Imperial College London

MEng Individual Project

Imperial College London
Department of Computing

Continuous Reasoning for Real-World Program Analysis Tools

Author:
Alexis Marinoiu

Supervisor:
Prof. Philippa Gardner

Second marker:
Dr. Mark Wheelhouse

June 15, 2020

Abstract

Having high-quality and reliable software is an increasingly stringent requirement for most organisations. While traditional static program analysis techniques, such as symbolic execution and formal verification, can help produce error-free software, their successful application to large, rapidly-changing codebases has been limited. In response to this, recent years have seen the development of a number of tools that specifically target their analysis at real-world projects. At Facebook, the Infer tool has been successfully deployed to find lightweight bugs in codebases spanning millions of lines of code, by leveraging various continuous reasoning techniques.

This project extends Gillian, a multi-language platform for symbolic analysis developed in the Verified Software group at Imperial, with continuous reasoning foundations that substantially advance its applicability to real-world projects. In particular, we have: extended its instantiations for C and JavaScript in order to allow them to analyse multi-file projects; and incorporated a mechanism for re-using previously-stored results in order to focus analysis on only the fragments of the source program that have changed. Finally, by analysing two real-world JavaScript and C projects, we have demonstrated the significant improvement in the usability of Gillian by a general developer, as well as the correctness of our implementation.

Acknowledgements

I would like thank my supervisor, Professor Philippa Gardner, for her enthusiastic support and encouragement throughout the project.

I would also like to extend a huge thanks to Sacha Ayoun and Petar Maksimović for their patience and near 24/7 availability in responding to my numerous questions.

Last, but not least, I would like thank my family and friends for their unfaltering support throughout my entire degree.

Contents

Chapter 1

Introduction

As the size and complexity of modern code continues to increase, so too does the challenge of developing adequate program analysis tools in order to reason about its safety, robustness and functional correctness. Bugs and vulnerabilities in programs can result in major economic and reputational cost to companies, while failures in safety-critical systems have resulted in fatalities in the past. Developers mitigate against these predominantly through writing tests and conducting code reviews, however these strategies require significant manual effort and are prone to human error. As such, the development of static analysis tools has received growing attention from both academia and industry in the last few decades.

Tools such as ASTRÉE [16] and SLAM [1] have been successfully used to verify programs spanning tens of thousands of lines of code, with the former being used to prove the absence of runtime errors in C programs and the latter being used to verify procedure calls in Windows device drivers. However, in general, applying formal verification techniques in industrial environments poses significant challenges. First, there is the question of scale: modern codebases typically span millions of lines of code across a large variety of programming languages, making the cost of traditional whole-program analysis prohibitive. Second, there is the challenge of integrating such analysis into the modern development workflow: large Internet companies such as Facebook and Google embrace the practice of *perpetual development* [19], where the software is not viewed as a finished product, and instead features are continuously added and shipped to users by hundreds of engineers working concurrently. For analysis results to be meaningful in such contexts, they need to be provided automatically and in a timely manner. Finally, there is the challenge of developer adoption: traditional verification tools require input from the user in the form of specifications, a process that is time-consuming and non-trivial to complete, or provide feedback that is difficult to understand without an extensive background in formal methods [38].

Recent years have seen the development of a number of static analysis tools aimed at addressing these challenges. The most prominent of these is Infer [11], developed by Peter O'Hearn and his team at Facebook, which has been able to find lightweight bugs such as null pointer dereferences and memory leaks in the company's mobile applications, and can scale to millions of lines of code. At Imperial, the Verified Software group, led by Philippa Gardner, has been developing Gillian

[22], a platform for building analysis tools that supports several different types of analysis, including whole-program symbolic execution, verification based on user-provided specifications, and automatic compositional testing of bare programs.

By translating the source code into their own symbolic intermediate languages, Infer and Gillian can both be used to analyse a variety of high-level programming languages, with the former restricted to static languages such as Java, C and Objective-C, and the latter being able to be *instantiated* to support any target language. In addition, both tools make extensive use of *separation logic*, a recent development in program logics that can reason about the heap in a composable and therefore scalable way. It is this formalism which paves the way for *continuous reasoning* [37], whereby an analysis tool is able to reason about a large, continually changing codebase and therefore be integrated into the perpetual development model outlined above. Infer achieves this by being run automatically as part of Facebook's continuous integration (CI) system upon each new code modification (i.e. *diff*) submitted by a developer, using previous analysis results in order to detect potential bugs introduced by the change and provide prompt feedback to the developer [37].

This project extends the implementation of Gillian with a similar continuous reasoning mechanism that paves the way for it to be integrated into modern development workflows and therefore reach wider adoption. The previous version of Gillian could only analyse single-file projects and did not keep track of any information obtained from previous analyses, thereby limiting its applicability to most real-world projects. In particular:

- We add support within Gillian's C instantiation for analysing multi-file C projects (Chapter 3). With reference to standard compiler toolchains such as GCC, we develop mechanisms for performing the equivalent of symbol resolution, linking, and loading, at the level of Gillian's intermediate language.
- We add support within Gillian's JavaScript instantiation for analysing projects using CommonJS modules (Chapter 4). To do this, we build a mechanism to emulate the runtime behaviour of Node's module loader.
- For each of Gillian's three analysis modes, we develop an *incremental* variant that can leverage previously-stored results in order to focus analysis on only the fragment of code that has changed between successive runs of Gillian (Chapter 5).
- By applying Gillian's analysis to two real-world projects, we demonstrate the vast improvement in its usability by a general developer as a result of our work (Chapter 6). In particular, we demonstrate the effectiveness of our import and export mechanics, and show that, with additional caching, we can produce analysis times that are within acceptable margins of those reported in the published Gillian paper.

Chapter 2

Background

2.1 Analysing software systems

Software systems are abound with defects. The impacts of these range from the mildly inconvenient, as in the case of Microsoft's Zune music players freezing on 31 December 2008 due to an incorrect handling of leap years [2], to the catastrophic, as in the case of the Therac-25 radiation therapy machines delivering fatal doses to six patients in the 1980s in part due to concurrency bugs and arithmetic overflows [34]. Manual testing, in the form of developers writing unit, integration and system tests, can go a long way in exposing such bugs and ensuring that they are not re-introduced. However, no amount of testing can guarantee that a system is *free* from defects.[1] In addition, writing tests that achieve a high coverage of the code is a time-consuming process, and developers may not always be able to spot the intricate, unusual input combinations that can result in failure. For this reason, more systematic methods of analysis have taken a foothold over the past few years. Such methods are evaluated with respect to two key metrics [8]. The first is *soundness*, which is the degree to which the analysis does not generate false negatives—instances of bugs being reported to not exist when in fact they do. The second is *precision*, which is the degree to which the analysis does not generate false positives—instances of bugs being reported to exist when in fact they cannot occur. The methods can be broadly categorised into two approaches.

Dynamic analysis [18, 8] involves executing a program, either directly or through emulation, with a range of test inputs and collecting information about its executions. It encompasses techniques such as fuzzing, in which many randomly-generated inputs are used in an attempt to exercise the program in interesting ways (for example, to make it crash), and the use of compiler sanitizers, in which the program is instrumented with checks that can expose safety violations, such as out-of-bounds memory accesses, at run time. These benefit from being precise, since they will never model a run of the program that cannot actually occur, and from being able to scale to programs with many lines of code, since only a single execution path is explored on each run. However, they can miss runs that lead to errors and are therefore not sound. In addition, dynamic analysis typically requires access to the entire system as well as an execution environment, making it hard to test small parts

[1] As Edsger Dijkstra famously put it [17]: "Testing shows the presence, not the absence of bugs."

of the code in isolation or during development.

In contrast, *static analysis* [18] does not require a program to be executed, and instead uses its syntactic structure in order to derive properties that hold across all control flow paths and therefore across all (or a large number of) possible executions. It allows for the discovery of logical errors, such as uninitialised variables, or bad programming practices, such as unused variables. In addition, the information obtained from a static analysis can be used as the basis for formal verification methods. Static analyses typically range between those that are precise, but extremely expensive computationally, and those that are fast, but extremely imprecise [8].

2.2 Static program analysis

In this section, we discuss some of the static analysis techniques that underpin current state-of-the-art program analysis tools such as Gillian and Infer, including symbolic execution (§ 2.2.1), verification based on separation logic (§ 2.2.3), and bi-abduction (§ 2.2.4).

2.2.1 Symbolic execution

Symbolic execution [32, 9] is a popular program analysis technique developed in the 1970s. During concrete (i.e. regular) execution, a program is run with a set of specific inputs, which means that only a single control flow path is followed. In contrast, during symbolic execution, a symbolic interpreter runs the program with a set of *symbolic* inputs that represent arbitrary values. As a result, program variables and outputs are expressed as functions of the symbolic inputs.

Symbolic execution allows for the different control flow paths that the program would take under different inputs to be explored at the same time. To do this, the interpreter maintains as state a symbolic *variable store*, which maps program variables to symbolic expressions, as well as a symbolic *path condition* π, which is a first-order logic formula representing the accumulated constraints that the symbolic inputs must satisfy in order for an execution to follow the current path. The path condition is initialised to *true* (i.e. $\top$). Upon reaching a conditional statement such as `if` (e) `then` $S1$ `else` $S2$, the execution branches, with one new execution path for each of the conditional branches. In the path that follows $S1$, the path condition is updated to $\pi' = \pi \wedge e$, while in the path that follows $S2$, it is updated to $\pi' = \pi \wedge \neg e$. A constraint solver, such as an SMT solver, is used to check the satisfiability of the updated path condition. If it is unsatisfiable, then the execution along that path terminates. Otherwise, the execution continues until the path terminates normally or with an error. At the end, each accumulated path condition can be solved using the constraint solver in order to obtain specific input values that would lead a concrete execution starting with those inputs to follow the same path [9]. Symbolic execution can therefore be used to automatically generate test cases that achieve a high statement and branch coverage of the code.

As an example, we consider the symbolic execution of the C function `fizz_buzz()` given in Figure 2.1, a simple variation of the solution to the popular coding inter-

```
1 void fizz_buzz(int n) {
2     if (n % 3 == 0) {
3         printf("Fizz\n");
4     } else if (n % 5 == 0) {
5         printf("Buzz\n");
6     } else {
7         printf("%i\n", n);
8     }
9 }
```

FIGURE 2.1: `fizz_buzz()` C example

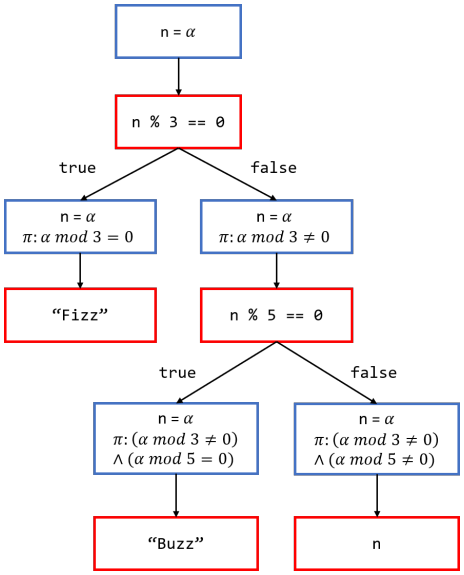

FIGURE 2.2: Symbolic execution graph for `fizz_buzz()`

view challenge of the same name.[2] At the start, since the value of integer `n` is unknown, it is assigned a symbolic value α. Upon reaching the first conditional statement on line 2, the execution forks, with the execution following the `if` branch being assigned the path condition $\alpha \bmod 3 = 0$ and the execution following the `else` branch being assigned $\alpha \bmod 3 \neq 0$. The first execution terminates upon reaching the print statement on line 3. The second execution will once again fork upon reaching the conditional statement on line 4, with the path conditions being updated to $(\alpha \bmod 3 \neq 0) \wedge (\alpha \bmod 5 = 0)$ and $(\alpha \bmod 3 \neq 0) \wedge (\alpha \bmod 5 \neq 0)$ for the paths following the `if` and `else` branches, respectively. The final symbolic execution tree is shown in Figure 2.2. We note that π is distinct in each terminal leaf node, and that choosing the values 1, 3, and 5 for `n` as concrete test inputs would result in all paths being taken and therefore a complete statement coverage of the code. In practice, the number of paths to be explored grows exponentially with the size of the code and, given the presence of unbounded loop iterations, may even be infinite. This is known as the *path explosion problem*, and can be addressed by using search heuristics to guide the path exploration process or by performing optimisations such as pruning redundant paths [6].

[2] `https://en.wikipedia.org/wiki/Fizz_buzz`

2.2.2 Hoare logic

In the late 1960s, Tony Hoare introduced Hoare logic [29], a formalism for reasoning about the correctness of simple imperative programs that alter the variable store. At its core is the notion of a *Hoare triple*, represented as $\{P\}\ C\ \{Q\}$,[3] which describes the effect of a command C in terms of first-order logic assertions P and Q; P is referred to as C's *pre-condition* and Q as its *post-condition*. The triple is read informally as follows: if C is executed in a state satisfying P, then this execution will not fault and, if it terminates, it terminates in a state that satisfies Q. Such specifications can only be used to prove *partial* correctness of programs—a *total* correctness specification, represented as $[P]\ C\ [Q]$, additionally requires that C must terminate [27]. Hoare logic additionally defines a set of axioms and inference rules [29] which can be used to build up proofs of programs. For example, there is the *assignment axiom*:

$$\frac{}{\vdash \{P[E/\mathtt{x}]\}\ \mathtt{x} := E\ \{P\}}$$

where $P[E/\mathtt{x}]$ denotes assertion P with all free occurrences of program variable $\mathtt{x}$ replaced by expression E. A valid Hoare triple that uses the assignment axiom is:

$$\frac{}{\vdash \{\mathtt{y} = 3\}\ \mathtt{x} := \mathtt{y}\ \{\mathtt{x} = 3\}}.$$

While the assertion language of Hoare logic is useful for describing how a variable store is changed by a program, it cannot be used to efficiently reason about mutations to dynamic memory. This is because assertions are *global*: they describe the whole heap. In the case of a command that accesses a single cell of memory, the assertions used to describe its behaviour would have to include parts of the heap that the command does not affect [26]. This is similar to the frame problem that arises in artificial intelligence. Therefore, this type of reasoning does not scale for programs written in real-world languages: for those, a formalism that supports *local reasoning* [35] is required, where a specification and proof can concentrate only on the cells in memory that are actually accessed by the program.

2.2.3 Separation logic

Building upon Hoare's work, in 2001, O'Hearn, Reynolds and Yang introduced separation logic [35], a formalism that extends Hoare logic in order to efficiently reason about programs that alter the heap. It allows for reasoning to be broken down into smaller parts that correspond to local operations on memory (such as allocation, deallocation, mutation, and lookup), and provides the necessary mechanism for these reasoning parts to be composed together. Assertions in separation logic are built from the standard connectives and quantifiers of first-order logic as well as the separating conjunction $*$, which is read as "and separately", and the empty assertion emp, which denotes the empty heap. In addition, they include the cell assertion $E_1 \mapsto E_2$, read as "E_1 points to E_2", that describes a cell with address given by E_1 and content given by E_2.

[3] In [29], Hoare used the notation $P\ \{C\}\ Q$; we use the form that is now more common.

In contrast to Hoare logic assertions, separation logic assertions are interpreted over small sections of the whole program heap known as *heaplets*. A heaplet is described by $P * Q$ when it can be separated into two disjoint heaplets, such that one satisfies P and the other satisfies Q. For example, the assertion $\mathtt{x} \mapsto 1 * \mathtt{y} \mapsto 2$ describes the heaplet consisting of exactly two cells, the first allocated at the address given by variable `x` and containing the value 1, and the second allocated at the address given by variable `y` and containing the value 2. We know that there are precisely two cells because $*$ stipulates that `x` and `y` must not have the same value.

Proofs in separation logic are constructed from specifications known as *local Hoare triples* [26]. These are also denoted as $\{P\}\ C\ \{Q\}$, but unlike in traditional Hoare logic, P is a separation logic assertion which describes the sufficient resource needed for command C to run (i.e. its *footprint*). Such local specifications can then be extended to include portions of the heap that are unaltered by the execution of command C through the *frame rule*:

$$\frac{\vdash \{P\}\ C\ \{Q\} \quad mod(C) \cap fv(R) = \emptyset}{\vdash \{P * R\}\ C\ \{Q * R\}}$$

where the side condition $mod(C) \cap fv(R) = \emptyset$ specifies that no variable modified by C occurs free in R. This is the key mechanism in separation logic which allows for local reasoning. For example,[4] suppose we represent a switch by a cell with a value of either 1 or 0, corresponding to the switch being on or off, respectively. In addition, suppose that we have a function `close_switch()` which takes as input a switch that is in the 'on' state and sets it to the 'off' state. A simple specification for `close_switch()` might look like:

$$\vdash \{\mathtt{s} \mapsto 1\}\ \mathtt{close\_switch(s)}\ \{\mathtt{s} \mapsto 0\}.$$

Now assume that we have two switches, `s1` and `s2`, and we close the first one of them. As the two cells are disjoint, we know that `s2` will remain unaltered, so we can apply the frame rule to obtain the following program specification:

$$\vdash \{\mathtt{s1} \mapsto 1 * \mathtt{s2} \mapsto 1\}\ \mathtt{close\_switch(s1)}\ \{\mathtt{s1} \mapsto 0 * \mathtt{s2} \mapsto 1\}.$$

2.2.4 Bi-abduction

In his work [28], the philosopher Charles Peirce defined abductive inference, alongside deductive and inductive inference, as one of the three main types of valid knowledge inference, with human thinking as a result of a combination of them. Given an assumption A and a goal G, abduction enables us to find the missing assumption M that in conjunction with A explains G. Formally, we require the missing M such that the entailment $A \wedge M \vdash G$ becomes true. In 2009, Calcagno et al. [13] incorporated this principle into a general technique for synthesising separation logic specifications of programs, which they named *bi-abduction*. It involves finding the missing parts $?M$ and $?F$ in the entailment

$$A * ?M \vdash G * ?F$$

[4] Adapted from the examples given in the Infer documentation [41].

where $?M$ is referred to as the *anti-frame* and $?F$ as the *frame*. If we consider G to be the (known) pre-condition of a procedure, then bi-abduction consists of two inference steps. The first is to abduce the anti-frame, which is akin to determining the minimum missing state needed in order to entail G and therefore be able to perform the procedure call. Then, determining the frame is akin to finding the leftover piece of state that was not 'consumed' by the procedure call. To illustrate how bi-abduction works in practice, we build upon the example from § 2.2.3. Suppose that we now wish to infer the pre- and post-conditions of a function `close_switches()` that has the following body:

$$\texttt{close\_switch(s1)}; \texttt{close\_switch(s2)}$$

In addition, assume that the specification of `close_switch()` as given in § 2.2.3 is known at the start; it could have been computed in a previous analysis or inputted manually by the user. We start by executing the body of the function symbolically, with our symbolic heap initialised to emp. When the reach the first command, we need to solve the bi-abductive task

$$\mathsf{emp} * ?M \vdash \texttt{s1} \mapsto 1 * ?F.$$

Clearly, the simplest solution is to take $?M = \texttt{s1} \mapsto 1$ and $?F = \mathsf{emp}$. We therefore infer that our starting state should have been $\texttt{s1} \mapsto 1$ rather than emp, and record it as a missing pre-condition. We can then proceed with with the function call, with the result that the pre-condition of `close_switch()` gets replaced by its post-condition in the symbolic heap:

$$\left\{\texttt{s1} \mapsto 1\right\} \texttt{close\_switch(s1)}; \left\{\texttt{s1} \mapsto 0\right\} \texttt{close\_switch(s2)}.$$

When we process the second command, we arrive at the bi-abductive task

$$\texttt{s1} \mapsto 0 * ?M \vdash \texttt{s2} \mapsto 1 * ?F$$

which we can solve by taking $?M = \texttt{s2} \mapsto 1$ and $?F = \texttt{s1} \mapsto 0$. We therefore record $\texttt{s2} \mapsto 1$ as a missing assumption and proceed with the function call in a similar way to before:

$$\left\{\texttt{s1} \mapsto 1\right\} \texttt{close\_switch(s1)}; \left\{\texttt{s1} \mapsto 0 * \texttt{s2} \mapsto 1\right\} \texttt{close\_switch(s2)}$$

$$\left\{\texttt{s1} \mapsto 0 * \texttt{s2} \mapsto 0\right\}.$$

In addition, we know that the cell $\texttt{s2} \mapsto 1$ is not touched by the command `close_switch(s1)`. We can therefore apply the frame rule and thread this assertion back to the start of the function body, making it part of the function's pre-condition. This leads us to obtain the following full specification for `close_switches()`:

$$\vdash \left\{\texttt{s1} \mapsto 1 * \texttt{s2} \mapsto 1\right\} \texttt{close\_switch(s1)}; \texttt{close\_switch(s2)} \left\{\texttt{s1} \mapsto 0 * \texttt{s2} \mapsto 0\right\}.$$

This example illustrates the main benefit of bi-abduction: we are able to analyse and synthesise the specification of a small part of a program (in this case, an

individual procedure) independently of its calling context. It therefore allows for the analysis of incomplete programs which would have otherwise needed extra scaffolding code, such as `main()` method that calls the program's procedures [13]. In addition, bi-abduction paves the way for static analysis tools that embrace the principle of *compositional analysis* outlined by O'Hearn in [37], where the analysis result of a program fragment is defined as a combination of the analysis results of its parts. These have a greater potential to scale to programs with millions of lines of code, as analysis results, once computed and persisted on disk, do not need to be recomputed for procedures that have not changed [13].

