IMPERIAL

Adding and Optimising Resource Algebras for a Parametric CSE

Author: Opale Sjöstedt

Supervisor: Philippa Gardner

September 5, 2024

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

Abstract

Compositional symbolic execution is a technique allowing for the scalable verification of code, using separation logic. A particularly rich branch of separation logic is abstract separation logic, that allows reasoning about an abstract state. Iris is a state of the art theory in the field, that introduces the notion of resource algebras, that enable the construction of sound complex state from simple elements.

Gillian is a compositional symbolic execution engine that has the unique property of being parametric on the state model, allowing verification of any language for which a state model can be defined.

While some preliminary work exists to adapt the notion of state model constructions to Gillian, it uses partially commutative monoids, a form of state representation that frequently causes soundness issues. In this project we thus show how resource algebras can be used to replace monoids in this setting, while providing easier to prove soundness.

In this work we adapt a abstract CSE engine to use state models backed by resource algebras. We then define a range of different resource algebras that can be used to construct complex state models, which allow for the verification of real-world languages. We implement these for the Gillian CSE engine, and show that the instantiations built with them are both simpler and more performant than the original Gillian instantiations.

Contents

Chapter 1

Introduction

1.1 Motivation

As software becomes increasingly central to critical infrastructure, the need to formally prove its correctness increases. A framework that enables scalable program verification is *separation logic* [52, 58], from which *emp*abstract separation logic followed, allowing one to reason about properties of the memory independently of its actual representation [8, 16]. An oustanding logic in this field is Iris [34], which proved itself to be more than capable of enabling reasoning for a wide range of applications. These logics have however been limited to manual pen and paper proofs, or inaccessible mechanisations with Rocq.

In parallel, symbolic execution tools have been developed to allow for the verification of programs automatically, with however poor scalability [1]. *Compositional symbolic execution* (CSE) uses advances made in separation logic to allow for the verification of code by fragments, improving drastically scalability and allowing for adoption in Continuous Integration pipelines [39].

A particular CSE engine is Gillian [59]; it has the unique property of being *parametric on the memory model*, allowing it to be re-used to verify any language, so long as the memory model is provided by the developer. In [2], the idea of *constructing* memory models was introduced, using partially commutative monoids. Gillian is a perfect match for this, as it is inherently compatible with any constructable memory model. The original definitions were however at times incomplete, or suffered from unsoundness due to their use of partially commutative monoid, the classical representation of memory in abstract separation logic.

1.2 Objectives

Given the great potential in facilitating state model creation by construction, we plan on better developing the theory surrounding it, borrowing the notion of *resource algebras* from Iris. To this end, we will also see

what changes need to be done to a theoretical CSE engine, first described in [43, 44], to take advantage of these resource algebras.

In particular, the main objectives of this project are:

- **RQ1**: what are the improvements provided by the use of resource algebras, compared to partially commutative monoids, when used for state models? What changes are needed to our state model definitions to take advantage of resource algebras?
- **RQ2**: how do we define state models and state model transformers to ensure they are provably sound, both when used in overapproximate and underapproximate reasoning? Can the optimisations made when implementing the state models be lifted back to the theory and proven to be sound?
- **RQ3**: are state model transformers a viable approach performance-wise? How do instantiations of Gillian using state model transformers perform when compared with pre-existing instantiations? Do the defined optimisations bring actual improvements to performance?

1.3 Contributions

The main contribution of this project is the adoption of resource algebras for the construction of state models that are compatible with CSE engines.

We define at the theory level 7 different state models, as well as 3 optimised versions of the partial map state model transformer. All of these are proven to be sound.

We then implement these state models along with additional ones in OCaml, creating a state model library for Gillian. We have then, using this library, instantiated Gillian to allow for the verification of code written in WISL, JavaScript and C. These instantiations have full parity with the pre-existing instantiations of Gillian, providing the same verification results with a fraction of the code needed for the instantiation.

Finally, we measure in great detail the performance of the instantiations and optimisations described, comparing them to performance in Gillian, and showing areas for improvement.

1.4 Outline

In chapter 2 we introduce the necessary theoretical background needed for the project; we go over separation logic and its variations, with incorrectness separation logic and abstract separation logic. In particular we retrace the history of abstract separation logic and its evolution, leading to the creation of Iris. We then describe Gillian, a compositional symbolic execution engine parametric on the memory model. We describe existing tools that enable symbolic execution and their differences. Finally, we briefly cover additional requirements relevant to this research area.

In chapter 3 we motivate the transition from partially commutative monoids to resoure algebras. We then go through the changes brought to our abstract CSE engines to accomodate the change, and define a wide range of state models. We also cover optimisations at the theoretical level for partial map transformer.

In chapter 4 we describe the implementation of these state models as a library, and how our theoretical constructs interface with the pre-existing definitions of Gillian. We also cover how a developer may use this library to in turn instantiate Gillian to the desired state model.

In chapter 5 we evaluate the work done so far, discussing first the usability of the theory and of the library. We then compare the performance of our instantiations with the reference Gillian instantiations. We also briefly observe the performance improvements offered by our partial map optimisations.

In chapter 6 we conclude this report, retracing the work done, and outlining the key achievements, the relevant limitations, and give different directions to build upon what is defined here, for future work.

Chapter 2

Background

In this chapter we introduce some of the core ideas used throughout this document, while mentioning pre-existing work in the field. In §2.1 we introduce Separation Logic, its variations, and the idea of Separation Algebras; in §2.2 we introduce Gillian, a CSE engine; in §2.3 we mention other similar verification tools; finally, in §2.4 we evoke the legal, social, ethical and professional impacts of this work.

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 [30] allows 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 helpful to reason about programs, an issue remains: shared mutable state. 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) [52, 58] is an extension of Hoare logic that permits this in a clear and scalable way. Its 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 of which code is modified, while reusing past analyses of unchanged code, allowing for incremental analysis.

SL also comes equipped with the *emp* predicate, representing an empty state (i.e. $P * emp = P$), and the separating implication $-\!\!*$. $P_0 -\!\!* P_1$ states that if the current state is extended with a disjoint part satisfying P_0, then P_1 holds in the extended heap [58].

Further predicates can then be defined; for instance the "points to" predicate, $a \mapsto x$, stating that the address a stores values x. For instance, $a \mapsto x * a \mapsto x$ does not hold, since we can't have two disjoint states both containing the address a. Similarly, $a \mapsto x * b \mapsto y$ only holds 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 can't occur are detected and flag the code as incorrect, hindering the use of a verification tool (for instance in a CI setting).

To solve this problem, a recent innovation in the field is Incorrectness Separation Logic (ISL) [57], derived from Incorrectness Logic [53]. It is the *under-approximate* (UX) equivalent to SL, instead ensuring that the detected errors actually exist, at the cost of possibly missing some executions of the code.

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. This enables us to do *true bug-finding*.

Another way of comparing SL to ISL is with consequence: in SL, the precondition implies the post-condition, whereas with ISL the result implies the presumption. 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 [46]. 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, we often need more complex state representations to verify real-world programming languages. For instance, the model used by the C language (in particular CompCert C [40], a verified C compiler) uses the notion of memory blocks and offsets: memory isn't just an assortment of different atomic cells. Furthermore, the basic "points to" predicate, while useful, has limited applications; for contexts such as 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 [9, 10]: 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" [54] 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 [17]. 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* [16, 22], which allow for the *construction* of complex separation algebras 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 [16] they are initially defined as a *cancellative* Partially Commutative Monoid (PCM) $(\Sigma, \cdot, 0)$. This definition is however too strong: for instance, the cancellativity property implies $\sigma \cdot \sigma' = \sigma \Rightarrow \sigma' = 0$. 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 have $\sigma \cdot \sigma = \sigma$, which dissatisfies cancellativity.

This approach is also taken in [22], 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 authors wanting to construct their models in Rocq (formerly Coq) [18], which only supports computable total functions. While the distinction is mostly syntactical and both relational and functional approaches are equivalent as long at the relation is functional, the former makes proof-work less practical when working with partial functions [56], whereas the functional approach allows one to reason equationally [34]. This paper also introduces the notion of *multi-unit separation algebras*, separation algebras that don't have a single unit 0, 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 rather a type of partially commutative semigroup.

In [56], 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 $\hat{-}$, the core of a resource, which is its *duplicable part*.

Finally, Iris [33–35, 37] is a state of the art "higher-order concurrent separation logic". It places itself as a solution to the problem mentioned in [54], and aims to provide a sound base logic that can be reused and extended to fit all needs. It defines *resource algebras* (RAs), objects that are similar to separation algebras. Because Iris supports "higher-order ghost state", more sophisticated mechanisms are needed to model state, and as such the composition function $\cdot$ needs to be total. To rule out invalid compositions, a validity function $\overline{\mathcal{V}}$ is instead used, which returns whether a given state is valid. The core function, written $|-|$, is made *partial*, rather than total – this, they argue, makes constructing state models from smaller components easier. This was confirmed when developing state models for Gillian, where having the core be a total function made some state models more complicated to make sound, as will be shown later.

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, such as *Gillian* [26, 45, 59]. It is a *compositional symbolic execution engine*, with the added property of being *parametric on the memory model*. It allows reasoning about *correctness and incorrectness*. We may now go over exactly what this means.

2.2.1 Symbolic Execution

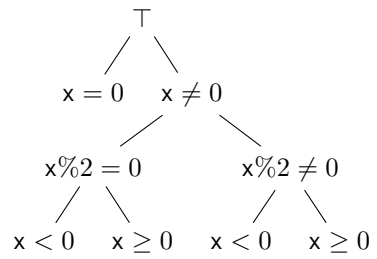
Traditionally, software is tested by calling the code with a predetermined or randomly generated input (for instance with fuzzing), and verifying that the output is as expected. These approaches, where the code is executed directly, are in the realm of *concrete* execution, as the code is run with concrete (i.e. 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 when generated. With fuzzing, methods exist to improve the odds of finding new paths [5], 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 [4]. These values are abstract, and are 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 [12] and EFFIGY [36] 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.[1] This would then result in 5 different branches, each with different constraints on the program variable x: $x = 0$, $x\%2 = 0 \wedge x < 0$, $x\%2 = 0 \wedge x > 0$, $x\%2 \neq 0 \wedge x < 0$ and $x\%2 \neq 0 \wedge x > 0$.[2]

```
1  if (x == 0) {
2    print("zero");
3    return;
4  }
5  if (x % 2 == 0) {
6    print("even");
7  } else {
8    print("odd");
9  }
10 if (x < 0) {
11   print("negative");
12 } else {
13   print("positive");
14 }
```

(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

2.2.2 Compositional Symbolic Execution

An issue that arises from symbolic execution is that is scales quite poorly, because of path explosion [1, 4], as each branching can potentially multiply by two the total number of branches. Furthermore, it lacks compositionality, as the code must be tested in its entirety, from `main` to `return`.

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

A solution to this is *compositional* symbolic execution (CSE), in which the code is tested by fragments, function by function. This allows for gradual adoption, as only parts of the code base can be tested. This minimises the effect of path explosion, since it limits the amount of branching that may occur.

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 [6], and was followed by later tools, such as JaVerT [28, 29], Infer [14], VeriFast [31], SpaceInvaders [61] 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 (also called *state model*). Typical CSE engines usually come with a built-in notion of memory, ranging from simple heaps with memory chunks [31] to more complex state models tailored to a language, like in JaVerT [28, 29] to represent JavaScript objects.

While a fixed state 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 allow it to be used for virtually any language, as long as the language's state model can be encoded into a separation logic equivalent; improvements done to the engine can then be carried to all of its instantiations for free. 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 [41]. 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. The language Gillian targets, GIL (Gillian's Intermediate Language), 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 specifica-

tions. For example, the separation logic $a \mapsto x$ predicate could be represented as a core predicate as $\langle \texttt{points\_to} \rangle(\texttt{a}; \texttt{x})$.

2.2.4 Correctness, Incorrectness

Another key difference between Gillian and other CSE engines is that it is capable of reasoning about programs both with correctness, in SL, and incorrectness, in ISL, thus enabling OX program verification and UX true bug-finding respectively. It also supports whole-program symbolic testing, as a non-compositional engine would.

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 [46] semantics of GIL, enabling automated proofs to be done both with forwards and backwards implications, representing OX and UX respectively.

Thanks to this and the aforementioned parametricity of the memory model, Gillian is therefore 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 prohibitively 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 [38] is a bounded model checker, that allows for the verification of C programs, via whole program symbolic testing – that is, it is not compositional, and instead symbolically executes a given codebase, and verifies that assertions hold on a generated formula. 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 [13] 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 does whole program testing, and as such doesn't scale with larger codebases. MACKE [51] 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, enabling better scaling and integration in CI pipelines, which allow rapidly giving 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 [6] was the first tool to enable compositional symbolic execution with SL, jStar [21] 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 with verification. It uses an intermediate representation of Java called Jimple, taken from the Java optimisation framework Soot. Similarly, VeriFast [31] is a CSE tool that is targeted at Java and a subset of C, while SpaceInvader [61] allows for the verification of C code in device drivers.

The Verified Software Toolchain [47] also allows verifying correctness of C code using SL; work has been done recently to bring some ideas from Iris such as ghost state into the toolchain. This makes it the only example of engines using Iris.

Infer [14] 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* [15], a technique enabling execution to proceed despite resources missing, via the construction of an anti-frame. Its approach, while initially stemming from separation logic, was used for finding bugs rather than proving correctness, which led to the creation of ISL [57] to provide a sound theory justifying this approach. Following this, Pulse [39] 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 [28, 29] is a CSE engine aimed at verifying JavaScript, with support for whole program testing, verification, and bi-abduction in the same fashion as Infer – it followed from Cosette [27], 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 this end, 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, at the cost of requiring a compiler to the intermediate language.

For instance Viper [49] is an infrastructure capable of program verification with a permission-based separation logic, akin to what is presented in [9]. It's intermediate language is a rich object-oriented imperative typed language, that operates on a built-in heap. A range of Viper frontends exist, allowing for the verification of Scala, Java and OpenCL.

One can still take genericity further, by also abstracting the state model the engine operates on. A tool that does this is coreStar [11], the CSE engine backing jStar with all the Java-related components removed and replaced. It does not target Java, but instead a generic intermediate language. Furthermore, instead of coming with predicates predefined, 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 our knowledge the first example of a tool that allows some flexibility in the underlying logic theory.

