and the special value empty. JSIL *expressions*, $e \in \mathcal{E}_{\mathsf{JSIL}}$, include JSIL values, JSIL program variables x, and various unary and binary operators, which, for instance, provide support for sets and lists.

JSIL *basic commands* enable the manipulation of extensible objects and do not affect control flow. They include skip, variable assignment, object creation, property access, property assignment, property deletion, membership check, property collection, and two special commands, assume and assert, essential for symbolic execution, but with trivial concrete semantics. JSIL *commands* include JSIL basic commands and several commands related to control flow: conditional gotos, unconditional gotos and dynamic procedure calls.[1] The two goto commands are straightforward: $\mathsf{goto}\ i$ jumps to the i-th command of the active procedure, and $\mathsf{goto}\ [e]\ i,\ j$ jumps to the i-th command if e evaluates to true, and to the j-th if it evaluates to false. The dynamic procedure call $x := e(\overline{e})\ \mathsf{with}\ j$ first evaluates e and $\overline{e}$ to obtain the procedure name and arguments, respectively, executes the appropriate procedure with these arguments, and, in the end, assigns its return value

[1] JSIL also has a ϕ-node assignment, which allows JS-2-JSIL to produce code directly in Static-Single-Assignment (SSA) [?]. To avoid clutter, we omit the ϕ-node assignment from this formalisation as it does not impact the reasoning in any way. Details can be found in [?].

Evaluation of expressions

$$[\![v]\!]_\rho \triangleq v \quad [\![x]\!]_\rho \triangleq \rho(x) \quad [\![\ominus\, e]\!]_\rho \triangleq \overline{\ominus}([\![e]\!]_\rho) \quad [\![e_1 \oplus e_2]\!]_\rho \triangleq \overline{\oplus}([\![e_1]\!]_\rho, [\![e_2]\!]_\rho)$$

Skip

$$\langle h, \rho, \textsf{skip}\,\rangle^{\top} \rightarrow \langle h, \rho \rangle^{\top}$$

Object Creation

$$\frac{h = h \uplus (l, @proto) \mapsto \textsf{null} \quad (l, -) \notin \textsf{dom}(h)}{\langle h, \rho, x := \textsf{new}\,()\rangle^{\top} \rightarrow \langle h, \rho[x \mapsto l]\rangle^{\top}}$$

Property Collection

$$\frac{[\![e]\!]_\rho = l \quad h = h' \uplus \big((l, s_i) \mapsto v_i\big)\,|_{i=0}^{n} \quad (l, -) \notin \textsf{dom}(h')}{\langle h, \rho, x := \textsf{getFields}\,(e)\rangle^{\top} \rightarrow \langle h, \rho[x \mapsto \{\, s_0, \ldots, s_n \,\}]\rangle^{\top}}$$

Assignment

$$\frac{[\![e]\!]_\rho = v \quad \rho' = \rho[x \mapsto v]}{\langle h, \rho, x := e\rangle^{\top} \rightarrow \langle h, \rho'\rangle^{\top}}$$

Property Access

$$\frac{[\![e_1]\!]_\rho = l \quad [\![e_2]\!]_\rho = s \quad h = - \uplus (l, s) \mapsto v}{\langle h, \rho, x := [e_1, e_2]\rangle^{\top} \rightarrow \langle h, \rho[x \mapsto v]\rangle^{\top}}$$

Property Deletion

$$\frac{[\![e_1]\!]_\rho = l \quad [\![e_2]\!]_\rho = s \quad h = h' \uplus (l, s) \mapsto -}{\langle h, \rho, \textsf{delete}\,(e_1, e_2)\rangle^{\top} \rightarrow \langle h', \rho\rangle^{\top}}$$

Property Assignment - Found

$$\frac{\begin{array}{c}[\![e_1]\!]_\rho = l \quad [\![e_2]\!]_\rho = s \quad [\![e_3]\!]_\rho = v \\ h = h' \uplus (l, s) \mapsto -\end{array}}{\langle h, \rho, [e_1, e_2] := e_3\rangle^{\top} \rightarrow \langle h' \uplus (l, s) \mapsto v, \rho\rangle^{\top}}$$

Property Assignment - Not Found

$$\frac{\begin{array}{c}[\![e_1]\!]_\rho = l \quad [\![e_2]\!]_\rho = s \quad [\![e_3]\!]_\rho = v \\ h = h' \quad (l, s) \notin \textsf{dom}(h)\end{array}}{\langle h, \rho, [e_1, e_2] := e_3\rangle^{\top} \rightarrow \langle h \uplus (l, s) \mapsto v, \rho\rangle^{\top}}$$

Member Check - True

$$\frac{[\![e_1]\!]_\rho = l \quad [\![e_2]\!]_\rho = s \quad (l, s) \in \textsf{dom}(h)}{\langle h, \rho, x := \textsf{hasField}\,(e_1, e_2)\rangle^{\top} \rightarrow \langle h, \rho[x \mapsto \textsf{true}]\rangle^{\top}}$$

Member Check - False

$$\frac{[\![e_1]\!]_\rho = l \quad [\![e_2]\!]_\rho = s \quad (l, s) \notin \textsf{dom}(h)}{\langle h, \rho, x := \textsf{hasField}\,(e_1, e_2)\rangle^{\top} \rightarrow \langle h, \rho[x \mapsto \textsf{false}]\rangle^{\top}}$$

Assume

$$\frac{[\![e]\!]_\rho = \textsf{true}}{\langle h, \rho, \textsf{assume}\,(e)\rangle^{\top} \rightarrow \langle h, \rho\rangle^{\top}}$$

Assert - True

$$\frac{[\![e]\!]_\rho = \textsf{true}}{\langle h, \rho, \textsf{assert}\,(e)\rangle^{\top} \rightarrow \langle h, \rho\rangle^{\top}}$$

Assert - False

$$\frac{[\![e]\!]_\rho = \textsf{false}}{\langle h, \rho, \textsf{assert}\,(e)\rangle^{\top} \rightarrow \langle h, \rho\rangle^{\bot}}$$

Fig. 2. Execution for Basic Commands: $\langle h, \rho, bc\rangle^{\mu} \rightarrow \langle h', \rho'\rangle^{\mu'}$

to x. If the procedure raises an error, control is transferred to the j-th command, and to the next otherwise.

A JSIL program $\mathsf{p} \in \mathsf{P}$ can be seen as a set of top-level procedures of the form proc $fid(\overline{x})\{\overline{c}\}$, where fid is the procedure name, $\overline{x}$ are its formal parameters, and its body $\overline{c}$ is a *command list* consisting of a sequence of JSIL commands. Every JSIL program contains a special procedure main, denoting the entry point of the program. JSIL procedures return via two dedicated indexes, i_{nm} and i_{er}, using two dedicated variables, `ret` and `err`. If a procedure reaches the i_{nm} index, it returns normally with the return value denoted by `ret`; when it reaches i_{er}, it returns an error, with the error value denoted by `err`.

JSIL: Semantics. The basic memory model of JSIL is as follows. A JSIL heap, $h \in \mathcal{H}$, is a partial function mapping pairs of object locations, and strings to heap values. Given a heap h, we denote: a heap cell by $(l, s) \mapsto v$, meaning that $h(l, s) = v$; the union of two disjoint heaps by $h_1 \uplus h_2$; heap lookup by $h(l, s)$; and the empty heap by $\emptyset$. A JSIL variable store, $\rho \in \mathcal{S}to$, is a mapping from JSIL program variables $x \in \mathcal{X}$ to JSIL values. Finally, JSIL has two execution modes, ranged over by μ: $\top$, which indicates that the execution can proceed; and $\bot$, which indicates that a failure has occurred and the execution must stop.

JSIL semantics is defined in small-step style. Transitions for basic commands, given in Figure 2, are of the form $\mu : \langle h, \rho, bc\rangle^{\mu'} \rightarrow \langle h', \rho'\rangle^{\mu'}$, meaning that the execution of the basic command bc in the heap h, store ρ, and execution mode μ results in the heap h', ρ', and execution mode μ'. We denote the semantic interpretation of unary operators $\ominus$ by $\overline{\ominus}$, the semantic interpretation of binary operators $\oplus$ by $\overline{\oplus}$.

To describe transitions for JSIL commands, we introduce call stacks, denoted C. Call stacks are lists of tuples of the form (f, ρ, x, i, j), where: **(1)** f is a procedure identifier; **(2)** ρ is the store of the procedure that called f; **(3)** x is the variable to which the return of f must be assigned in ρ; **(4)** i is the index of the command to which the control must jump after the execution of f in case of normal return; and **(5)** j the index to which it must jump in case of error return. Transitions for control flow commands have the form: $\mathsf{p} : \langle h, \rho, i \rangle_C^{\mu} \to \langle h', \rho', i' \rangle_{C'}^{\mu'}$, meaning that, in the context of the entire program p, the evaluation of the i-th command of the first procedure in the call stack C, in the heap h, store ρ, and execution mode μ, generates the heap h', store ρ', call stack C', and the next command to be evaluated is the i'-th command of the first procedure of the call stack C', in execution mode μ'. Due to space constraints and as the transitions for JSIL symbolic execution are similar, we give the full semantics for JSIL control flow commands in the Appendix.

JSIL: Symbolic Semantics. In order to symbolically execute JSIL programs, we extend the syntax of JSIL expressions with symbolic strings $\hat{s} \in \hat{\mathcal{S}}$ and symbolic numbers $\hat{n} \in \hat{\mathcal{N}}$. For convenience, we use $\hat{\mathcal{X}}$ to denote the union of $\hat{\mathcal{S}}$ and $\hat{\mathcal{N}}$ and $\hat{x}$ to range over $\hat{\mathcal{X}}$. We introduce: JSIL symbolic expressions, $\hat{e} \in \hat{\mathcal{E}}_{\mathsf{JSIL}}$, defined as follows: $\hat{e} \triangleq v \mid \hat{x} \mid \ominus\, \hat{e} \mid \hat{e} \oplus \hat{e}$; as well as JSIL extended symbolic expressions, $\hat{\mathrm{e}} \in \underline{\hat{\mathcal{E}}}_{\mathsf{JSIL}}$, defined as follows: $\hat{\mathrm{e}} \triangleq v \mid x \mid \hat{x} \mid \ominus\, \hat{\mathrm{e}} \mid \hat{\mathrm{e}} \oplus \hat{\mathrm{e}}$. Extended expressions differ from symbolic ones in that they can contain program variables.

We extend heaps, stores, and call stacks with symbolic values, obtaining symbolic heaps, stores, and call stacks, respectively ranged over by $\hat{h}$, $\hat{\rho}$, and $\hat{C}$. A symbolic heap, $\hat{h} \in \hat{\mathcal{H}}$, is a partial function mapping pairs of object locations and symbolic expressions to symbolic expressions. A symbolic store, $\hat{\rho} \in \hat{\mathcal{S}to}$, is a mapping from program variables $x \in \mathcal{X}$ to symbolic expressions. Therefore, an evaluation of a JSIL extended expression $\hat{\mathrm{e}}$ in a symbolic store $\hat{\rho}$ always yields a symbolic expression $\hat{e}$. A symbolic call stack $\hat{C}$ only differs from a concrete call stack in that it contains symbolic stores instead of concrete stores.

A *symbolic state* $\hat{\sigma} = (\hat{h}, \hat{\rho}, \hat{C}, \pi)$ is a 4-tuple consisting of a symbolic heap $\hat{h}$, a symbolic store $\hat{\rho}$, a symbolic call stack $\hat{C}$, and a path condition π. The *path condition* [?] is a first-order quantifier-free formula over symbolic strings and numbers, which accumulates constraints on the given symbolic inputs that trigger the execution to follow the path that led to the current symbolic state. Path conditions are given by the following grammar:

$$\pi \triangleq \hat{e}_1 = \hat{e}_2 \mid \hat{e}_1 \leq \hat{e}_2 \mid \pi_1 \ \wedge\ \pi_2 \mid \pi_1 \vee \pi_2 \mid \neg\pi \mid \mathsf{true} \mid \mathsf{false}$$

To avoid clutter, we conflate logical values with JSIL logical values and JSIL logical operators with the boolean logical operators. Alternatively, we could have chosen to have two different types, JSIL logical expressions and logical expressions, together with a lifting function for converting the former to the latter. Our choice simplifies both reasoning and presentation.

Figure 3 presents the symbolic execution rules for the JSIL basic commands. Rules have the form $\langle \hat{h}, \hat{\rho}, bc, \pi \rangle \rightsquigarrow \langle \hat{h}', \hat{\rho}', \pi' \rangle$, where: **(1)** $\hat{h}$ and $\hat{\rho}$ are the symbolic heap and store on which to evaluate bc, **(2)** π the current *path condition*, and **(3)** $\hat{h}'$, $\hat{\rho}'$, and π' the resulting symbolic heap, store, and path condition. Notice that the rules are non-deterministic.

Figure 4 presents the symbolic execution rules for JSIL commands. Rules have the form $\mathsf{p} : \langle \hat{h}, \hat{\rho}, i, \pi \rangle_{\hat{C}}^{\mu} \rightsquigarrow \langle \hat{h}', \hat{\rho}', i', \pi' \rangle_{\hat{C}'}^{\mu'}$; they are analogous to the semantic rules for JSIL commands, except that the heap, store, and call stack are symbolic and there is the additional path condition. For clarity, we keep the program and the context implicit wherever possible, and make use of a function $\mathsf{cmd}(\mathsf{p}, C, i)$, which returns the i-th command of the procedure that is first in C. We write $\mathsf{cmd}(i)$ when p and C are implicit.

The rules for skip, assignment, object creation, property collection, assume, and assert are straightforward. The remaining rules follow a specific pattern. To get a better intuition of how these rules work, let us take a look at the snippet of code shown on the right. This code: 0) creates a new object o; 1) assigns 42 to a symbolic property ŝ of o; 2) collects all the properties of o into a set and assigns this set to x; and 3) asserts that the cardinality of the set in x is 2, i.e. that o

```
0   o := new ()
1   o[ŝ] := 42;
2   x := getFields(o);
3   assert (card x == 2)
```

Evaluation of extended JSIL expressions

$$\begin{array}{c}\llbracket v \rrbracket_{\hat{\rho}} \triangleq v \\ \llbracket x \rrbracket_{\hat{\rho}} \triangleq \hat{\rho}(x) \\ \llbracket \hat{x} \rrbracket_{\hat{\rho}} \triangleq \hat{x}\end{array} \qquad \frac{\llbracket \hat{\mathrm{e}} \rrbracket_{\hat{\rho}} = v}{\llbracket \ominus \hat{\mathrm{e}} \rrbracket_{\hat{\rho}} \triangleq \overline{\ominus} v} \qquad \frac{\llbracket \hat{\mathrm{e}} \rrbracket_{\hat{\rho}} = \hat{e} \notin \mathcal{V}_{\mathrm{JSIL}}}{\llbracket \ominus \hat{\mathrm{e}} \rrbracket_{\hat{\rho}} \triangleq \ominus \hat{e}} \qquad \frac{v = \overline{\oplus}(\llbracket \hat{\mathrm{e}}_1 \rrbracket_{\hat{\rho}}, \llbracket \hat{\mathrm{e}}_2 \rrbracket_{\hat{\rho}})}{\llbracket \hat{\mathrm{e}}_1 \oplus \hat{\mathrm{e}}_2 \rrbracket_{\hat{\rho}} \triangleq v} \qquad \frac{\begin{array}{c}\llbracket \hat{\mathrm{e}}_1 \rrbracket_{\hat{\rho}} = \hat{e}_1 \quad \llbracket \hat{\mathrm{e}}_2 \rrbracket_{\hat{\rho}} = \hat{e}_2 \\ \hat{e}_1 \notin \mathcal{V}_{\mathrm{JSIL}} \ \lor \ \hat{e}_2 \notin \mathcal{V}_{\mathrm{JSIL}}\end{array}}{\llbracket \hat{\mathrm{e}}_1 \oplus \hat{\mathrm{e}}_2 \rrbracket_{\hat{\rho}} \triangleq \hat{e}_1 \oplus \hat{e}_2}$$

Skip

$$\langle \hat{h}, \hat{\rho}, \mathsf{skip}, \pi \rangle^{\top} \rightsquigarrow \langle \hat{h}, \hat{\rho}, \pi \rangle^{\top}$$

Object Creation

$$\frac{(l, -) \notin \mathsf{dom}(\hat{h}) \qquad \hat{h}' = \hat{h} \uplus (l, @\mathit{proto}) \mapsto \mathsf{null}}{\langle \hat{h}, \hat{\rho}, x := \mathsf{new}\,(), \pi \rangle^{\top} \rightsquigarrow \langle \hat{h}', \hat{\rho}[x \mapsto l], \pi \rangle^{\top}}$$

Assignment

$$\frac{\llbracket \hat{\mathrm{e}} \rrbracket_{\hat{\rho}} = \hat{e} \quad \hat{\rho}' = \hat{\rho}[x \mapsto \hat{e}]}{\langle \hat{h}, \hat{\rho}, x := \hat{\mathrm{e}}, \pi \rangle^{\top} \rightsquigarrow \langle \hat{h}, \hat{\rho}', \pi \rangle^{\top}}$$

Property Collection

$$\frac{\llbracket \hat{\mathrm{e}} \rrbracket_{\hat{\rho}} = l \quad \hat{h} = \hat{h}' \uplus \Big((l, \hat{p}_i) \mapsto -\Big) \,|_{i=0}^{n} \quad (l, -) \notin \mathsf{dom}(\hat{h}')}{\langle h, \rho, x := \mathsf{getFields}\,(\hat{\mathrm{e}}), \pi \rangle^{\top} \rightarrow \langle h, \rho[x \mapsto \{\, \hat{p}_0, \ldots, \hat{p}_n \,\}], \pi \rangle^{\top}}$$

Property Access

$$\frac{\begin{array}{c}\llbracket \hat{\mathrm{e}}_1 \rrbracket_{\hat{\rho}} = l \quad \llbracket \hat{\mathrm{e}}_2 \rrbracket_{\hat{\rho}} = \hat{e}_p \quad \hat{h} = \hat{h}' \uplus \Big((l, \hat{p}_i) \mapsto \hat{v}_i\Big) \,|_{i=0}^{n} \quad (l, -) \notin \mathsf{dom}(\hat{h}') \quad 0 \leq k \leq n \\ \pi' = \pi \ \land \ \Big((\hat{p}_k = \hat{e}_p) \ \land \Big(\land_{i=0,\, i \neq k}^{n} (\hat{p}_i \neq \hat{e}_p)\Big)\Big)\end{array}}{\langle \hat{h}, \hat{\rho}, x := [\hat{\mathrm{e}}_1, \hat{\mathrm{e}}_2], \pi \rangle^{\top} \rightsquigarrow \langle \hat{h}, \hat{\rho}[x \mapsto \hat{v}_k], \pi' \rangle^{\top}}$$

Property Assignment - Found

$$\frac{\begin{array}{c}\llbracket \hat{\mathrm{e}}_1 \rrbracket_{\hat{\rho}} = l \quad \llbracket \hat{\mathrm{e}}_2 \rrbracket_{\hat{\rho}} = \hat{e}_p \quad \llbracket \hat{\mathrm{e}}_3 \rrbracket_{\hat{\rho}} = \hat{e}_v \quad \hat{h} = \hat{h}' \uplus \Big((l, \hat{p}_i) \mapsto \hat{v}_i\Big) \,|_{i=0}^{n} \quad (l, -) \notin \mathsf{dom}(\hat{h}') \quad 0 \leq k \leq n \\ \pi' = \pi \ \land \ \Big((\hat{p}_k = \hat{e}_p) \ \land \Big(\land_{i=0,\, i \neq k}^{n} (\hat{p}_i \neq \hat{e}_p)\Big)\Big) \quad \hat{h}'' = \hat{h}' \uplus \Big((l, \hat{p}_i) \mapsto \hat{v}_i\Big) \,|_{i=0,\, i \neq k}^{n} \uplus (l, \hat{e}_p) \mapsto \hat{e}_v\end{array}}{\langle \hat{h}, \hat{\rho}, [\hat{\mathrm{e}}_1, \hat{\mathrm{e}}_2] := \hat{\mathrm{e}}_3, \pi \rangle^{\top} \rightsquigarrow \langle \hat{h}'', \hat{\rho}, \pi' \rangle^{\top}}$$

Property Assignment - Not Found

$$\frac{\begin{array}{c}\llbracket \hat{\mathrm{e}}_1 \rrbracket_{\hat{\rho}} = l \quad \llbracket \hat{\mathrm{e}}_2 \rrbracket_{\hat{\rho}} = \hat{e}_p \quad \llbracket \hat{\mathrm{e}}_3 \rrbracket_{\hat{\rho}} = \hat{e}_v \quad \hat{h} = \hat{h}' \uplus \Big((l, \hat{p}_i) \mapsto \hat{v}_i\Big) \,|_{i=0}^{n} \quad (l, -) \notin \mathsf{dom}(\hat{h}') \quad 0 \leq k \leq n \\ \pi' = \pi \ \land \ \Big(\land_{i=0}^{n} (\hat{p}_i \neq \hat{e}_p)\Big) \quad \hat{h}'' = \hat{h} \uplus (l, \hat{e}_p) \mapsto \hat{e}_v\end{array}}{\langle \hat{h}, \hat{\rho}, [\hat{\mathrm{e}}_1, \hat{\mathrm{e}}_2] := \hat{\mathrm{e}}_3, \pi \rangle^{\top} \rightsquigarrow \langle \hat{h}'', \hat{\rho}, \pi' \rangle^{\top}}$$

Property Deletion

$$\frac{\begin{array}{c}\llbracket \hat{\mathrm{e}}_1 \rrbracket_{\hat{\rho}} = l \quad \llbracket \hat{\mathrm{e}}_2 \rrbracket_{\hat{\rho}} = \hat{e}_p \quad \hat{h} = \hat{h}' \uplus \Big((l, \hat{p}_i) \mapsto -\Big) \,|_{i=0}^{n} \quad (l, -) \notin \mathsf{dom}(\hat{h}') \quad 0 \leq k \leq n \\ \pi' = \pi \ \land \ \Big((\hat{p}_k = \hat{e}_p) \ \land \Big(\land_{i=0,\, i \neq k}^{n} (\hat{p}_i \neq \hat{e}_p)\Big)\Big) \quad \hat{h}'' = \hat{h}' \uplus \Big((l, \hat{p}_i) \mapsto \hat{v}_i\Big) \,|_{i=0,\, i \neq k}^{n}\end{array}}{\langle \hat{h}, \hat{\rho}, \mathsf{delete}\,(\hat{\mathrm{e}}_1, \hat{\mathrm{e}}_2), \pi \rangle^{\top} \rightsquigarrow \langle \hat{h}'', \hat{\rho}, \pi' \rangle^{\top}}$$

Member Check - True

$$\frac{\begin{array}{c}\llbracket \hat{\mathrm{e}}_1 \rrbracket_{\hat{\rho}} = l \quad \llbracket \hat{\mathrm{e}}_2 \rrbracket_{\hat{\rho}} = \hat{e}_p \quad \hat{h} = \hat{h}' \uplus \Big((l, \hat{p}_i) \mapsto -\Big) \,|_{i=0}^{n} \quad (l, -) \notin \mathsf{dom}(\hat{h}') \quad 0 \leq k \leq n \\ \pi' = \pi \ \land \ \Big((\hat{p}_k = \hat{e}_p) \ \land \Big(\land_{i=0,\, i \neq k}^{n} (\hat{p}_i \neq \hat{e}_p)\Big)\Big)\end{array}}{\langle \hat{h}, \hat{\rho}, x := \mathsf{hasField}\,(\hat{\mathrm{e}}_1, \hat{\mathrm{e}}_2), \pi \rangle^{\top} \rightsquigarrow \langle \hat{h}, \hat{\rho}[x \mapsto \mathsf{true}], \pi' \rangle^{\top}}$$

Member Check - False

$$\frac{\llbracket \hat{\mathrm{e}}_1 \rrbracket_{\hat{\rho}} = l \quad \llbracket \hat{\mathrm{e}}_2 \rrbracket_{\hat{\rho}} = \hat{e}_p \quad \hat{h} = \hat{h}' \uplus \Big((l, \hat{p}_i) \mapsto -\Big) \,|_{i=0}^{n} \quad (l, -) \notin \mathsf{dom}(\hat{h}') \quad 0 \leq k \leq n \quad \pi' = \pi \ \land \ \Big(\land_{i=0}^{n} (\hat{p}_i \neq \hat{e}_p)\Big)}{\langle \hat{h}, \hat{\rho}, x := \mathsf{hasField}\,(\hat{\mathrm{e}}_1, \hat{\mathrm{e}}_2), \pi \rangle^{\top} \rightsquigarrow \langle \hat{h}, \hat{\rho}[x \mapsto \mathsf{false}], \pi' \rangle^{\top}}$$

Assert - True

$$\frac{\llbracket \hat{\mathrm{e}} \rrbracket_{\hat{\rho}} = \hat{e} \quad \pi \vdash \hat{e}}{\langle \hat{h}, \hat{\rho}, \mathsf{assert}\,(\hat{\mathrm{e}}), \pi \rangle^{\top} \rightsquigarrow \langle \hat{h}, \hat{\rho}, \pi \rangle^{\top}}$$

Assert - False

$$\frac{\llbracket \hat{\mathrm{e}} \rrbracket_{\hat{\rho}} = \hat{e} \quad (\pi \ \land \ \neg \hat{e}) \text{ satisfiable}}{\langle \hat{h}, \hat{\rho}, \mathsf{assert}\,(\hat{\mathrm{e}}), \pi \rangle^{\top} \rightsquigarrow \langle \hat{h}, \hat{\rho}, \pi \land \neg \hat{e} \rangle^{\bot}}$$

Assume

$$\frac{\llbracket \hat{\mathrm{e}} \rrbracket_{\hat{\rho}} = \hat{e} \quad \pi \vdash \hat{e}}{\langle \hat{h}, \hat{\rho}, \mathsf{assume}\,(\hat{\mathrm{e}}), \pi \rangle^{\top} \rightsquigarrow \langle \hat{h}, \hat{\rho}, \pi \land \hat{e} \rangle^{\top}}$$

Fig. 3. Symbolic Semantics for JSIL Basic Commands: $\langle \hat{h}, \hat{\rho}, bc, \pi \rangle^{\mu} \rightsquigarrow \langle \hat{h}', \hat{\rho}', \pi' \rangle^{\mu'}$

has two properties in the end. This last assertion will produce a failing symbolic execution. Let us understand why.

We start from an empty heap, empty store, and an empty path condition: $\langle \emptyset, \emptyset, 0, \mathsf{true} \rangle^{\top}$. After the execution of the first command, o := new (), using the Basic Command and Object Creation rules, we get to the state $\langle \{\mathtt{l}_{\mathtt{o}} : \{\text{“}\mathtt{@proto}\text{”} : \mathtt{null}\}\}, \{\mathtt{o} : \mathtt{l}_{\mathtt{o}}\}, 1, \mathsf{true} \rangle^{\top}$, illustrated at the top of Figure 5. The next command to be executed is the property assignment $\mathtt{o}[\hat{\mathtt{s}}] := 42$. In the symbolic semantics, there are two potential Property Assignment rules (Found, Not Found), and in our case, both of them are applicable. The key strategy is to branch on the targeted property of the object (in our case, the symbolic property $\hat{s}$ of object at location $\mathtt{l}_{\mathtt{o}}$) being equal to any one or none of the already

$$\textsc{Basic Command}\quad \frac{\mathsf{cmd}(i) = bc \quad \langle \hat{h}, \hat{\rho}, bc, \pi\rangle \rightsquigarrow \langle \hat{h}', \hat{\rho}', \pi'\rangle}{\langle \hat{h}, \hat{\rho}, i, \pi\rangle \rightsquigarrow \langle \hat{h}', \hat{\rho}', i+1, \pi'\rangle}$$

$$\textsc{Basic Command - Fail}\quad \frac{\mathsf{cmd}(i) = bc \quad \langle \hat{h}, \hat{\rho}, bc, \pi\rangle \rightsquigarrow \langle \bot, \pi\rangle}{\langle \hat{h}, \hat{\rho}, i, \pi\rangle \rightsquigarrow \langle \bot, \pi\rangle}$$

$$\textsc{Goto}\quad \frac{\mathsf{cmd}(i) = \mathrm{goto}\ j}{\langle \hat{h}, \hat{\rho}, i, \pi\rangle \rightsquigarrow \langle \hat{h}, \hat{\rho}, j, \pi\rangle}$$

$$\textsc{Cond. Goto - True}\quad \frac{\mathsf{cmd}(i) = \mathrm{goto}\ [e]\ j,\ k \quad [\![e]\!]_{\hat{\rho}} = \hat{e}}{\langle \hat{h}, \hat{\rho}, i, \pi\rangle \rightsquigarrow \langle \hat{h}, \hat{\rho}, j, \pi \wedge \hat{e}\rangle}$$

$$\textsc{Cond. Goto - False}\quad \frac{\mathsf{cmd}(i) = \mathrm{goto}\ [e]\ j,\ k \quad [\![e]\!]_{\hat{\rho}} = \hat{e}}{\langle \hat{h}, \hat{\rho}, i, \pi\rangle \rightsquigarrow \langle \hat{h}, \hat{\rho}, k, \pi \wedge \neg\hat{e}\rangle}$$

$$\textsc{Procedure Call}\quad \frac{\mathsf{cmd}(i) = x := e(e_i\ |_{i=0}^{n})\ \mathrm{with}\ j \quad [\![e]\!]_{\hat{\rho}} = f' \quad [\![e_i]\!]_{\hat{\rho}} = \hat{e}_i\ |_{i=0}^{n} \quad \mathsf{args}(f') = [x_1, \ldots, x_m] \quad \hat{e}_i = \mathsf{undefined}\ |_{i=n+1}^{m}}{\langle \hat{h}, \hat{\rho}, i, \pi\rangle_{\hat{C}} \rightsquigarrow \langle \hat{h}, [x_i \mapsto \hat{e}_i\ |_{i=0}^{m}], 0, \pi\rangle_{(f', \hat{\rho}, x, i+1, j)::\hat{C}}}$$

$$\textsc{Normal Return}\quad \frac{\hat{C} = (-, \hat{\rho}', x, i, -) :: \hat{C}' \quad \hat{\rho}(\mathsf{ret}) = \hat{e}}{\langle \hat{h}, \hat{\rho}, i_{\mathsf{nm}}, \pi\rangle_{\hat{C}} \rightsquigarrow \langle \hat{h}, \hat{\rho}'[x \mapsto \hat{e}], i, \pi\rangle_{\hat{C}'}}$$

$$\textsc{Error Return}\quad \frac{\hat{C} = (-, \hat{\rho}', x, -, j) :: \hat{C}' \quad \hat{\rho}(\mathsf{err}) = \hat{e}}{\langle \hat{h}, \hat{\rho}, i_{\mathsf{er}}, \pi\rangle_{\hat{C}} \rightsquigarrow \langle \hat{h}, \hat{\rho}'[x \mapsto \hat{e}], j, \pi\rangle_{\hat{C}'}}$$

Fig. 4. Symbolic Semantics for JSIL Commands: $\langle \hat{h}, \hat{\rho}, i, \pi\rangle_{\hat{C}} \rightsquigarrow \langle \hat{h}', \hat{\rho}', j, \pi'\rangle_{\hat{C}'}$

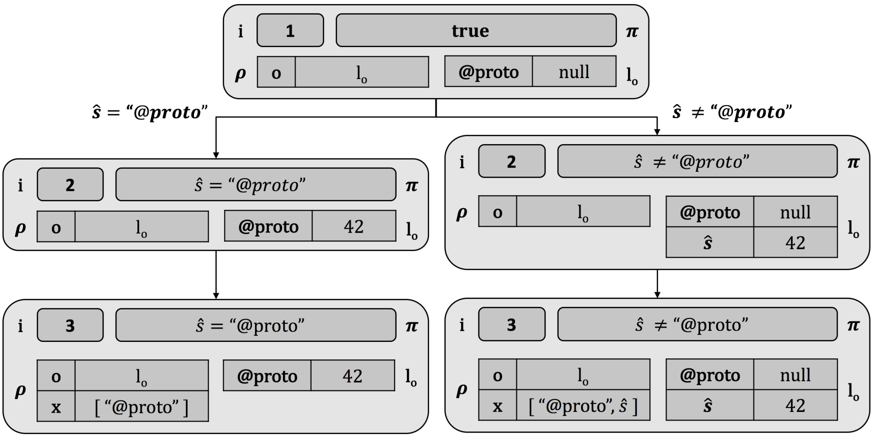

Fig. 5. Example of a Cosette symbolic execution

existing properties of the object (in our case, we have only "@proto"), adding the appropriate equalities and inequalities to the path condition, and proceeding with the symbolic execution for all obtained branches. In this case, this means that the symbolic execution will branch on whether or not $\hat{s}$ = "@proto". We obtain two symbolic states, shown in the second row of Figure 5. The left branch corresponds to the (Found) case, when $\hat{s}$ = "@proto": this equality is added to the path condition and the value of the property "@proto" is updated to 42. In the right branch, we have that $\hat{s} \neq$ "@proto" (Not Found), hence object o has two properties: "@proto", with value null; and $\hat{s}$, with value 42. The execution then continues in both branches with the property collection command x := getFields(o), which assigns the set of properties of the object o to the variable x (last row of Figure 5). Finally, we execute assert (card x = 2), asserting that o has exactly two properties, which we observe to hold in the right branch, but not in the left. Therefore, following the Assert - False rule, we obtain a failing symbolic execution trace, from which a concrete counter-model can be derived ($\hat{s}$ = "@proto").

We now present our theoretical results. We denote the reflexive-transitive closure of $\rightarrow$ by $\rightarrow^*$ and the reflexive-transitive closure of $\rightsquigarrow$ by $\rightsquigarrow^*$, and we define both in the usual way.

Transparency. Observe that a concrete state is also a symbolic state. Hence, we can feed a concrete state to the symbolic execution. In that case, the symbolic execution must behave exactly as the concrete execution. This is captured by the transparency theorem given below.

Theorem 2.1 (Transparency). $\forall\, \mathsf{p}, h, \rho, i, C, \mu, h', \rho', i', C', \mu'$.

$$\mathsf{p} : \langle h, \rho, i \rangle_C^{\mu} \rightarrow^* \langle h', \rho', i' \rangle_{C'}^{\mu'} \iff \mathsf{p} : \langle h, \rho, i, \mathsf{true} \rangle_C^{\mu} \rightsquigarrow^* \langle h', \rho', i', \mathsf{true} \rangle_{C'}^{\mu'}.$$

Soundness. To establish the soundness of symbolic execution, we need to relate symbolic states to concrete states. To this end, we make use of *symbolic environments* $\varepsilon : \hat{\mathcal{X}} \rightharpoonup \mathcal{V}_{\mathrm{JSIL}}$, mapping symbolic values to concrete values. A symbolic environment is *well-formed* if it maps symbolic values to concrete values of the appropriate type (i.e. symbolic strings are mapped to strings and symbolic numbers are mapped to numbers). In the following, we always assume well-formed symbolic environments. Given a symbolic environment ε, we define the interpretation of extended expressions and symbolic expressions as follows:

Interpretation of Symbolic Expressions

$$[\![v]\!]_\varepsilon \triangleq v \qquad [\![\hat{x}]\!]_\varepsilon \triangleq \varepsilon(\hat{x}) \qquad [\![\ominus\, \hat{e}]\!]_\varepsilon \triangleq \overline{\ominus}([\![\hat{e}]\!]_\varepsilon) \qquad [\![\hat{e}_1 \oplus \hat{e}_2]\!]_\varepsilon \triangleq \overline{\oplus}([\![\hat{e}_1]\!]_\varepsilon, [\![\hat{e}_2]\!]_\varepsilon)$$

Interpretation of Extended Expressions

$$\begin{array}{l} [\![v]\!]_\varepsilon \triangleq v \\ [\![x]\!]_\varepsilon \triangleq x \\ [\![\hat{x}]\!]_\varepsilon \triangleq \varepsilon(\hat{x}) \end{array} \qquad \frac{[\![\hat{\mathsf{e}}]\!]_\varepsilon = v}{[\![\ominus\, \hat{\mathsf{e}}]\!]_\varepsilon \triangleq \overline{\ominus} v} \qquad \frac{[\![\hat{\mathsf{e}}]\!]_\varepsilon = \hat{\mathsf{e}}' \notin \mathcal{V}_{\mathrm{JSIL}}}{[\![\ominus\, \hat{\mathsf{e}}]\!]_\varepsilon \triangleq \ominus\, \hat{\mathsf{e}}'} \qquad \frac{v = \overline{\oplus}([\![\hat{\mathsf{e}}_1]\!]_\varepsilon, [\![\hat{\mathsf{e}}_2]\!]_{\hat{\rho}})}{[\![\hat{\mathsf{e}}_1 \oplus \hat{\mathsf{e}}_2]\!]_\varepsilon \triangleq v} \qquad \frac{\begin{array}{c}[\![\hat{\mathsf{e}}_1]\!]_\varepsilon = \hat{\mathsf{e}}_1' \quad [\![\hat{\mathsf{e}}_2]\!]_\varepsilon = \hat{\mathsf{e}}_2' \\ \hat{\mathsf{e}}_1' \notin \mathcal{V}_{\mathrm{JSIL}} \vee \hat{\mathsf{e}}_2' \notin \mathcal{V}_{\mathrm{JSIL}}\end{array}}{[\![\hat{\mathsf{e}}_1 \oplus \hat{\mathsf{e}}_2]\!]_\varepsilon \triangleq \hat{\mathsf{e}}_1' \oplus \hat{\mathsf{e}}_2'}$$

We extend this interpretation to heaps, stores, contexts, basic commands, commands, procedures, and programs in the standard way, shown in the Appendix. We say that a symbolic environment is *consistent* with a path condition π, written $\varepsilon \vdash \pi$, if and only if $[\![\pi]\!]_\varepsilon = \mathsf{true}$. Given a symbolic state $(\hat{h}, \hat{\rho}, \hat{C})$ and a path condition π, we define the models of the symbolic state under π, written $\mathcal{M}_\pi(\hat{h}, \hat{\rho}, \hat{C})$, as the set of concrete states that can be obtained from $(\hat{h}, \hat{\rho}, \hat{C})$ using symbolic environments that are consistent with π. Formally:

$$\mathcal{M}_\pi(\hat{h}, \hat{\rho}, \hat{C}) = \left\{ (h, \rho, C, \varepsilon) \mid [\![(\hat{h}, \hat{\rho}, \hat{C})]\!]_\varepsilon = (h, \rho, C) \ \wedge\ \varepsilon \models \pi \right\} \tag{1}$$

$$\mathcal{M}_\pi(\hat{h}, \hat{\rho}) = \left\{ (h, \rho, \varepsilon) \mid [\![\hat{h}]\!]_\varepsilon = h \ \wedge\ [\![\hat{\rho}]\!]_\varepsilon = \rho \ \wedge\ \varepsilon \models \pi \right\} \tag{2}$$

The soundness theorem (Theorem 2.2) states that if we have a symbolic trace captured by $\mathsf{p} : \langle \hat{h}, \hat{\rho}, i, \pi \rangle_{\hat{C}}^{\mu} \rightsquigarrow^* \langle \hat{h}', \hat{\rho}', i', \pi' \rangle_{\hat{C}'}^{\mu'}$ and a concrete state (h, ρ, C) with a symbolic environment ε in the models of the initial symbolic state under the final path condition π', and ε concretises all symbolic variables of the program p, then there exists a concrete symbolic state (h', ρ', C') that is in the models of the final symbolic state under π' with the same symbolic environment ε, such that: $[\![\mathsf{p}]\!]_\varepsilon : \langle h, \rho, i \rangle_C^{\mu} \rightarrow^* \langle h', \rho', i' \rangle_{C'}^{\mu'}$. We use the final path condition π' for both the models of the initial and final symbolic states because we only care about the initial concrete states for which the concrete execution will follow the same path as the symbolic execution.