2.3 Related tools

The development of separation logic (SL) and bi-abduction has paved the way for a number of static analysis tools that can efficiently reason about the heap, with analysis ranging from full verification aimed at the specialist verifier to lightweight bug-finding aimed at the general developer. These tools can be broadly categorised into those that are *automatic*, which take as input bare programs and automatically synthesise specifications, and those that are *semi-automatic*, which require the user to annotate their programs with SL specifications, loop invariants, and proof tactics. We begin this section by giving a timeline of this development (§ 2.3.1), which culminates with the recent work on building *multi-language tools* that are able to target different high-level languages. We then explore several of these in more depth, including Infer (§ 2.3.2) and Gillian (§ 2.3.3).

2.3.1 Timeline

2005 Berdine et al. [3] develop a mechanism for proving separation-logic Hoare triples for basic loop-free programs using symbolic execution. This forms the basis for Smallfoot [4], the first semi-automatic verification tool based on SL, which can verify programs of a small while language with user-provided pre-/post-conditions and loop invariants. The assertion language is limited to built-in data structures such as lists and trees.

2008 Calcagno et al. develop SpaceInvader [12, 48], a tool that is able to verify the absence of null pointer dereferences and memory leaks in Linux and Windows device drivers (up to 10,000 lines of C code). Unlike Smallfoot, it is also able to infer some loop invariants.

2009 Calcagno et al. introduce Abductor [13], an extension of SpaceInvader that works on bare C programs. It uses bi-abduction to automatically infer pre-/post-conditions and loop invariants, and is therefore the first tool to provide compositional analysis, enabling it to scale to millions of lines of code. It leads to the creation of the startup Monoidics, which is later acquired by Facebook [15].

2010 At KU Leuven, Jacobs et al. develop VeriFast [31, 30], a semi-automatic verifier for Java and C that supports user-defined data structures for both single- and multi-threaded programs. Users provide their own SL specifications, invariants, and optional explicit proof tactics that dictate when to fold and unfold predicates.

2011 Adopting the principles of Abductor, the Monoidics team develop Infer. Like Abductor, the early version [10] of Infer is able to automatically verify memory safety properties of C code and produce a list of lightweight bugs. At Facebook, Infer is extended [11] to be able to analyse multiple high-level languages and is integrated into the company's code review system [37].

2018 At Imperial, Fragoso Santos et al. develop the JavaScript Verification Toolchain (JaVerT) [23], the first separation-logic-based tool for a dynamic programming language. It is able to support the full language semantics of ES5 strict (a restricted variant of JavaScript with semantics that are less error-prone) by using its own intermediate representation, the JavaScript Intermediate Language (JSIL).

2019 Fragoso Santos et al. introduce JaVerT 2.0 [24], which unifies the treatment of whole-program symbolic execution, verification based on SL, and automatic compositional testing. This leads to the current work on Gillian [22], a multi-language platform for building symbolic analysis tools.

2.3.2 Infer

Infer [10, 42] is a fully-automatic static analysis tool originally developed by the verification startup Monoidics and later, following its acquisition, by Facebook. It is written in OCaml and comprises Infer.SL, the original separation-logic-based bug-finding tool, as well as Infer.AI, a more general analysis framework based on abstract interpretation that has been instantiated for other analysis domains such as security and concurrency. While the original version of Infer aimed at proving the memory safety of C programs, it has since evolved into becoming a *multi-language* tool, being able to target a number of high-level static languages including Java, C, C++ and Objective-C. It performs its analysis in two phases:

1. The **capture phase**, where the input files are parsed and translated into Infer's *intermediate symbolic language*. This output is stored in the results directory, which by default is `infer-out` in the project root.

2. The **analysis phase**, performed by symbolically executing the intermediate files in `infer-out`. All information, errors, and warnings reported by Infer are stored within a text file (more precisely, a CSV file), and the errors which are considered most likely to be real are propagated to the user.

Figure 2.3 show an example Java program that dereferences a null string, and Figure 2.4 shows the summary produced by Infer after analysing the program. We

```
1 class Example {
2     int test() {
3         String s = null;
4         return s.length();
5     }
6 }
```

FIGURE 2.3: `Example.java`

```
Example.java:4: error: NULL_DEREFERENCE
  object s last assigned on line 3 could be null and is
  dereferenced at line 4.
  2.        int test() {
  3.            String s = null;
  4. >          return s.length();
  5.        }
  6.    }
```

FIGURE 2.4: Analysis results for `Example.java`

can see that Infer was able to analyse the method `test()` even if it was not part of a complete program (i.e. one with `main()` method). This is because Infer performs the analysis in a *bottom-up* fashion: it uses bi-abduction in order to derive specifications (as Hoare triples) of independent procedures at the bottom of the call graph, and then composes these triples together in order to derive specifications for their callers. Because the analysis is sound with respect to the underlying model of separation logic, it can conclude that a procedure is memory-safe if it succeeds in computing a Hoare triple for it. Otherwise, if it fails, it extracts from the failed proof the possible reasons why it could not establish memory safety [10]. Furthermore, Infer is *incremental*: it keeps track of the results obtained from analysing procedures, enabling successive invocations of it to only re-analyse procedures that have changed.

Because of its composable and incremental analysis, Infer can successfully scale to codebases spanning millions of lines of code. This has allowed it to have considerable impact at Facebook, where it is currently deployed to continuously run on every code modification made to the company's Android and iOS applications [42].

2.3.3 Gillian

Generalising the work of JaVerT 2.0, over the past few years the Verified Software group at Imperial has been developing Gillian [22, 21], a framework for building symbolic analysis tools for real-world programming languages. Gillian is written in OCaml and supports three types of analysis:

- *whole program symbolic testing*, where the user writes unit tests with symbolic inputs and outputs, and uses simple first-order assertions to describe the properties that the outputs must satisfy, while Gillian tries to generate symbolic traces that invalidate those assertions;

- *verification*, where functions in the program are annotated by the user with separation-logic pre- and post-conditions, loop invariants, and proof tactics, and Gillian verifies that the functions meet their specifications;
- *automatic compositional testing*, where, in the style of Infer, the user provides a bare program with no annotations, and Gillian uses bi-abduction to automatically generate specifications for the functions in the program up to a pre-established bound.

As with other multi-language tools such as Infer, Gillian is underpinned by a symbolic intermediate language that is used as a compilation target for a variety of high-level languages. However, such tools typically require the user to encode the memory model of the target language into the memory model of the tool's intermediate representation, which limits the number of languages they can feasibly support. In contrast, Gillian uses a symbolic intermediate language, GIL, that is *parametric* on the memory model of the target language. The fundamental ways in which programs of a particular target language interact with their concrete and symbolic memories are captured by a set of *actions*, and these are implemented by the developer during the instantiation of Gillian for that language. During compilation to GIL, these interactions are then translated into invocations of their corresponding actions. For example, in JavaScript, the mechanism specified in the standard for looking up the value of an object's property will at some point include a call to a specific action that actually inspects the heap.

Gillian employs a modular architecture, with each reasoning component built on top of another. This is shown in Figure 2.5. All components use the same underlying semantics of GIL, which is a simple goto language with top-level procedures. At the core of Gillian is a symbolic execution engine, with a state model that comprises the concrete and symbolic memory models provided by the user and Gillian's built-in reasoning about the variable store. On top of this is the verification component, which extends the state model with a set of user-defined *core predicates*. These are separation-logic assertions that describe the atomic building blocks of the target language's memory, and are each implemented by a pair of actions, consisting of: (i) a getter action, which removes the predicate's footprint from the state, akin to a 'frame off' operation in SL; and (ii) a setter action, which extends the state with the predicate's footprint, akin to a 'frame on' operation. Finally, the bi-abduction component extends the state with an additional set of user-defined *fixes*, which, whenever an action execution fails due to a incomplete state, describe the mechanism for inferring the missing resource [21]. A compiler from the target language to GIL is then the final user-provided component needed to complete the instantiation of Gillian for that particular language.

Gillian has currently been instantiated to provide analysis tools for JavaScript, in the form of Gillian-JS; C, in the form of Gillian-C; and WISL, a toy imperative language with basic control flow constructs and pointers, in the form of Gillian-WISL. Gillian-JS incorporates the JS-2-JSIL compiler from JaVerT and the JavaScript memory model of JaVerT 2.0, along with a straightforward compilation step from JSIL to GIL [22]. The current work done by the Verified Software group mainly revolves around extending Gillian to support reasoning about more complex language features such as events and concurrency.

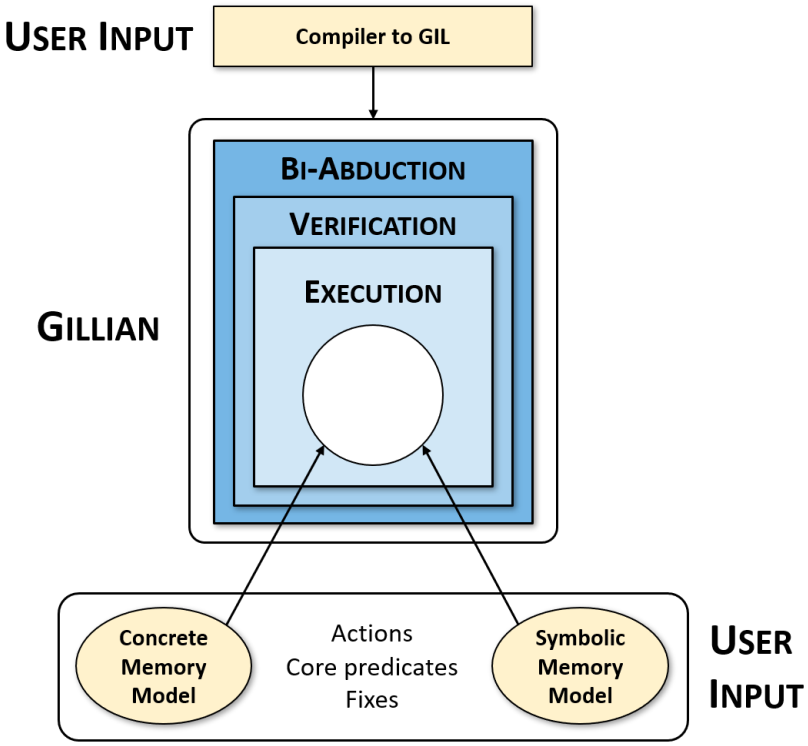

FIGURE 2.5: Gillian architecture [25]

2.4 Continuous reasoning

Along with the research and engineering effort required to ensure that the analyses produced are sound and precise, comes the more general challenge of developing verification tools that can reach widespread adoption. This may not always be the desired end goal for such tools, in particular when they are merely academic endeavours, such as the early separation-logic-based tools we have seen in § 2.3, or when they are principally designed for use by experts in formal methods, such as tools used to verify safety-critical systems. However, as O'Hearn argues in [37], shifting research focus towards creating tools that can be easily be integrated into modern development environments can yield significant benefits, as it would allow formal reasoning to reach many more programs and many more programmers. This involves tackling several challenges: (i) developing tools that are able to scale to millions of lines of code; (ii) ensuring that these can easily be integrated into existing development workflows; and (iii) ensuring that they provide feedback that is of value to developers, ideally in a timely manner.

In [37], O'Hearn suggests that tools can address these challenges by adopting the principle of *continuous reasoning*, where formal reasoning is performed in a way that mirrors the iterative, continuous model of software development widely practised in industry today. Technology companies have moved away from the *waterfall* method, where development progresses successively from requirements to planning, development and testing, to more *agile* methodologies, where new features are continually added and made available to end users, as this allows for the product to start generating revenue earlier and for development to be shaped by user feedback. This is enabled by the practice of *continuous integration* (CI), which ensures that

there is always a stable, working version of the code as it evolves. After developers implement changes on their local working copies of the codebase, they use a version control system to submit their changes to a central repository that is often backed by a *CI system*, which, upon each new change, automatically runs a build of the codebase along with a series of tests. Any changes that cause the build to break or tests to fail are promptly flagged and cause an alert to be sent to the developer that introduced them. As tens, if not hundreds, of engineers are likely to be submitting changes to the codebase at the same tine, it is only a matter of time before a local working copy falls behind, requiring the developer to update their copy to the latest version before their change is submitted. For this reason, the practice also encourages developers to integrate their changes frequently, for example, once a day [20].

Chapter 3

Multi-file Analysis of C

C is a ubiquitous statically-typed programming language dating from the 1970s. As with most languages, it allows for applications to be written as a series of smaller modules which can be modified and compiled separately, allowing programs to easily embrace the principle of separation of concerns and removing the need to recompile modules that have not changed. Cross-references between such modules are resolved by a process known as *linking*, which is responsible for combining these individual pieces of code into a single file that can be *loaded* (copied into memory) and executed. This process can be performed either statically, at compile time, or dynamically, at run time. It also removes the need for programmers to continually re-implement pieces of standard functionality (for example, routines for mathematical functions) by allowing them to instead link their programs against specialised, well-tested third-party libraries.

In this chapter, we detail the work done in extending Gillian-C to incorporate a similar mechanism for linking separate C modules that allows it to analyse real-world, multi-file projects. We begin by providing a more detailed but whirlwind exploration of the linking mechanism found within standard compiler toolchains such as GCC (§ 3.1). We then explore the existing architecture of Gillian-C, focusing on the compilation of C to GIL (§ 3.2). Finally, we present our implementation in § 3.3.

3.1 Static linking in C

Static linking has traditionally been the most common mechanism for separate compilation and code-reuse in C. It is also the simplest, and therefore forms a suitable starting point to guide our extension of Gillian-C. We ground our discussion, where relevant, in the context of a system running Linux and using the standard ELF object file format, though we stress that the concepts of linking are universal, regardless of the platform or file format used.

As an example, we consider the compilation of the C program consisting of the files `main.c` and `foo.c`, as shown in Figure 3.1. It has a `main` function calling into `foo`, which is defined in an external `foo.c` file. A user may compile the program by using the *compiler driver* of a given compilation system. For example, if they were using the GNU Compiler Collection (GCC), they may run the following commands

```c
#include <stdio.h>

int foo();

int main() {
    int x = foo();
    printf("%d\n", x);
    return 0;
}
```

(a) `main.c`

```c
int foo() {
    return 10;
}
```

(b) `foo.c`

FIGURE 3.1: A basic multi-file C program

to invoke its `gcc` driver, produce an executable called `prog`, and finally run the program:

```
$ gcc main.c foo.c -o prog
$ ./prog
10
```

Not surprisingly, we can see from the output that the program was able to resolve the references to `foo` and `printf` and branch to the correct function implementation in each case. How was it able to do that?

Behind the scenes, the compiler driver is responsible for invoking the individual components of the compilation system and for linking their inputs and outputs together [7]. The resulting *compilation pipeline* is shown in Figure 3.2. First, the C preprocessor (`cpp`) is invoked on the `main.c` file in order to produce a preprocessed `main.i` file. The preprocessor is responsible for finding and resolving any preprocessor directives (lines beginning with the '`#`' character) that are present. In our example, the `#include <stdio.h>` directive on line 1 would be replaced by the contents of the `stdio.h` file. Other examples include the expansion of macros declared using the `#define` directive and the conditional inclusion of statements nested within `#ifdef...#endif` directives. The preprocessed `main.i` is then compiled using the C compiler (`cc1`) to assembly code, which in turn is assembled into a `main.o` *relocatable object file* by calling the assembler (`as`).

These three steps are then carried out in the exact same way for `foo.c` in order to produce a `foo.o` relocatable object file. We note that the two input files are preprocessed, compiled and assembled independently of each other and of any other source files that may be part of the same program. Compilation occurs at the unit of preprocessed files—hence why these are also termed *translation units*—and they must individually contain sufficient type information to allow the compiler's semantic analysis to succeed. For example, all functions must have their prototype (i.e. type signature declaration) included in the translation unit before any of their usages or definitions. In our example, this is ensured by the inclusion of the `stdio.h` header (which contains a prototype for the `printf` function) and the prototype for `foo` at the beginning of `main.c`.

It is at this point that the linker (`ld`) presides over. Each relocatable object file is a binary file comprising code and data in a form that allows it to be combined

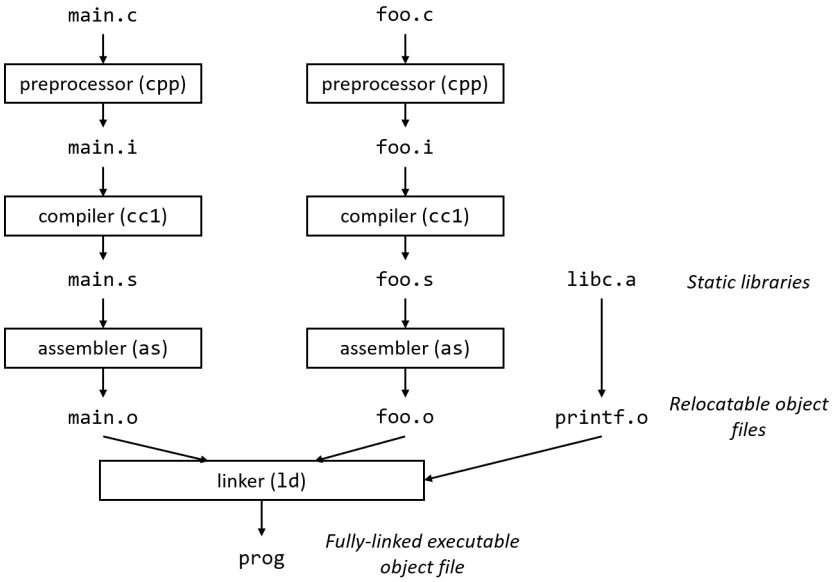

FIGURE 3.2: Linking process in C

with other relocatable object files in order to produce the final *executable object file*. It consists of a number of different sections, including a section for the assembled machine code, a section for initialised global variables, a placeholder section for any uninitialised global variables, as well as an optional table mapping source file lines to machine code instructions (present only if the compiler driver was invoked with the debugging option `-g`). In addition, each file contains a *symbol table*. It is built by the assembler using symbols exported by the compiler, and details all functions and global variables that are defined and referenced within the translation unit [7]. We explore its structure in more depth when we discuss our implementation in § 3.3. For now, it suffices to know that the symbol table for `main.o` would contain an entry for `main` indicating that it is a locally-defined function, and other entries for `foo` and `printf` indicating that they are undefined symbols of unknown type, as we cannot yet determine whether they really are functions (as opposed to, for example, global variables). `foo.o`'s symbol table, on the other hand, would only contain an entry for `foo` indicating that it is a locally-defined function.