This is also what Gillian [2, 26, 45, 59] does, a CSE engine parametric on the state model. 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, its intermediary language. Currently Gillian has been instantiated to the C language (via CompCert-C), JavaScript, and Rust [3]. It is a generalisation of JaVerT, which only targeted JavaScript. Following Gillian, work has also been done on formalising a theoretical CSE engine, providing axioms an implementation must follow [43, 44] as well as an implementation in Rocq of the engine to prove its soundness.

In Table 2.1 we show a side-by-side comparison of the aforementioned tools.

Table 2.1: Comparison of verification tools

	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

2.4 Legal, Social, Ethical and Professional Requirements

There aren't particular legal, social or ethical considerations associated with compositional symbolic execution. To the contrary, improving code verification and facilitating its use and adoption may result in positive outcomes in a wide range of fields where programs are critical. This includes, among others, medical, nuclear and spatial fields, where program failures can cause life loss, catastrophic environmental degradation, or significant economic loss. For instance, the infamous Therac-25 radiation therapy machine caused the death of four patients and the injury of two others between 1985 and 1987 due to an error when handling shared mutable state [42]. This could have been detected and avoided if proper verification of the program had been done.

Verification tools have the potential to cause harm, if used on open source software by malevolent users that want to find possible exploits.[3] Exploits such as Heartbleed [19] or more recently the `xz` backdoor [20] happened on Open Source code, and an attacker could very well automate verification of these libraries privately in an effort to find new exploits. It is thus important to facilitate and democratise verification tools to counteract these actors.

[3]Though, to our knowledge, no exploit was found this way.

Chapter 3

Theory

In this chapter we will cover the theoretical foundations for Resource Algebras in a CSE setting. We will first cover in §3.1 the current state of the art regarding state models, then in §3.2 what modifications are needed for a CSE to handle RAs. This is followed by a list of simple state models in §3.3, that can be used to construct more complex state models via transformers, as shown in §3.4. Finally, in §3.5 we look at the theory justifying some optimisations of the PMap state model transformer.

3.1 State of Affairs

3.1.1 State Models with PCMs for CSE

Abstract separation logic historically has mostly used Partially Commutative Monoids (PCMs) [8, 16, 22, 35]. These are defined as a tuple $(M, (\cdot) \colon M \times M \rightharpoonup M, 0)$, corresponding to the carrier set, composition operator, and a unique identity (or unit) element, which must satisfy the properties highlighted in Figure 3.1. PCMs are useful for representing state in SL, as they "mathematically represent the essential notions of state ownership and ownership transfer" [25], while being simple and abstract enough that they can represent a variety of concrete states (the traditional case being that of heaps).

When bringing this concept to a Compositional Symbolic Execution (CSE) engine, the PCM must be endowed with a set of actions $\mathcal{A}$, of core predicates Δ, and functions that allow modifying it; in particular, an `execute_action` function that reflects program commands that modify the state, a `produce` function

$$\forall a, b, c.\ (a \cdot b) \cdot c = a \cdot (b \cdot c) \qquad \text{(PCM-Assoc)}$$
$$\forall a, b.\ a \cdot b = b \cdot a \qquad \text{(PCM-Comm)}$$
$$\forall a.\ a \cdot 0 = a \qquad \text{(PCM-Identity)}$$

Equality means either both sides are defined and equal, or both are undefined.

Figure 3.1: Properties of Partially Commutative Monoids.

Elements:

- $(\Sigma, (\cdot) : \Sigma \times \Sigma \rightharpoonup \Sigma, 0)$: PCM of concrete states.
- $\hat{\Sigma}$: Set of symbolic states.
- $\mathcal{A}$: Set of actions.
- Δ: Set of core predicates.

Functions:

- `execute_action`: execute an action with arguments, returns a value.
- `produce`: add a core predicate assertion into the state, returns the new state.
- `consume`: removes a core predicate assertion from the state, returns the new state and the outs of the assertion.
- `fix`: returns the assertions needed to fix the given missing error.

Theoretical Properties: *Is 'properties' the right word?*

- $\models$: symbolic interpretation relation.
- $\models_\Delta$: core predicate satisfiability relation.

Figure 3.2: High-level description of the elements of a state model

to add an assertion to the state, a `consume` function to remove an assertion from the state, and a `fix` function to provide fixes for missing errors in bi-abduction. In [2, 43, 44], the notion of state model is thus introduced, to represent a PCM equipped with these attributes, which can be seen in Figure 3.2. A state model is denoted $\mathbb{S}$.[1]

All of these together enable a CSE engine to be *parametric* on the state model; as long as the provided functions follow a set of axioms, the engine using the state model can be proved to be sound, either in over-approximate (OX) or under-approximate (UX) mode, in turn allowing for program verification, or true bug finding.

Core predicates (sometimes simply called predicates) are used to make the assertion language of the engine parametric. They take inspiration from [16], and are written $\delta \in \Delta$. To enable this parametricity, the assertions of the language (see Figure 3.3) are extended with a core predicate assertion, written $\langle \delta \rangle(\vec{v}_i; \vec{v}_o)$, where δ is the core predicate, $\vec{v}_i$ are the *in-values* (or "ins") of the predicate, and $\vec{v}_o$ are the *out-values* (or "outs"). This distinction is used for automation purposes, to create matching plans: they tell the engine that given the values $\vec{v}_i$, the state can return the corresponding $\vec{v}_o$ associated with it; this is akin to the "parameter modes" described in [50]. For instance, the traditional "points to" assertion $a \mapsto v$, where a is an address and v a value, can be written as $\langle \texttt{points\_to} \rangle(a; v)$: given the address a, a heap can return the value store at the location v. Core predicate also imply a satisfiability relation, denoted $\sigma \models_\Delta \langle \delta \rangle(\vec{v}_i; \vec{v}_o)$ to signify a state satisfies a given predicate. Predicate satisfiability happens at the concrete level, and is later lifted to the symbolic realm by evaluating the ins and outs using a substitution θ and variable store s; expression evaluation is denoted $[\![\hat{e}]\!]_{\theta,s} = e$. Another traditional example is the *emp* assertion, that is satisfied by the empty memory; it can be defined as a core predicate: $\langle \texttt{emp} \rangle(;)$.

[1] In our definition, the path condition and the variable store are *not* part of the state model. These are handled by the engine "on top" of the state model.

$$
\begin{aligned}
&\hat{v} \in \text{LVal} \\
&\hat{x} \in \text{LVar} \subset \text{LVal} \\
&\delta \in \Delta \\
P, Q, ... \in \text{Asrt} \triangleq\ & \hat{v} \mid \texttt{true} \mid P \Rightarrow Q \mid P \vee Q \mid \exists \hat{x}.\, P \mid \\
& P * Q \mid\ \langle \delta \rangle(\vec{\hat{v}}_i; \vec{\hat{v}}_o)
\end{aligned}
$$

Figure 3.3: Assertion language

Remark (Semantics of emp)**.** *There are two accepted and clashing semantics of the emp assertion; either it is equivalent to* `true` *and any state satisfies emp, or it represents strictly the empty memory and nothing else – the former is more appropriate for garbage collected languages while the latter for* `alloc/free` *languages [17].*

Here we make the arbitrary choice of defining it as only being satisfied by the empty memory. Because the engine is parametric on the core predicates, a user is free to chose the semantics appropriate to their use case – this is a strength of the parametric approach.

Actions are used to represent program actions that interact with the state. Typical actions include `load`, `store`, `alloc` or `free`. These actions are called with a list of arguments, return a list of results, and may modify the state, as long as the modification are sound with respect to a set of axioms, presented later.

We may now look at an example of function, to understand how the state model is used. Consider the function `set_value` which has the following specification:

$$
\{\!\!\{ \texttt{a} = \hat{a} * \texttt{x} = \hat{x} * \langle \texttt{points\_to} \rangle(\hat{a}; \hat{v}) \}\!\!\}\ \texttt{set\_value(a, x)}\ \{\!\!\{ \texttt{ret} = \hat{a} * \langle \texttt{points\_to} \rangle(\hat{a}; \hat{x}) \}\!\!\}
$$

If the engine aims to verify the function, it will start with an empty state and *produce* all assertions of the pre-condition (the order being determined by the matching plan). Here, this means adding the assertions $\texttt{a} = \hat{a}$ and $\texttt{x} = \hat{x}$ to the path condition, and producing the core predicate $\langle \texttt{points\_to} \rangle(\hat{a}; \hat{v})$ into the state. It will then execute the action `store` on the state model, with arguments $[\hat{a}, \hat{x}]$ (note the program variables are substituted with the symbolic variable associated). In the linear heap, this simply modifies the value stored at $\hat{a}$, setting it to $\hat{x}$. The code then returns `a`, thus assigning its value, $\hat{a}$, to the special variable `ret` which represents the value returned by the function. Finally, to ensure the state at the end of the function matches the postcondition, the assertions are consumed; first asserting $\texttt{ret} = \hat{a}$ holds, and then consuming $\langle \texttt{points\_to} \rangle(\hat{a}; \hat{x})$, effectively removing the binding from the state. All the above executes succesfully, and the function is thus verified.

The same also applies with more complex functions. For the case of function calls using function specifications, the inverse is done: when calling a function its precondition is consumed, and its postcon-

dition is produced, effectively *framing off* the corresponding state, and then framing back on the post condition.

3.1.2 Where it goes wrong

While PCMs are an obvious and straightforward choice for representing state ownership, they come with certain drawbacks. Firstly, PCMs can be overly restrictive; in particular, the presence of a unit 0 is restrictive, as it makes construction of sums impossible: for that, one needs to allow for multiple units in the object. In [22], it is shown that where a PCM fulfills $\exists u.\, \forall x.\, x \cdot u = x$, one can allow for more units by simply swapping the existentials: $\forall x.\, \exists u.\, x \cdot u = x$. This rule is further relaxed in Iris [34], where it is shown that not having a unit at all can also be useful, in particular when one wants to construct *Resource Algebras* (RAs) from simpler elements. Finaly, and as noted in [34], PCMs are "not enough" for certain applications, such as when wanting to support higher-order logic.

In [2], the concept of RA constructions from [34] is brought into a CSE setting, using PCMs. The original work however contained some errors in constructions, that created unsoundness in the verification tool. These errors stemmed, from the most part, to the use of PCMs, and the existence of the unit 0. Indeed, this made some constructions, such as sum, freeable or the partial map unsound. We will now look at examples of these.

Unsoundness in Freeable

The first example is with $\textsc{Freeable}_{\text{PCM}}(\mathbb{S})$. This state model *transformer* is given a state model $\mathbb{S}$, and extends it with the capacity to free state, via the `free` action. It also adds a `freed` predicate, to represent freed memory.

Now consider the simple state model $\textsc{Freeable}_{\text{PCM}}(\textsc{Ex}_{\text{PCM}}(\mathbb{N}))$. Here $\textsc{Ex}_{\text{PCM}}(X)$ is the exclusive state model, that represents exclusive ownership of a value of type X[2]. Ownership is indicated via the `ex` core predicate: $\langle \mathsf{ex} \rangle(; x)$ means the state currently has the value x – this state model will be further defined later, and is only used here for the example. For now, we define the PCM version of $\textsc{Ex}_{\text{PCM}}(X)$ as:

$$\textsc{Ex}_{\text{PCM}}(X) \triangleq \text{ex}(x \colon X) \mid 0$$

The notation $\text{ex}(x \colon X)$ is equivalent to $\{\text{ex}(x) : \forall x \in X\}$. We thus define $\textsc{Ex}_{\text{PCM}}(X)$ as the set of elements of X *and* the 0 element. We also define composition such that $0 \cdot \sigma = \sigma \cdot 0 = \sigma$ and $\text{ex}(x) \cdot \text{ex}(y)$ is always undefined: since the value is owned exclusively, we own it as a whole and nothing can be added to it. This satisfies the axioms of PCMs. We now consider a naive definition of $\textsc{Freeable}_{\text{PCM}}(\mathbb{S})$'s PCM,

[2]Note we write $\textsc{Ex}_{\text{PCM}}(X)$ and $\textsc{Freeable}_{\text{PCM}}(\mathbb{S})$ – the $\textsc{Ex}_{\text{PCM}}$ state model accepts a set, X, whereas $\textsc{Freeable}_{\text{PCM}}$ accepts a state model, $\mathbb{S}$.

where Σ is the carrier set of $\mathbb{S}$'s PCM:

$$\textsc{Freeable}_{\text{PCM}}(\mathbb{S}) \triangleq \text{sub}(\sigma \colon \Sigma) \mid \text{freed} \mid 0$$

We then again define composition as $0 \cdot \sigma = \sigma \cdot 0 = \sigma$, and $\text{sub}(\sigma) \cdot \text{sub}(\sigma') = \text{sub}(\sigma \cdot \sigma')$, which forms a valid PCM.[3] $\textsc{Freeable}_{\text{PCM}}$ exposes a core predicate `freed`, as well as the core predicates of the underlying memory model; similarly, `consume` and `produce` are lifted. While we omit the full details of the state model, we note that producing `freed` only succeeds when in state 0 (since a state can't be both freed and something else), producing something other that `freed` calls the `produce` function of the underlying state model $\mathbb{S}$, and stores the resulting state σ as $\text{sub}(\sigma)$. Similarly, consuming a predicate of $\mathbb{S}$ while in state $\text{sub}(\sigma)$ calls `consume` on that predicate with σ, and then wraps the result σ' back into $\text{sub}(\sigma')$.

To show the unsoundness of this definition, consider calling the `dispose` function:

$$\langle\!\langle \langle \texttt{ex} \rangle(; \hat{x}) \rangle\!\rangle \; \texttt{dispose()} \; \langle\!\langle \langle \texttt{freed} \rangle(;) \rangle\!\rangle$$

Assume we're in the state $\text{sub}(\text{ex}(1))$. To call `dispose` by its specification, we first need to consume its precondition: we consume $\langle \texttt{ex} \rangle(; \hat{x})$ from the state. To do this, $\textsc{Freeable}_{\text{PCM}}$ calls the `consume` function of $\textsc{Ex}_{\text{PCM}}(\mathbb{N})$, with the state $\text{ex}(1)$, which returns 0. $\textsc{Freeable}_{\text{PCM}}$ then wraps this state back, resulting in $\text{sub}(0)$. We then produce the post-condition of `dispose`, thus producing $\langle \texttt{freed} \rangle(;)$ into $\text{sub}(0)$. This however is not valid, according to the rules outlined earlier! Indeed the state can't both be freed and something else. Of course, $\text{sub}(0)$ represents an empty state, but that means that the `consume` and `produce` rules of $\textsc{Freeable}_{\text{PCM}}$ must take this into account – a simple implementation could miss it, resulting in such errors.

