IMPERIAL

Adding and Optimising Resource Algebras for a Parametric CSE

Background & Progress Report

Author: Opale Sjöstedt

Supervisor: Philippa Gardner

August 24, 2024

Submitted in partial fulfilment of the requirements for the MSc Degree in Computing (Software Engineering)

Contents

Chapter 1

Introduction

As software becomes part of critical element, it needs to be verified formally to ensure it does not fault. To support such verification in a scalable fashion, *separation logic* [1], [2] was created, permitting compositional proofs on the behaviour of programs. Research has also been done in the field of separation algebra, to provide an abstraction over the state modelled within separation logic, to improve modularity and reduce the effort needed for proofs.

Thanks to separation logic, compositional verification via specifications has been automated, and allows automatic checking of properties via compositional symbolic execution (CSE). A wide range of tools exist, usually enabling the verification of a specific language they are tailored for. Gillian [3]–[5] is a CSE engine that is different, in that it is not made for one specific language but is instead *parametric on the state model*, allowing for it to be used to verify different language, like C, JavaScript or Rust.

Gillian has now been in development for several years, and the research landscape surrounding it has evolved in parallel, with for instance new techniques for state modelling and the creation of incorrectness separation logic.

In this report we will present Aether, a CSE engine that follows the steps of Gillian and that is both parametric in the state model and in the language, while being closely related to the theory underpinning it. It's goal is to serve as a simple, efficient and complete engine that supports multiple analysis methods such as OX verification and UX true bug finding, and that is easily extendable for different uses.

Chapter 2

Literature Review

2.1 Separation Logic

As the need to formally verify programs grew, methods needed to be built to provide a framework to do so. Hoare Logic [6] is a logic made specifically to allow proving properties of programs, by describing them axiomatically. Every statement can be expressed as having a precondition – the state before execution – and a postcondition – the state after *successful* execution – expressed as $\{P\}\, S\, \{Q\}$. For assignment for instance, one could have $\{\mathsf{x} = x\}\, \mathsf{x} := 0\, \{\mathsf{x} = 0\}$.

While already extremely helpful to reason about programs, an issue remains however, and it is to do with shared mutable state, like memory of a program – how does one describe that there is a list in memory of unknown length, that may be mutated at multiple places in the program, while ensuring properties hold at a specific point in time? Being able to describe the state and constraints upheld by global state is difficult, and past solutions scaled poorly.

Separation Logic (SL) [1], [2] is an extension of Hoare logic that permits this in a clear and scalable way. It's main addition is the separating conjunction $*$: $P * Q$ means not only that the heap satisfies P and Q, but that it can be split into two *disjoint* parts, such that one satisfies P and the other Q. This allows us to reason compositionally about the state, by not only stating what properties are upheld, but how they may be split for further proofs. For instance, given a *list* predicate, when calling a function that mutates the list one can simply substitute the part of the state corresponding to that list with the postcondition of the function, with the guarantee that the rest of the state is untouched, by using the *frame rule*:

$$\textsc{Frame}\quad \frac{\{P\}\, S\, \{Q\}}{\{P * F\}\, S\, \{Q * F\}}$$

Because we know that S only uses (either by reading or modifying) P, any disjoint frame F can be added to the state without altering the execution of S. This is very powerful: one can prove properties of smaller parts of code (like a function, or a loop), and these properties will be able to be carried to a different context that may have a more complex state. For instance, this can be used in a Continuous Integration (CI) setting, by only analysing functions whose code is modified, while reusing past analysis of unchanged code, allowing for incremental analysis.

SL also comes equipped with the *emp* predicate, representing an empty state (ie. $P * emp = P$), and the separating implication $-\!\!*$ (also called wand). $P_0 -\!\!* P_1$ states that if the current state is extended with a disjoint part satisfying P_0, then P_1 holds in the extended heap [2].

Further predicates can then be defined; for instance the "points to" predicate, $a \mapsto x$, stating that the address a stores values x. We may note that $a \mapsto x * a \mapsto x$ does not hold, since a heap with a pointing to x cannot be split into two disjoint heaps both satisfying $a \mapsto x$. Similarly, $a \mapsto x * b \mapsto y$ can only hold if $a \neq b$.

2.1.1 Incorrectness Separation Logic

Separation Logic is, by definition, *over-approximate* (OX): $\{P\}\, S\, \{Q\}$ means that given a precondition P, we are guaranteed to reach a state that satisfies Q. In other words, Q may encompass *more* than only the states reachable from P. This can become an issue when used for real-world code, where errors that don't actually exist may be flagged as such, hindering the use of a bug detection tool (for instance in a CI setting).

To solve this problem, a recent innovation in the field is Incorrectness Separation Logic (ISL) [7], derived from Incorrectness Logic [8]. It is similar to SL, but *under-approximate* (UX), instead ensuring that the detected bugs actually exist.

Where SL uses triplets of the form $\{\text{precondition}\}\, S\, \{\text{postcondition}\}$, incorrectness separation logic uses $[\text{presumption}]\, S\, [\text{result}]$, with the result being an under-approximation of the actual result of the code. The reasoning is thus flipped, where we start from a stronger assertion at the end of the function, and then step back and broaden our assumptions, until reaching the initial presumption. This means that all paths explored this way are guaranteed to exist, but may not encompass all possible paths.

Another way of comparing SL to ISL is with consequence: in SL, the precondition implies

the postcondition, whereas with ISL the result implies the precondition. This enables us to do *true bug-finding*. Furthermore, ISL triplets are extended with an *outcome* ϵ, resulting in $[P]\, S\, [\epsilon : Q]$, where ϵ is the outcome of the function, for instance *ok*, or an error.

While SL is over-approximate and ISL is under-approximate, we may call *exact* (EX) specifications that are both OX-sound and UX-sound [9]. Such triplets are then written $\langle\!\langle P \rangle\!\rangle\, S\, \langle\!\langle o : Q \rangle\!\rangle$, with o the outcome, and P and Q the pre and postcondition respectively.

2.1.2 Separation Algebras

While the above examples of separation logic use a simple *abstract heap*, mapping locations to values, this is not sufficient to model most real models used by programming languages. For instance, the model used by the C language (in particular CompCert C [10], a verified C compiler) uses the notion of memory blocks, and offsets: memory isn't just an assortment of different cells. Furthermore, the basic "points to" predicate, while useful, has limited applications; for contexts such as in concurrency, one needs to have a more precise level of sharing that goes beyond the *exclusive ownership* of "points to".

An example of such extension is fractional permissions [11], [12]: the "points to" predicate is extended with a *permission*, a fraction q in the $(0; 1]$ range, written $a \underset{q}{\mapsto} x$. A permission of 1 gives read and write permission, while anything lower only gives read permission. This allows one to split permissions, for instance $a \underset{1}{\mapsto} x$ is equivalent to $a \underset{0.5}{\longmapsto} x * a \underset{0.5}{\longmapsto}$, a program can thus concurrently execute two routines that read the same part of the state, while remaining sound – permissions can then be re-added as the routines exit, regaining full permissions over the cell.

Further changes and improvements to separation logic have been made, usually with the aim of adapting a particular language feature or mechanism that couldn't be expressed otherwise. "The Next 700 Separation Logics" argues that indeed too many subtly different separation logics are being created to accommodate specific challenges, which in turn require new soundness proofs that aren't compatible with each other. To tackle this, research has emerged around the idea of providing one sound metatheory, that would provide the tools necessary to building more complex abstractions *within* that logic, rather than in parallel to it.