Theorem 2.2 (Soundness). $\forall\, \mathsf{p}, \hat{h}, \hat{\rho}, i, \pi, \hat{C}, \mu, \hat{h}', \hat{\rho}', i', \pi', \hat{C}', \mu', h, \rho, C, \varepsilon$.

$$\mathsf{p} : \langle \hat{h}, \hat{\rho}, i, \pi \rangle_{\hat{C}}^{\mu} \rightsquigarrow^* \langle \hat{h}', \hat{\rho}', i', \pi' \rangle_{\hat{C}'}^{\mu'} \ \wedge\ (h, \rho, C, \varepsilon) \in \mathcal{M}_{\pi'}(\hat{h}, \hat{\rho}, \hat{C})$$
$$\Rightarrow\ \exists h', \rho', C' \,.\, [\![\mathsf{p}]\!]_\varepsilon : \langle h, \rho, i \rangle_C^{\mu} \rightarrow^* \langle h', \rho', i' \rangle_{C'}^{\mu'} \ \wedge\ (h', \rho', C') \in \mathcal{M}_{\pi'}(\hat{h}', \hat{\rho}', \hat{C}')$$

The *bug-finding* corollary (Corollary 2.3) states that if we find a symbolic trace that results in a failed assertion, then there also exists a concrete execution that will cause that assertion to fail. Observe that the analysis is designed in such a way that there are no false positives, meaning that if we find a failing symbolic trace, we can always instantiate its symbolic values obtaining a concrete counter-model for the failing assertion. This is essential, as Cosette is meant to be a *bug-finding* tool.

Corollary 2.3 (Bug-finding). $\forall\, \mathsf{p}, \hat{h}, \hat{\rho}, i, \pi, \hat{C}, \hat{h}', \hat{\rho}', j, \pi', \hat{C}'$.

$$\mathsf{p} : \langle \hat{h}, \hat{\rho}, i, \pi \rangle_{\hat{C}}^{\top} \rightsquigarrow^* \langle \hat{h}', \hat{\rho}', j, \pi' \rangle_{\hat{C}'}^{\perp}$$
$$\Rightarrow \exists h, \rho, C, \varepsilon \,.\, (h, \rho, C, \varepsilon) \in \mathcal{M}_{\pi'}(\hat{h}, \hat{\rho}, \hat{C}) \ \wedge\ [\![\mathsf{p}]\!]_\varepsilon : \langle h, \rho, i \rangle_C^{\top} \rightarrow^* \langle \_, \_, \_ \rangle_{\_}^{\perp}.$$

Finally, the *verification* corollary (Corollary 2.4) states that if we have symbolically explored all the possible execution paths starting from a given symbolic state $(\hat{h}, \hat{\rho}, \hat{C})$, then the execution of the program starting from any concrete state in the models of the initial symbolic state (under the initial path condition) will result in a final concrete state in the models of one of the final symbolic states (under its associated path condition). As Cosette does not infer loop invariants, if a JSIL program contains loops that cannot be unrolled statically, we will never be in the case of the verification corollary.

Corollary 2.4 (Verification).

$$\begin{array}{l}
\forall\, \mathsf{p}, \hat{h}, \hat{h}_1, \ldots, \hat{h}_n, \hat{\rho}, \hat{\rho}_1, \ldots, \hat{\rho}_n, \hat{C}, \hat{C}_1, \ldots, \hat{C}_n, i, j_1, \ldots, j_n, \pi, \pi_1, \ldots, \pi_n, \mu, \mu_1, \ldots, \mu_n . \\
\quad \wedge_{k=1}^{n} \left(\mathsf{p} : \langle \hat{h}, \hat{\rho}, i, \pi \rangle_{\hat{C}}^{\mu} \rightsquigarrow^{*} \langle \hat{h}_k, \hat{\rho}_k, j_k, \pi_k \rangle_{\hat{C}_k}^{\mu_k} \ |_{k=1}^{n} \right) \ \wedge \ \pi \vdash \bigvee_{k=1}^{n} \pi_k \\
\qquad \Rightarrow \Big(\forall h, \rho, C, \varepsilon \,.\, (h, \rho, C, \varepsilon) \in \mathcal{M}_{\pi}(\hat{h}, \hat{\rho}, \hat{C}) \\
\qquad\quad \Rightarrow \exists k, h', \rho', C' \,.\, [\![\mathsf{p}]\!]_{\varepsilon} : \langle h, \rho, i \rangle_{C}^{\mu} \rightarrow^{*} \langle h', \rho', j_k \rangle_{C'}^{\mu_k} \ \wedge \ (h', \rho', C', \varepsilon) \in \mathcal{M}_{\pi_k}(\hat{h}_k, \hat{\rho}_k, \hat{C}_k) \Big)
\end{array}$$

The **proofs** of the above results are given in the Appendix. What do these results mean with respect to related work? How is this maintainable?

2.2 Implementation

Implementing a symbolic execution engine for JSIL is a non-trivial task, requiring a substantial engineering effort. Hence, instead of implementing the symbolic semantics of JSIL from scratch, we leverage on Rosette [??], a symbolic virtual machine designed to enable swift development of new solver-aided languages. Rosette is a small extension of Racket [?] equipped with a symbolic compiler with support for symbolic values and first order assertions. Because Rosette is itself solver-aided, languages implemented in Rosette can also make use of the solver-aided facilities provided by Rosette. Hence, by implementing a *concrete* JSIL interpreter in Rosette, we obtain *for free* a symbolic interpreter for JSIL. The implementation of the concrete interpreter in Rosette must fulfil the following criteria:

- *Efficiency:* the implementation must promote Rosette's efficient behaviour;
- *Termination:* the user must be given a way to establish a bound for the symbolic execution of programs that loop on symbolic values;
- *Adequacy:* the symbolic execution of the concrete interpreter in Rosette must be consistent with the symbolic semantics described in §2.1. We discuss adequacy in §5 in the context of the global evaluation of Cosette.

Implementing JSIL in Rosette. We first show how to model the concrete JSIL state. Below are our Rosette encodings of JSIL heaps, stores, and call stacks. Heaps are modelled as pairs of locations and lists of property-value pairs, stores as lists of variable-value pairs, and call stacks as lists of lists, each of which contains the Rosette encoding of the appropriate four elements. JSIL variables and function identifiers are modelled as Racket symbols.[2] JSIL values with Rosette correspondents are mapped accordingly; the remaining ones are modelled as Racket symbols (e.g. JSIL types, undefined, null, and empty).

The implementation of JSIL commands precisely follows their concrete semantics as described in §2.1. To show this, let us consider the fragment of the JSIL interpreter that implements the [Property Assignment] rule, shown in Figure 6. The figure shows three functions: **(1)** `(run-bcmd bcmd heap store)`, for executing basic commands, **(2)** `(update-heap heap loc prop val)`, for updating the heap `heap` by setting the value of the the property `prop` of the object denoted by `loc` to `val`, and **(3)**

[2] Racket symbols are uninterpreted values and can be viewed as immutable strings.

```
(define (run-bcmd bcmd heap store)
  (let ((cmd-type (first bcmd)))
    (cond
        [(eq? cmd-type 'p-assign)
         (let* ((loc-val  (run-expr (second bcmd) store))
                (prop-val (run-expr (third bcmd)  store))
                (rhs-val  (run-expr (fourth bcmd) store)))
            (cons (update-heap heap loc-val prop-val rhs-val) store))]
         ...)))

(define (update-heap heap loc prop val)
  (cond
    [(null? heap) (error "Inexistent object")]
    [(equal? (caar heap) loc)
       (cons (cons loc (update-pv-list (cdar heap) prop val)) (cdr heap))]
    [ else (cons (car heap) (update-heap (cdr heap) loc prop val))]))

(define (update-pv-list pv-list prop new-val)
  (cond
    [(null? pv-list) (list (cons prop new-val))]
    [(equal? (caar pv-list) prop) (cons (cons prop new-val) (cdr pv-list))]
    [ else (cons (car pv-list) (update-pv-list (cdr pv-list) prop new-val))]))
```

Fig. 6. Fragment of the JSIL Interpreter in Rosette

(update-pv-list pv-list prop new-val), for updating the property-value list pv-list by setting prop to new-val.

Rosette implementation of the JSIL symbolic state

$$\textsc{Empty Heap}\qquad \mathcal{R}(\emptyset) \triangleq (\textbf{list})$$

$$\textsc{Non-empty Heap}\qquad \frac{\hat{h}_1 = \big((l, \hat{p}_i) \mapsto \hat{v}_i\big)\,|_{i=0}^{n} \qquad (l, -) \notin \mathrm{dom}(\hat{h}_2)}{\mathcal{R}(\hat{h}_1 \uplus \hat{h}_2) \triangleq (\textbf{cons}\ (\textbf{cons}\ l\ (\textbf{list}\ (\textbf{cons}\ \hat{p}_0\ \hat{v}_0) \cdots (\textbf{cons}\ \hat{p}_n\ \hat{v}_n)))\ \mathcal{R}(\hat{h}_2))}$$

$$\textsc{Empty Store}\qquad \mathcal{R}(\emptyset) \triangleq (\textbf{list})$$

$$\textsc{Non-Empty Store}\qquad \mathcal{R}((x : \hat{e}) \uplus \hat{\rho}) \triangleq (\textbf{cons}\ (\textbf{cons}\ (\textbf{quote}\ x)\ \hat{e})\ \mathcal{R}(\hat{\rho}))$$

$$\textsc{Empty Context}\qquad \mathcal{R}([\,]) \triangleq (\textbf{list})$$

$$\textsc{Non-Empty Context}\qquad \mathcal{R}((\mathit{fid}, \hat{\rho}, x, i, j) :: \hat{C}) \triangleq (\textbf{cons}\ (\textbf{list}\ (\textbf{quote}\ \mathit{fid})\ \mathcal{R}(\hat{\rho})\ (\textbf{quote}\ x)\ i\ j)\ \mathcal{R}(\hat{C}))$$

JSIL commands are represented in Rosette as *s-expressions*. For instance, the property assignment basic command $[e_1, e_2] := e_3$ is represented as:

$$(\texttt{list (quote p-assign)}\ \mathcal{R}(\mathsf{e}_1)\ \mathcal{R}(\mathsf{e}_2)\ \mathcal{R}(\mathsf{e}_3))$$

Therefore, the interpreter of basic commands first checks if the first element of the list representing the command is equal to p-assign. If it is, it uses the function run-expr to evaluate the three JSIL expressions comprising the property-assignment in the appropriate order. Then, it invokes the function update-heap to perform the actual update. The function update-heap recursively iterates over all heap objects to find the object whose property is to be updated and, when it finds it, uses the function update-pv-list to update the corresponding field-value list. If it does not find it, it will raise an error. However, due to the concrete semantics of JSIL, this case is never triggered.

Efficiency. It is often possible for more than one transition of the symbolic semantics to be applicable during symbolic execution, giving rise to a potentially intractable number of possible symbolic states. To counter this problem, Rosette uses a sophisticated symbolic state merging algorithm, which factors out the common part between multiple symbolic states in order to expose more opportunities for concrete evaluation. The non-mergeable portions of the state are represented as *guarded symbolic unions*. Our goal is to write the interpreter code in a way that helps Rosette merge sets of possible symbolic states, yielding minimal guarded symbolic unions. To illustrate this point, let us consider the symbolic execution of the property assignment $\mathsf{o}[\hat{\mathsf{s}}] := 42$ in the symbolic heap represented in Rosette as follows: Here, Rosette manages to push the guarded unions to the object cells that may be affected by the property assignment, maintaining a common property-value list for both resulting symbolic states. Now, let us change the code of line 6 of `update-pv-list` (in Figure 6) to:

This change in the interpreter makes the order of the cells in the property-value list change depending on which property gets updated. Now, Rosette needs to create a single guarded union with the two possible resulting property-value lists:

Termination. The JSIL symbolic execution engine does not include the abstraction mechanisms which would allow it to finitise the symbolic execution of loops depending on symbolic values [?]. Hence, the user is asked to specify an upper bound on the number of times the symbolic execution is allowed to branch on symbolic values by using conditional gotos. Once that upper bound is reached, if a conditional goto that branches on a symbolic value is encountered, the symbolic execution stops.

3 SYMBOLIC TESTING FOR JAVASCRIPT

Symbolic execution designed directly on JavaScript is not feasible, for the same reasons verification is not [?]: the semantics of the language is too complex, with numerous intertwined internal functions called under the hood. In particular, the simple assignment alone would have more than twenty potential branchings. Our approach is to symbolically analyse JavaScript code by first compiling it to JSIL, using JS-2-JSIL [?], and then feeding the compiled JSIL code to our JSIL symbolic interpreter, described in §2.

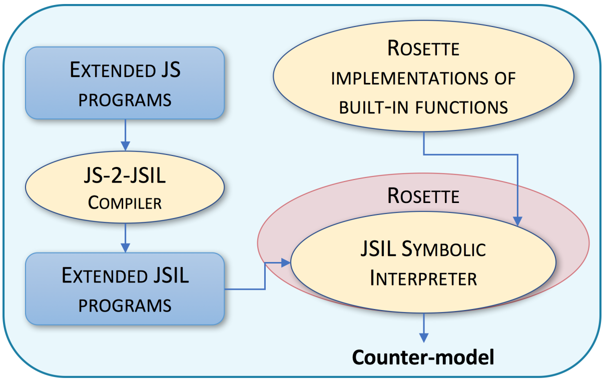

Fig. 7. Cosette: Symbolic execution for JavaScript

In §3.1, we explain how we enable symbolic execution for JavaScript using JS-2-JSIL and Rosette, by extending the language with new constructs for the creation of symbolic values and the checking of assertions. Next, in §3.2, we explain how this symbolic execution can be used for systematic symbolic testing of JavaScript code. Finally, in §3.3, we present a general approach for streamlined symbolic execution of JavaScript built-in libraries, which maximises the use of Rosette's native solver-aided facilities.

3.1 Symbolic Execution by Compilation

Extending JavaScript Syntax. We extend the syntax of JavaScript with logical expressions E, and the following constructs: **(1)** `assert`(E), stating that whenever the *assert* is reached, E must evaluate to true; **(2)** `assume`(E), stating that we *assume* E to hold along the current program path; **(3)** `__s`(), for creating a fresh symbolic string; and **(4)** `__n`(), for creating a fresh symbolic number. The *assert* and *assume* constructs expect as an argument a logical expression. Logical expressions E are given by the grammar $E \triangleq \mathtt{l} \mid \mathtt{x} \mid \ominus E \mid E \oplus E$, where `l` ranges over JavaScript literal values (numbers, booleans, strings, `undefined`, and `null`), `x` over JavaScript variables, $\ominus$ over the JSIL unary operators, and $\oplus$ over the JSIL binary operators. Note that the JavaScript binary and unary operators may have side effects; furthermore, the semantics of these operators often includes several implicit type coercions performed in a specific (and counter-intuitive) order. Hence, we do not allow arbitrary

JavaScript expressions (using JavaScript binary and unary operators) as the arguments for the *assume* and the *assert* constructs. Instead, we use the JSIL unary and binary operators, which have a very clear and simple semantics, without coercions or side-effects.

Extending JS-2-JSIL. Instead of giving a formal semantics for the newly introduced constructs, we explain their meaning by showing their compilation to JSIL. Importantly, the JavaScript variable store is emulated in the heap. Hence, a JavaScript variable x is not compiled to a JSIL variable x, but to a sequence of JSIL commands for retrieving the value of x from the heap cell in which it is stored. A full description of JS-2-JSIL is out of the scope of this paper; see [?] for further details. In the following, we will assume to have a function $\mathcal{C} : \mathcal{S}_{\text{JS}} \rightharpoonup \texttt{List}(Cmd) * \mathcal{X}$, mapping JavaScript expressions and statements to lists of JSIL commands paired up with JSIL variables. In a nutshell, $\mathcal{C}(s) = ([c_1, ..., c_n], x)$ means that the compilation of the JavaScript statement s results in the list of JSIL commands $[c_1, ..., c_n]$, and that after the execution of these commands, the value to which s evaluates in the JavaScript semantics is stored in the JSIL variable x. Below we show the extension of $\mathcal{C}$ for the constructs introduced above. The definition of $\mathcal{C}$ relies on an auxiliary compiler $\mathcal{C}_l : \mathcal{E}^L_{\text{JS}} \rightharpoonup \texttt{List}(Cmd) * \mathcal{E}_{\texttt{JSIL}}$, for translating JavaScript logical expressions.

Extension $\mathcal{C} : \mathcal{S}_{\text{JS}} \rightharpoonup \texttt{List}(Cmd) * \mathcal{X}$ **and** $\mathcal{C}_l : \mathcal{E}^L_{\text{JS}} \rightharpoonup \texttt{List}(Cmd) * \mathcal{E}_{\texttt{JSIL}}$

$$\textsc{Assume}\quad \frac{\mathcal{C}_l(E) = [c_1, \ldots, c_n], e \qquad x' \text{ fresh} \qquad c_{n+1} = x' := \mathsf{empty} \quad c_{n+2} = \mathsf{assume}\,(e)}{\mathcal{C}(\mathsf{assume}(E)) \triangleq [c_1, \ldots, c_n, c_{n+1}, c_{n+2}], x'}$$

$$\textsc{Assert}\quad \frac{\mathcal{C}_l(E) = [c_1, \ldots, c_n], e \qquad x' \text{ fresh} \qquad c_{n+1} = x' := \mathsf{empty} \quad c_{n+2} = \mathsf{assert}\,(e)}{\mathcal{C}(\mathsf{assert}(E)) \triangleq [c_1, \ldots, c_n, c_{n+1}, c_{n+2}], x'}$$

$$\textsc{Symbolic String}\quad \frac{\hat{s} \text{ fresh} \qquad x \text{ fresh}}{\mathcal{C}(\mathsf{\_\_s}()) \triangleq [x := \hat{s}], x}$$

$$\textsc{Symbolic Number}\quad \frac{\hat{n} \text{ fresh} \qquad x \text{ fresh}}{\mathcal{C}(\mathsf{\_\_n}()) \triangleq [x := \hat{n}], x}$$

$$\textsc{Literal}\quad \mathcal{C}_l(\mathtt{l}) \triangleq [], \mathtt{l}$$

$$\textsc{JS Var}\quad \frac{\mathcal{C}(\mathsf{x}) = [c_1, \ldots, c_n], x}{\mathcal{C}_l(\mathsf{x}) \triangleq [c_1, \ldots, c_n], x}$$

$$\textsc{Unary Operator}\quad \frac{\mathcal{C}_l(E) = [c_1, \ldots, c_n], x}{\mathcal{C}_l(\ominus E) \triangleq [c_1, \ldots, c_n], \ominus x}$$

$$\textsc{Binary Operator}\quad \frac{\mathcal{C}_l(E_1) = [c_1, \ldots, c_n], x_1 \quad \mathcal{C}_l(E_2) = [c'_1, \ldots, c'_n], x_2}{\mathcal{C}_l(E \oplus E) \triangleq [c_1, \ldots, c_n, c'_1, \ldots, c'_n], x_1 \oplus x_2}$$

3.2 Symbolic Testing by Example

We illustrate how Cosette can be used to write symbolic tests for JavaScript code by using the JavaScript implementation of a *key-value map* given in Figure 8 (left). This implementation contains four functions: `Map`, for constructing an empty map; `get`, for retrieving the value associated with the key given as input; `put`, for inserting a new *key-value pair* into the map and updating the values of existing keys; and `validKey`, for deciding whether a key is valid.

Prototype chains and `Object.prototype`. In order to better understand the implementation of the map library as well as its possible bugs, one must first understand the *prototype-based inheritance* mechanism of JavaScript. Every JavaScript object has a prototype, which (for presentation purposes) we assume to be stored in an internal property `@proto`. In order to determine the value of a property `p` of an object `o`, the semantics first checks if `o` has a property named `p`, in which case the property look-up yields its value. Otherwise, the semantics checks if `p` belongs to the properties of the prototype of `o` and so forth. Hence, in the example, when looking up the value of the property `hasOwnProperty` of the object `contents`, one gets the value associated with the property `hasOwnProperty` of its prototype. The sequence of objects that can be accessed from a given object through the inspection of the respective prototypes is called a *prototype chain*. Prototype chains typically finish with the object `Object.prototype` from which JavaScript programs can access a number of built-in functions, which are part of the language runtime environment and are used for inspecting and manipulating objects. An example of such a function is `hasOwnProperty(p)`, which checks whether or not the object on which it is invoked has the property `p` (e.g. `map.hasOwnProperty("_contents")` evaluates

```
function Map () { this._contents = {} }

Map.prototype.get = function (k) {
  var c = this._contents;
  if (c.hasOwnProperty(k)) {
    return this._contents[k]
  } else { return null }
}

Map.prototype.put = function (k, v) {
  var c = this._contents;
  if this._contents.validKey(k)) {
    contents[k] = v
  } else
    throw new Error("Invalid Key");
}

Map.prototype.validKey = function (k) {      }
```

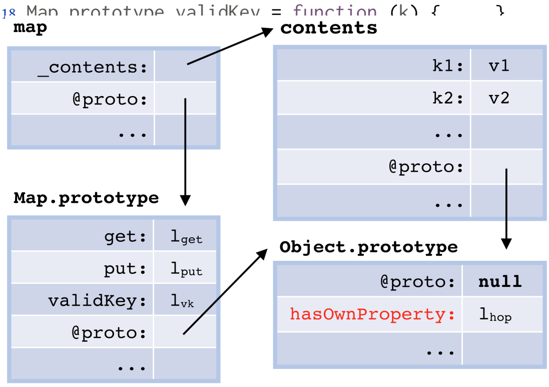

Fig. 8. JS map implementation (left) and example of a map library heap (right)

to true when evaluated in the heap shown in Fig. 8-(right), because the object map has a property named "_contents").

Bug-finding. The map library implements a *key-value map* as an object with property _contents, denoting the object storing the map contents. The named properties of _contents and their value attributes correspond to the map keys and values, respectively. The functions get, put, and validKey are to be shared between all map objects. Therefore, they are defined in Map.prototype, which is the prototype of all objects created using Map as a constructor (i.e. using new Map()). Note that one can insert a key-value pair with "hasOwnProperty" as a key into the map. By doing this, "hasOwnProperty" in the prototype chain of _contents is overridden and subsequent calls to get will fail. Consider the following symbolic test:

```
var s1 = __s(); var n1 = __n();
var m = new Map(); m.put(s1, n1); var r = m.get(s1);
assert(n1 = r)
```

The symbolic test above checks for a desired property of the library—if we were to put a key/value pair (k, v) into the map, then we should be able to retrieve the value v using the key k. We can run Cosette on this test to reveal the bug discusses above. Indeed, Cosette generates the failing model: s1 = "hasOwnProperty".

This example also highlights how Cosette does not require specialist knowledge, and can, therefore, be used by almost any JavaScript developer. The annotation burden amounts to the creation of symbolic variables and the writing of assertions, remaining minimal and intuitive, in stark contrast with the standard annotation burden of verification tools.

3.3 Supporting JavaScript Built-in Libraries

JavaScript comes equipped with a rich runtime environment (described in Chapter 15 of the ES5 standard [?]), which consists of *built-in libraries* that support advanced manipulation of, for example, objects, arrays, strings, regular expressions, and dates. In this section, we identify two important challenges related to supporting the symbolic execution of JavaScript programs that interact with built-in libraries and explain how we solve them in Cosette.

JS-2-JSIL models built-in library functions as JSIL procedures, which can be called from within the compiled JavaScript code. However, the current runtime environment of JS-2-JSIL does not support all built-in libraries described in the standard. In particular, it doesn't support regular expressions, parts of the String library that use regular expressions, parts of the Date library, and JSON objects. The partiality of JS-2-JSIL when it comes to supporting JavaScript built-in libraries gives rise to the first challenge:

> *Challenge 1:* Cosette should allow for modular addition of implementations of built-in libraries not yet covered by JS-2-JSIL.

Most JavaScript library functions are described in the language standard in terms of more elementary operations. For instance, operations on strings are often described in terms of operations on the characters that comprise the string, and often involve loops (c.f. `String.lastIndexOf`, Ch.15.5.4.8 [?]). The JSIL implementations of the JavaScript built-in libraries follow the standard line-by-line, emphasising full adherence to the language standard. However, for some of these built-in library functions, direct Rosette correspondents exist. In such cases, we argue that these correspondents should be used instead of the functions provided by JS-2-JSIL, to minimise potential branching and looping on symbolic values. Hence, we state the second challenge as follows:

> *Challenge 2:* Cosette should prioritise native Rosette operations to the operations provided by JS-2-JSIL in the cases in which the native Rosette operations and JavaScript operations exactly match.

Rosette models of JavaScript built-in functions. To solve these two challenges, Cosette supports the on-the-fly extension of the JSIL interpreter with Rosette implementations of JavaScript built-in functions, which we call Rosette *models*. Hence, every time the interpreter evaluates a procedure call, it first checks whether or not it corresponds to a built-in function for which there is a Rosette model. If it does, instead of executing the standard procedure call rule, the interpreter will instead execute the appropriate Rosette model.

Below, we show a simplified Rosette model for the built-in `String.prototype.replace` function (see Ch.15.5.4.11 [?]). This model uses the natively supported function `string-replace`, taking advantage of Rosette reasoning capabilities. This example illustrates an important point about the difference between JavaScript and Rosette operations: `string-replace` works only with strings, but `String.prototype. replace` can take arguments of any type that then get coerced to strings. Since JS-2-JSIL does not have an implementation of `String.replace`, the Rosette model will report an error if it receives non-string arguments.

```
(define (replace str from to)
  (if (and (string? str) (string? from) (string? to))
      (list (quote normal) (string-replace str from to #:all? #f))
      (error "Unsupported call to String.prototype.replace")))
```

Bug-finding with strings. By using the appropriate Rosette models for string operations, Cosette can find bugs in non-trivial JavaScript programs that manipulate strings. For instance, consider the following naive JavaScript implementation of a string sanitiser meant to remove all occurrences of *script tags* inside untrusted strings (for instance, those coming from user input or untrusted servers). To this end, the programmer chooses to replace all occurrences of the string `"script"` in the string given as input with `"s"`.

```
function sanitise (str) { return str.replace("script", "s") }
```

Now, we can use Cosette to test if the sanitiser meets its purpose, that is, that all strings, after being sanitised, do not contain the substring `"script"`:

```
var s = sanitise(__s());
var x = ! (s.contains("script")); d
assert(x)
```

Cosette will, in fact, be able to come up with a counter-model for this assertion. Concretely, this counter-model is `s1 = "scriptcript"`. Despite its simplicity, this example illustrates the complexities underpinning the design and implementation of robust string sanitisers, a commonly used defense mechanism against XSS attacks [?].

4 DEBUGGING SEPARATION LOGIC SPECIFICATIONS

We show how to use Cosette for debugging JavaScript code annotated with separation logic (SL) specifications. Tools that allow for SL-reasoning about functional correctness properties in general, and those targeting JavaScript in particular, require of the user to have substantial expertise and to go through a long and complex proof trace in order to find the precise source of an error. Cosette substantially simplifies this process by providing concrete counter-models that invalidate the specification. In §4.1, we extend the JSIL symbolic interpreter with a mechanism for asserting SL-assertions. In §4.2, we show how to implement this mechanism by giving a sound decision procedure for finding counter-models for SL-assertions. In §4.3, we present an algorithm for generating symbolic tests from SL-specifications, which guarantees that if a symbolic test fails, then Cosette must produce a concrete counter-model that invalidates the specification. Finally, in §4.4, we show how to use the proposed methodology for debugging JaVerT specifications [?] of JavaScript code.

4.1 JSIL Symbolic Execution with Separation Logic Assertions

We extend JSIL with a special construct, $\mathsf{assert}_*\,(P)$, for stating that the separation logic assertion P must hold whenever $\mathsf{assert}_*\,(P)$ is evaluated. We use the assertion language of [?], with the following adaptation: instead of *untyped logical variables*, we use *typed logical variables*, $\hat{x} \in \hat{\mathcal{X}}$, which include symbolic numbers, $\hat{n} \in \hat{\mathcal{N}}$, strings $\hat{s} \in \hat{\mathcal{S}}$, and locations, $\hat{l} \in \hat{\mathcal{L}}$.

JSIL assertions: syntax and semantics. JSIL assertions include: boolean operations; first-order connectives; the separating conjunction; existential quantification; and assertions for describing heaps. The `emp` assertion describes an empty heap. The cell assertion, $(E_1, E_2) \mapsto E_3$, describes an object at the location denoted by E_1 with a property denoted by E_2 that has the value denoted by E_3. The assertion $\mathsf{emptyFields}(E_1 \mid E_2)$ states that the object at the location denoted by E_1 has no properties other than possibly those included in the set denoted by E_2. As in [??], in order to define the semantics of assertions, we resort to *instrumented heaps* $\underline{h} \in \mathcal{H}_\emptyset$, which differ from concrete heaps in that they may map object properties to the special value $\varnothing$, explicitly indicating that the property does not exist (e.g. $\underline{h}(l, s) = \varnothing$ means that the object at location l in the heap $\underline{h}$ does not have a property named s). Instrumented heaps are related to heaps by means of an *erasure function*, $\lfloor . \rfloor : \mathcal{H}_\emptyset \to \mathcal{H}$, which simply removes the none-cells from the instrumented heap given as input. Below, we give the syntax and semantics of JSIL assertions. Note that we assume pure assertions to have an empty spatial footprint and allow logical negation, conjunction, and disjunction of pure formulae only.

Symbolic State			
Heap	*PC*	*Assertion*	*Failing Constraint*
$(l, \hat{p}_1) \mapsto \hat{v}$	true	$(l, \hat{p}_2) \mapsto \hat{v}$	$\hat{p}_1 \neq \hat{p}_2$
$(l, \hat{p}_1) \mapsto \hat{v}$	true	$(l, \hat{p}_1) \mapsto \hat{v} * (l, \hat{p}_2) \mapsto \varnothing$	$\hat{p}_1 = \hat{p}_2$
$(l, \hat{p}_1) \mapsto \hat{v}$	$\hat{p}_1 \neq \hat{p}_2$	$(l, \hat{p}_1) \mapsto \hat{v} * (l, \hat{p}_2) \mapsto \varnothing * \mathsf{emptyFields}(l \mid \{\hat{p}_2, \hat{p}_3\})$	$\hat{p}_1 \neq \hat{p}_3$
$(l, \hat{p}_1) \mapsto \hat{v}$	$\hat{p}_1 \notin \{\hat{p}_2, \hat{p}_3\}$	$(l, \hat{p}_1) \mapsto \hat{v} * (l, \hat{p}_2) \mapsto \varnothing * (l, \hat{p}_3) \mapsto \varnothing$	$\hat{p}_2 = \hat{p}_3$

Table 1. Symbolic States vs Assertions

JSIL Logic Assertions - Syntax and Semantics

$$
\begin{array}{lll}
E \triangleq & \lambda \mid x \mid \hat{x} \mid \ominus E \mid E \oplus E & \text{Logical Expressions} \\
P_\pi \triangleq & \mathsf{true} \mid \mathsf{false} \mid \neg P_\pi \mid P_\pi \wedge P_\pi \mid P_\pi \vee P_\pi \mid E = E \mid E \leq E & \text{Pure Assertions} \\
P \triangleq & P_\pi \mid \mathsf{emp} \mid (E, E) \mapsto E \mid \exists \hat{x}.P \mid P * P \mid \mathsf{emptyFields}(E \mid E) & \text{Assertions}
\end{array}
$$

$$
\begin{array}{lcl}
\underline{h}, \rho, \varepsilon \models \mathsf{emp} & \Leftrightarrow & \underline{h} = \emptyset \\
\underline{h}, \rho, \varepsilon \models (E_1, E_2) \mapsto E_3 & \Leftrightarrow & \underline{h} = ([\![E_1]\!]_{\rho,\varepsilon}, [\![E_2]\!]_{\rho,\varepsilon}) \mapsto [\![E_3]\!]_{\rho,\varepsilon} \\
\underline{h}, \rho, \varepsilon \models P * Q & \Leftrightarrow & \exists \underline{h}_1, \underline{h}_2 . \, \underline{h} = \underline{h}_1 \uplus \underline{h}_2 \ \wedge \ \underline{h}_1, \rho, \varepsilon \models P \ \wedge \ \underline{h}_2, \rho, \varepsilon \models Q \\
\underline{h}, \rho, \varepsilon \models \mathsf{emptyFields}(E_1 \mid E_2) & \Leftrightarrow & \underline{h} = \biguplus_{s \notin \{[\![E_2]\!]_{\rho,\varepsilon}\}} (([\![E_1]\!]_{\rho,\varepsilon}, s) \mapsto \varnothing)
\end{array}
$$

For convenience, we define:

$$\mathcal{M}_*(P) = \left\{ (h, \rho, \varepsilon) \mid \exists \underline{h} \, . \, h = \lfloor \underline{h} \rfloor \ \wedge \ \underline{h}, \rho, \varepsilon \models P \right\} \tag{3}$$

Given a symbolic heap $\hat{h}$, a symbolic store $\hat{\rho}$, a path condition π, and an assertion P, we say that $(\hat{h}, \hat{\rho}, \pi)$ *satisfies* P, written $\hat{h}, \hat{\rho}, \pi \models P$ *if and only if* $\mathcal{M}_\pi(\hat{h}, \hat{\rho}) \subseteq \mathcal{M}_*(P)$. We can now give an *ideal* symbolic semantics for the command $\mathsf{assert}_*\,(P)$ (which checks if the current symbolic state satisfies P):

the configurations need to be updated with the bot or top in the appropriate place. It appears 3 times: below, when the rules are re-stated with a call to the decision procedure, in Theorem 8.

$$
\textsc{Assert - True} \qquad \frac{\hat{h}, \hat{\rho}, \pi \models P}{\langle \hat{h}, \hat{\rho}, \mathsf{assert}_*\,(P), \pi \rangle \rightsquigarrow \langle \hat{h}, \hat{\rho}, \pi \rangle}
\qquad\qquad
\textsc{Assert - False} \qquad \frac{\hat{h}, \hat{\rho}, \pi \not\models P}{\langle \hat{h}, \hat{\rho}, \mathsf{assert}_*\,(P), \pi \rangle \rightsquigarrow \langle \bot, \pi \rangle}
$$

Determining whether or not a symbolic state satisfies an SL-assertion P is, in general, undecidable [?]. Since we do not want to produce *false positives*, in order to trigger an assertion failure, we need to find a concrete witness for that failure. More precisely, when executing $\mathsf{assert}_*\,(P)$ in the symbolic state $(\hat{h}, \hat{\rho}, \pi)$, the symbolic analysis must report an assertion failure only if it can find a concrete state (h, ρ) and a symbolic environment ε such that: $(h, \rho, \varepsilon) \in \mathcal{M}_\pi(\hat{h}, \hat{\rho})$ and $(h, \rho, \varepsilon) \notin \mathcal{M}_*(P)$.

4.2 Finding Counter Models for SL Assertions

We describe a partial decision procedure, which we implement as part of the JSIL symbolic interpreter, for proving entailments between symbolic states and SL-assertions and finding counter models in case of failure. As it is customary [???], the decision procedure works by first using *pattern-matching* on the spatial part of the SL-assertion, and then discharging the pure part of the entailment to an external constraint solver (in our case, Rosette).

