Gillian, Part I: A Multi-language Platform for Symbolic Execution

José Fragoso Santos
INESC-ID/Instituto Superior Técnico
Universidade de Lisboa, Portugal
Imperial College London, UK
jose.fragoso@tecnico.ulisboa.pt

Petar Maksimović
Imperial College London, UK
p.maksimovic@imperial.ac.uk

Sacha-Élie Ayoun
Imperial College London, UK
s.ayoun@imperial.ac.uk

Philippa Gardner
Imperial College London, UK
p.gardner@imperial.ac.uk

Abstract

We introduce Gillian, a platform for developing symbolic analysis tools for programming languages. Here, we focus on the symbolic execution engine at the heart of Gillian. We give a formal description of the symbolic analysis, parametric on the memory model of the target language, and a modular implementation that closely following the formal description. We prove a parametric soundness result, introducing *restriction* on abstract states which generalises path conditions used in classical symbolic execution. We instantiate Gillian to obtain trusted symbolic testing tools for JavaScript and C, and use these tools to find bugs in real-world code, thus demonstrating the viability of our Gillian approach.

CCS Concepts • **Theory of computation → Program analysis**; *Program semantics*; • **Software and its engineering → Formal language definitions**;

Keywords Symbolic execution, parametric semantics, bug-finding, bounded verification, JavaScript, C

ACM Reference Format:
José Fragoso Santos, Petar Maksimović, Sacha-Élie Ayoun, and Philippa Gardner. 2020. Gillian, Part I: A Multi-language Platform for Symbolic Execution. In *Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI '20), June 15–20, 2020, London, UK*. ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3385412.3386014

1 Introduction

Symbolic execution is a well-established analysis technique for reasoning about programs [11, 12]. The development of symbolic analyses for modern programming languages, however, constitutes a substantial effort, and transferring such analyses from one language to another often comes at a prohibitive cost. For this reason, frameworks and tools that can be re-used for multiple languages have been established, using essentially three approaches: *symbolic-lifting frameworks*, such as Rosette [62, 63] and Chef [8], which automatically lift a concrete interpreter for a target language (TL) into a symbolic interpreter; *semantic frameworks*, such as $\mathbb{K}$ [21], which provide a specification language for writing the TL semantics and automatically generate various analysis tools from this semantics; and *multi-language IR-based tools*, such as Viper [37, 38], SAW [17] and Infer [13], which compile high-level TLs into a relatively simple intermediate representation (IR) where the symbolic analysis is performed.

All three approaches have had substantial successes in academia and industry: for example, Rosette is regularly applied to analysis of domain-specific languages [6, §5] and has also been used to find bugs in parts of the Linux kernel [40]; $\mathbb{K}$ has been instantiated to various languages, such as Java, JavaScript, and C [5, 25, 43, 59], and is being used in industry for symbolic analysis of Ethereum bytecode [26, 44]; and the industrial tools SAW and Infer are developed and used in, respectively, Galois and Facebook.

In the first two approaches, a tool developer implements the TL memory model using the data structures made available by the framework, such as lists and maps. The framework provides general symbolic reasoning over these data structures, and the challenge is for the developer to find a way to use the data structures so that this general reasoning is optimised for the specific TL. In the third approach, the tools come with a fixed menu of memory models, for which the symbolic reasoning *is* optimised, but offer no mechanism for adding new memory models. With respect to correctness, the general symbolic reasoning of the first two approaches is correct-by-construction, whereas the correctness of the specific symbolic reasoning in the third approach needs to be argued on a case-by-case basis and is usually not discussed.

PLDI '20, June 15–20, 2020, London, UK

ACM ISBN 978-1-4503-7613-6/20/06.
https://doi.org/10.1145/3385412.3386014

We introduce Gillian, a novel IR-based multi-language platform for the development of symbolic-execution tools: like the first two approaches, Gillian gives a tool developer the freedom to create a new memory model for a given TL; like the third approach, it allows the tool developer to implement language-specific symbolic reasoning for this TL. The correctness of the resulting symbolic reasoning does not come for free, but requires only several lemmas associated with the specific memory models.

At the core of Gillian is GIL, an intermediate language *parametric* on the memory model of the TL: that is, on a set of actions capturing the fundamental ways in which TL programs interact with their memories. To instantiate Gillian to a given TL, the tool developer needs to: **(1)** implement the concrete and symbolic memory models of the TL in terms of its actions[1]; and **(2)** provide a trusted compiler from the TL to GIL instantiated with these actions, which preserves the TL memory models and semantics. The resulting tool allows the general developer to write standard symbolic unit tests [12, 63], with symbolic inputs and code annotations comprising first-order assumptions and assertions. Gillian symbolically executes that code, exploring all paths and unrolling loops up to a bound, providing either a true counter-model or a bounded verification guarantee for the given assertions.

We believe that our use of a parametric IR for multi-language tools is novel. It was partly inspired by tools developed for abstraction interpretation [36, 56]. The nearest approach for symbolic execution is likely that of the SAW tool [17], developed by Galois, which has several built-in memory models of low-level languages, such as LLVM and JVM, but no parameterisation mechanism. Unlike most IR-based tools [13, 17, 37], we emphasise the use of a trustworthy compiler preserving the TL memory models and semantics: the TL and IR memory models being the same simplifies the analysis and error reporting; and the control flow of the TL should simply be replaced by the control flow of the GIL instantiation. This emphasis was inspired in part by an analogous emphasis in JaVerT [52, 53], a symbolic analysis toolchain for JavaScript.

We present the core parametric symbolic execution engine of Gillian formally in §2, illustrating the definitions by instantiating GIL with the actions and the memory model of a simple While language with static objects.

Gillian comes with parametric soundness results and a modular implementation that closely follows the meta-theory; this strong connection is an important aim of our work. In §3, we give our parametric soundness results, proving that the symbolic analysis of Gillian has no false positives. These results are made possible by our novel concept of *restriction*, which generalises the well-known notion of path conditions [1], allowing us to abstractly characterise what it means for an individual symbolic execution trace to be sound in symbolic testing. Moreover, restriction unifies various techniques for directing concrete executions in soundness results, such as strengthening the initial symbolic state with the path condition of the final symbolic state [11, 12, 51] or with information about non-determinism arising from, for example, randomness or allocation. The parametric soundness results greatly simplify soundness proofs for new Gillian instantiations, in that the tool developer only needs to formalise the concrete/symbolic memory models of the TL and prove two lemmas on the behaviour of the actions; we illustrate this process for the simple While language.

Gillian can be used by tool developers to create symbolic execution tools for a broad range of TLs, whether they be toy (such as While), domain-specific, or real-world. In §4, we instantiate Gillian to obtain two trusted symbolic execution engines, Gillian-JS and Gillian-C, for JavaScript and C, respectively. These two languages are good examples for Gillian, as their memory models differ substantially and there already exist two established trustworthy compilers: the JaVerT compiler [53], which follows the JavaScript standard line-by-line and is thoroughly tested; and the CompCert C compiler [33], verified in Coq. Starting from these compilers, we obtain trustworthy compilers from JavaScript and C to GIL, supporting a significant subset of their respective languages. We evaluate Gillian-JS and Gillian-C on the real-world data-structure libraries, Buckets.js [54] and Collections-C [50], respectively, by creating comprehensive symbolic test suites. Our testing has discovered bugs in both libraries, with testing times that indicate that the analysis can scale to larger codebases, demonstrating that our parametric approach to symbolic execution, firmly grounded in theory, is viable.

A Broader Overview of the Gillian Project. In addition to symbolic testing, Gillian also supports full verification based on separation logic and automatic compositional testing based on bi-abduction [14]. These analyses come with parametric soundness results and are implemented modularly: verification is added on top of the symbolic testing presented here, and automatic compositional testing on top of verification. Importantly, the specifications resulting from bi-abduction are *precise* and can be re-used in both correctness analyses, such as verification, or incorrectness analyses, such as the incorrectness logic of O'Hearn [42]. We aim to publish the details of this work in future. Gillian is open-sourced and is integrally available online at [61].

2 Parametric Symbolic Execution

At the core of Gillian is a symbolic execution engine for GIL, the intermediate goto language of Gillian. We present the formal GIL semantics, which the OCaml implementation closely follows, in §2.1. The GIL semantics uses a general notion of a *state model*, which can be seen as a formal interface through which a programming language interacts with its state. In §2.3, we demonstrate how specific concrete and symbolic memory models can be automatically lifted

[1] Strictly speaking, the symbolic memory model is sufficient. We found it useful, however, to test the correctness of instantiations concretely.

to specific concrete and symbolic state models. As the running example, we use a simple While language with static objects, giving its actions and a compiler to GIL in §2.2, and its concrete and symbolic memory models in §2.4.

2.1 GIL Syntax and Semantics

GIL is a simple goto language with top-level procedures. It is parametric on a set of actions, $A \ni \alpha$. Actions provide a general mechanism for interacting with GIL states, allowing us to formulate parametric soundness results and keep GIL states opaque throughout the meta-theoretical development.

Syntax. The syntax of GIL is given below. GIL *values*, $v \in \mathcal{V}$, include numbers, strings, booleans, uninterpreted symbols, types, procedure identifiers, and lists of values. Types are standard: they include, e.g., the types of numbers, strings, booleans, and lists. GIL *expressions*, $e \in \mathcal{E}$, include values, program variables x, and various unary and binary operators.

The Syntax of GIL

$$
\begin{array}{l}
v \in \mathcal{V} \triangleq n \in \mathcal{N} \mid s \in \mathcal{S} \mid b \in \mathcal{B} \mid l, \varsigma \in \mathcal{U} \mid \tau \in \mathcal{T} \mid f \in \mathcal{F} \mid \overline{v} \\
e \in \mathcal{E} \triangleq v \mid x \in \mathcal{X} \mid \ominus e \mid e_1 \oplus e_2 \\
c \in C_A \triangleq x := e \mid \mathsf{ifgoto}\ e\ i \mid x := e(e') \mid \mathsf{return}\ e \mid \mathsf{fail}\ e \mid \\
\qquad \mathsf{vanish} \mid x := \alpha(e) \mid x := \mathsf{uSym}_j \mid x := \mathsf{iSym}_j \\
f(x)\{\overline{c}\} \in \mathcal{P}roc_A \qquad \mathsf{p} \in \mathcal{P}rog_A : \mathcal{F} \rightharpoonup \mathcal{P}roc_A
\end{array}
$$

GIL *commands*, $c \in C_A$, include the standard variable assignment, conditional goto, and dynamic procedure call. The procedure call is dynamic in that the identifier is a GIL expression; this allows us to model function calls of both static and dynamic languages. Next, the return command terminates the execution of the current procedure; fail terminates the execution of the entire program with an error; and vanish silently terminates program execution without generating a result. Finally, we have three GIL-specific commands: action execution, $x := \alpha(e)$, which executes the action $\alpha \in A$ with the argument obtained by evaluating e; and two analysis-related commands, $x := \mathsf{uSym}_j$ and $x := \mathsf{iSym}_j$, which use Gillian's built-in symbol generator to generate fresh symbols, similarly to the `gensym` command of Lisp and Racket. We call the symbols created using uSym_j *uninterpreted*, and the symbols created using iSym_j *interpreted*. In the symbolic analysis, uninterpreted symbols are used to represent instantiation-specific constants (e.g., the JavaScript `undefined` and `null`) or unique memory constituents (e.g., heap locations and objects); and interpreted symbols are used to represent logical variables, as in the standard symbolic execution literature. We explain this, together with the identifier j with which the two commands are annotated, in detail in §2.1, §2.3, and §3.2.

A GIL *procedure* is of the form $f(x)\{\overline{c}\}$, where f is its identifier, x is its formal parameter, and its body $\overline{c}$ is a sequence of GIL commands. A GIL *program*, $\mathsf{p} \in \mathcal{P}rog_A$, maps procedure identifiers to their corresponding procedures.

Semantics. The semantics of GIL is parameterised by a state model, $S \in \mathbb{S}$, defined as follows.

Definition 2.1 (State Model)**.** A *state model*, $S \in \mathbb{S}$, is a quadruple $\langle |S|, \mathsf{V}, A, \mathsf{ea} \rangle$, consisting of: **(1)** a set of states on which GIL programs operate, $|S| \ni \sigma$; **(2)** a set of values stored in those states, $\mathsf{V} \ni \mathsf{v}$; **(3)** a set of actions that can be performed on those states, $A \ni \alpha$; and **(4)** a function for executing actions on states, $\mathsf{ea} : A \to |S| \to \mathsf{V} \rightharpoonup \wp(|S| \times \mathsf{V})$. GIL states must contain an internal representation of a *variable store*, denoted by ρ, assigning values to program variables.

We write $\sigma.\alpha(\mathsf{v}) \rightsquigarrow (\sigma', \mathsf{v}')$ to mean $(\sigma', \mathsf{v}') \in \mathsf{ea}(\alpha, \sigma, \mathsf{v})$, and refer to σ' as the *state output* and to v' as the *value output* of action α on state σ with value input v. We write $-$ instead of v when the action argument does not impact the action behaviour, and omit the value output v' if not used afterwards. To streamline the semantics, we define the following two types of action composition:

$$
\begin{array}{l}
\sigma.(\alpha_2 \circ \alpha_1)(\mathsf{v}) \rightsquigarrow (\sigma', \mathsf{v}') \iff \\
\qquad \exists \sigma'', \mathsf{v}''.\ \sigma.\alpha_1(\mathsf{v}) \rightsquigarrow (\sigma'', \mathsf{v}'') \ \wedge\ \sigma''.\alpha_2(\mathsf{v}'') \rightsquigarrow (\sigma', \mathsf{v}') \\
\sigma.(\alpha_2 \bar{\circ} \alpha_1)([\mathsf{v}_1, \mathsf{v}_2]) \rightsquigarrow (\sigma', [\mathsf{v}_1', \mathsf{v}_2']) \iff \\
\qquad \exists \sigma''.\ \sigma.\alpha_1(\mathsf{v}_1) \rightsquigarrow (\sigma'', \mathsf{v}_1') \ \wedge\ \sigma''.\alpha_2(\mathsf{v}_2) \rightsquigarrow (\sigma', \mathsf{v}_2')
\end{array}
$$

which allow us to chain actions in two different ways in the GIL semantics. In particular, both compositions use the state output of α_1 as the state input of α_2, but the first composition uses the value output of α_1 as the value input for α_2, whereas the second takes a pair of value inputs, using the first as the value input of α_1 and the second as value input of α_2.