A first step in this quest towards a single core logic is the definition of an abstraction over the state. The main concept introduced for this is that of *Separation Algebras* [14], [15], which allow for the creation of complex state models from simpler elements. Because soundness is proven for these smaller elements, constructions made using them also carry this soundness, and alleviate users of complicated proofs.

Different authors used different definitions and axioms to define separation algebras.

In [14] they are initially defined as a cancellative partially commutative monoid (PCM) $(\Sigma, \bullet, 0)$. This definition is however too strong: for instance, the cancellativity property implies $\sigma \bullet \sigma' = \sigma \implies \sigma' = 0$. While this is true for some cases such as the points to predicate, this would invalidate useful constructions such as that of an *agreement*, where knowledge can be duplicated. For an agreement separation algebra, we thus have $\sigma \bullet \sigma = \sigma$, which of course dissatisfies cancellativity.

This approach is also taken in [15], where separation algebras are defined with a functional ternary relation $J(x, y, z)$ written $x \oplus y = z$ – the choice of using a relation rather than a partial function being due to the fact the authors wanted to construct their models in Rocq, which only supports computable total functions. While the distinction is mostly syntactical and both relational and functional approaches are equivalent, the former makes proof-work less practical [16], [17], whereas the functional approach allows one to reason equationally. This paper also introduces the notion of *multi-unit separation algebras*, separation algebras that don't have a single 0 unit, but rather enforce that every state has *a* unit, that can be different from other states' units. Formally, this means that instead of having $\exists u.\ \forall x.\ x \oplus u = x$, we have $\forall x.\ \exists u_x.\ x \oplus u_x = x$ – this allows one to properly define the disjoint union (or sum) of separation algebras, with both sides of the sum having a distinct unit. We may note that this is a first distinction from PCMs, as a partially commutative monoid cannot have two distinct units; according to the above definition, separation algebras are thus a type of partially commutative semigroup.

In [16], further improvements are done to the axiomatisation of separation algebras. Most notably, the property of cancellativity is removed, as it is too strong and unpractical for some cases, as shown previously. Furthermore, the idea of unit, or *core*, is made explicit, with the definition of the total function $\hat{\cdot}$, the core of a resource, which is its *duplicable part*.

Finally, Iris [17] is a state of the art "higher-order concurrent separation logic". It places itself as a solution to the problem mentioned in [13], and aims to provide a sound base logic that can be reused and extended to fit all needs. Its *resource algebras* (RAs), similar to separation algebras. Because Iris supports "higher-order ghost state", they need more sophisticated mechanisms to model state, and as such their composition function $\cdot$ needs to be total. To then rule out invalid compositions, they instead add a validity function $\overline{\mathcal{V}}$, which returns whether a given state is valid. They also keep the unit function, written $|\cdot|$, that they make *partial*, rather than total – this, they argue, makes constructing state models from smaller components easier[1]. This was confirmed when developing state models for Gillian, where having the core be a total function made some state models more

[1]The author of this report found it surprising that they removed partiality from composition to then re-add it via the core, as this lacks consistency

complicated to make sound.

2.2 Program Verification with Gillian

While having a logic to prove programs is a great step towards verifying codebases, doing it manually is tedious and time-consuming. Tools have been developed to automate this, and *Gillian* [3]–[5], the main inspiration of this project, is an example of such tool. It is a *compositional symbolic execution engine*, with the added property of being *parametric on the memory model*, and that allows reasoning about *correctness and incorrectness*. We may now go over exactly what this means.

2.2.1 Symbolic Execution

Traditionally, software is tested by calling the code with a predetermined or randomly generated input (for instance with fuzzing), and verifying that the output is as expected. These approaches, where the code is executed "directly", are of the realm of *concrete* execution, as the code is run with concrete – as in real, existing – values. While this method is straightforward to execute, it comes with flaws: it is limited by the imagination of the person responsible for writing the tests (when written manually), or by the probability of a given input to reach a specific part of the codebase. With fuzzing, methods exist to improve the odds of finding new paths [18], but it all amounts to luck nevertheless.

A solution to this is symbolic execution: rather than running the code with concrete values, *symbolic* values are used [19]. These values are abstract, and are then restricted as an interpreter steps through the code and conditionals are encountered. Once a branch of the code terminates, a constraint solver can be used against the accumulated constraints to obtain a possible concrete value, called an *interpretation*. This method appeared in the 70s, notably with the Select [20] and EFFIGY [21] systems.

For a simple C program that checks if a given number is positive or negative and even or odd (see Figure 2.1), symbolic execution would thus branch thrice, once at each condition[2]. This would then result in 5 different branches, each with different constraints on the program variable x: $\mathsf{x} = 0$, $\mathsf{x}\%2 = 0 \land \mathsf{x} < 0$, $\mathsf{x}\%2 = 0 \land \mathsf{x} > 0$, $\mathsf{x}\%2 \neq 0 \land \mathsf{x} < 0$ and $\mathsf{x}\%2 \neq 0 \land \mathsf{x} > 0$[3].

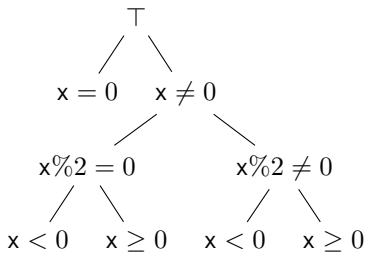
2.2.2 Compositional Symbolic Execution

An issue that arises from symbolic execution is that is scales quite poorly, because of path explosion [19], [22], as each branch can potentially multiply by two the total number of

[2]For simplicity, we omit the case where x is not an integer, which would lead to an error.
[3]The four last constraints also contain $\mathsf{x} = 0$, which is included in $\mathsf{x} > 0$ and $\mathsf{x} < 0$

```
if (x == 0) {
  print("zero");
  return;
}
if (x % 2 == 0) {
  print("even");
} else {
  print("odd");
}
if (x < 0) {
  print("negative");
} else {
  print("positive");
}
```

(a) Simple branching program



(b) Paths obtained from symbolically executing the code, with the added constraint at each node

Figure 2.1: Symbolic execution of a simple program

branches. Furthermore, it lacks compositionality, as the code must be tested in its entirety.

A solution to this is *compositional* symbolic execution (CSE), in which the code is tested compositionally, function by function. This allows for gradual adoption, as only parts of the code base can be tested, all while minimising the effect of path explosion.

A great tool to enable this is separation logic: one can specify the precondition and postcondition of a function, and the symbolic execution can then step through the code, by starting from a state satisfying the precondition and ensuring that once the branch terminates the postcondition is satisfied. Because of the frame rule, function specifications can then be re-used when calling functions, by symbolically executing the function specification on the corresponding state fragment. If the called function does not have a specification that can be used, it can instead be inlined, executing the code within the function, as would standard symbolic execution.

This approach was pioneered in Smallfoot [23], and was followed by later tools, such as JaVerT [24], [25], Infer [26], VeriFast [27], SpaceInvaders [28] and others – their individual contributions will be discussed in more detail in the next section.

2.2.3 Parametricity

An innovation of Gillian compared to other existing tools is the fact is is parametric on the memory model. Typical CSE engines usually come with a built-in notion of memory, ranging from simple heaps with memory chunks [27] to more complex state models tailored to a language, like in JaVerT [24], [25] to represent JavaScript objects.

While a fixed memory model is useful for specific languages or feature sets, this can become a limitation when porting the engine to other languages. The goal behind Gillian's parametricity is to enable any state model to be represented, enabling it to be used for virtually any language (of course as long as the language's memory model can be encoded

into a separation-logic equivalent). This enables it to verify programs written in JavaScript, C or Rust, despite their significant differences in terms of state representation and level of abstraction.