This difficulty is in fact caused by the existence of both 0 and $\text{sub}(0)$ as two distinct states, despite them being the same semantically: empty. To fix this, one must either make $\text{sub}(0)$ invalid, defining it as $\text{sub}(x \colon \Sigma \setminus \{0\})$, or instead remove the definition of 0 and instead use $\text{sub}(0)$ as the unit of $\textsc{Freeable}_{\text{PCM}}$. Either cases require one to be quite careful around the unit, as forgetting about it may lead to unwanted errors.

Unsoundness in Sum

The sum state model $\textsc{Sum}(\mathbb{S}_1, \mathbb{S}_2)$, also written $\mathbb{S}_1 + \mathbb{S}_2$, represents a state where either one of the two sides is owned, but not both. It allows representing states with transitions – for instance, $\textsc{Freeable}(\mathbb{S})$ can also be defined as $\mathbb{S} + \textsc{Ex}(\{\text{freed}\})$.

While this state is useful in that it allows modelling transitions from one side of the sum to another, it is in fact non-trivial to define in a way that is both OX-sound and UX-sound. In particular, we must

[3] All other compositions are undefined.

enforce three properties: neither sides of the sum must permit the 0 element, if a state σ allows swapping no other smaller state $\sigma' \preccurlyeq \sigma$ must allow swapping, and the state swapped into must be *exclusively owned*. This last property is taken from [34], and for PCMs can be defined as:

$$\texttt{is\_exclusively\_owned}_{\text{PCM}}\ \sigma \triangleq \forall \sigma'.\, \sigma' \neq 0 \implies \neg(\sigma \# \sigma')$$

The proof that the above properties must hold can be found in **??**. Again we note how the presence of a 0 adds requirements around these constructions, cluttering otherwise elegant definitions.

Unsoundness in PMap

Finally, we consider how PCMs can cause unsoundness in the partial map state model transformer, $\text{PMap}(I, \mathbb{S})$. It allows mappings from a domain I to a codomain state from $\mathbb{S}$. For this example we consider the state model construction $\text{PMap}_{\text{PCM}}(\mathbb{N}, \text{Ex}_{\text{PCM}}(\mathbb{N}))$, and we define the PCM of PMap_{PCM} as:

$$\text{PMap}_{\text{PCM}}(I, \mathbb{S}) \triangleq (I \xrightarrow{fin} \Sigma) \times \mathcal{P}(I)^?$$

We omit the precise details of how exactly it works – it will be extensively discussed in §3.4. $\text{PMap}_{\text{PCM}}(I, \mathbb{S})$ *lifts* all predicates of $\mathbb{S}$ with an additional in-value corresponding to the location $i \in I$ to which the predicate corresponds. All action executions receive an additional argument with the location, and similarly all `consume` and `produce` calls must have the index in their ins. The behaviour of a naive implementation would then be similar to what was explained for $\text{Freeable}_{\text{PCM}}$: calling `consume` or `produce` when the binding is not present executes it against the 0 element of the underlying PCM, and otherwise calls it on the state at the location and stores the result back at that location, overriding the previous binding. The right hand side of the product, $\mathcal{P}(I)^?$, represents the domain set of the partial map, in other words the set of addresses that are known to exist – any address outside of the set does not exist. The $-^?$ signifies that we extend $\mathcal{P}(I)$ with the element $\bot$, in which case the domain set is not known. The core predicate $\langle \texttt{domainset} \rangle(; d)$ is satisfied by the state with the domain set d. The `produce` function of PMap_{PCM} also ensures that a well-formedness constraint is upheld; in particular, given a state (h, d) with h the heap and d the domainset, if $d \neq \bot$, then $\text{dom}(h) \subseteq d$.

Now, consider the following function specification, that moves a value:

$$\{\!\{\langle \texttt{ex} \rangle(1; 1) * \langle \texttt{domainset} \rangle(; \{1\})\}\!\}\ \texttt{move()}\ \{\!\{\langle \texttt{ex} \rangle(2; 1) * \langle \texttt{domainset} \rangle(; \{2\})\}\!\}$$

Let the initial state be $(\{1 \mapsto \text{ex}(1)\}, \{1\})$: a heap with one cell, at address 1 – this matches the precondition of `move`. The engine first consumes the `ex` predicate from ex(1), at address 1 – this means the pointed-to state becomes 0, and the outer state becomes $(\{1 \mapsto 0\}, \{1\})$. The `domainset` predicate is

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

$$\text{where} \quad M^? \triangleq M \uplus \{\bot\}, \text{ with } a \cdot \bot \triangleq \bot \cdot a \triangleq a$$
$$a \preccurlyeq b \triangleq \exists c.\, b = a \cdot c$$
$$a \rightsquigarrow B \triangleq \forall c^? \in M^?.\, \overline{\mathcal{V}}(a \cdot c^?) \Rightarrow \exists b \in B.\, \overline{\mathcal{V}}(b \cdot c^?)$$
$$a \rightsquigarrow b \triangleq a \rightsquigarrow \{b\}$$

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

$$\overline{\mathcal{V}}(\epsilon) \qquad \forall a \in M. \epsilon \cdot a = a \qquad |\epsilon| = \epsilon$$

Figure 3.4: Definition of resource algebras in Iris

then successfully consumed too, leaving $(\{1 \mapsto 0\}, \bot)$. Now, the post-condition is produced onto the state. First, the engine produces $\langle \mathtt{ex} \rangle(2; 1)$, which adds a binding to PMAP_{PCM} and creates the cell: our bigger state is now $(\{1 \mapsto 0, 2 \mapsto \text{ex}(1)\}, \bot)$. Finally, $\langle \mathtt{domainset} \rangle(; \{2\})$ is produced onto the state, however this *doesn't succeed*. Indeed, we have $\text{dom}(\{1 \mapsto 0, 2 \mapsto \text{ex}(1)\}) = \{1, 2\}$, and of course $\{1, 2\} \not\subseteq \{2\}$: the domain of the heap is not a subset of the produced domain set $\{2\}$. This is caused because again we forgot to check if the state at location 1 became 0 when consuming ex(1). We would thus need to check for this, and potentially redefine the PCM of PMAP_{PCM} to forbid mappings to 0.

These examples hopefully showed that PCMs enforce a unit that is unwieldy, and can easily cause to unsound behaviour when forgotten. Even when taken into account, it leads to unpleasant definitions, as every state model must define a 0, and every state model transformer must then either exclude it in its construction, or explicitly take it into account.

3.1.3 RAs for Iris

Iris [33–35, 37] departs from the tradition of PCMs, and introduces Resource Algebras (RAs) to model state, defined as a tuple $(M, \overline{\mathcal{V}}\colon M \to \text{iProp}, |-|\colon M \to M^?, (\cdot)\colon M \times M \to M)$, containing the carrier set, a validity function, a partial core function and a *total* composition operator. This definition makes use of the option RA $M^? \triangleq M \uplus \{\bot\}$ [7], that extends the set of M with an "empty" element $\bot$, such that $m \cdot \bot = \bot \cdot m = \bot$. Similarly to PCMs, a set of axioms must be satisfied, as seen in Figure 3.4.

The *core function* $|-|\colon M \to M^?$ associates to an element m of the RA it's *duplicable core*. The crucial difference here, compared to PCMs, is that aside from allowing multiple units, it allows having *no unit*, in which case $|m| = \bot$. As we will see later, having no unit is a requirement for a state to be

exclusively owned (i.e. it has no frame). Indeed, if the state m has a core $|m| \neq \bot$, then that core always constitutes a valid frame.

The *validity function* $\overline{\mathcal{V}}\colon M \to \text{iProp}$ is used to rule out invalid compositions, rather than relying on the partiality of the composition operator. It returns an Iris property, iProp, which allows for step indexing and some more sophisticated logic, that it out of the scope of this project. As such, most RAs are extended with an invalid element $\lightning$, and validity is often defined as $\overline{\mathcal{V}}(m) \triangleq m \neq \lightning$. For instance for the Ex_{IRIS} RA:

$$\text{Ex}_{\text{IRIS}}(X) \triangleq \text{ex}(x\colon X) \mid \lightning$$
$$\overline{\mathcal{V}}(a) \triangleq a \neq \lightning$$
$$|\text{ex}(a)| \triangleq \bot$$
$$a \cdot b \triangleq \lightning$$

The validity and core functions thus make Iris states more powerful, in that they have more flexibility in what can be expressed; while a regular PCM can be trivially converted to an RA, the opposite is not true. An example of construction that can easily be more easily constructed is the sum state model. The requirement for a transition from one side of the sum to the other can be more elegantly expressed, without having to worry about the existence of 0s [34]. Given $\text{exclusive}(a) \triangleq \forall b.\, \neg\overline{\mathcal{V}}(a \cdot b)$, we have:

$$\frac{\text{exclusive(a)} \qquad \overline{\mathcal{V}}(b)}{a \rightsquigarrow b}\ \textsc{ExclusiveUpdate}$$

The Iris udpate $a \rightsquigarrow b$ is a form of frame preserving update: if we update a into b, we know that any frame that was compatible with a remains compatible with b.

Furthermore, Iris is a battle-tested logic,[4] that comes with plenty of proofs and properties making them easy to use and adapt, whereas PCMs can prove unwieldy even for simpler state models, as shown previously.

We note an interesting similarity between Iris and the PCM approach however, which is that "the global RA must be unital, which means it should have a unit element ϵ" [34]. This is similar to what one would find in a PCM, where we have the 0 element. Any RA can be trivially extended to have a unit, via the option RA. This in particular means that while the building blocks of both approaches are differents (PCMs or RAs), the end result that is used is similar. This is good news: it means that adapting an RA construction to a CSE engine that uses PCMs should be fairly straightforward, as one can rely on the added unit as a de facto 0.

[4] See https://iris-project.org/#publications, where more than a hundred publications are cited as using Iris, allowing for the verification of, among others, C, Go, OCaml, Rust, Scala or WASM.

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

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

where $M^? \triangleq M \uplus \{\bot\}$, with $a \cdot \bot \triangleq \bot \cdot a \triangleq a$
$a \preccurlyeq b \triangleq \exists c.\, b = a \cdot c$
$a \# b \triangleq a \cdot b$ 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 3.5: Definition of (partial) resource algebras

3.2 State Models with RAs

We have seen how PCMs are unpractical for a CSE engine, as they easily create incorrect behaviour that can be quite tricky to notice. We will now see how we may adapt a CSE to instead use RAs. We first define the notion of partial RAs, followed by the description of the changes done to the engine's layers to support partial RAs. Finally, we list the new axioms that must be respected by the functions of the state model.

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-indexing, this doesn't come into play when one is only manipulating RAs. As such, because having to define and later verify the validity of states can be quite unwieldy, we remove it by instead making use of partiality, as was originally the case in PCMs. This translation is done by removing the invalid state $\lightning$ when present, and instead defining compositions that lead to it as undefined. From this, we can get rid of the validity function, and define composition as being partial: $(\cdot)\colon M \times M \rightharpoonup M$. This is also in line with the core function $(-)$ being partial.

Remark (Partiality and the option RA)**.** *One may note that we use the term "partial" to refer to two different signatures; the core $|-|\colon M \to M^?$ is called partial but is in fact total, with undefined mappings being equal to $\bot$, while the composition $(\cdot)\colon M \times M \rightharpoonup M$ is an actual partial function.*

While one could adapt the core to be $M \rightharpoonup M$, this would prove unpractical, in cases where one still wants to use $|m|$ even when it is $\bot$. On the other hand, defining composition as $M \times M \to M^?$ is simply incorrect: let $a \cdot b$ undefined, it is untrue that $(a \cdot b) \cdot c = c$, while defining an undefined composition to equal $\bot$ would render this statement true.

Partial RAs are equivalent to regular RAs, as long as $\neg\overline{\mathcal{V}}(a) \Rightarrow \neg\overline{\mathcal{V}}(a \cdot b)$ holds; luckily for us, this is equivalent to the axiom Iris-RA-Valid-Op. Compositions that yield $\lightning$ (or any state m such that $\neg\overline{\mathcal{V}}(m)$) 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 change is that because validity is replaced by the fact composition is defined, the validity of a composition $\overline{\mathcal{V}}(a \cdot b)$ in a RA is exactly equivalent to the fact two elements are disjoint in a partial RA $a \# b$. We now define the properties of partial RAs taking this change into account – see Figure 3.5. From now, the term RA will be used to refer to these partial RAs.

We may compare the definition of Ex_{Iris}, presented earlier, with the equivalent definition using partial RAs, Ex. The main difference is the removal of $\overline{\mathcal{V}}$ and the $\lightning$ element, making the definition simpler.

$$\text{Ex}_{\text{Iris}}(X) \triangleq \text{ex}(x\colon X) \mid \lightning$$
$$\overline{\mathcal{V}}(a) \triangleq a \neq \lightning$$
$$|\text{ex}(a)| \triangleq \bot$$
$$a \cdot b \triangleq \lightning$$

$$\text{Ex}(X) \triangleq \text{ex}(x\colon X)$$
$$|\text{ex}(a)| \triangleq \bot$$
$$a \cdot b \text{ is always undefined}$$

3.2.2 Core Engine

We're now interested into what changes need to be brought to our CSE to handle RAs. To do this, we use the axioms and definitions specified in [43]. We will also specify additional changes that exist in Gillian and that were formalised in [2], but that haven't been added to the aforementioned formalisation.

This CSE engine is split into three layers, each adding functionality to the layer below, allowing for a clear separation of concerns. These layers are, from bottom to top: the core engine, the compositional engine, and the bi-abduction engine (see Figure 3.6).