A state model $S = \langle |S|, \mathsf{V}, A, \mathsf{ea} \rangle$ is *proper* if and only if its set of actions, A, includes the following distinguished actions and families of actions: **(1)** $\{\mathsf{setVar}_x\}_{x \in \mathcal{X}}$, for assigning a given value v to a given variable x in the store of a given state σ, pretty-printed $\sigma.\mathsf{setVar}_x(\mathsf{v})$; **(2)** setStore, for replacing the entire store of a given state σ with a new store ρ, pretty-printed $\sigma.\mathsf{setStore}(\rho)$[2]; **(3)** getStore, for obtaining the store of the given state σ, pretty-printed $\sigma.\mathsf{getStore}(-)$; **(4)** $\{\mathsf{eval}_e\}_{e \in \mathcal{E}}$, for evaluating the expression e in a given state σ, pretty-printed $\sigma.\mathsf{eval}_e(-)$, **(5)** assume, for extending the given state with the information denoted by a given value v, pretty-printed $\sigma.\mathsf{assume}(\mathsf{v})$; and **(6)** uSym and iSym, for generating new uninterpreted and interpreted symbols, and pretty-printed $\sigma.\mathsf{uSym}(\mathsf{v})$ and $\sigma.\mathsf{iSym}(\mathsf{v})$, respectively. From now on, we work with proper state models.

The semantics of GIL is defined in Figure 1; it uses the notions of *call stacks*, which keep track of the execution context; *outcomes*, which capture the flow of execution; and *configurations*, which keep track of the overall program state.

GIL Semantic Domains for $S = \langle |S|, \mathsf{V}, A, \mathbf{ea} \rangle$

$$
\begin{array}{ll}
cs \in Call_S \triangleq \langle f \rangle \mid \langle f, x, \rho, i \rangle : cs & cf \in Conf_S \triangleq \langle \sigma, cs, i \rangle \\
o \in O \triangleq \cdot \mid \mathsf{N}(\mathsf{v}) \mid \mathsf{E}(\mathsf{v}) &
\end{array}
$$

A call stack, $cs \in Call_S$, is a non-empty list of stack frames. A top-level stack frame, $\langle f \rangle$, contains the identifier of the

[2]Precisely, setStore and getstore use lists of pairs, with each pair containing a serialised program variable (e.g. as a string) and its corresponding value.

$$\textsc{Assignment}\ \frac{\mathsf{cmd}(\mathsf{p}, cs, i) = x := e \qquad \sigma.(\mathsf{setVar}_x \circ \mathsf{eval}_e)(-) \rightsquigarrow \sigma'}{\mathsf{p} \vdash \langle \sigma, cs, i\rangle \rightsquigarrow \langle \sigma', cs, i{+}1\rangle}$$

$$\textsc{IfGoto - True}\ \frac{\mathsf{cmd}(\mathsf{p}, cs, i) = \mathsf{ifgoto}\ e\ j \qquad \sigma.(\mathsf{assume} \circ \mathsf{eval}_e)(-) \rightsquigarrow \sigma'}{\mathsf{p} \vdash \langle \sigma, cs, i\rangle \rightsquigarrow \langle \sigma', cs, j\rangle}$$

$$\textsc{IfGoto - False}\ \frac{\mathsf{cmd}(\mathsf{p}, cs, i) = \mathsf{ifgoto}\ e\ j \qquad \sigma.(\mathsf{assume} \circ \mathsf{eval}_{\neg e})(-) \rightsquigarrow \sigma'}{\mathsf{p} \vdash \langle \sigma, cs, i\rangle \rightsquigarrow \langle \sigma', cs, i+1\rangle}$$

$$\textsc{Action}\ \frac{\mathsf{cmd}(\mathsf{p}, cs, i) = x := \alpha(e) \qquad \sigma.(\mathsf{setVar}_x \circ \alpha \circ \mathsf{eval}_e)(-) \rightsquigarrow \sigma'}{\mathsf{p} \vdash \langle \sigma, cs, i\rangle \rightsquigarrow \langle \sigma', cs, i{+}1\rangle}$$

$$\textsc{Call}\ \frac{\begin{array}{c}\mathsf{cmd}(\mathsf{p}, cs, i) = x := e(e') \quad \sigma.((\mathsf{getStore} \bar{\circ} \mathsf{eval}_{e'}) \bar{\circ} \mathsf{eval}_e)([-,[-,-]]) \rightsquigarrow (\sigma', [f,[\mathsf{v},\rho']]) \\ cs' = \langle f, x, \rho', i{+}1\rangle : cs \qquad \mathsf{p}(f) = f(y)\{\overline{c}\} \qquad \sigma'.\mathsf{setStore}([[y,\mathsf{v}]]) \rightsquigarrow \sigma''\end{array}}{\mathsf{p} \vdash \langle \sigma, cs, i\rangle \rightsquigarrow \langle \sigma'', cs', 0\rangle}$$

$$\textsc{Return}\ \frac{\begin{array}{c}\mathsf{cmd}(\mathsf{p}, cs, i) = \mathsf{return}\ e \qquad cs = \langle -, x, \rho, j\rangle : cs' \\ \sigma.\mathsf{eval}_e(-) \rightsquigarrow (\sigma', \mathsf{v}) \qquad \sigma'.(\mathsf{setVar}_x \bar{\circ} \mathsf{setStore})([\rho, \mathsf{v}]) \rightsquigarrow \sigma''\end{array}}{\mathsf{p} \vdash \langle \sigma, cs, i\rangle \rightsquigarrow \langle \sigma'', cs', j\rangle}$$

$$\textsc{Top Return}\ \frac{\mathsf{cmd}(\mathsf{p}, cs, i) = \mathsf{return}\ e \qquad cs = \langle f\rangle \quad \sigma.\mathsf{eval}_e(-) \rightsquigarrow (\sigma', \mathsf{v})}{\mathsf{p} \vdash \langle \sigma, cs, i\rangle \rightsquigarrow \langle \sigma', cs, i\rangle^{\mathsf{N}(\mathsf{v})}}$$

$$\textsc{Fail}\ \frac{\mathsf{cmd}(\mathsf{p}, cs, i) = \mathsf{fail}\ e \qquad \sigma.\mathsf{eval}_e(-) \rightsquigarrow (\sigma', \mathsf{v})}{\mathsf{p} \vdash \langle \sigma, cs, i\rangle \rightsquigarrow \langle \sigma', cs, i\rangle^{\mathsf{E}(\mathsf{v})}}$$

$$\textsc{uSym}\ \frac{\mathsf{cmd}(\mathsf{p}, cs, i) = x := \mathsf{uSym}_j \qquad \sigma.(\mathsf{setVar}_x \circ \mathsf{uSym})(j) \rightsquigarrow \sigma'}{\mathsf{p} \vdash \langle \sigma, cs, i\rangle \rightsquigarrow \langle \sigma', cs, i{+}1\rangle}$$

$$\textsc{iSym}\ \frac{\mathsf{cmd}(\mathsf{p}, cs, i) = x := \mathsf{iSym}_j \qquad \sigma.(\mathsf{setVar}_x \circ \mathsf{iSym})(j) \rightsquigarrow \sigma'}{\mathsf{p} \vdash \langle \sigma, cs, i\rangle \rightsquigarrow \langle \sigma', cs, i{+}1\rangle}$$

Figure 1. GIL Semantics: $\mathsf{p} \vdash \langle \sigma, cs, i\rangle^{\cdot} \rightsquigarrow \langle \sigma', cs', j\rangle^{o}$

procedure that started the execution. An inner stack frame, $\langle f, x, \rho, i\rangle$, contains: **(1)** the identifier f of the executing procedure; **(2)** the variable x to which f's return value will be assigned; **(3)** the store ρ of the caller of f; and **(4)** the index i to which control is transferred on termination of f.

GIL has three possible outcomes, $o \in O$: **(1)** continuation, $\cdot$, signifying that the execution may proceed with the next command; **(2)** return, $\mathsf{N}(\mathsf{v})$, signifying that there was a top-level return with value v; and **(3)** error, $\mathsf{E}(\mathsf{v})$, signifying that the execution *failed* with value v. In the rules, we elide the continuation outcome whenever it is clear from the context.

The GIL semantics relates configurations and outcomes under a given program. The transitions are of the form $\mathsf{p} \vdash \langle \sigma, cs, i\rangle^{\cdot} \rightsquigarrow \langle \sigma', cs', j\rangle^{o}$, read: given a program p, the evaluation of the i-th command of the top procedure of the call stack cs in the state σ generates the state σ', the call stack cs' and the outcome o, and the next command to be executed is the j-th command of the top procedure of cs'.

The rules of Figure 1 use the function $\mathsf{cmd}(\mathsf{p}, cs, i)$, which returns the i-th command of the top procedure of cs, given the program p. Most are straightforward, given the description of actions of proper state models. The two conditional goto rules correspond to the two goto branches, with each rule assuming the condition under which its branch is taken. Finally, fresh symbol generation remains opaque at this level—one implementation will be shown in §2.3, in the context of concrete and symbolic states.

GIL Allocation: Parametric Construction. The generation of fresh values is a common source of technical clutter often omitted or hand-waved in the formal presentation of program analyses. Gillian takes care of this issue for the tool developer by having built-in fresh-value *allocators* [2].

Definition 2.2 (Allocator)**.** An allocator, $AL \in \mathbb{AL}$, is a triple, $\langle |AL|, \mathsf{V}, \mathsf{alloc}\rangle$, consisting of: **(1)** a set $|AL| \ni \xi$ of allocation records; **(2)** a set V of all values that are allowed to be allocated; and **(3)** an allocation function:

$$\mathsf{alloc} : |AL| \to \mathbb{N} \to \wp(\mathsf{V}) \rightharpoonup |AL| \times \mathsf{V},$$

pretty-printed as $\xi.\mathsf{alloc}(j) \to_Y (\xi', y)$, which takes an allocation record ξ, an allocation site j, and an allocation range $Y \subseteq \mathsf{V}$, and returns a fresh value, $y \in Y$, together with the appropriately updated allocation record, ξ'.

Intuitively, an allocation record keeps track of already allocated values, but also values that should be allocated in the future. This approach is complementary to the free set approach (e.g., [48]), which keeps track of values that can still be allocated. An allocation site j is the program point associated with either the uSym_j or the iSym_j command. We show how allocators are used in §2.3 and discuss them in detail in §3.2, in the context of our soundness results.

2.2 While: Actions and Compilation to GIL

We show how to instantiate Gillian to a simple While language with static objects. The first step in this process is to identify the While actions and provide a trustworthy compiler from While to GIL. The While syntax is given below.

The Syntax of While

$$\begin{array}{rcl} s \in C_{\mathbb{W}} & \triangleq & x := e \mid \mathsf{if}(e)\{s_1\}\mathsf{else}\{s_2\} \mid \mathsf{while}(e)\{s\} \mid s_1; s_2 \mid \\ & & x := f(e) \mid \mathsf{return}\ e \mid \mathsf{assume}\ e \mid \mathsf{assert}\ e \mid \\ & & x := \{p_i : e_i \mid_{i=1}^{n}\} \mid \mathsf{dispose}\ e \mid x := e.p \mid e.p := e' \end{array}$$

While commands, $s \in C_{\mathbb{W}}$, include the variable assignment, the if-then-else conditional, the while loop, sequence, the static function call, the return command, assume and assert commands for symbolic analysis, object creation and disposal, and property lookup and mutation. For simplicity, we assume that the semantics of expressions and the variable store coincide for While and GIL.

To compile While to GIL, we first have to pick an appropriate set of actions, $A_{\mathbb{W}}$. Since we have four operations on objects (allocation, disposal, lookup, and mutation) assigning an action to each operations would be a reasonable first attempt. However, as Gillian has a built-in allocator, we do not need an action for object allocation. The set of actions for While is, therefore, $A_{\mathbb{W}} = \{\mathsf{lookup}, \mathsf{mutate}, \mathsf{dispose}\}$.

Part of our While to GIL compiler is given in Figure 2. It is modelled as a function $\mathcal{T} : C_{\mathbb{W}} \to \mathbb{N} \to C_{A_{\mathbb{W}}}\ \mathsf{list} \times \mathbb{N}$, mapping a While command $s \in C_{\mathbb{W}}$ and a natural number pc (read: *program counter*) to a sequence of GIL commands

Assignment
$\mathcal{T}(x := e, \mathsf{pc}) \triangleq$
$\mathsf{pc} : x := e$
$\cdot \leftarrow \mathsf{pc} + 1$

Assume
$\mathcal{T}(\mathsf{assume}\ e, \mathsf{pc}) \triangleq$
$\mathsf{pc} : \mathsf{ifgoto}\ e\ \ (\mathsf{pc} + 2)$
$\mathsf{pc} + 1 : \mathsf{vanish}$
$\cdot \leftarrow \mathsf{pc} + 2$

Assert
$\mathcal{T}(\mathsf{assert}\ e, \mathsf{pc}) \triangleq$
$\mathsf{pc} : \mathsf{ifgoto}\ e\ \ (\mathsf{pc} + 2)$
$\mathsf{pc} + 1 : \mathsf{fail}\ e$
$\cdot \leftarrow \mathsf{pc} + 2$

New
$\mathcal{T}(x := \{p_i : e_i \mid_{i=1}^{n}\}, \mathsf{pc}) \triangleq$
$\mathsf{pc} : x := \mathsf{uSym}$
$\mathsf{pc} + i : - := \mathsf{mutate}([x, p_i, e_i]) \mid_{i=1}^{n}$
$\cdot \leftarrow \mathsf{pc} + n + 1$

Lookup
$\mathcal{T}(x := e.p, \mathsf{pc}) \triangleq$
$\mathsf{pc} : x := \mathsf{lookup}([e, p])$
$\cdot \leftarrow \mathsf{pc} + 1$

Figure 2. The While to GIL Compiler (excerpt)

and the next available program counter, npc (denoted by $\cdot \leftarrow \mathsf{npc}$ in Figure 2). For instance, if $\mathcal{T}(s, \mathsf{pc}) = (\overline{c}, \mathsf{npc})$, then the command s compiles to the sequence of GIL commands given by $\overline{c}$, labelled with indexes pc to npc − 1.