To define a memory model to use in Gillian, a developer must implement it in OCaml, defining a specific interface. A memory model is a partially commutative monoid, comprised of a set of states, which can be modified via *actions* and be represented in specifications via *core predicates*.

Actions serve as the interface for programs to interact with the state, for instance by storing or loading values into the state, or allocating a new block of memory and then freeing it. As such, while the language Gillian targets, GIL, provides the base semantics of an imperative goto-language (variables, errors, functions...) which do not affect the state – actions are the only way of manipulating it.

Core predicate are predicates which have in and out parameters, and represent fragments of the state. They are the encoding of the state into separation logic, and can be used within function specifications. For example, the separation logic $a \mapsto x$ predicate could be represented as a core predicate as $\langle points\_to \rangle(\mathsf{a}; \mathsf{x})$.

As mentioned in subsection 2.1.2, literature already exists to enable sound modelling of state, and Iris [17] is an example of the state of the art in this field, with a rich theory and existing automation in Rocq. However, this had never been ported to automatic program verification, instead remaining in the meta-theory.

2.2.4 Correctness, Incorrectness

Another key difference between Gillian and other CSE engines is that is allows reasoning about programs both with correctness, in separation logic [1], [2] and incorrectness separation logic [7], thus allowing OX program verification and UX true bug-finding respectively. It also supports whole-program testing, as a non-compositional engine.

This allows one to instantiate Gillian to their preferred target language, and automatically benefit from all three modes of symbolic execution. This is in part possible thanks to the exact (EX) [9] semantics of GIL, enabling automated proofs to be done both with left-to-right and right-to-left implications, representing OX and UX respectively.

Thanks to this and the aforementioned parametricity of the memory model, Gillian is thus a unified CSE engine, that allows scalable verification of software in a unified setting, without requiring one to adapt to two different engines, which may be prohibitive and extremely time-consuming.

2.3 Existing Tools

A wide range of engines exist to verify the correctness of code, most targeted towards a specific language. This allows them to implement behaviour that is appropriate for that particular language. For instance, CBMC [29] is a bounded model checker, that allows for the verification of C programs, via whole program testing – that is, it is not compositional, and instead symbolically executes a given codebase, and verifies that properties hold. While clearly useful, as shown by its success, this solution doesn't scale particularly well; larger applications need to be split modularly to be verified, and manual stubbing is required to support testing parts of the code separately.

Slightly separate from symbolic execution, KLEE [30] allows for the *concolic* execution of LLVM code – thus supporting C, C++, Rust, and any other language that can be compiled to LLVM. Concolic execution is a hybrid of symbolic and concrete execution, allowing the use of symbolic values along a concrete execution path. Similarly to CBMC, it handles whole program testing, and as such doesn't scale with larger codebases. MACKE [31] is an adaptation of KLEE enabling compositional testing, proving that concolic execution can be extended to be more scalable, without losing accuracy in reported errors.

As codebases grow, there comes a need for compositional tools that can verify smaller parts of the code, thus enable greater scaling and integration in continuous integration (CI) pipelines, which permit giving rapid feedback to developers. Separation logic enables this, by allowing one to reason about a specific function and ignoring the rest of the program's state.

While Smallfoot [23] was the first tool to enable compositional symbolic execution, jStar [32] was the first to allow automated verification of a real-world language, Java. It is in particular tailored towards handling design patterns that are commonly found in object-oriented programming and that may be hard to reason about. It uses an intermediate representation of Java called Jimple, taken from the Java optimisation framework Soot. Similarly, VeriFast [27] is a CSE tool that is targeted at Java and a subset of C, while SpaceInvader [28] allows for the verification of C code in device drivers.

Infer [26] is a tool developed by Meta that uses separation logic and bi-abduction to find bugs in C, C++, Java and Objective-C programs, by attempting to prove correctness and extracting bugs from failures in the proof. It is the tool that pioneered *bi-abduction*, a technique enabling execution to proceed despite resources missing, via the construction of an anti-frame. It's approach, while initially stemming from separation logic, was used for finding bugs rather than proving correctness, which led to the creation of ISL [7] to provide a sound theory justifying this approach. Following this, Pulse [33] followed and

fully exploited the advances made in ISL to show that this new theory served as more than a justification to past tools, and to prove the existing of true bugs.

Also using separation logic, JaVerT [24], [25] is a CSE engine aimed to verifying JavaScript, with support for whole program testing, verification, and bi-abduction in the same fashion as Infer – it followed from Cosette [34], which already supported whole program testing.

While the above examples are targeted at specific target languages, some tools have also been designed with modularity in mind. To do such, they instead support a general intermediate language – one that isn't designed specifically for a language, like Jimple for Java – that a target language must be compiled to. This allows for greater flexibility in regards to what languages can be analysed by the engine, as all it needs to be capable of verifying a new language is a compiler.

Another example is Viper [35], an infrastructure capable of program verification with separation logic, with existing frontends that allow for the verification of Scala, Java and OpenCL. It's intermediate language is a rich object-oriented imperative typed language, that operates on a built-in heap.

One can still take genericity further, by also abstracting the state model the engine operations on. A tool that does this is coreStar [36], the CSE engine backing jStar with all the Java-related parts removed and replaced. It does not target Java, but instead a generic intermediate language. Furthermore, instead of coming with predicates pre-defined, it requires user-defined predicates to be provided, which are then used throughout the proof. They are defined in a language specified by coreStar, and are thus however quite limited in expressivity. Still, this is to the author's knowledge the first example of a tool that allows some flexibility in the underlying logic theory.

This is also what Gillian [3]–[5] does, a CSE engine parametric on the state model, and the main work upon which this project this project is based. It doesn't support any one language, but instead can target any language that provides a state model implementation in OCaml and a compiler to GIL, it's intermediary language. Currently Gillian has been instantiated to the C language (via CompCert-C [10]) and JavaScript, as well as Rust [37]. It is a generalisation of JaVerT, which only targeted JavaScript.

TODO: Mention social/legal/ethical/professional stuff !!!! eg. that time people died from bug with X-rays, or rockets, or other things... mention examples with Gillian too maybe, for a smaller scale.

	Target Language	Compositional	Parametric State Model	Mode
CBMC	C	✗	✗	WPST
KLEE	LLVM	✗	✗	Concolic
MACKE	LLVM	✓	✗	Concolic
jStar*	Java	✓	✗	OX
VeriFast	C, Java	✓	✗	OX
coreStar*	IR (coreStarIL)	✓	✓	OX
Infer	C, C++, Java, Objective-C	✓	✗	OX
Cosette*	JavaScript	✓	✗	WPST
JaVerT	JavaScript	✓	✗	WPST, OX
Viper	IR (Viper)	✓	✗	OX
Pulse	C, C++, Java, Objective-C	✓	✗	UX
Gillian	IR (GIL)	✓	✓	WPST, OX, UX

WPST = Whole Program Symbolic Testing.

*Unmaintained

Table 2.1: Comparison of verification tools

Chapter 3

Theory

3.1 State of Affairs

3.1.1 State Models with PCMs for CSE

CSE and Sacha's thesis do the traditional choice of using Partial Commutative Monoids (PCMs) to model state. They are defined as the tuple $(M, (\cdot) : M \times M \rightharpoonup M, 0)$. They are further equipped with a set of actions $\mathcal{A}$, an `execute_action` function, a set of core predicates Δ and a pair of `consume` and `produce` functions.

These additions are necessary for the engine to be parametric on the state model, as it provides an interface for interaction with the state.