Representation of SL-assertions. We target SL-assertions P that can be represented as quadruples, $(\mathsf{L}, \mathsf{C}, \mathsf{EF}, \pi)$, consisting of: **(1)** a set L of existentially quantified symbolic locations, **(2)** a list C containing the non-none cell assertions (those whose value is different from $\varnothing$), **(3)** a set EF containing the none-cell assertions and the empty-fields assertions, and **(4)** a pure assertion π. Hence, letting $\mathsf{L} = \{\hat{l}_1, \ldots, \hat{l}_k\}$, $\mathsf{C} = [\mathsf{c}_i \mid_{i=0}^{n}]$, and $\mathsf{EF} = \{\mathsf{ef}_i \mid_{i=0}^{m}\}$, we write $P \equiv (\mathsf{L}, \mathsf{C}, \mathsf{EF}, \pi)$ as shorthand for:

$$P = \exists \hat{l}_1, \ldots, \hat{l}_k \, . \Big(\mathop{\circledast}_{0 \leq i \leq n} \mathsf{c}_i \ * \mathop{\circledast}_{0 \leq i \leq m} \mathsf{ef}_i \Big) \ * \ \pi \tag{4}$$

Cell Assertion

$$\frac{\hat{h} = \hat{h}_f \uplus ((l, \hat{p}') \mapsto \hat{v}') \qquad \pi \vdash \hat{p} = \hat{p}' \land \hat{v} = \hat{v}'}{\pi \vdash \langle \hat{h}, ((l, \hat{p}) \mapsto \hat{v}) :: \mathsf{C} \rangle \rightarrow_{CU} \langle \hat{h}_f, \mathsf{C} \rangle}$$

Cell Assertion - Fail

$$\frac{\hat{h} = \hat{h}' \uplus \left((l, \hat{p}_i) \mapsto \hat{v}_i\right) |_{i=0}^{n} \quad l \notin \mathsf{locs}(\hat{h}') \qquad \pi_i = (\hat{p}_i = \hat{p}' \land \hat{v}_i = \hat{v}') |_{i=0}^{n} \quad \pi \nvdash \pi_i \; |_{i=0}^{n}}{\pi \vdash \langle \hat{h}, ((l, \hat{p}) \mapsto \hat{v}) :: \mathsf{C} \rangle \rightarrow_{CU} \mathsf{F}(\land_{0 \leq i \leq n} \neg \pi_i)}$$

Fig. 9. Unification of non-none cells: $\pi \vdash \langle \hat{h}, \mathsf{C} \rangle \rightarrow_{CU} \langle \hat{h}_f, \mathsf{C}' \rangle$ and $\pi \vdash \langle \hat{h}, \mathsf{C} \rangle \rightarrow_{CU} \mathsf{F}(\pi')$

Given a symbolic state $(\hat{h}, \hat{\rho}, \pi)$ and an assertion $P \equiv (\mathsf{L}, \mathsf{C}, \mathsf{EF}, \pi)$, we represent each possible mapping from the existentially quantified symbolic locations in P to the concrete locations in $\hat{h}$ as a *substitution function* $\theta : \hat{\mathcal{L}} \rightharpoonup \mathcal{L}$. Since both L and the set of concrete locations in $\hat{h}$ are finite, we conclude that there is a finite number of substitution functions to be considered when checking if $(\hat{h}, \hat{\rho}, \pi)$ satisfies P. Hence, in the following, we will assume a fixed substitution, θ. Furthermore, we assume that the SL-assertion given as input to the decision procedure does not contain any program variables. This we justify by noting that the following equivalence holds (the proof can be found in the Appendix):

$$\hat{h}, \hat{\rho}, \pi \models P \iff \hat{h}, \emptyset, \pi \models [\![P]\!]_{\hat{\rho}}, \tag{5}$$

where $[\![P]\!]_{\hat{\rho}}$ denotes the SL-assertion obtained by symbolically evaluating all the symbolic expressions in P under $\hat{\rho}$.

Unification rules. In Figure 9, we show the unification rules for non-none cell assertions. We write $\pi \vdash \langle \hat{h}, \mathsf{c} :: \mathsf{C} \rangle \rightarrow_{CU} -$ to denote the unification of a symbolic heap $\hat{h}$ against a single non-none cell c, given a path condition π.[3] This unification can either terminate successfully, with $\langle \hat{h}_f, \mathsf{C} \rangle$, in which case $\hat{h}_f$ denotes the symbolic heap that does not contain the footprint of c, and C denotes the list of remaining non-none cell assertions to be unified (Cell Assertion); or unsuccessfully, with $\mathsf{F}(\pi')$, in which case π' captures the constraints required for the unification to provably fail (Cell Assertion - Fail). For instance, in the first example of Table 1, in order to generate a witness for the entailment failure, we need to instantiate $\hat{p}_1$ and $\hat{p}_2$ so that the *failing constraint* $\hat{p}_1 \neq \hat{p}_2$ is satisfied.

Unification of negative resource, that is, none-cells and `emptyFields` assertions, is more intricate, as symbolic heaps do not maintain negative information. We denote by $\mathcal{U}_{NR}(\hat{h}, \mathsf{EF})$ the constraints that need to be satisfied for the unification of a symbolic heap $\hat{h}$ against the negative resource denoted by EF to succeed, and show the rules in Figure 10. First, unifying a symbolic heap $\hat{h}$ that contains the object l with properties $\hat{p}_i|_{i=0}^{n}$ against a none-cell $(l, \hat{p}) \mapsto \varnothing$ effectively means that $\hat{p}$ has to be different from all $\hat{p}_i|_{i=0}^{n}$ (NR-None Cell). Next, unifying a symbolic heap $\hat{h}$ that contains the object l with properties $\hat{p}_i|_{i=0}^{n}$ against an `emptyFields` assertion $\mathsf{emptyFields}(l \mid \hat{e}_d)$ means that all of the properties $\hat{p}_i|_{i=0}^{n}$ have to be in the domain of of the `emptyFields` assertion, $\hat{e}_d$ (NR-Empty Fields). The second and third examples of Table 1 illustrate unification failures resulting from NR-constraints not being satisfied: the second example targets the (NR-None Cell) rule, with the failing constraint: $\hat{p}_1 \neq \hat{p}_2$; and the third example targets the (NR-Empty Fields) rule, with the failing constraint $\hat{p}_1 \notin \{\, \hat{p}_2, \hat{p}_3 \,\}$.

Furthermore, due to the semantics of the separating conjunction, there are additional constraints imposed by the negative resource. Concretely, all none-cells for the same object have to have different property names, and if any none-cell is starred together with an `emptyFields` assertion for the same object, then its property name has to be in the domain of that `emptyFields` assertion (Separation-Constraints - Fixed Location). We call these constrants *separation constraints*, and denote them, for a fixed location l, by $\mathcal{S}_l(\mathsf{EF})$ in Figure 10. The point is that, if a given symbolic state entails a negative resource assertion, it must be the case that its path condition entails the

[3] We denote the transitive closure of $\rightarrow_{CU}$ by $\rightarrow_{CU}^{*}$.

NR-None Cell

$$\frac{\hat{h} = \hat{h}' \uplus \left((l, \hat{p}_i) \mapsto -\right) |_{i=0}^{n} \qquad l \notin \mathsf{locs}(\hat{h}') \quad \pi = \wedge_{0 \leq i \leq n} \hat{p} \neq \hat{p}_i}{\mathcal{U}_{NR}(\hat{h}, (l, \hat{p}) \mapsto \varnothing) = \pi}$$

NR-Empty Fields

$$\frac{\hat{h} = \hat{h}' \uplus \left((l, \hat{p}_i) \mapsto -\right) |_{i=0}^{n} \qquad l \notin \mathsf{locs}(\hat{h}') \quad \pi = \{\hat{p}_i \mid_{i=0}^{n}\} \subseteq \hat{e}_d}{\mathcal{U}_{NR}(\hat{h}, \mathsf{emptyFields}(l \mid \hat{e}_d)) = \pi}$$

NR-Unification

$$\mathcal{U}_{NR}(\hat{h}, \mathsf{EF}) = \bigwedge_{\mathsf{ef} \in \mathsf{EF}} \mathcal{U}_{NR}(\hat{h}, \mathrm{ef})$$

Separation Constraints - Fixed Location

$$\frac{\mathsf{EF} \mid_l = \mathsf{emptyFields}(l \mid \hat{e}_d) * \circledast_{i=0}^{n} \left((l, \hat{p}_i) \mapsto \varnothing\right)}{\mathcal{S}_l(\mathsf{EF}) = (\wedge_{0 \leq i, j \leq n, i \neq j} \hat{p}_i \neq \hat{p}_j) \wedge (\wedge_{0 \leq i \leq n} \hat{p}_i \in \hat{e}_d)}$$

Separation Constraints

$$\mathcal{S}(\mathsf{EF}) = \bigwedge_{l \in \mathsf{locs}(\mathsf{EF})} \mathcal{S}_l(\mathsf{EF})$$

Fig. 10. Unification of negative-resource assertions: $\mathcal{U}_{NR}(\hat{h}, \mathsf{EF}) = \pi$ and $\mathcal{S}(\mathsf{EF}) = \pi$

Fail - Cell Unification

$$\frac{\pi \vdash \langle \hat{h}, \mathsf{C} \rangle \rightarrow^{*}_{CU} \mathsf{F}(\pi'')}{\langle (\hat{h}, \pi), (\mathsf{C}, \mathsf{EF}, \pi') \rangle \in \mathcal{U}_{\mathsf{F}}(\pi'')}$$

Fail - Extra Resource

$$\frac{\pi \vdash \langle \hat{h}, \mathsf{C} \rangle \rightarrow^{*}_{CU} \langle \hat{h}_f, [] \rangle \qquad \hat{h}_f \neq \emptyset}{\langle (\hat{h}, \pi), (\emptyset, \mathsf{EF}, \pi') \rangle \in \mathcal{U}_{\mathsf{F}}(\mathsf{true})}$$

Fail - Pure Entailment

$$\frac{\pi \vdash \langle \hat{h}, \mathsf{C} \rangle \rightarrow^{*}_{CU} \langle \emptyset, [] \rangle \qquad \pi'' = \mathcal{U}_{NR}(\hat{h}, \mathsf{EF}) \wedge \mathcal{S}(\mathsf{EF}) \quad \pi \nvdash \pi' \wedge \pi''}{\langle (\hat{h}, \pi), (\mathsf{C}, \mathsf{EF}, \pi') \rangle \in \mathcal{U}_{\mathsf{F}}(\neg(\pi' \wedge \pi''))}$$

Success

$$\frac{\pi \vdash \langle \hat{h}, \mathsf{C} \rangle \rightarrow^{*}_{CU} \langle \emptyset, [] \rangle \qquad \pi \vdash \pi' \wedge \mathcal{U}_{NR}(\hat{h}, \mathsf{EF}) \wedge \mathcal{S}(\mathsf{EF})}{\langle (\hat{h}, \pi), (\mathsf{C}, \mathsf{EF}, \pi') \rangle \in \mathcal{U}_{\mathsf{S}}}$$

Fig. 11. Unification algorithm: $\langle (\hat{h}, \pi), (\mathsf{C}, \mathsf{EF}, \pi') \rangle \in \mathcal{U}_{\mathsf{S}}$ and $\langle (\hat{h}, \pi), (\mathsf{C}, \mathsf{EF}, \pi') \rangle \in \mathcal{U}_{\mathsf{F}}(\pi')$.

corresponding separation constraints. The last example of Table 1 illustrates a unification failure resulting from an EF-separation-constraint not being satisfied, namely $\hat{p}_2 \neq \hat{p}_3$.

Finally, the (NR-Unification) and (Separation Constraints) rules extend $\mathcal{U}_{NR}(\hat{h}, \mathsf{EF})$ and $\mathcal{S}_l(\mathsf{EF})$, respectively, to sets of negative resource formulas.

Unification Algorithm. Figure 11 presents the rules for the *unification algorithm*. We write $\langle (\hat{h}, \pi), (\mathsf{C}, \mathsf{EF}\pi') \rangle \in \mathcal{U}_{\mathsf{S}}$ to mean that the unification of the symbolic state $\hat{h}, \pi$ against the assertion $P \equiv (\emptyset, \mathsf{C}, \mathsf{EF}, \pi')$ succeeds, and $\langle (\hat{h}, \pi), (\mathsf{C}, \mathsf{EF}, \pi') \rangle \in \mathcal{U}_{\mathsf{F}}(\pi')$ to mean that it fails, with the *failing constraints* being: π'. Note that in order to generate a witness for the unification failure, one needs to find a symbolic environment satisfying the failing constraints.

The algorithm works by trying to unify the non-none cell assertions first. The unification succeeds when they are all successfully unified, the resulting symbolic heap is empty, and the final pure entailment is discharged by the constraint solver (Success). On the other hand, the unification can fail in three ways: a non-none cell may not be unifiable (Fail - Cell Unification); all non-none cells are unifiable, but there may be additional resource left (Fail - Extra Resource); and all non-none cells are unifiable, there is no additional resource left, but the final pure entailment may not be discharged by the constraint solver (Fail - Pure Entailment). Note that in case of failure, the algorithm always returns the constraints that need to be satisfied for a witness of that failure to be produced.

Below we present the rules of the unification algorithm for a general assertion P. For clarity, we use $\mathsf{substs}(\mathsf{L}, \hat{h})$ to denote the set of all substitutions mapping symbolic location in L to concrete locations in the domain of $\hat{h}$. For the success case, we simply need to find a substitution θ for which the unification algorithm terminates successfully. For the failure case, we need to compute the failing constraints for all possible substitutions, and return their conjunction.

Unification of General Assertions

Success

$$\frac{\exists \theta.\, [\![\theta(P)]\!]_{\hat{\rho}} \equiv (\emptyset, \mathsf{C}, \mathsf{EF}, \pi') \qquad \langle (\hat{h}, \pi), (\mathsf{C}, \mathsf{EF}, \pi') \rangle \in \mathcal{U}_{\mathsf{S}}}{\langle (\hat{h}, \hat{\rho}, \pi), P \rangle \in \mathcal{U}_{\mathsf{S}}}$$

Failure

$$\frac{P \equiv (\mathsf{L}, -, -, -) \qquad \mathsf{substs}(\mathsf{L}, \hat{h}) = \{\theta_1, \ldots, \theta_n\} \qquad \forall_{1 \leq k \leq n}.\, \left([\![\theta(P)]\!]_{\hat{\rho}} \equiv (\emptyset, \mathsf{C}, \mathsf{EF}, \pi') \wedge \langle (\hat{h}, \pi), (\mathsf{C}, \mathsf{EF}, \pi') \rangle \in \mathcal{U}_{\mathsf{F}}(\pi_k)\right)}{\langle (\hat{h}, \hat{\rho}, \pi), P \rangle \in \mathcal{U}_{\mathsf{F}}(\wedge_{0 \leq k \leq n} \pi_k)}$$

```
$\mathcal{T}_{\mathsf{nm}}(\mathit{fid}, \hat{x}_i \mid_{i=0}^{n}, \pi, Q) \triangleq$
    proc main () {
        0                   : assume ($\pi$)
        1                   : $x := \mathit{fid}(\hat{x}_0, \ldots, \hat{x}_n)$ with $i_{\mathsf{er}}$
        $i_{\mathsf{nm}}$   : assert$_*$ ($Q[x/\mathtt{ret}]$)
        $i_{\mathsf{er}}$   : assert (false)
    }
```

Generated testing procedure for normal-return specification:
a. Assume the initial path condition
b. Call the procedure to be tested *fid* with symbolic arguments $\hat{x}_i \mid_{i=0}^{n}$
c. In case of normal return, assert the nm-postcondition of *fid*
d. In case of error return, assert false

```
$\mathcal{T}_{\mathsf{er}}(\mathit{fid}, \hat{x}_i \mid_{i=0}^{n}, \pi, Q) \triangleq$
    proc main () {
        0                   : assume ($\pi$)
        1                   : $x := \mathit{fid}(\hat{x}_0, \ldots, \hat{x}_n)$ with $i_{\mathsf{er}}$
        $i_{\mathsf{nm}}$   : assert (false)
        $i_{\mathsf{er}}$   : assert$_*$ ($Q[x/\mathtt{err}]$)
    }
```

Generated testing procedure for error-return specification:
a. Assume the initial path condition
b. Call the procedure to be tested *fid* with symbolic arguments $\hat{x}_i \mid_{i=0}^{n}$
c. In case of normal return, assert false
d. In case of error return, assert the er-postcondition of *fid*

```
$\mathcal{T}(\{P\}\, \mathit{fid}(\overline{x})\, \{Q\}^{\mathit{fl}}) \triangleq$
    let $\theta$ = pick [slocs($P$) $\to \mathcal{L}\backslash$locs($P$)] in
    let $(-, \mathsf{C}, \mathsf{EF}, \pi) = \theta(P)$ in
    let $\hat{\rho} = [x_i \mapsto \hat{x}_i \mid_{i=0}^{n}]$ in
    let $\hat{h} = [\![\mathsf{C}]\!]_{\hat{\rho}}$ in
    let $\pi' = \mathcal{S}([\![\mathsf{EF}]\!]_{\hat{\rho}}) \wedge \mathcal{U}_{NR}(\hat{h}, [\![\mathsf{EF}]\!]_{\hat{\rho}})$ in
    let $\pi'' = [\![\theta(\pi)]\!]_{\hat{\rho}} \wedge \pi'$ in
    let $\mathit{proc} = \mathcal{T}_{\mathit{fl}}(\mathit{fid}, \hat{x}_i \mid_{i=0}^{n}, \pi'', [\![\theta(Q)]\!]_{\hat{\rho}})$ in
        $(\mathit{proc}, \hat{h})$
```

a. Compute a substitution from the symbolic locations in P to randomly picked concrete locations not overlapping with those in P
b. Apply the substitution to P obtaining $(-, \mathsf{C}, \mathsf{EF}, \pi)$
c. Compute the initial store mapping each argument to a symbolic value of the appropriate type
d. Symbolically evaluate EF under $\hat{\rho}$, obtaining $\hat{h}$
e. Compute the appropriate NR- and separation constraints
f. Construct the initial path condition from π and π'
g. Synthesise the symbolic test

Fig. 12. Symbolic Test Generation Algorithm

We can now give the *implemented* symbolic semantics for the command assert$_*$ (P) which makes use of the unification algorithm described above:

Assert - True

$$\frac{\langle(\hat{h}, \hat{\rho}, \pi), P\rangle \in \mathcal{U}_{\mathsf{S}}}{\langle \hat{h}, \hat{\rho}, \mathsf{assert}_*\ (P), \pi\rangle \rightsquigarrow \langle \hat{h}, \hat{\rho}, \pi\rangle}$$

Assert - False

$$\frac{\langle(\hat{h}, \hat{\rho}, \pi), P\rangle \in \mathcal{U}_{\mathsf{F}}(\pi') \qquad (\pi \wedge \pi') \text{ is satisfiable}}{\langle \hat{h}, \hat{\rho}, \mathsf{assert}_*\ (P), \pi\rangle \rightsquigarrow \langle \bot, \pi \wedge \pi'\rangle}$$

Soundness. Theorem 4.1 states that the unification algorithm is sound: given an SL-assertion P and a symbolic state $(\hat{h}, \hat{\rho}, \pi)$, if $\langle(\hat{h}, \hat{\rho}, \pi), P\rangle \in \mathcal{U}_{\mathsf{S}}$ then the symbolic state $(\hat{h}, \hat{\rho}, \pi)$ satisfies P. The bug-finding theorem, Theorem 4.2, is more subtle. It states that, in case of failure, to find a counter-model for P, one has to pick a concretisation of the symbolic state that is consistent with the failing constraint generated by the unification algorithm.

Theorem 4.1 (Soundness of Unification). $\forall P, \hat{h}, \hat{\rho}, \pi$.

$$\langle(\hat{h}, \hat{\rho}, \pi), P\rangle \in \mathcal{U}_{\mathsf{S}} \implies \mathcal{M}_{\pi}(\hat{h}, \hat{\rho}) \subseteq \mathcal{M}_*(P)$$

Theorem 4.2 (Bug-finding for SL). $\forall P, \hat{h}, \hat{\rho}, \pi$.

$$\langle(\hat{h}, \hat{\rho}, \pi), P\rangle \in \mathcal{U}_{\mathsf{F}}(\pi') \implies \mathcal{M}_{\pi \wedge \pi'}(\hat{h}, \hat{\rho}) \cap \mathcal{M}_*(P) = \emptyset$$

The **proofs** of the above results are given in the Appendix.

4.3 Compiling JSIL Logic Specifications to Symbolic Tests

JSIL Logic specifications. JSIL Logic specifications have the form $\{P\}\, \mathit{fid}(\overline{x})\, \{Q\}^{\mathit{fl}}$, where P and Q are the pre- and postconditions of the function with identifier *fid*, and $\overline{x}$ its list of formal parameters. Each specification is associated with a return mode $\mathit{fl} \in \{\mathsf{nm}, \mathsf{er}\}$, indicating if the function returns normally or with an error. If it returns normally, then its return value can be accessed via a dedicated variable `ret`, and `err` otherwise. Intuitively, a specification $\{P\}\, \mathit{fid}(\overline{x})\, \{Q\}^{\mathit{fl}}$ is valid for a given JSIL program p, if p contains a procedure with identifier *fid* and "whenever *fid* is executed in a state satisfying P, then, if it terminates, it does so in a state satisfying Q, with return mode *fl*". The formal definition is given below.

Definition 4.3 (Validity of JSIL Logic Specifications). A JSIL logic specification $\{P\}\, fid(\overline{x})\, \{Q\}^{fl}$ is valid with respect to a program p, written $\mathsf{p} \models \{P\}\, fid(\overline{x})\, \{Q\}^{fl}$, if and only if, for all logical contexts $(\underline{h}, \rho, \varepsilon)$, heaps h_f, stores ρ_f, and flags fl', it holds that:

$$\begin{array}{l} \underline{h}, \rho, \varepsilon \models P \ \wedge\ \langle \lfloor \underline{h} \rfloor, \rho, C[0] \rangle \rightarrow^* \langle h_f, \rho_f, C[i_{fl'}] \rangle \\ \quad \implies fl' = fl \ \wedge\ \exists \underline{h}_f \,.\, \underline{h}_f, \rho_f, \varepsilon \models Q \ \wedge\ \lfloor \underline{h}_f \rfloor = h_f \end{array}$$

Symbolic test generation. Given a JSIL program p containing a procedure *fid* with spec $\{P\}\, fid(\overline{x})\, \{Q\}^{fl}$, our goal is to construct a symbolic test for checking whether or not *fid* behaves as its specification mandates. A symbolic test is a pair $(proc, \hat{h})$ consisting of a JSIL procedure with the code of the test and the initial symbolic heap on which to execute the test. Figure 12 presents the test generation procedure. Intuitively, $\mathcal{T}(\{P\}\, fid(\overline{x})\, \{Q\}^{fl})$ returns the symbolic test for $\{P\}\, fid(\overline{x})\, \{Q\}^{fl}$. The test generation function $\mathcal{T}$ is defined in terms of two auxiliary functions, $\mathcal{T}_{\mathsf{nm}}$ and $\mathcal{T}_{\mathsf{er}}$, for generating tests for nm-mode and er-mode specifications, respectively. The test program p', denoted by $\mathsf{p}[\mathsf{main} \mapsto proc]$, is obtained from the original program p and the test procedure *proc* by replacing the main of p with the new test procedure, *proc*. Finally, Theorem 4.4 states that if the symbolic execution of the test generated for $\{P\}\, fid(\overline{x})\, \{Q\}^{fl}$ finds a bug, then the specification is not valid.

Theorem 4.4 (Bug-finding for SL Specifications).

$$\begin{array}{l} \mathcal{T}(\{P\}\, fid(\overline{x})\, \{Q\}^{fl}) = (proc, \hat{h}) \ \wedge\ \mathsf{p}[main \mapsto proc] : \langle \hat{h}, [], i, \pi \rangle_{\hat{C}^0_{\mathsf{main}}} \rightsquigarrow^* \langle \bot, \pi' \rangle \\ \qquad \implies \mathsf{p} \not\models \{P\}\, fid(\overline{x})\, \{Q\}^{fl} \end{array}$$

where $\hat{C}^0_{\mathsf{main}}$ *denotes the initial call stack:* $[(main, -, -, -)]$.

Symbolic States versus SL-Assertions. Note that the conversion of the precondition P of a JSIL procedure *fid* to a symbolic state $(\hat{h}, \hat{\rho}, \pi)$ involves transforming the negative resource assertions in P into pure constraints encoded by the initial path condition π and not captured by $\hat{h}$. We will illustrate this with an example in the following section.

Inductive Predicates. Cosette does not have built-in abstraction mechanisms. Hence, it does not support symbolic execution over inductive predicates describing recursive data structures, which are commonplace in separation-logic-style specifications [??]. As in [?], we deal with user-defined inductive predicate assertions by *unfolding* those assertions up to a fixed bound, which is established by the user. This unfolding mechanism is routine and its details are, therefore, omitted from the paper.

4.4 Compiling JaVerT Specifications to Symbolic Tests

JaVerT Specifications. JaVerT specifications of JavaScript functions are analogous to JSIL specifications of JSIL procedures, and are of the form $\{P_{\mathsf{JS}}\}\, fid(\overline{x})\, \{Q_{\mathsf{JS}}\}^{fl}$. A JaVerT specification $\{P_{\mathsf{JS}}\}\, fid(\overline{x})\, \{Q_{\mathsf{JS}}\}^{fl}$ is valid for a JavaScript program s, written $s \models \{P_{\mathsf{JS}}\}\, fid(\overline{x})\, \{Q_{\mathsf{JS}}\}^{fl}$, if and only if s contains a function literal with identifier *fid* and "whenever *fid* is executed in a state satisfying P_{JS}, then, if it terminates, it does so in a state satisfying Q_{JS}, with return mode *fl*". The assertion language used by JaVerT is very similar to the assertion language of JSIL Logic and is described in detail in [?]. There, the authors present a compiler from JaVerT specifications to JSIL Logic specifications, denoted by C_A, and prove the soundness of that compiler, assuming a correct compiler C from JavaScript to JSIL. This theoretical result ensures that the verification of JSIL programs can be lifted to the verification

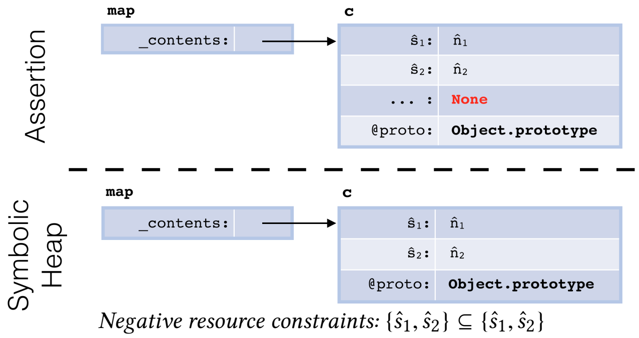

Fig. 13. Assertion vs. Symbolic Heap: Map(map, $\{(\hat{s}_1, \hat{n}_1), (\hat{s}_2, \hat{n}_2)\}$)

of JavaScript programs. More concretely, a JaVerT specification $\{P_{\mathrm{JS}}\}\, fid(\overline{x})\, \{Q_{\mathrm{JS}}\}^{fl}$ is valid for a given JavaScript program s if and only if the translated specification $C_A(\{P_{\mathrm{JS}}\}\, fid(\overline{x})\, \{Q_{\mathrm{JS}}\}^{fl})$ is valid for the compilation of s by a correct compiler C. Put formally:

$$s \vDash \{P_{\mathrm{JS}}\}\, fid(\overline{x})\, \{Q_{\mathrm{JS}}\}^{fl} \iff C(s) \vDash C_A(\{P_{\mathrm{JS}}\}\, fid(\overline{x})\, \{Q_{\mathrm{JS}}\}^{fl}). \tag{6}$$

We generate symbolic tests from JavaScript programs and their JaVerT specifications in the following way. First, we convert the JavaScript program and its JaVerT specification to a JSIL program with its JSIL Logic specification, using the JS-2-JSIL and C_A compilers. Next, we generate a set of symbolic tests from the obtained JSIL program and JSIL Logic specification, as described in §4.3. Finally, we run the generated JSIL symbolic tests on the JSIL symbolic execution engine. If Cosette finds a bug while running these tests, we will obtain a concrete counter-example triggering that bug.

Example. We illustrate the debugging of SL-specifications by appealing to the Map example shown in Figure 8. In order to reason about a key-value map, we define several predicates, whose definitions we show below.

```
Map (m, kvs) :=
  DataProp(m, "_contents", c) * JSObject(c) *
    KVPairs(c, kvs) * first(kvs, keys) * emptyFields(c, keys)

KVPairs (o, kvs) :=
  (kvs = { }),
  (kvs = (k, v) -u- kvs') * ValidKey(k) * DataProp(o, k, v) * KVPairs(o, kvs')

ValidKey (k) := types(k : Str) * (k <> "hasOwnProperty")
```

The Map predicate captures the resource corresponding to a map object. Concretely, it first states that the map object has the property _contents, which points to a default JavaScript object c, using the predicates DataProp and JSObject. DataProp(o, p, v) captures the property p of object o and states that it has value v, while abstracting over other associated JavaScript internals, whereas JSObject(o) states that the object o is an extensible object of class "Object", whose prototype is Object.prototype (for more details, see [?]). Next, using the KVPairs predicate (explained shortly), it states that c holds the key-value pairs kvs. Finally, it states that c has no other properties except the keys present in kvs. For this, it first obtains the set of keys from the set of key-value pairs kvs using the predicate first(kvs, keys), which states that the first projection of kvs equals keys (its definition is standard), and then uses the emptyFields assertion to state that all other properties are absent from the object.

The KVPairs(o, kvs) predicate talks about key-value pairs of an object o. It is defined recursively on the structure of kvs and it has two definitions, separated by a comma. We have that kvs is either

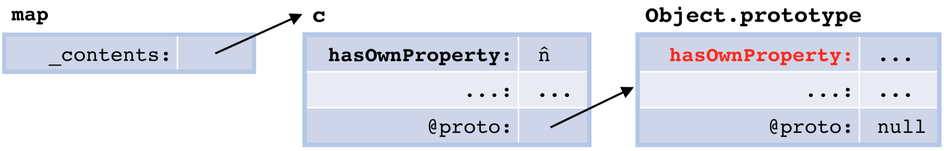

Fig. 14. Property shadowing: c.hasOwnProperty(...) cannot reach Object.prototype.

empty or that it contains at least one key-value pair (k, v).[4] In the latter case, we state that the key k must be valid, that the object o has the property k with value v, and proceed recursively. Note that the uniqueness of keys in kvs is guaranteed by the DataProp predicate of KVPairs and the separating conjunction.

The ValidKey(k) predicate captures the validity of a given key and holds *iff* the corresponding JavaScript function validKey(k) returns true. In the definition of ValidKey, we highlight in red a potential source of errors on which we will focus shortly.

To give a better intuition of how the Map predicate works, we show the full unfolding of Map(map, $\{(\hat{s}_1, \hat{n}_1), (\hat{s}_2, \hat{n}_2)\}$) in Figure 13. There, we can also see how the negative resource captured by the SL-assertion Map(map, $\{(\hat{s}_1, \hat{n}_1), (\hat{s}_2, \hat{n}_2)\}$), namely the resource captured by emptyFields(c, first(kvs)), disappears from the symbolic heap and is transformed into the negative resource constraint $\{\hat{s}_1, \hat{s}_2\} \subseteq \{\hat{s}_1, \hat{s}_2\}$, which states that all properties of the object c in the symbolic heap (in our case, $\hat{s}_1$ and $\hat{s}_2$) must be in the set of properties of the corresponding emptyFields assertion (in our case, first($\{(\hat{s}_1, \hat{n}_1), (\hat{s}_2, \hat{n}_2)\}$) = $\{\hat{s}_1, \hat{s}_2\}$). Such constraints are generated in item **e**. of the test generation algorithm presented in Figure 12.

Below, we show the relevant parts of the specifications of get(k) and put(k, v), for the case in which k already exists in the map:

```
{  Map(this, kvs -u- (k, v)) * ObjProtoF() *  }
{ (this, "@proto") -> mp * MapProto(mp) * ... }
                    get(k)
        { Precondition * (ret = v) }
```

```
{  Map(this, kvs -u- (k, v')) * ObjProtoF() *  }
{ (this, "@proto") -> mp * MapProto(mp) * ... }
                   put(k, v)
{  Map(this, kvs -u- (k, v)) * ObjProtoF() *  }
{ (this, "@proto") -> mp * MapProto(mp) * ... }
```

The predicate ObjProtoF() describes the resource captured by the Object.prototype object. In particular, it is needed because get uses the hasOwnProperty function, which is defined as a property of Object.prototype. The predicate MapProto specifies the resource of a valid map prototype: in particular, the map prototype needs to define the methods put, get, and validKey. Finally, note that, given the definition of the Map and KVPairs predicates, both preconditions shown entail that k is a valid key.

Now, if we forgot to state the part of the ValidKey(k) predicate highlighted in red, that is, if we did not state that k needed to be different from "hasOwnProperty", the symbolic test generated for the specification of get would fail for unfoldings of KVPairs of depth ≥ 1, with the counter-model k = "hasOwnProperty". In that case, as illustrated in Figure 14, the "hasOwnProperty" property of Object.prototype would no longer be reachable by property lookup from c, and the execution of line 5 (if (c.hasOwnProperty(k))) would raise an error, as it would attempt to call the "hasOwnProperty" property of object c as a function instead. Since this specification of get(k) requires normal termination, the jump to the error label in the compiled JSIL code will trigger the assert (false) of the generated symbolic test and the developer will be presented with the counter-model k = "hasOwnProperty".

What else would we like to say here?

[4] We write -u- for set union and omit the brackets around singleton sets.

5 EVALUATION

6 RELATED WORK

The existing literature covers a wide range of analysis techniques for JavaScript programs, including: type systems [???????], control flow analysis [?], pointer analysis [??] and abstract interpretation [????], among others. Here, we focus on the existing work on logic-based analysis and symbolic execution for JavaScript.

Symbolic Execution. Ooga. Booga. Boo.

Logic-based Analysis. [?] have developed a separation logic for a small fragment of ECMAScript 3, to reason about the variable store emulated in the JavaScript heap. [?] have developed $\mathbb{K}$, a term-rewriting framework for formalising the operational semantics of programming languages. In particular, they have developed KJS [?] which provides a $\mathbb{K}$-interpretation of the core language and part of the built-in libraries of the ES5 standard. KJS has been tested against the official ECMAScript Test262 test suite and passed all 2782 tests for the core language; the testing results for the built-in libraries are not reported. [?] introduce a language-independent verification infrastructure that can be instantiated with a $\mathbb{K}$-interpretation of a language to automatically generate a symbolic verification tool for that language based on the $\mathbb{K}$ reachability logic. They apply this infrastructure to KJS to generate a verification tool for JavaScript, which they use to verify functional correctness properties of operations for manipulating data structures such as binary search trees, AVL trees, and lists.