The linker then performs a left-to-right scan of the relocatable object files provided as its arguments. The order of these corresponds to the order in which the initial source files were passed to the compiler driver. In addition, it is passed the paths to a number of *static libraries* containing implementations of standard C functions. On Unix systems, each static library is stored in an *archive* (`.a`) file format, which contains a collection of concatenated relocatable object files. For example, the definition of the standard `printf` function comes from a `printf.o` object file that is, in turn, stored inside the `libc.a` archive. During the scan, the linker maintains a set E of relocatable object files that will be merged to form the final executable, a set U of unresolved symbols (i.e. symbols referenced but not yet defined), and a set D of symbols that have been defined in previously seen object files, with all sets initially being empty. Then, for each input file f, it updates the sets according to the following rules [7]:

1. If f is an object file, the linker adds f to E, and updates U and D to reflect the symbol references and definitions in f.

2. Otherwise, if f is an archive file, the linker iterates through all the object files contained within f. If an archive member m defines a symbol that resolves a reference in U, then m is added to E, and U and D are updated to reflect the symbol references and definitions in m. The process is repeated across all member object files until a fixed point is reached where U and D stop changing. Any archive members that are not used are then discarded.

At the end, if the set U is non-empty, the linker throws an error. Otherwise, it merges all the object files in E to build the final `prog` executable. At this stage, the linker is also responsible for writing the final runtime addresses for any labels used within branching machine instructions as, during assembly, these have previously been assigned placeholder addresses. This process is known as *relocation*; we omit its details as it would not be relevant in the context of a symbolic execution tool. However, we refer the interested reader to [7] for more details.

3.2 The Gillian-C compiler

Gillian-C is the instantiation of Gillian for the C language, which, as we have seen in § 2.3.3, means that it provides an implementation of a concrete memory model, a symbolic memory model, as well as a compiler to Gillian's intermediate language for analysis, GIL. As with many other program analysis tools, it uses a compiler frontend in order to save on the otherwise substantial development effort required to implement a correct parser and perform the necessary lexical and semantic analysis. In particular, it incorporates the frontend of CompCert [33], a formally-verified optimising compiler developed at INRIA and intended for use within the compilation of safety- and mission-critical software. CompCert supports most of ISO C99 and has been verified, using machine-assisted formal proofs, to produce executable code that behaves exactly as specified by the semantics of the source C program. It achieves this by performing a series of transformations of the source program that have each been verified (using the Coq proof assistant) to be semantically-preserving, beginning with the initial CompCert C abstract syntax tree (AST) and ending with the AST of the target assembly language. This involves progressing through a total of ten intermediate representations, each exposing an increasingly stack-based (and therefore simplified) view of the program [33].

Gillian-C directly leverages the compiler from C to C#minor, the second of the ten intermediate languages, by extracting OCaml modules from the original CompCert code written in Coq. C#minor was chosen as it is low-level enough to incorporate a number of useful simplifications of the original program but high-level enough to not be too hard to translate to GIL—for example, it still has variables, unlike the lower-level representations. In addition, it only deviates from the C standard by fixing the order of argument evaluation. Gillian-C then provides an additional compiler from C#minor to GIL as well as a compiler for translating, when the tool is run in verification mode, user-defined separation-logic annotations and proof tactics to GIL annotations and logic commands, respectively. The compilation

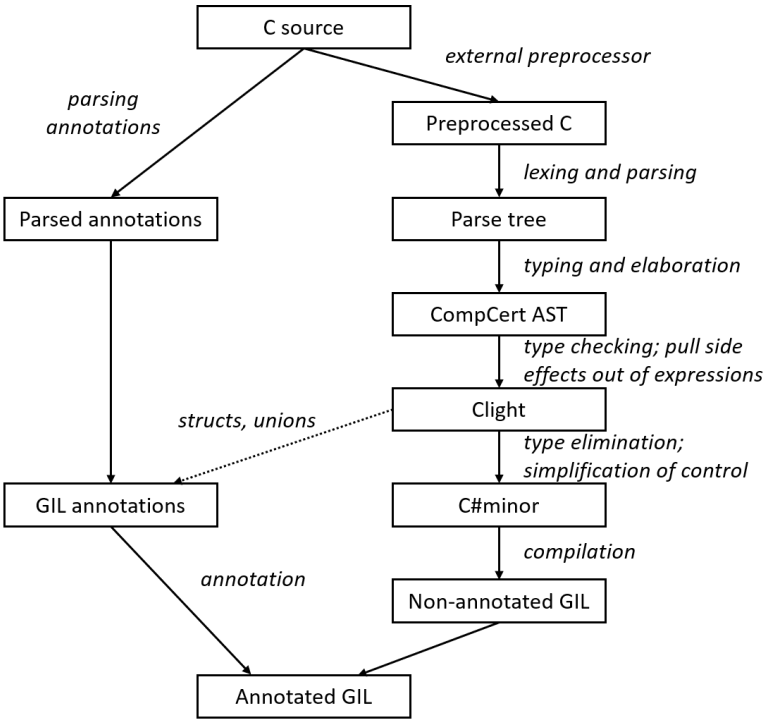

FIGURE 3.3: Compilation process in Gillian-C

process to transform an input C file to GIL then consists of the following steps, as illustrated in Figure 3.3:

1. **Preprocessing of the source C file.** This is done by invoking the system's preprocessor (e.g. `cpp`) to produce a preprocessed C file.

2. **Parsing of the preprocessed file to produce a parse tree.** Any lexical errors that are detected at this stage are propagated back to the user.

3. **Construction of a CompCert C AST.** The AST is elaborated with type information, and any semantic errors that are detected are propagated back to the user. At this stage, a number of simplifications are made. These include cleanups, such as collapsing multiple declarations of the same variable, as well as source-to-source transformations aimed at removing constructs that are not supported by CompCert, such as the pulling of local `static` variables to the global scope (renaming them if needed to keep names unique) and the emulation of bit fields in terms of bit-level operations [33].

4. **Compilation of the CompCert C AST to Clight and C#minor.** In the first stage, side effects are pulled out of expressions (for example, function calls become statements, not expressions). In the second stage, control flow constructs are simplified and types (for example, `struct` and `union` definitions) are eliminated [33].

5. **Compilation from C#minor to GIL.** This forms the bulk of Gillian-C. Control flow constructs are directly mapped to the ones in GIL, and memory interactions are formulated in terms of their respective actions on the C memory model. Verification proof tactics, such predicate FOLD, UNFOLD, and

lemma Apply statements, as well as constructs specific to the tool's symbolic testing mode, such as Assume and Assert statements, are translated into their corresponding GIL logic commands.

6. **Extracting annotations from the source C file.** In parallel, a custom parser is invoked on the original unprocessed file to extract any separation-logic annotations, which are declared using special C-style multi-line comments. These can define predicates, lemmas as well as function specifications. The latter consists of a function identifier as well as the SL pre- and post-conditions of the function (or multiple pairs of these) that are to be verified. For example, an annotated C function for allocating and initialising a new binary search tree node is shown in Figure 3.4.

7. **Translation of the parsed annotations to GIL annotations.** This is a straightforward translation step. In addition, the `struct` definitions that were eliminated in step 4 are extracted directly from the Clight program and converted to predicate definitions that may be used during the verification or bi-abduction proof.

8. **Annotation of the GIL program.** All predicate and lemmas definitions from step 7 are added to the bare GIL program created in step 5. In addition, GIL procedures (which carry their names from the original C functions) are annotated with any specifications that were defined for them. Finally, the resulting GIL program is returned to Gillian for analysis.

```
1  #include <stdlib.h>
2
3  typedef struct bstn {
4      int value;
5      struct bstn *left;
6      struct bstn *right;
7  } BST;
8
9  /*@ spec make_node(v) {
10   requires: (v == #v) * (#v == int(#vv))
11   ensures:  BST(ret, -{ #vv }-)
12 }*/
13 BST *make_node(int v) {
14     BST *new_node = malloc(sizeof(BST));
15     new_node->value = v;
16     new_node->left = NULL;
17     new_node->right = NULL;
18     return new_node;
19 }
```

Figure 3.4: C code annotated with Gillian SL specifications

Steps 1–4 occur inside CompCert, while the remaining steps occur inside Gillian-C. In the full CompCert compiler, the final assembly AST is written to a file as concrete assembly syntax. The system's assembler and linker are then invoked in order to

produce the final relocatable and executable object files, thereby removing the need for CompCert to solve symbol references itself. As Gillian performs its analyses entirely through the symbolic execution of the GIL code (with no object files being produced), it is clear that we need to implement a linking mechanism ourselves. In addition, as this is a mechanism largely pertaining to the analysis of C programs, it is clear that it needs to be incorporated directly within Gillian-C.

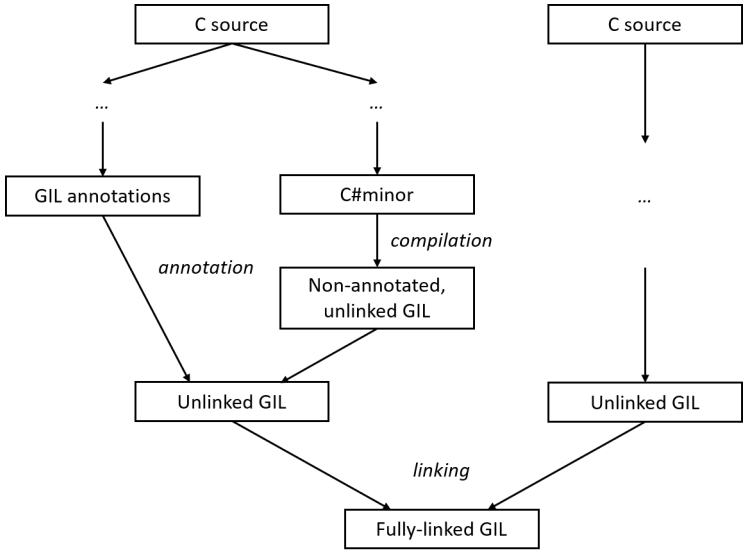

FIGURE 3.5: Design for Gillian-C's linking mechanism

3.3 Design and implementation

Our design for Gillian-C's import mechanism was shaped by observing the parallels between the compilation process of Gillian-C and the one that typically occurs during regular C execution. In the original Gillian-C implementation, a C file gets compiled to exactly one annotated GIL program. The source file must, post preprocessing, contain all the type, function, global variable, and separation-logic-related definitions it requires for its compilation and subsequent symbolic execution. This therefore mirrors the creation of the final executable object file during regular compilation. However, if we add another intermediate step just the final GIL program is produced, creating a valid but non-final GIL program that may not yet have all the definitions it requires, we would have a translation step that mirrors the creation of a relocatable object file. The one-one correspondence now resides between the source C file and its *unlinked* GIL program. We can envision a similar translation happening for other input C files during the same running instance of Gillian-C, resulting in a list of unlinked GIL programs. A linking and combination step, where the unlinked GIL programs have their cross-references resolved, can then be added to produce the final fully-linked GIL program that is returned to Gillian for execution. In order to make this linking step work, we also need to export, along with each unlinked GIL program, information about the symbols the program defines and the symbols that it references but which are undefined. This design is shown in Figure 3.5.

```
module type ParserAndCompiler = sig
  module TargetLangOptions : sig
    type t
    ...
  end

  (** Type for parsing and compilation errors. *)
  type err

  (** Executed at initialisation. *)
  val initialize : ExecMode.t -> unit

  (** Takes a path and returns a GIL AST or an error. *)
  val parse_and_compile_file : string -> (Prog.t, err) result
  ...
end
```

FIGURE 3.6: `ParserAndCompiler` signature

3.3.1 Compiler interface

We begin by discussing the interface for the extended Gillian-C compiler and how it fits into the wider Gillian ecosystem.

Adhering to its goal of being a language-independent platform, Gillian defines standard interfaces in the form of OCaml module types for all the components it is parameterised by. It can then interact with these through the use of *functors*, which are special OCaml modules that can be parameterised by other modules. For example, much of Gillian's top-level code resides in a `CommandLine` functor that is parametric on those component types. It contains functions to handle command-line options that are common to all types of analyses as well as scaffolding code responsible for, among other things, invoking the GIL compiler before any analysis takes place.

Each instantiation's compiler has to comply with the `ParserAndCompiler` module type, the signature for which is shown in Figure 3.6; we omit some of its less important members. The key function is `parse_and_compile_file`. When it is called, it is passed the path of the source file in the target language (as provided by the user on the command line) and is expected to return either an error value (the type of which is itself determined by the instantiation) or the compiled GIL program. It is clear that this will need to be changed. Unlike other languages (such as ES6 JavaScript) which feature explicit `import` or `export` constructs, a C file does not in itself indicate what external files it depends on. Indeed, looking back at the example in Figure 3.1, we can see that `main.c` does not hold any clues to the location of `foo`'s definition. In practice, it is customary for a file to have the prototypes of the functions it wishes to share in a separate header file of the same name. In our example, we would have a `foo.h` file containing the `foo` prototype that is included by both `main.c` and `foo.c`, removing the need for them to declare it themselves. However, this is merely a convention, and the compiler cannot assume that the `foo.h` header really does correspond to a source file named `foo.c`.

We therefore adopt an interface where the user is responsible for the providing the

paths of *all* source files within the C program they wish to analyse. This is similar to what compiler drivers such as `gcc` expect. To make it slightly less cumbersome, we also leverage the fact that each instantiation can define command-line options that are specific to it (through the `TargetLangOptions.t` type in Figure 3.6) to provide the following optional arguments (shown in both their short and long forms), each of which can specified multiple times by the user when invoking Gillian-C:

- `-I` *dir*, `--include` *dir*: if specified, *dir* is added to the list of directories used by the preprocessor to search for included header files. This intended to replicate the option of the same name that is generally supported by compiler drivers such as `gcc`. In our case, we intercept CompCert's call into the system preprocessor and provide the directory paths in the same order in which they are specified. These are not searched recursively.
- `-S` *dir*, `--source` *dir*: if specified, *dir* is added to the list of directories used to recursively search for the program's source files. While it is not by any means a standard option found in compiler drivers, we consider it is a useful addition until Gillian-C is mature enough to be able to be integrated directly within widely-used C build systems (such as Make), where it could automatically learn about the source files in the program by intercepting all commands issued by the build system to the compiler driver.

3.3.2 Defined and undefined symbols

We now delve into the deeper mechanics of the linking step, by first considering all the possible ways in which a given C file could hold references to internally- and externally-defined components.

In § 3.1, we mentioned that information about all the function and global variable identifiers (henceforth referred to as *symbols*) that are defined and referenced within a translation unit are stored in a special data structure known as a symbol table, which is built by the assembler. We will now study the information it holds in more detail. To keep things concrete, we will use the example of `main1.c` (shown in Figure 3.7), which has been constructed to feature the range of ways in which C symbols can be declared and used. We can inspect the symbol table that gets generated for it by telling the `gcc` driver, using the `-c` option, to stop just before invoking the linker and output the relocatable object file instead. We can then use the `readelf` tool (from the GNU Binary Utilities collection) with the `-s` option to extract the symbol table:

```
$ gcc -c main1.c
$ readelf -s main1.o
```

This outputs the following symbol table entries, with the ones not shown corresponding to symbols used by the linker internally:

```
Num:    Value  Size Type    Bind   Vis      Ndx Name
  5:        4     4 OBJECT  LOCAL  DEFAULT    3 y
  6:        0    12 FUNC    LOCAL  DEFAULT    1 bar
  7:        8     4 OBJECT  LOCAL  DEFAULT    3 tmp.2182
```

```
12:        0     4 OBJECT  GLOBAL DEFAULT     3 x
13:        c    65 FUNC    GLOBAL DEFAULT     1 main
14:        0     0 NOTYPE  GLOBAL DEFAULT   UND z
15:        0     0 NOTYPE  GLOBAL DEFAULT   UND foo
16:        0     0 NOTYPE  GLOBAL DEFAULT   UND printf
```

There are three different kinds of symbols within each relocatable object module [7]:

- *Global symbols* (also known as those with *external linkage*) defined by the module and which can be referenced by other modules. These correspond to global[1] variables and functions defined without the `static` attribute. In our example, `main1.c` defines two global symbols, namely the integer `x` and function `main`. We can see that the symbol table marks them as having a '`GLOBAL`' bind, and that the compiler has allocated space for them since their size attribute (in bytes) is non-zero and they have been assigned a section within the object file (the index of which is given by the `Ndx` attribute).

- Global symbols referenced by the module but defined by some other module. These correspond to global variables declared using the `extern` attribute and functions that are declared but not defined. In our example, there are three such symbols, namely the integer `z` and functions `foo` and `printf`. We can see that the symbol table still marks them as having a '`GLOBAL`' bind, but this time the compiler has not allocated any space for them nor has it assigned them to a section in the object file (indicated by the '`UND`', or undefined, value for `Ndx` attribute).

- *Local symbols* (also known as those with *internal linkage*) that are defined and referenced exclusively by the module. These correspond to global variables and functions defined with the `static` attribute. They are visible anywhere within the module, but cannot be referenced by other modules. In our example, there are two such symbols, namely the integer `y` and function `bar`. We can see that the symbol table marks them as having a '`LOCAL`' bind, and that the compiler has allocated space for them, since they have a non-zero size field.

It is worth noting that local function variables marked with the `static` keyword are also included in the symbol table. They are allocated within the same section as global variables, and have their names appended with an integer in order to keep them unique. We can see this from the entry for the `tmp` variable used inside `bar`, which gets the name `tmp.2182`. On the other hand, local variables not marked as `static` (such as the variable `v` used inside `main`) are not included, since they are managed at run time on the stack and are therefore of no interest to the linker.

We now consider the symbol information we need to export during our own linking mechanism within Gillian-C. Fortunately, as we are constructing a symbol table during the translation step from C#minor to GIL, we can leverage the simplifications that have already been applied by CompCert when translating the source C

[1] Here, 'global' refers to variables declared outside of a function body and which are stored in a separate section of the object file. They can still have either internal or external linkage.

```c
#include <stdio.h>

int x = 1;
static int y = 2;
extern int z;

int foo();

static int bar() {
    static tmp = 5;
    return tmp;
}

int main() {
    int v;
    v = z;
    v = foo();
    v = bar();
    printf("%d\n", v);
    return 0;
}
```

FIGURE 3.7: `main1.c`

program to C#minor. We can also leverage the information held within the CompCert AST (a subset of whose definitions are shown in Figure 3.8). The whole program is represented by an OCaml record containing a of list global variable and function definitions, and a list of all identifiers within the program not marked as `static`. Each global variable definition is a record with a field containing a list of initialisation-related data. A non-empty list indicates that the compiler has allocated space for it, whereas an empty list indicates that it is externally-defined (i.e. with `extern`). Similarly, each function definition is a variant with a tag indicating whether it is internally- or externally-defined. Furthermore, as have seen in § 3.2, CompCert elevates all local `static` variable definitions to global variable definitions, performing any necessary mangling to keep the names unique. This means we do not need to handle them in any special way.