The usage of PCMs comes with issues: the requirement of a single 0 for each state model means that state models such as the sum state model $\mathbb{S}_1 + \mathbb{S}_2$ come with unwieldy requirements to prove soundness – this comes into play for the `Freeable` state model, that could use a sum (like what is done in [38]) but can't because of this.

3.1.2 Where it goes wrong

3.1.3 RAs for Iris

Iris [17] departs from this tradition and introduces Resource Algebras (RAs) to model state, defined as a tuple $(M, \overline{\mathcal{V}} : M \to \mathbb{B}, |-| : M \to M^?, (\cdot) : M \times M \to M)$, being respectively the state elements, a validity function, a partial core function and a composition function.

This makes Iris states more powerful, in that they have more flexibility in what they can express; for instance sum state models can be easily and soundly expressed, which isn't possible with PCMs due to the requirement of a single 0 element.

Furthermore, Iris RAs comes with plenty proofs and properties making them easy to

use and adapt, whereas PCMs can prove unwieldy even for simpler state models (eg. with the Freeable state model transformer).

A similarity however is that the global RA in Iris must be unital, meaning it must have a single ϵ element, very much as it is the case with the 0 in PCMs. Any RA can be trivially extended to have a unit, which is what Iris defines as the option resource algebra [39].

3.2 State Models with RAs

3.2.1 Partial RAs

A property of Iris RAs is that composition is *total* – to take into account invalid composition, states are usually extended with a $\lightning$ state, such that $\neg\overline{\mathcal{V}}(\lightning)$ (while for states $\sigma \neq \lightning$, $\overline{\mathcal{V}}(\sigma)$ holds). While this is needed in the Iris framework for higher-order ghost state and step-index, this doesn't come into play when only manipulating RAs. As such, because this is quite unwieldy, we can remove it by adding partiality instead, such that invalid ($\lightning$) states simply don't exist and the need for a $\overline{\mathcal{V}}$ function vanishes. This is also inline with the core function $(-)$ being partial.

It is worth noting that *partial* RAs are equivalent to regular RAs, *so long as* $\overline{\mathcal{V}}$ *always holds for valid states*[1]. Indeed, compositions that yield $\lightning$ can be made undefined, and the validity function removed, to gain partiality, and inversely to go back to the Iris definition.

An interesting property of this is that because validity is replaced by the fact composition is defined, the validity of a composition is equivalent to the fact two states are disjoint: $\overline{\mathcal{V}}(a \cdot b) \iff a \# b$.

We now define the properties of RAs taking this change into account – see Figure 3.1. From now, the term RA will be used to refer to these partial RAs.

3.2.2 Core Engine

The core engine enables whole-program symbolic execution. For this state models must firstly define the set of states the execution will happen on; this is done via a partial resource algebra: a tuple $(M, |-| : M \to M^?, (\cdot) : M \times M \rightharpoonup M)$. They are further equipped with a set of actions $\mathcal{A}$, an `execute_action` function and a `sat` relation.

$$\texttt{execute\_action} : \hat{\Sigma}^? \to \mathcal{A} \to \text{Val } \texttt{list} \to \mathcal{P}(\underline{\mathcal{O}_e} \times \hat{\Sigma}^? \times \text{Val } \texttt{list} \times \Pi)$$

$$\texttt{sat} : \Theta \to \text{Store} \to \hat{\Sigma} \to \mathcal{P}(\Sigma)$$

[1] This, to our knowledge, is the case for all of the simpler RAs defined in Iris: Ex, Ag_0, sum, product, etc.

A *resource algebra* (RA) is a triple $(M, |-| : M \to M^?, (\cdot) : M \times M \rightharpoonup M)$

$$\forall a, b, c.\, (a \cdot b) \cdot c = a \cdot (b \cdot c) \qquad \text{(RA-Assoc)}$$
$$\forall a, b.\, a \cdot b = b \cdot a \qquad \text{(RA-Comm)}$$
$$\forall a.\, |a| \in M \Rightarrow |a| \cdot a = a \qquad \text{(RA-Core-ID)}$$
$$\forall a.\, |a| \in M \Rightarrow ||a|| = |a| \qquad \text{(RA-Core-Idem)}$$
$$\forall a, b.\, |a| \in M \land a \preccurlyeq b \Rightarrow |b| \in M \land |a| \preccurlyeq |b| \qquad \text{(RA-Core-Mono)}$$

where $M^? \stackrel{\text{def}}{=} M \uplus \{\bot\}$, with $a \cdot \bot \stackrel{\text{def}}{=} \bot \cdot a \stackrel{\text{def}}{=} a$

$a \preccurlyeq b \stackrel{\text{def}}{=} \exists c.\, b = a \cdot c$

$a \# b \stackrel{\text{def}}{=} a \cdot b$ is defined

A *unital* resource algebra is a resource algebra M with an element $\epsilon \in M$ such that:

$$\forall a \in M. \epsilon \cdot a = a \qquad |\epsilon| = \epsilon$$

Figure 3.1: Definition of Resource Algebras

The arguments of `execute_action` are, in order: the *optional* state the action is executed on, the action, and the received arguments. It returns a set of branches, with an outcome, the new state, the returned values, and the path condition of that branch. It is pretty-printed as $\alpha(\hat{\sigma}, \vec{\hat{v}}_i) \rightsquigarrow (o, \hat{\sigma}', \vec{\hat{v}}_o, \pi)$.

Here, the outcome is an outcome in the set of *full execution outcomes* $\underline{\mathcal{O}}_e = \{\texttt{Ok}, \texttt{Err}\}$. In the next subsection, this set will be extended to account for misses and logical failures, but these do not exist with full semantics.

The main difference here is that the state may be $\bot$, if the action is executed on empty state. This ensures non-unital RAs are not ruled out as invalid – indeed, many useful RAs are not unital and sometimes don't have a unit at all, as is the case for instance for Ex, the exclusively owned cell. One could decide to internally make all RAs of state models unital, and have the state model provide an `empty` function that returns said unit (this is what happens in Gillian). However this introduces unsoundness to certain state model constructions (in particular the sum), as this means the state cannot be *exclusively owned* – the empty state could always be composed with it.

Whole-program symbolic execution is, by definition, non-compositional – it thus operates on *full state*, a notion introduced in [40]. As such, the only valid outcomes here are `Ok` and `Err`.

Because we operate in symbolic memory, an additional piece of information is the *path condition*, the set of constraints accumulated throughout execution. A path condition $\pi \in \Pi$ is a *list of symbolic values*, that evaluates to a boolean. We decide to define it as a list rather than a single conjunction of boolean symbolic values, as this allows us to easily

check if a path condition is an extension (or a strengthening) of another, with $\pi' \supseteq \pi$. We further define the predicate SAT, that is true if, given the substitution θ, store s and a path condition π, the conjunction of the elements of π resolves to `true` once evaluated:

$$\mathsf{SAT}_{\theta,s}(\pi) \stackrel{\text{def}}{=} \left[\!\!\left[\bigwedge_{i}^{|\pi|} \pi(i) \right]\!\!\right]_{\theta,s} = \mathtt{true}$$

We note that **cse2** also has an 'SV' argument in action execution, that contains all existing symbolic variables, and that must be used when creating a new symbolic variable to ensure it is fresh. While necessary for proofs within the engine with Rocq, we omit it here. It is only used for allocation and it can instead be kept implicit, by assuming we can always generate a fresh symbolic variable.