7 CONCLUSIONS

*** PM : Can we be more general, and say something like ’logic-based specifications’? It’s all about translating to FOL, or even some version of PL. Also, we need to say at some point why we care about specifications written in separation logic. ***

A JSIL SYNTAX AND SEMANTICS

Numbers: $n \in \mathcal{N}$ Booleans: $b \in \mathcal{B}$ Strings: $s \in \mathcal{S}$ Locs: $l \in \mathcal{L}$ Vars: $x \in \mathcal{X}$

Types: $\tau \in \mathtt{Types}$ Values: $v \in \mathcal{V}_{\mathtt{JSIL}} ::= n \mid b \mid s \mid \mathsf{undefined} \mid \mathsf{null} \mid l \mid \tau \mid \mathit{fid} \mid \mathsf{empty}$

Expressions: $e \in \mathcal{E}_{\mathtt{JSIL}} ::= v \mid x \mid \ominus\, e \mid e \oplus e$

Basic Commands: $bc \in \mathcal{B}cmd ::= \mathsf{skip} \mid x := e \mid x := \mathsf{new}\,() \mid x := [e, e] \mid [e, e] := e \mid$
$\mathsf{delete}\,(e, e) \mid x := \mathsf{hasField}\,(e, e) \mid x := \mathsf{getFields}\,(e) \mid \mathsf{assume}\,(e) \mid \mathsf{assert}\,(e)$

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

Procedures : $\mathit{proc} \in \mathtt{Proc} ::= \mathsf{proc}\ \mathit{fid}(\overline{x})\{\overline{\mathsf{c}}\}$ Programs : $\mathsf{p} \in \mathcal{P}\ :\ \mathit{fid} \rightharpoonup \mathtt{Proc}$

Fig. 15. Syntax of the JSIL Language

Skip
$$\langle \sigma, \mathsf{skip}\,\rangle \to \sigma$$

Property Collection
$$\frac{\rho = \sigma.\mathsf{sto} \quad \mathcal{GD}(\sigma, e) = v \\ \rho' = \rho[x \mapsto v]}{\langle \sigma, x := \mathsf{getFields}\,(e)\rangle \to \sigma[\mathsf{sto} \mapsto \rho']}$$

Assignment
$$\frac{\rho = \sigma.\mathsf{sto} \quad \mathcal{E}v(e, \rho) = v \\ \rho' = \rho[x \mapsto v]}{\langle \sigma, x := e\rangle \to \sigma[\mathsf{sto} \mapsto \rho']}$$

Object Creation
$$\frac{h = \sigma.\mathsf{hp} \quad d = \sigma.\mathsf{dom} \quad \rho = \sigma.\mathsf{sto} \quad (l, -) \notin \mathsf{dom}(h) \\ h' = \mathcal{HU}(h, (l, \mathit{@proto}), \mathsf{null}) \quad d' = d \uplus (l \mapsto \{\, \mathit{@proto}\,\})}{\langle \sigma, x := \mathsf{new}\,()\rangle \to \sigma[\mathsf{hp} \mapsto h', \mathsf{sto} \mapsto \rho[x \mapsto l], \mathsf{dom} \mapsto d']}$$

Property Access
$$\frac{\rho = \sigma.\mathsf{sto} \quad \mathcal{GC}(\sigma, (e_1, e_2)) = (-, -, v) \\ v \neq \varnothing \quad \rho' = \rho[x \mapsto v]}{\langle \sigma, x := [e_1, e_2]\rangle \to \sigma[\mathsf{sto} \mapsto \rho']}$$

Property Assignment
$$\frac{h = \sigma.\mathsf{hp} \quad \rho = \sigma.\mathsf{sto} \quad \mathcal{GC}(\sigma, (e_1, e_2)) = (l, p, -) \\ \mathcal{E}v(e_3, \rho) = v \quad h' = \mathcal{HU}(h, (l, p), v)}{\langle \sigma, [e_1, e_2] := e_3\rangle \to \sigma[\mathsf{hp} \mapsto h']}$$

Property Deletion
$$\frac{h = \sigma.\mathsf{hp} \quad \mathcal{GC}(\sigma, (e_1, e_2)) = (l, p, v) \\ v \neq \varnothing \quad h' = \mathcal{HU}(h, (l, p), \varnothing)}{\langle \sigma, \mathsf{delete}\,(e_1, e_2)\rangle \to \sigma[\mathsf{hp} \mapsto h']}$$

Member Check - True
$$\frac{\rho = \sigma.\mathsf{sto} \quad \mathcal{GC}(\sigma, (e_1, e_2)) = (-, -, v) \\ v \neq \varnothing \quad \rho' = \rho[x \mapsto \mathsf{true}]}{\langle \sigma, x := \mathsf{hasField}\,(e_1, e_2)\rangle \to \sigma[\mathsf{sto} \mapsto \rho']}$$

Member Check - True
$$\frac{\rho = \sigma.\mathsf{sto} \quad \mathcal{GC}(\sigma, (e_1, e_2)) = (-, -, \varnothing) \\ \rho' = \rho[x \mapsto \mathsf{false}]}{\langle \sigma, x := \mathsf{hasField}\,(e_1, e_2)\rangle \to \sigma[\mathsf{sto} \mapsto \rho']}$$

Fig. 16. Execution for Basic Commands: $\langle \sigma, bc\rangle \to \sigma'$

Basic Command
$$\frac{\mathsf{cmd}(\mathsf{p}, cs, i) = bc \\ \langle \sigma, bc\rangle \to \sigma'}{\langle \sigma, cs, i\rangle^{\top} \to \langle \sigma', cs, i{+}1\rangle^{\top}}$$

Cond. Goto - True
$$\frac{\mathsf{cmd}(\mathsf{p}, cs, i) = \mathsf{goto}\ [e]\, j,\ k \\ \mathcal{E}v(e, \sigma.\mathsf{sto}\,) = \mathsf{true}}{\langle \sigma, cs, i\rangle^{\top} \to \langle \sigma, cs, j\rangle^{\top}}$$

Cond. Goto - False
$$\frac{\mathsf{cmd}(\mathsf{p}, cs, i) = \mathsf{goto}\ [e]\, j,\ k \\ [\![e]\!]_{\sigma.\mathsf{sto}} = \mathsf{false}}{\langle \sigma, cs, i\rangle^{\top} \to \langle \sigma, cs, k\rangle^{\top}}$$

Goto
$$\frac{\mathsf{cmd}(\mathsf{p}, cs, i) = \mathsf{goto}\ j}{\langle \sigma, cs, i\rangle^{\top} \to \langle \sigma, cs, j\rangle^{\top}}$$

Normal Return
$$\frac{cs = (-, \rho', x, i, -) :: cs' \quad \rho = \sigma.\mathsf{sto} \\ \sigma' = \sigma[\mathsf{sto} \mapsto \rho'[x \mapsto \rho(\mathsf{ret})]]}{\langle \sigma, cs, i_{\mathsf{nm}}\rangle^{\top} \to \langle \sigma', cs', i\rangle^{\top}}$$

Error Return
$$\frac{cs = (-, \rho', x, -, j) :: cs' \quad \rho = \sigma.\mathsf{sto} \\ \sigma' = \sigma[\mathsf{sto} \mapsto \rho'[x \mapsto \rho(\mathsf{err})]]}{\langle \sigma, cs, i_{\mathsf{er}}\rangle^{\top} \to \langle \sigma', cs', j\rangle^{\top}}$$

Procedure Call
$$\frac{\mathsf{cmd}(\mathsf{p}, cs, i) = x := e(e_i\ |_{i=0}^{n})\ \mathsf{with}\, j \quad \rho = \sigma.\mathsf{sto} \quad \mathcal{E}v(e, \rho) = f' \quad \mathsf{args}(f', \mathsf{p}) = [x_1, \ldots, x_m] \\ v_i = \mathcal{E}v(e_i, \rho)\ |_{i=0}^{n} \quad v_i = \mathsf{undefined}\ |_{i=n+1}^{m} \quad \rho' = [x_i \mapsto v_i\ |_{i=0}^{m}] \quad cs' = (f', \rho, x, i{+}1, j) :: cs}{\langle \sigma, cs, i\rangle^{\top} \to \langle \sigma[\mathsf{sto} \mapsto \rho'], cs', 0\rangle^{\top}}$$

Assume
$$\frac{\mathsf{cmd}(\mathsf{p}, cs, i) = \mathsf{assume}\,(e) \\ \mathcal{E}v(e, \sigma.\mathsf{sto}\,) = \mathsf{true}}{\langle \sigma, cs, i\rangle^{\top} \to \langle \sigma, cs, i{+}1\rangle^{\top}}$$

Assert - True
$$\frac{\mathsf{cmd}(\mathsf{p}, cs, i) = \mathsf{assert}\,(e) \\ \mathcal{E}v(e, \sigma.\mathsf{sto}\,) = \mathsf{true}}{\langle \sigma, cs, i\rangle^{\top} \to \langle \sigma, cs, i{+}1\rangle^{\top}}$$

Assert - False
$$\frac{\mathsf{cmd}(\mathsf{p}, cs, i) = \mathsf{assert}\,(e) \\ \mathcal{E}v(e, \sigma.\mathsf{sto}\,) = \mathsf{false}}{\langle \sigma, cs, i\rangle^{\top} \to \langle \sigma, cs, i\rangle^{\bot}}$$

Fig. 17. Execution for Control Flow Commands: $\langle \sigma, cs, i\rangle^{\mu} \to \langle \sigma', cs', j\rangle^{\mu'}$

Notation

SYMBOLIC JSIL EXPRESSIONS:	$\hat{e} \in \hat{\mathcal{E}}_{\text{JSIL}}$	$::= \; v \mid \hat{s} \mid \hat{n} \mid \ominus\, \hat{e} \mid \hat{e} \oplus \hat{e}$
EXTENDED JSIL EXPRESSIONS:	$\hat{e} \in \underline{\hat{\mathcal{E}}}_{\text{JSIL}}$	$::= \; v \mid x \mid \hat{s} \mid \hat{n} \mid \ominus\, \hat{e} \mid \hat{e} \oplus \hat{e}$

SKIP

$$\langle \Sigma, \text{skip} \rangle \rightsquigarrow \Sigma$$

PROPERTY COLLECTION

$$\frac{\mathsf{P} = \Sigma.\text{sto} \quad \mathcal{GD}(\Sigma, e) \rightsquigarrow \mathsf{v} \qquad \mathsf{P}' = \mathsf{P}[x \mapsto \mathsf{v}]}{\langle \Sigma, x := \text{getFields}\,(e) \rangle \rightsquigarrow \Sigma[\text{sto} \;\mapsto \mathsf{P}']}$$

ASSIGNMENT

$$\frac{\mathsf{P} = \Sigma.\text{sto} \quad \mathcal{E}v(e, \mathsf{P}) = \mathsf{v} \qquad \mathsf{P}' = \mathsf{P}[x \mapsto \mathsf{v}]}{\langle \Sigma, x := e \rangle \rightsquigarrow \Sigma[\text{sto} \;\mapsto \mathsf{P}']}$$

OBJECT CREATION

$$\frac{\mathsf{h} = \Sigma.\text{hp} \quad \mathsf{d} = \Sigma.\text{dom} \quad \mathsf{P} = \Sigma.\text{sto} \quad (l, -) \notin \text{dom}(\mathsf{h}) \qquad \mathsf{h}' = \mathcal{HU}(\mathsf{h}, (l, @proto), \text{null}) \quad \mathsf{d}' = \mathsf{d} \uplus (l \mapsto \{\, @proto \,\})}{\langle \Sigma, x := \text{new}\,() \rangle \rightsquigarrow \Sigma[\text{hp} \;\mapsto \mathsf{h}', \text{sto} \;\mapsto \mathsf{P}[x \mapsto l], \text{dom} \;\mapsto \mathsf{d}']}$$

PROPERTY ACCESS

$$\frac{\mathcal{GC}(\Sigma, (e_1, e_2)) \rightsquigarrow \Sigma', (-, -, \mathsf{v}) \quad \mathsf{P} = \Sigma'.\text{sto} \qquad \mathsf{v} \neq \varnothing \quad \mathsf{P}' = \mathsf{P}[x \mapsto \mathsf{v}]}{\langle \Sigma, x := [e_1, e_2] \rangle \rightsquigarrow \Sigma'[\text{sto} \;\mapsto \mathsf{P}']}$$

PROPERTY ASSIGNMENT

$$\frac{\mathcal{GC}(\Sigma, (e_1, e_2)) \rightsquigarrow \Sigma', (\mathtt{l}, \mathsf{p}, -) \quad \mathsf{h} = \Sigma'.\text{hp} \quad \mathsf{P} = \Sigma'.\text{sto} \qquad \mathcal{E}v(e_3, \mathsf{P}) = \mathsf{v} \quad \mathsf{h}' = \mathcal{HU}(\mathsf{h}, (\mathtt{l}, \mathsf{p}), \mathsf{v})}{\langle \Sigma, [e_1, e_2] := e_3 \rangle \rightsquigarrow \Sigma'[\text{hp} \;\mapsto \mathsf{h}']}$$

PROPERTY DELETION

$$\frac{\mathcal{GC}(\Sigma, (e_1, e_2)) \rightsquigarrow \Sigma', (\mathtt{l}, \mathsf{p}, \mathsf{v}) \quad \mathsf{h} = \Sigma'.\text{hp} \qquad \mathsf{v} \neq \varnothing \quad \mathsf{h}' = \mathcal{HU}(\mathsf{h}, (\mathtt{l}, \mathsf{p}), \varnothing)}{\langle \Sigma, \text{delete}\,(e_1, e_2) \rangle \rightsquigarrow \Sigma'[\text{hp} \;\mapsto \mathsf{h}']}$$

MEMBER CHECK - TRUE

$$\frac{\mathcal{GC}(\Sigma, (e_1, e_2)) \rightsquigarrow \Sigma', (-, -, \mathsf{v}) \quad \mathsf{P} = \Sigma'.\text{sto} \qquad \Sigma'' = \mathcal{A}sm(\Sigma', \mathsf{v} \neq \varnothing) \quad \mathsf{P}' = \mathsf{P}[x \mapsto \text{true}]}{\langle \Sigma, x := \text{hasField}\,(e_1, e_2) \rangle \rightsquigarrow \Sigma''[\text{sto} \;\mapsto \mathsf{P}']}$$

MEMBER CHECK - TRUE

$$\frac{\mathcal{GC}(\Sigma, (e_1, e_2)) \rightsquigarrow \Sigma', (-, -, \mathsf{v}) \quad \mathsf{P} = \Sigma'.\text{sto} \qquad \Sigma'' = \mathcal{A}sm(\Sigma', \mathsf{v} = \varnothing) \quad \mathsf{P}' = \mathsf{P}[x \mapsto \text{false}]}{\langle \Sigma, x := \text{hasField}\,(e_1, e_2) \rangle \rightsquigarrow \Sigma''[\text{sto} \;\mapsto \mathsf{P}']}$$

Fig. 18. Abstract Execution for Basic Commands: $\langle \Sigma, bc \rangle \rightsquigarrow \Sigma'$

BASIC COMMAND

$$\frac{\text{cmd}(\mathsf{p}, \mathsf{cs}, i) = bc \qquad \langle \Sigma, bc \rangle \rightsquigarrow \Sigma'}{\langle \Sigma, \mathsf{cs}, i \rangle^{\top} \rightsquigarrow \langle \Sigma', \mathsf{cs}, i{+}1 \rangle^{\top}}$$

COND. GOTO - TRUE

$$\frac{\text{cmd}(\mathsf{p}, \mathsf{cs}, i) = \text{goto}\;\; [e]\; j,\;\; k \qquad \mathcal{A}sm(\Sigma, e = \text{true}) = \Sigma'}{\langle \Sigma, \mathsf{cs}, i \rangle^{\top} \rightsquigarrow \langle \Sigma', \mathsf{cs}, j \rangle^{\top}}$$

COND. GOTO - FALSE

$$\frac{\text{cmd}(\mathsf{p}, \mathsf{cs}, i) = \text{goto}\;\; [e]\; j,\;\; k \qquad \mathcal{A}sm(\Sigma, e = \text{false}) = \Sigma'}{\langle \Sigma, \mathsf{cs}, i \rangle^{\top} \rightsquigarrow \langle \Sigma', \mathsf{cs}, k \rangle^{\top}}$$

GOTO

$$\frac{\text{cmd}(\mathsf{p}, cs, i) = \text{goto}\; j}{\langle \Sigma, \mathsf{cs}, i \rangle^{\top} \rightsquigarrow \langle \Sigma, \mathsf{cs}, j \rangle^{\top}}$$

NORMAL RETURN

$$\frac{\mathsf{cs} = (-, \mathsf{P}', x, i, -) :: \mathsf{cs}' \quad \mathsf{P} = \sigma.\text{sto} \qquad \Sigma' = \Sigma[\text{sto} \;\mapsto \mathsf{P}'[x \mapsto \mathsf{P}(\mathtt{ret})]]}{\langle \Sigma, \mathsf{cs}, i_{\mathsf{nm}} \rangle^{\top} \rightsquigarrow \langle \Sigma', \mathsf{cs}', i \rangle^{\top}}$$

ERROR RETURN

$$\frac{\mathsf{cs} = (-, \mathsf{P}', x, -, j) :: \mathsf{cs}' \quad \mathsf{P} = \Sigma.\text{sto} \qquad \Sigma' = \Sigma[\text{sto} \;\mapsto \mathsf{P}'[x \mapsto \mathsf{P}(\mathtt{err})]]}{\langle \Sigma, \mathsf{cs}, i_{\mathsf{er}} \rangle^{\top} \rightsquigarrow \langle \Sigma', \mathsf{cs}', j \rangle^{\top}}$$

PROCEDURE CALL

$$\frac{\begin{array}{cccc} \text{cmd}(\mathsf{p}, \mathsf{cs}, i) = x := e(e_i \;|_{i=0}^{n})\; \text{with}\; j & \mathsf{P} = \Sigma.\text{sto} & \mathcal{E}v(e, \mathsf{P}) = f' & \text{args}(f', \mathsf{p}) = [x_1, \ldots, x_m] \\ \mathsf{v}_i = \mathcal{E}v(e_i, \mathsf{P}) \;|_{i=0}^{n} & \mathsf{v}_i = \text{undefined} \;|_{i=n+1}^{m} & \mathsf{P}' = [x_i \mapsto v_i \;|_{i=0}^{m}] & \mathsf{cs}' = (f', \mathsf{P}, x, i{+}1, j) :: \mathsf{cs} \end{array}}{\langle \Sigma, \mathsf{cs}, i \rangle^{\top} \rightsquigarrow \langle \Sigma[\text{sto} \;\mapsto \mathsf{P}'], \mathsf{cs}', 0 \rangle^{\top}}$$

ASSUME

$$\frac{\text{cmd}(\mathsf{p}, \mathsf{cs}, i) = \text{assume}\,(e) \qquad \mathsf{P} = \sigma.\text{sto} \quad \mathcal{E}v(e, \mathsf{P}) = \mathsf{v} \qquad \mathcal{A}sm(\Sigma, \mathsf{v}) = \Sigma'}{\langle \Sigma, \mathsf{cs}, i \rangle^{\top} \rightsquigarrow \langle \Sigma', \mathsf{cs}, i{+}1 \rangle^{\top}}$$

ASSERT - TRUE

$$\frac{\text{cmd}(\mathsf{p}, \mathsf{cs}, i) = \text{assert}\,(e) \qquad \mathsf{P} = \sigma.\text{sto} \quad \mathcal{E}v(e, \mathsf{P}) = \mathsf{v} \qquad \mathcal{S}at(\Sigma, \neg \mathsf{v}) = \text{false}}{\langle \Sigma, \mathsf{cs}, i \rangle^{\top} \rightsquigarrow \langle \Sigma, \mathsf{cs}, i{+}1 \rangle^{\top}}$$

ASSERT - FALSE

$$\frac{\text{cmd}(\mathsf{p}, \mathsf{cs}, i) = \text{assert}\,(e) \qquad \mathsf{P} = \sigma.\text{sto} \quad \mathcal{E}v(e, \mathsf{P}) = \mathsf{v} \qquad \mathcal{S}at(\Sigma, \neg \mathsf{v}) = \text{true}}{\langle \Sigma, \mathsf{cs}, i \rangle^{\top} \rightsquigarrow \langle \Sigma, \mathsf{cs}, i \rangle^{\perp}}$$

Fig. 19. Abstract Execution for Control Flow Commands: $\langle \Sigma, \mathsf{cs}, i \rangle^{\mu} \rightsquigarrow \langle \Sigma', \mathsf{cs}', j \rangle^{\mu'}$

Symbolic Semantics Rules:

GetDomain

$$\frac{\begin{array}{c}\llbracket e \rrbracket_{\hat\rho} = \hat l \quad \hat h = \hat h' \uplus \Big((\hat l, \hat p_i) \mapsto \hat v_i\Big)\,|_{i=0}^{m} \\ (\hat l, -) \notin \mathsf{dom}(\hat h') \quad \pi \vdash \{\, \hat p_1, \ldots, \hat p_m \,\} = \hat d(\hat l) \\ \forall_{0 \le i \le n}\, \hat v_i \neq \varnothing \quad \forall_{n < i \le m}\, \hat v_i = \varnothing\end{array}}{\mathcal{GD}((\hat h, \hat d, \hat\rho, \pi), e) \leadsto \{\, \hat p_1, \ldots, \hat p_n \,\}}$$

GetCell - Domain

$$\frac{\begin{array}{c}\llbracket e_1 \rrbracket_{\hat\rho} = \hat l \quad \llbracket e_2 \rrbracket_{\hat\rho} = \hat p \quad \hat h = \hat h'' \uplus \Big((\hat l, \hat p_i) \mapsto \hat v_i\Big)\,|_{i=0}^{m} \\ (\hat l, -) \notin \mathsf{dom}(\hat h'') \quad \hat d(\hat l) = \hat v_d \quad \hat h' = \hat h[(\hat l, \hat p) \mapsto \varnothing] \\ \hat d' = \hat d[\hat l \mapsto \hat v_d \cup \{\, \hat p \,\}] \quad \pi' = \pi \wedge \hat p \notin (\hat v_d \cup \{\, \hat p_i \,|_{i=0}^{m} \,\})\end{array}}{\mathcal{GC}((\hat h, \hat d, \hat\rho, \pi), (e_1, e_2)) \leadsto (\hat h', \hat d', \hat\rho, \pi'), (\hat l, \hat p, \varnothing)}$$

GetCell - Explicit Cell

$$\frac{\llbracket e_1 \rrbracket_{\hat\rho} = \hat l \quad \llbracket e_2 \rrbracket_{\hat\rho} = \hat p \quad \hat h = - \uplus (\hat l, \hat p') \mapsto \hat v \qquad \hat\sigma' = (\hat h, \hat d, \hat\rho, \pi \wedge (\hat p = \hat p'))}{\mathcal{GC}((\hat h, \hat d, \hat\rho, \pi), (e_1, e_2)) \leadsto \hat\sigma', (\hat l, \hat p, \hat v)}$$

Heap Update

$$\frac{\hat\sigma = (\hat h, \hat d, \hat\rho, \pi) \quad \hat h' = \hat h[(\hat l, \hat p) \mapsto \hat v]}{\mathcal{HU}(\hat\sigma, (\hat l, \hat p), \hat v) = (\hat h', \hat d, \hat\rho, \pi)}$$

Assume

$$\frac{\hat\sigma = (\hat h, \hat d, \hat\rho, \pi)}{\mathcal{Asm}(\hat\sigma, \hat v) = (\hat h, \hat d, \hat\rho, \pi \wedge (\hat v = \mathsf{true}))}$$

Satisfiable - True

$$\frac{\pi = \hat\sigma.\mathsf{pc} \quad (\pi \wedge (\hat v = \mathsf{true}))\ \mathrm{SAT}}{\mathcal{Sat}(\hat\sigma, \hat v) = \mathsf{true}}$$

Satisfiable - False

$$\frac{\pi = \hat\sigma.\mathsf{pc} \quad (\pi \wedge (\hat v = \mathsf{true}))\ \mathrm{UNSAT}}{\mathcal{Sat}(\hat\sigma, \hat v) = \mathsf{false}}$$

Instrumented Semantics Rules:

GetDomain

$$\frac{\begin{array}{c}\llbracket e \rrbracket_{\rho} = l \quad \underline{h} = \underline{h}' \uplus \Big((l, p_i) \mapsto -\Big)\,|_{i=0}^{m} \\ (l, -) \notin \mathsf{dom}(\underline{h}') \quad \{\, p_1, \ldots, p_m \,\} = d(l) \\ \forall_{0 \le i \le n}\, v_i \neq \varnothing \quad \forall_{n < i \le m}\, v_i = \varnothing\end{array}}{\mathcal{GD}((\underline{h}, d, \rho), e) \leadsto \{\, p_1, \ldots, p_n \,\}}$$

GetCell - Domain

$$\frac{\begin{array}{c}\llbracket e_1 \rrbracket_{\rho} = l \quad \llbracket e_2 \rrbracket_{\rho} = p \quad \underline{h} = \underline{h}'' \uplus \Big((l, p_i) \mapsto v_i\Big)\,|_{i=0}^{m} \\ (l, -) \notin \mathsf{dom}(\underline{h}'') \quad d(l) = v_d \quad p \notin v_d \cup \{\, p_i \,|_{i=0}^{m} \,\}) \\ \underline{h}' = \underline{h}[(l, p) \mapsto \varnothing] \quad d' = d[l \mapsto v_d \cup \{\, p \,\}]\end{array}}{\mathcal{GC}((\underline{h}, d, \rho), (e_1, e_2)) \leadsto (\underline{h}', d', \rho), (l, p, \varnothing)}$$

GetCell - Explicit Cell

$$\frac{\llbracket e_1 \rrbracket_{\rho} = l \quad \llbracket e_2 \rrbracket_{\rho} = p \quad \underline{h} = - \uplus (l, p) \mapsto v}{\mathcal{GC}((\underline{h}, d, \rho), (e_1, e_2)) \leadsto (\underline{h}, d, \rho), (l, p, v)}$$

Heap Update

$$\frac{\underline{\sigma} = (\underline{h}, d, \rho) \quad \underline{h}' = \underline{h}[(l, p) \mapsto v]}{\mathcal{HU}(\underline{\sigma}, (l, p), v) = (\underline{h}, d, \rho)}$$

Assume

$$\mathcal{Asm}(\underline{\sigma}, \mathsf{true}) = \underline{\sigma}$$

Satisfiable

$$\mathcal{Sat}(\underline{\sigma}, b) = b$$

Concrete Semantics Rules:

GetDomain

$$\frac{\llbracket e \rrbracket_{\rho} = l \quad h = h' \uplus \Big((l, p_i) \mapsto v_i\Big)\,|_{i=0}^{m} \quad (l, -) \notin \mathsf{dom}(h')}{\mathcal{GD}((h, \rho), e) \leadsto \{\, p_1, \ldots, p_n \,\}}$$

GetCell - Explicit Cell

$$\frac{\llbracket e_1 \rrbracket_{\rho} = l \quad \llbracket e_2 \rrbracket_{\rho} = p \quad h = - \uplus (l, p) \mapsto v}{\mathcal{GC}((h, \rho), (e_1, e_2)) \leadsto (h, \rho), (l, p, v)}$$

Heap Update - Non-∅

$$\frac{\sigma = (h, \rho) \quad h' = h[(l, p) \mapsto v] \quad v \neq \varnothing}{\mathcal{HU}(\sigma, (l, p), v) = (h', \rho)}$$

Heap Update - ∅

$$\frac{\sigma = (h, \rho) \quad h = h' \uplus (l, p) \mapsto -}{\mathcal{HU}(\sigma, (l, p), \varnothing) = (h', \rho)}$$

Assume

$$\mathcal{Asm}(\sigma, \mathsf{true}) = (\sigma, \mathsf{true})$$

Satisfiable

$$\mathcal{Sat}(\sigma, b) = b$$

B OLD

Evaluation of extended JSIL expressions

$$\begin{array}{c}\llbracket v \rrbracket_{\hat\rho} \triangleq v \\ \llbracket x \rrbracket_{\hat\rho} \triangleq \hat\rho(x) \\ \llbracket \hat x \rrbracket_{\hat\rho} \triangleq \hat x\end{array} \qquad \frac{\llbracket \hat{\mathrm{e}} \rrbracket_{\hat\rho} = v}{\llbracket \ominus \hat{\mathrm{e}} \rrbracket_{\hat\rho} \triangleq \overline{\ominus} v} \qquad \frac{\llbracket \hat{\mathrm{e}} \rrbracket_{\hat\rho} = \hat e \notin \mathcal{V}_{\mathrm{JSIL}}}{\llbracket \ominus \hat{\mathrm{e}} \rrbracket_{\hat\rho} \triangleq \ominus \hat e} \qquad \frac{v = \overline{\oplus}(\llbracket \hat{\mathrm{e}}_1 \rrbracket_{\hat\rho}, \llbracket \hat{\mathrm{e}}_2 \rrbracket_{\hat\rho})}{\llbracket \hat{\mathrm{e}}_1 \oplus \hat{\mathrm{e}}_2 \rrbracket_{\hat\rho} \triangleq v} \qquad \frac{\begin{array}{c}\llbracket \hat{\mathrm{e}}_1 \rrbracket_{\hat\rho} = \hat e_1 \quad \llbracket \hat{\mathrm{e}}_2 \rrbracket_{\hat\rho} = \hat e_2 \\ \hat e_1 \notin \mathcal{V}_{\mathrm{JSIL}} \vee \hat e_2 \notin \mathcal{V}_{\mathrm{JSIL}}\end{array}}{\llbracket \hat{\mathrm{e}}_1 \oplus \hat{\mathrm{e}}_2 \rrbracket_{\hat\rho} \triangleq \hat e_1 \oplus \hat e_2}$$

Skip

$$\langle \hat h, \hat\rho, \mathsf{skip}, \pi \rangle^{\top} \leadsto \langle \hat h, \hat\rho, \pi \rangle^{\top}$$

Object Creation

$$\frac{(l, -) \notin \mathsf{dom}(\hat h) \qquad \hat h' = \hat h \uplus (l, @\mathit{proto}) \mapsto \mathsf{null}}{\langle \hat h, \hat\rho, x := \mathsf{new}\,(), \pi \rangle^{\top} \leadsto \langle \hat h', \hat\rho[x \mapsto l], \pi \rangle^{\top}}$$

Assignment

$$\frac{\llbracket \hat{\mathrm{e}} \rrbracket_{\hat\rho} = \hat e \quad \hat\rho' = \hat\rho[x \mapsto \hat e]}{\langle \hat h, \hat\rho, x := \hat{\mathrm{e}}, \pi \rangle^{\top} \leadsto \langle \hat h, \hat\rho', \pi \rangle^{\top}}$$

Property Collection

$$\frac{\llbracket \hat{\mathrm{e}} \rrbracket_{\hat\rho} = l \quad \hat h = \hat h' \uplus \Big((l, \hat p_i) \mapsto -\Big)\,|_{i=0}^{n} \quad (l, -) \notin \mathsf{dom}(\hat h')}{\langle h, \rho, x := \mathsf{getFields}\,(\hat{\mathrm{e}}), \pi \rangle^{\top} \rightarrow \langle h, \rho[x \mapsto \{\, \hat p_0, \ldots, \hat p_n \,\}], \pi \rangle^{\top}}$$

Property Access

$$\frac{\begin{array}{c}\llbracket \hat{\mathrm{e}}_1 \rrbracket_{\hat\rho} = l \quad \llbracket \hat{\mathrm{e}}_2 \rrbracket_{\hat\rho} = \hat e_p \quad \hat h = \hat h' \uplus \Big((l, \hat p_i) \mapsto \hat v_i\Big)\,|_{i=0}^{n} \quad (l, -) \notin \mathsf{dom}(\hat h') \quad 0 \le k \le n \\ \pi' = \pi \ \wedge \Big((\hat p_k = \hat e_p) \ \wedge \Big(\wedge_{i=0,\, i \neq k}^{n} (\hat p_i \neq \hat e_p)\Big)\Big)\end{array}}{\langle \hat h, \hat\rho, x := [\hat{\mathrm{e}}_1, \hat{\mathrm{e}}_2], \pi \rangle^{\top} \leadsto \langle \hat h, \hat\rho[x \mapsto \hat v_k], \pi' \rangle^{\top}}$$

Property Assignment - Found

$$\frac{\begin{array}{c}\llbracket \hat{\mathrm{e}}_1 \rrbracket_{\hat\rho} = l \quad \llbracket \hat{\mathrm{e}}_2 \rrbracket_{\hat\rho} = \hat e_p \quad \llbracket \hat{\mathrm{e}}_3 \rrbracket_{\hat\rho} = \hat e_v \quad \hat h = \hat h' \uplus \Big((l, \hat p_i) \mapsto \hat v_i\Big)\,|_{i=0}^{n} \quad (l, -) \notin \mathsf{dom}(\hat h') \quad 0 \le k \le n \\ \pi' = \pi \ \wedge \Big((\hat p_k = \hat e_p) \ \wedge \Big(\wedge_{i=0,\, i \neq k}^{n} (\hat p_i \neq \hat e_p)\Big)\Big) \quad \hat h'' = \hat h' \uplus \Big((l, \hat p_i) \mapsto \hat v_i\Big)\,|_{i=0,\, i \neq k}^{n} \uplus (l, \hat e_p) \mapsto \hat e_v\end{array}}{\langle \hat h, \hat\rho, [\hat{\mathrm{e}}_1, \hat{\mathrm{e}}_2] := \hat{\mathrm{e}}_3, \pi \rangle^{\top} \leadsto \langle \hat h'', \hat\rho, \pi' \rangle^{\top}}$$

Property Assignment - Not Found

$$\textsc{Basic Command}\quad \frac{\mathsf{cmd}(i) = bc \quad \mathsf{p} : \langle \hat{h}, \hat{\rho}, bc, \pi\rangle^{\mu} \rightsquigarrow \langle \hat{h}', \hat{\rho}', \pi'\rangle^{\mu'}}{\mathsf{p} : \langle \hat{h}, \hat{\rho}, i, \pi\rangle^{\mu} \rightsquigarrow \langle \hat{h}', \hat{\rho}', i+1, \pi'\rangle^{\mu'}}$$

$$\textsc{Goto}\quad \frac{\mathsf{cmd}(i) = \mathrm{goto}\ j}{\mathsf{p} : \langle \hat{h}, \hat{\rho}, i, \pi\rangle^{\top} \rightsquigarrow \langle \hat{h}, \hat{\rho}, j, \pi\rangle^{\top}}$$

$$\textsc{Cond. Goto - True}\quad \frac{\mathsf{cmd}(i) = \mathrm{goto}\ [\hat{\mathsf{e}}]\ j,\ k \quad [\![\hat{\mathsf{e}}]\!]_{\hat{\rho}} = \hat{e}}{\mathsf{p} : \langle \hat{h}, \hat{\rho}, i, \pi\rangle^{\top} \rightsquigarrow \langle \hat{h}, \hat{\rho}, j, \pi \wedge \hat{e}\rangle^{\top}}$$

$$\textsc{Cond. Goto - False}\quad \frac{\mathsf{cmd}(i) = \mathrm{goto}\ [\hat{\mathsf{e}}]\ j,\ k \quad [\![\hat{\mathsf{e}}]\!]_{\hat{\rho}} = \hat{e}}{\mathsf{p} : \langle \hat{h}, \hat{\rho}, i, \pi\rangle^{\top} \rightsquigarrow \langle \hat{h}, \hat{\rho}, k, \pi \wedge \neg\hat{e}\rangle^{\top}}$$

$$\textsc{Procedure Call}\quad \frac{\mathsf{cmd}(i) = x := \hat{\mathsf{e}}(\hat{\mathsf{e}}_i\ |_{i=0}^{n})\ \mathrm{with}\ j \quad [\![\hat{\mathsf{e}}]\!]_{\hat{\rho}} = f' \quad [\![\hat{\mathsf{e}}_i]\!]_{\hat{\rho}} = \hat{e}_i\ |_{i=0}^{n} \quad \mathsf{args}(f') = [x_1, \ldots, x_m] \quad \hat{e}_i = \mathsf{undefined}\ |_{i=n+1}^{m}}{\mathsf{p} : \langle \hat{h}, \hat{\rho}, i, \pi\rangle^{\top}_{\hat{C}} \rightsquigarrow \langle \hat{h}, [x_i \mapsto \hat{e}_i\ |_{i=0}^{m}], 0, \pi\rangle^{\top}_{(f', \hat{\rho}, x, i+1, j)::\hat{C}}}$$

$$\textsc{Normal Return}\quad \frac{\hat{C} = (-, \hat{\rho}', x, i, -) :: \hat{C}' \quad \hat{\rho}(\mathsf{ret}) = \hat{e}}{\mathsf{p} : \langle \hat{h}, \hat{\rho}, i_{\mathrm{nm}}, \pi\rangle^{\top}_{\hat{C}} \rightsquigarrow \langle \hat{h}, \hat{\rho}'[x \mapsto \hat{e}], i, \pi\rangle^{\top}_{\hat{C}'}}$$

$$\textsc{Error Return}\quad \frac{\hat{C} = (-, \hat{\rho}', x, -, j) :: \hat{C}' \quad \hat{\rho}(\mathsf{err}) = \hat{e}}{\mathsf{p} : \langle \hat{h}, \hat{\rho}, i_{\mathrm{er}}, \pi\rangle^{\top}_{\hat{C}} \rightsquigarrow \langle \hat{h}, \hat{\rho}'[x \mapsto \hat{e}], j, \pi\rangle^{\top}_{\hat{C}'}}$$

Fig. 20. Symbolic Semantics for JSIL Commands: $\mathsf{p} : \langle \hat{h}, \hat{\rho}, i, \pi\rangle^{\mu}_{\hat{C}} \rightsquigarrow \langle \hat{h}', \hat{\rho}', j, \pi'\rangle^{\mu'}_{\hat{C}'}$

Interpretation of Symbolic Expressions

$$\llbracket v \rrbracket_\varepsilon \triangleq v \qquad \llbracket \hat{x} \rrbracket_\varepsilon \triangleq \varepsilon(\hat{x}) \qquad \llbracket \ominus \hat{e} \rrbracket_\varepsilon \triangleq \overline{\ominus}(\llbracket \hat{e} \rrbracket_\varepsilon) \qquad \llbracket \hat{e}_1 \oplus \hat{e}_2 \rrbracket_\varepsilon \triangleq \overline{\oplus}(\llbracket \hat{e}_1 \rrbracket_\varepsilon, \llbracket \hat{e}_2 \rrbracket_\varepsilon)$$

Interpretation of Extended Expressions

$$\begin{array}{c} \llbracket v \rrbracket_\varepsilon \triangleq v \\ \llbracket x \rrbracket_\varepsilon \triangleq x \\ \llbracket \hat{x} \rrbracket_\varepsilon \triangleq \varepsilon(\hat{x}) \end{array} \qquad \frac{\llbracket \hat{\mathrm{e}} \rrbracket_\varepsilon = v}{\llbracket \ominus \hat{\mathrm{e}} \rrbracket_\varepsilon \triangleq \overline{\ominus} v} \qquad \frac{\llbracket \hat{\mathrm{e}} \rrbracket_\varepsilon = \hat{\mathrm{e}}' \notin \mathcal{V}_{\mathsf{JSIL}}}{\llbracket \ominus \hat{\mathrm{e}} \rrbracket_\varepsilon \triangleq \ominus \hat{\mathrm{e}}'} \qquad \frac{v = \overline{\oplus}(\llbracket \hat{\mathrm{e}}_1 \rrbracket_\varepsilon, \llbracket \hat{\mathrm{e}}_2 \rrbracket_{\hat{\rho}})}{\llbracket \hat{\mathrm{e}}_1 \oplus \hat{\mathrm{e}}_2 \rrbracket_\varepsilon \triangleq v} \qquad \frac{\llbracket \hat{\mathrm{e}}_1 \rrbracket_\varepsilon = \hat{\mathrm{e}}'_1 \quad \llbracket \hat{\mathrm{e}}_2 \rrbracket_\varepsilon = \hat{\mathrm{e}}'_2 \quad \hat{\mathrm{e}}'_1 \notin \mathcal{V}_{\mathsf{JSIL}} \lor \hat{\mathrm{e}}'_2 \notin \mathcal{V}_{\mathsf{JSIL}}}{\llbracket \hat{\mathrm{e}}_1 \oplus \hat{\mathrm{e}}_2 \rrbracket_\varepsilon \triangleq \hat{\mathrm{e}}'_1 \oplus \hat{\mathrm{e}}'_2}$$

Interpretation of Symbolic Heaps

$$\llbracket \emptyset \rrbracket_\varepsilon \triangleq \emptyset \qquad \llbracket (l, \hat{e}_p) \mapsto \hat{e}_v \rrbracket_\varepsilon \triangleq (l, \llbracket \hat{e}_p \rrbracket_\varepsilon) \mapsto \llbracket \hat{e}_v \rrbracket_\varepsilon \qquad \llbracket \hat{h}_1 \uplus \hat{h}_2 \rrbracket_\varepsilon \triangleq \llbracket \hat{h}_1 \rrbracket_\varepsilon \uplus \llbracket \hat{h}_2 \rrbracket_\varepsilon$$

Interpretation of Symbolic Stores

$$\llbracket \emptyset \rrbracket_\varepsilon \triangleq \emptyset \qquad \llbracket (x : \hat{e}) \uplus \hat{\rho} \rrbracket_\varepsilon \triangleq (x : \llbracket \hat{e} \rrbracket_\varepsilon) \uplus \llbracket \hat{\rho} \rrbracket_\varepsilon$$

Interpretation of Symbolic Contexts

$$\llbracket [\,] \rrbracket_\varepsilon \triangleq [\,] \qquad \llbracket (f, \hat{\rho}, x, i, j) :: \hat{C} \rrbracket_\varepsilon \triangleq (f, \llbracket \hat{\rho} \rrbracket_\varepsilon, x, i, j) :: \llbracket \hat{C} \rrbracket_\varepsilon$$

Interpretation of Basic Commands

$$\llbracket \mathsf{skip} \rrbracket_\varepsilon = \mathsf{skip} \qquad \llbracket x := \hat{\mathrm{e}} \rrbracket_\varepsilon = x := \llbracket \hat{\mathrm{e}} \rrbracket_\varepsilon \qquad \llbracket x := \mathsf{new}\,() \rrbracket_\varepsilon = x := \mathsf{new}\,()$$

$$\llbracket x := [\hat{\mathrm{e}}_1, \hat{\mathrm{e}}_2] \rrbracket_\varepsilon = x := [\llbracket \hat{\mathrm{e}}_1 \rrbracket_\varepsilon, \llbracket \hat{\mathrm{e}}_2 \rrbracket_\varepsilon] \qquad \llbracket [\hat{\mathrm{e}}_1, \hat{\mathrm{e}}_2] := \hat{\mathrm{e}}_3 \rrbracket_\varepsilon = [\llbracket \hat{\mathrm{e}}_1 \rrbracket_\varepsilon, \llbracket \hat{\mathrm{e}}_2 \rrbracket_\varepsilon] := \llbracket \hat{\mathrm{e}}_3 \rrbracket_\varepsilon$$

$$\llbracket \mathsf{delete}\,(\hat{\mathrm{e}}_1, \hat{\mathrm{e}}_2) \rrbracket_\varepsilon = \mathsf{delete}\,(\llbracket \hat{\mathrm{e}}_1 \rrbracket_\varepsilon, \llbracket \hat{\mathrm{e}}_2 \rrbracket_\varepsilon) \qquad \llbracket x := \mathsf{hasField}\,(\hat{\mathrm{e}}_1, \hat{\mathrm{e}}_2) \rrbracket_\varepsilon = x := \mathsf{hasField}\,(\llbracket \hat{\mathrm{e}}_1 \rrbracket_\varepsilon, \llbracket \hat{\mathrm{e}}_2 \rrbracket_\varepsilon)$$

$$\llbracket x := \mathsf{getFields}\,(\hat{\mathrm{e}}) \rrbracket_\varepsilon = x := \mathsf{getFields}\,(\llbracket \hat{\mathrm{e}} \rrbracket_\varepsilon)$$

$$\llbracket \mathsf{assume}\,(\hat{\mathrm{e}}) \rrbracket_\varepsilon = \mathsf{assume}\,(\llbracket \hat{\mathrm{e}} \rrbracket_\varepsilon) \qquad \llbracket \mathsf{assert}\,(\hat{\mathrm{e}}) \rrbracket_\varepsilon = \mathsf{assert}\,(\llbracket \hat{\mathrm{e}} \rrbracket_\varepsilon)$$

Interpretation of Commands

$$\llbracket bc \rrbracket_\varepsilon = \llbracket bc \rrbracket_\varepsilon \qquad \llbracket \mathsf{goto}\ i \rrbracket_\varepsilon = \mathsf{goto}\ i \qquad \llbracket \mathsf{goto}\ [e]\ i,\ j \rrbracket_\varepsilon = \mathsf{goto}\ [\llbracket e \rrbracket_\varepsilon]\ i,\ j$$

$$\llbracket x := e(\overline{e})\ \mathsf{with}\ j \rrbracket_\varepsilon = x := \llbracket e \rrbracket_\varepsilon(\llbracket \overline{e} \rrbracket_\varepsilon)\ \mathsf{with}\ j$$

Interpretation of Procedures: $\llbracket \mathsf{proc}\ \mathit{fid}(\overline{x})\{\overline{\mathsf{c}}\} \rrbracket_\varepsilon = \mathsf{proc}\ \mathit{fid}(\overline{x})\{\llbracket \overline{\mathsf{c}} \rrbracket_\varepsilon\}$

Interpretation of Programs: $\llbracket \mathsf{p} \rrbracket_\varepsilon = \{\ (\mathit{fid}, \llbracket \mathsf{p}(\mathit{fid}) \rrbracket_\varepsilon)\ \mid\ \mathit{fid} \in \mathsf{dom}(\mathsf{p})\ \}$

Fig. 21. Interpretation of extended expressions, symbolic heaps, symbolic stores, etc.