The core engine enables whole-program symbolic execution. For this, state models must firstly define the set of states the execution will happen on. While *concrete* compositional state is modelled as an RA, *symbolic* compositional state is simply a set of elements that does not come with composition or cores. This is because when one handles symbolic values, composition isn't a partial function anymore, as it can branch, and thus output more than one value. Take for instance the RA of trees, where nodes are defined as having an offset and size. Composing two trees made of a single node with a symbolic offset may lead to different results, depending on the interpretation of the offsets: are the two nodes adjacent? Which is on the left or the right? This cannot be modelled with an RA. Soundness of the engine is instead

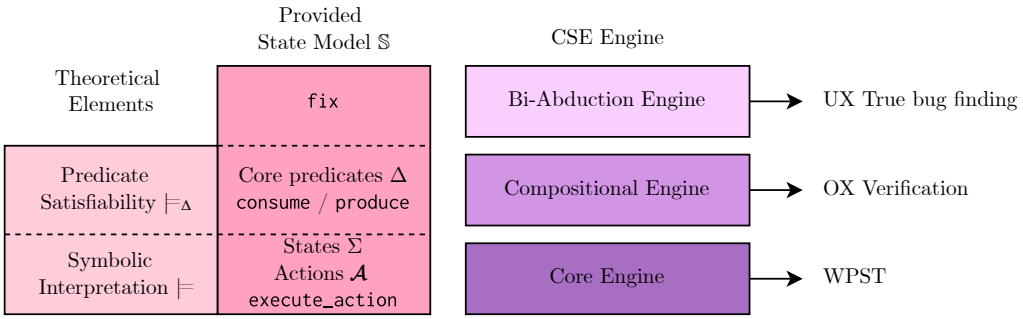


Figure 3.6: Layers of the CSE engine

verified by unlifting symbolic states to concrete states via symbolic interpretation ($\models$), and then using the properties of these concretes states, which do form a valid RA.

Once the set of symbolic states $\hat{\Sigma} \ni \hat{\sigma}$ is defined, it must be further equipped with a set of actions $\mathcal{A}$ and an `execute_action` function that allows the program to modify the state. We also include the definition of the $\models$ relation, called `sat`, noting that this is a purely theoretical construct that isn't implemented in the actual engine.

$$\texttt{execute\_action}\colon \Sigma^? \to \mathcal{A} \to \text{Val } \texttt{list} \to \mathcal{O}_e \times \Sigma^? \times \text{Val } \texttt{list}$$

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

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

We define two different versions of `execute_action`: one operating on concrete states, that is deterministic, and one operating on symbolic states, that allows branching. The concrete version of the function is used in the theory, to prove the soundness of its symbolic counterpart, while the symbolic version is what is actually used by the engine to execute the analysed program.

In the concrete case, the arguments of `execute_action` are, in order: the *optional* state the action is executed on, the action, and the received arguments. It returns an outcome, the new state and the returned values. It is pretty-printed as $\alpha(\sigma, \vec{v}_i) = (o, \sigma', \vec{v}_o)$.

For its symbolic counterpart, the arguments are lifted to the symbolic realm. It also returns a set of branches, along with a path condition (PC) π representing the condition for the branch to exist – if the PC is not satisfiable, the branch must be cut. 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 in the set of *execution outcomes* $\mathcal{O}_e = \{\texttt{Ok}, \texttt{Err}, \texttt{Miss}\}$. Where `Ok` represents succesful execution and `Err` program errors, `Miss` outcomes come from the compositional nature of the state. An action that leads to a `Miss` doesn't represent a bug in the code, but rather that additional state is needed to determine whether execution is succesful or not.

The main difference with [43] is that the state may be $\bot$, if the action is executed on empty state; this is made explicit from using $\hat{\Sigma}^?$ (rather than $\hat{\Sigma}$). 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.

We also modify the definition of a path condition. It is typically defined as a logical value LVal that must evaluate to a boolean, and which is extended via conjunction. Internally, the engine however still needs to split the conjunction into a list of terms to manipulate it. As such, to make the theory closer to what is effectively done within the engine, we make the choice of defining the path condition as a *list of logical values*: $\pi \in \Pi = \text{LVal list}$. Extension is then defined as concatening new logical values to the path condition, and strenghtening of the path condition can be trivially checked 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:

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

The original definition of the symbolic execute_action presented in [43] also has an "SV" argument, a set containing all existing symbolic variables. This set is used when an action creates a fresh symbolic variable, to ensure the variable is fresh. While necessary for proofs within the engine in Rocq, we omit it here, as it is only relevant for one action (alloc), and pen and paper proofs do not require the same rigidity as a Rocq proof.

Finally, the user must define the sat relation, relating concrete and symbolic states. It is pretty printed as $\theta, s, \sigma \models \hat{\sigma}$ for $\sigma \in \mathtt{sat}\ \theta\ s\ \hat{\sigma}$, meaning that given a substitution θ and a store s, the concrete state σ can be matched by a symbolic state $\hat{\sigma}$. The user must only define this relation for non-$\bot$ states; we can then lift it to the option RA, by simply adding that $\forall \theta, s.\, \mathtt{sat}\ \theta\ s\ \bot = \{\bot\}$. This relieves users from needing to take $\bot$ into account when defining $\models$ and from proving the 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 calls by specification. 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, a core predicate satisfiability relation $\models_\Delta$ (called $\mathtt{sat}_\Delta$) must be defined in the theory, to prove soundness of consume and produce.

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

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 $\bot$ (as there must be a non-empty state to consume an assertion from or produce an assertion into), 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 for 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 $\hat{x} \mapsto 0$ in a state $[1 \mapsto 2]$ results in a new state $[1 \mapsto 2, \hat{x} \mapsto 0]$ with the path condition $[\hat{x} \neq 1]$. If the produced predicate is incompatible with the state (e.g. 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 $[1 \mapsto 2] \models_\Delta \langle\texttt{points\_to}\rangle(1; 2)$. This definition is similar to the definition of predicates in [16, 62], where a predicate is defined exactly as a set of states. We lift $\texttt{sat}_\Delta$ to the symbolic realm:

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

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 $1 \mapsto 1$ from a state $[1 \mapsto 2]$ would yield a `LFail`, while consuming it from state $[2 \mapsto 1]$ would yield a `Miss`, as the state $[1 \mapsto 1]$ could be composed with it to yield a non-miss outcome.

An addition to what [43] previously defined, taking inspiration from [2], is thus the split of what was the `Abort` outcome into `LFail` and `Miss`, which improves the quality of error messages and allows *fixing* consumptions that yield a `Miss` – this will be described in the next subsection.

Remark (`Miss` in `consume`)**.** *It is argued in [44] that* `consume` *shouldn't report* `Miss`*, as the missing resource might not actually be used by the function, which would then be UX unsound; we believe this preoccupation is not linked to* `consume` *but to function call semantics. If including* `Miss` *here is problematic,*

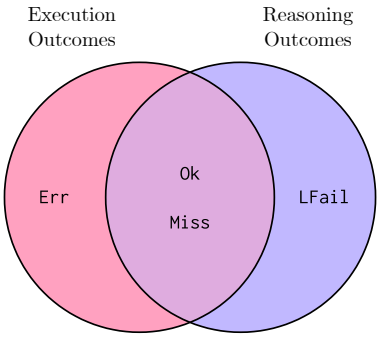


Figure 3.7: Overlap of outcomes

it should be up to the engine to correctly abort when doing function calls by specification in UX mode, rather to the `consume` *to unnecessarily limit its capabilities.*

We thus have $\mathcal{O}_e$ for execution outcomes, and $\mathcal{O}_l$ for reasoning outcomes – one is returned by action execution, and one for predicate consumption (see Figure 3.7). They overlap on one particular feature of the engine, which is function calls by specification: when calling a function, any outcome can happen: `Ok` if the function call succeeds, `Err` if the function call faults (as defined by its specification), `Miss` if additional resources are needed to satisfy the precondition of the function specification, and `LFail` if the state is incompatible with the function specification.

A last change done to what [43] defines is that the path condition input of `consume` and `produce` is removed, disallowing both functions from inspecting the path condition – the functions instead directly return the path condition required for the resulting branches. This brings several advantages: firstly, it simplifies the axioms for `consume` and `produce`, as there is no need to require that they strengthen the PC. Instead, the returned PC is always concatenated with the current PC, ensuring it can only be strengthened. Secondly, it simplifies the definition of `consume`, as whether the branch is cut or not (to satisfy `OX` and `UX` soundness) is also delegated to the engine. For instance, in UX all consumption branches that result in `LFail` can be dropped, as dropping branches is UX-sound – in OX however, this is not sound. A typical pattern in `consume` implementations is thus to evaluate the modified PC, and if an `LFail` is reached drop the branch depending on the mode. By omitting the PC from the arguments, this responsability is thus delegated to the engine. Finally, this makes a particular optimisation of the engine simpler to implement: incremental SMT solving.[5] This is a mechanism that allows storing an intermediary state in the SMT solver (a "checkpoint"), adding to the state, and later backtracking to the checkpointed state to evaluate another branch if needed. By ensuring `consume` and `produce` only return what to append to the PC, the checkpoint can easily be added before the call, and then each branch's PC added and evaluated. Allowing `consume` and `produce` to modify the PC directly would mean the PC must be parsed and filtered, to tell apart old from new terms – modified terms would make this even harder.

[5] Z3, which is used in Gillian, uses "scopes" for this – see https://microsoft.github.io/z3guide/docs/logic/basiccommands/#using-scopes

3.2.4 Bi-Abduction Engine

To support bi-abduction in the style of Infer:Pulse [39], `Miss` outcomes must be fixed. This enables true UX bug finding in under-specified or unspecified functions. For this, the state model must provide a `fix` function, that given the details of a miss error (these details being of type LVal `list` and returned with the outcome) returns a *list of sets of assertions* that must be produced to fix the missing error.

$$\texttt{fix}\colon \text{LVal } \texttt{list} \to \mathcal{P}(\text{Asrt}) \texttt{ list}$$

A *list* of different fixes is returned, 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, trying to load a cell that is not found in the state can lead to two possible fixes: either the cell is present, and the fix is $\{\langle \texttt{points\_to} \rangle(\hat{a}; \hat{x})\}$, either the cell is not present anymore because it has been freed (which would then lead to a use after free error), giving the fix $\{\langle \texttt{freed} \rangle(\hat{a};)\}$.

Furthermore, we define fixes as assertions rather than lists of core predicates. This is because a fix often contains *more than just spatial information*. The simplest example is the fix for a missing cell at address $\hat{a}$ in the linear heap: we must ensure the fix can be soundly applied to an existing state, without any variable clashes. For this, we must use an existential, giving us the fix $\exists \hat{x}.\, \langle \texttt{points\_to} \rangle(\hat{a}; \hat{x})$. More generally, it is sometimes needed that the fix contains logical information, for instance specifying constraints on a variable.

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 only mention the axioms related specifically to the state models – axioms relating to the overall soundness of the engine or the core semantics are out of the scope of this report.

For all of the axioms we assume we have a symbolic state model $\mathbb{S}$, made of the concrete states RA $\Sigma \ni \sigma$, symbolic states set $\hat{\Sigma} \ni \hat{\sigma}$, actions $\mathcal{A}$ and core predicates Δ.

We split the axioms into two parts: those relating to the symbolic nature of state and ensuring soundness with respect to a concrete state, and those relating to the parametricity of the state.

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.

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

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

The above two axioms are at the core of OX and UX soundness, respectively. The former ensures that if a concrete execution exists, then it must also exist in the symbolic semantics; it however doesn't ensure that more executions that those that actually exist may happen. The latter ensures that if a symbolic execution exists, then it must also exist in the concrete world; this time, no guarantees are made regarding the existence of such symbolic transition: for instance, symbolic semantics that never result in any branches are UX-sound (though they are of limited use).

Because incorrectness separation logic does execution backwards, we also prove UX soundness of the memory model backwards: if the resulting state has a concrete model, then so must the state before. In OX, soundness is instead forwards: if the starting state is soundness, so must be the resulting state.

Compositionality Axioms

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

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

The above two axioms are what enable the traditional frame rule of separation logic, but bringing it to operational semantics. Frame substraction in particular is almost identical to the "Frame property" defined in [62]: if executing an action on a state from which a frame was removed yields a non-`Miss`, then that action did not need that frame, and the frame can be added to the result. For frame addiction, which is needed in UX, we go backwards again: if a frame can be added to the state after executing an action and the action doesn't cause a `Miss`, then it can also be added to the state before without interfering with the action execution.

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

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

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

The above three axioms show the soundness of `consume` for OX and UX operations. They are similar to Memory Model OX Soundness and Memory Model UX Soundness respectively. We also have an additional requirement for OX execution, stating that if there is no erroneous execution of `consume` then at least one succesful execution exists.

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

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

Again, these two axioms show soundness of `produce` for OX and UX operation. For the former, we show that if two disjoint states exist in the concrete world and one of the two satisfies a core predicate, then producing that predicate onto the other yields their composition. For the latter, we state that if `produce` yields a branch that is satisfiable then there must exist a model of the result.

There are no axioms that `fix` needs to satisfy – as it simply results in assertions, it cannot break soundness. Either it produces no fixes or only incompatible fixes and we vanish, which is UX sound, or it finds fixes that work, in which case we already know the execution exists in the concrete world and we remain UX sound.

3.3 State Models

Now that the CSE engine is defined and the properties of state models are axiomatised, we focus on state model that can be later used to build up complex state. We first look at the base case, Ex, followed by

ExStoreOk
$\mathtt{store}(\mathrm{ex}(\hat{x}), [\hat{x}']) \rightsquigarrow (\mathtt{Ok}, \mathrm{ex}(\hat{x}'), [], [])$

ExStoreMiss
$\mathtt{store}(\bot, [\hat{x}']) \rightsquigarrow (\mathtt{Miss}, \bot, [], [])$

ExConsOk
$\mathtt{consume}(\mathrm{ex}(\hat{x}), \mathtt{ex}, []) \rightsquigarrow (\mathtt{Ok}, \bot, [\hat{x}], [])$

ExProd
$\mathtt{produce}(\bot, \mathtt{ex}, [], [\hat{x}]) \rightsquigarrow (\mathrm{ex}(\hat{x}), [])$

ExFix
$\mathtt{fix}\ [] = [\{\exists \hat{x}.\ \langle \mathtt{ex} \rangle(; \hat{x})\}]$

Figure 3.8: Some rules for Ex

CExStoreOk
$\mathtt{store}(\mathrm{ex}(x), [x']) = (\mathtt{Ok}, \mathrm{ex}(x'), [])$

ExStoreOk
$\mathtt{store}(\mathrm{ex}(\hat{x}), [\hat{x}']) \rightsquigarrow (\mathtt{Ok}, \mathrm{ex}(\hat{x}'), [], [])$

CExStoreMiss
$\mathtt{store}(\bot, [x']) = (\mathtt{Miss}, \bot, [])$

ExStoreMiss
$\mathtt{store}(\bot, [\hat{x}']) \rightsquigarrow (\mathtt{Miss}, \bot, [], [])$

Figure 3.9: Side by side comparison of concrete and symbolic action rules for Ex – concrete rules are prefixed with C.

the state model of knowledge or agreement, Ag. Finally, we look at an in-between, Frac, that allows exclusive ownership with a form of sharing.