The compilation is straightforward; we explain the rules given in Figure 2. The While assignment is compiled directly to a GIL assignment. The assume command, assume e, compiles to a goto command, ifgoto e (pc + 2), which branches on the value of the expression to be assumed, followed by a silent cutting of the branch in which it does not hold by using the GIL vanish command. The assert command, assert e, is compiled to the same goto command that branches on e, but this time, if the branch in which e does not hold is reached, the execution terminates with an error via the GIL fail command. The compilation of the object creation command is instructive, as it illustrates how uninterpreted symbols can be used to represent unique object locations (which is their primary use case) and also how actions are to the called. In particular, we first create a fresh location for the new object using the GIL uSym command, and then set its properties via the mutate action, passing as its single parameter a GIL list containing the object location, the property to be set, and the expression denoting its value. Finally, the lookup of While is compiled to a call to the corresponding action, lookup, passing as elements in a GIL list the expression denoting the location of the object, e, and the looked-up property, p.

2.3 Concrete and Symbolic States

Reasoning about programs can intuitively be separated into reasoning about the *variable store* and about the *memory model* of the programming language in question. Gillian simplifies this process by providing built-in reasoning about the variable store, leaving to the tool developer only to take care of the concrete and symbolic memory models. In particular, it is possible to lift a given memory to a GIL state by combining it with an appropriate variable store and allocator. In this section, we illustrate this lifting for *concrete* and *symbolic* memories, obtaining concrete and symbolic states.

Concrete memories store GIL values, $v \in \mathcal{V}$, whereas symbolic memories store logical expressions, $\hat{e} \in \hat{\mathcal{E}}$, generated by the grammar $\hat{e} \in \hat{\mathcal{E}} \triangleq v \mid \hat{x} \in \hat{\mathcal{X}} \mid \ominus \hat{e} \mid \hat{e}_1 \oplus \hat{e}_2$, where $\hat{x}$ ranges over a set of *logical variables*, $\hat{\mathcal{X}}$. Symbolic memories and states also depend on *path conditions* (boolean logical expressions), $\pi \in \Pi \subseteq \hat{\mathcal{E}}$, used to bookkeep the constraints on logical variables that led the execution to the current symbolic state [1]. Concrete allocators allocate from the set of GIL values, and symbolic allocators allocate from the set of logical expressions.

Definition 2.3 (Concrete Memory Model)**.** Given the set of GIL values, $\mathcal{V}$, a concrete memory model, $M \in \mathbb{M}$, is a triple $\langle |M|, A, \underline{\mathrm{ea}} \rangle$, consisting of: **(1)** a set of concrete memories, $|M| \ni \mu$; **(2)** a set of actions $A \ni \alpha$; and **(3)** the action execution function: $\underline{\mathrm{ea}} : A \to |M| \to \mathcal{V} \rightharpoonup \wp(|M| \times \mathcal{V})$, pretty-printed $\mu.\alpha(v) \rightsquigarrow (\mu', v')$.

Definition 2.4 (Symbolic Memory Model)**.** Given the set of logical expressions, $\hat{\mathcal{E}}$, and the set of path conditions, Π, a symbolic memory model, $\hat{M} \in \hat{\mathbb{M}}$, is a triple, $\langle |\hat{M}|, A, \underline{\hat{\mathrm{ea}}} \rangle$, consisting of: **(1)** a set of symbolic memories, $|\hat{M}| \ni \hat{\mu}$; **(2)** a set of actions, $A \ni \alpha$; and **(3)** an action execution function, $\underline{\hat{\mathrm{ea}}} : A \to |\hat{M}| \to \hat{\mathcal{E}} \to \Pi \rightharpoonup \wp(|\hat{M}| \times \hat{\mathcal{E}} \times \Pi)$, pretty-printed $\hat{\mu}.\alpha\,(\hat{e}, \pi) \rightsquigarrow (\hat{\mu}', \hat{e}', \pi')$.

We formally describe the liftings from concrete and symbolic memory models to the appropriate state models below, with $[\![e]\!]_\rho$ and $[\![e]\!]_{\hat{\rho}}$ denoting expression evaluation with respect to a given concrete and symbolic variable store, and

$$A_{proper} = \{\mathrm{setVar}_x\}_{x \in \mathcal{X}} \cup \{\mathrm{setStore}, \mathrm{getStore}\} \cup \{\mathrm{eval}_e\}_{e \in \mathcal{E}} \cup \{\mathrm{assume}, \mathrm{uSym}, \mathrm{iSym}\}.$$

Definition 2.5 (Concrete State Constructor)**.** Given a concrete allocator, $AL = \langle |AL|, \mathcal{V}, \mathrm{alloc} \rangle$, the concrete state constructor, $CSC_{AL} : \mathbb{M} \to \mathbb{S}$, is defined as $CSC_{AL}(\langle |M|, A, \underline{\mathrm{ea}} \rangle) \triangleq \langle |S|, \mathcal{V}, A_{proper} \uplus A, \mathrm{ea} \rangle$, where $\alpha \in A$ and:

$$
\begin{array}{ll}
|S| = |M| \times (\mathcal{X} \rightharpoonup \mathcal{V}) \times |AL| & \\
\mathrm{ea}(\mathrm{setVar}_x, \langle \mu, \rho, \xi \rangle, v) & \triangleq \{(\langle \mu, \rho[x \mapsto v], \xi \rangle, \mathsf{true})\} \\
\mathrm{ea}(\mathrm{setStore}, \langle \mu, -, \xi \rangle, \rho) & \triangleq \{(\langle \mu, \rho, \xi \rangle, \mathsf{true})\} \\
\mathrm{ea}(\mathrm{getStore}, \langle \mu, \rho, \xi \rangle, -) & \triangleq \{(\langle \mu, \rho, \xi \rangle, \rho)\} \\
\mathrm{ea}(\mathrm{eval}_e, \langle \mu, \rho, \xi \rangle, -) & \triangleq \{(\langle \mu, \rho, \xi \rangle, [\![e]\!]_\rho\} \\
\mathrm{ea}(\mathrm{assume}, \sigma, v) & \triangleq \{(\sigma, v) \mid v = \mathsf{true}\} \\
\mathrm{ea}(\mathrm{uSym}, \langle \mu, \rho, \xi \rangle, j) & \triangleq \{(\langle \mu, \rho, \xi' \rangle, \varsigma) \mid \xi.\mathrm{alloc}\,(j) \rightarrow_{\mathcal{U}} (\xi', \varsigma)\} \\
\mathrm{ea}(\mathrm{iSym}, \langle \mu, \rho, \xi \rangle, j) & \triangleq \{(\langle \mu, \sigma, \xi' \rangle, v) \mid \xi.\mathrm{alloc}\,(j) \rightarrow_{\mathcal{V}} (\xi', v)\} \\
\mathrm{ea}(\alpha, \langle \mu, \rho, \xi \rangle, v) & \triangleq \{(\langle \mu', \rho, \xi \rangle, v') \mid (\mu', v') \in \underline{\mathrm{ea}}\,(\alpha, \mu, v)\}
\end{array}
$$

Definition 2.6 (Symbolic State Constructor)**.** Given a symbolic allocator, $\hat{AL} = \langle |\hat{AL}|, \hat{\mathcal{E}}, \mathrm{alloc} \rangle$, the symbolic state constructor $SSC_{\hat{AL}} : \hat{\mathbb{M}} \to \hat{S}$ is defined as $SSC_{\hat{AL}}(\langle |\hat{M}|, A, \underline{\hat{\mathrm{ea}}} \rangle) \triangleq \langle |\hat{S}|, \hat{\mathcal{E}}, A_{proper} \uplus A, \mathrm{ea} \rangle$, where $\alpha \in A$ and:

$$
\begin{array}{ll}
|\hat{S}| = |\hat{M}| \times (\mathcal{X} \rightharpoonup \hat{\mathcal{E}}) \times |\hat{AL}| \times \Pi & \\
\mathrm{ea}(\mathrm{setVar}_x, \langle \hat{\mu}, \hat{\rho}, \hat{\xi}, \pi \rangle, \hat{e}) & \triangleq \{(\langle \hat{\mu}, \hat{\rho}[x \mapsto \hat{e}], \hat{\xi}, \pi \rangle, \mathsf{true})\} \\
\mathrm{ea}(\mathrm{setStore}, \langle \hat{\mu}, -, \hat{\xi}, \pi \rangle, \hat{\rho}) & \triangleq \{(\langle \hat{\mu}, \hat{\rho}, \hat{\xi}, \pi \rangle, \mathsf{true})\} \\
\mathrm{ea}(\mathrm{getStore}, \langle \hat{\mu}, \hat{\rho}, \hat{\xi}, \pi \rangle, -) & \triangleq \{(\langle \hat{\mu}, \hat{\rho}, \hat{\xi}, \pi \rangle, \hat{\rho})\} \\
\mathrm{ea}(\mathrm{eval}_e, \langle \hat{\mu}, \hat{\rho}, \hat{\xi}, \pi \rangle, -) & \triangleq \{(\langle \hat{\mu}, \hat{\rho}, \hat{\xi}, \pi \rangle, [\![e]\!]_{\hat{\rho}})\} \\
\mathrm{ea}(\mathrm{assume}, \langle \hat{\mu}, \hat{\rho}, \hat{\xi}, \pi \rangle, \pi') & \triangleq \{(\langle \hat{\mu}, \hat{\rho}, \hat{\xi}, \pi \wedge \pi' \rangle, \mathsf{true}) \mid \pi \wedge \pi'\ \mathsf{SAT}\} \\
\mathrm{ea}(\mathrm{uSym}, \langle \hat{\mu}, \hat{\rho}, \hat{\xi}, \pi \rangle, j) & \triangleq \{(\langle \mu, \rho, \hat{\xi}', \pi \rangle, \varsigma) \mid \hat{\xi}.\mathrm{alloc}\,(j) \rightarrow_{\mathcal{U}} (\hat{\xi}', \varsigma)\} \\
\mathrm{ea}(\mathrm{iSym}(\langle \hat{\mu}, \hat{\rho}, \hat{\xi}, \pi \rangle, j) & \triangleq \{(\langle \mu, \rho, \hat{\xi}', \pi \rangle, \hat{x}) \mid \hat{\xi}.\mathrm{alloc}\,(j) \rightarrow_{\hat{\mathcal{X}}} (\hat{\xi}', \hat{x})\} \\
\mathrm{ea}(\alpha, \langle \hat{\mu}, \hat{\rho}, \hat{\xi}, \pi \rangle, \hat{e}) & \triangleq \{(\langle \hat{\mu}', \hat{\rho}, \hat{\xi}, \pi \wedge \pi' \rangle, \hat{e}') \\
 & \qquad \mid (\hat{\mu}', \hat{e}', \pi') \in \underline{\hat{\mathrm{ea}}}\,(\alpha, \hat{\mu}, \hat{e}, \pi)\}
\end{array}
$$

$$\frac{\mu = \_ \uplus l.p \mapsto v}{\mu.\text{lookup}([l,p]) \rightsquigarrow (\mu, v)}\ \text{C-Lookup}$$

$$\frac{\hat{\mu} = \_ \uplus \hat{e}'.p \mapsto \hat{e}_v \quad \pi \wedge \hat{e} = \hat{e}' \text{ SAT}}{\hat{\mu}.\text{lookup}([\hat{e},p],\pi) \rightsquigarrow \{(\hat{\mu}, \hat{e}_v, \hat{e} = \hat{e}')\}}\ \text{S-Lookup}$$

$$\frac{\mu = \mu' \uplus l.p \mapsto - \quad \mu'' = \mu' \uplus l.p \mapsto v}{\mu.\text{mutate}([l,p,v]) \rightsquigarrow (\mu'', v)}\ \text{C-Mutate-Present}$$

$$\frac{\hat{\mu} = \hat{\mu}' \uplus \hat{e}''.p \mapsto - \quad \pi \wedge \hat{e} = \hat{e}'' \text{ SAT} \quad \hat{\mu}'' = \hat{\mu}' \uplus \hat{e}''.p \mapsto \hat{e}'}{\hat{\mu}.\text{mutate}([\hat{e},p,\hat{e}'],\pi) \rightsquigarrow \{(\hat{\mu}'', \text{true}, \hat{e} = \hat{e}'')\}}\ \text{S-Mutate-Present}$$

$$\frac{(l,p) \notin \text{dom}(\mu) \quad \mu' = \mu \uplus l.p \mapsto v}{\mu.\text{mutate}([l,p,v]) \rightsquigarrow (\mu', v)}\ \text{C-Mutate-Absent}$$

$$\frac{\pi' = \hat{e} \notin \text{locs}(\hat{\mu}) \vee \hat{e} \in \text{locs}_{\hat{p}}(\hat{\mu}) \quad \pi \wedge \pi' \text{ SAT} \quad \hat{\mu}' = \hat{\mu} \uplus \hat{e}.p \mapsto \hat{e}'}{\hat{\mu}.\text{mutate}([\hat{e},p,\hat{e}'],\pi) \rightsquigarrow \{(\hat{\mu}', \text{true}, \pi')\}}\ \text{S-Mutate-Absent}$$

Figure 3. While: Concrete and Symbolic Actions (excerpt)

Both liftings construct the actions a state model exposes with the help of the action execution function of the parameter memory and the alloc function of the parameter allocator. In both cases, the construction of the store-related functions is straightforward. We describe the remaining cases below.

[EvalExpr]. In the concrete case, expression evaluation is performed in the standard way. In the symbolic case, it amounts to substituting all the program variables in e with their associated logical expressions given by the store. In the implementation, Gillian's first-order solver applies a number of algebraic identities to simplify the resulting expression.