C PROOFS - SECTION 2

Lemma C.1. *Some Properties of Interpretation and Evaluation.*

(1) $\forall\, \hat{\mathrm{e}}, \varepsilon.\ \mathsf{svars}(\hat{\mathrm{e}}) \subseteq \mathsf{dom}(\varepsilon) \implies \llbracket \hat{\mathrm{e}} \rrbracket_\varepsilon \in \mathcal{E}_{\mathsf{JSIL}}$.
(2) $\forall\, \hat{\mathrm{e}}, \hat{\rho}, \varepsilon.\ \llbracket \llbracket \hat{\mathrm{e}} \rrbracket_\varepsilon \rrbracket_{\llbracket \hat{\rho} \rrbracket_\varepsilon} = \llbracket \llbracket \hat{\mathrm{e}} \rrbracket_{\hat{\rho}} \rrbracket_\varepsilon$.

Proof. By induction on the structure of the appropriate syntactic category. □

Lemma C.2 (Transparency for single-step executions). $\forall\, h, \rho, i, C, \mu, h', \rho', i', C', \mu'$.

$$\mathsf{p} : \langle h, \rho, i \rangle_C^\mu \rightarrow \langle h', \rho', i' \rangle_{C'}^{\mu'} \iff \mathsf{p} : \langle h, \rho, i, \mathsf{true} \rangle_C^\mu \rightsquigarrow \langle h', \rho', i', \mathsf{true} \rangle_{C'}^{\mu'}$$

Proof. Given the rules of the concrete and symbolic JSIL semantics, we have that $\mu = \top$. The interesting case is when $\mu' = \bot$. Then, it must be that the rule applied on both sides of the equivalence is the [Basic Command] rule. Moreover, the associated basic command rule must be [Assert - False]. This means that, in the concrete semantics, we should have:

$$\textsc{Assert - False}\quad \frac{\llbracket e \rrbracket_\rho = \mathsf{false}}{\langle h, \rho, \mathsf{assert}\,(e) \rangle^\top \rightarrow \langle h, \rho \rangle^\bot}$$

whereas in the symbolic semantics, we should have:

$$\textsc{Assert - False}\quad \frac{(\mathsf{true}\ \land\ \neg[\![e]\!]_\rho)\ \mathsf{satisfiable}}{\langle h, \rho, \mathsf{assert}\ (e), \mathsf{true}\rangle^\top \rightsquigarrow \langle h, \rho, \mathsf{true} \land \neg[\![e]\!]_\rho\rangle^\bot}$$

for some concrete expression e. From this, in the left-to-right case, we observe that $[\![e]\!]_\rho = \mathsf{false}$, that is, that its negation is $\neg[\![e]\!]_\rho = \mathsf{true}$ and the symbolic [Assert - False] rule can be applied to obtain the desired goal. In the right-to-left case, since $\mathsf{true} \land \neg[\![e]\!]_\rho$ is satisfiable and e is concrete, it must be that $\neg[\![e]\!]_\rho = \mathsf{true}$, that is, that $[\![e]\!]_\rho = \mathsf{false}$. Given that, we can apply the concrete [Assert - False] rule to obtain the desired goal.

All remaining cases, for which $\mu' = \top$, follow directly from the definitions of the concrete and symbolic semantics and the fact that the heaps, stores, and call stacks are concrete. □

Lemma C.3 (Transparency for fixed-length executions). $\forall\, k, h, \rho, i, C, \mu, h', \rho', i', C', \mu'\ .$

$$\mathsf{p} : \langle h, \rho, i\rangle^\mu_C \rightarrow^k \langle h', \rho', i'\rangle^{\mu'}_{C'} \iff \mathsf{p} : \langle h, \rho, i, \mathsf{true}\rangle^\mu_C \rightsquigarrow^k \langle h', \rho', i', \mathsf{true}\rangle^{\mu'}_{C'}$$

Proof. By induction on k. The base case, when $k = 0$, amounts to reflexivity. Let us assume that $k = n$ and prove the claim for $k = n + 1$. We have the inductive hypothesis:

IH: $\forall\, h, \rho, i, C, \mu, h', \rho', i', C', \mu'\ .$
$\mathsf{p} : \langle h, \rho, i\rangle^\mu_C \rightarrow^n \langle h', \rho', i'\rangle^{\mu'}_{C'} \iff \mathsf{p} : \langle h, \rho, i, \mathsf{true}\rangle^\mu_C \rightsquigarrow^n \langle h', \rho', i', \mathsf{true}\rangle^{\mu'}_{C'}.$

Let us prove the left-to-right direction for $k = n + 1$. By definition of $\rightarrow^k$, we have that

$$\exists\, h'', \rho'', i'', C'', \mu''.\ \mathsf{p} : \langle h, \rho, i\rangle^\mu_C \rightarrow^n \langle h'', \rho'', i''\rangle^{\mu''}_{C''}\ \land\ \mathsf{p} : \langle h'', \rho'', i''\rangle^{\mu''}_{C''} \rightarrow \langle h', \rho', i'\rangle^{\mu'}_{C'}.$$

From this, by the IH, we obtain that $\mathsf{p} : \langle h, \rho, i, \mathsf{true}\rangle^\mu_C \rightsquigarrow^n \langle h'', \rho'', i'', \mathsf{true}\rangle^{\mu'}_{C''}$. Also, from Lemma C.2, we obtain that $\mathsf{p} : \langle h'', \rho'', i', \mathsf{true}\rangle^{\mu''}_{C''} \rightsquigarrow \langle h', \rho', i', \mathsf{true}\rangle^{\mu'}_{C'}$, concluding the proof.

The right-to-left direction is proven analogously. □

1. *Theorem 2.1 - Transparency.* $\forall\, h, \rho, i, C, \mu, h', \rho', i', C', \mu'\ .$

$$\mathsf{p} : \langle h, \rho, i\rangle^\mu_C \rightarrow^* \langle h', \rho', i'\rangle^{\mu'}_{C'} \iff \mathsf{p} : \langle h, \rho, i, \mathsf{true}\rangle^\mu_C \rightsquigarrow^* \langle h', \rho', i', \mathsf{true}\rangle^{\mu'}_{C'}$$

Proof. Directly from Lemma C.3 and the definitions of $\rightarrow^*$ and $\rightsquigarrow^*$. □

Lemma C.4 (Soundness of symbolic execution for JSIL basic commands).

$$\begin{array}{l} \forall \hat{h}, \hat{\rho}, bc, \pi, \mu, \hat{h}', \hat{\rho}', \pi', \mu', h, \rho, \varepsilon, \pi''\ . \\ \quad \langle \hat{h}, \hat{\rho}, bc, \pi\rangle^\mu \rightsquigarrow \langle \hat{h}', \hat{\rho}', \pi'\rangle^{\mu'}\ \land\ \pi'' \vdash \pi'\ \land\ (h, \rho, \varepsilon) \in \mathcal{M}_{\pi''}(\hat{h}, \hat{\rho})\ \land\ \mathsf{svars}(bc) \subseteq \mathsf{dom}(\varepsilon) \\ \qquad \implies\ \exists h', \rho'\ .\ \langle h, \rho, [\![bc]\!]_\varepsilon\rangle^\mu \rightarrow \langle h', \rho'\rangle^{\mu'}\ \land\ (h', \rho', \varepsilon) \in \mathcal{M}_{\pi''}(\hat{h}', \hat{\rho}') \end{array}$$

Proof. We proceed by case analysis on $\langle \hat{h}, \hat{\rho}, bc, \pi\rangle^\mu \rightsquigarrow \langle \hat{h}', \hat{\rho}', \pi'\rangle^{\mu'}$. Given the symbolic semantics for JSIL basic commands, we obtain that $\mu = \top$.

We first prove the cases for which $\mu' = \bot$. The only applicable rule is [Assert - False]. We conclude that $bc = \mathsf{assert}\ (\hat{e})$ for some extended expression $\hat{e}$, and that $\pi\ \land\ \neg[\![\hat{e}]\!]_{\hat{\rho}}$ is satisfiable and $\pi' = \pi\ \land\ \neg[\![\hat{e}]\!]_{\hat{\rho}}$. From $(h, \rho, \varepsilon) \in \mathcal{M}_{\pi''}(\hat{h}, \hat{\rho})$, we conclude that $h = [\![\hat{h}]\!]_\varepsilon$, $\rho = [\![\hat{\rho}]\!]_\varepsilon$, and $\varepsilon \vdash \pi''$. Since $\pi'' \vdash \pi'$, we conclude that $\pi'' \vdash \neg[\![e]\!]_{\hat{\rho}}$. We let $e = [\![\hat{e}]\!]_\varepsilon$ and note that $[\![bc]\!]_\varepsilon = x := \mathsf{assert}\ (e)$. We now have to prove that we can apply the [Assert - False] rule in the concrete state. To this end, we have to show that: $[\![e]\!]_\rho = \mathsf{false}$. Noting that $[\![e]\!]_\rho = [\![e]\!]_{[\![\hat{\rho}]\!]_\varepsilon} = [\![[\![\hat{e}]\!]_\varepsilon]\!]_{[\![\hat{\rho}]\!]_\varepsilon} = [\![[\![\hat{e}]\!]_{\hat{\rho}}]\!]_\varepsilon$, we conclude (using $\varepsilon \vdash \pi''$ and $\pi'' \vdash \neg[\![\hat{e}]\!]_{\hat{\rho}}$) that $[\![e]\!]_\rho = \mathsf{false}$, from which we obtain that $h' = h$ and $\rho' = \rho$, and the result follows.

Onward, we assume that $\mu' = \top$ and elide the modes from the rules. Throughout the proof, we use the fact that $\llbracket bc \rrbracket_\varepsilon$ is concrete for all ε, such that $\mathsf{svars}(bc) \subseteq \mathsf{dom}(\varepsilon)$.

[Skip] We conclude that $bc = \mathsf{skip}$, and that $\hat{h}' = \hat{h}$, $\hat{\rho}' = \hat{\rho}$, and $\pi' = \pi$. By picking $h' = h$, $\rho' = \rho$, the result follows.

[Assignment] We conclude that $bc = x := \hat{\mathrm{e}}$, for some variable x and extended expression $\hat{\mathrm{e}}$, and that $\hat{h}' = \hat{h}$, $\hat{\rho}' = \hat{\rho}[x \mapsto \llbracket \hat{\mathrm{e}} \rrbracket_{\hat{\rho}}]$, and $\pi' = \pi$. From $(h, \rho, \varepsilon) \in \mathcal{M}_{\pi''}(\hat{h}, \hat{\rho})$, we conclude that $\varepsilon \models \pi''$, $h = \llbracket \hat{h} \rrbracket_\varepsilon$ and $\rho = \llbracket \hat{\rho} \rrbracket_\varepsilon$. Now, letting $e = \llbracket \hat{\mathrm{e}} \rrbracket_\varepsilon$, noting that $\llbracket bc \rrbracket_\varepsilon = x := e$, and noting that

$$\langle h, \rho, x := e \rangle \rightarrow \langle h, \rho[x \mapsto \llbracket e \rrbracket_\rho] \rangle$$

we pick $h' = h$ and $\rho' = \rho[x \mapsto \llbracket e \rrbracket_\rho]$. We now have to prove that $(h', \rho', \varepsilon) \in \mathcal{M}_{\pi''}(\hat{h}', \hat{\rho}')$. Observing that:

$$h' = \llbracket \hat{h} \rrbracket_\varepsilon = \llbracket \hat{h}' \rrbracket_\varepsilon \quad \rho' = \llbracket \hat{\rho} \rrbracket_\varepsilon [x \mapsto \llbracket e \rrbracket_{\llbracket \hat{\rho} \rrbracket_\varepsilon}] = \llbracket \hat{\rho}[x \mapsto \llbracket \hat{e} \rrbracket_\rho] \rrbracket_\varepsilon = \llbracket \hat{\rho}' \rrbracket_\varepsilon$$

the result follows.

[Object Creation] We conclude that $bc = x := \mathsf{new}\,()$, for some variable x, and that $\hat{h}' = \hat{h} \uplus (l, @proto) \mapsto \mathsf{null}$, $\hat{\rho}' = \hat{\rho}[x \mapsto l]$, and $\pi' = \pi$, where $(l, -) \notin \mathsf{dom}(\hat{h})$. From $(h, \rho, \varepsilon) \in \mathcal{M}_{\pi''}(\hat{h}, \hat{\rho})$, we conclude that $\varepsilon \vdash \pi''$, $h = \llbracket \hat{h} \rrbracket_\varepsilon$ and $\rho = \llbracket \hat{\rho} \rrbracket_\varepsilon$. Noting that $\llbracket bc \rrbracket_\varepsilon = bc$, and also that

$$\langle h, \rho, x := \mathsf{new}\,() \rangle \rightarrow \langle h \uplus (l, @proto) \mapsto \mathsf{null}, \rho[x \mapsto l] \rangle$$

where: $(l, -) \notin \mathsf{dom}(h)$, we pick $h' = \llbracket \hat{h} \rrbracket_\varepsilon \uplus (l, @proto) \mapsto \mathsf{null}$ and $\rho' = \llbracket \hat{\rho} \rrbracket_\varepsilon [x \mapsto l]$. We now have to prove that $(h', \rho', \varepsilon) \in \mathcal{M}_{\pi''}(\hat{h}', \hat{\rho}')$. Noting that:

$$\begin{aligned} h' &= \llbracket \hat{h} \rrbracket_\varepsilon \uplus (l, @proto) \mapsto \mathsf{null} = \llbracket \hat{h} \uplus (l, @proto) \mapsto \mathsf{null} \rrbracket_\varepsilon = \llbracket \hat{h}' \rrbracket_\varepsilon \\ \rho' &= \llbracket \hat{\rho} \rrbracket_\varepsilon [x \mapsto l] = \llbracket \hat{\rho} \rrbracket_\varepsilon [x \mapsto \llbracket l \rrbracket_\varepsilon] = \llbracket \hat{\rho}[x \mapsto l] \rrbracket_\varepsilon = \llbracket \hat{\rho}' \rrbracket_\varepsilon \end{aligned}$$

the result follows.

[Property Access] We conclude that $bc = x := [\hat{\mathrm{e}}_1, \hat{\mathrm{e}}_2]$, for some variable x, and extended expressions $\hat{\mathrm{e}}_1$ and $\hat{\mathrm{e}}_2$, and that $\hat{h}' = \hat{h}$, $\hat{\rho}' = \hat{\rho}[x \mapsto \hat{v}_k]$, and:

$$\pi' = \pi \ \wedge \ \Big((\hat{p}_k = \hat{e}_p) \ \wedge \bigwedge_{i=0,\, i \neq k}^{n} (\hat{p}_i \neq \hat{e}_p) \Big)$$

where: $\llbracket \hat{\mathrm{e}}_1 \rrbracket_{\hat{\rho}} = l$, $\llbracket \hat{\mathrm{e}}_2 \rrbracket_{\hat{\rho}} = \hat{e}_p$, $\hat{h} = \hat{h}'' \uplus \Big((l, \hat{p}_i) \mapsto \hat{v}_i \Big) |_{i=0}^{n}$, $(l, -) \notin \mathsf{dom}(\hat{h}')$, and $0 \leq k \leq n$. From $(h, \rho, \varepsilon) \in \mathcal{M}_{\pi''}(\hat{h}, \hat{\rho})$, we conclude that $h = \llbracket \hat{h} \rrbracket_\varepsilon$, $\rho = \llbracket \hat{\rho} \rrbracket_\varepsilon$, and $\varepsilon \vdash \pi''$. We let $e_1 = \llbracket \hat{\mathrm{e}}_1 \rrbracket_\varepsilon$ and $e_2 = \llbracket \hat{\mathrm{e}}_2 \rrbracket_\varepsilon$, and note that $\llbracket bc \rrbracket_\varepsilon = x := [e_1, e_2]$. We now have to prove that we can apply the [Property Access] rule in the concrete state. To this end, we have to show that there is a concrete heap h'' such that: $h = h'' \uplus (\llbracket e_1 \rrbracket_\rho, \llbracket e_2 \rrbracket_\rho) \mapsto \llbracket \hat{v}_k \rrbracket_\varepsilon$. Note that:

$$\begin{aligned} & \varepsilon \vdash \pi'' \ \wedge \ \pi'' \vdash \pi' \ \wedge \ \pi' \vdash \hat{p}_k = \hat{e}_p \implies \varepsilon \vdash \hat{p}_k = \hat{e}_p \\ & \llbracket e_1 \rrbracket_\rho = \llbracket e_1 \rrbracket_{\llbracket \hat{\rho} \rrbracket_\varepsilon} = \llbracket \llbracket \hat{\mathrm{e}}_1 \rrbracket_\varepsilon \rrbracket_{\llbracket \hat{\rho} \rrbracket_\varepsilon} = \llbracket \llbracket \hat{\mathrm{e}}_1 \rrbracket_{\hat{\rho}} \rrbracket_\varepsilon = \llbracket l \rrbracket_\varepsilon = l \\ & \llbracket e_2 \rrbracket_\rho = \llbracket e_2 \rrbracket_{\llbracket \hat{\rho} \rrbracket_\varepsilon} = \llbracket \llbracket \hat{\mathrm{e}}_2 \rrbracket_\varepsilon \rrbracket_{\llbracket \hat{\rho} \rrbracket_\varepsilon} = \llbracket \llbracket \hat{\mathrm{e}}_2 \rrbracket_{\hat{\rho}} \rrbracket_\varepsilon = \llbracket \hat{e}_p \rrbracket_\varepsilon = \llbracket \hat{p}_k \rrbracket_\varepsilon \text{ (because } \varepsilon \vdash \hat{p}_k = \hat{e}_p) \\ & h = \llbracket \hat{h}'' \uplus \Big((l, \hat{p}_i) \mapsto \hat{v}_i \Big) |_{i=0}^{n} \rrbracket_\varepsilon = \llbracket \hat{h}'' \uplus \Big((l, \hat{p}_i) \mapsto \hat{v}_i \Big) |_{i=0,\, i \neq k}^{n} \rrbracket_\varepsilon \uplus \llbracket (l, \hat{p}_k) \mapsto \hat{v}_k \rrbracket_\varepsilon \\ & \quad = \llbracket \hat{h}'' \uplus \Big((l, \hat{p}_i) \mapsto \hat{v}_i \Big) |_{i=0,\, i \neq k}^{n} \rrbracket_\varepsilon \uplus (l, \llbracket \hat{p}_k \rrbracket_\varepsilon) \mapsto \llbracket \hat{v}_k \rrbracket_\varepsilon \\ & \quad = \llbracket \hat{h}'' \uplus \Big((l, \hat{p}_i) \mapsto \hat{v}_i \Big) |_{i=0,\, i \neq k}^{n} \rrbracket_\varepsilon \uplus (\llbracket e_1 \rrbracket_\rho, \llbracket e_2 \rrbracket_\rho) \mapsto \llbracket \hat{v}_k \rrbracket_\varepsilon \end{aligned}$$

We can now apply the [Property Access] rule of JSIL semantics, concluding:

$$\langle h, \rho, x := [e_1, e_2] \rangle \rightarrow \langle h, \rho[x \mapsto \llbracket \hat{v}_k \rrbracket_\varepsilon] \rangle$$

meaning that: $h' = h$ and $\rho' = \rho[x \mapsto \llbracket \hat{v}_k \rrbracket_\varepsilon]$. We have now to prove that $(h', \rho', \varepsilon) \in \mathcal{M}_{\pi''}(\hat{h}', \hat{\rho}')$. Observe that:

$$\begin{array}{l} h' = h = \llbracket \hat{h} \rrbracket_\varepsilon = \llbracket \hat{h}' \rrbracket_\varepsilon \text{ (because } h' = h \text{ and } \hat{h} = \hat{h}'\text{)} \\ \rho' = \llbracket \hat{\rho} \rrbracket_\varepsilon [x \mapsto \llbracket \hat{v}_k \rrbracket_\varepsilon] = \llbracket \hat{\rho}[x \mapsto \hat{v}_k] \rrbracket_\varepsilon = \llbracket \hat{\rho}' \rrbracket_\varepsilon \end{array}$$

which concludes the proof.

[Property Deletion] We conclude that $bc = \text{delete}\,(\hat{\mathrm{e}}_1, \hat{\mathrm{e}}_2)$, for some expressions $\hat{\mathrm{e}}_1$ and $\hat{\mathrm{e}}_2$ and that:

$$\begin{array}{l} \hat{h}' = \hat{h}'' \uplus \Big((l, \hat{p}_i) \mapsto \hat{v}_i \Big) \,|_{i=0,\, i \neq k}^{n} \\ \hat{\rho}' = \hat{\rho} \\ \pi' = \pi \ \wedge \ \Big((\hat{p}_k = \hat{e}_p) \ \wedge \bigwedge_{i=0,\, i \neq k}^{n} (\hat{p}_i \neq \hat{e}_p) \Big) \end{array}$$

where $l = \llbracket \hat{\mathrm{e}}_1 \rrbracket_{\hat{\rho}}$ and $\hat{e}_p = \llbracket \hat{\mathrm{e}}_2 \rrbracket_{\hat{\rho}}$. From $(h, \rho, \varepsilon) \in \mathcal{M}_{\pi''}(\hat{h}, \hat{\rho})$, we conclude that $h = \llbracket \hat{h} \rrbracket_\varepsilon$, $\rho = \llbracket \hat{\rho} \rrbracket_\varepsilon$, and $\varepsilon \vdash \pi''$. We let $e_1 = \llbracket \hat{\mathrm{e}}_1 \rrbracket_\varepsilon$ and $e_2 = \llbracket \hat{\mathrm{e}}_2 \rrbracket_\varepsilon$, and note that $\llbracket bc \rrbracket_\varepsilon = \text{delete}\,(e_1, e_2)$. Now, we have to prove that we can apply the [Property Deletion] rule in the concrete state. To this end, we have to show that: $h = h' \uplus (\llbracket e_1 \rrbracket_\rho, \llbracket e_2 \rrbracket_\rho) \mapsto -$. Note that:

$$\begin{array}{l} \varepsilon \vdash \pi'' \ \wedge \ \pi'' \vdash \pi' \ \wedge \ \pi' \vdash \hat{p}_k = \hat{e}_p \implies \varepsilon \vdash \hat{p}_k = \hat{e}_p \\ \llbracket e_1 \rrbracket_\rho = \llbracket e_1 \rrbracket_{\llbracket \hat{\rho} \rrbracket_\varepsilon} = \llbracket \llbracket \hat{\mathrm{e}}_1 \rrbracket_\varepsilon \rrbracket_{\llbracket \hat{\rho} \rrbracket_\varepsilon} = \llbracket \llbracket \hat{\mathrm{e}}_1 \rrbracket_{\hat{\rho}} \rrbracket_\varepsilon = \llbracket l \rrbracket_\varepsilon = l \\ \llbracket e_2 \rrbracket_\rho = \llbracket e_2 \rrbracket_{\llbracket \hat{\rho} \rrbracket_\varepsilon} = \llbracket \llbracket \hat{\mathrm{e}}_2 \rrbracket_\varepsilon \rrbracket_{\llbracket \hat{\rho} \rrbracket_\varepsilon} = \llbracket \llbracket \hat{\mathrm{e}}_2 \rrbracket_{\hat{\rho}} \rrbracket_\varepsilon = \llbracket \hat{e}_p \rrbracket_\varepsilon = \llbracket \hat{p}_k \rrbracket_\varepsilon \text{ (because } \ \text{and } \varepsilon \vdash \hat{p}_k = \hat{e}_p\text{)} \\ h = \llbracket \hat{h}'' \uplus \Big((l, \hat{p}_i) \mapsto \hat{v}_i \Big) \,|_{i=0}^{n} \rrbracket_\varepsilon = \llbracket \hat{h}'' \uplus \Big((l, \hat{p}_i) \mapsto \hat{v}_i \Big) \,|_{i=0,\, i \neq k}^{n} \rrbracket_\varepsilon \uplus \llbracket (l, \hat{p}_k) \mapsto \hat{v}_k \rrbracket_\varepsilon \\ \quad = \llbracket \hat{h}'' \uplus \Big((l, \hat{p}_i) \mapsto \hat{v}_i \Big) \,|_{i=0,\, i \neq k}^{n} \rrbracket_\varepsilon \uplus (l, \llbracket \hat{p}_k \rrbracket_\varepsilon) \mapsto \llbracket \hat{v}_k \rrbracket_\varepsilon \\ \quad = \llbracket \hat{h}'' \uplus \Big((l, \hat{p}_i) \mapsto \hat{v}_i \Big) \,|_{i=0,\, i \neq k}^{n} \rrbracket_\varepsilon \uplus (\llbracket e_1 \rrbracket_\rho, \llbracket e_2 \rrbracket_\rho) \mapsto \llbracket \hat{v}_k \rrbracket_\varepsilon \\ \quad = \llbracket \hat{h}' \rrbracket_\varepsilon \uplus (\llbracket e_1 \rrbracket_\rho, \llbracket e_2 \rrbracket_\rho) \mapsto - \end{array}$$

We can now apply the [Property Deletion] rule of the JSIL semantics, concluding:

$$\langle h, \rho, \text{delete}\,(e_1, e_2) \rangle \to \langle \llbracket \hat{h}' \rrbracket_\varepsilon, \rho \rangle$$

meaning that: $h' = \llbracket \hat{h}' \rrbracket_\varepsilon$ and $\rho' = \rho$. Now, we have to prove that $(h', \rho', \varepsilon) \in \mathcal{M}_{\pi''}(\hat{h}', \hat{\rho}')$. Noting that $h' = \llbracket \hat{h}' \rrbracket_\varepsilon$ and $\rho' = \rho = \llbracket \hat{\rho} \rrbracket_\varepsilon = \llbracket \hat{\rho}' \rrbracket_\varepsilon$, the result follows.

[Property Assignment - Found] We conclude that $bc = [\hat{\mathrm{e}}_1, \hat{\mathrm{e}}_2] := \hat{\mathrm{e}}_3$ for some extended expressions $\hat{\mathrm{e}}_1$, $\hat{\mathrm{e}}_2$, and $\hat{\mathrm{e}}_3$, and that:

$$\begin{array}{l} \hat{h} = \hat{h}'' \uplus \Big((l, \hat{p}_i) \mapsto \hat{v}_i \Big) \,|_{i=0}^{n} \\ \hat{h}' = \hat{h}'' \uplus \Big((l, \hat{p}_i) \mapsto \hat{v}_i \Big) \,|_{i=0,\, i \neq k}^{n} \uplus (l, \hat{e}_p) \mapsto \hat{e}_v \\ \hat{\rho}' = \hat{\rho} \\ \pi' = \pi \ \wedge \ \Big((\hat{p}_k = \hat{e}_p) \ \wedge \bigwedge_{i=0,\, i \neq k}^{n} (\hat{p}_i \neq \hat{e}_p) \end{array}$$

where $\llbracket \hat{\mathrm{e}}_1 \rrbracket_{\hat{\rho}} = l$, $\llbracket \hat{\mathrm{e}}_2 \rrbracket_{\hat{\rho}} = \hat{e}_p$, $\llbracket \hat{\mathrm{e}}_3 \rrbracket_{\hat{\rho}} = \hat{e}_v$. From $(h, \rho, \varepsilon) \in \mathcal{M}_{\pi'}(\hat{h}, \hat{\rho})$, we conclude that $h = \llbracket \hat{h} \rrbracket_\varepsilon$, $\rho = \llbracket \hat{\rho} \rrbracket_\varepsilon$, and $\varepsilon \vdash \pi''$. We let $e_1 = \llbracket \hat{\mathrm{e}}_1 \rrbracket_\varepsilon$, $e_2 = \llbracket \hat{\mathrm{e}}_2 \rrbracket_\varepsilon$, and $e_3 = \llbracket \hat{\mathrm{e}}_3 \rrbracket_\varepsilon$, and note that $\llbracket bc \rrbracket_\varepsilon = [e_1, e_2] := e_3$. Now, we have to prove that we can apply the [Property Assignment - Found] rule in the concrete state. To this end, we have to show that there is a concrete heap h'' such that:

$h = h'' \uplus (\llbracket e_1 \rrbracket_\rho, \llbracket e_2 \rrbracket_\rho) \mapsto -$. Note that:

$$\begin{aligned}
&\varepsilon \vdash \pi'' \ \wedge\ \pi'' \vdash \pi' \ \wedge\ \pi' \vdash \hat{p}_k = \hat{e}_p \implies \varepsilon \vdash \hat{p}_k = \hat{e}_p \\
&\llbracket e_1 \rrbracket_\rho = \llbracket e_1 \rrbracket_{\llbracket \hat{\rho} \rrbracket_\varepsilon} = \llbracket \llbracket \hat{\mathrm{e}}_1 \rrbracket_\varepsilon \rrbracket_{\llbracket \hat{\rho} \rrbracket_\varepsilon} = \llbracket \llbracket \hat{\mathrm{e}}_1 \rrbracket_{\hat{\rho}} \rrbracket_\varepsilon = \llbracket l \rrbracket_\varepsilon = l \\
&\llbracket e_2 \rrbracket_\rho = \llbracket e_2 \rrbracket_{\llbracket \hat{\rho} \rrbracket_\varepsilon} = \llbracket \llbracket \hat{\mathrm{e}}_2 \rrbracket_\varepsilon \rrbracket_{\llbracket \hat{\rho} \rrbracket_\varepsilon} = \llbracket \llbracket \hat{\mathrm{e}}_2 \rrbracket_{\hat{\rho}} \rrbracket_\varepsilon = \llbracket \hat{e}_p \rrbracket_\varepsilon = \llbracket \hat{p}_k \rrbracket_\varepsilon \text{ (because } \varepsilon \vdash \hat{p}_k = \hat{e}_p) \\
&\llbracket e_3 \rrbracket_\rho = \llbracket e_3 \rrbracket_{\llbracket \hat{\rho} \rrbracket_\varepsilon} = \llbracket \llbracket \hat{\mathrm{e}}_3 \rrbracket_\varepsilon \rrbracket_{\llbracket \hat{\rho} \rrbracket_\varepsilon} = \llbracket \llbracket \hat{\mathrm{e}}_3 \rrbracket_{\hat{\rho}} \rrbracket_\varepsilon = \llbracket \hat{e}_v \rrbracket_\varepsilon \\
&h = \llbracket \hat{h}'' \uplus \big((l, \hat{p}_i) \mapsto \hat{v}_i\big) \mid_{i=0}^{n} \rrbracket_\varepsilon = \llbracket \hat{h}'' \uplus \big((l, \hat{p}_i) \mapsto \hat{v}_i\big) \mid_{i=0,\, i \neq k}^{n} \rrbracket_\varepsilon \uplus \llbracket (l, \hat{p}_k) \mapsto \hat{v}_k \rrbracket_\varepsilon \\
&\quad = \llbracket \hat{h}'' \uplus \big((l, \hat{p}_i) \mapsto \hat{v}_i\big) \mid_{i=0,\, i \neq k}^{n} \rrbracket_\varepsilon \uplus (l, \llbracket \hat{p}_k \rrbracket_\varepsilon) \mapsto \llbracket \hat{v}_k \rrbracket_\varepsilon \\
&\quad = \llbracket \hat{h}'' \uplus \big((l, \hat{p}_i) \mapsto \hat{v}_i\big) \mid_{i=0,\, i \neq k}^{n} \rrbracket_\varepsilon \uplus (\llbracket e_1 \rrbracket_\rho, \llbracket e_2 \rrbracket_\rho) \mapsto \llbracket \hat{v}_k \rrbracket_\varepsilon
\end{aligned}$$

We can now apply the [Property Assignment - Found] rule of JSIL semantics, concluding:

$$\langle h, \rho, [e_1, e_2] := e_3 \rangle \to \langle \llbracket \hat{h}'' \uplus \big((l, \hat{p}_i) \mapsto \hat{v}_i\big) \mid_{i=0,\, i \neq k}^{n} \rrbracket_\varepsilon \uplus (\llbracket e_1 \rrbracket_\rho, \llbracket e_2 \rrbracket_\rho) \mapsto \llbracket e_3 \rrbracket_\rho, \rho \rangle$$

meaning that: $h' = \llbracket \hat{h}'' \uplus \big((l, \hat{p}_i) \mapsto \hat{v}_i\big) \mid_{i=0,\, i \neq k}^{n} \rrbracket_\varepsilon \uplus (\llbracket e_1 \rrbracket_\rho, \llbracket e_2 \rrbracket_\rho) \mapsto \llbracket e_3 \rrbracket_\rho$ and $\rho' = \rho$. Now we have to prove that $(h', \rho', \varepsilon) \in \mathcal{M}_{\pi''}(\hat{h}', \hat{\rho}')$. Noting that:

$$\begin{aligned}
h' &= \llbracket \hat{h}'' \uplus \big((l, \hat{p}_i) \mapsto \hat{v}_i\big) \mid_{i=0,\, i \neq k}^{n} \rrbracket_\varepsilon \uplus (\llbracket e_1 \rrbracket_\rho, \llbracket e_2 \rrbracket_\rho) \mapsto \llbracket e_3 \rrbracket_\rho \\
&= \llbracket \hat{h}'' \uplus \big((l, \hat{p}_i) \mapsto \hat{v}_i\big) \mid_{i=0,\, i \neq k}^{n} \rrbracket_\varepsilon \uplus (l, \llbracket \hat{e}_p \rrbracket_\varepsilon) \mapsto \llbracket \hat{e}_v \rrbracket_\varepsilon \\
&= \llbracket \hat{h}'' \uplus \big((l, \hat{p}_i) \mapsto \hat{v}_i\big) \mid_{i=0,\, i \neq k}^{n} \uplus (l, \hat{e}_p) \mapsto \hat{e}_v \rrbracket_\varepsilon \\
&= \llbracket \hat{h}' \rrbracket_\varepsilon \\
\rho' &= \rho = \llbracket \hat{\rho} \rrbracket_\varepsilon = \llbracket \hat{\rho}' \rrbracket_\varepsilon
\end{aligned}$$

the result follows.