3.3.1 Exclusive

The exclusive state model $\mathrm{Ex}(X)$ presented earlier is the simplest form of state model. It asserts ownership of an element of the set X, provides one core predicate `ex`, and two actions to modify the state, `load` and `store`. The `ex` predicate has no ins, and one out: the value stored.

$$\mathrm{Ex}(X) \triangleq \mathrm{ex}(x \colon X)$$

$$|\mathrm{ex}(x)| \triangleq \bot$$

$$\mathrm{ex}(x_1) \cdot \mathrm{ex}(x_2) \text{ is always undefined}$$

The first notation, $\mathrm{Ex}(X)$ is the state model instantiation from the set X to the Ex resource algebra. The notation $\mathrm{ex}(x \colon X)$ stands for $\{\mathrm{ex}(x) : \forall x \in X\}$ – here "$\mathrm{ex}(x)$" refers to the particular element of the $\mathrm{Ex}(X)$ RA where the value is $x \in X$.

See Figure 3.8 for an extract of the rules for Ex – the full definition can be found in **??**. We also show side by side the rules for the concrete and symbolic actions in Figure 3.9: in particular we note that they are strikingly similar, the latter simply being lifted from the former.

The only difference for Ex between RAs and PCMs is the absence of a 0, which is instead $\bot$; in fact $\mathrm{Ex}(X)^?$ is strictly equivalent to $\mathrm{Ex}_{\mathrm{PCM}}(X)$.

AgLoadOk
$\mathsf{load}(\mathrm{ag}(\hat{x}), []) \rightsquigarrow (\mathsf{Ok}, \mathrm{ag}(\hat{x}), [\hat{x}], [])$

AgConsOk
$\mathsf{consume}(\mathrm{ag}(\hat{x}), \mathsf{ag}, []) \rightsquigarrow (\mathsf{Ok}, \mathrm{ag}(\hat{x}), [\hat{x}], [])$

AgProdBot
$\mathsf{produce}(\bot, \mathsf{ag}, [], [\hat{x}]) \rightsquigarrow (\mathrm{ag}(\hat{x}), [])$

AgProdEq
$\mathsf{produce}(\mathrm{ag}(\hat{x}), \mathsf{ag}, [], [\hat{x}']) \rightsquigarrow (\mathrm{ag}(\hat{x}), [\hat{x} = \hat{x}'])$

Figure 3.10: Some rules for Ag

To provide at least one full of example of state model, we also include the definition of the symbolic interpretation and core predicate satisfiability relations – both are trivial, simply checking for equality:

ExSymInterpretation
$$\frac{[\![\hat{x}]\!]_{\theta,s} = x}{\theta, s, \mathrm{ex}(x) \models \mathrm{ex}(\hat{x})}$$

ExPredSat
$$\frac{\sigma = \mathrm{ex}(x)}{\sigma \models_\Delta \langle \mathsf{ex} \rangle([]; [x])}$$

3.3.2 Agreement

The agreement state model $\mathrm{Ag}(X)$ is the counterpart to the exclusively owned state model – it allows sharing, and is duplicable. It defines one predicate, ag, and a single action, load. Again, ag has no ins and one out, the value. Mutating state is not sound with a duplicable resource, as that doesn't satisfy Frame Addition or Frame Addition: a frame that is compatible with the original value is not compatible with the modified value, as they are not equal anymore.

$$\mathrm{Ag}(X) \triangleq \mathrm{ag}(x\colon X)$$

$$|\mathrm{ag}(x)| \triangleq \mathrm{ag}(x)$$

$$\mathrm{ag}(x) \cdot \mathrm{ag}(x') \triangleq \begin{cases} \mathrm{ag}(x) & \text{if } x = x' \\ \text{undefined} & \text{otherwise} \end{cases}$$

See Figure 3.10 for an extract of its rules. Its duplicative nature can be seen in the rules AgConsOk and AgProdEq: consuming the predicate doesn't turn it into $\bot$ but leaves it untouched, while producing into an already present state suceeds as long as the predicate's out and the state are equal, as seen in the path condition: $[\hat{x} = \hat{x}']$.

An observation that can be made from this is that for consumption to be sound, the resulting state can never be *anything less than the input's core. Otherwise function call by specification is not frame preserving, I think? (At least with Ag.) But this isn't defined as an axiom or anything, so I'm not sure what to do with it. It doesn't follow from any axiom, and I haven't proved it for any state model (though I'm sure it holds). Is it OK to just leave it here as it is still an insight, without building upon it?*

$$\mathsf{consume}(\hat{\sigma}, \vec{\hat{v}}_i) \rightsquigarrow (\mathsf{Ok}, \hat{\sigma}', \vec{\hat{v}}_o, \pi) \land \theta, s, \sigma \models \hat{\sigma} \land \theta, s, \sigma' \models \hat{\sigma}' \implies |\sigma| \preccurlyeq \sigma'$$

Unline Ex, Ag is *not cancellative*: in other words, $a \cdot b = a \cdot c \nRightarrow b = c$, since for instance $\mathrm{ag}(x) \cdot \bot = \mathrm{ag}(x) \cdot \mathrm{ag}(x)$, but $\bot \neq \mathrm{ag}(x)$. Cancellativity is a property that was initially thought to be required for separation algebras [16, 22], but that has since been dropped [34, 56].

The agreement state model is thus useful to model immutable, shareable state – for instance, it is used in the JavaScript instantiation (discussed in §4.2) to represent the address of an object's metadata, as this information is immutable. This was first brought to a CSE engine in JaVerT [28, 29], though when that was developed the notion of an "agreement state model" didn't exist and as such it wasn't identified as one.

3.3.3 Fraction

The fractional state model $\textsc{Frac}(X)$ allows for exclusive ownership while allowing sharing, via fractional permissions. This idea originates from [9, 10], where the "points to" assertion is equipped with a fraction q: a state satisfying $a \overset{1}{\mapsto} x$ has full ownership (read and write) of x, and can be split into $a \overset{q}{\mapsto} x * a \overset{1-q}{\longmapsto} x$, creating two states with read-only permissions. The Frac memory model is a generalisation of this at the value level. It defines one predicate, `frac`, that has one in, the fraction q, and one out, the value x. Just like Ex, it provides a `load` and a `store` action – the only difference being that a fraction of 1 is required to modify the value. Because the max fraction is 1, we know that when it is 1 no frame can have ownership of the value, and it can thus safely be modified.

$$\textsc{Frac}(X) \triangleq \mathrm{frac}(x\colon X, q\colon (0;1])$$

$$|\mathrm{frac}(x,q)| \triangleq \bot$$

$$\mathrm{frac}(x,q) \cdot \mathrm{frac}(x',q') \triangleq \begin{cases} \mathrm{frac}(x, q+q') & \text{if } x = x' \wedge q + q' \leq 1 \\ \text{undefined} & \text{otherwise} \end{cases}$$

We put the fraction q in the in values of the `frac` core predicate to allow `consume` to subtract exactly the permission required from the state – if both the permission and the value were out-values, then `consume` would always evaluate to $\bot$, cancelling the main advantage of this state model. The rules for `consume` can be seen in Figure 3.11. We note here that in the symbolic state model, the fraction part is also lifted to the symbolic realm, allowing for a symbolic permission.

Because the outcome of actions depends on the amount of permission owned, Frac is also a good example to show how one may adapt a concrete state model into a symbolic state model, by moving the relevant concrete conditions into the path condition. The rules for the concrete and symbolic `store` are shown in Figure 3.12. Note how, between CFracStorePerm and FracStorePerm, the condition $q < 1$ is replaced by $\hat{q} < 1$, and is returned in the path condition, rather than being checked for in the rule itself, since the value of $\hat{q}$ depends on the current PC.

FracConsAll
$\mathtt{consume}(\mathrm{frac}(\hat{x}, \hat{q}), \mathtt{frac}, [\hat{q}']) \rightsquigarrow (\mathtt{Ok}, \bot, [\hat{x}], [\hat{q} = \hat{q}'])$

FracConsSome
$\mathtt{consume}(\mathrm{frac}(\hat{x}, \hat{q}), \mathtt{frac}, [\hat{q}']) \rightsquigarrow (\mathtt{Ok}, \mathrm{frac}(\hat{x}, \hat{q} - \hat{q}'), [\hat{x}], [0 < \hat{q}' < \hat{q}])$

FracConsMiss
$\mathtt{consume}(\mathrm{frac}(\hat{x}, \hat{q}), \mathtt{frac}, [\hat{q}']) \rightsquigarrow (\mathtt{Miss}, \mathrm{frac}(\hat{x}, \hat{q}), [\hat{q}' - \hat{q}], [\hat{q} < \hat{q}' \leq 1])$

FracConsFail
$\mathtt{consume}(\mathrm{frac}(\hat{x}, \hat{q}), \mathtt{frac}, [\hat{q}']) \rightsquigarrow (\mathtt{LFail}, \mathrm{frac}(\hat{x}, \hat{q}), [], [\hat{q}' \leq 0 \vee 1 < \hat{q}'])$

Figure 3.11: `consume` rules for Frac

CFracStoreOk
$$\frac{q = 1}{\mathtt{store}(\mathrm{frac}(x, q), [x']) = (\mathtt{Ok}, \mathrm{frac}(x', q), [])}$$

FracStoreOk
$\mathtt{store}(\mathrm{frac}(\hat{x}, \hat{q}), [\hat{x}']) \rightsquigarrow (\mathtt{Ok}, \mathrm{frac}(\hat{x}', \hat{q}), [], [\hat{q} = 1])$

CFracStorePerm
$$\frac{q < 1}{\mathtt{store}(\mathrm{frac}(x, q), [x']) = (\mathtt{Miss}, \mathrm{frac}(x, q), [1 - q])}$$

FracStorePerm
$\mathtt{store}(\mathrm{frac}(\hat{x}, \hat{q}), [\hat{x}']) \rightsquigarrow (\mathtt{Miss}, \mathrm{frac}(\hat{x}, \hat{q}), [1 - \hat{q}], [\hat{q} < 1])$

Figure 3.12: `store` rules for Frac

While this state model is conceived taking into account fractional permissions, other separation algebras exist to tackle the same problem. For instance, [9] also proposes a counting permissions system: $a \overset{}{\underset{0}{\mapsto}} x$ is a writing permission, and it can issue reading permissions denoted $a \rightharpoonup x$, such that $a \underset{q}{\mapsto} x \iff a \underset{q+1}{\longmapsto} x * a \rightharpoonup x$. Though we do not define it in full, this could also be implemented as a state model, providing two core predicates: $\langle \mathtt{write} \rangle(q; x)$ and $\langle \mathtt{read} \rangle(; x)$, with every consumption of `read` from a writeable state increasing q, and inversely every production of `read` decreasing it, and checking $q = 0$ when using `store`. This shows state modes seem to be the right abstraction, as they support a variety of resource algebras.

3.4 State Model Transformers

While the above state models can prove useful, they are limited in use, only allowing the storage of one value. One would likely want to model state that is more complex than this, storing a range of values in a variety of ways. Iris introduces the idea of RA constructions [34] – this idea has then been adapted to Gillian in [2], with PCMs. Here we revisit these constructions, porting them to RAs. We call them *state model transformers*,[6] or state transformers.

[6]Not to be confused with "state monad transformers", which allow adding state to a monad.

3.4.1 Sum

The sum state model $\textsc{Sum}(\mathbb{S}_1, \mathbb{S}_2)$, also written $\mathbb{S}_1 + \mathbb{S}_2$, allows representing states that can be in either the states of $\mathbb{S}_1$ or $\mathbb{S}_2$. This state model doesn't introduce any core predicates or actions, and instead simply lifts those of the two underlying state models. Let $\mathcal{A}_1$ and Δ_1 the actions and core predicates of $\mathbb{S}_1$, and $\mathcal{A}_2$ and Δ_2 those of $\mathbb{S}_2$, the actions of the sum are defined as $\mathcal{A} = \{\mathtt{L}\alpha : \alpha \in \mathcal{A}_1\} \uplus \{\mathtt{R}\alpha : \alpha \in \mathcal{A}_2\}$, and the core predicates as $\Delta = \{\mathtt{L}\delta : \delta \in \Delta_1\} \uplus \{\mathtt{R}\delta : \delta \in \Delta_2\}$. Actions and predicates are simply tagged with what side of the sum they originate from.

Let Σ_1 and Σ_2 be the carrier sets of the RAs of $\mathbb{S}_1$ and $\mathbb{S}_2$ respectively; the RA of $\textsc{Sum}(\mathbb{S}_1, \mathbb{S}_2)$ is:

$$\textsc{Sum}(\mathbb{S}_1, \mathbb{S}_2) \triangleq \mathbb{S}_1 + \mathbb{S}_2 \triangleq l(\sigma\colon \Sigma_1) \mid r(\sigma\colon \Sigma_2)$$

$$l(\sigma) \cdot l(\sigma') \triangleq l(\sigma \cdot \sigma')$$

$$r(\sigma) \cdot r(\sigma') \triangleq r(\sigma \cdot \sigma')$$

$$|l(\sigma)| \triangleq \begin{cases} \bot & \text{if } |\sigma| = \bot \\ l(|\sigma|) & \text{otherwise} \end{cases}$$

$$|r(\sigma)| \triangleq \begin{cases} \bot & \text{if } |\sigma| = \bot \\ r(|\sigma|) & \text{otherwise} \end{cases}$$

In §3.1.2 we showed how $\textsc{Sum}_{\text{PCM}}$ must disallow the unit on each side to be sound. In RAs however, this is not an issue anymore, as the underlying state models can simply not define a unit. We may also redefine `is_exclusively_owned` to take into account that the unit is not present in the carrier set, giving a much simpler definition:

$$\texttt{is\_exclusively\_owned}\ \sigma \triangleq \nexists \sigma' \in \Sigma.\, \sigma \# \sigma'$$

The rules of the Sum state model are otherwise straightforward, lifting `execute_action`, `consume` and `produce` to the sum from the underlying state models. The only minor difference is when a `Miss` occurs – because fixes are only generated from a LVal `list`, Sum has no way of knowing which side raised the `Miss`, and can thus not call the `fix` function of the correct side. To avoid this, when the outcome of `execute_action` or `consume` is a `Miss` we concatenate an identifier to the values to allow discriminating between the two. An extract of these rules is shown in Figure 3.13.