[Assume]. The function σ.assume(v) assumes that the value v *holds* in the state σ. In the concrete case, σ.assume(b) simply returns the singleton set containing the original state when b = true and the empty set otherwise. In the symbolic case, $\hat{\sigma}$.assume(π') returns $\hat{\sigma}$ with its path condition strengthened with π' if this new path condition is satisfiable, and the empty set otherwise. Revisiting the conditional goto rules of Figure 1, we can observe that, in the concrete semantics, the conditional goto deterministically takes the appropriate branch, whereas in the symbolic semantics it may branch if both $\pi \wedge [\![e]\!]_{\hat{\rho}}$ and $\pi \wedge [\![\neg e]\!]_{\hat{\rho}}$ are satisfiable.

[uSym/iSym]. The uSym and iSym actions generate symbols using the parameter allocator. In particular, both concretely and symbolically, uSym picks a fresh uninterpreted symbol, $\varsigma \in \mathcal{U}$. In contrast, iSym picks an arbitrary value concretely and a fresh logical variable symbolically. In §3.2, we will see that this choice corresponds to the standard interpretation of logical variables in symbolic execution.

[Action]. Action execution on states amounts to calling action execution on the parameter memory. As symbolic actions, unlike concrete actions, may branch, they additionally generate a logical expression, π', describing the conditions under which the chosen branch is taken. Hence, the path condition of the obtained state is a conjunction of π' with the path condition π of the original state.

2.4 While: Concrete and Symbolic Memories

The second (and final) step required to instantiate Gillian to the While language is to create the While concrete and symbolic memory models, which are then automatically lifted to the appropriate state models, as per Definitions 2.5 and 2.6.

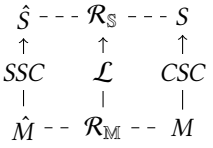
Concrete While memories, $\mu \in |M_{\mathbb{W}}| : \mathcal{U} \times \mathcal{S} \rightharpoonup \mathcal{V}$, partially map uninterpreted symbols (corresponding to object locations) and strings (corresponding to property names) to GIL values. Analogously, symbolic While memories, $\hat{\mu} \in \hat{M}_{\mathbb{W}} : \hat{\mathcal{E}} \times \mathcal{S} \rightharpoonup \hat{\mathcal{E}}$, partially map logical expressions and strings to logical expressions. Property names are not lifted to logical expressions in symbolic memories, as While objects have static properties. The set of While actions, $A_{\mathbb{W}} = \{\text{lookup}, \text{mutate}, \text{dispose}\}$, was given previously in §2.2. We give an excerpt of the concrete and symbolic action execution functions for While in Figure 3, focussing on lookup and mutate; the rules for dispose are analogous. The concrete rules are straightforward; we describe the symbolic rules:

[S-Lookup]. The symbolic lookup of a property p of an object at location denoted by $\hat{e}$ branches on all locations potentially equal to $\hat{e}$ given π and returns the appropriate values from the memory. It also returns the learned constraint, $\hat{e} = \hat{e}'$, which will be added to the path condition of the state.

[S-Mutate]. The symbolic mutation of a property p of an object at location denoted by $\hat{e}$ first branches on whether or not the object defines the property. For the former ([S-Mutate-Present]), the action proceeds analogously to the lookup. For the latter ([S-Mutate-Absent]), the action adds p to the properties of $\hat{e}$, under the condition that $\hat{e}$ is either a new object or it does not define the property p, expressed by π', where $\text{locs}_{\hat{p}}(\hat{\mu})$ denotes the set of object locations in $\hat{\mu}$ that do not define the property p.

3 Parametric Soundness

Proving symbolic analyses sound is time-consuming and often requires a considerable number of auxiliary lemmas and definitions. The complexity of such proofs becomes unwieldy as we move towards real-world programming languages, detracting from mathematical rigour in favour of less demanding, but also less trustworthy, informal arguments. Gillian streamlines the development of soundness proofs for its instantiations, by focussing the proof effort *only* on the memory of the target language and the actions it exposes.



We propose a parametric proof infrastructure, illustrated on the right, which consists of: **(1)** a set of soundness relations between state models, $\mathcal{R}_{\mathbb{S}}$; **(2)** a set of soundness relations between memory models, $\mathcal{R}_{\mathbb{M}}$; and **(3)** a lifting mechanism, $\mathcal{L}$, which, given a relation in $\mathcal{R}_{\mathbb{M}}$, generates a relation in $\mathcal{R}_{\mathbb{S}}$. This infrastructure is underpinned by our novel concept of *restriction*, introduced in §3.1, which generalises the notion of path conditions used in classical symbolic execution. In §3.2, we show

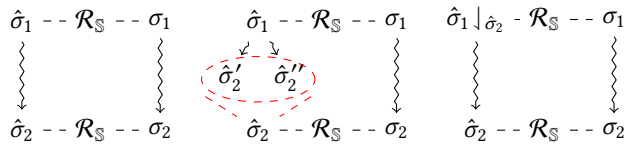


Figure 4. Soundness Properties

that proving the soundness of a given symbolic semantics w.r.t. a given concrete semantics amounts to proving that the soundness relation between the two corresponding memory models is in $\mathcal{R}_{\mathbb{M}}$. Finally, in §3.3, we use our proof infrastructure to show that the While symbolic analysis is sound.

3.1 Parametric Soundness

The standard approach for defining soundness can be coarsely described as in the diagram of Figure 4 (left), where: **(1)** $\hat{\sigma}_1$ is an abstract state over-approximating a concrete state, σ_1; **(2)** $\hat{\sigma}_2$ is the abstract state obtained by abstractly executing a given command on $\hat{\sigma}_1$; and **(3)** σ_2 is the state obtained by concretely executing the same command on σ_1. In this setting, the abstract semantics is *sound* with respect to the concrete one if $\hat{\sigma}_2$ also over-approximates σ_2.

In the context of abstract analyses that may branch, this characterisation of soundness cannot describe what it means for a single abstract trace to be sound. For instance, in Figure 4 (mid), the abstract execution of a given command on $\hat{\sigma}_1$ results in two abstract states, $\hat{\sigma}_2'$ and $\hat{\sigma}_2''$. There, the only way to provide a final abstract state that over-approximates the final concrete state is to *merge* $\hat{\sigma}_2'$ and $\hat{\sigma}_2''$, which requires reasoning about all possible abstract traces at the same time.

In symbolic execution literature [11, 12, 45], however, it is possible to express soundness for a single symbolic trace by strengthening the path condition of the initial symbolic state with the path condition of the final state. This strengthening directs the concrete execution by filtering out all initial concrete states for which the concrete execution diverges from the execution path taken by the symbolic trace.

Inspired by this approach, we give a parametric characterisation of soundness that allows us to describe what it means for a single abstract trace to be sound, but also to recover the standard definition of soundness when given the precise set of all abstract traces or its over-approximation. We achieve this by introducing the notion of *restriction*, which generalises path conditions and provides a unified methodology for directing concrete executions in soundness results. Using this characterisation, we prove that the symbolic analysis provided by Gillian is sound and has no false positives.

Restriction. Restriction is used for pinpointing the set of concrete traces modelled by a given abstract trace; it is applied to states and allocators. Informally, the restriction of an abstract state $\hat{\sigma}_1$ with abstract state $\hat{\sigma}_2$, written $\hat{\sigma}_1 \lfloor_{\hat{\sigma}_2}$, denotes the state $\hat{\sigma}_1$ strengthened with some information from $\hat{\sigma}_2$. In symbolic execution, for example, that additional information could be the path condition of $\hat{\sigma}_2$. In Figure 4 (right), we give a more general intuition: if $\hat{\sigma}_1 \lfloor_{\hat{\sigma}_2}$ over-approximates σ_1, then $\hat{\sigma}_2$ must over-approximate σ_2. This holds because $\hat{\sigma}_1$ has been extended with the exact information needed to direct the execution to $\hat{\sigma}_2$, meaning that the symbolic execution from $\hat{\sigma}_1 \lfloor_{\hat{\sigma}_2}$ to $\hat{\sigma}_2$ will not branch and, consequently, the concrete execution from σ_1 and σ_2 will follow that same path.

Definition 3.1 (Restriction)**.** A *restriction* on a set X is a binary associative function, $\lfloor : X \to X \rightharpoonup X$, written $x_1 \lfloor_{x_2}$ for $\lfloor (x_1, x_2)$, satisfying the following three properties:

$$\textsc{Idempotence}\quad x \lfloor_x = x \qquad \textsc{Right Commutativity}\quad (x_1 \lfloor_{x_2}) \lfloor_{x_3} = (x_1 \lfloor_{x_3}) \lfloor_{x_2} \qquad \textsc{Weakening}\quad \frac{x_1 \lfloor_{x_2} \lfloor_{x_3} = x_1}{x_1 \lfloor_{x_2} = x_1 \lfloor_{x_3} = x_1}$$

Every restriction $\lfloor$ induces a pre-order $(X, \sqsubseteq)$, given by $x_2 \sqsubseteq x_1 \iff x_2 \lfloor_{x_1} = x_2$.

These properties intuitively mean that self-restriction does not gain information, the order of applied restrictions does not influence the accumulated information gain and, if x_1 cannot gain information from the combined knowledge of x_2 and x_3, then it cannot gain information from either x_2 or x_3.

During symbolic execution, path conditions can only get strengthened. Similarly, allocators can only get extended with information about newly allocated values. This monotonicity is captured generally by the following definitions.

Definition 3.2 (State Restriction)**.** Given a state model, $S = \langle |S|, \mathsf{V}, A, \mathsf{ea} \rangle$, a *state restriction* $\lfloor$ on S is a restriction operator on $|S|$ that is monotonic w.r.t. action execution: that is, $\sigma.\alpha(\mathsf{v}) \rightsquigarrow (\sigma', -) \implies \sigma' \sqsubseteq \sigma$, for $\alpha \in A$.

Definition 3.3 (Allocator Restriction)**.** Given an allocator $AL = \langle |AL|, \mathsf{V}, \mathsf{alloc} \rangle$, an *allocator restriction* $\lfloor$ on AL is a restriction on $|AL|$ that is monotonic w.r.t. the allocation function: that is, $\xi.\mathsf{alloc}(j) \rightarrow_Y (\xi', -) \implies \xi' \sqsubseteq \xi$.

Compatibility. To state our soundness results, we require a pre-order relation on abstract states, which we will denote by $\leq$. Intuitively, $\hat{\sigma}_2 \leq \hat{\sigma}_1$ means that the set of concrete states described by $\hat{\sigma}_2$ is contained in those described by $\hat{\sigma}_1$. This pre-order need not coincide with the pre-order $\sqsubseteq$ induced by a given restriction $\lfloor$: for example, in symbolic execution, the fact that the path condition of a state $\hat{\sigma}_2$ implies the path condition of a state $\hat{\sigma}_1$ (i.e., $\hat{\sigma}_2 \sqsubseteq \hat{\sigma}_1$) does not necessarily mean that all the models of $\hat{\sigma}_2$ are contained in the models of $\hat{\sigma}_1$ (i.e., $\hat{\sigma}_2 \leq \hat{\sigma}_1$), as these two states may describe different stores and/or memories. We do, however, require the $\leq$ and $\sqsubseteq$ pre-orders to be compatible, as per the following definition.

Definition 3.4 (Compatibility)**.** A pre-order $(X, \leq)$ is *compatible* with a restriction $\lfloor$ on X *iff* the following hold:

$$\lfloor\text{-}\leq \textsc{ Compat}\quad x_1 \lfloor_{x_2} \leq x_1 \qquad \leq\text{-}\lfloor \textsc{ Compat}\quad \frac{x_2 \leq x_1}{x_2 \sqsubseteq x_1} \qquad \textsc{Strengthening}\quad \frac{x_2 \leq x_1 \quad x_2' \sqsubseteq x_1'}{x_2 \lfloor_{x_2'} \leq x_1 \lfloor_{x_1'}}$$

The first two properties state that restriction increases $\leq$-precision and that $\leq$-precision implies $\sqsubseteq$-precision. The

third, given our intuition, states that if $\hat{\sigma}_2$ describes fewer concrete states than $\hat{\sigma}_1$, then $\hat{\sigma}_2$ restricted with a stronger path condition (that of $\hat{\sigma}_2'$) will describe fewer concrete states than $\hat{\sigma}_1$ restricted with a weaker path condition (that of $\hat{\sigma}_1'$). Expectedly, the pre-order $\sqsubseteq$ induced by $\lfloor$ is compatible with $\lfloor$.

Parametric Soundness. We describe the set of soundness relations that we use to reason parametrically about the soundness of symbolic execution on state models. To this end, we first define a set of candidate soundness relations, and then refine in Definition 3.5 to soundness relations. Later, in Theorem 3.6, we prove that soundness relations are preserved by the GIL semantics.

Given two state models, $\hat{S} = \langle |\hat{S}|, \hat{\mathbb{V}}, A, \widehat{\mathrm{ea}} \rangle$ and $S = \langle |S|, \mathbb{V}, A, \mathrm{ea} \rangle$,[3] a *candidate soundness relation* between $\hat{S}$ and S is a triple $\langle \lfloor, \sim_s, \sim_v \rangle$, consisting of: **(1)** a state restriction $\lfloor$ on $|\hat{S}|$; **(2)** a binary relation $\sim_s \subseteq |\hat{S}| \times |S|$, where $\hat{\sigma} \sim_s \sigma$ means that $\hat{\sigma}$ is an over-approximation of σ; and **(3)** a ternary relation $\sim_v \subseteq |\hat{S}| \times \hat{\mathbb{V}} \times \mathbb{V}$, where $\hat{\sigma} \vdash \hat{v} \sim_v v$ means that, given the information in $\hat{\sigma}$, $\hat{v}$ is an over-approximation of v. In the following, we use the pre-order on $|\hat{S}|$ induced by $\sim_s$; more concretely, we write $\hat{\sigma}_1 \leq_{\sim_s} \hat{\sigma}_2$ to mean that the set of states of $|S|$ represented by $\hat{\sigma}_1$ is contained in the set of states of $|S|$ represented by $\hat{\sigma}_2$: that is, $\{\sigma \mid \hat{\sigma}_1 \sim_s \sigma\} \subseteq \{\sigma \mid \hat{\sigma}_2 \sim_s \sigma\}$. We elide the $\sim_s$ in $\leq_{\sim_s}$ when it is clear from the context.