Finally, the user must define the `sat` relation, relating concrete and symbolic states. It is pretty printed as $\theta, s, \sigma \models \hat{\sigma}$, meaning that given a substitution θ and a store s, the concrete state σ can be matched by a symbolic state $\hat{\sigma}$. We define this relation for non-$\bot$ states; we can then lift it to the option state model $\Sigma^?$, by simply adding that $\forall \theta, s.\, \theta, s, \bot \models \bot$. This relieves users from needing to take $\bot$ into account when defining $\models$ and from proving the fairly trivial axiom Empty Memory.

3.2.3 Compositional Engine

The compositional engine, built on top of the core engine, allows for verification of function specifications, and handles calling functions by their specification. As such, the state model must be extended with a set of core predicates Δ and a pair of `consume` and `produce` functions (equivalent, respectively, to a resource assert and assume). Finally, to link core predicates to states, it provides a sat_Δ relation.

$$\begin{aligned}
&\text{for } M = \{\mathtt{OX}, \mathtt{UX}\} \\
&\qquad \mathtt{consume} : M \to \hat{\Sigma}^? \to \Delta \to \text{Val } \mathtt{list} \to \mathcal{P}(\mathcal{O}_l \times \hat{\Sigma}^? \times \text{Val } \mathtt{list} \times \Pi) \\
&\qquad \mathtt{produce} : \hat{\Sigma}^? \to \Delta \to \text{Val } \mathtt{list} \to \text{Val } \mathtt{list} \to \mathcal{P}(\hat{\Sigma}^? \times \Pi) \\
&\qquad\quad \mathsf{sat}_\Delta : \Sigma^? \to \Delta \to \text{Val } \mathtt{list} \to \mathcal{P}(\text{Val } \mathtt{list})
\end{aligned}$$

Similarly to `execute_action`, the input state can be $\bot$. While intuitively one may assume that the input state of `consume` and the output state of `produce` may never be $\bot$, this would limit what core predicates can do. In particular, this means an *emp* predicate couldn't be defined, since it's production on an empty state results in an empty state.

The arguments of `consume` are, in order: the mode of execution to distinguish between

under-approximate and over-approximate reasoning, the state, the core predicate being consumed, the ins of the predicate. It outputs a *logical outcome*, the state with the matching predicate removed (which may result in an empty state $\bot$), the outs of the predicate and the associated path condition. It is pretty-printed as $\texttt{consume}(m, \hat{\sigma}, \delta, \vec{\hat{v}}_i) \rightsquigarrow (o, \hat{\sigma}_f, \vec{\hat{v}}_o, \pi)$, and when the consumption is valid in both OX and UX the mode is omitted.

For `produce`, the arguments are the state, the core predicate being produced, the ins and the outs of the predicate, resulting in a set of new states and their associated path condition. As an example, producing $x \mapsto 0$ in a state $[1 \mapsto 2]$ results in a new state $[1 \mapsto 2, x \mapsto 0]$ with the path condition $x \neq 1$. If the produced predicate is incompatible with the state (eg. by producing $1 \mapsto y$ in a state containing $1 \mapsto x$), the producer *vanishes*. Inversely, if the assertion can be interpreted in several ways, the producer may branch. It is pretty-printed as $\texttt{produce}(\hat{\sigma}, \delta, \vec{\hat{v}}_i, \vec{\hat{v}}_o) \rightsquigarrow (\hat{\sigma}', \pi)$.

The $\texttt{sat}_\Delta$ relation relates a possibly empty *concrete* state, core predicate and in-values to a set of out-values. It is pretty-printed as $\sigma \models_\Delta \langle\delta\rangle(\vec{v}_i; \vec{v}_o)$ for $\vec{v}_o \in \texttt{sat}_\Delta\ \sigma\ \delta\ \vec{v}_i$. For instance in the linear heap state model, we have $[1 \mapsto 2] \models_\Delta \langle\texttt{points\_to}\rangle(1; 2)$. We also lift this relation to the symbolic realm:

$$\theta, s, \sigma \models_\Delta \langle\delta\rangle(\vec{\hat{v}}_i; \vec{\hat{v}}_o) \stackrel{\text{def}}{=} [\![\vec{\hat{v}}_i]\!]_{\theta,s} = \vec{v}_i \wedge [\![\vec{\hat{v}}_o]\!]_{\theta,s} = \vec{v}_o \wedge \sigma \models_\Delta \langle\delta\rangle(\vec{v}_i; \vec{v}_o)$$

Here we define logical outcomes $\mathcal{O}_l = \{\texttt{Ok}, \texttt{LFail}, \texttt{Miss}\}$. These are outcomes that happen during reasoning; in particular, `LFail` equates to a logical failure due to an incompatibility between the consumed predicate and the state. For instance, consuming $1 \mapsto 1$ when in state $1 \mapsto 2$ would yield a `LFail`, while consuming it in state $5 \mapsto 3$ would yield a `Miss`, as a state $1 \mapsto x$ could be composed with it to yield a non-miss outcome.

We must also modify the signature of `execute_action`, to include `Miss` outcomes, via the set of execution outcomes $\mathcal{O}_e = \{\texttt{Ok}, \texttt{Err}, \texttt{Miss}\}$.

An addition to what CSE previously defined is thus the split of what was the `Abort` outcome into `LFail` and `Miss`, this improves the quality of error messages and allows fixing consumption errors due to missing state – this will be described in the next subsection.

A last change compared to CSE is that we drop the path condition parameter to consume and produce – the function instead directly returns the path condition required for the resulting branches, and the engine can filter these. For instance, in UX all consumption branches that result in `LFail` can be dropped, as dropping branches is allowed in UX. This has the advantage of simplifying the axioms, as the path condition is strengthened by definition; the function itself has no way of weakening it. *Mention that this also makes checkpointing with the SAT engine trivial – just set a checkpoint before* `consume`/`produce`,

add whatever that returns. No need to separate the old from new, avoids duplicating/deduplicating, etc.

3.2.4 Bi-Abduction Engine

To support bi-abduction in the style of Infer:Pulse [33], `Miss` outcomes must be fixed. These outcomes may happen during consumption or during action execution. For this, the state model must provide a `fix` function, that given the details of a miss error (these details being of type Val and returned with the outcome) returns a list of sets of assertions that must be produced to fix the missing error.

$$\texttt{fix} : \text{Val} \to \mathcal{P}(\text{Asrt})\ \texttt{list}$$

Note here we return a *list* of different fixes, which themselves are a set of assertions – this is because, for a given missing error, multiple fixes may be possible which causes branching. For instance, in the typical linear heap, accessing a cell that is not in the state fragment at address a results in a miss that has two fixes: either the cell exists and points to some existentially quantified variable (the fix is thus $\exists x.\, a \mapsto x$), or the cell exists and has been freed ($a \mapsto \varnothing$).

This approach is different from how Gillian handles it. There the function `fix` returns *pure* assertions (type information, pure formulae) and arbitrary values of type `fix_t`, which can then be used with the $\texttt{apply\_fix} : \Sigma \to \texttt{fix\_t} \to \Sigma$ method of the monadic state. This means fixes can be arbitrary modifications to the state that don't necessarily equate to new assertions to add to the anti-frame.

This is a source of unsoundness, as the engine may interpret these modifications as fixes despite them not reliably modifying the state. This can be seen in [40], where not finding the binding in a PMap(X) returns a `MissingBinding` error. While being labelled as a miss, this error can actually not be fixed; PMap simply *lifts* predicates with an additional in parameter for the index. An implementation of that version of PMap(X) could attempt to fix this state by add a binding to $X.0$ (PMaps were originally made for PCMs, which always have a 0 element), which would then eventually lead to another error once the action gets called on the empty state. On top of being under-performing (as several fixes would need to be generated for one action), this requires PMap(X) to allow empty states in the codomain, which means a PMap is never exclusively owned (as a state with a singleton map to $X.0$ can always be composed with it), which limits its usability; aside from not being modelable using RAs, since $\bot$ is not an element of X's carrier set. Finally, if the underlying state model doesn't provide any additional fixes, then the fix for `MissingBinding` cannot be added to

the UX specification of a function: there is no assertion generatable from within PMap to represent this modification. As such, having `fix` returns assertions without modifying any state directly ensures fixes are always soundly handled.