This example is interesting in that it also shows the constraint imposed by the fact $\bot \notin \hat{\Sigma}$: it is often needed to define auxiliary *unwrap* and *wrap* functions, that get the state out a value if it is not $\bot$, and returns $\bot$ otherwise. In other words, *unwrap*: $\hat{\Sigma}_1^? \rightharpoonup \hat{\Sigma}_2^?$ and *wrap*: $\hat{\Sigma}_2^? \to \hat{\Sigma}_1^?$. While these are straightforward to define and use, they can clutter rules a bit, in favour of not needing to handle $\bot$ and non-$\bot$ cases separately.

$$\text{Given } \mathit{wrap}_l(x) = \begin{cases} \bot & \text{if } x = \bot \\ l(x) & \text{otherwise} \end{cases} \text{ and } \mathit{unwrap}_l(x_l) = \begin{cases} \bot & \text{if } x = \bot \\ x & \text{if } x_l = l(x) \\ \text{undefined} & \text{otherwise} \end{cases}$$

$$\textsc{SumLAction}\ \frac{\hat{\sigma} = \mathit{unwrap}_l(\hat{\sigma}_l) \qquad \alpha(\hat{\sigma}, \vec{\hat{v}}_i) \rightsquigarrow (o, \hat{\sigma}', \vec{\hat{v}}_o, \pi) \qquad \hat{\sigma}'_l = \mathit{wrap}_l(\hat{\sigma}') \qquad o \neq \texttt{Miss}}{\texttt{L}\alpha(\hat{\sigma}_l, \vec{\hat{v}}_i) \rightsquigarrow (o, \hat{\sigma}'_l, \vec{\hat{v}}_o, \pi)}$$

$$\textsc{SumLActionMiss}\ \frac{\hat{\sigma} = \mathit{unwrap}_l(\hat{\sigma}_l) \qquad \alpha(\hat{\sigma}, \vec{\hat{v}}_i) \rightsquigarrow (\texttt{Miss}, \hat{\sigma}', \vec{\hat{v}}_o, \pi) \qquad \hat{\sigma}'_l = \mathit{wrap}_l(\hat{\sigma}')}{\texttt{L}\alpha(\hat{\sigma}_l, \vec{\hat{v}}_i) \rightsquigarrow (\texttt{Miss}, \hat{\sigma}'_l, \texttt{`l'} :: \vec{\hat{v}}_o, \pi)} \qquad \textsc{SumLFix}\ \frac{\mathbb{S}_1.\texttt{fix}\ \vec{\hat{v}}_i = a}{\texttt{fix `l'} :: \vec{\hat{v}}_i = a}$$

Figure 3.13: Rules for handling `Miss` in Sum

3.4.2 Product

The product state model $\textsc{Product}(\mathbb{S}_1, \mathbb{S}_2)$, also written $\mathbb{S}_1 \times \mathbb{S}_2$, allows representing pairs of states, where each side belongs to a specific state model. For instance, one could chose to represent the $\textsc{Frac}(X)$ state model as $\textsc{Ex}(X) \times \textsc{Frac}_{\mathbb{Q}}$, where $\textsc{Frac}_{\mathbb{Q}}$ is an exclusively owned rational in $(0; 1]$, and with composition defined as addition. This is, in fact, the way Iris defines Frac [32] – however, this approach makes using the state model less practical, as one would need to define a separate predicate for each side.

Iris also defines the product RA, by simply lifting pointwise the operations on each side of the product. We call this initial definition $\textsc{Product}_0$, defining it as:

$$\textsc{Product}_0(\mathbb{S}_1, \mathbb{S}_2) \triangleq \Sigma_1 \times \Sigma_2$$

$$(\sigma_l, \sigma_r) \cdot (\sigma'_l, \sigma'_r) \triangleq (\sigma_l \cdot \sigma'_l, \sigma_r \cdot \sigma'_r)$$

$$|(\sigma_l, \sigma_r)| \triangleq \begin{cases} (|\sigma_l|, |\sigma_r|) & \text{if } |\sigma_l| \neq \bot \wedge |\sigma_r| \neq \bot \\ \bot & \text{otherwise} \end{cases}$$

This definition, while straightforward, has a problem: indeed, while its sets of states is $\Sigma_1 \times \Sigma_2$, we still have that $\bot \notin \Sigma$. If an empty state ($\bot$) produces a core predicate for one of its sides, what happens to the other side? Indeed, while one side becomes defined, such that $\sigma_l \in \Sigma_1$, $(\sigma_l, \bot)$ is not a valid state, since $\bot \notin \Sigma_2$. This is a problem, since in our CSE engine all states are initialised as $\bot$ and can only be built upon via actions or predicate production, one predicate at a time. It seems here that the consume-produce interface of our engine, which allows creating state bit by bit, can thus be unadapted for certain RAs. Of course one can define additional predicates for a specific product instantiation that combines predicates of underlying state models, however this goes against the core idea of state model transformers to minimise the amount of effort needed to construct new state models. Finally, we also note that this problem is not encountered in $\textsc{Product}_{\text{PCM}}$, since there the unit of the PCM can be defined as $(0_1, 0_2)$ and work straightforwardly.

$$\text{Given } \mathit{wrap}(x,y) = \begin{cases} \bot & \text{if } x = \bot \wedge y = \bot \\ (x,y) & \text{otherwise} \end{cases} \text{ and } \mathit{unwrap}(s) = \begin{cases} (\bot,\bot) & \text{if } s = \bot \\ (x,y) & \text{otherwise} \end{cases}$$

$$\frac{(\hat{\sigma}_l, \hat{\sigma}_r) = \mathit{unwrap}(\hat{\sigma}) \qquad \alpha(\hat{\sigma}_l, \vec{\hat{v}}_i) \rightsquigarrow (o, \hat{\sigma}'_l, \vec{\hat{v}}_o, \pi) \qquad \hat{\sigma}' = \mathit{wrap}(\hat{\sigma}'_l, \hat{\sigma}_r) \qquad o \neq \texttt{Miss}}{\mathsf{L}\alpha(\hat{\sigma}, \vec{\hat{v}}_i) \rightsquigarrow (o, \hat{\sigma}', \vec{\hat{v}}_o, \pi)} \text{ ProductLAction}$$

Figure 3.14: Example of action rules in Product

We define an alternative product RA that is more suited to this engine:

$$\textsc{Product}(\mathbb{S}_1, \mathbb{S}_2) \triangleq \mathbb{S}_1 \times \mathbb{S}_2 \triangleq (\Sigma_1^? \times \Sigma_2^?) \setminus \{(\bot,\bot)\}$$

$$(\sigma_l, \sigma_r) \cdot (\sigma'_l, \sigma'_r) \triangleq (\sigma_l \cdot \sigma'_l, \sigma_r \cdot \sigma'_r)$$

$$|(\sigma_l, \sigma_r)| \triangleq \begin{cases} \bot & \text{if } |\sigma_l| = \bot \wedge |\sigma_r| = \bot \\ (|\sigma_l|, |\sigma_r|) & \text{otherwise} \end{cases}$$

We take advantage of the option state model $-^?$ and use that instead, thus allowing either sides of the product to be $\bot$. Crucially however, we exclude the $(\bot,\bot)$ state and ensure in the *wrap* helper that it never occurs (see Figure 3.14). This is important, because it means *exclusivity* is carried: if for (σ_l, σ_r) we have `is_exclusively_owned` $\sigma_l \wedge$ `is_exclusively_owned` σ_r, then it also follows that `is_exclusively_owned` (σ_l, σ_r). If we hadn't excluded $(\bot,\bot)$, then it would always be the case that $(\sigma_l, \sigma_r) \cdot (\bot,\bot)$ holds, meaning that the state could never be exclusively owned. This is unpractical, as it would mean that disposing of a product is not frame preserving.

3.4.3 Freeable

The Freeable($\mathbb{S}$) state model transformer allows extending a state model with a `free` action, that allows freeing a part of memory. The freed memory can then not be accessed, and attempting to use it raises a use-after-free error. The notion of marking freed memory as freed, rather than simply discarding it, has come from Incorrectness Separation Logic [57], as indeed simply discarding the memory would break Frame Addition.

Freeable is similar to the OneShot RA of Iris, with the key difference being that the `freed` predicate used to mark a resource as freed is *not duplicable*. Iris defines $\textsc{OneShot}(X) \triangleq \textsc{Ex}(\{1\}) + \textsc{Ag}(X)$ [34], which is not UX-sound; in order for the switch of the side of a sum to be UX-sound, the resulting state (here, freed) must be exclusively owned, which Ag never is. This is not a problem for Iris, that is only concerned with OX soundness, but because our goal is to make our state models both OX and UX sound for additional flexibility, we need to make this change.

To ensure frame preservation, the underlying state model must provide an additional function, equivalent to the aforementioned property "exclusive": `is_exclusively_owned`$\colon \Sigma \to \mathbb{B}$. Because actions in

FreeableActionFree
$\texttt{free}(l(\hat{\sigma}), []) \rightsquigarrow (\texttt{Ok}, r(\text{ex}(\text{freed})), [], [\texttt{is\_exclusively\_owned}\ \hat{\sigma}])$

FreeableActionFreeErr
$\texttt{free}(l(\hat{\sigma}), []) \rightsquigarrow (\texttt{Miss}, l(\hat{\sigma}), \texttt{fix\_owned}\ \hat{\sigma}, [\neg\texttt{is\_exclusively\_owned}\ \hat{\sigma}])$

FreeableActionFreeMiss
$\texttt{free}(\bot, []) \rightsquigarrow (\texttt{Miss}, \bot, \texttt{fix\_owned}\ \bot, [])$

FreeableActionDoubleFree
$\texttt{free}(r(\text{ex}(\text{freed})), []) \rightsquigarrow (\texttt{Err}, r(\text{ex}(\text{freed})), [], [])$

Figure 3.15: Symbolic action rule of Freeable

the symbolic realm need to be sound with respect to the concrete realm, we then lift its definition to $\texttt{is\_exclusively\_owned}\colon \hat{\Sigma} \to \text{LVal}$, making it return a symbolic value that can be appended to the path condition.

While not needed for the core and compositional engines, the bi-abductive engine requires that misses may be fixed; Freeable($\mathbb{S}$) however cannot access directly the core predicates of $\mathbb{S}$ to provide fixes when freeing an empty state. As such, the state model must also provide a $\texttt{fix\_owned}\colon \hat{\Sigma}^? \to \text{LVal}\ \texttt{list}$ function that returns possible fixes to make the given state exclusively owned. This also implies that for any state σ of $\mathbb{S}$, if it is not exclusively owned then there exists a bigger state σ' such that $\sigma \preccurlyeq \sigma' \wedge \texttt{is\_exclusively\_owned}\ \sigma'$. This is not the case, for instance, for Ag, meaning a construction such as Freeable(Ag(X)) is not sound.

Similarly to Iris, we define the Freeable($\mathbb{S}$) state model by construction: Freeable($\mathbb{S}$) $\triangleq \mathbb{S} +$ Ex({freed}), which allows all associated rules to be kept. For clarity, we rename the core predicate $\texttt{R ex}$ (defined by Ex({freed})) as $\texttt{freed}$. We also extend its actions with the $\texttt{free}$ action.

Because Freeable is constructed via other state model transformers, we only need to describe the rules for $\texttt{free}$, which are shown in Figure 3.15 – the rest of the construction is already defined. This shows how simpler state models can be extended while alleviating the user from the burden of proving the soundness of the base construction.

Conveniently, use-after-free errors are already handled by the sum construction, thanks to the SumLActionIncompat rule which forbids actions from one side of the sum to be executed on the other side.

3.4.4 Partial Map

The partial map state model transformer PMap($I, \mathbb{S}$) is perhaps one of the most important state model transformers, as it allows having multiple state instances at different addresses and accessing them. It is constructed from a domain, $I \subseteq \text{Val}$, and a state model for the codomain. For instance, the traditional separation logic heap can be modelled as PMap($\mathbb{N}$, Ex(Val)). The C and the JavaScript memory models can also both be modelled using a PMap, despite them being radically different languages.

It defines one action, `alloc`, and one core predicate, `domainset`. It also *lifts* all actions and predicates of the underlying model, adding to them an index argument or in-value. For instance, while for $\textsc{Ex}(X)$ one has the core predicate $\langle \mathsf{ex} \rangle(; x)$, for $\textsc{PMap}(\textsc{Ex}(X))$ one would instead have predicates of the form $\langle \mathsf{ex} \rangle(i; x)$, which corresponds to the "points to" assertion $i \mapsto x$. We define it's RA as:

$$\textsc{PMap}(I, \mathbb{S}) \triangleq (I \xrightarrow{fin} \mathbb{S}.\Sigma) \times \mathcal{P}(I)^?$$

$$(h, d) \cdot (h', d') \triangleq (h'', d'')$$

$$\text{where } h'' \triangleq \lambda i. \begin{cases} h(i) \cdot h'(i) & \text{if } i \in \mathrm{dom}(h) \cap \mathrm{dom}(h') \\ h(i) & \text{if } i \in \mathrm{dom}(h) \setminus \mathrm{dom}(h') \\ h'(i) & \text{if } i \in \mathrm{dom}(h') \setminus \mathrm{dom}(h) \\ \text{undefined} & \text{otherwise} \end{cases}$$

$$\text{and } d'' \triangleq \begin{cases} d & \text{if } d' = \bot \\ d' & \text{if } d = \bot \\ \text{undefined} & \text{otherwise} \end{cases}$$

$$\text{and } d'' = \bot \vee \mathrm{dom}(h'') \subseteq d''$$

$$|(h, d)| \triangleq \begin{cases} \bot & \text{if } \mathrm{dom}(h') = \emptyset \\ (h', \bot) & \text{otherwise} \end{cases}$$

$$\text{where } h' \triangleq \lambda i. \begin{cases} |h(i)| & \text{if } i \in \mathrm{dom}(h) \wedge |h(i)| \neq \bot \\ \text{undefined} & \text{otherwise} \end{cases}$$

An element of PMap is of the form (h, d), where h are the mappings from locations to states, and d is the *domain set*, the set of indices that are known to exist. It is defined such that $\mathrm{dom}(h) \subseteq d$ always holds. The domain set can also be $\bot$, in which case there is no knowledge on what addresses exist or don't. The aim of the domain set is to allow separating invalid accesses from misses: if $d \neq \bot$, then any access to $i \notin d$ leads to an `Err` outcome, however if $i \in d$ and the binding does not exist in the heap, then a `Miss` is raised.