Definition 3.5 (Soundness Relation)**.** A candidate soundness relation $\langle \lfloor, \sim_s, \sim_v \rangle$ between two state models $\hat{S} = \langle |\hat{S}|, \hat{\mathbb{V}}, A, \widehat{\mathrm{ea}} \rangle$ and $S = \langle |S|, \mathbb{V}, A, \mathrm{ea} \rangle$ is a *soundness relation*, $R_S \in \mathcal{R}_{\mathbb{S}}$, if and only if the following constraints hold:

State Action - Restricted Soundness (SA-RS)
$$\hat{\sigma}.\alpha(\hat{v}) \rightsquigarrow (\hat{\sigma}', \hat{v}') \land \hat{\sigma}'' \leq \hat{\sigma}\lfloor_{\hat{\sigma}'} \land \hat{\sigma}'' \sim_s \sigma \land \hat{\sigma}'' \vdash \hat{v} \sim_v v \land \sigma.\alpha(v) \rightsquigarrow (\sigma', v') \implies \hat{\sigma}'\lfloor_{\hat{\sigma}''} \sim_s \sigma' \land \hat{\sigma}'\lfloor_{\hat{\sigma}''} \vdash \hat{v}' \sim_v v'$$

State Action - Restricted Completeness (SA-RC)
$$\hat{\sigma}.\alpha(\hat{v}) \rightsquigarrow (\hat{\sigma}', \hat{v}') \land \hat{\sigma}'' \leq \hat{\sigma}\lfloor_{\hat{\sigma}'} \land \hat{\sigma}'' \sim_s \sigma \land \hat{\sigma}'' \vdash \hat{v} \sim_v v \implies \exists \sigma', v'. \sigma.\alpha(v) \rightsquigarrow (\sigma', v')$$

Weakening (W)
$$\hat{\sigma} \sqsubseteq \hat{\sigma}' \land \hat{\sigma} \vdash \hat{v} \sim_v v \implies \hat{\sigma}' \vdash \hat{v} \sim_v v$$

We discuss the three constraints in detail below.

[SA - Restricted Soundness]. Given previous soundness results for symbolic execution [11, 12, 51], one might expect the following, simpler constraint:

SA - Restricted Soundness - Simple (SA-RS-S)
$$\hat{\sigma}.\alpha(\hat{v}) \rightsquigarrow (\hat{\sigma}', \hat{v}') \land \hat{\sigma}\lfloor_{\hat{\sigma}'} \sim_s \sigma \land \hat{\sigma}\lfloor_{\hat{\sigma}'} \vdash \hat{v} \sim_v v \land \sigma.\alpha(v) \rightsquigarrow (\sigma', v') \implies \hat{\sigma}' \sim_s \sigma' \land \hat{\sigma}' \vdash \hat{v}' \sim_v v'$$

which requires that if we execute a given action both symbolically and concretely and have that the concrete input state/value is over-approximated by the symbolic input state/value *strengthened with the final path condition*, then the concrete output state/value must also be over-approximated by the symbolic output state/value.

[3] We purposefully overload the notation for symbolic and concrete states ($\hat{S}/S$ and $\hat{\sigma}/\sigma$) to mean that states in $\hat{S}$ are more abstract than those in S.

This constraint is, however, not strong enough to guarantee that the composition of two constraint-abiding actions is still constraint-abiding. For that, we need the more general, strictly stronger constraint given in the definition, from which [SA-RS-S] can be obtained by picking $\hat{\sigma}'' = \hat{\sigma}\lfloor_{\hat{\sigma}'}$.

The more general constraint allows us to strengthen the input symbolic state with *any path condition that implies the final path condition* ($\hat{\sigma}'' \leq \hat{\sigma}\lfloor_{\hat{\sigma}'}$), and not just the final path condition itself. This is essential for proving that, given two actions α_1 and α_2 that satisfy restricted soundness, their composition, $\alpha_2 \circ \alpha_1$, also satisfies restricted soundness. In particular, in the proof, we have to strengthen the symbolic state given to α_1 with the path condition resulting from the execution of α_1 *followed by* α_2, not just the execution of α_1.

[SA - Restricted Completeness]. This constraint states that every abstract action has a concrete counterpart. The details are analogous to the [SA-RS] constraint.

[Weakening]. In the context of symbolic execution, $\hat{\sigma} \vdash \hat{v} \sim_v v$ intuitively means that there exists a logical environment ϵ that satisfies the path condition of $\hat{\sigma}$ and maps $\hat{v}$ to v. Given that $\hat{\sigma} \sqsubseteq \hat{\sigma}'$, meaning that the path condition of $\hat{\sigma}$ is stronger than that of $\hat{\sigma}'$, ϵ must also satisfy the path condition of $\hat{\sigma}'$, from which the conclusion follows.

Finally, we note that both forms of action composition introduced in §2.1 preserve the action constraints, [SA-RS] and [SA-RC]. Put formally, if $\alpha_1, \alpha_2 \in A$ satisfy [SA-RS] and [SA-RC], then so do $\alpha_2 \circ \alpha_1$ and $\alpha_2 \bar{\circ} \alpha_1$. This is essential for proving that the GIL semantics also preserves these constraints, given that, at its core, the semantics essentially associates every command to a composition of specific actions.

GIL Soundness. Theorem 3.6 states that the GIL semantics preserves soundness relations. To state it, we lift restriction to configurations: $\langle \sigma, cs, i \rangle\lfloor_{\langle \sigma', -, - \rangle} \triangleq \langle \sigma\lfloor_{\sigma'}, cs, i \rangle$, Similarly, we lift $\sim_v$, $\sim_s$, and $\leq_{\sim_s}$ to call stacks and configurations. Finally, we use $\rightsquigarrow^n$ to denote an n-step execution and $\rightsquigarrow^*$ to denote the reflexive-transitive closure of $\rightsquigarrow$.

Theorem 3.6 (GIL Soundness)**.** *Let $\langle \lfloor, \sim_s, \sim_v \rangle$ be a soundness relation between $\hat{S} = \langle |\hat{S}|, \hat{\mathbb{V}}, A, \widehat{ea} \rangle$ and $S = \langle |S|, \mathbb{V}, A, ea \rangle$ and $\leq$ the pre-order induced by $\sim_s$. Then, the following hold:*

GIL - Restricted Soundness
$$\mathsf{p} \vdash \hat{cf} \rightsquigarrow^n \hat{cf}' \land (\hat{cf}\lfloor_{\hat{cf}'}) \sim_s cf \land \mathsf{p} \vdash cf \rightsquigarrow^n cf' \implies \hat{cf}' \sim_s cf'$$

GIL - Restricted Completeness
$$\mathsf{p} \vdash \hat{cf} \rightsquigarrow^n \hat{cf}' \land (\hat{cf}\lfloor_{\hat{cf}'}) \sim_s cf \implies \exists cf'. \mathsf{p} \vdash cf \rightsquigarrow^n cf'$$

These properties state that if we have an abstract n-trace such that the initial concrete configuration cf is over-approximated by the initial abstract configuration strengthened with information from the final abstract configuration, $\hat{cf}\lfloor_{\hat{cf}'}$, then all concrete n-traces starting from cf will have their final configuration over-approximated by the final abstract configuration, and at least one such n-trace must exist. As a consequence, any bugs found by the abstract execution must

also exist in the concrete execution, meaning that the symbolic execution of Gillian has no false positive bug reports.

Trace Composition. The set of abstract traces that satisfy the criteria given in Theorem 3.6 is, in fact, broader than those captured by the $\rightsquigarrow^*$ relation. In particular, at any point during trace construction, we can extend the current configuration with additional information that does not conflict with what is already known. We capture this via a relaxed closure operator for trace construction, $\rightsquigarrow^*$, for which we can easily prove a soundness result analogous to Theorem 3.6:

$$\textsc{Reflexivity}\quad cf \rightsquigarrow^* cf \qquad \textsc{One-Step}\quad \frac{cf_1 \rightsquigarrow cf_2}{cf_1 \rightsquigarrow^* cf_2} \qquad \textsc{Composition}\quad \frac{cf_1 \rightsquigarrow^* cf_1' \quad cf_2 \rightsquigarrow^* cf_2' \quad cf_1' \downarrow_{cf_2} = cf_2}{cf_1 \rightsquigarrow^* cf_2'}$$

In the context of symbolic states, $cf_1' \downarrow_{cf_2} = cf_2$ means that, at any point during the construction of the symbolic trace, we may safely add more information to the current path condition. This gives us permission to arbitrarily drop paths in the analysis by need, a technique commonly used for achieving better scalability of symbolic execution tools.

Relaxed trace composition also allows us to use symbolic execution summaries [24] and separation-logic specifications [**?**]. The latter, however, are useful only if the memory model satisfies the frame property [49], allowing us to frame off the irrelevant part of the caller state.

3.2 Concrete-Symbolic Soundness

We establish a set of soundness relations between symbolic and concrete memory models, $\mathcal{R}_{\mathbb{M}}$, and a set of soundness relations between symbolic and concrete allocators, $\mathcal{R}_{\mathbb{AL}}$, and, in Theorem 3.10, prove that, together, they can be automatically lifted to soundness relations between the corresponding symbolic and concrete state models, $\mathcal{R}_{\mathbb{S}}$. Given this theorem and Gillian's built-in allocators, a tool developer needs only to provide a soundness relation connecting their symbolic memory to their concrete memory in order to prove the soundness of the resulting GIL symbolic analysis.

Memory Interpretation. Recall the definitions of concrete and symbolic memory models from §2.3. The interpretation of a symbolic memory model $\hat{M} = \langle |\hat{M}|, A, \underline{\hat{ea}} \rangle$ w.r.t. a concrete memory model $M = \langle |M|, A, \underline{ea} \rangle$ is given by a memory interpretation function $\mathcal{I}$ that takes a *logical environment*, $\varepsilon : \hat{\mathcal{X}} \rightharpoonup \mathcal{V}$, mapping logical variables to concrete values, and a symbolic memory $\hat{\mu} \in |\hat{M}|$, and generates a concrete memory $\mu \in |M|$. Memory interpretation functions must link symbolic memory actions to concrete memory actions so that they satisfy the appropriate restricted soundness and completeness properties on actions. We write $[\![\hat{e}]\!]_\varepsilon$ to denote the interpretation of the logical expression $\hat{e}$ under ε.

Definition 3.7 (Memory Interpretation)**.** Given a symbolic memory model $\hat{M} = \langle |\hat{M}|, A, \underline{\hat{ea}} \rangle$ and a concrete memory model $M = \langle |M|, A, \underline{ea} \rangle$, a *memory interpretation function* $\mathcal{I} : (\hat{\mathcal{X}} \rightharpoonup \mathcal{V}) \rightharpoonup |\hat{M}| \to |M|$ must satisfy:

Memory Action - Restricted Soundness (MA-RS)
$$\hat{\mu}.\alpha(\hat{e}, \pi) \rightsquigarrow (\hat{\mu}', \hat{e}', \pi') \land \mu = \mathcal{I}(\varepsilon, \hat{\mu}) \land [\![\pi \land \pi']\!]_\varepsilon = \text{true} \land \mu.\alpha([\![\hat{e}]\!]_\varepsilon) \rightsquigarrow (\mu', v) \implies \mu' = \mathcal{I}(\varepsilon, \hat{\mu}') \land v = [\![\hat{e}']\!]_\varepsilon$$

Memory Action - Restricted Completeness (MA-RC)
$$\hat{\mu}.\alpha(\hat{e}, \pi) \rightsquigarrow (\hat{\mu}', \hat{e}', \pi') \land \mu = \mathcal{I}(\varepsilon, \hat{\mu}) \land [\![\pi \land \pi']\!]_\varepsilon = \text{true} \implies \exists \mu', v\,.\, \mu.\alpha([\![\hat{e}]\!]_\varepsilon) \rightsquigarrow (\mu', v)$$

A memory interpretation function $\mathcal{I}$ induces a relation $R_{\mathcal{I}}$ between symbolic and concrete memories, such that $\hat{\mu}\ R_{\mathcal{I}}\ \mu$ *iff* there is a logical environment ϵ for which $\mathcal{I}(\varepsilon, \hat{\mu}) = \mu$; we denote the set of such relations by $\mathcal{R}_{\mathbb{M}}$.

Allocator Interpretation. Recall the definitions of concrete and symbolic allocators from §2.3. The interpretation of a symbolic allocator, $\hat{AL} = \langle |\hat{AL}|, \hat{\mathcal{E}}, \text{alloc} \rangle$, with respect to a concrete allocator, $AL = \langle |AL|, \mathcal{V}, \text{alloc} \rangle$, is given by an allocator interpretation function, $\mathcal{I}_{AL}$, that receives as input a logical environment, ε, and a symbolic allocator record, $\hat{\xi} \in |\hat{AL}|$, and generates a concrete allocator record, $\xi \in |AL|$. Analogously to the actions of memory models, allocator interpretation functions must link symbolic allocation to concrete allocation in such a way that they satisfy the restricted soundness and completeness properties on alloc.

Definition 3.8 (Allocator Interpretation)**.** Given a symbolic allocator $\hat{AL} = \langle |\hat{AL}|, \hat{\mathcal{E}}, \text{alloc} \rangle$ and a concrete allocator $AL = \langle |AL|, \mathcal{V}, \text{alloc} \rangle$, an allocator interpretation function $\mathcal{I}_{AL} : (\hat{\mathcal{X}} \rightharpoonup \mathcal{V}) \rightharpoonup |\hat{AL}| \to |AL|$ must satisfy:

Allocator - Restricted Soundness (AL-RS)
$$\hat{\xi}.\text{alloc}(j) \to_Y (\hat{\xi}', \hat{e}) \land \hat{\xi}'' \sqsubseteq \hat{\xi} \downarrow_{\hat{\xi}'} \land \xi.\text{alloc}(j) \to_{[\![Y]\!]_\varepsilon} (\xi', v) \land \xi = \mathcal{I}_{AL}(\varepsilon, \hat{\xi}'') \implies \xi' = \mathcal{I}_{AL}(\varepsilon, \hat{\xi}' \downarrow_{\hat{\xi}''}) \land v = [\![\hat{e}]\!]_\varepsilon$$

Allocator - Restricted Completeness (AL-RC)
$$\hat{\xi}.\text{alloc}(j) \to_Y (\hat{\xi}', \hat{e}) \land \hat{\xi}'' \sqsubseteq \hat{\xi} \downarrow_{\hat{\xi}'} \land \xi = \mathcal{I}_{AL}(\varepsilon, \hat{\xi}'') \implies \exists \xi', v.\ \xi.\text{alloc}(j) \to_{[\![Y]\!]_\varepsilon} (\xi', v)$$

where $[\![Y]\!]_\varepsilon$ is shorthand for $\{[\![y]\!]_\varepsilon \mid y \in Y\}$.