There are other simplifications we can also take into account. First, we do not have to record the type of each symbol—whether it is a '`FUNCTION`' or an '`OBJECT`' as above—since this information is not needed in the symbol resolution step itself. Second, we do not need to keep entries for local symbols, since their definitions are not exposed to other modules and are therefore not taken into account during symbol resolution. Third, as we are not constructing an object file, there is no notion of a 'section' in which the symbol will be stored. With this, we can record for each symbol its name and whether a definition exists for it in the current module:

```
type symbol = { name : string; defined : bool }
```

and define a symbol table as a list of `symbol` records.

Finally, we do not actually need to maintain symbol entries for all functions

```
type 'f fundef =
  | Internal of 'f
  | External of external_function

type 'v globvar = {
  gvar_init : init_data list;
  ...
}

type ('f, 'v) globdef =
  | Gfun of 'f
  | Gvar of 'v globvar

type ('f, 'v) program = {
  prog_defs : (ident * ('f, 'v) globdef) list;
  prog_public : ident list;
  prog_main : ident;
}
```

FIGURE 3.8: CompCert AST definition (as OCaml code extracted from Coq)

referenced within the program. In its current form, Gillian-C replaces calls to standard library functions with calls to their corresponding GIL implementations. These are currently limited to the functions needed to enable the symbolic testing of Collections-C, a data structure library used to benchmark Gillian-C in [22]. They include `malloc`, `calloc`, `free`, `memcpy`, `memmove`, `memset`, and `strcmp`.

With this, we arrive at the following possible cases and rules for symbol definitions obtained from CompCert:

1. `Internal` function f. If f is a member of `prog_public`, create a symbol entry for f with `defined` set to true. Otherwise, proceed to the next symbol.
2. `External` function f. If f is an standard library function for which an existing implementation exists, or if it corresponds to a GIL-specific construct such the ASSUME and ASSERT statements used within symbolic testing, then proceed to the next symbol. Otherwise, create a symbol entry for f with `defined` set to false.
3. Global variable v with no initialisation data (i.e. with an empty `gvar_init` field). Create a symbol entry for v with `defined` set to false.
4. Global variable v with some initialisation data. If v is a member of `prog_public`, create a symbol entry for v with `defined` set to true. Otherwise, proceed to the next symbol.

3.3.3 Symbol resolution

We are now in a position to define our full symbol resolution algorithm. We adhere to the following two rules:

```
1: procedure LINK(paths)
2:     U := ∅
3:     D := ∅
4:     for all p ∈ paths do
5:         S := GETSYMBOLS(p)
6:         U_p := {s ∈ S | s.defined = false}
7:         D_p := {s ∈ S | s.defined = true}
8:         if D ∩ D_p ≠ ∅ ∧ ¬ IGNOREMULTDEF then ERROR
9:         end if
10:        D := D ∪ D_p
11:        U := (U ∪ U_p) \ D
12:    end for
13:    if U ≠ ∅ ∧ ¬ IGNOREUNDEF then ERROR
14:    end if
15: end procedure
```

FIGURE 3.9: Symbol resolution algorithm

1. Every symbol referenced within the program must be matched by a corresponding symbol definition.
2. No two definitions for the same symbol are allowed.

If any of these do not hold, then the linking step should raise an error. However, as Gillian-C is still under development, we acknowledge that it would be useful to provide a way to override such errors. This is particularly the case for the first rule, since there are still many standard library functions that do not currently have an implementation in GIL and so would be missing definitions. We therefore also define the following command-line flags:

- `--ignore-undef`: if provided, any errors regarding undefined symbols are ignored.
- `--ignore-multdef`: if provided, any errors regarding symbols with multiple definitions are ignored.

We present the algorithm in Figure 3.9 and walk through it step-by-step below:

1. We begin by initialising the set of unresolved symbols U and the set of defined symbols D to the empty set.
2. For each input source path p, we fetch the symbol table that was constructed during its compilation step from C#minor to the unlinked GIL program. We partition these into two sets: those that are undefined (U_p), and those that are defined (D_p).
3. If any of the newly-defined symbols already have a definition (i.e. are members of both D_p and D), then raise a linker error, unless such errors are being suppressed. Otherwise, update U and D to reflect the symbol references and definitions of p, and proceed to the next path.

```c
#include <stdio.h>

void foo();

int x = 5;

int main() {
    foo();
    printf("%d\n", x);
    return 0;
}
```

(a) `main2.c`

```c
int x;

void foo() {
    x = 10;
}
```

(b) `foo2.c`

FIGURE 3.10: Weak and strong symbols example

4. At the end, if U is non-empty, then raise a linker error, unless such errors are being suppressed.

In practice, linkers usually have some tolerance for multiply-defined global symbols. They additionally categorise symbols into those that are *strong* and those that are *weak* based on information held within the symbol table. Functions and initialised global variables are assigned strong symbols, whereas uninitialised global variables become weak symbols. Multiply-defined symbols are then resolved according to the following rules [7]:

1. Multiple strong symbols are not allowed.
2. Given a strong symbol and multiple weak symbols, choose the strong symbol.
3. Given multiple weak symbols, choose any of the weak symbols.

This can lead to runtime behaviour that may seem baffling to the user. For example, when the program in Figure 3.10 is compiled using `gcc`, the linker assigns a weak symbol to the definition of `x` in `foo2.c` and a strong symbol to the definition in `main2.c`. It then proceeds to discard the weak symbol in favour of the strong symbol. As a result, the author of `main2.c` would discover during execution that the value of `x` gets changed from 5 to 10 by the call to `foo` on line 8.

As these rules are not included as part of the C standard, however, we do not model them within our symbol resolution algorithm.

3.3.4 Combination step

Once we have determined that there are no unresolved symbols, we can begin to piece together the separately-compiled GIL programs.

Gillian already provides support for a basic import mechanism within GIL. Each instantiation can store the GIL implementations of standard functions pertaining

to that target language in a series of GIL *runtime files*. In Gillian-C, these contain the implementations of the standard library functions it currently supports, such as `malloc` and `free`; 32-bit and 64-bit implementations of standard arithmetic and bitwise operators; and definitions of core predicates related to its memory models. Depending on the system's architecture as well the type of analysis for which the program is being compiled, the instantiation's compiler can then specify the runtime files it needs by including a GIL `import` declaration with those file names. When Gillian processes the program, it scans the import declaration (if included), parses any referenced GIL programs, and combines them into a single AST. The combination step is straightforward aggregation of all procedure, predicate, and lemma definitions across the programs. This is because members of a particular component type are considered to be part of the same global namespace and therefore assumed to have unique names.

Therefore, by designating one the unlinked GIL programs created by Gillian-C as the 'main' one, and by adding to it a GIL `import` declaration with references to all the remaining programs, we can delegate much of the work for building the final linked program to Gillian itself. There are two things that Gillian-C needs to handle itself, however.

First, it must ensure that the unique identifier requirement is respected. The symbol resolution step takes care of this for all global symbols. There might, however, be several local symbols with the same name defined in separate modules, which would clash during the merging step. This can be addressed by mangling all local symbols references and definitions during compilation, for example by prepending to each symbol the filename of the source file in which it was defined.

Second, it must correctly initialise the state of the program's memory before this can be executed. In general, this step depends heavily on how the memory model is implemented by the instantiation. In Gillian-C, it consists of initialising the memory with all global variable and function declarations. The compiler does this by building an `i__initialize_genv` GIL procedure that registers all such declarations and by adding a call to it as the first command inside the compiled `main` function. In addition, when the program is being compiled for verification or bi-abduction, it defines a `i__global_env` predicate that is appended to the pre- and post-condition of every procedure. The '`i__`' prefix is used within their names in order to indicate that they are *internal*, in that they do not have a real correspondence to the user-defined source code.

Within our own mechanism, we delay the inclusion of the initialisation-related code until after compilation has taken place. Then, along with its symbol table, each unlinked GIL program can export a list of commands and assertions that it requires for its own initialisation. At the end, we aggregate these into a single `i__initialize_genv` procedure and a single `i__global_env` predicate, and include these within the same 'main' program as above. For simplicity, we consider this to be the GIL program corresponding to the first C source path provided as an argument to Gillian-C.

Finally, the semantics of the `import` declaration within Gillian itself need to be revised slightly. Gillian currently assumes that all imported GIL files are runtime files. This means that during verification and bi-abduction, it excludes procedures

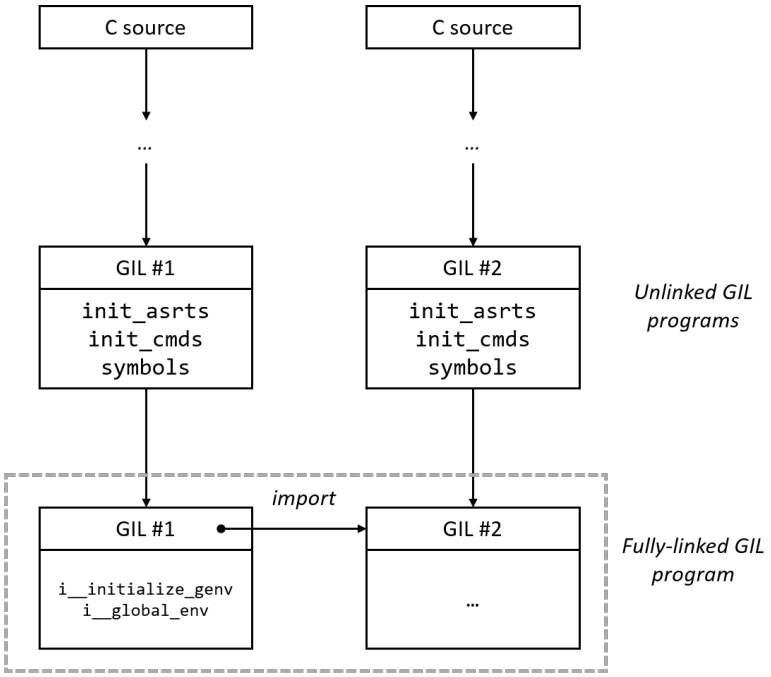

FIGURE 3.11: Creating the final GIL program

defined within those files from the set of procedures it verifies, since runtime-related procedures are always assumed to be 'correct'. While we still want to avoid verifying runtime procedures, we want Gillian to verify all procedures from GIL files corresponding to user source files. We achieve this by extending GIL's syntax to incorporate a new type of import declaration, `import verify`, and changing Gillian's parser to mark any procedures contained within files referenced this way as needing to be verified.

Figure 3.11 shows a summary of the combination step. As an example, we also show the compilation of the two source files given in Figure 3.12, assuming that they are passed to Gillian-C's compiler in the order `main3.c`, `foo3.c`. The compiler produces two corresponding GIL files, `main3.gil` and `foo3.gil`, shown in Figure 3.13. `main3.gil` includes initialisation-related code that incorporates its own global variable and function declarations as well as those of `foo3` (which are highlighted in purple). It also has an `import verify` declaration in order to ensure that `foo3`'s procedure definitions are included in the final GIL program when Gillian resolves its imports.

```c
int x = 5;

int foo(void);

int main() {
    int v;
    v = x;
    v = foo();
    return 0;
}
```

(a) main3.c

```c
int foo() {
    return 10;
}
```

(b) foo3.c

FIGURE 3.12: Example C source files

```
import "unops_common.gil", "binops_common.gil",
       "internals.gil", "logic_common.gil", ...;

import verify "foo3.gil";

pred i__global_env() :
    i__glob_fun("main", "main") *
    i__glob_fun("foo", "foo");

proc i__initialize_genv() {
    u := "i__glob_set_var"("x",
            "x", 4., {{ {{ "int32", 5. }} }},
            "Writable");
    u := "i__glob_set_fun"("main", "main");
     u := "i__glob_set_fun"("foo", "foo");
    ret := undefined;
    return
};

proc main() {
    gvar__0 := "i__initialize_genv"();
    ...
};
```

(a) main3.gil

```
proc foo() {
    ret := {{ "int", 10. }};
    return
};
```

(b) foo3.gil

FIGURE 3.13: Compiled GIL code

Chapter 4

Multi-file Analysis of JavaScript

JavaScript is an object-oriented, dynamic programming language that is used extensively within client-side webpage scripting. It is also used in non-browser environments, such for server-side scripting with the Node.js runtime (also known as Node). Its syntax, core language semantics and built-in libraries are described by the international ECMAScript standard maintained by Ecma. Much of the recent work done by the Verified Software group to build symbolic testing and verification tools for JavaScript (in the form of JaVerT and JaVerT 2.0, as seen in § 2.3) has focused on analysing the fifth edition of the ECMAScript standard (ES5), which was released in 2009. In particular, these tools—and by extension, Gillian-JS—target ES5 *strict*, a variant of ES5 that excludes some of the more error-prone language constructs (such as the `with` statement) and has better-defined semantics.

Unlike subsequent versions of the standard (beginning with the release of ES6 in 2015), ES5 has no built-in constructs to support modules. Over the years, the JavaScript community has come up with a number of different ways to overcome this limitation. The first, and simplest approach is to leverage the existing constructs of the language in order to implement one of several *module design patterns*. Using JavaScript's built-in support for closures, they allow for variables and functions to be grouped into their own namespaces, and for limiting which of these should be 'exported' and available for use elsewhere. The second approach has been to rely on *external module loaders* such as Require.js and the built-in loader within Node, which define an explicit import and export syntax that the program may use. They are then responsible for resolving dependencies between modules, and for loading and executing referenced module code at the appropriate time. The details of this process (for example, whether it happens synchronously or asynchronously) is dictated by the *module system specification* that the loader implements.

In this chapter, we detail the work done in extending Gillian-JS to support the analysis of ES5 programs that rely on such external module systems. In particular, we adopt techniques from the first approach in order to model the runtime behaviour of Node's module loader. We begin by taking a whirlwind tour of some of JavaScript's key concepts, focusing on variable binding and scoping (§ 4.1). We then explore the most common module design patterns (§ 4.2) and discuss Node's module loader in more detail (§ 4.3). Finally, we discuss the existing architecture of the Gillian-JS compiler (§ 4.4), and present our implementation in § 4.5.

4.1 Key JavaScript concepts

JavaScript [39] is an object-based language, in that objects are the primary constructs through which most of the language's features are implemented. A JavaScript object is a collection of properties, split into that are *named* and those that are *internal*. Named properties can be considered to be equivalent to object fields in languages such as C++ and Java, and can hold primitive values as well as references to other objects. However, unlike those languages, objects in JavaScript can be marked as *extensible*, which allows more properties to be added to them. In addition, named properties are not only associated with a value, but also with a list of attributes that indicate how they can be used. For example, they have a `Writable` attribute which determines whether the value can be modified or not. On the other hand, internal properties are hidden from the user and exist purely to support the underlying mechanics of JavaScript. For example, every object has an internal property `@proto` (where we use the '`@`' prefix to denote that it is internal) which holds a reference to another object called its *prototype*. This, in turn, holds a reference to another prototype object, and so on until an object is reached that has a `null` prototype. These prototype objects form a *prototype chain*, and are the backbone of how JavaScript implements inheritance and other mechanisms related to object-oriented programming (OOP). In particular, it uses a *prototype-based*, rather than class-based, approach to the OOP paradigm. The language standard does not, in general, specify how such internal properties should be represented—it is left up to each implementation.

JavaScript can also be considered to be a functional language, since functions are treated like first-class citizens: they can be assigned to variables, passed as arguments to functions, returned from functions, and even be declared within other functions. This is because functions are also stored as objects on the heap. For example, when the following statement is executed, a new function object called `Person` is created:

```
var Person = function (name, age) {
  this.name = name;
  this.age = age;
}
```

The above example uses an *anonymous* function expression, since there is no identifier provided after the `function` keyword. It is worth noting that only function expressions can be anonymous. Functions are also used to define object types that can be instantiated to create any number of new objects, in which case they are termed *constructor functions*. New instances of a given object type are created by prepending the `new` keyword to the constructor invocation. When the constructor body is executed, the `this` keyword is bound to the new object being created, and any properties set via `this` will correspond to new named properties on the object. For example, the snippet above defines a `Person` object type, while in the code below, a new object `p` of type `Person` is created. `p` is assigned the properties `name` and `age`, which get initialised with the values `"Tom"` and `20`, respectively:

```
var p = new Person("Tom", 20);
```

Each function object that gets created has two specific internal properties: `@code`,

```
1  var Person = function (name, age) {
2    this.name = name;
3    this.age = age;
4  }
5
6  var f = function (name, age) {
7    var p = new Person(name, age);
8  }
9
10 f("Tom", 20);
```

FIGURE 4.1: Variable binding in JavaScript

which holds a representation of the ECMAScript function code, and `@scope`, which holds information about the environment in which the function object was defined and which therefore will be available to it upon its execution. An environment in JavaScript is represented by a list of *environment record* (ER) objects, which together form a *scope chain*. Each ER object has as its properties the variables and functions declared within the corresponding lexical scope, with the order in which it appears in the list reflecting the nesting of scope. The outermost ER is the *global* object, which is responsible for holding all the variables and functions declared in the global scope. We name this object `global`. When a function gets invoked, a new ER object is created that maps the function's parameters, local variables and nested functions declared in the function's body to their respective values. The ER object is the appended to the end of the current scope chain. During the execution of the function's body, variable references are resolved by traversing the scope chain up until the global object is reached. If the variable is not found in this object, then the variable is marked as `undefined` [39].