To finish this, we may note the solution to the above bug is to proceed executing the action on the underlying state model, giving it an empty state – it will then raise the appropriate `Miss`, which can be fixed, as it is aware of what core predicates are needed to create the required state. For instance, for PMap(Exc) a `load` action on a missing binding would be executed against $\bot$, which would return a `MissingValue` error. The PMap could then wrap the error with information about the index at which the error occurred, $\texttt{SubError}(i, \texttt{MissingValue})$. When getting the fix, PMap can then call Exc.`fix`, which returns $\exists x.\, \langle \mathit{points\_to} \rangle(; x)$, and lift the fix by adding the index as an in-argument, resulting in the final fix $\exists x.\, \langle \mathit{points\_to} \rangle(i; x)$, which is a valid assertion and can be added to the UX specification for this execution.

3.2.5 Axioms

We may now go over the axioms that must be respected by the above defined functions for the soundness of the engine. Note we will thus focus on the axioms related to the state models in particular, and not the general semantics of the engine.

It is worth noting that Gillian supports both over-approximate (OX) and under-approximate (UX) reasoning – for which *frame subtraction* or *frame addition* must hold, respectively.

For all of the axioms we assume we have a symbolic state model $\mathbb{S}$, made of the RA $\hat{\Sigma} \ni \hat{\sigma}$. We consider the initial state $\hat{\sigma}$ well-formed.

Symbolicness Axioms

$$\theta, s, \sigma \models \bot \iff \sigma = \bot \qquad \text{(Empty Memory)}$$

The above axiom is obtained for free, by lifting the $\models$ relation to the option RA.

$$\begin{aligned}
&\theta, s, \sigma \models \hat{\sigma} \land \alpha(\sigma, \vec{v}_i) = (o, \sigma', \vec{v}_o) \land [\![\vec{\hat{v}}_i]\!]_{\theta,s} = \vec{v}_i \land \\
&(\forall o', \hat{\sigma}', \vec{\hat{v}}'_o, \pi'.\, \alpha(\hat{\sigma}, \vec{\hat{v}}_i) \rightsquigarrow (o', \hat{\sigma}', \vec{\hat{v}}'_o, \pi') \Rightarrow o' \in \{\texttt{Ok}, \texttt{Err}\}) \implies \exists \hat{\sigma}', \vec{\hat{v}}_o, \pi, \theta'. \\
&\quad \hat{\alpha}(\hat{\sigma}, \vec{\hat{v}}_i) \rightsquigarrow (o, \hat{\sigma}', \vec{\hat{v}}_o, \pi) \land \theta', s, \sigma' \models \hat{\sigma}' \land \mathsf{SAT}_{\theta',s}(\pi) \land [\![\vec{\hat{v}}_o]\!]_{\theta',s} = \vec{v}_o
\end{aligned}$$

(Memory Model OX Soundness)

$$\hat{\alpha}(\hat{\sigma}, \vec{\hat{v}}_i) \rightsquigarrow (o, \hat{\sigma}', \vec{\hat{v}}_o, \pi) \land \mathsf{SAT}_{\theta',s}(\pi) \land \theta, s, \sigma' \models \hat{\sigma}' \land$$
$$[\![\vec{\hat{v}}_o]\!]_{\hat{s},\pi} \rightsquigarrow (\vec{v}_o, \pi') \land [\![\vec{\hat{v}}_i]\!]_{\hat{s},\pi'} \rightsquigarrow (\vec{v}_i, \pi'') \implies \qquad \text{(Memory Model UX Soundness)}$$
$$\exists \sigma.\, \theta, s, \sigma \models \hat{\sigma} \land \alpha(\sigma, \vec{v}_i) = (o, \sigma', \vec{v}_o)$$

Compositionality Axioms

$$\sigma \# \sigma_f \land \alpha(\sigma \cdot \sigma_f, \vec{\hat{v}}_i) = (o, \sigma', \vec{v}_o) \implies$$
$$\exists \sigma'', o', \vec{v}'_o.\, \alpha(\sigma, \vec{v}_i) = (o', \sigma'', \vec{v}'_o) \land \qquad \text{(Frame subtraction)}$$
$$(o' \neq \mathtt{Miss} \implies o' = o \land \vec{v}'_o = \vec{v}_o \land \sigma' = \sigma'' \cdot \sigma_f)$$

$$\alpha(\sigma, \vec{v}_i) = (o, \sigma', \vec{v}_o) \land o \neq \mathtt{Miss} \land \sigma' \# \sigma_f \implies$$
$$\sigma \# \sigma_f \land \alpha(\sigma \cdot \sigma_f, \vec{v}_i) = (o, \sigma' \cdot \sigma_f, \vec{v}_o) \qquad \text{(Frame Addition)}$$

Here, we may note that the frame-preserving update $a \rightsquigarrow b$ from Iris is a form of frame subtraction: it guarantees $\forall c.\, \overline{\mathcal{V}}(a \cdot c) \Rightarrow \overline{\mathcal{V}}(b \cdot c)$, with c a frame that can be added to the state ($\hat{\sigma}_f$ in the axiom). This becomes evident when noticing that disjointness of partial RAs equates to validity in Iris RAs, giving us $\forall c.\, a \# c \Rightarrow b \# c$. In fact, the Iris frame-preserving update implies frame subtraction modulo action outcomes. This makes sense, as Iris is used for OX reasoning, and frame subtraction is the property needed for OX soundness. *Maybe move this elsewhere.*

$$\mathsf{consume}(\hat{\sigma}, \delta, \vec{\hat{v}}_i) \rightsquigarrow (\mathtt{Ok}, \hat{\sigma}_f, \vec{\hat{v}}_o, \pi) \implies \forall \theta, s, \sigma_f, \sigma_\delta.$$
$$\theta, s, \sigma_f \models \hat{\sigma}_f \land \theta, s, \sigma_\delta \models_\Delta \langle\delta\rangle(\vec{\hat{v}}_i; \vec{\hat{v}}_o) \land \sigma_f \# \sigma_\delta \implies \qquad \text{(Consume OX Soundness)}$$
$$\exists \sigma.\, \sigma = \sigma_\delta \cdot \sigma_f \land \theta, s, \sigma \models \hat{\sigma} \land \mathsf{SAT}_{\theta,s}(\pi)$$

$$(\forall o, \hat{\sigma}_f, \vec{\hat{v}}_o, \pi.\, \mathsf{consume}(\mathtt{OX}, \hat{\sigma}, \delta, \vec{\hat{v}}_i) \rightsquigarrow (o, \hat{\sigma}_f, \vec{\hat{v}}_o, \pi) \Rightarrow o_c = \mathtt{Ok}) \implies$$
$$\exists \hat{\sigma}'_f, \vec{\hat{v}}'_o, \pi'.\, \mathsf{consume}(\mathtt{OX}, \hat{\sigma}, \delta, \vec{v}_i, \pi) \rightsquigarrow (\mathtt{Ok}, \hat{\sigma}'_f, \vec{\hat{v}}'_o, \pi')$$
(Consume OX: No Path Drops)