The constraint $\hat{\xi}'' \sqsubseteq \hat{\xi} \downarrow_{\hat{\xi}'}$ is analogous to the constraint $\hat{\sigma}'' \leq \hat{\sigma} \downarrow_{\hat{\sigma}'}$, required for state action soundness in Definition 3.5. The set of soundness relations between symbolic and concrete allocators, $\mathcal{R}_{\mathbb{AL}}$, is defined analogously to $\mathcal{R}_{\mathbb{M}}$.

Parametric Lifting. We introduce a lifting mechanism for obtaining candidate soundness relations between symbolic and concrete state models given appropriate memory/allocator interpretation functions, and prove that those candidate relations are indeed soundness relations as per Definition 3.5.

Definition 3.9 (Lifted Relation)**.** Given a memory interpretation function $\mathcal{I} : (\hat{\mathcal{X}} \rightharpoonup \mathcal{V}) \rightharpoonup |\hat{M}| \to |M|$ and an allocator interpretation function $\mathcal{I}_{AL} : (\hat{\mathcal{X}} \rightharpoonup \mathcal{V}) \rightharpoonup |\hat{AL}| \to |AL|$, the lifted candidate soundness relation $\mathcal{L}(\mathcal{I}, \mathcal{I}_{AL}) = \langle \downarrow, \sim_s, \sim_v \rangle$

for $SSC_{AL}(\hat{M})$ with respect to $CSC_{AL}(M)$ is defined by:

$$\begin{array}{lcl} \langle \hat{\mu}, \hat{\rho}, \hat{\xi}, \pi \rangle \downarrow_{\langle -,-,\hat{\xi}',\pi' \rangle} & \triangleq & \langle \hat{\mu}, \hat{\rho}, \hat{\xi} \downarrow_{\hat{\xi}'}, \pi \ \wedge \ \pi' \rangle \\ \hat{\sigma} \sim_s \sigma & \triangleq & \exists\, \varepsilon \,.\, (\sigma, \varepsilon) \in \mathcal{M}od(\hat{\sigma}) \\ \langle -,-,-,\pi \rangle \vdash \hat{e} \sim_v v & \triangleq & \exists\, \varepsilon \,.\, [\![\pi]\!]_\varepsilon = \mathsf{true} \ \wedge \ [\![\hat{e}]\!]_\varepsilon = v \end{array}$$

where:

$$\mathcal{M}od(\langle \hat{\mu}, \hat{\rho}, \hat{\xi}, \pi \rangle) \triangleq \left\{ (\langle \mu, \rho, \xi \rangle, \varepsilon) \mid [\![\pi]\!]_\varepsilon = \mathsf{true} \wedge \mu = \mathcal{I}(\varepsilon, \hat{\mu}) \ \wedge \ \rho = [\![\hat{\rho}]\!]_\varepsilon \ \wedge \ \xi \sqsubseteq \mathcal{I}_{AL}(\varepsilon, \hat{\xi}) \right\}$$

Theorem 3.10 (Correctness of Lifting). *Given a memory interpretation function $\mathcal{I} : (\hat{\mathcal{X}} \rightharpoonup \mathcal{V}) \rightharpoonup |\hat{M}| \to |M|$ and an allocator interpretation function $\mathcal{I}_{AL} : (\hat{\mathcal{X}} \rightharpoonup \mathcal{V}) \rightharpoonup |\hat{AL}| \to |AL|$, the lifted relation $\mathcal{L}(\mathcal{I}, \mathcal{I}_{AL})$ is a soundness relation between $SSC_{AL}(\hat{M})$ and $CSC_{AL}(M)$.*

Combining Theorems 3.10 and 3.6, we conclude that lifted relations are preserved by the GIL semantics.

3.3 While: Sound Symbolic Analysis

The While interpretation function, $\mathcal{I}_{\mathbb{W}} : (\hat{\mathcal{X}} \rightharpoonup \mathcal{V}) \rightharpoonup |\hat{M}_{\mathbb{W}}| \to |M_{\mathbb{W}}|$, is defined straightforwardly as follows:

$$\textsc{Empty}\ \ \mathcal{I}_{\mathbb{W}}(\varepsilon, \emptyset) \triangleq \emptyset \qquad \textsc{Cell}\ \ \frac{l = [\![\hat{e}]\!]_\varepsilon \quad v = [\![\hat{e}']\!]_\varepsilon}{\mathcal{I}_{\mathbb{W}}(\varepsilon, \hat{e}.p \mapsto \hat{e}') \triangleq l.p \mapsto v} \qquad \textsc{Union}\ \ \frac{\mu_i = \mathcal{I}_{\mathbb{W}}(\varepsilon, \hat{\mu}_i)|_{i=1}^2}{\mathcal{I}_{\mathbb{W}}(\varepsilon, \hat{\mu}_1 \uplus \hat{\mu}_2) \triangleq \mu_1 \uplus \mu_2}$$

Lemma 3.11 states that $\mathcal{I}_{\mathbb{W}}$ preserves the actions of While, $A_{\mathbb{W}} = \{\text{lookup}, \text{mutate}, \text{dispose}\}$. Its proof is a straightforward case analysis on the rules of Figure 3, and is much simpler than the custom inductive proofs on semantic derivations that underpin standalone soundness proofs.

Lemma 3.11 (While Memory Interpretation). *The While interpretation function, $\mathcal{I}_{\mathbb{W}}$, is a memory interpretation function.*

Combining Lemma 3.11 and Theorems 3.6 and 3.10, we obtain that the While symbolic semantics satisfies restricted soundness and completeness w.r.t its concrete semantics.

4 Case Studies: JavaScript and C

We instantiate Gillian to obtain symbolic testing tools for JavaScript and C. Each instantiation requires a trusted compiler from the target language to GIL, and an OCaml implementation of its concrete and symbolic memory models. We discuss the coverage and trustworthiness of the resulting tools, and evaluate them by performing symbolic testing for the real-world data-structure libraries, Buckets.js [54] and Collections-C [50], detecting bugs in both, in times that indicate that our analysis should scale to larger codebases. We also discuss the workload for each instantiation, and conclude with an evaluation of the overall usability of Gillian.

4.1 Gillian-JS

Compiler. The Gillian-JS compiler from ECMAScript 5 Strict (ES5 Strict) to GIL uses the JaVerT compiler [52] to compile ES5 Strict to JaVerT's intermediate goto language, JSIL, and a straightforward compiler from JSIL to GIL, developed here. The JaVerT compiler includes implementations of the internal and built-in functions of ES5 Strict in JSIL, which get compiled to GIL. The coverage of the Gillian-JS compiler is broad, inherited from JaVerT. It covers the entire core language except the regular expression literal, and the majority of the built-in libraries. It does not support: the Date, RegExp and JSON libraries; parts of the String library that use regular expressions; and global object functionalities related to URIs and parsing of numerics. Additionally, indirect eval is not supported, as it is meant to be executed as non-strict code, and all Function constructor code runs as strict-mode code.

The trustworthiness of the Gillian-JS compiler is established following the methodology introduced for JaVerT [52, 53]: it preserves the ES5 Strict memory model; it follows the standard faithfully line-by-line with the control flow of JavaScript trivially compiled to the control flow of GIL; and it is thoroughly tested against Test262 [18], the official JavaScript test suite. Here, we filter the latest version of the test suite for the tests applicable to ES5 Strict, identifying 9013 and passing 9005 tests. The failing tests are all due to a discrepancy between how Unicode characters are treated in JavaScript (either UCS-2 or UTF-16) and OCaml (sequences of bytes). A detailed breakdown is available online [61].

Memory Models. Our OCaml implementation of the memory models comes from JaVerT 2.0 [53], and is adapted slightly to the setting of Gillian. Here, we give the theoretical account, which the implementation closely follows; for space reasons, we focus only on the part relevant for symbolic execution.[4]

A concrete JS memory, $\mu \in |M_{\mathbb{JS}}|$, is a pair comprising a concrete heap and a concrete metadata table. A concrete heap, $h : \mathcal{U} \times \mathcal{S} \rightharpoonup \mathcal{V}$, maps object locations and property names to values, with object locations represented by uninterpreted symbols and property names by strings. A concrete metadata table, $m : \mathcal{U} \rightharpoonup \mathcal{V}$, which maps objects to values that hold useful information about these objects, and is used for modelling the internal properties of JS objects [53].

A symbolic JS memory, $\hat{\mu} \in |\hat{M}_{\mathbb{JS}}|$, is a pair comprising a symbolic heap and a symbolic metadata table. A symbolic heap, $\hat{h} : \hat{\mathcal{E}} \times \hat{\mathcal{E}} \rightharpoonup \hat{\mathcal{E}}$, maps pairs of logical expressions to logical expressions. Unlike for While, where objects are static, property names also have to be logical expressions, since JavaScript has dynamic property access. Analogously, the signature of symbolic metadata tables is $\hat{m} : \hat{\mathcal{E}} \rightharpoonup \hat{\mathcal{E}}$.

Our JS memory model has eight actions, which include retrieval/update/deletion of object properties and metadata, and the creation and deletion of objects. Below, we give

[4] The full memory model that we use also includes a mechanism for capturing property absence, as in [51–53], which ensures that the JS semantics respects the frame property [49] in the presence of extensible objects. This mechanism is required for the verification and bi-abduction aspects of Gillian, which are out of scope for this paper.

one of the rules for the symbolic getProp action, which receives a symbolic object location and a property name, and retrieves the symbolic value of the property. This rule is non-deterministic; it allows the execution to branch on the looked-up property $\hat{e}_p$ of object $\hat{e}_l$ equalling an arbitrary property $\hat{e}_i$ of an arbitrary object $\hat{e}'_l$ in the heap, if that equality is permitted by the path condition. Note how the branching condition, $\hat{e}_p = \hat{e}'_p \wedge \hat{e}_l = \hat{e}'_l$, is passed back to the state.

$$\textsc{SGetProp - Branch - Found}\quad \frac{\hat{\mu} = (\hat{h}, \hat{d}, \_) \quad \hat{h} = \_ \uplus (\hat{e}'_l.\hat{e}_i \mapsto \hat{e}) \quad \pi \wedge (\hat{e}_l = \hat{e}'_l \wedge \hat{e}_p = \hat{e}'_p)\ \mathsf{SAT}}{\hat{\mu}.\mathsf{getProp}([\hat{e}_l, \hat{e}_p], \pi) \rightsquigarrow (\hat{\mu}, \hat{e}, \hat{e}_l = \hat{e}'_l \wedge \hat{e}_p = \hat{e}'_p)}$$

Evaluation. There are currently no benchmarks with which to assess tools that provide symbolic testing of JavaScript. Therefore, as in [51, 53], we evaluate the symbolic testing of Gillian-JS using the real-world Buckets.js library, a self-contained data-structure library widely used by developers, with more than 65K downloads on npm [41]. Buckets.js contains approximately 1K lines of JavaScript code, uses almost every JS-specific feature, and comes with a unit test suite. It implements a variety of data structures, including linked lists, sets, multi-sets, maps, queues, and stacks.

We re-use our symbolic test suite for Buckets.js [51, 53, 60]. which has 100% line coverage, and whose symbolic tests were purposefully written to cover multiple execution traces. We run the test suite using Gillian-JS, obtaining results presented in Table 1. We report, per data-structure: the number of symbolic tests (#*T*); the number of executed GIL commands; the times of JaVerT 2.0 [53] (J2); and the times of Gillian-JS (GJS). The Gillian-JS times are roughly twice as fast as those of JaVerT 2.0, because of our improvements to the symbolic execution engine (e.g., more efficient use of OCaml features, such as hashtables), and the first-order solver (e.g., better simplifications and better caching of results). Our testing has not found any additional bugs in Buckets.js, but was able to detect the two bugs found in our previous work [51, 53].

Table 1. Buckets.js

Name	#T	GIL Cmds	Time (J2)	Time (GJS)
`array`	9	330,147	5.02s	**2.58s**
`bag`	7	1,343,393	10.50s	**4.81s**
`bst`	11	3,751,092	18.67s	**10.37s**
`dict`	7	401,575	4.25s	**1.88s**
`heap`	4	1,492,204	5.70s	**2.93s**
`llist`	9	588,714	7.56s	**4.00s**
`mdict`	6	1,106,650	8.94s	**3.77s**
`pqueue`	5	2,312,226	8.60s	**3.87s**
`queue`	6	407,106	4.58s	**2.09s**
`set`	6	2,178,222	16.49s	**4.46s**
`stack`	4	306,449	3.28s	**1.66s**
`Total`	74	14,217,778	93.59s	**42.42s**

Workload. The Gillian-JS compiler comprises 7694 lines of OCaml, and the implementations of the concrete and symbolic memory models of ES5 Strict comprise 1620 lines of OCaml. As we re-used the JaVerT compiler [52], the associated workload was minimal: out of the 7694 lines, only 347, for the compiler from JSIL to GIL, are entirely new. We adapted the Gillian-JS memory models from JaVerT 2.0 so that they match the expected interface signatures and the slightly different treatment of the errors. As Gillian-JS was developed in parallel with the core of Gillian, we cannot say precisely how long it took us to create it: we estimate that a tool developer experienced in the intricacies of JavaScript would be able to implement the compiler from scratch in approximately three months, and the memory models within a month. Finally, the writing of the symbolic tests for Buckets.js took us approximately two weeks.

4.2 Gillian-C

Compiler. The Gillian-C compiler uses the verified C compiler, CompCert [33], from a large fragment of ISO C 99 to C#minor, an intermediate representation of CompCert, and a compiler from C#minor to GIL, developed here, which trivially compiles the control flow constructs and restates memory management in terms of the identified actions of the C memory model. We compile to C#minor for two main reasons: C#minor still preserves the memory model of C; and this compilation deviates from the C standard only by fixing the order of argument evaluation. The former is essential for the Gillian approach, while the latter means that the analysis of Gillian-C does not lose the potential to catch various common bugs, such as those due to undefined behaviour.