To illustrate the process of variable binding during function calls, we consider a revised version of our earlier example, shown in Figure 4.1. When this code is executed, there are two function objects created, `Person` and `f`, as shown by the blue boxes in Figure 4.2. As both functions are declared in the global scope, their internal `@scope` property will be the scope chain `[global]` (we use `[]` to denote a list). The `global` object, in turn, has the properties `Person` and `f`, containing references to the corresponding function objects. When the call to `f` is made on line 10, a new environment record object (`ER-f`) is created, containing the properties `name`, `age`, and `p`, with the first two being the formal parameters of `f` and the last being the local variable used in its body. The first two get the values `"Tom"` and 20, respectively, as these are the values passed as arguments during the call to `f`. The body of `f` is then executed in the scope `[global, ER-f]`, with the references to `name` and `age` on line 7 being resolved using the last environment record, and the reference to `Person` being resolved using the first environment record. The creation of the `ER-f` object ensures that local variables of `f` cannot be used outside of the scope in which they were defined. For example, adding the following line after the call to `f` would result in an error:

```
console.log(p); // ReferenceError: p is not defined
```

The existence of environment record objects means that JavaScript can naturally

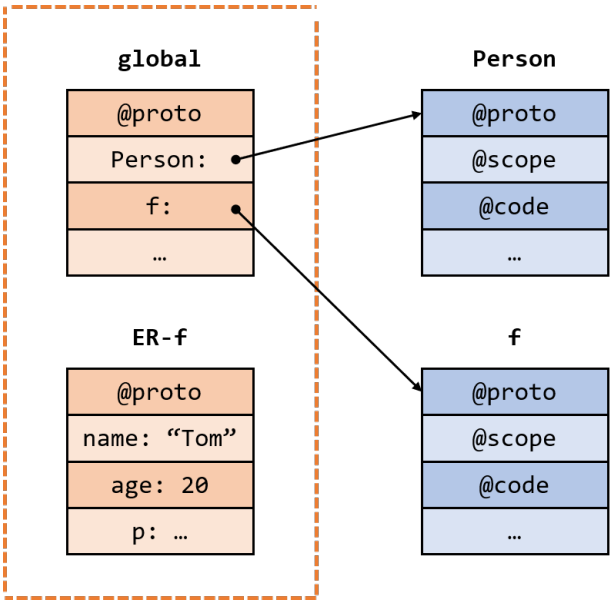

FIGURE 4.2: Variable binding in the JavaScript heap

```
1  function f(a, b) {
2    var c = a + b;
3    var g = function (d) {
4      return c + d;
5    }
6    return g;
7  }
8
9  var h = f(1, 2);
10
11 console.log(h(3)); // 6
```

FIGURE 4.3: Example of a closure in JavaScript

support the creation of closures. Instead of being garbage collected, the ER objects created during a function's execution can be kept in memory, meaning that a returned closure can still keep a reference to any non-local variables that were bound at the time of the enclosing function's execution. An example of this is shown in Figure 4.3. As before, `f` is function object whose scope is `[global]`. When `f` is executed on line 9, an `ER-f` object is created that binds the formal parameters `a` and `b`, and local variable `c` to the values 1, 2, and 3, respectively. When the function object `g` created, its internal `@scope` property will be the chain `[global, ER-f]`, and because `g` is returned by `f`, the reference to `ER-f` will be kept. Finally, when the call to `g` is made on line 11 through the variable `h`, its body will be executed in the scope `[global, ER-f, ER-g]`, where `ER-g` is a newly-created ER object holding a binding for the parameter `d`. The reference to `d` will then be resolved using this last environment record, while the reference to `c` will be resolved using the second environment record, leading to the expected value of 6 being returned. In addition, the binding for `c` will remain internal to `g`—any attempts to access it directly will result in a `ReferenceError` being thrown.

4.2 Module design patterns

As we have seen, any JavaScript variable declared outside the body of a top-level function automatically becomes part of the global scope. As a codebase grows in size, there will inadvertently be instances of separate parts of the code accidentally sharing or changing each other's global variables, leading to an undesirable effect known as *global namespace pollution*. In order avoid this, as well as to make code more maintainable, JavaScript developers have devised a number of design patterns that allow for greater encapsulation between unrelated code parts. They leverage the existing ES5 constructs in order to organise code into module-like segments that keep their internal state hidden and only allow access to it through an interface. In addition, the segments can still be kept inside the same JavaScript file.

As we have seen in the Figure 4.3 example, variables that get bound when a closure is created will remain accessible only from within the closure itself. This fact is exploited by the *module pattern* [36], an example of which is shown Figure 4.4. The closure returned on line 1 is an object with accessor and mutator methods for the module's internal `counter` variable, which remains hidden from the global scope and so behaves almost like a `private` field in Java. We can also see that the methods reference the same `counter` variable, since the expected value of 2 is outputted by line 16. In addition, the methods of the module are effectively namespaced, since any references to them in the outside scope must be prefixed with the module's name (in our case, `counterModule`).

```
1  var counterModule = (function () {
2    var counter = 0;
3
4    return {
5      increment: function () {
6        counter++;
7      },
8      get: function () {
9        return counter;
10     }
11   }
12 })();
13
14 counterModule.increment();
15 counterModule.increment();
16 console.log(counterModule.get()); // 2
```

FIGURE 4.4: Example of the JavaScript module pattern

The code also uses another JavaScript construct we have not yet introduced: an *immediately-invoked function expression* (IIFE). This is done with the `(function(){...})()` syntax, with the first outer pair of brackets being used to encapsulate the function expression in its own lexical scope and the second being used to immediately execute its body [44]. It saves us from otherwise having to declare an additional function object in the global scope:

```
var m = function () { ... }
```

```
var counterModule = m();
```

The module pattern can also be extended to introduce an import-like mechanism [36]. Global variables can be passed directly to the IIFE as function parameters, and the module would be able to use and alias these in any way it wanted. Figure 4.5 shows a variation of the previous example where the module 'imports' the starting value of the counter.

```
1  var counterStart = 5;
2
3  var counterModule = (function (startVal) {
4    var counter = startVal;
5
6    return {
7      increment: function () {
8        counter++;
9      },
10     get: function () {
11       return counter;
12     }
13   }
14 })(counterStart);
15
16 counterModule.increment();
17 console.log(counterModule.get()); // 6
```

FIGURE 4.5: Importing values with the module pattern

4.3 Node.js modules

Node.js[1] is a cross-platform runtime that allows JavaScript code to be executed outside of a browser environment. It is built on top of Google's V8 JavaScript execution engine (the same engine used within the Chrome browser), and features an extensive collection of built-libraries to support, among other things, filesystem I/O and networking-related operations. These allow it to be used for server-side scripting—where a webpage content is dynamically generated on a server before being returned to the user—and to build web servers themselves, unifying the development of a web application's frontend and backend under the same language. Outside of the core libraries, developers are also able to choose from the vast array of third-party libraries available through the Node package manager[2] (`npm`) ecosystem.

In order to support the usage of such libraries, as well as to allow developers to better structure their code, Node introduced one of JavaScript's earliest module systems. With it, a JavaScript program can be split across a number of different files, with each file becoming a module. Modules explicitly mark the variables and

[1] `https://nodejs.org/en/`
[2] `https://www.npmjs.com/`

functions they wish to make available to other modules by assigning these as properties to a special `module.exports` object, or alternatively to `exports`, which by default is an alias to it. In turn, a module can import the functionality provided by an external module using the `require` function, which takes as an argument the identifier of the module (typically its relative path) and returns its `exports` object [47]. Figure 4.6 shows an example. There are two modules, `main.js` and `circle.js`, with the latter exporting two functions, `area` and `circum`, using both the full `module.exports` reference as well as the `exports` shorthand.

```
var circle = require("./circle");

console.log(circle.area(2));
// 12.56...

console.log(circle.circum(4));
// 25.13...
```

(a) `main.js`

```
var pi = Math.PI;

var area = function (r) {
  return pi * Math.pow(r, 2);
}

var circum = function (r) {
  return 2 * pi * r;
}

module.exports.area = area;
// Using just `exports`:
exports.circum = circum;
```

(b) `circle.js`

FIGURE 4.6: A Node program with two modules

As well as specifying properties on the `exports` object, an exporting module can also directly replace it with a particular function or object, which may be useful if the module defines a single object type. In such case, the module must use the full `module.exports` reference. Figure 4.7 shows an example of this.

```
var Person = function (name, age) {
  this.name = name;
  this.age = age;
}

module.exports = Person;
```

FIGURE 4.7: `person.js`

In both examples, any variables or functions local to the module that are not exported (such as the variable `pi` in `circle.js`) are hidden from other modules. Node achieves this by using a technique similar to the one we have seen being employed by JavaScript module patterns. Before executing the module code, Node wraps it in the following function expression [47]:

```
(function (exports, require, module, __filename, __dirname) {
    // Module code lives here
});
```

This keeps top-level variables scoped to the module rather than the global namespace, and provides the module with access to the `exports` and `module` objects. It

also provides access to the convenience variables `__filename` and `__dirname`, which hold the absolute paths of the module file and the directory it is situated in, respectively. While these objects and variables appear as if they were defined globally, they are actually specific to each module. In particular, `module` is an object that gets created for each module when it is loaded the first time. For the module serving as the entry point to Node (for example, because it is the one passed as an argument to Node on the command line), this happens at the very beginning. For any other module, this happens during the first call to a `require` that references it. The module code is evaluated once, and the `module` object, along with its populated `exports` property, is cached. Any subsequent references to the same module will return the cached `exports` object, meaning that only one instance will exist of each module. This has some interesting consequences, particularly when the module dependencies form a cycle.

The mechanism for handling such cycles, as well as the names and semantics of the `module` and `require` constructs, largely stem from the CommonJS module specification [40], which influenced Node's early development. This was written as part of a project launched in 2009 (around the same time that Node was released) that aimed to create a standard module system for JavaScript outside the browser. The project has sine been abandoned, with the last edit to the specification made in late 2014, and with each server-side JavaScript implementation (including Node) continuing on its own path. However, the original specification document is still useful for understanding the core of Node's module system, and we will refer to it when discussing our implementation in § 4.5.

With reference to the program in Figure 4.6, we consider the full steps involved during its execution by Node, assuming `main.js` serves as the entry point:

1. Node begins by loading the `main.js` file, creating a module object for it and executing its body up until the call to `require`. As the identifier is a relative path (since it begins with `"./"`), Node attempts to resolve it by appending the it to the absolute path of `main.js`. As the file is referenced without an extension, Node will first attempt to look-up the exact filename. Assuming that a file called `circle` does not actually exist in the same directory, it will then try to resolve the path by adding the extensions `.js`, `.json`, and `.node`, in that order. In our case, the `.js` extension would match.

2. Node will then look up the path of `circle.js` in its cache. As this does not correspond to a previously-loaded module, it will switch to *synchronously* loading and executing the body of `circle.js`, creating a new module object for it and adding this to the cache. If `circle.js` had any `require` calls within its body, Node would switch to loading and executing the referenced modules in the same way.

3. Finally, when Node finishes executing `circle.js`, it switches back to `main.js`, assigning the `exports` property of the exported module object to the variable `circle`. It will then continue with the evaluation of `main.js`.

The approach taken by Node's module system has a number of interesting implications, particularly when compared to JavaScript's built-in module system that

was later added in ES6. The first is that it naturally supports dynamic imports. Because all of the module's code up until the `require` statement is executed before the referenced module is loaded, the identifier passed to the `require` call can be constructed during execution, for example by using variables [14]. In addition, the call to `require` may also be performed conditionally, such as by nesting it within an `if` block. In contrast, all ES6 imports—and the imports of subsequent editions of the standard until ES2020, which is set to have support for a dynamic `import()`—are resolved statically.

The second implication is that the main thread executing the body of module blocks while any referenced modules are loaded and executed. As Node is intended to be run server-side, the time the thread spends blocked would largely be determined by the time taken for Node to fetch the file from the filesystem, which, assuming a reasonable disk access time, would not be too significant. This design becomes far more problematic if used within the context of an application running client-side in the browser, since most of this time would be spent waiting for the file to be downloaded from the server [14]. Assuming typical latency numbers[3], this can be around 10 times slower than a disk access, and would therefore negatively impact the user experience. To avoid a similar issue, ES6 decouples the parsing and fetching of module files from their execution, with the first step responsible solely for building the module dependency graph and the latter for assigning values to variables shared between modules.

4.4 The Gillian-JS compiler

Gillian-JS is the instantiation of Gillian for the JavaScript language (in particular, targeting ES5 strict). As with Gillian-C, the compilation of the input source file to GIL happens across a number of intermediate representations. While some encompass various simplifications made to the language, others exist merely for backwards compatibility. This is because Gillian-JS includes a significant portion of the original JaVerT compiler [23]. Since the JaVerT work, however, the parser has been changed from Esprima, which is written in JavaScript and therefore had to be run in a separate process, to Flow, which is written in OCaml. The latter is maintained by Facebook and is frequently updated to be able to target latest additions to the ECMAScript standard, making it suitable for future use when Gillian-JS is extended to support these editions of the standard.

Figure 4.8 illustrates the structure of the Gillian-JS compiler. The steps within it are, in order:

1. **Parsing of the source JavaScript file to build a Flow AST.** Any lexical errors produced at this stage are propagated back to the user. The resulting AST conforms to the ESTree specification [46], which is the widely-adopted standard for JavaScript parsers. Every node in the AST contains information about its exact source code location, including the source file path, its start position, and its end position, with each position comprising a line number,

[3] `https://gist.github.com/jboner/2841832`

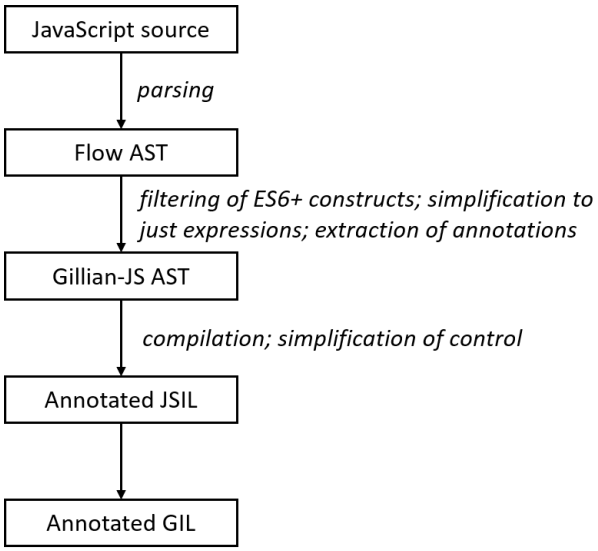

FIGURE 4.8: Compilation process in Gillian-JS

column number, and character offset. The parser also extracts comments from the file, keeping them in separate list. Like nodes, they are each associated with a source code location.

2. **Construction of a Gillian-JS AST from the Flow AST.** This stage is responsible for detecting the usage of any unsupported constructs, such as those coming from later editions of the ECMAScript standard (i.e. ES6+), and for building a simplified, almost JSON-like representation of the program that contains only expressions. In addition, separation-logic-related annotations and proof tactics are parsed from the comment strings captured by Flow and added, based on their source locations, to their corresponding expressions within the Gillian-JS AST.

3. **Compilation of the Gillian-JS AST to JSIL.** This step forms the bulk of the Gillian-JS compiler, and has been leveraged directly from the JaVerT work. JSIL (which stands for the JavaScript Intermediate Language) is a basic goto language with top-level procedures. It has a very similar syntax to GIL, and this is because it directly influenced GIL's design. For full details of this process and the semantics of JSIL, we refer the reader to the original JaVerT literature [23], as it is still relevant to the current implementation.

4. **Compilation of JSIL to GIL.** This is a straightforward translation step, with each JSIL command being expressed in terms of its equivalent GIL command.

4.5 Design and implementation

Deciding on how to best integrate support for Node's module system within the existing Gillian-JS compiler proved to be more challenging than the related work in C. Based on the design of both Gillian-JS and the Gillian framework, there were

two approaches we could take, both of which have their own advantages and difficulties: (i) resolve and load all referenced modules *before* the program is symbolically executed by Gillian; or (ii) do it *during* the process of symbolic execution. In other words, the choice is roughly between a static versus a dynamic approach, although these terms are used very loosely—everything Gillian-JS does is 'static', since the input program is not being executed concretely.

The first approach involves combining the code of all referenced modules into a single AST, and using syntactic wrappers similar to the ones we have seen in § 4.2 in order to keep their namespaces private. In addition, glue code needs to be added to ensure that, *during* symbolic execution, each module is executed in the same order that it would be by the real Node module loader. Since the loading and evaluation process happens synchronously in Node, this is not too difficult to emulate.

The second approach, on the other hand, involves passing control back to Gillian-JS whenever a call in GIL to `require` is encountered by Gillian. Gillian-JS would then be responsible for resolving the referenced path, parsing the file, compiling it to GIL, and passing it to Gillian for execution in a separate thread. In addition, the compilation step also needs to be modified so as to provide the module code with access to its own `module` and `exports` objects. Then, Gillian-JS needs to ensure that the same `exports` instance is returned to the importing module.

We ended up designing our implementation around the first approach, since it is the simplest and can be contained within the Gillian-JS compiler. It does, however, have some limitations, particularly when it comes to handling `require` calls with dynamically-constructed identifiers. We address this point as part of our evaluation in Chapter 6.

4.5.1 Overview

Figure 4.9 shows a summary of how we process a JavaScript program with multiple modules in order to create a single AST that can be symbolically executed. Unlike the extended version of Gillian-C, Gillian-JS still takes only one file path as input. We consider this to be the 'main' module. The file gets parsed and compiled to the Gillian-JS AST in the regular way. We then traverse the AST to extract the paths of all modules the file imports. Then, syntactic wrappers are added to the AST directly in order to produce an *augmented* AST. This process is repeated for all the imported module paths, as well as their transitive dependants. In addition, a separate `preamble.js` file is used to keep the definitions of built-in constructs such as `require`. This is parsed and combined with the rest of the augmented AST in order to create the final program.

In principle, the transformation and combination steps can be performed at any of Gillian-JS's intermediate representations that we saw in § 4.4. We chose, however, to perform them at the level of the Gillian-JS AST, since this is much simpler than the Flow AST but still maintains a close correspondence to the JavaScript language (unlike the lower-level JSIL representation, which eliminates most of the language's constructs).

In the remaining subsections, we detail how the transformations ensure that

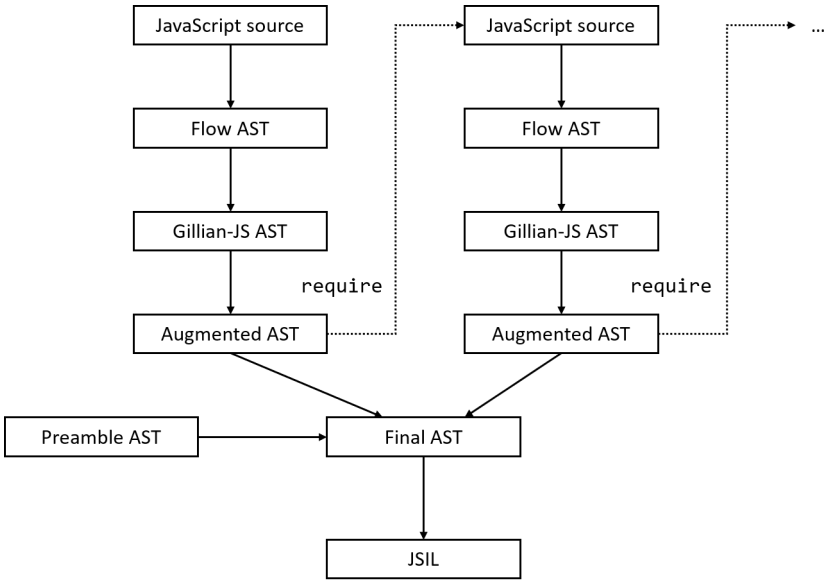

FIGURE 4.9: Design for Gillian-JS's module loader

the final program emulates the behaviour of Node's module loader. To guide our implementation, we refer to the CommonJS specification, the official Node documentation, and, when there is ambiguity, to Node's observable behaviour when ran on small examples.

4.5.2 Path resolution

To determine which other files are imported by each module, we perform a depth-first traversal of the AST data structure, and, if we encounter a call to `require`, resolve the imported module identifier to a full path by applying the following rules:

1. If the identifier begins with `"."` or `".."`, then we treat it as relative to the directory path of the current module. If this file does not exist, we throw an error. Otherwise, we obtain the file's canonical path, and add this to the set of imported modules.

2. Otherwise, the identifier is assumed to be a reference to either a core Node module or to a module imported via the Node package manager. At this point, if Node does not recognise it as a core module, it would attempt to find it in the `node_modules` directory (if it exists) of the current working directory, the parent directory, and so on until root of the filesystem is reached. However, as we are limiting our implementation to user-defined modules, we would throw an error.

In addition, if we did not throw an error, we rewrite the call to `require` by a call that uses the full file path. This makes the implementation of `require` simpler, as it eliminates any ambiguities that arise when modules use different relative paths to refer to the same module.