$$\mathsf{consume}(\hat{\sigma}, \delta, \vec{\hat{v}}_i) \rightsquigarrow (\mathtt{Ok}, \hat{\sigma}_f, \vec{\hat{v}}_o, \pi) \implies \forall \theta, s, \sigma.$$
$$\theta, s, \sigma \models \hat{\sigma} \land \mathsf{SAT}_{\theta,s}(\pi) \implies \exists \sigma_\delta, \sigma_f.$$
$$\sigma_\delta \# \sigma_f \land \sigma = \sigma_\delta \cdot \sigma_f \land \theta, s, \sigma_\delta \models_\Delta \langle\delta\rangle(\vec{\hat{v}}_i; \vec{\hat{v}}_o) \land \theta, s, \sigma_f \models \hat{\sigma}_f$$
(Consume UX Soundness)

$$\theta, s, \sigma_f \models \hat{\sigma}_f \land \theta, s, \sigma_\delta \models_\Delta \langle\delta\rangle(\vec{\hat{v}}_i; \vec{\hat{v}}_o) \land \sigma_f \# \sigma_\delta \implies$$
$$\exists \hat{\sigma}.\, \mathsf{produce}(\hat{\sigma}_f, \delta, \vec{\hat{v}}_i, \vec{\hat{v}}_o) \rightsquigarrow (\hat{\sigma}, \pi) \land \mathsf{SAT}_{\theta,s}(\pi) \land \theta, s, (\sigma_f \cdot \sigma_\delta) \models \hat{\sigma}$$
(Produce: OX Soundness)

$$\mathsf{produce}(\hat{\sigma}_f, \delta, \vec{\hat{v}}_i, \vec{\hat{v}}_o) \rightsquigarrow (\hat{\sigma}, \pi) \implies$$
$$\forall \theta, s, \sigma.\, \mathsf{SAT}_{\theta,s}(\pi) \land \theta, s, \sigma \models \hat{\sigma} \implies \exists \sigma_\delta, \sigma_f.$$
$$\sigma_\delta \# \sigma_f \land \sigma = \sigma_\delta \cdot \sigma_f \land \theta, s, \sigma_\delta \models_\Delta \langle\delta\rangle(\vec{\hat{v}}_i; \vec{\hat{v}}_o) \land \theta, s, \sigma_f \models \hat{\sigma}_f$$

(Produce: UX Soundness)

3.3 State Models

3.3.1 Exclusive

3.3.2 Agreement

3.3.3 Fraction

3.4 State Model Transformers

3.4.1 Sum

3.4.2 Product

3.4.3 Freeable

3.4.4 Partial Map

3.4.5 Dynamic Partial Map

3.4.6 List

3.4.7 General Map

3.5 Optimising the Partial Maps

3.5.1 Syntactic Checking

3.5.2 Split PMap

3.5.3 Abstract Location PMap

Chapter 4

Implementation

4.1 State Model Library

4.1.1 State Model Interface

4.1.2 Exclusive

4.1.3 Partial Map

4.1.4 Helper State Models

4.2 Instantiations

4.2.1 WISL

4.2.2 JavaScript

4.2.3 C

Chapter 5

Evaluation

5.1 Usability of Theory

5.2 Usability of Library

5.3 Comparison with Gillian

5.3.1 WISL

5.3.2 JavaScript

5.3.3 C

5.4 Partial Map Performance

Chapter 6

Conclusion

6.1 Future Work

Bibliography

[1] P. O'Hearn, J. Reynolds, and H. Yang, "Local Reasoning about Programs that Alter Data Structures," in *Computer Science Logic*, L. Fribourg, Ed., Berlin, Heidelberg: Springer Berlin Heidelberg, 2001, pp. 1–19, ISBN: 978-3-540-44802-0.

[2] J. Reynolds, "Separation logic: a logic for shared mutable data structures," in *Proceedings 17th Annual IEEE Symposium on Logic in Computer Science*, 2002, pp. 55–74. DOI: 10.1109/LICS.2002.1029817.

[3] J. F. Santos, P. Maksimović, S.-É. Ayoun, and P. Gardner, *Gillian: Compositional Symbolic Execution for All*, 2020. arXiv: 2001.05059 [cs.PL].

[4] J. Fragoso Santos, P. Maksimović, S.-E. Ayoun, and P. Gardner, "Gillian, Part I: A Multi-Language Platform for Symbolic Execution," in *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI 2020, London, UK: Association for Computing Machinery, 2020, pp. 927–942, ISBN: 9781450376136. DOI: 10.1145/3385412.3386014. [Online]. Available: https://doi.org/10.1145/3385412.3386014.

[5] P. Maksimović, S.-E. Ayoun, J. F. Santos, and P. Gardner, "Gillian, Part II: Real-World Verification for JavaScript and C," in *Computer Aided Verification: 33rd International Conference, CAV 2021, Virtual Event, July 20–23, 2021, Proceedings, Part II*, Berlin, Heidelberg: Springer-Verlag, 2021, pp. 827–850, ISBN: 978-3-030-81687-2. DOI: 10.1007/978-3-030-81688-9_38. [Online]. Available: https://doi.org/10.1007/978-3-030-81688-9_38.

[6] C. A. R. Hoare, "An axiomatic basis for computer programming," *Commun. ACM*, vol. 12, no. 10, pp. 576–580, 1969, ISSN: 0001-0782. DOI: 10.1145/363235.363259. [Online]. Available: https://doi.org/10.1145/363235.363259.

[7] A. Raad, J. Berdine, H.-H. Dang, D. Dreyer, P. O'Hearn, and J. Villard, "Local Reasoning About the Presence of Bugs: Incorrectness Separation Logic," in *Computer Aided Verification*, S. K. Lahiri and C. Wang, Eds., Cham: Springer International Publishing, 2020, pp. 225–252, ISBN: 978-3-030-53291-8.

[8] P. W. O'Hearn, "Incorrectness logic," *Proc. ACM Program. Lang.*, vol. 4, no. POPL, 2019. DOI: 10.1145/3371078. [Online]. Available: https://doi.org/10.1145/3371078.

[9] P. Maksimović, C. Cronjäger, A. Lööw, J. Sutherland, and P. Gardner, "Exact Separation Logic: Towards Bridging the Gap Between Verification and Bug-Finding," en, Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2023. DOI: 10.4230/LIPICS.ECOOP.2023.19. [Online]. Available: https://drops.dagstuhl.de/entities/document/10.4230/LIPIcs.ECOOP.2023.19.

[10] X. Leroy, A. Appel, S. Blazy, and G. Stewart, "The CompCert Memory Model, Version 2," Jun. 2012.

[11] R. Bornat, C. Calcagno, P. Hearn, and H. Yang, "Fractional and counting permissions in separation logic,"

[12] R. Bornat, C. Calcagno, P. O'Hearn, and M. Parkinson, "Permission accounting in separation logic," *SIGPLAN Not.*, vol. 40, no. 1, pp. 259–270, 2005, ISSN: 0362-1340. DOI: 10.1145/1047659.1040327. [Online]. Available: https://doi.org/10.1145/1047659.1040327.

[13] M. J. Parkinson, "The next 700 separation logics," in *2010 Verified Software: Theories, Tools, Experiments*, ser. Lecture Notes in Computer Science, vol. 6217, Springer Berlin / Heidelberg, 2010, pp. 169–182. [Online]. Available: https://www.microsoft.com/en-us/research/publication/the-next-700-separation-logics/.