Gillian-C covers the features supported by CompCert, implemented by need with the goal of analysing the Collections-C library. This effectively means that we do not support several arithmetic operators, and that, when it comes to the functions of the standard library, we have implemented only `calloc`, `free`, `malloc`, `memcpy`, `memmove`, `memset`, and `strcmp`. We will implement other operators and library functions, as well as operating-system calls, by need. We also plan to formalise GIL in Coq and re-use the proof techniques of CompCert to obtain a fully certified Gillian-C compiler.

Memory Model. We use the OCaml implementation of the extracted concrete memory model of CompCert [33], adding a thin layer on top for encoding C values in GIL, and provide an OCaml implementation of the symbolic memory model, taking inspiration from CompCertS [3], an extension of CompCert with symbolic values. Here, we give a theoretical account. In both models, the memory is a collection of separated blocks where each block is an array of a given size. Pointers are represented as block-offset pairs: that is, a pointer $[l, off]$ points to the off-th element of the memory block l. For clarity, we elide the details of the memory models related to concurrency, which Gillian does not support.

A concrete C memory, $\mu \in |M_{\mathbb{C}}|$, is a pair comprising a concrete heap and a concrete permission table. In CompCert, C values, such as integers, floats, and pointers, are stored in memory as sequences of byte-sized *memory values*, allowing for fine-grained memory access. In particular, a C memory value, $mv \in \mathcal{V}$, is either: a byte, $b \in [0, 255]$; the special value `undefined`, denoting uninitialised memory; or

a three-element list $[l, off, k]$, denoting the k-th byte of the pointer $[l, off]$. A concrete heap, $h : \mathcal{U} \times \mathbb{Z} \rightharpoonup \mathcal{V}$, maps cells (block-offset pairs) to C memory values, with blocks encoded as uninterpreted symbols and offsets as integers. A concrete permission table, $d : \mathcal{U} \times \mathbb{Z} \rightharpoonup \mathbb{N}$, maps cells to their permissions, which describe the allowed operations for a given cell (e.g. `Readable` and `Writable`). We model permissions as integers, in ascending order of permisiveness.

A symbolic C memory, $\hat{\mu} \in |\widehat{M}_{\mathbb{C}}|$, is a pair comprising a symbolic heap and a symbolic permission table. A symbolic heap, $\hat{h} : (\hat{\mathcal{E}} \times \hat{\mathcal{E}}) \rightharpoonup \hat{\mathcal{E}}$, models symbolic blocks and offsets using logical expressions, and symbolic memory values, $\hat{mv} \in \hat{\mathcal{E}}$, using three-element lists, $[\hat{e}, k, n]$, which denote the k-th out of n bytes of the C value represented by the logical expression $\hat{e}$; as we always statically know the size of a value, the k and n will always be concrete. This unified treatment of memory values, introduced by CompCertS [3], could also be applied to the CompCert concrete memory model. Analogously to symbolic heaps, the signature of symbolic permission tables is $\widehat{pt} : \hat{\mathcal{E}} \times \hat{\mathcal{E}} \rightharpoonup \hat{\mathcal{E}}$.

SLoad - Valid Access

$$\frac{\hat{\mu} = (\hat{h}, \widehat{pt}) \quad \pi \vdash \hat{e}_l = \hat{e}'_l \wedge \hat{e}_o = \hat{e}'_o \quad \pi \vdash \hat{e}'_o \bmod al = 0 \quad (\pi \vdash \widehat{pt}(\hat{e}'_l, \hat{e}_o + i) \geq \mathtt{Readable})|_{i=0}^{sz-1} \quad (\hat{h}(\hat{e}'_l, \hat{e}_o + i) = [\hat{e}', i, sz-1])|_{i=0}^{sz-1} \quad \hat{e} = \mathsf{decodeSymb}(\hat{e}', type)}{\hat{\mu}.\mathsf{load}([[sz, al, type], \hat{e}_l, \hat{e}_o], \pi) \rightsquigarrow (\hat{\mu}, \hat{e}, \mathsf{true})}$$

The CompCert C memory model has sixteen actions, which account for the management of the global environment, the heap, and the permission table. Above, we present one rule for the symbolic load action, which retrieves a value from the memory. When *loading*/*storing* a value, a memory chunk has to be provided to indicate the size, alignment, and type of the value to be read from/written to the memory. We present chunks as three-element lists: $mch = [sz, al, type]$. The load function receives a memory chunk, the location, and the offset. First, it ensures that the value is correctly aligned and readable. Next, it confirms that the read part of the memory represents the symbolic value $\hat{e}'$. Finally, it decodes $\hat{e}'$ using its type. The decoding understands, e.g., if the result should be an integer or a float, and of which precision.

Table 2. Collections-C

Name	#T	GIL Cmds	Time
`array`	22	109,290	**4.21s**
`deque`	34	106,737	**6.57s**
`list`	37	730,655	**13.02s**
`pqueue`	2	15,726	**0.64s**
`queue`	4	39,828	**0.65s**
`rbuf`	3	27,284	**0.52s**
`slist`	38	325,383	**7.18s**
`stack`	2	5,211	**0.28s**
`treetbl`	13	618,326	**2.98s**
`treeset`	6	108,583	**3.29s**
`Total`	161	2,097,023	**39.34s**

Evaluation. We evaluate Gillian-C by symbolically testing Collections-C [50], a real-world data-structure library for C with almost 2K stars on GitHub. It has approximately 5.2K lines of code and uses C-specific constructs and idioms, such as structures and pointer arithmetic. The data structures it provides include, for example, arrays, lists, treetables, hashtables, ring buffers and queues.

We write an extensive symbolic test suite for Collections-C, with results shown in Table 2. We report, per data structure: **(1)** the number of symbolic tests (#T); **(2)** the number of executed GIL commands; and **(4)** the obtained testing times for Gillian-C. Our testing has revealed the following issues, which have been fixed by the developers of Collections-C:

1. a buffer overflow bug in the implementation of dynamic arrays, caused by an off-by-one index;
2. usage of undefined behaviours (pointer comparison, in particular) that can lead to buggy behaviours in the presence of compiler optimisations;
3. several bugs in the concrete test suite: in particular, comparing freed pointers, unchecked function returns, and incorrect checks with serendipitously correct values;
4. over-allocation in the ring-buffer data structure, but with correct behaviour of the associated functions.
5. a bug in the string hashing function for hashtables that could lead to performance loss.

These initial results show that Gillian-C can handle the complexity of C's memory model on a real-world, albeit modest-in-size, example. In future, when Gillian-C is more mature, we will compare its coverage and performance against appropriate C tools, such as CBMC [31] and KLEE [9].

Workload. Gillian-C uses the CompCert compiler from ISO C 99 to C#minor, and the concrete memory model of CompCert. We additionaly wrote the compiler from C#minor to GIL (1451 lines of OCaml); the compiler runtime, consisting of GIL implementations of unary/binary operators and standard library functions (1121 lines of GIL); a wrapper around the CompCert concrete memory model (304 lines of OCaml); and the entire symbolic memory model (1618 lines of OCaml). This development took us approximately six weeks: three for the compiler/runtime; and three for the memory models. Additionally, the writing of the symbolic tests for Collections-C took us approximately two weeks.

Current Limitations. As Gillian-C is a proof-of-concept tool, it comes with several limitations. For example, we do not reason about allocation of symbolic size and do not model the cases in which the memory management functions fail. The former is an open research problem that we plan to address taking inspiration from similar work done for Frama-C [30]. We also do not yet model all of the undefined behaviour (UB) of C. In particular, we model the UB for static and dynamic memory management well, as demonstrated by the discovered bugs in Collections-C, but need to improve substantially on the UB related to arithmetic. Further, as our first-order solver cannot reason about hash functions, we are not able to test the `hashtbl` and `hashset` data structures of Collections-C. We have, however, written the appropriate tests, some of which surprisingly passed, revealing the above-mentioned bug. We are working on applying the approach

of KLEE [29] in order to solve this issue. Finally, we do not support any concurrency-related features of C, as Gillian, for the moment, only handles sequential programs.

4.3 Usability

To instantiate Gillian to a new target language (TL), a tool developer must provide a trusted compiler from the TL to GIL, and OCaml implementations of the concrete and symbolic memory models of the TL. Thus, they must have an in-depth understanding of the language standard, a working knowledge of OCaml, and a basic understanding of the interface of the Gillian memory models. We are working on developing a template language and auxiliary functions that would streamline this process.

To use existing instantiations of Gillian, a general developer writes symbolic tests in the style of Rosette and KLEE; a tutorial can be found online [61]. Our experience with the creation of symbolic test suites for Buckets.js and Collections-C suggests that the use of Gillian by general developers is within reach. However, in order for this to materialise, we first have to improve its debugging and error reporting mechanisms, as the produced logs are lengthy and the information is not lifted back from GIL to the TL; the latter, in particular, will be easier to achieve for Gillian than for other IR-based tools, as the TL memory model is maintained by the compilation. We are also working on enabling Gillian to be used in IDEs as well as in a continuous-integration setting, so that developers would be able to get real-time feedback and automatically run symbolic test suites as their codebase changes.

5 Related Work

We centre our discussion on: **(1)** symbolic lifting frameworks; **(2)** semantic frameworks; **(3)** multi-language tools with symbolic intermediate representations (IRs); **(4)** parametric frameworks for abstract interpretation; and **(5)** specific symbolic execution tools for JavaScript and C. The aim of Gillian is similar to the framework approach of (1), (2) and (4) in that an independent tool developer instantiates Gillian with a particular target language (TL), using a symbolic IR in the style of the multi-language tools of (3).

Symbolic Lifting Frameworks. Symbolic-lifting frameworks, such as Rosette and Chef, automatically lift a concrete TL interpreter into a symbolic interpreter. Rosette [62, 63] extends Racket [46] with solver-aided facilities for creating symbolic values and expressing constraints on those values. With Rosette, the concrete TL interpreter is written in Racket and is then *symbolically interpreted* using Rosette's core symbolic execution engine. Rosette has been successfully applied to domain-specific languages [6, §5], and has found bugs in parts of the Linux kernel [40]. Chef [8] takes a specially-packaged TL interpreter as input and analyses TL programs by symbolically executing the interpreter's *binary*. Chef has been applied so far to dynamic languages, such as Python and Lua. However, the authors comment that its applicability is limited to languages with moderately-sized interpreters. Chef would, therefore, not be an appropriate tool for analysis of, for example, Java programs.

Semantic Frameworks. *Semantic frameworks*, such as $\mathbb{K}$ [21], Ott [58], Lem [39] and Redex [20], provide specification languages in which users can write the semantics of their TL and automatically generate various tools from this semantics, ranging from interpreters and compilers for multiple backends to sophisticated program analysers. Among these frameworks, $\mathbb{K}$ has the agenda closest to ours, as it automatically generates tools that support various forms of program analysis, including symbolic execution and deductive verification. $\mathbb{K}$ has so far been instantiated to a number of programming languages, including JavaScript and C [25, 43, 59], and is also used in industry for the symbolic analysis of Ethereum bytecode [26, 44].

IR-Based Multi-Language Tools. IR-based multi-language symbolic analysis tools, such as the academic tools Viper [37, 38] and CoreStar [7], and Facebook's Infer [13], compile high-level TLs into simpler IRs on which the analysis is performed. None of these tools is parametric on the TL memory model. Instead, to extend a tool to a new TL, the in-house tool developer must encode the TL memory model into the IR memory model, the difficulties of which depend on how close the memory models and the TL are. Moreover, the correctness of the compilers used by IR-based tools is often not justified. In our approach, the memory models of the TL and the IR are the same by design, and the compiler from the TL to the IR is required to be trustworthy. This both simplifies correctness proofs and allows us to have optimal performance by tailoring the symbolic reasoning for each TL.

The SAW tool [17], developed by Galois, is based on a slightly different approach, which is probably closest to ours. SAW uses several built-in memory models of low-level languages, such as LLVM and JVM. It has a modular implementation where an in-house tool developer can extend the tool with a new low-level memory model, but offers no mechanism for instantiation with a user-provided memory model.

Parametric Frameworks for Abstract Interpretation. A key insight of Gillian is to make GIL parametric on the TL memory model, splitting its semantics into the control-flow/variable store component provided by the platform and the memory component implemented by the tool developer. Similar decompositions have arisen in the context of abstract interpretation [15], but we believe we are the first to use it in the design of a general symbolic execution tool.

We can coarsely divide the existing parametric abstract interpretation frameworks into two groups: those based on small-step semantics [16, 27, 28, 36, 57] and those based on co-inductive big-step semantics [4, 56]. *Small-step frameworks* follow a general methodology, first proposed in [36],

for deriving sound and computable abstract interpreters from a given concrete interpreter written in small-step style. This methodology has not been fully automated so far, with its application requiring non-negligible developer effort. *Big-Step frameworks* follow an alternative approach, first outlined in [56], which introduces a methodology for designing abstract interpreters based on co-inductively defined big-step semantics. Following similar ideas, Bodin et al. [4] developed a general *skeletal semantics* framework in Coq, for creating concrete and abstract big-step semantics connected with a general consistency result, leaving the user to prove only a number of language-dependent lemmas. This work, however, has only been applied to a simple While language, making its broader applicability difficult to assess.

Symbolic Execution Tools for JavaScript and C. Most existing symbolic execution tools for JavaScript are aimed at bug-finding and target specific types of bugs, such as security vulnerabilities related to the misuse of strings [55], malformed Web API requests [64], DOM API specific bugs [34], or bugs involving regular expressions [35]. These tools aim at code in the large, primarily focusing on scalability and coverage issues. None of them, however, follows the JS semantics precisely, often ignoring the full complexity of the standard to achieve better scalability. In contrast, Gillian-JS tool is a general symbolic execution tool: the concrete and symbolic JS memory models have been implemented directly, the Gillian-JS compiler precisely follows the JS semantics and the resulting tool has been fully tested. In this respect, the work closest to ours is JaVerT 2.0 [53], and indeed Gillian-JS inherits the trusted JaVerT compiler [52, 53]. While Gillian-JS and JaVerT 2.0 are built on the same principles, Gillian-JS is approximately twice as fast, due to improvements made in the symbolic execution engine.