4.5.3 Module context

The CommonJS specification [40] defines the *module context* as the set of free variables that should be available to each module. They are 'free' in the sense that, from the module's point of view, they appear to be defined outside its top-level scope. At minimum, these must include: the `require` function; the `exports` object to which the module adds its API; and the `module` object, containing an `id` property that is the top-level identifier of the module.

To give each module access to these variables, we take direct inspiration from Node's function wrapper. We use the preamble file in order to define a global `Module` object type, which gets placed at the beginning of the combined program:

```
var Module = function (id, dirname) {
    this.id = id;
    this.filename = id;
    this.dirname = dirname;
    this.exports = {};
}

var _cache = {};
var _module;
```

Then, as part of the augmenting step, each module body gets wrapped in a function expression that is either executed immediately or stored for later. Each module also gets top-level responsible for creating its own instance of `Module` that can then be passed as the `module` parameter to the function. When the object gets instantiated, the `filename` and `id` properties both get assigned the canonical module file path, while the `dirname` property is assigned the canonical path of its parent directory. The `filename` and `dirname` properties are kept so that their values can be used to provide the `__filename` and `__dirname` parameters. Finally, the module object gets added to a cache using the file path as its key.

For example, every module other than the main one gets wrapped inside the following top-level code:

```
_module = new Module("/path/of/module.js", "/path/of");
_cache["/path/of/module.js"] = _module;
_module.load = function (exports, module, __filename, __dirname) {
    // ...
};
```

The function expression is assigned to a property since the module code must only be evaluated when the execution reaches a `require` call to it. In contrast, the main module gets wrapped in an IIFE, since it needs to start executing immediately:

```
_module = new Module("/path/of/main.js", "/path/of");
_cache["/path/of/main.js"] = _module;
(function (exports, module, __filename, __dirname) {
    // ...
})(_module.exports, _module, _module.filename, _module.dirname);
```

At this point, we should note that top-level code we generate does *not* run in strict mode, since this would mean overriding the strict mode configuration of each

individual module. This means that, although the function wrappers hide all of a module's variable and function declarations from the global scope, they cannot prevent a module running in non-strict mode from polluting the global namespace. This is because, in non-strict JavaScript, undeclared variables automatically become part of the global scope, even if they are used within an IIFE:

```
(function () {
  x = 10;
})();

console.log(x); // Does not throw ReferenceError
```

However, this is not something that Node can prevent against either.

4.5.4 The `require` function

The missing piece of the top-level code is the definition for `require`. In principle, this is straightforward: given the module identifier (which is now always a canonical file path), it needs to fetch the corresponding `Module` object from the cache, call its `load` function, and return its `exports` property. However, there is one catch, arising when two modules form a dependency cycle. For example[4], we consider the situation presented in Figure 4.10, assuming that we execute `a.js` first. `a.js` loads `b.js`, and in turn `b.js` attempts to load `a.js`, which would trigger an infinite loop. The CommonJS specification states that, in such case [40]:

" [...] the object returned by "require" must contain at least the exports that the foreign module has prepared before the call to require that led to the current module's execution. "

This means that the `require` call should return an object that contains the property `done` with the value `false`, since this property was set by `a.js` before the `require` call into `b.js` was made.

```
1 console.log("a starting");
2 exports.done = false;
3
4 var b = require("./b");
5 console.log("b.done: " + b.done);
6
7 exports.done = true;
8 console.log("a done");
```

(a) `a.js`

```
1 console.log("b starting");
2 exports.done = false;
3
4 var a = require("./a");
5 console.log("a.done: " + a.done);
6
7 exports.done = true;
8 console.log("b done");
```

(b) `b.js`

FIGURE 4.10: A dependency cycle

Looking at the output produced by running the program in Node, we can see that this exactly what happens:

```
a starting
b starting
```

[4] Adapted from the example in the Node documentation [47].

```
a.done: false
b done
b.done: true
a done
```

To ensure we produce the same behaviour, we additionally associate each `Module` object with a *status* property that indicates which stage of the loading and evaluation process the corresponding module has reached. When we instantiate the `Module` object, this gets initialised to `"NOT_LOADED"`. Then, within the module's function expression, we add a statement before the module's code to set the status to `"LOADING"` and another at the end to set it to `"LOADED"`. Finally, the module's code gets evaluated only the first time it is required, since we make the `require` function only call `load` when the module's status is `"NOT_LOADED"`. However, all `require` calls return the module's `exports` object, meaning that it can be used by other modules even when it is only partially prepared:

```
function require(id) {
    var module = _cache[id];
    if (module.status === "NOT_LOADED") {
        module.load(module.exports, module, module.filename,
            module.dirname);
    }
    return module.exports;
}
```

Chapter 5

Continuous Reasoning

The work undertaken in the previous two chapters serves to enable Gillian-C and Gillian-JS to analyse programs spanning different source files, by extending them to incorporate the importing and exporting mechanics of their respective target languages. This is an essential first step in the two tools being able to target *any* real-world project, irrespective of the number of lines of code. In this chapter, we detail the work done in paving the way for the next step: allowing their analyses to scale to large codebases. It is desirable to have this capability within any future instantiation of Gillian, not just the current ones for C and JavaScript. For this reason, we focus our work on developing *general* mechanics for tracking source code changes and re-using previous analysis results, and on incorporating them within the Gillian library itself. In particular, there are two challenges we address:

1. constructing a generic (i.e. language-independent) representation of an input program's source files as well as their inter-dependencies, and building a mechanism that can determine, based on the set of changed files, which parts of the GIL program are affected; and
2. for each type of analysis supported by Gillian (verification, automatic compositional testing, and symbolic testing), developing an *incremental* variant that uses the information from the first step together with previously-stored results—the format of which is particular to that analysis—in order to only analyse the parts of the program that have changed.

5.1 Overview

We develop Gillian's continuous reasoning system using a modular design, shown in Figure 5.1. At its core is a module that tracks the changes made to the source program across two successive invocations of Gillian and determines which other parts of the program they affect. It consists of two parts:

1. A base component responsible for directly relating the changes made in the source code to the parts of the GIL program they correspond to. It operates

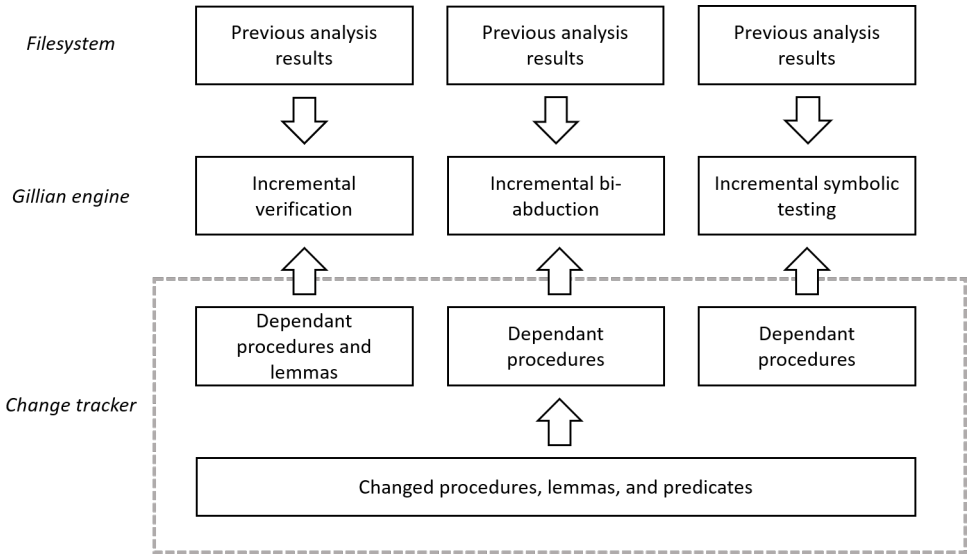

FIGURE 5.1: Design of Gillian's continuous reasoning system

at the granularity of changed GIL components, which we recall consist of procedures, predicates and lemmas. For each type of GIL component, it returns the set of components that have been added, the set of components that have been deleted, and the set that of components that existed previously but have been modified.

2. Three analysis-mode-specific modules, each responsible for computing the transitive dependants of the procedures, lemmas and predicates that have been marked as changed by the base component, adhering to rules that are specific to that analysis.

The information from the second part is fed into Gillian's three analysis modules, each of which now operates in two modes: normal, where the whole program is analysed, and incremental, where only the subset deemed necessary is. The user can switch between these two modes by using the `--incremental` command-line flag. When running in incremental mode, the analysis modules display the results computed from the previous run for the parts of the program that do not get analysed.

5.2 Tracking source code changes

Each procedure, lemma, and predicate that forms part of a compiled GIL program can always be categorised into one of three types:

1. Those whose definitions come from one of the user source files.
2. Those whose definitions come from one of the instantiation's runtime files. For example, in Gillian-C, this would include the GIL implementations of the C standard library functions.
3. Those that get constructed by the instantiation's compiler, for example because they relate to the mechanics of that particular instantiation's memory

model. Examples of these include the `i__initialize_genv` procedure and `i__global_env` predicate that get created by Gillian-C's compiler, which we saw in § 3.3.4.

Henceforth, we refer to procedures, lemmas and predicates that belong to the first category as *non-internal*, and those that belong to the second and third categories as *internal*.

Procedures, lemmas and predicates that belong to the second category can automatically be ignored by the change tracker, since they will not change between successive runs of Gillian.

While those in the third category may certainly change, they only change *in response* to changes made elsewhere within the source code. We therefore do not need to track them separately. For example, in Gillian-C, the `i__initialize_genv` procedure will only change when a new global variable or function declaration is added to the program by the user.

This leaves us to consider those in the first category. For each of them, we can record the path of the source file that contains their corresponding definition. For example, for a Gillian-C procedure, this would be the file containing the corresponding function declaration, and for a Gillian-C predicate, it would be the file containing the comment. Then, if we have the set of files that have changed, we can use this to build an over-approximation of the GIL components have changed, based on whether their source definitions belonged to those files.

To determine how the source files have changed, we implement a hash-based algorithm. We make each instantiation's compiler responsible for providing Gillian, along with the compiled GIL program, a set F of *all* source file paths that it used to construct the program. For every one of these paths, Gillian computes a hash of the file contents. Then, if we have the set F_{prev} of all source files used in the previous run together with their hashes (for example, because they were persisted between runs), we can determine the following:

- The set A of all files that have been *added*, given by $F \setminus F_{prev}$.
- The set D of all files that have been *deleted*, given by $F_{prev} \setminus F$.
- The set of all files that were present in both runs, given by $F_{prev} \cap F$. For every such file f, we compare its hash from the current run to its hash from the previous run. If they are different, then f is added to the set M of *modified* files.

5.2.1 Tracking header file changes

For the most part, the semantics of a file within the program changing are straightforward: if the file has been modified, then we assume that everything it defines must have changed. We then determine the relevant subset of the GIL program that directly corresponds to these definitions. However, when it comes to C, determining what it means for a header file to change is less obvious. We saw in Chapter 3 that

header files are typically used to store the forward declarations of functions that are used across multiple source files. They are also commonly used to store `struct` and `union` type definitions. If these change, then the analysis results for any functions using them may no longer be valid.

To handle this, for each header file we keep track of the list of files that include it. Then, if by the above algorithm a header file is determined to have been modified, all the files that include it are also marked as having been modified. For example, for a C program consisting of the files `foo.c`, `bar.c`, and `foo.h`, where the header `foo.h` is included by both `foo.c` and `bar.c`, we produce the following mapping:

- `foo.h`: [`foo.c`, `bar.c`];
- `foo.c`: [];
- `bar.c`: [].

To build this mapping, we first determine the list of header files that each source file includes. We do this by invoking the system C preprocessor with the `-MM` option. This causes it to output all the dependencies of the source file, excluding standard system headers. Then, for every header within the program, we look up the source files that include it within their list of dependencies.

5.3 Computing dependencies

Before delving into methods for determining transitive dependencies, which are specific to each analysis mode, we first consider the possible ways in which procedures, lemmas, and predicates can directly depend on each other.

Procedures For simplicity, within the context of the change tracker, we treat a procedure's body and its separation-logic specification (if provided) as a single atomic unit. The primary dependencies of a procedure are the other procedures that it calls, which may include itself. In verification, a procedure may also directly depend on a number of predicates and lemmas. Predicates may form part of its pre- and post-condition assertions, or may folded and unfolded at various points within the body (through the FOLD and UNFOLD proof tactics). Lemmas may similarly be invoked within the procedure's body with the APPLY proof tactic; using them is akin to performing a 'logical function call' that consumes the current symbolic state and produces another symbolic state.

Predicates A predicate consists of a name and a definition, which is the SL assertion to which the predicate gets unfolded to. The only direct dependencies of predicates, therefore, are the predicates that appear within their definitions.

Lemmas Although lemmas have not yet been formalised within Gillian-C and Gillian-JS, they feature heavily within Gillian-WISL, the instantiation of Gillian for a toy imperative language intended for teaching SL. The primary components of a lemma are: its *hypothesis*, which is an SL assertion describing its pre-condition; its *conclusion*, which is an SL assertion describing its post-condition; and its *proof*, which is a list of proof tactics that, assuming the hypothesis, allow Gillian to prove its conclusion. It therefore has two types of dependencies: predicates, which are either included within the SL assertions or used within the proof, and lemmas, which may also be used within the proof.

Each of these dependencies can either be computed *before* the program is symbolically executed, (i.e. 'statically'), by inspecting the GIL AST directly, or *during* the process of symbolic execution (i.e. 'dynamically'), by for example detecting the execution of each procedure call, predicate fold/unfold, and lemma application.

Determining the dependencies between procedures statically will clearly not suffice, as both C and JavaScript allow for functions to be called indirectly. In C, this can be done through the use of function pointers, such as the example in Figure 5.2 shows. It would be impossible to establish that `bar` calls `foo` based purely on `bar`'s syntactic structure. The call would, however, be observed during the program's symbolic execution. This means that a dynamic approach is more suitable.

```
1  #include "stdio.h"
2
3  int foo() {
4      return 10;
5  }
6
7  int bar(int (*f)(void)) {
8      return 5 * f();
9  }
10
11 int main() {
12     int (*f)(void) = &foo;
13     printf("%d\n", bar(f));
14 }
```

FIGURE 5.2: Indirect function call in C

On the other hand, there is no notion of an 'indirect' predicate usage or lemma application. The predicates that occur within a procedure can be determined by looking at all the assertions within its specification and by inspecting its body for any fold and unfold proof tactics. Similarly, we can check the procedure's body for any lemma applications. This means that a static approach is more suitable.

Because of this, we choose to split the construction of the dependency graph in two stages:

1. Before symbolic execution, we traverse the GIL AST in order to determine the direct dependencies involving predicates and lemmas.

2. During symbolic execution, we update the dependency graph each time a new procedure call is made, unless that edge already exists.

5.4 Incremental verification

In Gillian's verification mode, analysis happens at the 'unit' of functions and lemmas. Each function (respectively lemma) within the program is verified by initialising the symbolic state with its pre-condition, symbolically executing its body (respectively proof), and attempting to unify the final symbolic state with its post-condition. If the unification step succeeds, then the function or lemma is proved to be correct with respect to its specification. Otherwise, the verification fails.

Each such 'test' is effectively carried in isolation from all the other procedures and lemmas within the program. If the procedure being verified makes a call, then this call is checked with respect to the specification of the callee, not its code. Similarly, if the procedure has a lemma application within its body, then this is checked with respect to the lemma's specification, not its proof.

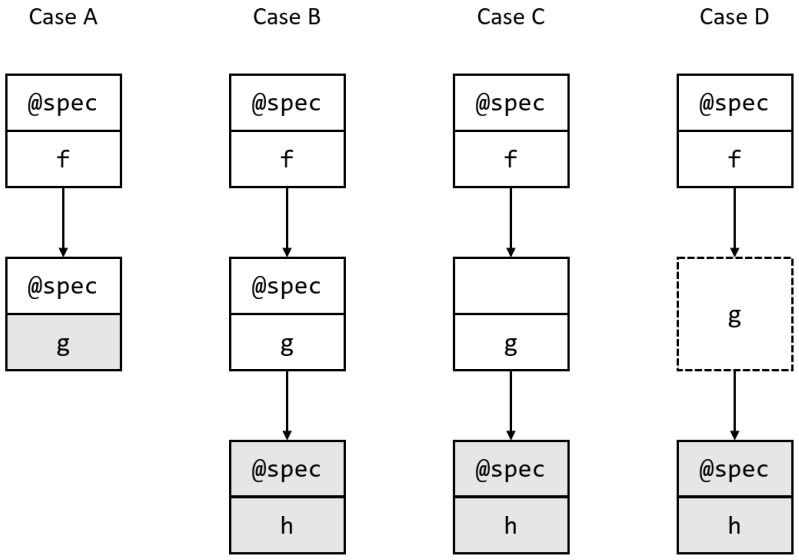

FIGURE 5.3: Incremental verification cases (grey denotes a change)

In this way, verification proofs are inherently incremental: if the specifications of a procedure's callees do not change between subsequent analyses, then the verification result for that procedure still holds. This forms the general intuition behind how we compute transitive dependants in verification. To help refine it into a full algorithm, we consider the main cases that can occur. We show these in Figure 5.3, and discuss them below. In the diagram, solid boxes are used to represent the bodies and specifications of user-defined procedures, whereas dotted boxes are used to represent internal GIL procedures.

Case A Assume that the program contains the functions `f` and `g`, both annotated with specifications, and `f` calls `g`. If, between two runs, the body of `g` changes but its specification does not, then `f` does not need to be re-verified.

Case B Assume that program contains the functions `f`, `g`, and `h`, all annotated with specifications and calling each other in that order. If *both* the body and specification of `h` changes between two runs, then `g` must be also be re-verified, since there is no guarantee that its proof will still succeed. However, as the specification of `g` has not changed, `f` does not need to be re-verified.

Case C Assume that the program contains the function `f`, `g`, and `h` as in Case B, but this time `g` is a function provided by the user *without* a specification. In verification, Gillian executes calls to procedures without specifications in the normal way. This means that we need to consider the body of `g` as effectively inlined within `f`. Therefore, if both the specification and body of `h` changes, then `f` also needs to be re-verified.

Case D Assume that the program contains the functions `f`, `g`, and `h` as previously, but this time `g` is an internal GIL function. This case can certainly occur in Gillian-JS, since all JavaScript function objects inherit an `apply` method that provides an alternative way of invoking them. The snippet below shows an example—the call to `apply` would be compiled to a call to the equivalent GIL implementation:

```
// @spec
var h = function (n) {
  console.log(n);
}

// @spec
var f = function () {
  return h.apply(null, [10]);
}

f(); // 10
```

The internal function `g` is treated like a function that was provided without a specification, which means that this scenario is equivalent to that in Case C. Therefore, if the specification of `h` changes, then `f` must also be re-verified.

With these, we arrive at the following heuristic for computing dependencies in verification: if a function changes, then its *closest non-internal ancestors with specifications* according to the call graph must also be re-verified.