[Property Assignment - Not Found] We conclude that $bc = [\hat{\mathrm{e}}_1, \hat{\mathrm{e}}_2] := \hat{\mathrm{e}}_3$ for some extended expressions $\hat{\mathrm{e}}_1$, $\hat{\mathrm{e}}_2$, and $\hat{\mathrm{e}}_3$, and that:

$$\begin{aligned}
&\hat{h} = \hat{h}'' \uplus \big((l, \hat{p}_i) \mapsto \hat{v}_i\big) \mid_{i=0}^{n} \\
&\hat{h}' = \hat{h} \uplus (l, \hat{e}_p) \mapsto \hat{e}_v \\
&\hat{\rho}' = \hat{\rho} \\
&\pi' = \pi \ \wedge\ \textstyle\bigwedge_{i=0}^{n} (\hat{p}_i \neq \hat{e}_p)
\end{aligned}$$

where: $\llbracket \hat{\mathrm{e}}_1 \rrbracket_{\hat{\rho}} = l$, $\llbracket \hat{\mathrm{e}}_2 \rrbracket_{\hat{\rho}} = \hat{e}_p$, $\llbracket \hat{\mathrm{e}}_3 \rrbracket_{\hat{\rho}} = \hat{e}_v$, $(l, -) \notin \mathrm{dom}(\hat{h}'')$, and $0 \leq k \leq n$. From $(h, \rho, \varepsilon) \in \mathcal{M}_{\pi'}(\hat{h}, \hat{\rho})$, we conclude that $h = \llbracket \hat{h} \rrbracket_\varepsilon$, $\rho = \llbracket \hat{\rho} \rrbracket_\varepsilon$, and $\varepsilon \vdash \pi''$. We let $e_1 = \llbracket \hat{\mathrm{e}}_1 \rrbracket_\varepsilon$, $e_2 = \llbracket \hat{\mathrm{e}}_2 \rrbracket_\varepsilon$, and $e_3 = \llbracket \hat{\mathrm{e}}_3 \rrbracket_\varepsilon$, and note that $\llbracket bc \rrbracket_\varepsilon = [e_1, e_2] := e_3$. We now have to prove that we can apply the [Property Assignment - Found] rule in the concrete state. To this end, we have to show that: $(\llbracket e_1 \rrbracket_\rho, \llbracket e_2 \rrbracket_\rho) \notin \mathrm{dom}(h)$. Note that:

$$\begin{aligned}
&\varepsilon \vdash \pi'' \ \wedge\ \pi'' \vdash \pi' \ \wedge\ \forall_{0 \leq i \leq n}\, \pi' \vdash \hat{p}_i \neq \hat{e}_p \implies \forall_{0 \leq i \leq n}\, \varepsilon \vdash \hat{p}_i \neq \hat{e}_p \\
&\llbracket e_1 \rrbracket_\rho = \llbracket e_1 \rrbracket_{\llbracket \hat{\rho} \rrbracket_\varepsilon} = \llbracket \llbracket \hat{\mathrm{e}}_1 \rrbracket_\varepsilon \rrbracket_{\llbracket \hat{\rho} \rrbracket_\varepsilon} = \llbracket \llbracket \hat{\mathrm{e}}_1 \rrbracket_{\hat{\rho}} \rrbracket_\varepsilon = \llbracket l \rrbracket_\varepsilon = l \\
&\llbracket e_2 \rrbracket_\rho = \llbracket e_2 \rrbracket_{\llbracket \hat{\rho} \rrbracket_\varepsilon} = \llbracket \llbracket \hat{\mathrm{e}}_2 \rrbracket_\varepsilon \rrbracket_{\llbracket \hat{\rho} \rrbracket_\varepsilon} = \llbracket \llbracket \hat{\mathrm{e}}_2 \rrbracket_{\hat{\rho}} \rrbracket_\varepsilon = \llbracket \hat{e}_p \rrbracket_\varepsilon \\
&\llbracket e_3 \rrbracket_\rho = \llbracket e_3 \rrbracket_{\llbracket \hat{\rho} \rrbracket_\varepsilon} = \llbracket \llbracket \hat{\mathrm{e}}_3 \rrbracket_\varepsilon \rrbracket_{\llbracket \hat{\rho} \rrbracket_\varepsilon} = \llbracket \llbracket \hat{\mathrm{e}}_3 \rrbracket_{\hat{\rho}} \rrbracket_\varepsilon = \llbracket \hat{e}_v \rrbracket_\varepsilon \\
&h = \llbracket \hat{h}'' \uplus \big((l, \hat{p}_i) \mapsto \hat{v}_i\big) \mid_{i=0}^{n} \rrbracket_\varepsilon = \llbracket \hat{h}'' \rrbracket_\varepsilon \uplus \textstyle\biguplus_{0 \leq i \leq n} ((l, \llbracket \hat{p}_i \rrbracket_\varepsilon) \mapsto \llbracket \hat{v}_i \rrbracket_\varepsilon)
\end{aligned}$$

From $(l, -) \notin \mathrm{dom}(\hat{h}'')$, we conclude that $(l, -) \notin \mathrm{dom}(\llbracket \hat{h}'' \rrbracket_\varepsilon)$. Since for $0 \leq i \leq n$, it holds that $\varepsilon \vdash \varepsilon \vdash \hat{p}_i \neq \hat{e}_p$, we additionally conclude that:

$$\forall_{0 \leq i \leq n}\, \llbracket \hat{p}_i \rrbracket_\varepsilon \neq \llbracket \hat{e}_p \rrbracket_\varepsilon$$

Recalling that $\llbracket e_2 \rrbracket_\rho = \llbracket \hat{e}_p \rrbracket_\varepsilon$, we conclude that $\forall_{0 \leq i \leq n}\, \llbracket \hat{p}_i \rrbracket_\varepsilon \neq \llbracket e_2 \rrbracket_\rho$, from which it follows (together with $(l, -) \notin \mathrm{dom}(\llbracket \hat{h}'' \rrbracket_\varepsilon)$) that $(\llbracket e_1 \rrbracket_\rho, \llbracket e_2 \rrbracket_\rho) \notin \mathrm{dom}(h)$. We can now apply the [Property

Assignment - Not Found] rule of JSIL semantics, concluding:

$$\langle h, \rho, [e_1, e_2] := e_3 \rangle \rightarrow \langle h \uplus (\llbracket e_1 \rrbracket_\rho, \llbracket e_2 \rrbracket_\rho) \mapsto \llbracket e_3 \rrbracket_\rho, \rho \rangle$$

meaning that $h' = h \uplus (\llbracket e_1 \rrbracket_\rho, \llbracket e_2 \rrbracket_\rho) \mapsto \llbracket e_3 \rrbracket_\rho$ and $\rho' = \rho$. Now, we have to prove that $(h', \rho', \varepsilon) \in \mathcal{M}_{\pi''}(\hat{h}', \hat{\rho}')$. Noting that:

$$\begin{aligned} h' &= \llbracket \hat{h} \rrbracket_\varepsilon \uplus (\llbracket e_1 \rrbracket_\rho, \llbracket e_2 \rrbracket_\rho) \mapsto \llbracket e_3 \rrbracket_\rho \\ &= \llbracket \hat{h} \rrbracket_\varepsilon \uplus (l, \llbracket \hat{e}_p \rrbracket_\varepsilon) \mapsto \llbracket \hat{e}_v \rrbracket_\varepsilon \\ &= \llbracket \hat{h} \uplus (l, \hat{e}_p) \mapsto \hat{e}_v \rrbracket_\varepsilon \\ &= \llbracket \hat{h}' \rrbracket_\varepsilon \\ \rho' &= \rho = \llbracket \hat{\rho} \rrbracket_\varepsilon = \llbracket \hat{\rho}' \rrbracket_\varepsilon \end{aligned}$$

the result follows.

[Member Check - True] We conclude that $bc = x := \mathsf{hasField}\,(\hat{\mathrm{e}}_1, \hat{\mathrm{e}}_2)$ for some variable x and expressions $\hat{\mathrm{e}}_1$ and $\hat{\mathrm{e}}_2$, and that:

$$\begin{aligned} &\hat{h} = \hat{h}'' \uplus \Big((l, \hat{p}_i) \mapsto -\Big)\,\Big|_{i=0}^{n} \\ &\hat{h}' = \hat{h} \\ &\hat{\rho}' = \hat{\rho}[x \mapsto \mathsf{true}] \\ &\pi' = \pi \;\wedge\; \Big((\hat{p}_k = \hat{e}_p) \;\wedge \bigwedge\nolimits_{i=0,\, i \neq k}^{n} (\hat{p}_i \neq \hat{e}_p)\Big) \end{aligned}$$

where $\llbracket \hat{\mathrm{e}}_1 \rrbracket_{\hat{\rho}} = l$, $\llbracket \hat{\mathrm{e}}_2 \rrbracket_{\hat{\rho}} = \hat{e}_p$, $(l, -) \notin \mathrm{dom}(\hat{h}'')$, and $0 \leq k \leq n$. From $(h, \rho, \varepsilon) \in \mathcal{M}_{\pi''}(\hat{h}, \hat{\rho})$, we conclude that $h = \llbracket \hat{h} \rrbracket_\varepsilon$, $\rho = \llbracket \hat{\rho} \rrbracket_\varepsilon$, and $\varepsilon \vdash \pi''$. We let $e_1 = \llbracket \hat{\mathrm{e}}_1 \rrbracket_\varepsilon$ and $e_2 = \llbracket \hat{\mathrm{e}}_2 \rrbracket_\varepsilon$, and note that $\llbracket bc \rrbracket_\varepsilon = x := \mathsf{hasField}\,(e_1, e_2)$. Now we have to prove that we can apply the [Member Check - True] rule in the concrete state. To this end, we have to show that: $h = h'' \uplus (\llbracket e_1 \rrbracket_\rho, \llbracket e_2 \rrbracket_\rho) \mapsto -$, for some concrete heap h''. Note that:

$$\begin{aligned} &\varepsilon \vdash \pi'' \;\wedge\; \pi'' \vdash \pi' \;\wedge\; \pi' \vdash \hat{p}_k = \hat{e}_p \implies \varepsilon \vdash \hat{p}_k = \hat{e}_p \\ &\llbracket e_1 \rrbracket_\rho = \llbracket e_1 \rrbracket_{\llbracket \hat{\rho} \rrbracket_\varepsilon} = \llbracket \llbracket \hat{\mathrm{e}}_1 \rrbracket_\varepsilon \rrbracket_{\llbracket \hat{\rho} \rrbracket_\varepsilon} = \llbracket \llbracket \hat{\mathrm{e}}_1 \rrbracket_{\hat{\rho}} \rrbracket_\varepsilon = \llbracket l \rrbracket_\varepsilon = l \\ &\llbracket e_2 \rrbracket_\rho = \llbracket e_2 \rrbracket_{\llbracket \hat{\rho} \rrbracket_\varepsilon} = \llbracket \llbracket \hat{\mathrm{e}}_2 \rrbracket_\varepsilon \rrbracket_{\llbracket \hat{\rho} \rrbracket_\varepsilon} = \llbracket \llbracket \hat{\mathrm{e}}_2 \rrbracket_{\hat{\rho}} \rrbracket_\varepsilon = \llbracket \hat{e}_p \rrbracket_\varepsilon \\ &h = \llbracket \hat{h}'' \uplus \Big((l, \hat{p}_i) \mapsto \hat{v}_i\Big)\,\Big|_{i=0}^{n} \rrbracket_\varepsilon \\ &\qquad = \llbracket \hat{h}'' \uplus \Big((l, \hat{p}_i) \mapsto \hat{v}_i\Big)\,\Big|_{i=0,\, i \neq k}^{n} \uplus (l, \hat{p}_k) \mapsto \hat{v}_k \rrbracket_\varepsilon \\ &\qquad = \llbracket \hat{h}'' \uplus \Big((l, \hat{p}_i) \mapsto \hat{v}_i\Big)\,\Big|_{i=0,\, i \neq k}^{n} \rrbracket_\varepsilon \uplus \llbracket (l, \hat{p}_k) \mapsto \hat{v}_k \rrbracket_\varepsilon \\ &\qquad = \llbracket \hat{h}'' \uplus \Big((l, \hat{p}_i) \mapsto \hat{v}_i\Big)\,\Big|_{i=0,\, i \neq k}^{n} \rrbracket_\varepsilon \uplus (l, \llbracket \hat{p}_k \rrbracket_\varepsilon) \mapsto \llbracket \hat{v}_k \rrbracket_\varepsilon \\ &\qquad = \llbracket \hat{h}'' \uplus \Big((l, \hat{p}_i) \mapsto \hat{v}_i\Big)\,\Big|_{i=0,\, i \neq k}^{n} \rrbracket_\varepsilon \uplus (l, \llbracket \hat{e}_p \rrbracket_\varepsilon) \mapsto \llbracket \hat{v}_k \rrbracket_\varepsilon \text{ (using } \varepsilon \vdash \hat{p}_k = \hat{e}_p) \\ &\qquad = \llbracket \hat{h}'' \uplus \Big((l, \hat{p}_i) \mapsto \hat{v}_i\Big)\,\Big|_{i=0,\, i \neq k}^{n} \rrbracket_\varepsilon \uplus (\llbracket e_1 \rrbracket_\rho, \llbracket e_2 \rrbracket_\rho) \mapsto - \end{aligned}$$

We can now apply the [Member Check - True] rule of JSIL semantics, concluding that:

$$\langle h, \rho, x := \mathsf{hasField}\,(e_1, e_2) \rangle \rightarrow \langle h, \rho[x \mapsto \mathsf{true}] \rangle$$

which means that $h' = h$ and $\rho' = \rho[x \mapsto \mathsf{true}]$. We now have to prove that $(h', \rho', \varepsilon) \in \mathcal{M}_{\pi''}(\hat{h}', \hat{\rho}')$. Noting that:

$$\begin{aligned} &h' = h = \llbracket \hat{h} \rrbracket_\varepsilon = \llbracket \hat{h}' \rrbracket_\varepsilon \\ &\rho' = \rho[x \mapsto \mathsf{true}] = \llbracket \hat{\rho} \rrbracket_\varepsilon[x \mapsto \mathsf{true}] = \llbracket \hat{\rho}[x \mapsto \mathsf{true}] \rrbracket_\varepsilon = \llbracket \hat{\rho}' \rrbracket_\varepsilon \end{aligned}$$

the result follows.

[Member Check - False] We conclude that $bc = x := \textsf{hasField}\,(\hat{\mathrm{e}}_1, \hat{\mathrm{e}}_2)$ for some variable x and expressions $\hat{\mathrm{e}}_1$ and $\hat{\mathrm{e}}_2$, and that:

$$\begin{array}{l} \hat{h} = \hat{h}'' \uplus \Big((l, \hat{p}_i) \mapsto -\Big)\,|_{i=0}^{n} \\ \hat{h}' = \hat{h} \\ \hat{\rho}' = \hat{\rho}[x \mapsto \mathsf{false}] \\ \pi' = \pi \ \wedge\ \bigwedge_{i=0}^{n}(\hat{p}_i \neq \hat{e}_p) \end{array}$$

where $\llbracket \hat{\mathrm{e}}_1 \rrbracket_{\hat{\rho}} = l$, $\llbracket \hat{\mathrm{e}}_2 \rrbracket_{\hat{\rho}} = \hat{e}_p$, $(l, -) \notin \mathsf{dom}(\hat{h}'')$, and $0 \leq k \leq n$. From $(h, \rho, \varepsilon) \in \mathcal{M}_{\pi'}(\hat{h}, \hat{\rho})$, we conclude that $h = \llbracket \hat{h} \rrbracket_{\varepsilon}$, $\rho = \llbracket \hat{\rho} \rrbracket_{\varepsilon}$, and $\varepsilon \vdash \pi''$. We let $e_1 = \llbracket \hat{\mathrm{e}}_1 \rrbracket_{\varepsilon}$ and $e_2 = \llbracket \hat{\mathrm{e}}_2 \rrbracket_{\varepsilon}$, and note that $\llbracket bc \rrbracket_{\varepsilon} = x := \textsf{hasField}\,(e_1, e_2)$. We now have to prove that we can apply the [Member Check - False] rule in the concrete state. To this end, we have to show that: $(\llbracket e_1 \rrbracket_{\rho}, \llbracket e_2 \rrbracket_{\rho}) \notin \mathsf{dom}(h)$. Note that:

$$\begin{array}{l} \varepsilon \vdash \pi'' \ \wedge\ \pi'' \vdash \pi' \ \wedge\ \forall_{0 \leq i \leq n}\, \pi' \vdash \hat{p}_i \neq \hat{e}_p \implies \forall_{0 \leq i \leq n}\, \varepsilon \vdash \hat{p}_i \neq \hat{e}_p \\ \llbracket e_1 \rrbracket_{\rho} = \llbracket e_1 \rrbracket_{\llbracket \hat{\rho} \rrbracket_{\varepsilon}} = \llbracket \llbracket \hat{\mathrm{e}}_1 \rrbracket_{\varepsilon} \rrbracket_{\llbracket \hat{\rho} \rrbracket_{\varepsilon}} = \llbracket \llbracket \hat{\mathrm{e}}_1 \rrbracket_{\hat{\rho}} \rrbracket_{\varepsilon} = \llbracket l \rrbracket_{\varepsilon} = l \\ \llbracket e_2 \rrbracket_{\rho} = \llbracket e_2 \rrbracket_{\llbracket \hat{\rho} \rrbracket_{\varepsilon}} = \llbracket \llbracket \hat{\mathrm{e}}_2 \rrbracket_{\varepsilon} \rrbracket_{\llbracket \hat{\rho} \rrbracket_{\varepsilon}} = \llbracket \llbracket \hat{\mathrm{e}}_2 \rrbracket_{\hat{\rho}} \rrbracket_{\varepsilon} = \llbracket \hat{e}_p \rrbracket_{\varepsilon} \\ h = \llbracket \hat{h}'' \uplus \Big((l, \hat{p}_i) \mapsto \hat{v}_i\Big)\,|_{i=0}^{n} \rrbracket_{\varepsilon} \\ \quad\;\; = \llbracket \hat{h}'' \rrbracket_{\varepsilon} \uplus \biguplus_{0 \leq i \leq n} (l, \llbracket \hat{p}_i \rrbracket_{\varepsilon}) \mapsto \llbracket \hat{v}_i \rrbracket_{\varepsilon} \end{array}$$

Since $(l, -) \notin \mathsf{dom}(\hat{h}'')$, we conclude that $(l, -) \notin \llbracket \hat{h}'' \rrbracket_{\varepsilon}$. Since for all $0 \leq i \leq n$, it holds that $\varepsilon \vdash \hat{p}_i \neq \hat{e}_p$, we additionally conclude that:

$$\forall_{0 \leq i \leq n}\, \llbracket \hat{p}_i \rrbracket_{\varepsilon} \neq \llbracket \hat{e}_p \rrbracket_{\varepsilon}$$

Recalling that $\llbracket e_2 \rrbracket_{\rho} = \llbracket \hat{e}_p \rrbracket_{\varepsilon}$, we conclude that $\forall_{0 \leq i \leq n}\, \llbracket \hat{p}_i \rrbracket_{\varepsilon} \neq \llbracket e_2 \rrbracket_{\rho}$, from which it follows (together with $(l, -) \notin \mathsf{dom}(\llbracket \hat{h}'' \rrbracket_{\varepsilon})$) that $(\llbracket e_1 \rrbracket_{\rho}, \llbracket e_2 \rrbracket_{\rho}) \notin \mathsf{dom}(h)$. We can now apply the [Member Check - False] rule of JSIL semantics, concluding:

$$\langle h, \rho, x := \textsf{hasField}\,(e_1, e_2) \rangle \rightarrow \langle h, \rho[x \mapsto \mathsf{false}] \rangle$$

meaning that $h' = h$ and $\rho' = \rho[x \mapsto \mathsf{false}]$. Now we have to prove that $(h', \rho', \varepsilon) \in \mathcal{M}_{\pi''}(\hat{h}', \hat{\rho}')$. Noting that:

$$\begin{array}{l} h' = h = \llbracket \hat{h} \rrbracket_{\varepsilon} = \llbracket \hat{h}' \rrbracket_{\varepsilon} \\ \rho' = \rho[x \mapsto \mathsf{false}] = \llbracket \hat{\rho} \rrbracket_{\varepsilon}[x \mapsto \mathsf{false}]] = \llbracket \hat{\rho}[x \mapsto \mathsf{false}] \rrbracket_{\varepsilon} = \llbracket \hat{\rho}' \rrbracket_{\varepsilon} \end{array}$$

the result follows.

[Assert - True] We conclude that $bc = \textsf{assert}\,(\hat{\mathrm{e}})$ for some extended expression $\hat{\mathrm{e}}$, and that:

$$\hat{h}' = \hat{h} \quad \hat{\rho}' = \hat{\rho} \quad \pi' = \pi \quad \pi \vdash \llbracket \hat{\mathrm{e}} \rrbracket_{\hat{\rho}}$$

From $(h, \rho, \varepsilon) \in \mathcal{M}_{\pi''}(\hat{h}, \hat{\rho})$, we conclude that $h = \llbracket \hat{h} \rrbracket_{\varepsilon}$, $\rho = \llbracket \hat{\rho} \rrbracket_{\varepsilon}$, and $\varepsilon \vdash \pi''$. Because $\pi'' \vdash \pi' = \pi$, we conclude that $\varepsilon \vdash \pi$. We let $e = \llbracket \hat{\mathrm{e}} \rrbracket_{\varepsilon}$, and note that $\llbracket bc \rrbracket_{\varepsilon} = x := \textsf{assert}\,(e)$. We now have to prove that we can apply the [Assert - True] rule in the concrete state. To this end, we have to show that: $\llbracket e \rrbracket_{\rho} = \mathsf{true}$. Noting that: $\llbracket e \rrbracket_{\rho} = \llbracket e \rrbracket_{\llbracket \hat{\rho} \rrbracket_{\varepsilon}} = \llbracket \llbracket \hat{\mathrm{e}} \rrbracket_{\hat{\rho}} \rrbracket_{\varepsilon}$, we conclude (using $\varepsilon \vdash \pi$ and $\pi \vdash \llbracket \hat{\mathrm{e}} \rrbracket_{\hat{\rho}}$) that $\llbracket e \rrbracket_{\rho} = \mathsf{true}$. We can now apply the [Assert - True] rule of JSIL semantics, concluding:

$$\langle h, \rho, \textsf{assert}\,(e) \rangle \rightarrow \langle h, \rho \rangle$$

meaning that $h' = h$ and $\rho' = \rho$. Now we have to prove that $(h', \rho', \varepsilon) \in \mathcal{M}_{\pi''}(\hat{h}', \hat{\rho}')$. Noting that:

$$h' = h = \llbracket \hat{h} \rrbracket_{\varepsilon} = \llbracket \hat{h}' \rrbracket_{\varepsilon} \quad \rho' = \rho = \llbracket \hat{\rho} \rrbracket_{\varepsilon} = \llbracket \hat{\rho} \rrbracket_{\varepsilon} = \llbracket \hat{\rho}' \rrbracket_{\varepsilon}$$

the result follows. The case for the [Assume] rule is analogous, noting that if $\varepsilon \vdash \pi \ \wedge\ \hat{e}$, then $\varepsilon \vdash \pi$ and $\varepsilon \vdash \hat{e}$. □

Lemma C.5 (Soundness of JSIL symbolic execution - Single Step).

$$\begin{array}{l}\forall\, \mathsf{p}, \hat{h}, \hat{\rho}, i, \pi, \hat{C}, \mu, \hat{h}', \hat{\rho}', i', \pi', \hat{C}', \mu', h, \rho, C, \varepsilon, \pi''. \\ \quad \mathsf{p} : \langle \hat{h}, \hat{\rho}, i, \pi \rangle^{\mu}_{\hat{C}} \rightsquigarrow \langle \hat{h}', \hat{\rho}', i', \pi' \rangle^{\mu'}_{\hat{C}'} \;\wedge\; \pi'' \vdash \pi' \;\wedge\; (h, \rho, C, \varepsilon) \in \mathcal{M}_{\pi''}(\hat{h}, \hat{\rho}, \hat{C}) \;\wedge\; \mathsf{svars}(\mathsf{p}) \subseteq \mathsf{dom}(\varepsilon) \\ \qquad \implies \;\; \exists h', \rho', C'.\, \llbracket \mathsf{p} \rrbracket_{\varepsilon} : \langle h, \rho, i \rangle^{\mu}_{C} \rightarrow \langle h', \rho', i' \rangle^{\mu'}_{C'} \;\wedge\; (h', \rho', C', \varepsilon) \in \mathcal{M}_{\pi''}(\hat{h}', \hat{\rho}', \hat{C}')\end{array}$$

Proof. We proceed by case analysis on $\langle \hat{h}, \hat{\rho}, i, \pi \rangle_{\hat{C}} \rightsquigarrow \langle \hat{h}', \hat{\rho}', j, \pi' \rangle_{\hat{C}'}$. Given the symbolic semantics for JSIL commands, we obtain that $\mu = \top$. Throughout the proof, we use the following facts:

[F1] If $\mathsf{cmd}(\mathsf{p}, \hat{C}, i) = \mathsf{C}$ and $\mathsf{svars}(\mathsf{p}) \subseteq \mathsf{dom}(\varepsilon)$, then it holds that $\llbracket \mathsf{p} \rrbracket_{\varepsilon}$, $\llbracket \hat{C} \rrbracket_{\varepsilon}$, and $\llbracket \mathsf{C} \rrbracket_{\varepsilon}$ are concrete, and $\mathsf{cmd}(\llbracket \mathsf{p} \rrbracket_{\varepsilon}, \llbracket \hat{C} \rrbracket_{\varepsilon}, i) = \llbracket \mathsf{C} \rrbracket_{\varepsilon}$.

[F2] If $(h, \rho, C, \varepsilon) \in \mathcal{M}_{\pi}(\hat{h}, \hat{\rho}, \hat{C})$, then it holds that $h = \llbracket \hat{h} \rrbracket_{\varepsilon}$, $\rho = \llbracket \hat{\rho} \rrbracket_{\varepsilon}$, $C = \llbracket \hat{C} \rrbracket_{\varepsilon}$, $\varepsilon \vdash \pi$, and also that $(h, \rho, \varepsilon) \in \mathcal{M}_{\pi}(\hat{h}, \hat{\rho})$

We first prove the cases for which $\mu' = \bot$. The only applicable rule is [Basic Command]. We conclude that $\mathsf{cmd}(\mathsf{p}, \hat{C}, i) = bc$ for some basic command bc, $\langle \hat{h}, \hat{\rho}, bc, \pi \rangle^{\top} \rightsquigarrow \langle \hat{h}, \hat{\rho}, \pi' \rangle^{\bot}$, $\hat{C}' = \hat{C}$, and $j = i + 1$. Since $\mathsf{svars}(\mathsf{p}) \subseteq \mathsf{dom}(\varepsilon)$, we conclude that $\mathsf{svars}(bc) \subseteq \mathsf{dom}(\varepsilon)$. Applying Lemma C.4 to:

$$\langle \hat{h}, \hat{\rho}, bc, \pi \rangle^{\top} \rightsquigarrow \langle \hat{h}, \hat{\rho}, \pi' \rangle^{\bot} \qquad (h, \rho) \in \mathcal{M}_{\pi''}(\hat{h}, \hat{\rho})\ \textbf{[F1]} \qquad \pi'' \vdash \pi' \qquad \mathsf{svars}(bc) \subseteq \mathsf{dom}(\varepsilon)$$

we conclude that there is a concrete heap h' and store ρ' such that:

$$\langle h, \rho, \llbracket bc \rrbracket_{\varepsilon} \rangle^{\top} \rightarrow \langle h', \rho' \rangle^{\bot} \qquad (h', \rho') \in \mathcal{M}_{\pi''}(\hat{h}, \hat{\rho})$$

From the concrete semantics of JSIL basic commands, we obtain that $h = h'$ and $\rho = \rho'$. Using **[F1]** and $\langle h, \rho, \llbracket bc \rrbracket_{\varepsilon} \rangle^{\top} \rightarrow \langle h, \rho \rangle^{\bot}$, we can apply the [Basic Command] rule of JSIL semantics to conclude that: $\llbracket \mathsf{p} \rrbracket_{\varepsilon} : \langle h, \rho, i \rangle^{\top}_{C} \rightarrow \langle h, \rho, i + 1 \rangle^{\bot}_{C'}$. Since $h' = h$, $\rho' = \rho$, $\hat{h}' = \hat{h}$, $\hat{\rho}' = \hat{\rho}$, $\hat{C}' = \hat{C}$, from **[F1]** we obtain that $C' = C$ and, from that, we obtain that $(h', \rho', C', \varepsilon) \in \mathcal{M}_{\pi''}(\hat{h}', \hat{\rho}', \hat{C}')$, concluding the proof.

Onward, we assume that $\mu' = \top$ and elide the modes from the rules.

[Basic Command] We conclude that $\mathsf{cmd}(\mathsf{p}, \hat{C}, i) = bc$ for some basic command bc, $\langle \hat{h}, \hat{\rho}, bc, \pi \rangle \rightsquigarrow \langle \hat{h}', \hat{\rho}', \pi' \rangle$, $\hat{C}' = \hat{C}$, and $j = i + 1$. Since $\mathsf{svars}(\mathsf{p}) \subseteq \mathsf{dom}(\varepsilon)$, we conclude that $\mathsf{svars}(bc) \subseteq \mathsf{dom}(\varepsilon)$. Applying Lemma C.4 to:

$$\langle \hat{h}, \hat{\rho}, bc, \pi \rangle \rightsquigarrow \langle \hat{h}', \hat{\rho}', \pi' \rangle \qquad (h, \rho) \in \mathcal{M}_{\pi''}(\hat{h}, \hat{\rho})\ \textbf{[F1]} \qquad \pi'' \vdash \pi' \qquad \mathsf{svars}(bc) \subseteq \mathsf{dom}(\varepsilon)$$

we conclude that there is a concrete heap h' and store ρ' such that:

$$\langle h, \rho, \llbracket bc \rrbracket_{\varepsilon} \rangle \rightarrow \langle h', \rho' \rangle \qquad (h', \rho') \in \mathcal{M}_{\pi}(\hat{h}', \hat{\rho}')$$

Using **[F1]** and $\langle h, \rho, \llbracket bc \rrbracket_{\varepsilon} \rangle \rightarrow \langle h', \rho' \rangle$, we can apply the [Basic Command] rule of JSIL semantics to conclude that: $\llbracket \mathsf{p} \rrbracket_{\varepsilon} : \langle h, \rho, i \rangle_{C} \rightarrow \langle h', \rho', i + 1 \rangle_{C'}$. From $\hat{C}' = \hat{C}$, $(h', \rho', \varepsilon) \in \mathcal{M}_{\pi''}(\hat{h}', \hat{\rho}')$, and $(h, \rho, C, \varepsilon) \in \mathcal{M}_{\pi''}(\hat{h}, \hat{\rho}, \hat{C})$, it follows that $(h', \rho', C', \varepsilon) \in \mathcal{M}_{\pi''}(\hat{h}, \hat{\rho}, \hat{C})$, which concludes the proof.

[Goto] We conclude that $\mathsf{cmd}(\mathsf{p}, \hat{C}, i) = \mathsf{goto}\ k$ for some program index k and that $\langle \hat{h}, \hat{\rho}, i, \pi \rangle_{\hat{C}} \rightsquigarrow \langle \hat{h}, \hat{\rho}, k, \pi \rangle_{\hat{C}}$, meaning that $\hat{h}' = \hat{h}$, $\hat{\rho}' = \hat{\rho}$, $\hat{C}' = \hat{C}$, $j = k$, and $\pi' = \pi$. Given **[F1]**, we conclude that $\llbracket \mathsf{p} \rrbracket_{\varepsilon} : \langle h, \rho, i \rangle_{C} \rightarrow \langle h, \rho, k \rangle_{C}$. Noting that:

$$(h', \rho', C', \varepsilon) = (h, \rho, C, \varepsilon) \in \mathcal{M}_{\pi''}(\hat{h}, \hat{\rho}, \hat{C}) = \mathcal{M}_{\pi''}(\hat{h}', \hat{\rho}', \hat{C}')$$

the result follows.

[Cond. Goto - True] We conclude that $\mathrm{cmd}(\mathsf{p}, \hat{C}, i) = \mathtt{goto}\ [\hat{e}]\ k_1,\ k_2$ for some extended expression ê and indexes k_1 and k_2 and that $\langle \hat{h}, \hat{\rho}, i, \pi\rangle_{\hat{C}} \rightsquigarrow \langle \hat{h}, \hat{\rho}, k_1, \pi \wedge \llbracket\hat{e}\rrbracket_{\hat{\rho}}\rangle_{\hat{C}}$, meaning that $\hat{h}' = \hat{h}$, $\hat{\rho}' = \hat{\rho}$, $\hat{C}' = \hat{C}$, $j = k$, and $\pi' = \pi \wedge \llbracket\hat{\mathrm{e}}\rrbracket_{\hat{\rho}}$. Given $\pi'' \vdash \pi'$ and **[F2]**, we conclude that $\varepsilon \vdash \pi'$. From $\varepsilon \vdash \pi' = (\pi \wedge \llbracket\hat{\mathrm{e}}\rrbracket_{\hat{\rho}})$, it follows that $\varepsilon \vdash \llbracket\hat{\mathrm{e}}\rrbracket_{\hat{\rho}}$, implying that $\llbracket\llbracket\hat{\mathrm{e}}\rrbracket_{\hat{\rho}}\rrbracket_\varepsilon = \mathrm{true}$. Letting $e = \llbracket\hat{\mathrm{e}}\rrbracket_\varepsilon$, from **[F1]**, we obtain that $\mathrm{cmd}(\llbracket\mathsf{p}\rrbracket_\varepsilon, \llbracket\hat{C}\rrbracket_\varepsilon, i) = \mathtt{goto}\ [e]\ k_1,\ k_2$. Noting that:

$$\llbracket e\rrbracket_\rho = \llbracket e\rrbracket_{\llbracket\hat{\rho}\rrbracket_\varepsilon} = \llbracket\llbracket\hat{\mathrm{e}}\rrbracket_\varepsilon\rrbracket_{\llbracket\hat{\rho}\rrbracket_\varepsilon} = \llbracket\llbracket\hat{\mathrm{e}}\rrbracket_{\hat{\rho}}\rrbracket_\varepsilon = \mathsf{true}$$

we conclude that $\llbracket\mathsf{p}\rrbracket_\varepsilon : \langle h, \rho, i\rangle_C \rightarrow \langle h, \rho, k_1\rangle_C$. Noting that:

$$(h', \rho', C', \varepsilon) = (h, \rho, C, \varepsilon) \in \mathcal{M}_{\pi''}(\hat{h}, \hat{\rho}, \hat{C}) = \mathcal{M}_{\pi''}(\hat{h}', \hat{\rho}', \hat{C}')$$

the result follows. The [Cond. Goto - False] case is analogous.

[Procedure Call] We conclude that $\mathrm{cmd}(\mathsf{p}, \hat{C}, i) = x := \hat{\mathrm{e}}(\hat{\mathrm{e}}_i|_{i=0}^{n})\ \mathtt{with}\ j$ for some variable x, extended expressions ê, ê$_0$, ..., ê$_n$, and index j, and $\langle \hat{h}, \hat{\rho}, i, \pi\rangle_{\hat{C}} \rightsquigarrow \langle \hat{h}, \hat{\rho}', 0, \pi\rangle_{\hat{C}'}$ where:

$$\hat{h}' = \hat{h} \qquad \hat{\rho}' = [x_i \mapsto \hat{e}_i\ |_{i=0}^{m}] \qquad C' = ((f', \hat{\rho}, x, i+1, j) :: \hat{C}$$
$$f' = \llbracket\hat{\mathrm{e}}\rrbracket_{\hat{\rho}} \qquad [x_1, \ldots, x_m] = \mathsf{args}(f') \qquad \hat{e}_i = \begin{cases} \llbracket\hat{\mathrm{e}}_i\rrbracket_{\hat{\rho}} & \text{if } 0 \le i \le n \\ \text{undefined} & \text{otherwise} \end{cases}$$

Letting $e = \llbracket\hat{\mathrm{e}}\rrbracket_\varepsilon$, $e_i|_{i=0}^{n} = \llbracket\hat{\mathrm{e}}_i\rrbracket_\varepsilon$, $e'_i|_{i=0}^{m} = \begin{cases} e_i & \text{if } 0 \le i \le n \\ \text{undefined} & \text{otherwise} \end{cases}$, and given **[F1]**, we obtain that $\mathrm{cmd}(\llbracket\mathsf{p}\rrbracket_\varepsilon, \llbracket\hat{C}\rrbracket_\varepsilon, i) = x := e(e_i|_{i=0}^{n})\ \mathtt{with}\ j$. Given **[F2]**, and noting that:

$$\llbracket e\rrbracket_\rho = \llbracket e\rrbracket_{\llbracket\hat{\rho}\rrbracket_\varepsilon} = \llbracket\llbracket\hat{\mathrm{e}}\rrbracket_\varepsilon\rrbracket_{\llbracket\hat{\rho}\rrbracket_\varepsilon} = \llbracket\llbracket\hat{\mathrm{e}}\rrbracket_{\hat{\rho}}\rrbracket_\varepsilon = f' \qquad \llbracket e'_i\rrbracket_\rho = \llbracket e'_i\rrbracket_{\llbracket\hat{\rho}\rrbracket_\varepsilon} = \llbracket\hat{e}_i\rrbracket_\varepsilon$$

we conclude that $\llbracket\mathsf{p}\rrbracket_\varepsilon : \langle h, \rho, i\rangle_C \rightarrow \langle h, [x_i \mapsto \llbracket e'_i\rrbracket_\rho\ |_{i=0}^{m}], 0\rangle_{(f', \rho, x, i+1, j)::C}$. From:

$$\begin{aligned} & h' = h = \llbracket\hat{h}\rrbracket_\varepsilon = \llbracket\hat{h}'\rrbracket_\varepsilon \\ & \rho' = [x_i \mapsto \llbracket e'_i\rrbracket_\rho\ |_{i=0}^{m}] = \llbracket[x_i \mapsto \hat{e}_i\ |_{i=0}^{m}]\rrbracket_\varepsilon = \llbracket\hat{\rho}'\rrbracket_\varepsilon \\ & C' = (f', \rho, x, i+1, j) :: C = (f', \llbracket\hat{\rho}\rrbracket_\varepsilon, x, i+1, j) :: \llbracket\hat{C}\rrbracket_\varepsilon = \llbracket\hat{C}'\rrbracket_\varepsilon \end{aligned}$$

the result follows.

[Normal Return] We have that $i = i_{\mathrm{nm}}$, and

$$\mathsf{p} : \langle \hat{h}, \hat{\rho}, i_{\mathrm{nm}}, \pi\rangle_{\hat{C}} \rightsquigarrow \langle \hat{h}, \hat{\rho}'[x \mapsto \hat{e}], i, \pi\rangle_{\hat{C}'}$$

where: $\hat{C} = (-, \hat{\rho}', x, i, -) :: \hat{C}'$ and $\hat{e} = \hat{\rho}(\mathsf{ret})$. From $C = \llbracket\hat{C}\rrbracket_\varepsilon$ (obtained from **[F1]**) and $\hat{C} = (-, \hat{\rho}', x, i, -) :: \hat{C}'$, we conclude that: $C = (-, \llbracket\hat{\rho}'\rrbracket_\varepsilon, x, i, -) :: \llbracket\hat{C}'\rrbracket_\varepsilon$. Hence, from **[F2]** and $i = i_{\mathrm{nm}}$, we conclude that: $\llbracket\mathsf{p}\rrbracket_\varepsilon : \langle h, \rho, i_{\mathrm{nm}}\rangle_C \rightarrow \langle h, \llbracket\hat{\rho}'\rrbracket_\varepsilon[x \mapsto v], i\rangle_{\llbracket\hat{C}'\rrbracket_\varepsilon}$, where $v = \rho(\mathsf{ret}) = \llbracket\hat{\rho}\rrbracket_\varepsilon(\mathsf{ret})$. Noting that:

$$\begin{aligned} & h' = h = \llbracket\hat{h}\rrbracket_\varepsilon = \llbracket\hat{h}'\rrbracket_\varepsilon \\ & \rho' = \llbracket\hat{\rho}'\rrbracket_\varepsilon[x \mapsto v] = \llbracket\hat{\rho}'\rrbracket_\varepsilon[x \mapsto \llbracket\hat{\rho}\rrbracket_\varepsilon(\mathsf{ret})] = \llbracket\hat{\rho}'\rrbracket_\varepsilon[x \mapsto \llbracket\hat{\rho}(\mathsf{ret})\rrbracket_\varepsilon] = \\ & \qquad \llbracket\hat{\rho}'[x \mapsto \hat{\rho}(\mathsf{ret})]\rrbracket_\varepsilon = \llbracket\hat{\rho}'[x \mapsto \hat{e}]\rrbracket_\varepsilon \\ & C' = \llbracket\hat{C}'\rrbracket_\varepsilon \end{aligned}$$

the result follows. The [Error Return] case is analogous.