As a consequence of this, $\textsc{PMap}(I, \mathbb{S})$ has an additional requirement for $\mathbb{S}$: any action execution on $\bot$ must lead to a `Miss`.

$$\alpha(\bot, \vec{v}_i) \rightsquigarrow (o, \sigma', \vec{v}_o, \pi) \implies o = \mathtt{Miss}$$

$$\textsc{PMapGetMatch}\ \frac{(\hat{h},\hat{d}) = \mathit{unwrap}(\hat{\sigma}) \qquad \hat{i}' \in \mathrm{dom}(\hat{h}) \qquad \hat{\sigma}_i = \hat{h}(\hat{i}')}{\texttt{get}(\hat{\sigma},\hat{i}) \rightsquigarrow (\hat{i}', \hat{\sigma}_i, [\hat{i} = \hat{i}'])}$$

$$\textsc{PMapGetAdd}\ \frac{(\hat{h},\hat{d}) = \mathit{unwrap}(\hat{\sigma}) \qquad \hat{i} \notin \mathrm{dom}(\hat{h}) \qquad \hat{d} \neq \bot}{\texttt{get}(\hat{\sigma},\hat{i}) \rightsquigarrow (\hat{i}, \bot, [\hat{i} \notin \mathrm{dom}(\hat{h}) \wedge \hat{i} \in \hat{d}])}$$

$$\textsc{PMapGetBotDomain}\ \frac{(\hat{h},\hat{d}) = \mathit{unwrap}(\hat{\sigma}) \qquad \hat{i} \notin \mathrm{dom}(\hat{h}) \qquad \hat{d} = \bot}{\texttt{get}(\hat{\sigma},\hat{i}) \rightsquigarrow (\hat{i}, \bot, [\hat{i} \notin \mathrm{dom}(\hat{h})])}$$

Figure 3.16: Rules for `get` for PMap

This is needed for frame preservation; if this wasn't the case, one could run into a case where the action of a cell in the heap succeeds on an empty heap, but an empty domain set would be compatible with the state, despite its composition with the state leading to a different outcome (an `Err`).

This constraint also comes from the fact that PMap can't automatically yield a `Miss` when executing actions on missing cells, and must instead execute the action on $\bot$. Indeed, because $\textsc{PMap}(I, \mathbb{S})$ can accept any underlying state model, it does not "know" what values are needed to fix the missing cell. Instead, it needs $\mathbb{S}$ to raise a `Miss` that it can then lift with an index.

As mentioned before, PMap lifts actions and core predicates, adding to them an index. Getting the state associated to an index is not simple, and will in fact be the target of several optimisations in §3.5. For now we consider the simplest form of symbolic matching: either the index is present in the heap, or it isn't – if it isn't and the domain set is not $\bot$, we also know that it must be in the domain set. This creates three rules one must consider. For instance, given a state $([\hat{a} \mapsto \hat{x}, \hat{b} \mapsto \hat{y}], \bot)$, when an action is executed with index $\hat{c}$, three branches are created: either $\hat{a} = \hat{c}$, or $\hat{b} = \hat{c}$, or $\hat{c}$ is not part of the current heap, giving us $\hat{c} \notin \{\hat{a}, \hat{b}\}$.

Because most `execute_action`, `consume` and `produce` rules of PMap require retrieving states at a specific index, we introduce an abstraction over this, with the `get` and `set` functions. The former allows getting the state at an index *with branching*; its signature is $\texttt{get}\colon \textsc{PMap}(I, \mathbb{S})^? \to I \to \mathcal{P}(I \times \Sigma^? \times \Pi)$, as it returns the index to modify exactly the returned state, the state itself, and a path condition corresponding to the branch. The `set` function has signature $\texttt{set}\colon \textsc{PMap}(I, \mathbb{S})^? \to I \to \Sigma^? \to \textsc{PMap}(I, \mathbb{S})^?$. Unlike for `get`, we don't allow branching when setting: because branching already happened when getting, and `get` returns the index to use to modify the state, we can use this new index and be guaranteed that it is sound. For instance for the above example, the three executions are $\texttt{get}(\hat{\sigma}, \hat{c}) \rightsquigarrow (\hat{a}, \hat{\sigma}_a, [\hat{c} = \hat{a}])$, $\texttt{get}(\hat{\sigma}, \hat{c}) \rightsquigarrow (\hat{b}, \hat{\sigma}_b, [\hat{c} = \hat{b}])$ and $\texttt{get}(\hat{\sigma}, \hat{c}) \rightsquigarrow (\hat{c}, \bot, [\hat{c} \notin \{\hat{a}, \hat{b}\}])$. We present the rules for `get` in Figure 3.16.

Another unseemingly complex aspect of PMap is allocation, for which the rules are presented in Figure 3.17. To allow instantiating, the wrapped state model must provide an `instantiate` function defined as $\texttt{instantiate}\colon \mathrm{Val} \to \Sigma$, such that given the arguments given to `alloc`, it returns a newly instantiated state. An initial definition of PMap had `alloc` always succeed; for instance, a $\bot$ state could alloc, resulting in $([\hat{i} \mapsto \sigma_i], \bot)$, with σ_i the instantiated state. This however does not satisfy Frame

$$\textsc{PMapAlloc}\quad \frac{\hat{d} \neq \bot \qquad \hat{i} = \texttt{fresh}\ I \qquad \hat{\sigma}_i = \texttt{instantiate}(\vec{\hat{v}}_i) \qquad \hat{h}' = \hat{h}[\hat{i} \leftarrow \hat{\sigma}_i] \qquad \hat{d}' = \hat{d} \uplus \{\hat{i}\}}{\texttt{alloc}((\hat{h}, \hat{d}), \vec{\hat{v}}_i) \rightsquigarrow (\texttt{Ok}, (\hat{h}', \hat{d}'), [\hat{i}], [\hat{i} = \hat{i}])}$$

$$\textsc{PMapAllocMiss}\quad \frac{(\hat{h}, \hat{d}) = \mathit{unwrap}(\hat{\sigma}) \qquad \hat{d} = \bot}{\texttt{alloc}(\hat{\sigma}, \vec{\hat{v}}_i) \rightsquigarrow (\texttt{Miss}, \hat{\sigma}, [\texttt{`domainset'}], [])}$$

Figure 3.17: Rules for `alloc` for PMap

subtraction: the frame $(\emptyset, \{\})$, made of an empty heap and the empty domain set, is compatible with the state before the allocation, but is not compatible with the state after, since the domain of the heap $\{\hat{i}\}$ is not a subset of $\{\}$. We must thus enforce ownership of the domain set when allocating. This is odd: in allocation-based languages like C, one can always allocate a new cell, without needing to "own" anything, as the allocator is global. With this definition of `alloc` however, one can't allocate without it, which can be problematic: for instance in a multi-threaded setting, two threads cannot own the domain set, since it is exclusively owned (in order to allow modification).

This is a weakness of this PMap definition, as we require a more restrictive behaviour than what is usually admitted with allocation. This also has the unfortunate side effect that whenever a function needs to allocate, it must specify a `domainset` core predicate in its pre and postcondition, which can become quite verbose and unpractical. Attempts have been made to preserve the domain set while allowing for more flexible allocation semantics, for instance by having the domain set be a duplicable resource that allows modification, by defining composition as set union. However these relaxations often had as a consequence that out of bounds accesses couldn't reliably be detected anymore, as indeed more heap cells with a larger domain set could always be composed to yield a succesful outcome. In fact, *allocation seems to be a topic that is in essence hard to tackle with separation logic*; [58, 62] already mentioned some of these difficulties regarding freshness of the new index. Even when using sophisticated logics like Iris, works on verifying CompCert-C code doesn't differentiate misses from out of bounds: "An `rmap` does not distinguish between an unallocated block and a block on which it holds no ownership" [48].

A solution to the above problem is to simply get rid of the domain set, and define PMap only as the mappings from indices to states. This works, however one loses the ability to detect out of bounds accesses entirely, as additional cells can always be composed with the state to yield a non-`Miss` outcome. For our purposes however, we keep the definition using a domain set, as it provides better automation and is adapted to our needs.

3.4.5 Partial Map Variations

The behaviour of PMap is not always adapted for the state one wants to model. Here we describe in a shorter form some state models that function similarly to PMap.

Dynamic Partial Map

The *dynamic* partial map, DynPMap$(I, \mathbb{S})$, allows dynamically allocating missing cells when they're accessed. It is used, for instance, when modelling objects in JavaScript. In JavaScript, accessing a property that hasn't been set simply yields a default value, `undefined`, and any property of an object can be set directly without needing to allocate it first. This can be easily achieved by modifying PMap, such that any out of bounds accesses have the effect of instantiating the state at that location, storing it, and executing the action against it. In other words, instead of using the domain set to distinguish between `Err` and `Miss`, we use it to distinguish between `Ok` and `Miss` [2].

We also need to remove the `alloc` action, since allocation has no reason to exist; the cell can be modified directly instead.

List

The list state model, List$(\mathbb{S})$, allows storing a list of states up to a bound. Instead of having a domain set, we define it as having a bound n such that all indices i are in $[0; n($. It is useful when wanting to represent continuous blocks of memory of known size. We use it for the state model of WISL, which uses a simple block-offset memory model: each memory access requires an address of the block, and the offset within the block to read from.

Because the bound can also be $\bot$, it serves the exact same purpose as the domain set in PMap: distinguishing out of bounds from misses.

General Map

Due to the strong similarity between PMap and List, it is tempting to define a more general sort of map state model, that allows replicating both behaviours. The general map GMap allows this; constructed as GMap$(I, \mathbb{S}, \mathbb{S}_D)$, it receives the same domain set and codomain state model as before, but it also receives a *discriminator* state model $\mathbb{S}_D$, which represents the state which is capable of discriminating out of bounds accesses from missing accesses.

The discriminator must come equipped with an `is_within`: $\Sigma_D \to I \to \mathbb{B}$ function that given an index returns whether or not the state is within bounds. Here, the input discriminator state is never $\bot$: if the discriminator is not known then a `Miss` must be issued, since composing a state with a non-$\bot$ discriminator can change the outcome. `is_within` must also be lifted to the symbolic realm, resulting in `is_within`: $\hat{\Sigma}_D \to I \to \mathcal{P}(\text{LVal})$.

Defining PMap and List with a GMap would require using the discriminator state models $\text{Ex}(\mathcal{P}(I))$ and $\text{Ex}(\mathbb{N})$ respectively, with the following `is_within` definitions:

$$\texttt{is\_within}_{\text{PMap}}\ d\ i \triangleq i \in d$$

$$\texttt{is\_within}_{\text{List}}\ n\ i \triangleq 0 \leq i \wedge i < n$$

For the above two examples it is also needed to exclude the `store` actions defined by the discriminator (via Ex), as modifying the domain set of a partial map or the bound of a list is not frame preserving. In general, it is unsound to modify the discriminator directly without also modifying the map: "increasing" it breaks Frame Addition (as states compatible with the new state aren't compatible with the old state), while "decreasing" it breaks Frame subtraction (for the inverse reason). Here increasing and decreasing refer to modifying the discriminator such that more or less indices are considered *within* it.

3.5 Optimising Partial Maps

Map-like state models are both widely used and of varied applications, as seen before. They are also quite complex, as they can lead to many branches, even when only a small number of the branches is actually feasible. Consider the PMap state $([\hat{a}_1 \mapsto x_1, ..., \hat{a}_n \mapsto x_n], \bot)$. Executing a lookup on it will yield $n+1$ branches: one for each possible match, and one for the case where the index is different from all others. Such a state also needs to carry the fact that all addresses are distinct; for n addresses, this is $\frac{n(n+1)}{2}$ inequalities. While at the theory level this isn't an issue, as we only worry about the soundness of the actions when the path condition is satisfiable, this can have a significant performance impact on the implementation.

Optimisations exist to reduce the number of branches generated, and while they have been implemented succesfully in Gillian, there currently exists no theoretical justification to them proving their soundness. Here, we *theoretically* define how to optimise PMap-like state models, how to prove the soundness of these optimisations, and explore three different optimisations that already exist in Gillian.

3.5.1 Syntactic Checking

The simplest optimisation done to lookups is *syntactic* checking: before attempting to branch on every index on the map, we check if the symbolic index is already present in the map, in which case no branching is needed.

This only requires modifying the matching rule for the `get` function, adding $\hat{i} \notin \text{dom}(\hat{h})$ to the precondition, and adding a rule that directly gets the relevant state; both rules can be seen in Figure 3.18.

This optimisation is what justifies returning an index for every action in PMap. As an example, take the state model PMap(Ex(Val)) in the state $([\hat{a}_1 \mapsto x_a, ..., \hat{a}_n \mapsto x_n])$. When executing `load(`$\hat{b}$`)` with a

$$\frac{(\hat{h}, \hat{d}) = \mathit{unwrap}(\hat{\sigma}) \qquad \hat{i} \in \mathrm{dom}(\hat{h}) \qquad \hat{\sigma}_i = \hat{h}(\hat{i})}{\mathtt{get}(\hat{\sigma}, \hat{i}) \rightsquigarrow (\hat{i}, \hat{\sigma}_i, [])} \text{ SyntacticPMapGetMatch}$$

$$\frac{(\hat{h}, \hat{d}) = \mathit{unwrap}(\hat{\sigma}) \qquad \hat{i} \notin \mathrm{dom}(\hat{h}) \qquad \hat{i}' \in \mathrm{dom}(\hat{h}) \qquad \hat{\sigma}_i = \hat{h}(\hat{i}')}{\mathtt{get}(\hat{\sigma}, \hat{i}) \rightsquigarrow (\hat{i}', \hat{\sigma}_i, [\hat{i} = \hat{i}'])} \text{ SyntacticPMapGetBranch}$$

Figure 3.18: Modified rules for syntactic matching

fresh symbolic variable $\hat{b}$, we branch $n+1$ times, and return the value stored there *along with the used index*, for instance returning $[\hat{a}_3, \hat{x}_3]$. The program is then free to use $\hat{a}_3$ instead of $\hat{b}$ for further accesses to that cell, and thanks to syntactic checking no branching will be needed. Without syntactic checking, every access invariably causes branching, even if the index is already present.