5.5 Incremental bi-abduction

For the most part, bi-abduction proofs share the same incremental characteristics as those in verification, since procedure calls are also checked with respect to the specifications of the callees. However, unlike in verification, the specifications are *not* known upfront, since the analysis is performed on bare programs. This means that, if a procedure has changed between two analysis runs, the only way to determine whether its specification has also changed, and hence whether its dependants up

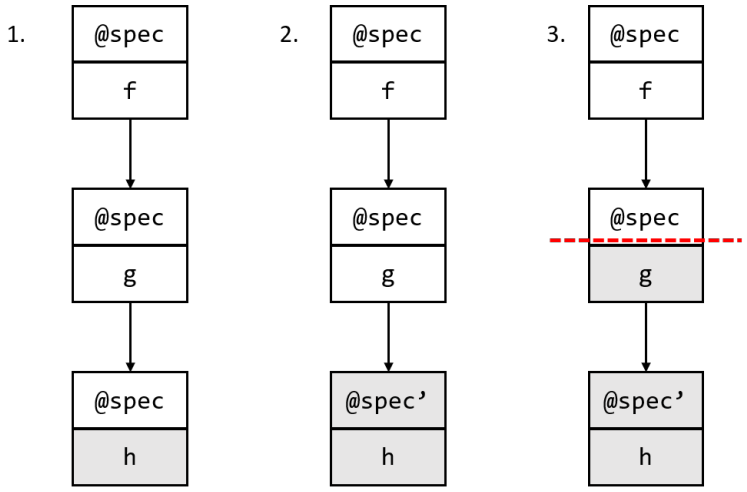

FIGURE 5.4: Incremental bi-abduction example

the call graph must also be re-verified, is to: (i) analyse it and (ii) compare the newly-derived specification with the one that was previously stored for it.

If it is unchanged, then the analysis can stop. Otherwise, the analysis must propagate up the call graph until any non-internal procedures are reached, at which point they will be subjected to the same check. We note that we no longer have to distinguish between dependants that have specifications and those that do not, since Gillian's bi-abduction mode derives specifications for all functions within the program.

To illustrate this, we consider the example shown in Figure 5.4. We assume that the program consists of the three functions `f`, `g`, and `h`, that call each other in that order and for which we have previously derived (and stored) specifications. We also assume that the function `h` is determined to have changed. In this case, we do not know whether the specification we had previously derived for it still holds, so we proceed to re-check `h`. Assuming that its specification has in fact changed (from `@spec` to `@spec'`), we then proceed one level up the call graph. As `g` is a non-internal function calling `h`, it will also get re-checked. However, assuming that Gillian derives the same specification for `g` as the one that was previously stored, the analysis does not proceed up to `f`.

5.6 Incremental symbolic testing

It is less obvious what constitutes an incremental analysis within the context of Gillian's symbolic testing mode, as this mode requires everything that is transitively called by the program's `main` method to be executed on each invocation. However, considering that it is most commonly used within Gillian's bulk execution mode, where a collection of symbolic test files (each with its own `main` method) are executed within the same running instance, we can define the following basic notion of incrementality: each test file should only be re-executed if its `main` method transitively calls a function that has changed.

To achieve this, we apply the change-tracking and dependency-computation

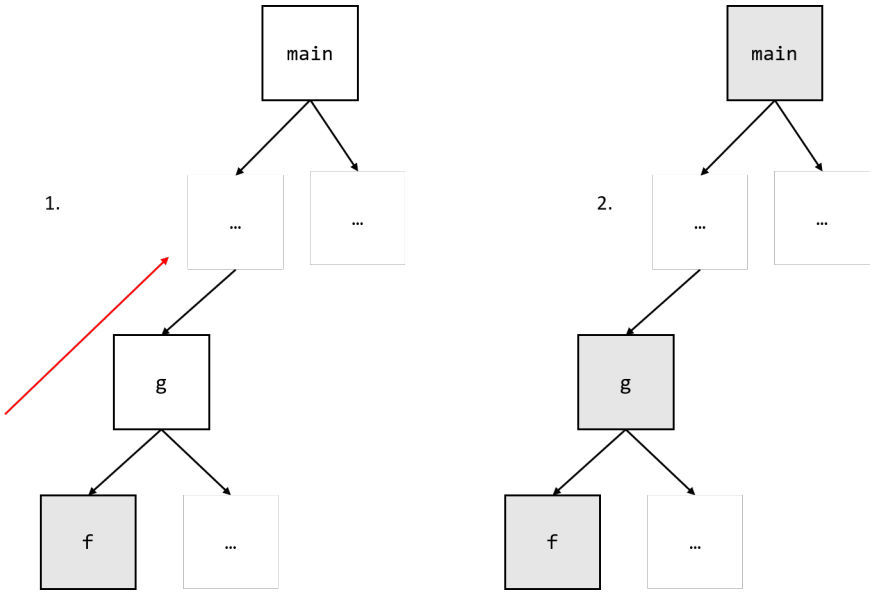

FIGURE 5.5: Incremental symbolic testing example

mechanisms to each test file individually. We do this because each test file is an independent program, and keeping procedures for distinct programs within the same call graph will invariably lead to name clashes—not least because they each have a definition for `main`. The incremental algorithm then consists of computing the transitive dependencies all the way up the call graph, as shown by the example in Figure 5.5. In symbolic testing mode, we have no notion of specifications, so this step a straightforward traversal. Then, if the `main` function is within this set of transitive dependants, as in the example, the test should be re-ran. Otherwise, its previously-stored result should be fetched.

Chapter 6

Evaluation

The primary objective of this project has been to provide the Gillian ecosystem with the necessary foundations to begin analysing large, real-world codebases that span multiple files. To this end, we have extended the implementation of the core Gillian platform along with its instantiations for C and JavaScript in two important ways: (i) we have added the support for each language's specific import and export mechanisms, and (ii) for each of Gillian's analysis modes, we have developed an incremental variant that can leverage previously-stored analysis results in order to minimise the analysis work done between successive runs of Gillian. We evaluate each of these deliverables with respect to the following criteria:

1. The degree to which it makes Gillian's adoption within real-world projects easier and/or more viable.
2. Its demonstrative functional correctness.
3. Its performance in terms of the time taken for Gillian to perform an analysis.

We structure our evaluation as follows:

- We start by evaluating Gillian-C's new import/export mechanism, by using it to symbolically test a popular data structure library (§ 6.2).
- We then perform a similar evaluation for Gillian-JS (§ 6.3).
- Next, we provide an evaluation of Gillian's incremental mode, by constructing a small test suite that exercises it under a range of important scenarios (§ 6.4).
- We then discuss the main limitations of our overall work (§ 6.5).
- Finally, we conclude our evaluation by discussing the key lessons we learnt along the way, and reflect on our overall experience with contributing to the Gillian platform (§ 6.6).

6.1 Experiment setup

Unless stated otherwise, all experiments and benchmarks within this evaluation have been performed on a machine with an Intel Core i7-4980HQ CPU 2.80 GHz, DDR3 RAM 16GB, and a 256GB solid-state hard-drive running OSX. This is the same machine on which all original Gillian experiments were performed.

All test times quoted represent the 'real' wall-clock time as measured by the OSX `time` utility, and are the average of five runs carried out under similar conditions, excluding an initial run done to discount the effects of filesystem caching.

6.2 Gillian-C evaluation

We evaluate the ability of Gillian-C's extension to analyse multi-file C projects by symbolically testing Collections-C,[1] a real-world data-structure library for C with over 2K stars on GitHub. It has approximately 5.2K lines of code and incorporates a wide-array of C-specific features and idioms, such as pointer arithmetic and structures. The data structures it provides include arrays, lists, hash tables, ring buffers, and priority queues. To test Collections-C, we leverage the comprehensive symbolic test suite written for Gillian-C by Fragoso Santos et al. as part of [22].

6.2.1 Ease-of-use

Because of Gillian-C's previous limitations, each test file within the original suite had to contain a copy of the library code it tested. In many cases, this was an amalgamation of various library files that depended on each other. For example, because the library uses an array-based implementation for stacks, the stack tests additionally had to include the entirety of the library's array module. Moreover, since there was no way to interface with CompCert's call to the preprocessor and provide a header search path, the library's header files had to be included within the same directories as the tests that needed them.

```
├── tests/
│   ├── array/
│   ├── deque/
│   ...
│   └── treetable/
├── test-utils/
│   ├── utils.h
│   └── utils.c
└── lib/
    ├── include/
    ├── array.c
    ├── common.c
    ...
    └── treetable.c
```

With Gillian-C's new capabilities, we have been able to entirely rewrite the original test suite in order to separate the library code from the tests and extract utility functions (which were also duplicated across the test files) into a separate file. The library code is kept entirely within its own directory, and this directory maintains the exact same structure as the Collections-C repository does on GitHub. This layout is shown on the right.

The tests can now be run by issuing a command that is similar to what a developer might use if they were compiling the project via the command line (for

[1] `https://github.com/srdja/Collections-C`

example, using `gcc`). In particular, they can now easily specify header source paths as well as the locations of the source code. For example, if they wanted run the `array` symbolic tests from the project's root directory, they could run:

```
gillian-c bulk-wpst tests/array -S lib -S test-utils \
    -I lib/include -I test-utils
```

This shows that Gillian-C now has far greater potential to be used 'out-of-the-box' by a general developer, since it can be fit to analyse a C project regardless of its structure, as opposed to the other way round. However, we acknowledge that this command-line workflow will probably not suffice for most real-world C project, as these would typically use build tools such as `Make` in order to compile their sources. For these, it would be more helpful if Gillian-C could be directly integrated within the build system—for example, by being able to prepend `gillian-c bulk-wpst` before the build command. We will revisit this point in Chapter 7.

6.2.2 Correctness and performance

To test the correctness of our implementation, we run Gillian-C on the new version of the test suite, and compare the results with those that were published in [22], which used the single-file version. We present the outcome in Figure 6.1. For each test folder, which corresponds to a particular data structure in Collections-C, we report: (i) the number of symbolic tests; (ii) the number of symbolic tests passing; and (iii) the time taken to run the entire folder, in seconds.

Test folder	# Tests	# Passing	Time (s)
`array`	21	21	4.98
`deque`	34	34	5.06
`list`	37	37	12.45
`pqueue`	2	2	2.32
`queue`	4	4	1.48
`ring_buffer`	3	3	0.93
`slist`	37	37	7.52
`stack`	2	2	0.83
`treeset`	6	6	2.08
`treetable`	13	13	4.64
Total	159	159	42.29

FIGURE 6.1: Collections-C test results

We note that [22] reported two extra tests for `array` and `slist` (singly-linked list); discounting these, we obtain the same number of tests passing.

When running the tests, we had to use the `--ignore-undef` option we built into Gillian-C in order to stop our linker from complaining about references to undefined symbols. This is because some functions within the library code makes use of the `qsort` sorting function that comes from the C standard library, and there is no current implementation for it in GIL. As these functions never actually get called during the test suite's execution, suppressing the error had no bearing on the results. In general, it is right for Gillian-C to throw such errors, since most other analysis

tools (including Infer) would also complain if the program cannot be compiled. However, we acknowledge that, until Gillian-C matures further and supports more of standard library, most projects will require the use of `--ignore-undef` when being analysed.

The total time taken to run all the tests shows an apparent overhead of about 7.5% when compared to the total in [22]. The main reason for this is that, unlike the paper, we were not able to run the tests in Gillian's parallel mode due to issues relating to our setup. In this mode, the symbolic engine forks execution when reaching a program branch by actually making a `fork` system call, as opposed to merely executing both branches sequentially.

However, it is worth noting that, despite the fact that our test results were only executed sequentially, the times are almost identical with, if not better, than those in the paper for the larger test suites such as `list`. This is because our work has also added a number of optimisations to Gillian related to the caching of compiled GIL files. The effect of these is most visible when the number of test files, and therefore total number of files that get compiled during the same running instance of Gillian, increases. In fact, when we use our setup to run the entire 159 tests together (i.e. by running the `tests` folder), we obtain a total execution time of **34.35 seconds**, which is around 13% lower than the time quoted in the paper.

6.3 Gillian-JS evaluation

We evaluate our extension of Gillian-JS in a similar spirit to Gillian-C, choosing as our case study Buckets.js,[2] a real-world, self-contained data-structure library for JavaScript. It has over 1K stars on GitHub and contains about 1K lines of JavaScript, using almost every JS-specific construct. It provides implementations of linked lists, sets, multi-sets, and heaps, among other data structures. We test it using Gillian-JS's whole-program symbolic execution mode, and, as with Collections-C, leverage an already-existing symbolic test suite. In particular, we adopt the tests written by Fragoso Santos et al. that were previously used to evaluate JaVerT 2.0 in [24] and Gillian-JS in [22].

6.3.1 Ease-of-use

In the original test suite, Gillian-JS's previous limitations necessitated the library code to be copied in each individual test file, since the library functions had to be in the same global scope as the top-level code testing them.

With Gillian-JS's new capabilities, we have been able to rewrite the original tests and extract the library code into its own CommonJS module, `buckets.js`. As the library defines a single top-level `buckets` object, it suffices to add the following statement within the library file:

```
module.exports = buckets;
```

[2] `https://github.com/mauriciosantos/Buckets-JS`

Then, given the directory structure shown on the right, each test file (e.g. located in `array`) can import it in the following way:

```
var buckets = require("../../buckets");
```

```
├── tests/
│   ├── array/
│   ├── bag/
│   ...
│   └── stack/
└── buckets.js
```

The project structure we adopted is largely arbitrary. We made the choice to keep the entire library within its own file since the original Buckets.js repository does *not* actually use Node modules (or in fact, any JavaScript module system). While the source code on GitHub appears to be composed of several different files, they actually all get combined within the same namespace when the project gets built.

However, we believe this example still highlights the vast leap made by Gillian-JS in being able to target real-world repositories running in Node (or using Node-like constructs) with minimum setup by the user. In particular, a developer simply needs to specify the name of the script (or symbolic test) they wish to analyse (or execute); Gillian-JS quietly resolves and loads any external modules referenced. For example, if they wanted to run a symbolic test `arrays1.js` within the `array` folder, they would run:

```
gillian-js wpst tests/array/arrays1.js
```

6.3.2 Correctness and performance

We evaluate the correctness of our implementation by running the new version of the tests, and comparing the results with those reported in [22], which used the single-file version. We present the outcome in Figure 6.2. As before, we report for each folder: (i) the number of symbolic tests; (ii) the number of symbolic tests passing; and (iii) the time taken to run the entire folder, in seconds.

Test folder	# Tests	# Passing	Time (s)
`array`	9	9	3.59
`bag`	7	7	6.53
`bstree`	11	11	27.12
`dictionary`	7	7	3.09
`heap`	4	4	8.18
`linkedlist`	9	9	5.12
`multidictionary`	6	6	6.87
`priorityqueue`	5	5	15.66
`queue`	6	6	2.64
`set`	6	6	14.66
`stack`	4	4	1.63
Total	74	74	95.09

FIGURE 6.2: Buckets.js test results

While we pass all the tests, we can immediately notice a significant slowdown (approximately 2x) in performance when compared with the times reported in [22],

which were obtained when running using Gillian's parallel mode. The fact that an absence of parallelism produces such a performance hit suggests that the Buckets.js library features a lot more complex branching than Collections-C, where the difference between the two execution modes was much smaller. It also suggests that, despite the other optimisations made on our part (such as better caching during compilation), the symbolic engine continues to be determining factor that affects the performance of a Gillian instantiation.

6.4 Evaluation of Gillian's incremental mode

6.4.1 Setup

To evaluate the implementation correctness of Gillian's incremental mode, we construct a test suite that incorporates a number of key scenarios of a source program changing between successive analysis runs. We have chosen to ground the tests in the context of Gillian-C, however, since they are testing mechanics specific to the Gillian library, their results are applicable to Gillian-JS as well.

Each test directory consists of:

- A `src_before` sub-directory, containing an initial set of C source files.
- A `src_after` sub-directory, containing the same copy of the source files, but with a slight modification made in one (or several) of them.
- An OCaml test script that, when executed: (i) copies the files from `src_before` into the test directory; (ii) analyses the files using Gillian-C in normal mode, and records its output; (iii) copies the files from `src_after` into the test directory, thereby 'applying' a set of changes to the program; (iv) runs Gillian-C again, this time with the `--incremental` flag enabled, and compares the two outputs in order to check that only the affected functions (and their transitive dependants) were re-analysed.

Test name	Analysis mode	Passing
`add_proc`	Verif.	✓
`change_header`	Verif.	✓
`change_pred`	Verif.	✓
`change_proc`	Verif.	✓
`no_change`	Verif.	✓
`remove_proc`	Verif.	✓
`transitive_call`	Verif.	✓
`change_proc`	Bi-abd.	✓

FIGURE 6.3: Incremental test results

6.4.2 Tests summary

Figure 6.3 shows a breakdown of the test suite. We focus most of the tests on Gillian-C's verification mode, since this has the most intricate rules regarding the decision of what to re-analyse. This is in part due to the many extra constructs present, such as specifications, predicates, proof tactics, etc.

For most, the name provides an apt description of the scenario being tested. For those that are more ambiguous, we provide a brief description of the scenario below:

- `change_proc` (verif.): The program contains three functions, `f`, `g` and `h`, all annotated with SL specifications. `f` calls `g`, and `g` (together with its specification) resides in a file separate to the other two. If the file containing `g` is modified, then Gillian must assume that the specification of `f` could have changed. In this case, only the following two functions should get re-verified: `f` and `g`.
- `transitive_call`: The program contains four functions, `f`, `a`, `b`, and `g`, calling each other in that order and with only `f` and `g` being annotated with SL specifications. `g` resides in a file separate to the other three. If that file changes, then only the following two functions should get re-verified: `f` and `g`.
- `change_proc` (bi-abd.): The program contains three functions, `f`, `g`, and `h`, with no SL specifications (since we are running in bi-abduction) and calling each other in that order. `h` resides in a file separate to the other two. If the file containing `h` is modified, then Gillian must first check `h`. However, the change made ensures that `h`'s derived specification does not change. Therefore, only the only function that ends up being checked again is `h`.

6.4.3 Discussion

Our implementation passes all the test scenarios that we have constructed. While there are certainly many more that could (and should) be explored, we believe the ones we have demonstrate that Gillian's incremental mode has the necessary foundations to start being applied to larger codebases. In particular, they show it can make sense of some of the more complex modifications a developer might make to a program between successive invocations of Gillian, and use re-use previous results in order minimise the analysis work done. We also note that the change-tracking mechanism we have developed does not rely on any external tools such as Git, making it easier for Gillian to be used 'out-of-the-box'.

Granted, it is unlikely that the more intricate cases arising in verification will have a chance to be exercised on large codebases, since this would require a substantial effort on the part of the Gillian researchers in order to write all the necessary annotations. Instead, we envisage that Gillian's automatic compositional testing (i.e. bi-abduction) mode is the most likely out of the three analysis modes to be successfully deployed at scale. When this happens, even if the incremental algorithm itself does not get used, we believe that the infrastructure we built *around* it, such as the dynamic call graph generation and the mechanism for computing transitive dependants, will certainly be of use.

6.5 Known limitations

In this section, we discuss some of the main limitations of our work pertaining to both multi-file and incremental analysis.

Lack of support for dynamic `require` calls. The approach we have taken to integrate support for Node-style modules within Gillian-JS means that we can only handle `require` calls that use static module paths (i.e. string literals). This is because, for simplicity, we perform the module resolution and loading step during compilation to GIL. In contrast, because Node executes all the code up to the `require` statement before resolving the required module, the module specifier can use variables. For example:

```
var lang = "en";
var path = "module_" + lang;
var m = require(path);
```

It is unclear how much of a serious limitation this is, since it is hard to tell whether it is a commonly-used feature within real Node projects. Adding support for it will be somewhat involved, as it would require a tighter integration between the module loader and Gillian's symbolic execution engine.