□

Lemma C.6 (Monotonicity of Path Condition).

$$\begin{aligned} & \forall\, \mathsf{p}, \hat{h}, \hat{\rho}, i, \pi, \mu, j, \hat{C}, \hat{h}', \hat{\rho}', j, \hat{C}', \pi', \mu'\ . \\ & \qquad \mathsf{p} : \langle \hat{h}, \hat{\rho}, i, \pi\rangle_{\hat{C}}^{\mu} \rightsquigarrow^* \langle \hat{h}', \hat{\rho}', j, \pi'\rangle_{\hat{C}'}^{\mu'} \implies \pi' \vdash \pi \end{aligned}$$

Proof. By induction on the length of the symbolic trace $\mathsf{p} : \langle \hat{h}, \hat{\rho}, i, \pi \rangle^{\mu}_{\hat{C}} \rightsquigarrow^{*} \langle \hat{h}', \hat{\rho}', j, \pi' \rangle^{\mu'}_{\hat{C}'}$. The proof becomes trivial once we've observed that one step of the symbolic execution either does not change the path condition or further constrains it through conjunction), i.e., the path condition is never relaxed by the symbolic execution. Therefore, all models of π' must also be the models of π. □

Lemma C.7 (Soundness of the JSIL symbolic execution).

$$
\begin{array}{l}
\forall\, \mathsf{p}, \hat{h}, \hat{\rho}, i, \pi, \hat{C}, \mu, \hat{h}', \hat{\rho}', j, \pi', \hat{C}', \mu', \pi'', h, \rho, C, \varepsilon \,. \\
\quad \mathsf{p} : \langle \hat{h}, \hat{\rho}, i, \pi \rangle^{\mu}_{\hat{C}} \rightsquigarrow^{*} \langle \hat{h}', \hat{\rho}', j, \pi' \rangle^{\mu'}_{\hat{C}'} \ \wedge\ \pi'' \vdash \pi' \ \wedge\ (h, \rho, C, \varepsilon) \in \mathcal{M}_{\pi''}(\hat{h}, \hat{\rho}, \hat{C}) \\
\qquad \Longrightarrow\ \exists h', \rho', C' \,.\, [\![\mathsf{p}]\!]_{\varepsilon} : \langle h, \rho, i \rangle^{\mu}_{C} \rightarrow^{*} \langle h', \rho', j \rangle^{\mu'}_{C'} \ \wedge\ (h', \rho', C') \in \mathcal{M}_{\pi''}(\hat{h}', \hat{\rho}', \hat{C}')
\end{array}
$$

Proof. Suppose $\mathsf{p} : \langle \hat{h}, \hat{\rho}, i, \pi \rangle^{\mu}_{\hat{C}} \rightsquigarrow^{*} \langle \hat{h}', \hat{\rho}', j, \pi' \rangle^{\mu'}_{\hat{C}'}$ (**H1**) and $(h, \rho, C, \varepsilon) \in \mathcal{M}_{\pi''}(\hat{h}, \hat{\rho}, \hat{C})$ (**H2**). We have to prove that there is a concrete heap h', store ρ', and context C' such that $[\![\mathsf{p}]\!]_{\varepsilon} : \langle h, \rho, i \rangle^{\mu}_{C} \rightarrow^{*} \langle h', \rho', j \rangle^{\mu'}_{C'}$ and $(h', \rho', C', \varepsilon) \in \mathcal{M}_{\pi''}(\hat{h}', \hat{\rho}', \hat{C}')$. We proceed by induction on the length of the symbolic trace

$$\langle \hat{h}, \hat{\rho}, i, \pi \rangle_{\hat{C}} \rightsquigarrow^{*} \langle \hat{h}', \hat{\rho}', j, \pi' \rangle_{\hat{C}'}.$$

Suppose the symbolic trace has length n.

[Base Case] $n = 0$. It follows that $\hat{h}' = \hat{h}$, $\hat{\rho}' = \hat{\rho}$, $\hat{C}' = \hat{C}$, $\pi' = \pi$, and $j = i$. Noting that $\langle h, \rho, i \rangle_{C} \rightarrow^{*} \langle h, \rho, i \rangle_{C}$ and $(h, \rho, C) \in \mathcal{M}_{\pi''}(\hat{h}, \hat{\rho}, \hat{C}) = \mathcal{M}_{\pi''}(\hat{h}', \hat{\rho}', \hat{C}')$, the result follows.

[Inductive Case] $n = m + 1$. It follows that there is a symbolic heap $\hat{h}''$, store $\hat{\rho}''$, context $\hat{C}''$, mode μ'', and path condition π''' such that:

$$\mathsf{p} : \langle \hat{h}, \hat{\rho}, i, \pi \rangle^{\mu}_{\hat{C}} \rightsquigarrow^{*} \langle \hat{h}'', \hat{\rho}'', k, \pi''' \rangle^{\mu''}_{\hat{C}''} \ \textbf{(I1)} \qquad \mathsf{p} : \langle \hat{h}'', \hat{\rho}'', k, \pi''' \rangle^{\mu''}_{\hat{C}''} \rightsquigarrow \langle \hat{h}', \hat{\rho}', j, \pi' \rangle^{\mu'}_{\hat{C}'} \ \textbf{(I2)}$$

Applying Lemma C.6 to **I2**, we conclude that $\pi' \vdash \pi'''$ (**I3**), from which it follows (recalling that $\pi'' \vdash \pi'$) that $\pi'' \vdash \pi'''$ (**I4**). Applying the induction hypothesis to **I1**, **H2**, and **I4**, and, we conclude that there is a heap h'', store ρ'', and context C'', such that:

$$[\![\mathsf{p}]\!]_{\varepsilon} : \langle h, \rho, i \rangle^{\mu}_{C} \rightarrow^{*} \langle h'', \rho'', k \rangle^{\mu''}_{C''} \ \textbf{(I5)} \quad (h'', \rho'', C'', \varepsilon) \in \mathcal{M}_{\pi''}(\hat{h}'', \hat{\rho}'', \hat{C}'') \ \textbf{(I6)}$$

Applying Lemma C.5 to **I2**, **I6**, and $\pi'' \vdash \pi'$, we conclude that there is a heap h', store ρ', and context C', such that:

$$[\![\mathsf{p}]\!]_{\varepsilon} : \langle h'', \rho'', k \rangle^{\mu''}_{C''} \rightarrow \langle h', \rho', j \rangle^{\mu'}_{C'} \ \textbf{(I7)} \quad (h', \rho', C', \varepsilon) \in \mathcal{M}_{\pi''}(\hat{h}', \hat{\rho}', \hat{C}') \ \textbf{(I8)}$$

Combining **I5** and **I7**, we conclude that $[\![\mathsf{p}]\!]_{\varepsilon} : \langle h, \rho, i \rangle^{\mu}_{C} \rightarrow^{*} \langle h', \rho', j \rangle^{\mu''}_{C'}$ (**I9**). Finally, equations **I8** and **I9** conclude the proof. □

2 (Theorem 2.2 - Soundness of the JSIL symbolic execution).

$$
\begin{array}{l}
\forall\, \mathsf{p}, \hat{h}, \hat{\rho}, i, \pi, \hat{C}, \mu, \hat{h}', \hat{\rho}', i', \pi', \hat{C}', \mu', h, \rho, C, \varepsilon \,. \\
\quad \mathsf{p} : \langle \hat{h}, \hat{\rho}, i, \pi \rangle^{\mu}_{\hat{C}} \rightsquigarrow^{*} \langle \hat{h}', \hat{\rho}', i', \pi' \rangle^{\mu'}_{\hat{C}'} \ \wedge\ (h, \rho, C, \varepsilon) \in \mathcal{M}_{\pi'}(\hat{h}, \hat{\rho}, \hat{C}) \\
\qquad \Rightarrow\ \exists h', \rho', C' \,.\, [\![\mathsf{p}]\!]_{\varepsilon} : \langle h, \rho, i \rangle^{\mu}_{C} \rightarrow^{*} \langle h', \rho', i' \rangle^{\mu'}_{C'} \ \wedge\ (h', \rho', C') \in \mathcal{M}_{\pi'}(\hat{h}', \hat{\rho}', \hat{C}')
\end{array}
$$

Proof. Immediate, from Lemma C.7, noting that $\pi' \vdash \pi'$. □

1 (Corollary 2.3 - Bug-finding).

$$
\begin{array}{l}
\forall\, \mathsf{p}, \hat{h}, \hat{\rho}, i, \pi, \hat{C}, \hat{h}', \hat{\rho}', j, \pi', \hat{C}' \,. \\
\quad \mathsf{p} : \langle \hat{h}, \hat{\rho}, i, \pi \rangle^{\top}_{\hat{C}} \rightsquigarrow^{*} \langle \hat{h}', \hat{\rho}', j, \pi' \rangle^{\bot}_{\hat{C}'} \\
\qquad \Longrightarrow\ \exists h, \rho, C, \varepsilon \,.\, (h, \rho, C, \varepsilon) \in \mathcal{M}_{\pi'}(\hat{h}, \hat{\rho}, \hat{C}) \ \wedge\ [\![\mathsf{p}]\!]_{\varepsilon} : \langle h, \rho, i \rangle^{\top}_{C} \rightarrow^{*} \langle \_, \_, \_ \rangle^{\bot}_{\_}.
\end{array}
$$

Proof. We suppose that $\mathsf{p} : \langle \hat{h}, \hat{\rho}, i, \pi \rangle^{\top}_{\hat{C}} \rightsquigarrow^{*} \langle \hat{h}', \hat{\rho}', j, \pi' \rangle^{\bot}_{\hat{C}'}$, and will first show that π' is satisfiable. Given the symbolic execution rules, the last rule that was applied in the symbolic execution of $\mathsf{p} : \langle \hat{h}, \hat{\rho}, i, \pi \rangle^{\top}_{\hat{C}} \rightsquigarrow^{*} \langle \hat{h}', \hat{\rho}', j, \pi' \rangle^{\bot}_{\hat{C}'}$ is the [Basic Command] rule combined with an [Assert - False] rule:

$$\textsc{Basic Command}\quad \frac{\mathsf{cmd}(j-1) = \mathsf{assert}\,(\hat{\mathsf{e}}) \qquad \mathsf{p} : \langle \hat{h}', \hat{\rho}', \mathsf{assert}\,(\hat{\mathsf{e}}), \pi'' \rangle^{\top} \rightsquigarrow \langle \hat{h}', \hat{\rho}', \pi' \rangle^{\bot}}{\mathsf{p} : \langle \hat{h}', \hat{\rho}', j-1, \pi'' \rangle^{\top} \rightsquigarrow \langle \hat{h}', \hat{\rho}', j, \pi' \rangle^{\bot}}$$

$$\textsc{Assert - False}\quad \frac{\pi' = \pi'' \land \neg[\![\hat{\mathsf{e}}]\!]_{\rho} \qquad \pi'\text{is satisfiable}}{\langle \hat{h}', \hat{\rho}', \mathsf{assert}\,(\hat{\mathsf{e}}), \pi'' \rangle^{\top} \rightsquigarrow \langle \hat{h}', \hat{\rho}', \pi' \rangle^{\bot}}$$

Therefore, π' is satisfiable. Now, since π' is satisfiable, we conclude that there exist a concrete heap h, store ρ, context C, and symbolic environment ε, such that $(h, \rho, C, \varepsilon) \in \mathcal{M}_{\pi'}(\hat{h}, \hat{\rho}, \hat{C})$. Then, using Theorem 2.2, we obtain the desired goal. □

2 (Corollary 2.4 - Verification).

$$\begin{array}{l}
\forall\, \mathsf{p}, \hat{h}, \hat{h}_1, \ldots, \hat{h}_n, \hat{\rho}, \hat{\rho}_1, \ldots, \hat{\rho}_n, \hat{C}, \hat{C}_1, \ldots, \hat{C}_n, i, j_1, \ldots, j_n, \pi, \pi_1, \ldots, \pi_n, \mu, \mu_1, \ldots, \mu_n \,. \\
\land_{k=1}^{n} \left(\mathsf{p} : \langle \hat{h}, \hat{\rho}, i, \pi \rangle^{\mu}_{\hat{C}} \rightsquigarrow^{*} \langle \hat{h}_k, \hat{\rho}_k, j_k, \pi_k \rangle^{\mu_k}_{\hat{C}_k} |_{k=1}^{n} \right) \ \land\ \pi \vdash \bigvee_{k=1}^{n} \pi_k \\
\quad \Rightarrow \Big(\forall h, \rho, C, \varepsilon \,.\, (h, \rho, C, \varepsilon) \in \mathcal{M}_{\pi}(\hat{h}, \hat{\rho}, \hat{C}) \\
\qquad \Rightarrow \exists k, h', \rho', C' \,.\, [\![\mathsf{p}]\!]_{\varepsilon} : \langle h, \rho, i \rangle^{\mu}_{C} \rightarrow^{*} \langle h', \rho', j_k \rangle^{\mu_k}_{C'} \ \land\ (h', \rho', C', \varepsilon) \in \mathcal{M}_{\pi_k}(\hat{h}_k, \hat{\rho}_k, \hat{C}_k) \Big)
\end{array}$$

Proof. Suppose that $(h, \rho, C, \varepsilon) \in \mathcal{M}_{\pi}(\hat{h}, \hat{\rho}, \hat{C})$. Then, since $\pi \vdash \bigvee_{k=1}^{n} \pi_k$, we conclude that there exists k, $1 \leq k \leq n$, such that: $(h, \rho, C, \varepsilon) \in \mathcal{M}_{\pi_k}(\hat{h}, \hat{\rho}, \hat{C})$. Observing that $\mathsf{p} : \langle \hat{h}, \hat{\rho}, i, \pi \rangle^{\mu}_{\hat{C}} \rightsquigarrow^{*} \langle \hat{h}_k, \hat{\rho}_k, j_k, \pi_k \rangle^{\mu'}_{\hat{C}_k}$, we can apply Theorem 2.2 to conclude the result. □

D PROOFS OF SECTION 4

JSIL Logic Assertions - Syntax and Semantics

$$\begin{array}{lll}
E \triangleq & \lambda \mid x \mid \hat{x} \mid \ominus E \mid E \oplus E & \text{Logical Expressions} \\
P_\pi \triangleq & \text{true} \mid \text{false} \mid \neg P_\pi \mid P_\pi \wedge P_\pi \mid P_\pi \vee P_\pi \mid E = E \mid E \leq E & \text{Pure Assertions} \\
P \triangleq & P_\pi \mid \mathsf{emp} \mid (E, E) \mapsto E \mid \exists \hat{x}.P \mid P * P \mid \mathsf{emptyFields}(E \mid E) & \text{Assertions}
\end{array}$$

$$\begin{array}{ll}
\emptyset, \rho, \varepsilon \models \text{true} & \Leftrightarrow \text{always} \\
\emptyset, \rho, \varepsilon \models \text{false} & \Leftrightarrow \text{never} \\
\emptyset, \rho, \varepsilon \models \neg P_\pi & \Leftrightarrow \emptyset, \rho, \epsilon \not\models P_\pi \\
\emptyset, \rho, \varepsilon \models P_\pi \wedge Q_\pi & \Leftrightarrow \emptyset, \rho, \varepsilon \models P_\pi \wedge \emptyset, \rho, \varepsilon \models Q_\pi \\
\emptyset, \rho, \varepsilon \models E_1 = E_2 & \Leftrightarrow \llbracket E_1 \rrbracket_{\rho,\varepsilon} = \llbracket E_2 \rrbracket_{\rho,\varepsilon} \\
\emptyset, \rho, \varepsilon \models E_1 \leq E_2 & \Leftrightarrow \llbracket E_1 \rrbracket_{\rho,\varepsilon} \leq \llbracket E_2 \rrbracket_{\rho,\varepsilon} \\
\underline{h}, \rho, \varepsilon \models \exists \hat{x}.P & \Leftrightarrow \exists \lambda \in \mathcal{L}it.\, \underline{h}, \rho, \varepsilon[\hat{x} \mapsto \lambda] \models P
\end{array}$$

$$\begin{array}{l}
\underline{h}, \rho, \varepsilon \models \mathsf{emp} \Leftrightarrow \underline{h} = \emptyset \\
\underline{h}, \rho, \varepsilon \models (E_1, E_2) \mapsto E_3 \\
\;\Leftrightarrow \underline{h} = (\llbracket E_1 \rrbracket_{\rho,\varepsilon}, \llbracket E_2 \rrbracket_{\rho,\varepsilon}) \mapsto \llbracket E_3 \rrbracket_{\rho,\varepsilon} \\
\underline{h}, \rho, \varepsilon \models P * Q \Leftrightarrow \exists \underline{h}_1, \underline{h}_2.\, \underline{h} = \underline{h}_1 \uplus \underline{h}_2 \\
\qquad \wedge \underline{h}_1, \rho, \varepsilon \models P \wedge \underline{h}_2, \rho, \varepsilon \models Q \\
\underline{h}, \rho, \varepsilon \models \mathsf{emptyFields}(E_1 \mid E_2) \\
\;\Leftrightarrow \underline{h} = \biguplus_{s \notin \{\llbracket E_2 \rrbracket_{\rho,\varepsilon}\}} ((\llbracket E_1 \rrbracket_{\rho,\varepsilon}, s) \mapsto \varnothing)
\end{array}$$

Lemma D.1 (CA-Unification: Success).

$$\pi \vdash \langle \hat{h}, \mathsf{c} :: \mathsf{C} \rangle \to_{CU} \langle \hat{h}_f, \mathsf{C}' \rangle \implies \exists \hat{h}'.\, \hat{h} = \hat{h}' \uplus \hat{h}_f \wedge \mathcal{M}_\pi(\hat{h}', \hat{\rho}) \subseteq \mathcal{M}_*(\mathsf{c})$$

Proof. We conclude from the hypothesis that $\mathsf{c} = (l, \hat{p}) \mapsto \hat{v}$ (**I1**), $\hat{h} = \hat{h}_f \uplus ((l, \hat{p}') \mapsto \hat{v}')$ (**I2**), and $\pi \vdash \hat{p} = \hat{p}' \wedge \hat{v} = \hat{v}'$ (**I3**). We pick $\hat{h}' = (l, \hat{p}') \mapsto \hat{v}')$ (**I4**). We have to prove that for all symbolic environments ε such that $\varepsilon \models \pi$, it holds that:

$$\begin{array}{ll}
\multicolumn{2}{l}{(\llbracket \hat{h}' \rrbracket_\varepsilon, \llbracket \hat{\rho} \rrbracket_\varepsilon) \in \mathcal{M}_*((l, \hat{p})) \mapsto \hat{v})} \\
\iff & (\llbracket ((l, \hat{p}') \mapsto \hat{v}') \rrbracket_\varepsilon, \llbracket \hat{\rho} \rrbracket_\varepsilon) \in \mathcal{M}_*((l, \hat{p})) \mapsto \hat{v}) \\
\iff & (l, \llbracket \hat{p}' \rrbracket_\varepsilon) \mapsto \llbracket \hat{v}' \rrbracket_\varepsilon, -, \varepsilon \models (l, \hat{p})) \mapsto \hat{v} \\
\iff & \llbracket \hat{p}' \rrbracket_\varepsilon = \llbracket \hat{p} \rrbracket_\varepsilon \wedge \llbracket \hat{v}' \rrbracket_\varepsilon = \llbracket \hat{v} \rrbracket_\varepsilon \\
\iff & \text{true (because } \varepsilon \models \pi = \hat{p} = \hat{p}' \wedge \hat{v} = \hat{v}')
\end{array}$$

□

Lemma D.2 (CA-Unification: Uniqueness of Solutions).

$$\pi \vdash \langle \hat{h}, \mathsf{c} :: \mathsf{C} \rangle \to_{CU} \langle \hat{h}_f, \mathsf{C} \rangle \implies \neg \exists \hat{h}'', \hat{h}'_f.\, \hat{h}_f = \hat{h}'' \uplus \hat{h}'_f \wedge \mathcal{M}_\pi(\hat{h}'', \hat{\rho}) \subseteq \mathcal{M}_*(\mathsf{c})$$

Proof. We prove the result by *contradiction*, there are two symbolic heaps $\hat{h}''$ and $\hat{h}'_f$ such that $\hat{h}_f = \hat{h}'' \uplus \hat{h}'_f$ (**I1**) and $\mathcal{M}_\pi(\hat{h}'', \hat{\rho}) \subseteq \mathcal{M}_*(\mathsf{c})$ (**I2**). Applying Lemma D.1 to the hypothesis, we conclude that there is a we conclude that there is a symbolic heap $\hat{h}'$ such that: $\hat{h} = \hat{h}' \uplus \hat{h}_f$ (**I3**) and $\mathcal{M}_\pi(\hat{h}', \hat{\rho}) \subseteq \mathcal{M}_*(\mathsf{c})$ (**I4**). Combining **I1** and **I3**, we conclude that: $\hat{h} = \hat{h}' \uplus \hat{h}'' \uplus \hat{h}'_f$ (**I5**). Letting $\mathsf{c} = (l, \hat{p}) \mapsto \hat{v}$, we conclude, from **I2** and **I4**, that for symbolic environments $\varepsilon \models \pi$, it holds that:

$$(\llbracket \hat{h}' \rrbracket_\varepsilon, \llbracket \hat{\rho} \rrbracket_\varepsilon) \in \mathcal{M}_*((l, \hat{p}) \mapsto \hat{v}) \qquad (\llbracket \hat{h}'' \rrbracket_\varepsilon, \llbracket \hat{\rho} \rrbracket_\varepsilon) \in \mathcal{M}_*((l, \hat{p}) \mapsto \hat{v})$$

Let us pick an arbitrary ε satisfying π. It must be the case that:

$$\hat{h}' = (\llbracket l \rrbracket_{\rho,\varepsilon}, \llbracket \hat{p} \rrbracket_{\rho,\varepsilon}) \mapsto \llbracket \hat{v} \rrbracket_{\rho,\varepsilon} = (l, \llbracket \hat{p}) \rrbracket_\varepsilon) \mapsto \llbracket \hat{v} \rrbracket_\varepsilon = \hat{h}'' \; \textbf{(I6)}$$

Observe that **I6** contradicts **I5**, from which the result follows. □

Lemma D.3 (CA-Unification: Failure).

$$\begin{array}{l}
\pi \vdash \langle \hat{h}, \mathsf{c} :: - \rangle \to_{CU} \mathsf{F}(\pi') \implies \\
\quad \left(\forall \hat{h}', \hat{h}_f.\, \hat{h} = \hat{h}' \uplus \hat{h}_f \implies \mathcal{M}_{\pi \wedge \pi'}(\hat{h}', \hat{\rho}) \cap \mathcal{M}_*(\mathsf{c}) = \emptyset \right)
\end{array}$$

Proof. Suppose $\pi \vdash \langle \hat{h}, \mathsf{c} :: - \rangle \to_{CU} \mathsf{F}(\pi')$ (**H1**) and $\hat{h} = \hat{h}' \uplus \hat{h}_f$ (**H2**), for $\hat{h}' = (l', \hat{p}') \mapsto \hat{v}'$. We have to show that: $\mathcal{M}_{\pi \wedge \pi'}(\hat{h}', \hat{\rho}) \cap \mathcal{M}_*(\mathsf{c}) = \emptyset$. We proceed by *contradiction* assuming that there is a symbolic environment $\varepsilon \models \pi \wedge \pi'$ (**I1**) such that $\hat{h}, -, \varepsilon \models \mathsf{c}$ (**I2**). Given that $\mathsf{c} = (l, \hat{p}) \mapsto \hat{v}$ for some location l and symbolic expressions $\hat{p}$ and $\hat{v}$, we conclude that:

$$\begin{aligned}
&[\![\hat{h}']\!]_\varepsilon, -, \varepsilon \models (l, \hat{p}) \mapsto \hat{v} \\
&\quad\iff \Big((l', [\![\hat{p}']\!]_\varepsilon) \mapsto [\![\hat{v}']\!]_\varepsilon\Big), -, \varepsilon \models (l, \hat{p}) \mapsto \hat{v} \\
&\quad\iff \Big((l', [\![\hat{p}']\!]_\varepsilon) \mapsto [\![\hat{v}']\!]_\varepsilon\Big) = (l, [\![\hat{p}]\!]_\varepsilon) \mapsto [\![\hat{v}]\!]_\varepsilon \\
&\quad\iff l' = l \wedge [\![\hat{p}']\!]_\varepsilon = [\![\hat{p}]\!]_\varepsilon \wedge [\![\hat{v}']\!]_\varepsilon = [\![\hat{v}]\!]_\varepsilon \ \textbf{(I3)}
\end{aligned}$$

Observing that $l = l'$, we conclude, from the definition of $\rightarrow_{CU}$, that: $\pi' \vdash \hat{p}' \neq \hat{p} \vee \hat{v}' \neq \hat{v}$ (**I4**). From Equations **I1** and **I4**, we conclude that: $[\![\hat{p}']\!]_\varepsilon \neq [\![\hat{p}]\!]_\varepsilon \vee [\![\hat{v}']\!]_\varepsilon \neq [\![\hat{v}]\!]_\varepsilon$ (**I5**). Observe that **I5** contradicts **I3**, from which the result follows. □

Lemma D.4 (Iterated Cell Unification).

$$\begin{aligned}
&\pi \vdash \langle \hat{h}, \mathsf{C} \rangle \rightarrow^*_{CU} \langle \hat{h}_f, \mathsf{C}' \rangle \wedge \mathsf{C} = \mathsf{C}'' \cdot \mathsf{C}' \\
&\quad \implies \exists \hat{h}'.\, \hat{h} = \hat{h}' \uplus \hat{h}_f \wedge \mathcal{M}_\pi(\hat{h}', \hat{\rho}) \subseteq \mathcal{M}_*(\mathsf{C}'')
\end{aligned}$$

Proof. We proceed by induction on the length of the derivation $\hat{\rho}, \pi, \theta \vdash \langle \hat{h}, \mathsf{C} \rangle \rightarrow^*_{CU} \langle \hat{h}_f, \mathsf{C}' \rangle$. We assume that C has at least one element.

[Base Case] The base case corresponds to a derivation of length 0. We conclude that $\mathsf{C} = \mathsf{C}''$, $\mathsf{C}' = \mathsf{emp}$, $\hat{h} = \hat{h}_f$, and $\hat{h}' = \emptyset$. Noting that $\emptyset, \hat{\rho}, - \models \mathsf{emp}$, the result follows.

[Recursive Case] We assume a derivation of length $n = k + 1$. We conclude that: $\mathsf{C}'' = \mathsf{c} :: \mathsf{C}'''$ (**I1**) for some C'''. It follows that $\pi \vdash \langle \hat{h}, (\mathsf{c} :: \mathsf{C}''') \cdot \mathsf{C}' \rangle \rightarrow_{CU} \langle \hat{h}'_f, (\mathsf{C}''' \cdot \mathsf{C}') \rangle$ (**I2**) and $\pi \vdash \langle \hat{h}'_f, (\mathsf{C}''' \cdot \mathsf{C}') \rangle \rightarrow^*_{CU} \langle \hat{h}_f, \mathsf{C}' \rangle$ (**I3**). Applying Lemma D.1 to **I2**, we conclude that there is a symbolic heap $\hat{h}''$ such that:

$$\hat{h} = \hat{h}'' \uplus \hat{h}'_f \ \textbf{(I4)} \qquad \mathcal{M}_\pi(\hat{h}'', \hat{\rho}) \subseteq \mathcal{M}_*(\mathsf{c}) \ \textbf{(I5)}$$

Applying the induction hypothesis to **I3**, we conclude that there is a symbolic heap $\hat{h}'''$ such that:

$$\hat{h}'_f = \hat{h}''' \uplus \hat{h}_f \ \textbf{(I6)} \qquad \mathcal{M}_\pi(\hat{h}''', \hat{\rho}) \subseteq \mathcal{M}_*(\mathsf{C}''') \ \textbf{(I7)}$$

If we pick $\hat{h}' = \hat{h}'' \uplus \hat{h}'''$, it follows that:

$$\begin{aligned}
&\hat{h}'' \uplus \hat{h}''' \uplus \hat{h}_f = \hat{h}'' \uplus \hat{h}'_f = \hat{h} \text{ from } \textbf{I4} \text{ and } \textbf{I6} \\
&\mathcal{M}_\pi(\hat{h}'' \uplus \hat{h}''', \hat{\rho}) \subseteq \mathcal{M}_*(\mathsf{c} :: \mathsf{C}''') = \mathcal{M}_*(\mathsf{C}'') \text{ from } \textbf{I6} \text{ and } \textbf{I8}
\end{aligned}$$

which concludes the result. □

Lemma D.5 (Separation Constraints). *For* $\pi = \mathcal{S}(\mathsf{EF})$, $\mathsf{pvars}(\mathsf{EF}) = \emptyset$, *and* $\mathsf{slocs}(\mathsf{EF}) = \emptyset$, *it holds that:* $\varepsilon \models \pi \iff \exists \underline{h}\,.\, \underline{h}, -, \varepsilon \models \mathsf{EF}$.

Proof. We proceed by induction on the size of $\mathsf{locs}(\mathsf{EF})$.

[Base Case] $|\mathsf{locs}(\mathsf{EF})| = 0$. We conclude that $\pi = \mathsf{true}$ and $\mathsf{EF} \equiv \mathsf{emp}$. We rewrite the equivalence as follows: $\varepsilon \models \mathsf{true} \iff \exists \underline{h}\,.\, \underline{h}, -, \varepsilon \models \mathsf{emp}$, from which the result follows (it suffices to pick $\underline{h} = \emptyset$).

[Inductive Case] $|\mathsf{locs}(\mathsf{EF})| = k + 1$. We conclude that $\mathsf{locs}(\mathsf{EF}) = \{l\} \uplus L$. We can therefore conclude that:

$$\begin{aligned}
&\mathsf{EF} = \mathsf{EF}\,|_L * \mathsf{EF}\,|_l \ \ \textbf{(I1)} \\
&\pi = \pi_l \wedge \pi' \ \textbf{(I2)} \\
&\mathsf{EF}\,|_l = \mathsf{emptyFields}(l \mid \hat{e}_d) * \circledast_{i=0}^{n} \Big((l, \hat{p}_i) \mapsto \varnothing\Big) \ \textbf{(I3)} \\
&\pi_l = (\wedge_{0 \leq i,j \leq n, i \neq j} \hat{p}_i \neq \hat{p}_j) \wedge \Big(\{\hat{p}_i \mid_{i=0}^{n}\} \subseteq \hat{e}_d\Big) \ \textbf{(I4)}
\end{aligned}$$

where $\pi' = \mathcal{S}(\mathsf{EF}\,|_L)$. It follows from the hypothesis that $\mathsf{pvars}(\mathsf{EF}\,|_L) = \emptyset$, and $\mathsf{slocs}(\mathsf{EF}\,|_L) = \emptyset$. Hence, applying the inductive hypothesis to $\mathcal{S}(\mathsf{EF}\,|_L)$, we conclude that:

$$\varepsilon \models \pi' \iff \exists \underline{h}\,.\,\underline{h}, -, \varepsilon \models \mathsf{EF}\,|_L \;\; \textbf{(I5)}$$

We now consider both directions of the equivalence separately.

- [Left-to-Right] We assume that $\varepsilon \models \pi$ **(I6)**. We conclude from **I2** that $\varepsilon \models \pi'$ **(I7)**, from which it follows, with **I5**, that there is an instrumented heap $\underline{h}$ such that: $\underline{h}, -, \varepsilon \models \mathsf{EF}\,|_L$ **(I8)**. From **I6** and **I2**, we conclude that $\varepsilon \models \pi_l$ **(I9)**, from which it follows, with **I3** and **I4**, that $\underline{h}', -, \varepsilon \models \mathsf{EF}\,|_l$ **(I10)** for:

  $$\underline{h}' = \biguplus_{i=0}^{n} \Big((l, [\![\hat{p}_i]\!]_\varepsilon) \mapsto \varnothing \Big) \uplus \biguplus_{s \notin [\![\hat{e}_d]\!]_\varepsilon} \Big((l, s) \mapsto \varnothing \Big)$$

  Because the concrete locations in $\mathsf{EF}\,|_l$ do not overlap with those in $\mathsf{EF}\,|_L$, we conclude, from **I8** and **I10**, that $\underline{h}\#\underline{h}'$ and $\underline{h} \uplus \underline{h}' \models \mathsf{EF}$.
- [Right-to-Left] We assume that there is an instrumented heap $\underline{h}$ such that $\underline{h}, -, \varepsilon \models \mathsf{EF}$ **(I11)**, from which it follows that there are two disjoint instrumented heaps $\underline{h}'$ and $\underline{h}''$ such that: $\underline{h} = \underline{h}' \uplus \underline{h}''$ **(I12)**, $\underline{h}', -, \varepsilon \models \mathsf{EF}\,|_l$ **(I13)**, and $\underline{h}'', -, \varepsilon \models \mathsf{EF}\,|_L$ **(I14)**. From **I5** and **I14**, we conclude that: $\varepsilon \models \pi'$ **(I15)**. From **I13**, we conclude that:

  $$\forall_{0 \leq i, j \leq n, i \neq j} [\![\hat{p}_i]\!]_\varepsilon \neq [\![\hat{p}_j]\!]_\varepsilon \;\wedge\; \{[\![\hat{p}_i]\!]_\varepsilon \,|_{i=0}^{n}\} \subseteq [\![\hat{e}_d]\!]_\varepsilon \; \textbf{(I16)}$$

  From **I16**, we conclude that $\varepsilon \models \pi_l$ **(I17)**. From **I2**, **I15** and **I17**, we conclude that $\varepsilon \models \pi$.

□

same assumptions of the previous lemma

Lemma D.6 (Negative Resource Unification).

$$(h, \rho) \in \mathcal{M}_\pi(\hat{h}, \hat{\rho}) \;\wedge\; \pi \vdash \mathcal{U}_{NR}(\hat{h}, \mathsf{EF}) \;\wedge\; \mathcal{S}(\mathsf{EF})$$
$$\implies \forall \varepsilon\,.\,\varepsilon \models \pi \implies \exists \underline{h}.\,\underline{h}, \rho, \varepsilon \models \mathsf{EF} \;\wedge\; h\#\underline{h}$$

Proof. We have to show that given a symbolic environment ε such that $\varepsilon \models \pi$ **(I1)**, we can construct an instrumented heap $\underline{h}$ such that $\underline{h}, \rho, \varepsilon \models \mathsf{EF}$ and $h\#\underline{h}$. Using Lemma D.5, we conclude that there is an instrumented heap $\underline{h}$ such that $\underline{h}, \hat{\rho}, \varepsilon \models \mathsf{EF}$ **(I2)**. We now have to prove that $h\#\underline{h}$. We proceed by induction on the size of EF.

[Base Case] We conclude that $\mathsf{EF} = \emptyset$; hence, we conclude, from **I2**, that $\underline{h} = \emptyset$ and the result holds.

[Inductive Case] We conclude that $\mathsf{EF} = \mathsf{ef} \uplus \mathsf{EF}'$ **(I3)**. We conclude, from **I2**, that there are two disjoint instrumented heaps $\underline{h}'$ and $\underline{h}''$ such that $\underline{h}', \hat{\rho}, \varepsilon \models \mathsf{ef}$ **(I4)** and $\underline{h}'', \hat{\rho}, \varepsilon \models \mathsf{EF}'$ **(I5)**. We conclude, from the definition of $\mathcal{U}_{NR}(\hat{h}, \mathsf{EF})$, that: $\mathcal{U}_{NR}(\hat{h}, \mathsf{EF}) \vdash \mathcal{U}_{NR}(\hat{h}, \mathsf{EF}')$ **(I6)**. From **I6** and the hypothesis, it follows that $\pi \vdash \mathcal{U}_{NR}(\hat{h}, \mathsf{EF}')$ **(I7)**. Justify EF'. Applying the induction hypothesis to $\pi \vdash \mathcal{U}_{NR}(\hat{h}, \mathsf{EF}') \wedge \mathcal{S}(\mathsf{EF}')$, $(h, \rho) \in \mathcal{M}_\pi(\hat{h}, \hat{\rho})$, $\varepsilon \models \pi$, and $\underline{h}'', \hat{\rho}, \varepsilon \models \mathsf{EF}'$, we conclude that: $\underline{h}''\#h$ **(I8)**. Now we have to prove that $\underline{h}'\#h$. We proceed by case analysis on ef.

- $\mathsf{ef} = (l, \hat{p}) \mapsto \varnothing$ for some location l and symbolic expression $\hat{p}$. There are two cases to consider: **(i)** $l \notin \mathsf{locs}(\hat{h})$, in which case it immediately holds that $h\#\underline{h}'$ and **(ii)** $l \in \mathsf{locs}(\hat{h})$. Let us now prove the result for **(ii)**. It follows from $\mathsf{ef} = (l, \hat{p}) \mapsto \varnothing$ that $\underline{h}' = (l, [\![\hat{p}]\!]_\varepsilon) \mapsto \varnothing$ **(I9)**. From **I3**, and the definition of $\mathcal{U}_{NR}(\hat{h}, \mathsf{EF})$, we conclude that:

  $$\mathcal{U}_{NR}(\hat{h}, \mathsf{EF}) \vdash \mathcal{U}_{NR}(\hat{h}, (l, \hat{p}) \mapsto \varnothing) = \hat{p} \notin \{\hat{p}_i \,|_{i=0}^{n}\} \; \textbf{(I10)}$$

for $\hat{h} = \hat{h}' \uplus \Big((l, \hat{p}_i) \mapsto \hat{v}_i\Big)\,|_{i=0}^{n}$ and $(l, -) \notin \mathsf{dom}(\hat{h}')$. From the hypothesis, **I1**, and **I10**, we conclude $\varepsilon \models \hat{p} \notin \{\hat{p}_i\,|_{i=0}^{n}\}$ **(I11)**, from which it follows that $(l, \llbracket \hat{p} \rrbracket_\varepsilon) \notin \mathsf{dom}(\llbracket \hat{h} \rrbracket_\varepsilon) = \mathsf{dom}(h)$ **(I12)**. From **I9** and **I12**, we conclude $h \# \underline{h}'$.