[14] C. Calcagno, P. W. O'Hearn, and H. Yang, "Local Action and Abstract Separation Logic," in *22nd Annual IEEE Symposium on Logic in Computer Science (LICS 2007)*, 2007, pp. 366–378. DOI: 10.1109/LICS.2007.30.

[15] R. Dockins, A. Hobor, and A. W. Appel, "A Fresh Look at Separation Algebras and Share Accounting," in *Programming Languages and Systems*, Z. Hu, Ed., Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 161–177, ISBN: 978-3-642-10672-9.

[16] F. Pottier, "Syntactic soundness proof of a type-and-capability system with hidden state," *Journal of Functional Programming*, vol. 23, no. 1, pp. 38–144, 2013. DOI: 10.1017/S0956796812000366.

[17] R. Jung, R. Krebbers, J.-H. Jourdan, A. Bizjak, L. Birkedal, and D. Dreyer, "Iris from the ground up: A modular foundation for higher-order concurrent separation logic," *Journal of Functional Programming*, vol. 28, e20, 2018. DOI: 10.1017/S0956796818000151.

[18] S. Bekrar, C. Bekrar, R. Groz, and L. Mounier, "Finding Software Vulnerabilities by Smart Fuzzing," in *2011 Fourth IEEE International Conference on Software Testing, Verification and Validation*, 2011, pp. 427–430. DOI: 10.1109/ICST.2011.48.

[19] R. Baldoni, E. Coppa, D. C. D'elia, C. Demetrescu, and I. Finocchi, "A Survey of Symbolic Execution Techniques," *ACM Comput. Surv.*, vol. 51, no. 3, 2018, ISSN: 0360-0300. DOI: 10.1145/3182657. [Online]. Available: https://doi.org/10.1145/3182657.

[20] R. S. Boyer, B. Elspas, and K. N. Levitt, "Select—a formal system for testing and debugging programs by symbolic execution," in *Proceedings of the International Conference on Reliable Software*, Los Angeles, California: Association for Computing Machinery, 1975, pp. 234–245, ISBN: 9781450373852. DOI: 10.1145/800027.808445. [Online]. Available: https://doi.org/10.1145/800027.808445.

[21] J. C. King, "Symbolic execution and program testing," *Commun. ACM*, vol. 19, no. 7, pp. 385–394, 1976, ISSN: 0001-0782. DOI: 10.1145/360248.360252. [Online]. Available: https://doi.org/10.1145/360248.360252.

[22] S. Anand, P. Godefroid, and N. Tillmann, "Demand-Driven Compositional Symbolic Execution," in *Tools and Algorithms for the Construction and Analysis of Systems*, C. R. Ramakrishnan and J. Rehof, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 367–381, ISBN: 978-3-540-78800-3.

[23] J. Berdine, C. Calcagno, and P. O'Hearn, "Modular automatic assertion checking with separation logic," vol. 4111, Nov. 2005, pp. 115–137, ISBN: 978-3-540-36749-9. DOI: 10.1007/11804192_6.

[24] J. Fragoso Santos, P. Maksimovic, D. Naudziuniene, T. Wood, and P. Gardner, "JaVerT: JavaScript verification toolchain," *Proc. ACM Program. Lang.*, vol. 2, no. POPL, 2017. DOI: 10.1145/3158138. [Online]. Available: https://doi.org/10.1145/3158138.

[25] J. Fragoso Santos, P. Maksimović, G. Sampaio, and P. Gardner, "JaVerT 2.0: compositional symbolic execution for JavaScript," *Proc. ACM Program. Lang.*, vol. 3, no. POPL, 2019. DOI: 10.1145/3290379. [Online]. Available: https://doi.org/10.1145/3290379.

[26] C. Calcagno and D. Distefano, "Infer: an automatic program verifier for memory safety of C programs," in *Proceedings of the Third International Conference on NASA Formal Methods*, ser. NFM'11, Pasadena, CA: Springer-Verlag, 2011, pp. 459–465, ISBN: 9783642203978.

[27] B. Jacobs, J. Smans, and F. Piessens, "A Quick Tour of the VeriFast Program Verifier," vol. 6461, Nov. 2010, pp. 304–311, ISBN: 978-3-642-17163-5. DOI: 10.1007/978-3-642-17164-2_21.

[28] H. Yang, O. Lee, J. Berdine, C. Calcagno, B. Cook, D. Distefano, and P. O'Hearn, "Scalable shape analysis for systems code," Jul. 2008, pp. 385–398, ISBN: 978-3-540-70543-7. DOI: 10.1007/978-3-540-70545-1_36.

[29] D. Kroening, P. Schrammel, and M. Tautschnig, *CBMC: The C Bounded Model Checker*, 2014. arXiv: 2302.02384 [cs.SE].

[30] C. Cadar, D. Dunbar, and D. Engler, "KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs," in *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI'08, San Diego, California: USENIX Association, 2008, pp. 209–224.

[31] S. Ognawala, M. Ochoa, A. Pretschner, and T. Limmer, "Macke: Compositional analysis of low-level vulnerabilities with symbolic execution," in *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '16, Singapore, Singapore: Association for Computing Machinery, 2016, pp. 780–785, ISBN: 9781450338455. DOI: 10.1145/2970276.2970281. [Online]. Available: https://doi.org/10.1145/2970276.2970281.

[32] D. Distefano and M. J. Parkinson J, "jStar: towards practical verification for java," in *Proceedings of the 23rd ACM SIGPLAN Conference on Object-Oriented Programming Systems Languages and Applications*, ser. OOPSLA '08, Nashville, TN, USA: Association for Computing Machinery, 2008, pp. 213–226, ISBN: 9781605582153. DOI: 10.1145/1449764.1449782. [Online]. Available: https://doi.org/10.1145/1449764.1449782.

[33] Q. L. Le, A. Raad, J. Villard, J. Berdine, D. Dreyer, and P. W. O'Hearn, "Finding real bugs in big programs with incorrectness logic," *Proc. ACM Program. Lang.*, vol. 6, no. OOPSLA1, 2022. DOI: 10.1145/3527325. [Online]. Available: https://doi.org/10.1145/3527325.

[34] J. Fragoso Santos, P. Maksimović, T. Grohens, J. Dolby, and P. Gardner, "Symbolic Execution for JavaScript," Sep. 2018, pp. 1–14. DOI: 10.1145/3236950.3236956.

[35] P. Müller, M. Schwerhoff, and A. J. Summers, "Viper: A Verification Infrastructure for Permission-Based Reasoning," in *Verification, Model Checking, and Abstract Interpretation*, B. Jobstmann and K. R. M. Leino, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2016, pp. 41–62, ISBN: 978-3-662-49122-5.

[36] M. Botincan, D. Distefano, M. Dodds, R. Grigore, D. E, and M. Parkinson, "Corestar: The core of jstar," May 2012.

[37] S.-É. Ayoun, X. Denis, P. Maksimović, and P. Gardner, *A hybrid approach to semi-automated Rust verification*, 2024. arXiv: 2403.15122 [cs.PL].

[38] R. Jung, *Understanding and evolving the rust programming language*, 2020. DOI: http://dx.doi.org/10.22028/D291-31946.

[39] L. Birkedal, *Iris: Higher-order concurrent separation logic - lecture 10: Ghost state*, 2020. [Online]. Available: https://iris-project.org/tutorial-pdfs/lecture10-ghost-state.pdf.

[40] S.-E. Ayoun, "Gillian: Foundations, Implementation and Applications of Compositional Symbolic Execution."

Appendix A

State Model Definitions

We list here the different state models developed. In particular, we list their components in the following order: the definition of the Resource Algebra, along with its core $|-|$ and composition $(\cdot)$