Format of persisted results. When implementing Gillian's incremental mode, we have focused on achieving correct functionality first. In order to allow for easier debugging, we have used JSON as the format for storing most of the intermediate files that get produced in the `.gillian` directory, including the call graph, source files table, and derived function specifications (in bi-abduction). However, JSON is certainly not the most suitable format, since the parsing time would not scale well as the size of the codebase (and hence of the intermediate files) grows larger. In this regard, there is plenty more work to be done to investigate faster storage formats and benchmark their serialisation/deserialisation times. A natural starting point would be to look into binary data formats, such as the one provided by the `biniou`[3] library in OCaml.

File-based incremental analysis. The main limitation of our incremental analysis work is the fact that our change-tracking mechanism works at the granularity of changed files. In particular, it cannot establish *which* of the functions within a file have changed, or, in verification, whether *both* the specification and body of a function have been changed, or only one of them. These result in the starting set of changed procedures from which we compute the transitive dependants being larger than it should be, which means that Gillian performs more re-analysis work than it really needs to. Pushing down the granularity to changed procedures (or even changed lines) will only affect the size of this set—the incremental mechanism itself will not require changing.

In addition, it is worth noting that other tools that have successfully applied continuous reasoning to large codebases, such as Infer, also work at the granularity

[3] `https://github.com/ocaml-community/biniou`

of changed files. This is because, in the context of projects spanning hundreds, if not thousands of files, limiting the analysis to only the files that have changed already has a drastic effect on the time taken to provide feedback to the developer.

6.6 Lessons learnt

Working on this project has been quite a different experience to the previous work I have done as part of my undergraduate studies and during my internships. For those projects, the tasks were less open-ended, smaller in scope, and generally isolated from the work being done by others within the same team. In contrast, Gillian has a significantly large and mature codebase, since it is the amalgamation of several years of related research work done by the Verified Software group. It is also under active development by a number of different people working concurrently. There are a number of key lessons I have learnt along the way.

The importance of good design and simplicity. This is a lesson that frequently gets drilled into Computing students throughout their degree, particularly during the second-year Software Engineering course but also as early as the first-year Haskell lectures. However, it hard to grasp just *how* important maintaining good code hygiene is until you are required to work on a large, complex codebase whose original developers may no longer be within easy reach to explain their rationale. Choosing descriptive but succinct identifier names, separating concerns between different functions and modules, and eliminating duplication can go a long way to limit the accumulation of technical debt and ensure that the codebase remains maintainable for other developers in the future.

The importance of adequate regression testing. The main Gillian repository has a continuous integration pipeline running that checks the code continues to be buildable, and that it passes certain tests, as it evolves. This has been vital in ensuring that the changes made by the four of us (an MEng student, two researchers, and myself) working on Gillian at around the same time did not clash. However, while our CI setup detects regressions in functionality, it does not currently check for regressions in performance.

At one point, I made a change that unknowingly induced a significant overhead in the memory consumption of Gillian-JS during the compilation process to GIL. This led to our Test262 test suite (used to check for compliance with the ECMAScript standard, and comprising around 9000 tests) take almost 10x longer to run. However, this went unnoticed for almost two weeks, and by that time, other changes have been to the code which made the issue harder to debug. Learning from this, I will pay much closer attention to ensuring the projects I am working on have adequate performance regression testing in place.

Chapter 7

Conclusions and Future Work

In this project, we have successfully extended Gillian, a state-of-the-art program analysis framework, to have the foundations necessary for it to be integrated into modern development workflows and therefore reach greater adoption. In particular, we have enabled its C and JavaScript instantiations to analyse multi-file projects by incorporating the import and export mechanics specific to each of those languages. We have demonstrated the correctness of our implementation, and how much more approachable the tools have become as a result, by using them to analyse two real-world projects. In addition, we have developed incremental techniques for each of Gillian's three analysis modes that allow them to focus their analysis on fragments of code in code changes, rather than whole programs, therefore paving the way for them to perform continuous reasoning as outlined by O'Hearn in [37].

Our work has led us to make a number of important observations. It is undoubtedly easier for a tool developer to instantiate Gillian for a particular target language as opposed to building an equivalent analysis tool from scratch. This is in no small due to its built-in reasoning capabilities and symbolic execution engine. However, even if we discount the implementation of the memory model, capturing the full mechanics of that target language is by no means an easy feat for the tool developer. From our experience, even implementing the target language's module system (if it has one) will require the tool developer to spend time understanding its intricacies, ensure these are faithfully reproduced in the translation process to GIL, and consider the trade-offs that need to be made given Gillian's capabilities.

In addition, if the tool developer determines that the core Gillian framework itself is too limiting to support the mechanics of that target language, they will likely consider changing it to better adapt their needs, as we have done at various points in our work. Doing this, all the while ensuring that the framework remains *language-independent*, will always present some challenges.

7.1 Future work

Although we believe we have made good progress with our project, we acknowledge that our work is not yet complete. There is plenty of scope for improvement, both by addressing the limitations we outlined in § 6.5 and by extending the project in

new, interesting directions. We discuss some of these below.

Procedure-based incremental analysis. Our current incremental mechanism within Gillian only tracks changes at the granularity of changed files. Pushing this granularity down, in order to determine *which* of the exact functions within a file have changed, would further reduce the amount of analysis work that Gillian must do between successive runs. We believe this extension should not be too difficult to implement. By persisting the compiled GIL files across multiple runs of Gillian, we could use a similar hash-based approach to determine which parts of the GIL AST corresponding each file have changed.

Support for ES6+ imports and exports in Gillian-JS. As of the sixth edition of the ECMAScript standard, JavaScript has its own native support for modules. Unlike CommonJS modules, these are resolved statically and *asynchronously*, making them suitable for use within browser environments—as of writing, almost all major browsers support them [45]. Therefore, integrating support for them within Gillian-JS's analyses would undoubtedly allow us to target a whole-f real-world, frontend JavaScript applications.

There are a number of approaches we could take. We could compile any `import` and `export` statements within the program to their Node-style equivalents, which Gillian-JS already supports. This, in fact, is the approach taken by JavaScript transpilers such as Babel,[1] which allow code written using the more recent constructs in the language to be compiled to a version that can be supported by older browsers as well. For example, a named `import` statement could be translated as follows:

```
// ES6+ syntax:
import { a, b } from "./foo";

// Becomes:
var _tmp = require("./foo");
var a = _tmp.a;
var b = _tmp.b;
```

This, however, would not preserve the full runtime semantics of ES6+ modules. Therefore, a better approach would be to use the official ECMAScript standard to guide the implementation, as Gillian-JS has done for implementing other internal JavaScript mechanics.

Integration within common build systems. Many of the static analysis tools that have reached widespread adoption in industry, such as Infer and Coverity [5], owe (at least some) of their success to the ease with which they can be invoked on an existing project. Both of them allow the developer to start using the tool by simply prepending the tool's run command to the command they would normally issue to build their project. For example, to analyse a C project that uses the `Make` build system with Infer, a user can run [43]:

```
infer run -- make
```

[1] `https://babeljs.io/`

Infer then uses the compilation commands that get issued by the build system (such as calls to `gcc`) in order to work out the structure of the project and determine the source files it needs to analyse. Adding a similar capability to Gillian-C and Gillian-JS would greatly improve their ease-of-adoption, since it would allow them to automatically determine the source files they need to compile to GIL.

Bibliography

[1] Ball, T., Bounimova, E., Cook, B., Levin, V., Lichtenberg, J., McGarvey, C., Ondrusek, B., Rajamani, S. K., and Ustuner, A. Thorough Static Analysis of Device Drivers. In *EuroSys '06: Proceedings of the 1st ACM SIGOPS/EuroSys European Conference on Computer Systems 2006, Leuven, Belgium, April 18–21, 2006* (2006), ACM, pp. 73–85.

[2] BBC News. Microsoft Zune affected by 'bug', 2008. Available at: `http://news.bbc.co.uk/1/hi/technology/7806683.stm` [Accessed 9th June 2020].

[3] Berdine, J., Calcagno, C., and O'Hearn, P. W. Symbolic Execution with Separation Logic. In *Programming Languages and Systems: Third Asian Symposium, APLAS 2005, Tsukuba, Japan, November 2–5, 2005, Proceedings* (2005), K. Yi, Ed., vol. 3780 of *Lecture Notes in Computer Science*, Springer, pp. 52–68.

[4] Berdine, J., Calcagno, C., and O'Hearn, P. W. Smallfoot: Modular Automatic Assertion Checking with Separation Logic. In *Formal Methods for Components and Objects: 4th International Symposium, FMCO 2005, Amsterdam, The Netherlands, November 1–4, 2005, Revised Lectures* (2006), F. S. de Boer, M. M. Bonsangue, S. Graf, and W. P. de Roever, Eds., vol. 4111 of *Lecture Notes in Computer Science*, Springer, pp. 115–137.

[5] Bessey, A., Block, K., Chelf, B., Chou, A., Fulton, B., Hallem, S., Henri-Gros, C., Kamsky, A., McPeak, S., and Engler, D. A Few Billion Lines of Code Later: Using Static Analysis to Find Bugs in the Real World. *Communications of the ACM 53*, 2 (2010), 66–75.

[6] Boonstoppel, P., Cadar, C., and Engler, D. RWset: Attacking Path Explosion in Constraint-Based Test Generation. In *Tools and Algorithms for the Construction and Analysis of Systems: 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29–April 6, 2008, Proceedings* (2008), C. R. Ramakrishnan and J. Rehof, Eds., vol. 4963 of *Lecture Notes in Computer Science*, Springer, pp. 351–366.

[7] Bryant, R. E., and O'Hallaron, D. R. *Computer Systems: A Programmer's Perspective*, 2 ed. Prentice Hall, Boston, MA, 2011, ch. 7.

[8] Cadar, C. Software Reliability, 2020. Lecture notes, Imperial College London.

[9] Cadar, C., Godefroid, P., Khurshid, S., Păsăreanu, C. S., Sen, K., Tillmann, N., and Visser, W. Symbolic Execution for Software Testing in Practice: Preliminary Assessment. In *ICSE '11: Proceedings of the 33rd International Conference on Software Engineering, ICSE 2011, Waikiki, Honolulu, HI, USA, May 21–28, 2011* (2011), ACM, pp. 1066–1071.

[10] Calcagno, C., and Distefano, D. Infer: An Automatic Program Verifier for Memory Safety of C Programs. In *NASA Formal Methods: Third International Symposium, NFM 2011, Pasadena, CA, USA, April 18–20, 2011, Proceedings* (2011), M. Bobaru, K. Havelund, G. J. Holzmann, and R. Joshi, Eds., vol. 6617 of *Lecture Notes in Computer Science*, Springer, pp. 459–465.

[11] Calcagno, C., Distefano, D., Dubreil, J., Gabi, D., Hooimeijer, P., Luca, M., O'Hearn, P., Papakonstantinou, I., Purbrick, J., and Rodriguez, D. Moving Fast with Software Verification. In *NASA Formal Methods: 7th International Symposium, NFM 2015, Pasadena, CA, USA, April 27–29, 2015, Proceedings* (2015), K. Havelund, G. Holzmann, and R. Joshi, Eds., vol. 9058 of *Lecture Notes in Computer Science*, Springer, pp. 3–11.

[12] Calcagno, C., Distefano, D., O'Hearn, P., and Yang, H. Space Invading Systems Code. In *Logic-Based Program Synthesis and Transformation: 18th International Symposium, LOPSTR 2008, Valencia, Spain, July 17–18, 2008, Revised Selected Papers* (2009), M. Hanus, Ed., vol. 5438 of *Lecture Notes in Computer Science*, Springer, pp. 1–3.

[13] Calcagno, C., Distefano, D., O'Hearn, P., and Yang, H. Compositional Shape Analysis by Means of Bi-Abduction. In *POPL '09: Proceedings of the 36th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2009, Savannah, GA, USA, January 21–23, 2009* (2009), Z. Shao and B. C. Pierce, Eds., ACM, pp. 289–300.

[14] Clark, L. ES modules: A cartoon deep-dive, 2018. Available at: `https://hacks.mozilla.org/2018/03/es-modules-a-cartoon-deep-dive/` [Accessed 10th June 2020].

[15] Constine, J. Facebook Acquires Assets Of UK Mobile Bug-Checking Software Developer Monoidics, 2013. Available at: `https://techcrunch.com/2013/07/18/facebook-monoidics/` [Accessed 19th January 2020].

[16] Cousot, P., Cousot, R., Feret, J., Mauborgne, L., Miné, A., Monniaux, D., and Rival, X. The ASTRÉE Analyzer. In *Programming Languages and Systems: 14th European Symposium on Programming, ESOP 2005, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2005, Edinburgh, UK, April 4–8, 2005, Proceedings* (2005), S. Mooly, Ed., vol. 3444 of *Lecture Notes in Computer Science*, Springer, pp. 21–30.

[17] Dijkstra, E. Software Quality. In *Software Engineering Techniques: Report on a conference sponsored by the NATO Science Committee, Rome, Italy, October 27–31, 1969* (1970), J. N. Buxton and B. Randell, Eds., NATO Science Committee, p. 16.

[18] Fairley, R. E. Tutorial: Static Analysis and Dynamic Testing of Computer Software. *Computer 11*, 4 (1978), 14–23.

[19] Feitelson, D. G., Frachtenberg, E., and Beck, K. L. Development and Deployment at Facebook. *IEEE Internet Computing 17*, 4 (2013), 8–17.

[20] Fowler, M. Continuous Integration, 2006. Available at: `https://martinfowler.com/articles/continuousIntegration.html` [Accessed 22nd January 2020].

[21] Fragoso Santos, J., Maksimović, P., Ayoun, S. É., and Gardner, P. Gillian: Compositional Symbolic Execution for All, 2020. `arXiv:2001.05059 [cs.PL]`.

[22] Fragoso Santos, J., Maksimović, P., Ayoun, S. É., and Gardner, P. Gillian, Part I: A Multi-Language Platform for Symbolic Execution. In *PLDI, London, UK, June 15–19, 2019* (2020).

[23] Fragoso Santos, J., Maksimović, P., Naudziuniene, D., Wood, T., and Gardner, P. JaVerT: JavaScript Verification Toolchain. *Proceedings of the ACM on Programming Languages 2*, POPL (2018), 50:1–50:33.

[24] Fragoso Santos, J., Maksimović, P., Sampaio, G., and Gardner, P. JaVerT 2.0: Compositional Symbolic Execution for JavaScript. *Proceedings of the ACM on Programming Languages 3*, POPL (2019), 66:1–66:31.

[25] Gardner, P. Gillian: A General Static Analysis Framework based on Separation Logic. Presentation, ECOOP 2019 Summer School, 18th July 2019.

[26] Gardner, P. Separation Logic: Scalable Reasoning about Programs, 2019. Lecture notes, Imperial College London.

[27] Gordon, M. Background reading on Hoare Logic, 2016. Available at: `https://www.cl.cam.ac.uk/archive/mjcg/HL/Notes/Notes.pdf` [Accessed 11th January 2020].

[28] Harteshorne, C., and Weiss, P., Eds. *Collected Papers of Charles Sanders Peirce*. Harvard University Press, Cambridge, MA, 1958.

[29] Hoare, C. A. R. An Axiomatic Basis for Computer Programming. *Communications of the ACM 12*, 10 (1969), 576–580.

[30] Jacobs, B., Smans, J., Philippaerts, P., Vogels, F., Penninckx, W., and Piessens, F. VeriFast: A Powerful, Sound, Predictable, Fast Verifier for C and Java. In *NASA Formal Methods: Third International Symposium, NFM 2011, Pasadena, CA, USA, April 18–20, 2011, Proceedings* (2011), M. Bobaru, K. Havelund, G. J. Holzmann, and R. Joshi, Eds., vol. 6617 of *Lecture Notes in Computer Science*, Springer, pp. 41–55.

[31] Jacobs, B., Smans, J., and Piessens, F. A Quick Tour of the VeriFast Program Verifier. In *Programming Languages and Systems: 8th Asian Symposium, APLAS 2010, Shanghai, China, November 28–December 1, 2010, Proceedings* (2010), K. Ueda, Ed., vol. 6461 of *Lecture Notes in Computer Science*, Springer, pp. 304–311.

[32] King, J. C. Symbolic Execution and Program Testing. *Communications of the ACM 19*, 7 (1976), 385–394.

[33] Leroy, X. The CompCert verified compiler: Documentation and user's manual, 2020. Available at: `http://compcert.inria.fr/man/manual.pdf` [Accessed 9th June 2020].

[34] Leveson, N. G., and Turner, C. S. An Investigation of the Therac-25 Accidents. *IEEE Computer 26*, 7 (1993), 18–41.

[35] O'Hearn, P., Reynolds, J., and Yang, H. Local Reasoning about Programs that Alter Data Structures. In *Computer Science Logic: 15th International Workshop, CSL 2001, 10th Annual Conference of the EACSL, Paris, France, September 10—13, 2001, Proceedings* (2001), L. Fribourg, Ed., vol. 2142 of *Lecture Notes in Computer Science*, Springer, pp. 1–19.

[36] Osmani, A. Learning JavaScript Design Patterns, Volume 1.7.0, 2017. Available at: `https://addyosmani.com/resources/essentialjsdesignpatterns/book/#modulepatternjavascript` [Accessed 10th June 2020].

[37] O'Hearn, P. W. Continuous Reasoning: Scaling the impact of formal methods. In *LICS '18: Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science, Oxford, UK, July 9–12, 2018* (2018), ACM, pp. 13–25.

[38] Rustan, K., Leino, M., and Moskal, M. Usable Auto-Active Verification. In *Workshop on Usable Verification, Microsoft Research, Redmond, WA, USA, November 15–16, 2010* (2010).

[39] TC39, E. Ecma-262 Edition 5.1, The ECMAScript Language Specification, 2011. Available at: `https://www.ecma-international.org/ecma-262/5.1/#sec-10.2` [Accessed 10th June 2020].

[40] The CommonJS Team. Modules/1.1.1, 2014. Available at: `http://wiki.commonjs.org/wiki/Modules/1.1.1` [Accessed 10th June 2020].

[41] The Infer Team. Separation logic and bi-abduction, 2016. Available at: `https://fbinfer.com/docs/separation-logic-and-bi-abduction/` [Accessed 10th May 2020].

[42] The Infer Team. About Infer, 2017. Available at: `https://fbinfer.com/docs/about-Infer` [Accessed 10th May 2020].

[43] The Infer Team. Hello, World!, 2020. Available at: `https://fbinfer.com/docs/hello-world` [Accessed 10th June 2020].

[44] The Mozilla Team. IIFE, 2020. Available at: `https://developer.mozilla.org/en-US/docs/Glossary/IIFE` [Accessed 10th June 2020].

[45] The Mozilla Team. JavaScript modules, 2020. Available at: `https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Modules` [Accessed 10th June 2020].

[46] The Mozilla Team. Parser API, 2020. Available at: `https://developer.mozilla.org/en-US/docs/Mozilla/Projects/SpiderMonkey/Parser_API` [Accessed 10th June 2020].

[47] The Node Team. Node.js v14.4.0 Documentation, 2020. Available at: `https://nodejs.org/api/modules.html` [Accessed 10th June 2020].

[48] Yang, H., Lee, O., Berdine, J., Calcagno, C., Cook, B., Distefano, D., and O'Hearn, P. Scalable Shape Analysis for Systems Code. In *Computer Aided Verification: 20th International Conference, CAV 2008 Princeton, NJ, USA, July 7–14, 2008, Proceedings* (2008), A. Gupta and S. Malik, Eds., vol. 5123 of *Lecture Notes in Computer Science*, Springer, pp. 385–398.