- ef = $\mathsf{emptyFields}(l \mid \hat{e}_d)$ for some location l and symbolic expression $\hat{e}_d$. There are two cases to consider: **(i)** $l \notin \mathsf{locs}(\hat{h})$, in which case it immediately holds that $h \# \underline{h}'$ and **(ii)** $l \in \mathsf{locs}(\hat{h})$. Let us now prove the result for **(ii)**. It follows from ef = $\mathsf{emptyFields}(l \mid \hat{e}_d)$ that $\underline{h}' = \uplus_{s \notin \llbracket \hat{e}_d \rrbracket_\varepsilon} \Big((l, s) \mapsto \varnothing\Big)$ **(I13)**. From **I3**, and the definition of $\mathcal{U}_{NR}(\hat{h}, \mathsf{EF})$, we conclude that:

$$\mathcal{U}_{NR}(\hat{h}, \mathsf{EF}) \vdash \mathcal{U}_{NR}(\hat{h}, \mathsf{emptyFields}(l \mid \hat{e}_d)) = \{\hat{p}_i\,|_{i=0}^{n}\} \subseteq \hat{e}_d \ \textbf{(I14)}$$

for $\hat{h} = \hat{h}' \uplus \Big((l, \hat{p}_i) \mapsto \hat{v}_i\Big)\,|_{i=0}^{n}$ and $(l, -) \notin \mathsf{dom}(\hat{h}')$. From the hypothesis, **I1**, and **I14**, we conclude $\varepsilon \models \{\hat{p}_i\,|_{i=0}^{n}\} \subseteq \hat{e}_d$ **(I15)**, from which it follows that $\mathsf{dom}(\llbracket \hat{h} \rrbracket_\varepsilon) = \mathsf{dom}(h) \subseteq \{(l, s) \mid s \in \llbracket \hat{e}_d \rrbracket_\varepsilon\}$ **(I16)**. From **I13**, we conclude that: $\mathsf{dom}(\underline{h}') \cap \{(l, s) \mid s \in \llbracket \hat{e}_d \rrbracket_\varepsilon\} = \emptyset$ **(I17)**. From **I16** and **I17**, it follows that $\mathsf{dom}(\underline{h}') \cap \mathsf{dom}(h) = \emptyset$.

□

Lemma D.7 (Expressions and store evaluation).

$$\forall E, \hat{\rho}, \varepsilon, \rho.\ \llbracket \hat{\rho} \rrbracket_\varepsilon = \rho \implies \llbracket E \rrbracket_{\rho, \varepsilon} = \llbracket \llbracket E \rrbracket_{\hat{\rho}} \rrbracket_{\emptyset, \varepsilon}.$$

Proof. By induction on the structure of E. The only case we will show is for $E = x$, the remaining cases are straightforward. We need to prove that:

$$\begin{aligned} \forall \hat{\rho}, \varepsilon, \rho.\ \llbracket \hat{\rho} \rrbracket_\varepsilon = \rho &\implies \llbracket x \rrbracket_{\rho, \varepsilon} = \llbracket \llbracket x \rrbracket_{\hat{\rho}} \rrbracket_{\emptyset, \varepsilon}, && \text{i.e.} \\ \forall \hat{\rho}, \varepsilon, \rho.\ \llbracket \hat{\rho} \rrbracket_\varepsilon = \rho &\implies \rho(x) = \llbracket \hat{\rho}(x) \rrbracket_{\emptyset, \varepsilon}, && \text{i.e.} \\ \forall \hat{\rho}, \varepsilon.\ (\llbracket \hat{\rho} \rrbracket_\varepsilon)(x) &= \llbracket \hat{\rho}(x) \rrbracket_{\emptyset, \varepsilon}, && \text{i.e.} \\ \forall \hat{\rho}, \varepsilon.\ (\llbracket \hat{\rho}(x) \rrbracket_\varepsilon) &= \llbracket \hat{\rho}(x) \rrbracket_{\emptyset, \varepsilon}, \end{aligned}$$

which trivially holds. □

Lemma D.8 (Symbolic Store and Satisfiability).

$$\forall P, \underline{h}, \hat{\rho}, \varepsilon.\ \underline{h}, \llbracket \hat{\rho} \rrbracket_\varepsilon, \varepsilon \models P \iff \underline{h}, \llbracket \emptyset \rrbracket_\varepsilon, \varepsilon \models \llbracket P \rrbracket_{\hat{\rho}}.$$

Proof. By induction on the structure of P. The base cases true, false, and emp are straightforward, as they contain no program variables and for them, we have that $\llbracket P \rrbracket_{\hat{\rho}} = P$. The cases in which we can apply the induction hypothesis, namely $\neg P$ $P_1 \wedge P_2$, $P_1 \vee P_2$, $\exists \hat{x}.P$, and $P_1 * P_2$, follow immediately from the definitions and the appropriate induction hypotheses. Let us now do the case when $P \equiv E_1 = E_2$ (the case when $P \equiv E_1 \leq E_2$ is analogous). We have to prove that:

$$\underline{h}, \llbracket \hat{\rho} \rrbracket_\varepsilon, \varepsilon \models E_1 = E_2 \iff \underline{h}, \llbracket \emptyset \rrbracket_\varepsilon, \varepsilon \models \llbracket E_1 \rrbracket_{\hat{\rho}} = \llbracket E_2 \rrbracket_{\hat{\rho}}.$$

By unfolding the satisfiability relation, this becomes:

$$\underline{h} = \emptyset \wedge \llbracket E_1 \rrbracket_{\llbracket \hat{\rho} \rrbracket_\varepsilon, \varepsilon} = \llbracket E_2 \rrbracket_{\llbracket \hat{\rho} \rrbracket_\varepsilon, \varepsilon} \iff \underline{h} = \emptyset \wedge \llbracket \llbracket E_1 \rrbracket_{\hat{\rho}} \rrbracket_{\emptyset, \varepsilon} = \llbracket \llbracket E_2 \rrbracket_{\hat{\rho}} \rrbracket_{\emptyset, \varepsilon},$$

which holds directly, given Lemma D.8.

We will also prove the case when $P \equiv (E_1, E_2) \mapsto E_3$ (the case when $P \equiv \mathsf{emptyFields}(E_1 \mid E_2)$ is analogous). We have to prove that:

$$\underline{h}, \llbracket \hat{\rho} \rrbracket_\varepsilon, \varepsilon \models (E_1, E_2) \mapsto E_3 \iff \underline{h}, \llbracket \emptyset \rrbracket_\varepsilon, \varepsilon \models (\llbracket E_1 \rrbracket_{\hat{\rho}}, \llbracket E_2 \rrbracket_{\hat{\rho}}) \mapsto \llbracket E_3 \rrbracket_{\hat{\rho}}).$$

By unfolding the satisfiability relation, this becomes:

$$\underline{h} = (\llbracket E_1 \rrbracket_{\llbracket \hat{\rho} \rrbracket_\varepsilon, \varepsilon}, \llbracket E_2 \rrbracket_{\llbracket \hat{\rho} \rrbracket_\varepsilon, \varepsilon}) \mapsto \llbracket E_3 \rrbracket_{\llbracket \hat{\rho} \rrbracket_\varepsilon, \varepsilon} \iff \underline{h} = (\llbracket \llbracket E_1 \rrbracket_{\hat{\rho}} \rrbracket_{\emptyset, \varepsilon}, \llbracket \llbracket E_1 \rrbracket_{\hat{\rho}} \rrbracket_{\emptyset, \varepsilon}) \mapsto \llbracket \llbracket E_1 \rrbracket_{\hat{\rho}} \rrbracket_{\emptyset, \varepsilon}$$

which, again, holds directly, given Lemma D.8. □

Lemma D.9 (Symbolic Store and Entailment, Extended Models). *We define extended models in the following way:*

$$\begin{aligned}\mathcal{M}^{+}_{\pi}(\hat{h},\hat{\rho}) &= \left\{(h,\rho,\varepsilon) \mid \llbracket \hat{h} \rrbracket_{\varepsilon} = h \ \wedge \ \llbracket \hat{\rho} \rrbracket_{\varepsilon} = \rho \ \wedge \ \varepsilon \models \pi \right\};\\ \mathcal{M}^{+}_{*}(P) &= \left\{(h,\rho,\varepsilon) \mid \exists \underline{h} \,.\, h = \lfloor \underline{h} \rfloor \ \wedge \ \underline{h},\rho,\varepsilon \models P \right\}.\end{aligned}$$

Given this definition, it holds that

$$\mathcal{M}^{+}_{\pi}(\hat{h},\hat{\rho}) \subseteq \mathcal{M}^{+}_{*}(P) \iff \mathcal{M}^{+}_{\pi}(\hat{h},\emptyset) \subseteq \mathcal{M}^{+}_{*}(\llbracket P \rrbracket_{\hat{\rho}}).$$

Proof. After unfolding the definitions, we obtain:

$$\begin{aligned}&\forall P, \hat{h}, \hat{\rho}, \varepsilon.\\ &\quad \left(\varepsilon \models \pi \implies \exists \underline{h}.\, \llbracket \hat{h} \rrbracket_{\varepsilon} = \lfloor \underline{h} \rfloor \ \wedge \ \underline{h}, \llbracket \hat{\rho} \rrbracket_{\varepsilon}, \varepsilon \models P\right)\\ &\qquad\qquad \iff\\ &\quad \left(\varepsilon \models \pi \implies \exists \underline{h},\, \llbracket \hat{h} \rrbracket_{\varepsilon} = \lfloor \underline{h} \rfloor \ \wedge \ \underline{h}, \llbracket \emptyset \rrbracket_{\varepsilon}, \varepsilon \models \llbracket P \rrbracket_{\hat{\rho}}\right).\end{aligned}$$

We will prove the left-to-right direction (the right-to-left direction is analogous). We have two hypotheses: $\varepsilon \models \pi \implies \exists \underline{h}.\, \llbracket \hat{h} \rrbracket_{\varepsilon} = \lfloor \underline{h} \rfloor \ \wedge \ \underline{h}, \llbracket \hat{\rho} \rrbracket_{\varepsilon}, \varepsilon \models P$ (**H1**); and $\varepsilon \models \pi$ (**H2**). Our goal is $\exists \underline{h},\, \llbracket \hat{h} \rrbracket_{\varepsilon} = \lfloor \underline{h} \rfloor \ \wedge \ \underline{h}, \llbracket \emptyset \rrbracket_{\varepsilon}, \varepsilon \models \llbracket P \rrbracket_{\hat{\rho}}$ (**G1**). From (**H2**) and (**H1**), we obtain $\underline{h}$, such that $\llbracket \hat{h} \rrbracket_{\varepsilon} = \lfloor \underline{h} \rfloor \ \wedge \ \underline{h}, \llbracket \hat{\rho} \rrbracket_{\varepsilon}, \varepsilon \models P$ (**I1**). From (**I1**) and Lemma D.8, we obtain that $\llbracket \hat{h} \rrbracket_{\varepsilon} = \lfloor \underline{h} \rfloor \ \wedge \ \underline{h}, \llbracket \emptyset \rrbracket_{\varepsilon}, \varepsilon \models \llbracket P \rrbracket_{\hat{\rho}}$, which gives us the witness for the goal (**G1**), namely $\underline{h}$. □

Lemma D.10 (Connecting Models and Extended Models).

$$\begin{aligned}&\forall \hat{h}, \hat{\rho}, \varepsilon, \pi, h, \rho.\ (h,\rho,\varepsilon) \in \mathcal{M}^{+}_{\pi}(\hat{h},\hat{\rho}) \implies (h,\rho) \in \mathcal{M}_{\pi}(\hat{h},\hat{\rho});\\ &\forall \hat{h}, \hat{\rho}, \pi, h, \rho.\ (h,\rho) \in \mathcal{M}_{\pi}(\hat{h},\hat{\rho}) \iff \exists \varepsilon.\ (h,\rho,\varepsilon) \in \mathcal{M}^{+}_{\pi}(\hat{h},\hat{\rho})\\ &\forall P, h, \rho, \varepsilon.\ (h,\rho,\varepsilon) \in \mathcal{M}^{+}_{*}(P) \implies (h,\rho) \in \mathcal{M}_{*}(P);\\ &\forall P, h, \rho.\ (h,\rho) \in \mathcal{M}_{*}(P) \iff \exists \varepsilon.\ (h,\rho,\varepsilon) \in \mathcal{M}^{+}_{*}(P).\end{aligned}$$

Proof. Directly from the appropriate definitions. □

Lemma D.11 (Symbolic Store and Entailment).

$$\mathcal{M}_{\pi}(\hat{h},\hat{\rho}) \subseteq \mathcal{M}_{*}(P) \iff \mathcal{M}_{\pi}(\hat{h},\emptyset) \subseteq \mathcal{M}_{*}(\llbracket P \rrbracket_{\hat{\rho}})$$

Proof. We will prove that $\mathcal{M}_{\pi}(\hat{h},\hat{\rho}) \subseteq \mathcal{M}_{*}(P) \iff \mathcal{M}^{+}_{\pi}(\hat{h},\hat{\rho}) \subseteq \mathcal{M}^{+}_{*}(P)$. Then, from that and Lemma D.9, the desired goal directly holds.

□

Proof. By induction on the structure of P. The base cases true, false, and emp are straightforward, as they contain no program variables and for them, we have that $\llbracket P \rrbracket_{\hat{\rho}} = P$. The cases in which we can apply the induction hypothesis, namely $\neg P$ $P_1 \wedge P_2$, $P_1 \vee P_2$, and $P_1 * P_2$, follow immediately from the definitions and the appropriate induction hypotheses.

Let us now do the case when $P \equiv E_1 = E_2$ (the case when $P \equiv E_1 \leq E_2$ is analogous). We have to prove that:

$$\forall \hat{h}, \hat{\rho}, h, \rho.\ (\exists \varepsilon.\ \llbracket \hat{h} \rrbracket_{\varepsilon} = h \ \wedge\ \llbracket \hat{\rho} \rrbracket_{\varepsilon} = \rho \ \wedge\ \varepsilon \models \pi) \implies (\exists \underline{h}, \varepsilon.\, h = \lfloor \underline{h} \rfloor \ \wedge\ \underline{h}, \rho, \varepsilon \models E_1 = E_2)$$
$$\Longleftrightarrow$$
$$\forall \hat{h}, \hat{\rho}, h, \rho.\ (\exists \varepsilon.\ \llbracket \hat{h} \rrbracket_{\varepsilon} = h \ \wedge\ \llbracket \emptyset \rrbracket_{\varepsilon} = \rho \ \wedge\ \varepsilon \models \pi) \implies (\exists \underline{h}, \varepsilon.\, h = \lfloor \underline{h} \rfloor \ \wedge\ \underline{h}, \rho, \varepsilon \models \llbracket E_1 \rrbracket_{\hat{\rho}} = \llbracket E_2 \rrbracket_{\hat{\rho}}).$$

Given the satisfiability relation for SL-assertions, we have that $\hat{h} = h = \underline{h} = \emptyset$ in both implications and also $\rho = \emptyset$ in the implication below:

$$\forall \hat{\rho}, \rho.\ (\exists \varepsilon\,.\ \llbracket \hat{\rho} \rrbracket_{\varepsilon} = \rho \ \wedge\ \varepsilon \models \pi) \implies (\exists \varepsilon.\, \emptyset, \rho, \varepsilon \models E_1 = E_2)$$
$$\Longleftrightarrow$$
$$\forall \hat{\rho}.\ (\exists \varepsilon\,.\,\varepsilon \models \pi) \implies (\exists \varepsilon\,.\,\emptyset, \emptyset, \varepsilon \models \llbracket E_1 \rrbracket_{\hat{\rho}} = \llbracket E_2 \rrbracket_{\hat{\rho}}).$$

Let us first do the left-to-right implication. The hypotheses we have are: $\forall \hat{\rho}, \rho.\ (\exists \varepsilon\,.\ \llbracket \hat{\rho} \rrbracket_{\varepsilon} = \rho \ \wedge\ \varepsilon \models \pi) \implies (\exists \varepsilon.\, \emptyset, \rho, \varepsilon \models E_1 = E_2)$ (**H1**) and $\exists \varepsilon\,.\,\varepsilon \models \pi$ (**H2**), whereas our goal is $\exists \varepsilon\,.\,\emptyset, \emptyset, \varepsilon \models \llbracket E_1 \rrbracket_{\hat{\rho}} = \llbracket E_2 \rrbracket_{\hat{\rho}}$ (**G1**), and $\hat{\rho}$ is arbitrary. From (**H2**), we obtain an ε, such that $\varepsilon \models \pi$. We instantiate (**H1**) with $\hat{\rho}$, $\llbracket \hat{\rho} \rrbracket_{\varepsilon}$, and ε, obtaining ε' such that $\emptyset, \llbracket \hat{\rho} \rrbracket_{\varepsilon}, \varepsilon' \models E_1 = E_2$, which means that $\llbracket E_1 \rrbracket_{\llbracket \hat{\rho} \rrbracket_{\varepsilon}, \varepsilon'} = \llbracket E_2 \rrbracket_{\llbracket \hat{\rho} \rrbracket_{\varepsilon}, \varepsilon'}$. We claim that the witness for the goal (**G1**) is ε', i.e. that $\emptyset, \emptyset, \varepsilon' \models \llbracket E_1 \rrbracket_{\hat{\rho}} = \llbracket E_2 \rrbracket_{\hat{\rho}}$, i.e. that $\llbracket \llbracket E_1 \rrbracket_{\hat{\rho}} \rrbracket_{\emptyset, \varepsilon'} = \llbracket \llbracket E_2 \rrbracket_{\hat{\rho}} \rrbracket_{\emptyset, \varepsilon'}$. Using the previous lemma, this is equivalent to $\llbracket E_1 \rrbracket_{\llbracket \hat{\rho} \rrbracket_{\varepsilon'}, \varepsilon'} = \llbracket E_2 \rrbracket_{\llbracket \hat{\rho} \rrbracket_{\varepsilon'}, \varepsilon'}$.

Next, cell assertion.

Next, empty fields. □

3 (Theorem 4.1 - Soundness of Unification). $\forall P, \mathsf{C}, \mathsf{EF}, \pi', \hat{h}, \hat{\rho}, \pi, \theta\,.$

$$\llbracket \theta(P) \rrbracket_{\hat{\rho}} \equiv (\emptyset, \mathsf{C}, \mathsf{EF}, \pi') \implies \langle (\hat{h}, \pi), (\mathsf{C}, \mathsf{EF}, \pi') \rangle \in \mathcal{U}_{\mathsf{S}} \implies \mathcal{M}_{\pi}(\hat{h}, \hat{\rho}) \subseteq \mathcal{M}_{*}(P)$$

Proof. It must be the case that:

$$\pi \vdash \langle \hat{h}, \mathsf{C} \rangle \rightarrow^{*}_{CU} \langle \emptyset, [] \rangle \ (\mathbf{I1})$$
$$\pi \vdash \pi' \ \wedge\ \mathcal{S}(\mathsf{EF}) \ \wedge\ \mathcal{U}_{NR}(\hat{h}, \mathsf{EF}) \ (\mathbf{I2})$$

(1) We start by proving that $\mathcal{M}_{\pi}(\hat{h}, \hat{\rho}) \subseteq \mathcal{M}_{*}(\mathsf{C} \wedge \pi)$. Applying the Iterated-Cell-Unification Lemma (Lemma D.4) to **I1**, we conclude that:

$$\mathcal{M}_{\pi}(\hat{h}, \hat{\rho}) \subseteq \mathcal{M}_{*}(\mathsf{C}) \ (\mathbf{I2})$$

From **I2**, noting that $\mathcal{M}_{\pi}(\hat{h}, \hat{\rho}) \subseteq \mathcal{M}_{*}(\pi)$, we conclude that: this needs to be changed because now we use the star

$$\mathcal{M}_{\pi}(\hat{h}, \hat{\rho}) \subseteq \mathcal{M}_{*}(\mathsf{C} \wedge \pi) \ (\mathbf{I3})$$

(2) We will now prove that $\mathcal{M}_{\pi}(\hat{h}, \hat{\rho}) \subseteq \mathcal{M}_{*}(\mathsf{C} * \mathsf{EF})$. To this end, let us assume that $(h, \rho) \in \mathcal{M}_{\pi}(\hat{h}, \hat{\rho})$ (**I4**). From **I4**, we conclude that there is a symbolic environment ε such that $\varepsilon \models \pi$ (**I5**), $h = \llbracket \hat{h} \rrbracket_{\varepsilon}$ (**I6**), and $\rho = \llbracket \hat{\rho} \rrbracket_{\varepsilon}$ (**I6**). We will now prove that there is an instrumented heap $\underline{h}$ and ε' such that: $\underline{h}, \rho, \varepsilon' \models \mathsf{C} * \mathsf{EF}$ and $\lfloor \underline{h} \rfloor = h$. From **I3** and **I4**, we conclude that there is a symbolic environment ε' and an instrumented heap $\underline{h}$ such that: $\underline{h}, \rho, \varepsilon' \models \mathsf{C} * \pi$ (**I7**). From **I7**, it follows that: $\underline{h}, \rho, \varepsilon' \models \mathsf{C}$ (**I8**) and $\varepsilon' \models \pi$ (**I9**). Applying Lemma D.6 to **I2**, **I4**, and **I9**, we conclude that there is an instrumented heap $\underline{h}'$ such that: $\underline{h}', -, \varepsilon' \models \mathsf{EF}$ (**I10**) and $h \# \underline{h}'$ (**I11**). Since C does not have a negative footprint, we conclude that $\mathrm{dom}(\underline{h}) = \mathrm{dom}(h)$ (**I12**). From

I11 and **I12**, $\underline{h} \# \underline{h}'$ (**I13**). From **I8**, **I10**, and **I13**, it follows that: $\underline{h} \uplus \underline{h}', \rho, \varepsilon' \models \mathsf{C} * \mathsf{EF}$ (**I14**) which concludes the proof that: $\mathcal{M}_\pi(\hat{h}, \hat{\rho}) \subseteq \mathcal{M}_*(\mathsf{C} * \mathsf{EF})$ (**I15**).

(3) From **I15**, we conclude that $\mathcal{M}_\pi(\hat{h}, \hat{\rho}) \subseteq \mathcal{M}_*(\mathsf{C} * \mathsf{EF} \wedge \pi)$ (**I16**). From **I2** and **I16**, we conclude that:

$$\mathcal{M}_\pi(\hat{h}, \hat{\rho}) \subseteq \mathcal{M}_*(\mathsf{C} * \mathsf{EF}) \qquad \mathcal{M}_\pi(\hat{h}, \hat{\rho}) \subseteq \mathcal{M}_*(\pi')$$

from which it follows that:

$$\mathcal{M}_\pi(\hat{h}, \hat{\rho}) \quad \subseteq \quad \mathcal{M}_*(\mathsf{C} * \mathsf{EF}) \ \cap \ \mathcal{M}_*(\pi') = \mathcal{M}_*(\mathsf{C} * \mathsf{EF} \ \wedge \ \pi') \ (\mathbf{I17})$$

Applying Lemma D.11 to **I17**, we conclude that $\mathcal{M}_\pi(\hat{h}, \hat{\rho}) \subseteq \mathcal{M}_*(\theta(P))$ (**I18**). Observing that the domain of θ coincides with the logical variables existentially quantified ($\mathsf{dom}(\theta) = L$), it follows: $\mathcal{M}_\pi(\hat{h}, \hat{\rho}) \subseteq \mathcal{M}_*(P)$, which concludes the proof.

□

Lemma D.12 (No double unification).

$$\hat{h}, \pi \models \mathsf{c}_1 \ \wedge \ \hat{h}, \pi \models \mathsf{c}_2 \implies \mathcal{M}_*(\mathsf{c}_1 * \mathsf{c}_2 * \pi) = \emptyset$$

for $\mathsf{pvars}(\mathsf{EF}) = \emptyset$ *and* $\mathsf{slocs}(\mathsf{EF}) = \emptyset$.

Proof. Given that $\mathsf{pvars}(\mathsf{EF}) = \emptyset$ and $\mathsf{slocs}(\mathsf{EF}) = \emptyset$, we can assume that there are symbolic expressions $\hat{p}, \hat{p}_1, \hat{p}_2, \hat{v}_1, \hat{v}$, and $\hat{v}_2$ and a concrete location l, such that:

$$\hat{h} = (l, \hat{p}) \mapsto \hat{v} \ (\mathbf{I1}) \qquad \mathsf{c}_1 = (l, \hat{p}_1) \mapsto \hat{v}_1 \ (\mathbf{I2}) \qquad \mathsf{c}_2 = (l, \hat{p}_2) \mapsto \hat{v}_2 \ (\mathbf{I3})$$

From **I2** and **I3**, we conclude that: $\pi \vdash \hat{p}_1 = \hat{p}_2 \ \wedge \ \hat{v}_1 = \hat{v}_2$ (**I4**), from which the result follows immediately. □

Lemma D.13 (Separation Constraints: Failure).

$$\forall \mathsf{EF} \,.\, \mathsf{pvars}(\mathsf{EF}) = \emptyset \implies \forall \underline{h}, \varepsilon, \mathsf{EF} \,.\, \varepsilon \models \neg \mathcal{S}(\mathsf{EF}) \implies \underline{h}, \varepsilon \not\models \mathsf{EF}$$

Proof. Assuming that $\mathsf{pvars}(\mathsf{EF}) = \emptyset$ (**H1**) and $\varepsilon \models \neg \mathcal{S}(\mathsf{EF})$ (**H2**), we have to prove that $\underline{h}, \varepsilon \not\models \mathsf{EF}$ for an arbitrary instrumented heap $\underline{h}$. Given that:

$$\begin{aligned} \varepsilon \models \neg \mathcal{S}(\mathsf{EF}) \quad & \iff \varepsilon \models \neg \Big(\bigwedge\nolimits_{l \in \mathsf{locs}(\mathsf{EF})} \mathcal{S}_l(\mathsf{EF}) \Big) \\ & \iff \varepsilon \models \bigvee\nolimits_{l \in \mathsf{locs}(\mathsf{EF})} \neg \mathcal{S}_l(\mathsf{EF}) \end{aligned}$$

we conclude that there is a concrete location $l \in \mathsf{locs}(\mathsf{EF})$ such that $\varepsilon \models \neg \mathcal{S}_l(\mathsf{EF})$ (**I1**). From **I1**, we conclude that:

$$\begin{aligned} & \varepsilon \models \neg \Big((\wedge_{0 \le i,j \le n, i \ne j} \hat{p}_i \ne \hat{p}_j) \ \wedge \ (\wedge_{0 \le i \le n} \hat{p}_i \in \hat{e}_d) \Big) \\ \iff & \ \varepsilon \models (\vee_{0 \le i,j \le n, i \ne j} \hat{p}_i = \hat{p}_j) \ \vee \ (\vee_{0 \le i \le n} \hat{p}_i \notin \hat{e}_d) \Big) \end{aligned}$$

where: $\mathsf{EF}\,|_l = \mathsf{emptyFields}(l \mid \hat{e}_d) * \circledast_{i=0}^{n} \Big((l, \hat{p}_i) \mapsto \varnothing \Big)$. From the above, we conclude that either: **(1)** there are two integers i and j such that $i \ne j$ and $\varepsilon \models \hat{p}_i = \hat{p}_j$ or **(1)** there is an integer i such that $\varepsilon \models \hat{p}_i \notin \hat{e}_d$. We proceed by case analysis.

- Suppose that $\varepsilon \models \hat{p}_i = \hat{p}_j$, we conclude that there is no instrumented heap $\underline{h}$, such that $\underline{h}, \varepsilon \models (l, \hat{p}_i) \mapsto \varnothing * (l, \hat{p}_j) \mapsto \varnothing$, from which it follows that there is no instrumented heap $\underline{h}$, such that $\underline{h}, \varepsilon \models \mathsf{EF}$, which concludes the result.
- Suppose that $\varepsilon \models \hat{p}_i \notin \hat{e}_d$, we conclude that there is no instrumented heap $\underline{h}$, such that $\underline{h}, \varepsilon \models \mathsf{emptyFields}(l \mid \hat{e}_d) * (l, \hat{p}_i) \mapsto \varnothing$, from which it follows that there is no instrumented heap $\underline{h}$, such that $\underline{h}, \varepsilon \models \mathsf{EF}$, which concludes the result.

□

Lemma D.14 (Separation Constraints: Failure).

$$\forall h, \rho, \varepsilon, \pi, \hat{h}, \hat{\rho}, \mathsf{EF}\,. \\ (h, \rho, \varepsilon) \in \mathcal{M}_{\pi}(\hat{h}, \hat{\rho}) \;\wedge\; \varepsilon \models \neg\mathcal{U}_{NR}(\hat{h}, \mathsf{EF}) \;\wedge\; [\![\hat{h}]\!]_{\varepsilon} \# \underline{h} \implies \underline{h}, \varepsilon \not\models \mathsf{EF}$$

Proof. We assume that: $(h, \rho, \varepsilon) \in \mathcal{M}_{\pi}(\hat{h}, \hat{\rho})$ (**H1**), $\varepsilon \models \neg\mathcal{U}_{NR}(\hat{h}, \mathsf{EF})$ (**H2**), and $[\![\hat{h}]\!]_{\varepsilon} \# \underline{h}$ (**H3**); and we have to prove that $\underline{h}, \varepsilon \not\models \mathsf{EF}$. We prove the result by contradiction, assuming that $\underline{h}, \varepsilon \models \mathsf{EF}$ (**H4**). Expanding **H2**, we obtain:

$$\begin{aligned} \varepsilon \models \neg\mathcal{U}_{NR}(\hat{h}, \mathsf{EF}) &\iff \varepsilon \models \neg \textstyle\bigwedge_{\mathrm{ef} \in \mathsf{EF}} \mathcal{U}_{NR}(\hat{h}, \mathrm{ef}) \\ &\iff \varepsilon \models \textstyle\bigvee_{\mathrm{ef} \in \mathsf{EF}} \neg\mathcal{U}_{NR}(\hat{h}, \mathrm{ef}) \\ &\iff \exists_{\mathrm{ef} \in \mathsf{EF}}\, \varepsilon \models \neg\mathcal{U}_{NR}(\hat{h}, \mathrm{ef}) \end{aligned}$$

Hence, we conclude that there is an NR-assertion ef $\in \mathsf{EF}$ such that $\varepsilon \models \neg\mathcal{U}_{NR}(\hat{h}, \mathrm{ef})$ (**I1**). We proceed by case analysis on ef.

- ef $=$ $\mathsf{emptyFields}(l \mid \hat{e}_d)$ (**I2**) for some concrete location l and symbolic expression $\hat{e}_d$. It follows from **I1** and **I2** that: $\hat{h} = \hat{h}' \uplus \big((l, \hat{p}_i) \mapsto -\big) \mid_{i=0}^{n}$ (**I3**), $l \notin \mathsf{locs}(\hat{h}')$ (**I4**), and $\varepsilon \models \{\hat{p}_i \mid_{i=0}^{n}\} \nsubseteq \hat{e}_d$ (**I5**). From **H4** and **I2**, it follows that $\{(l, s) \mid s \notin [\![\hat{e}_d]\!]_{\varepsilon}\} \subseteq \mathsf{dom}(\underline{h})$ (**I6**). From **I3** and **I4**, we conclude that: $\{(l, [\![\hat{p}_i]\!]_{\varepsilon}) \mid_{i=0}^{n}\} \subseteq \mathsf{dom}([\![\hat{h}]\!]_{\varepsilon})$ (**I7**). From **I5**-**I7**, we conclude that $\mathsf{dom}([\![\hat{h}]\!]_{\varepsilon}) \cap \mathsf{dom}(\underline{h}) \neq \emptyset$, which contradicts **H3**.
- ef $= (l, \hat{p}) \mapsto \varnothing$ (**I8**) for some concrete location l and symbolic expression $\hat{p}$. It follows from **I1** and **I2** that: $\hat{h} = \hat{h}' \uplus \big((l, \hat{p}_i) \mapsto -\big) \mid_{i=0}^{n}$ (**I9**), $l \notin \mathsf{locs}(\hat{h}')$ (**I10**), and $\varepsilon \models \neg \wedge_{0 \leq i \leq n} \hat{p} \neq \hat{p}_i$ (**I11**). We conclude that there is an i such that: $\varepsilon \models \hat{p} = \hat{p}_i$ (**I12**). From **H4** and **I8**, it follows that $(l, [\![\hat{p}]\!]_{\varepsilon}) \in \mathsf{dom}(\underline{h})$ (**I13**). From **I9**, it follows that $(l, [\![\hat{p}_i]\!]_{\varepsilon}) \in \mathsf{dom}([\![\hat{h}]\!]_{\varepsilon})$ (**I14**). From **I12**-**I14**, we conclude that $\mathsf{dom}([\![\hat{h}]\!]_{\varepsilon}) \cap \mathsf{dom}(\underline{h}) \neq \emptyset$, which contradicts **H3**.

□

Lemma D.15 (Counter Models for Failed Grounded Entailement).

$$\langle(\hat{h}, \pi), (\mathsf{C}, \mathsf{EF}, \pi')\rangle \in \mathcal{U}_{\mathsf{F}}(\pi'') \implies \mathcal{M}_{\pi \wedge \pi''}(\hat{h}) \cap \mathcal{M}_{*}(\mathsf{C} * \mathsf{EF} * \pi') = \emptyset$$

Proof. There are three possible causes of failure: a cell unification failure, a pure entailment failure, and an extra resource failure. We proceed by case analysis on the cause of failure.

- [Cell unification failure] We conclude that there is a cell assertion c and a symbolic heap $\hat{h}_f$ such that: $\pi \vdash \langle \hat{h}, \mathsf{C}\rangle \rightarrow^{*}_{CU} \langle \hat{h}_f, \mathsf{c} :: \mathsf{C}''\rangle$ (**I1**) and $\pi \vdash \langle \hat{h}_f, \mathsf{c}\rangle \rightarrow_{CU} \mathsf{F}(\pi'')$ (**I2**), and $\mathsf{C} = \mathsf{C}' \cdot (\mathsf{c} :: \mathsf{C}'')$ (**I3**). Applying Lemma D.4 to **I1** and **I3**, we conclude that there is a symbolic heap $\hat{h}'$ such that $\hat{h} = \hat{h}' \uplus \hat{h}_f$ (**I4**) and $\mathcal{M}_{\pi}(\hat{h}') \subseteq \mathcal{M}_{*}(\mathsf{C}')$ (**I5**). Applying Lemma D.3 to **I2**, we conclude that:

  $$\forall \hat{h}'', \hat{h}'_f.\, \hat{h}_f = \hat{h}'' \uplus \hat{h}'_f \implies \mathcal{M}_{\pi \wedge \pi'}(\hat{h}'') \cap \mathcal{M}_{*}(\mathsf{c}) = \emptyset \; (\mathbf{I6})$$

  Equation **I6** tells us that there is no way to unify $\hat{h}_f$ against c. However, to establish the result, one needs to prove that there is no way to unify $\hat{h}$ against c. Put formally, we need to prove that:

  $$\forall \hat{h}'', \hat{h}'_f.\, \hat{h} = \hat{h}'' \uplus \hat{h}'_f \implies \big(\mathcal{M}_{\pi \wedge \pi'}(\hat{h}'', \hat{\rho}) \cap \mathcal{M}_{*}(\mathsf{c}) = \emptyset \vee \mathcal{M}_{*}(\mathsf{C} * \pi) = \emptyset\big)$$

  Observing that $\mathcal{M}_{\pi \wedge \pi'}(\hat{h}'', \hat{\rho}) \cap \mathcal{M}_{*}(\mathsf{C}) = \mathcal{M}_{\pi \wedge \pi'}(\hat{h}'', \hat{\rho}) \cap \mathcal{M}_{*}(\mathsf{C} * (\pi \wedge \pi'))$, we rewrite the implication above as follows:

  $$\forall \hat{h}'', \hat{h}'_f.\, \hat{h} = \hat{h}'' \uplus \hat{h}'_f \implies \mathcal{M}_{\pi \wedge \pi'}(\hat{h}'', \hat{\rho}) \cap \mathcal{M}_{*}(\mathsf{C} * (\pi \wedge \pi')) = \emptyset$$

We prove the above by contradiction. Let us assume that there are two symbolic heaps $\hat{h}''$ and $\hat{h}'_f$ such that: $\hat{h} = \hat{h}'' \uplus \hat{h}'_f$ (**I7**) and $\big(\mathcal{M}_{\pi \wedge \pi'}(\hat{h}'', \hat{\rho}) \cap \mathcal{M}_*(\mathsf{c} * (\pi \wedge \pi')) \neq \emptyset) \vee \mathcal{M}_*(\mathsf{C} * \pi) = \emptyset\big)$ (**I8**). We have two cases to consider: either $\hat{h}''$ is part of $\hat{h}'$ or it is part of $\hat{h}_f$. We prove the two cases separately.

- If $\hat{h}''$ is part of $\hat{h}'$, it means that $\hat{h}'', \pi \models \mathsf{c}$ and there is a cell c', such that $\hat{h}'', \pi \models \mathsf{c}'$. Using Lemma D.12, we conclude that $\mathcal{M}_*(\mathsf{c} * \mathsf{c}' * \pi) = \emptyset$, from which it follows that $\mathcal{M}_*(\mathsf{C} * (\pi \wedge \pi')) = \emptyset$, obtaining a contradiction with **I8**.
- If $\hat{h}''$ is part of $\hat{h}_f$, we immediately get a contradiction with **I6**.

Having established that:

$$\forall \hat{h}'', \hat{h}'_f .\, \hat{h} = \hat{h}'' \uplus \hat{h}'_f \implies \Big(\mathcal{M}_{\pi \wedge \pi'}(\hat{h}'', \hat{\rho}) \cap \mathcal{M}_*(\mathsf{C} * (\pi \wedge \pi')) = \emptyset\Big) \vee \mathcal{M}_*(\mathsf{C}) = \emptyset \ \textbf{(I9)}$$

the result follows immediately.

- [Extra Resource Failure] We conclude that $\pi \vdash \langle \hat{h}, \mathsf{C} \rangle \to^*_{CU} \langle \hat{h}_f, [] \rangle$ and $\hat{h}_f \neq \emptyset$. Observing that the size of the symbolic heap is smaller than the number of cells against which it needs to be unified, we conclude the result.
- [Pure Entailment Failure] We conclude that:

$$\pi \vdash \langle \hat{h}, \mathsf{C} \rangle \to^*_{CU} \langle \emptyset, [] \rangle \ \textbf{(I10)} \qquad \pi'' = \neg(\pi' \wedge \mathcal{U}_{NR}(\hat{h}, \mathsf{EF}) \wedge \mathcal{S}(\mathsf{EF}))\ \textbf{(I11)}$$

□