There are many mature symbolic execution tools for C [9, 10, 22–24, 47], the majority of which follows the *dynamic/-concolic* discipline pioneered by Dart [22]. Such tools pair up symbolic execution and concrete execution to allow the symbolic execution to fall back to the concrete whenever it produces symbolic formulae unsupported by the underlying constraint solver. They often combine symbolic execution with additional techniques, such as re-use of summaries [24] and lazy initialisation [10, 19, 47] to achieve better coverage/performance. Gillian also supports summary re-use for verification and lazy initialisation in the form of bi-abduction. This discussion, however, is out of the scope of this paper.

6 Conclusions and Further Work

We have introduced Gillian, a multi-language platform for the development of symbolic analysis tools based on our intermediate goto language GIL. We have demonstrated that our parametrisation on the memory models is viable by instantiating Gillian to obtain symbolic testing engines Gillian-JS and Gillian-C. Both these instantiations have been used to find bugs in real-world code in times that indicate that the analysis of Gillian can scale to larger codebases. Our work on JavaScript is mature, building on previous work in the JaVerT project [52, 53]. In contrast, our work on C is on-going. We plan to improve Gillian-C based on CompCert, to explore a different Gillian-C based on Cerberus [32] by compiling its simpler Core language to GIL, and to compare with established specialist C tools such as CBMC [31] and KLEE [9] using accepted benchmarks. We are also investigating the possibility of using Gillian to analyse Rust programs.

We have given a formal account of the GIL semantics that links closely to the modular OCaml implementation; this connection was an important aim for us. We have given a parametric definition of soundness and used it to prove that the symbolic analysis provided by Gillian is sound and has no false positives. This was made possible by our novel concept of restriction, which generalises the path conditions of classical symbolic execution. *** PM : fundamental ***

The infrastructure presented in this paper substantially eases the burden of the tool developer when instantiating Gillian to a new target language. To make the Gillian instantiations more accessible to the general developer, we plan to improve its error reporting mechanisms and develop interactive tools (e.g. a trace visualiser and a code-stepper) for the debugging of analysed programs.

Gillian is open-sourced and is available online [61]. It already supports semi-automatic verification based on separation logic and automatic compositional testing based on bi-abduction [**?**], unified by the underlying symbolic execution presented here; these additional analyses will be presented in a separate paper. In future, we hope to support incorrectness reasoning introduced in [42], encouraged by our identification of trace composition in §3.2. We also plan to extend Gillian with support for reasoning about complex language features, such as events and concurrency, as well as with additional forms of analysis, such as concolic execution; Gillian's modular design lends itself well to these extensions.

Acknowledgments

We would like to thank the reviewers, whose comments have improved the overall quality of the paper. Fragoso Santos, Gardner, and Maksimović were partially supported by the EPSRC Programme Grant 'REMS: Rigorous Engineering for Mainstream Systems' (EP/K008528/1) and the EPSRC Fellowship 'VetSpec: Verified Trustworthy Software Specification' (EP/R034567/1). Fragoso Santos was partially supported by national funds through Fundação para a Ciência e a Tecnologia (FCT), with reference UIDB/50021/2020 (INESC-ID multi-annual funding). Maksimović was partially supported by the UK National Cyber Security Centre, grant reference RFA20789: 'Gillian: A General Verification and Testing Framework'. Ayoun was supported by an Imperial College Department of Computing PhD Scholarship.

References

[1] R. Baldoni, E. Coppa, D. Cono D'Elia, C. Demetrescu, and I. Finocchi. 2018. A Survey of Symbolic Execution Techniques. *ACM Computing Surveys* 51, 3 (2018), 50:1–50:39.

[2] A. Banerjee and D. A. Naumann. 2002. Secure Information Flow and Pointer Confinement in a Java-like Language. In *CSFW*.

[3] F. Besson, S. Blazy, and P. Wilke. 2017. CompCertS: A Memory-Aware Verified C Compiler Using Pointer as Integer Semantics. In *ITP*.

[4] M. Bodin, P. Gardner, T. Jensen, and A. Schmitt. 2019. Skeletal Semantics and their Interpretations. *PACMPL* 3, POPL (2019), 44:1–44:31.

[5] D. Bogdanas and G. Rosu. 2015. K-Java: A Complete Semantics of Java. In *POPL*.

[6] J. Bornholt and E. Torlak. 2018. Finding Code that Explodes under Symbolic Evaluation. *PACMPL* 2, OOPSLA (2018), 149:1–149:26.

[7] M. Botinčan, D. Distefano, M. Dodds, R. Grigore, D. Naudžiūnienė, and M. J. Parkinson. 2011. coreStar: The Core of jStar. In *Boogie*.

[8] S. Bucur, J. Kinder, and G. Candea. 2014. Prototyping Symbolic Execution Engines for Interpreted Languages. In *ASPLOS*.

[9] C. Cadar, D. Dunbar, and D. R. Engler. 2008. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs.. In *OSDI*.

[10] C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill, and D. R. Engler. 2008. EXE: Automatically Generating Inputs of Death. *ACM Transactions on Information and System Security* 12, 2 (2008), 10:1–10:38.

[11] C. Cadar, P. Godefroid, S. Khurshid, C. S. Păsăreanu, K. Sen, N. Tillmann, and W. Visser. 2011. Symbolic Execution for Software Testing in Practice: Preliminary Assessment. In *ICSE*.

[12] C. Cadar and K. Sen. 2013. Symbolic Execution for Software Testing: Three Decades Later. *Commun. ACM* 56 (2013), 82–90.

[13] C. Calcagno and D. Distefano. 2011. Infer: An Automatic Program Verifier for Memory Safety of C Programs. In *NASA Formal Methods Symposium*.

[14] C. Calcagno, D. Distefano, P. W. O'Hearn, and H. Yang. 2011. Compositional Shape Analysis by Means of Bi-Abduction. *JACM* 58 (2011), 26:1–26:66.

[15] P. Cousot and R. Cousot. 1977. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *POPL*.

[16] D. Darais, M. Might, and D. Van Horn. 2015. Galois transformers and modular abstract interpreters: reusable metatheory for program analysis. In *OOPSLA*.

[17] R. Dockins, A. Foltzer, J. Hendrix, B. Huffman, D. McNamee, and A. Tomb. 2016. Constructing Semantic Models of Programs with the Software Analysis Workbench. In *VSSTE*.

[18] ECMA TC39. 2017. Test262 Test Suite. https://github.com/tc39/test262.

[19] D. Engler and D. Dunbar. 2007. Under-constrained Execution: Making Automatic Code Destruction Easy and Scalable. In *ISSTA*.

[20] R. B. Findler, M. Klein, B. Fetscher, and M. Felleisen. 2018. *Redex: Practical Semantics Engineering*. Technical Report.

[21] G. Roşu and T. Florin Şerbănuţă . 2010. An Overview of the K Semantic Framework. *Journal of Logic and Algebraic Programming* 79, 6 (2010), 397–434.

[22] P. Godefroid, N. Klarlund, and K. Sen. 2005. DART: Directed Automated Random Testing. In *ACM Sigplan Notices*.

[23] P. Godefroid, M. Y. Levin, and D. A. Molnar. 2008. Automated Whitebox Fuzz Testing. In *NDSS*.

[24] P. Godefroid, A. V. Nori, S. K. Rajamani, and S. Tetali. 2010. Compositional May-must Program Analysis: Unleashing the Power of Alternation. In *POPL*.

[25] C. Hathhorn, C. Ellison, and G. Rosu. 2015. Defining the undefinedness of C. In *PLDI*.

[26] E. Hildenbrandt, M. Saxena, N. Rodrigues, X. Zhu, P. Daian, D. Guth, B. M. Moore, D. Park, Y. Zhang, A. Stefanescu, and G. Rosu. 2018. KEVM: A Complete Formal Semantics of the Ethereum Virtual Machine. In *CSF*.

[27] D. Van Horn and M. Might. 2010. Abstracting Abstract Machines. In *ICFP*.

[28] D. Van Horn and Matthew Might. 2012. Systematic Abstraction of Abstract Machines. *J. Funct. Program.* 22, 4-5 (2012), 705–746.

[29] T. Kapus and C. Cadar. 2019. A Segmented Memory Model for Symbolic Execution. In *ESEC/FSE*.

[30] F. Kirchner, N. Kosmatov, V. Prevosto, J. Signoles, and B. Yakobowski. 2015. Frama-C: A software analysis perspective. *Formal Aspects of Computing* 27, 3 (2015), 573–609.

[31] D. Kroening and M. Tautschnig. 2014. CBMC – C Bounded Model Checker. In *TACAS*.

[32] S. Lau, V. B. F. Gomes, K. Memarian, J. Pichon-Pharabod, and Sewell P. 2019. Cerberus-BMC: A Principled Reference Semantics and Exploration Tool for Concurrent and Sequential C. In *CAV*.

[33] X. Leroy, A. W. Appel, S. Blazy, and G. Stewart. 2012. *The CompCert Memory Model, Version 2*. Research Report RR-7987. INRIA. 26 pages.

[34] G. Li, E. Andreasen, and I. Ghosh. 2014. SymJS: Automatic Symbolic Testing of JavaScript Web Applications. In *FSE*.

[35] B. Loring, D. Mitchell, and J. Kinder. 2019. Sound Regular Expression Semantics for Dynamic Symbolic Execution of JavaScript. In *PLDI*.

[36] M. Might. 2010. Abstract Interpreters for Free. In *SAS*.

[37] P. Müller, M. Schwerhoff, and A. J. Summers. 2016. Viper: A Verification Infrastructure for Permission-Based Reasoning. In *VMCAI*.

[38] P. Müller, M. Schwerhoff, and A. J. Summers. 2017. Viper: A Verification Infrastructure for Permission-Based Reasoning. In *Dependable Software Systems Engineering*.

[39] D. P. Mulligan, S. Owens, K. E. Gray, T. Ridge, and P. Sewell. 2014. Lem: Reusable Engineering of Real-World Semantics. In *ICFP*.

[40] L. Nelson, J. Bornholt, R. Gu, A. Baumann, E. Torlak, and X. Wang. 2019. Scaling Symbolic Evaluation for Automated Verification of Systems Code with Serval. In *SOSP*.

[41] npm, Inc. 2018. npm, a Package Manager for JavaScript. https://www.npmjs.com.

[42] Peter W. O'Hearn. 2020. Incorrectness logic. *PACMPL* 4, POPL (2020), 10:1–10:32.

[43] D. Park, A. Stefanescu, and G. Rosu. 2015. KJS: a Complete Formal Semantics of JavaScript. In *PLDI*.

[44] D. Park, Y. Zhang, M. Saxena, P. Daian, and G. Rosu. 2018. A Formal Verification Tool for Ethereum VM Bytecode. In *FSE*.

[45] Y. Phang Khoo, B.-Y. E. Chang, and J. S. Foster. 2010. Mixing type checking and symbolic execution. In *PLDI*.

[46] Racket. 2017. The Racket Programming Language. racket-lang.org.

[47] D. A. Ramos and D. R. Engler. 2015. Under-Constrained Symbolic Execution: Correctness Checking for Real Code. In *USENIX Security Symposium*.

[48] M. Raza and P. Gardner. 2009. Footprints in Local Reasoning. *Logical Methods in Computer Science* 5, 2 (2009).

[49] J. C. Reynolds. 2002. Separation Logic: A Logic for Shared Mutable Data Structures. In *LICS*.

[50] S. Panić. 2014. Collections-C: A Library of Generic Data Structures. https://github.com/srdja/Collections-C.

[51] J. Fragoso Santos, P. Maksimovic, T. Grohens, J. Dolby, and P. Gardner. 2018. Symbolic Execution for JavaScript. In *PPDP*.

[52] J. Fragoso Santos, P. Maksimovic, D. Naudziuniene, T. Wood, and P. Gardner. 2018. JaVerT: JavaScript Verification Toolchain. *PACMPL* 2, POPL (2018), 50:1–50:33.

[53] J. Fragoso Santos, P. Maksimovic, G. Sampaio, and P. Gardner. 2019. JaVerT 2.0: Compositional Symbolic Execution for JavaScript. *PACMPL* 3, POPL (2019), 66:1–66:31.

[54] M. Santos. 2016. Buckets-JS: A JavaScript Data Structure Library. https://github.com/mauriciosantos/Buckets-JS.

[55] P. Saxena, D. Akhawe, S. Hanna, F. Mao, S. McCamant, and D. Song. 2010. A Symbolic Execution Framework for JavaScript. In *S&P*.
[56] D. A. Schmidt. 1995. Natural-Semantics-Based Abstract Interpretation (Preliminary Version). In *SAS*.
[57] I. Sergey, D. Devriese, M. Might, J. Midtgaard, D. Darais, D. Clarke, and F. Piessens. 2013. Monadic Abstract Interpreters. In *PLDI*.
[58] P. Sewell, F. Zappa Nardelli, S. Owens, G. Peskine, T. Ridge, S. Sarkar, and R. Strnisa. 2010. Ott: Effective Tool Support for the Working Semanticist. *J. Funct. Program.* 20, 1 (2010), 71–122.
[59] A. Stefanescu, D. Park, S. Yuwen, Y. Li, and G. Rosu. 2016. Semantics-based Program Verifiers for All Languages. In *OOPSLA*.
[60] The Gillian Team. 2020. Gillian on GitHub. https://github.com/GillianPlatform/Gillian.
[61] The Gillian Team. 2020. The Official Gillian Website. https://gillianplatform.github.io.
[62] E. Torlak and R. Bodík. 2013. Growing Solver-aided Languages with Rosette. In *Onward!*
[63] E. Torlak and R. Bodík. 2014. A Lightweight Symbolic Virtual Machine for Solver-Aided Host Languages. In *PLDI*.
[64] E. Wittern, A. T. T. Ying, Y. Zheng, J. Dolby, and J. Alain Laredo. 2017. Statically checking web API requests in JavaScript. In *ICSE*.