An observation is that thanks to our `get` and `set` abstractions over accessing states at a given index, we can restrict ourselves to only modify these two definitions and carry over the rest of the rules of PMap. Because these optimisations are purely due to the symbolicness of the heap indices, the concrete state model is untouched and we can prove soundness with respect to the concrete PMap. Furthermore, we define symbolic `get` *axioms*, which further ease proving soundness of the optimisations: when proving soundness of PMap, we only use these axioms, meaning that any optimisation that satisfies the axioms is also automatically sound.

$$\begin{aligned}&\theta, s, \sigma \models \hat{\sigma} \land \mathtt{get}(\sigma, i) = \sigma_i \land [\![\hat{i}]\!]_{\theta,s} = i \implies \exists \hat{\sigma}_i, \hat{i}', \pi. \\ &\quad \mathtt{get}(\hat{\sigma}, \hat{i}) \rightsquigarrow (\hat{i}', \hat{\sigma}_i, \pi) \land \theta, s, \sigma_i \models \hat{\sigma}_i \land [\![\hat{i}']\!]_{\theta,s} = i \land \mathtt{SAT}_{\theta,s}(\pi)\end{aligned} \quad \text{(Get OX Soundness)}$$

$$\begin{aligned}&\mathtt{get}(\hat{\sigma}, \hat{i}) \rightsquigarrow (\hat{i}', \hat{\sigma}_i, \pi) \land [\![\hat{i}]\!]_{\theta,s} = [\![\hat{i}']\!]_{\theta,s} = i \land \mathtt{SAT}_{\theta,s}(\pi) \implies \\ &\quad \forall \sigma.\, \theta, s, \sigma \models \hat{\sigma} \implies \exists \sigma_i.\, \mathtt{get}(\sigma, i) = \sigma_i \land \theta, s, \sigma_i \models \hat{\sigma}_i\end{aligned} \quad \text{(Get UX Soundness)}$$

They are quite simple: for the OX case, any `get` call that exists in the concrete world must exist in the symbolic world. For the UX soundness, if a symbolic `get` returns a satisfiable branch then there is also a matching state σ_i in the concrete world.

We also define two additional axioms that we do not prove but that must hold for all `get` and `set` pairs:

$$\begin{aligned}&\mathtt{get}(\hat{\sigma}, \hat{i}) \rightsquigarrow (\hat{i}', \hat{\sigma}_i) \land \mathtt{set}(\hat{\sigma}, \hat{i}', \hat{\sigma}_i') = \hat{\sigma}' \implies \\ &\quad (\exists \sigma, \sigma_i'.\, \theta, s, \sigma \models \hat{\sigma} \land \theta, s, \sigma_i' \models \hat{\sigma}_i') \implies \\ &\quad\quad \exists \sigma'.\, \theta, s, \sigma' \models \hat{\sigma}'\end{aligned} \quad \text{(Get-Set Forwards Soundness)}$$

$$\begin{aligned}&\mathtt{get}(\hat{\sigma}, \hat{i}) \rightsquigarrow (\hat{i}', \hat{\sigma}_i) \land \mathtt{set}(\hat{\sigma}, \hat{i}', \hat{\sigma}_i') = \hat{\sigma}' \implies \\ &\quad (\exists \sigma', \sigma_i.\, \theta, s, \sigma' \models \hat{\sigma}' \land \theta, s, \sigma_i \models \hat{\sigma}_i) \implies \\ &\quad\quad \exists \sigma.\, \theta, s, \sigma \models \hat{\sigma}\end{aligned} \quad \text{(Get-Set Backwards Soundness)}$$

These are quite straightforward: the former states that if a state exists in the concrete world and is modified with a substate that also exists in the concrete world, then the result must exist too. The latter states the reverse: if a state after `set` exists in the concrete world, and we know that the subtate that used to be at the modified location also exists in the concrete world, then the original state also must exist.

These two axioms simply follow from the fact that $\hat{i}'$ represents the location to directly modify the returned state, and that `set` only modifies the state at the given index, the rest being untouched. The axioms are used when proving OX and UX soundness respectively, as they each use either forwards or backwards reasoning.

3.5.2 Split PMap

The *split* partial map, $\text{PMAP}_{\text{Split}}$, goes further to reduce branches: it split the heap into two, one side being the *concrete* part, and the other the *symbolic*. A concrete value is a value that contains no symbolic variables, such that it's evaluation is unaffected by substitutions: $\exists e. \forall \theta, s. [\![\hat{e}]\!]_{\theta,s} = e$. We introduce the predicate `is_concrete` that is true if a value is concrete. The symbolicness of a binding in the heap is determined from the value (the substate) *and the key*; a concrete key pointing to a symbolic substate, or a symbolic key pointing to a concrete substate, both go in the symbolic part of the heap. Similarly to values, a state is concrete if its interpretation is unaffected by substitutions: $\exists \sigma. \forall \theta, s. \theta, s, \sigma \models \hat{\sigma}$. $\text{PMAP}_{\text{Split}}(\text{I},\mathbb{S})$ thus requires the state model to implement an $\texttt{is\_concrete}_{\mathbb{S}} \colon \hat{\Sigma} \to \mathbb{B}$ function, which determines whether the entire state is concrete. For instance for the $\text{EX}(X)$ state model, we have $\texttt{is\_concrete}_{\text{EX}}\ \text{ex}(\hat{x}) \triangleq \texttt{is\_concrete}\ \hat{x}$.

This has two advantages; the first minor advantage is that when looking for syntactic matches in the heaps, if the address is symbolic then we only must check in the symbolic part of the heap, as symbolic keys are not permitted in the concrete part.[7] The second major advantage of this technique is related to an implementation detail of some CSE engines: *substitution*.

Substitution is the process of replacing logical values in a state with a new value.[8] For instance when learning $\hat{a}_1 = \hat{a}_2$, the engine may decide to substitute all occurrences of $\hat{a}_2$ with $\hat{a}_1$ and remove the equality from the PC if it is sound to do so (if $\hat{a}_2$ never occurs again). Because this process applies to the entire state, it can lead to unfortunate consequences, such as needing to merge two state fragments that live at the same location. Reusing the above example, if the state contained $[\hat{a}_1 \mapsto \hat{\sigma}_1, \hat{a}_2 \mapsto \hat{\sigma}_2]$, then the substituted state would be $[\hat{a}_1 \mapsto \hat{\sigma}_1 \cdot \hat{\sigma}_2]$. This is problematic, as composition is not defined for symbolic state; symbolic states are an arbitrary set, unlike concrete states which form an RA. Current implementations usually mimic the composition of the concrete counterpart, branching when relevant; this

[7] This doesn't hold for concrete keys, as it can be that the concrete key maps to a symbolic state, in which case it goes in the symbolic part of the heap.

[8] Here we define substitution in states; this is different to substitution in expressions – also known as α-conversion – which cannot be avoided. State substitution usually uses expression substitution.

SplitPMapSetSomeCon

$$\frac{(\hat{h}_c, \hat{h}_s, \hat{d}) = \mathit{unwrap}(\hat{\sigma}) \quad \hat{\sigma}_i \neq \bot \quad \texttt{is\_concrete}_\Sigma\ \hat{\sigma}_i \quad \texttt{is\_concrete}\ \hat{i} \quad \hat{h}'_c = \hat{h}_c[\hat{i} \leftarrow \hat{\sigma}_i] \quad \hat{h}'_s = \hat{h}_s[\hat{i} \not\leftarrow] \quad \hat{\sigma}' = \mathit{wrap}(\hat{h}'_c, \hat{h}'_s, \hat{d})}{\texttt{set}(\hat{\sigma}, \hat{i}, \hat{\sigma}_i) = \hat{\sigma}'}$$

SplitPMapSetSomeSym

$$\frac{(\hat{h}_c, \hat{h}_s, \hat{d}) = \mathit{unwrap}(\hat{\sigma}) \quad \hat{s}_i \neq \bot \quad \neg(\texttt{is\_concrete}_\Sigma\ \hat{\sigma}_i \vee \texttt{is\_concrete}\ \hat{i}) \quad \hat{h}'_c = \hat{h}_c[\hat{i} \not\leftarrow] \quad \hat{h}'_s = \hat{h}_s[\hat{i} \leftarrow \hat{\sigma}_i] \quad \hat{\sigma}' = \mathit{wrap}(\hat{h}'_c, \hat{h}'_s, \hat{d})}{\texttt{set}(\hat{\sigma}, \hat{i}, \hat{\sigma}_i) = \hat{\sigma}'}$$

SplitPMapSetNone

$$\frac{(\hat{h}_c, \hat{h}_s, \hat{d}) = \mathit{unwrap}(\hat{\sigma}) \quad \hat{\sigma}_i = \bot \quad \hat{h}'_c = \hat{h}_c[\hat{i} \not\leftarrow] \quad \hat{h}'_s = \hat{h}_s[\hat{i} \not\leftarrow] \quad \hat{\sigma}' = \mathit{wrap}(\hat{h}'_c, \hat{h}'_s, \hat{d})}{\texttt{set}(\hat{\sigma}, \hat{i}, \hat{\sigma}_i) = \hat{\sigma}'}$$

Figure 3.19: Rules for `set` in PMap$_{\text{Split}}$

is however not justified in the theory presented in this project, as it is caused by an implementation quirk rather than a theoretical necessity. In fact, what causes this is also a "defect" from the implementation, as it would mean the engine didn't assert that two different locations are in fact different. This phenomenon exists in Gillian under the name substitutions [45], and in Viper under the name consolidation [24].

Despite the fact it is only needed due to implementation details, we argue that optimisations are also something that is only needed because of an implementation, which justifies attempting to prove soundness of techniques that are caused by theoretically unsound solutions.

The second advantage of PMap$_{\text{Split}}$ relating to substitutions is that these can be entirely skipped for the concrete part of the heap, since we already know that it doesn't contain any symbolic variables. Substitutions are computationally expensive, as they require traversing the entire state and attempting substitutions on all values; for large states, this can take a significant amount of time.

In Figure 3.19 we present the rules relating to `set` for PMap$_{\text{Split}}$. In particular, when this requires modifying the binding for one side of the heap, we must delete it from the other, as it can be that the state was concrete and became symbolic, or inversely.

We also note that this optimisation, while at times performant, also comes with a cost: checking if a state is concrete also requires traversing the tree (although it doesn't require modifying it), which comes at a cost for larger trees. This could be further optimised, for instance by caching whether larger states are concrete, and invalidating the cache on modification. However, we have not found this cost to be significant enough to warrant this modification. This improvement is thus highly dependent on what code is verified; if there is a majority of symbolic values, then the speed gain is minimal, and it could even be cancelled by the cost of needing to check for concreteness of substates.

3.5.3 Abstract Location PMap

The last, and by far most agressive optimisation we present is the *abstract location* optimisation. Rather than straightforwardly stating how the optimisation works, we will first take a detour, explaining what abtract locations are.

Abstract Locations?

First, we define locations. A location $\text{loc}(x) \in \text{Loc} \subseteq \text{Val}$ is an uninterpreted value, representing a location in memory. It only has a name x, such that two locations with different names are always different.

Abstract locations (or ALocs) are, as their name implies, locations lifted to the symbolic realm. An abstract location $\text{aloc}(x) \in \text{ALoc} \subseteq \text{LVal}$ also is characterized by its name, and evaluates to a location: $\forall a.\, \exists b.\, [\![\text{aloc}(a)]\!]_{\theta,s} = \text{loc}(b)$. Unlike for locations, two syntactically distinct ALocs can still be equal once concretised. Whether they are equal is decided by the engine according to an additional flag of the path condition: the *matching mode*. We denote it $m \in \{\texttt{MATCH}, \texttt{NO\_MATCH}\}$. When matching is enabled, abstract locations behave similarly to symbolic variables: their equality is decided by the path condition, such that for an empty path condition and $a \neq b$, both $\text{aloc}(a) = \text{aloc}(b)$ and $\text{aloc}(a) \neq \text{aloc}(b)$ are satisfiable. When matching is disabled however, equality of ALocs is only decided by their name: $a = b \Leftrightarrow \text{aloc}(a) = \text{aloc}(b)$. Matching is always enabled for `consume`, and always disabled for `produce` and `execute_action`.

To be able to use abstract locations, the engine must be able to get the ALoc associated to a value; to this end, we introduce the function $\texttt{to\_aloc}\colon \text{LVal} \to \text{Str}^?$. Given a logical value, it *attempts* to find a location or abstract location that it is equal to, and returns its name, if it finds it – otherwise, it returns $\bot$. It attempts to guess the ALoc by traversing the value's AST, and by looking in the PC for an equality. The crucial point of this is that this is a best effort function, that can fail to find an ALoc the value is associated to. When $\texttt{to\_aloc}\ e = \bot$, the engine is then free to use `fresh_aloc` to generate a fresh new abstract location, and add $e = \text{aloc}(a)$ to the PC, where a is the name of the fresh ALoc. This mechanism is a form of *semantic hash consing*, where the engine looks for any value semantically equal to the expression, and looks for an ALoc associated to it. This approach is sound, as the generated location is fresh, such that even if the value was already bound to a location, it can always be found later that the new location is equal to the old one.

Abstract Location Optimisation

We now present the optimisation of PMap using abstract locations: $\text{PMap}_{\text{ALoc}}(\mathbb{S})$. Firstly, and unlike the syntactic optimisation or $\text{PMap}_{\text{Split}}$, this optimisation is only applicable to $\text{PMap}(\text{Loc}, \mathbb{S})$, since abstract locations can only evaluate to locations. The optimisation consists in retrieving the abstract location associated with the given index, and then looking for matches. The strength (and as will be shown later, weakness) of this optimisation is that *there is no branching when matching is disabled*, as

$$\text{with } \hat{i} \in^? \hat{d} \triangleq \begin{cases} \texttt{true} & \text{if } \hat{d} = \bot \\ \hat{i} \in \hat{d} & \text{otherwise} \end{cases}$$

$$\frac{\text{ALocPMapGetNoMatchNotFound} \quad (\hat{h}, \hat{d}) = \mathit{unwrap}(\hat{\sigma}) \qquad a = \texttt{to\_aloc}\ \hat{i} \qquad a \neq \bot \qquad a \notin \mathrm{dom}(\hat{h})}{\texttt{get}(\hat{\sigma}, \hat{i}, \texttt{NO\_MATCH}) \rightsquigarrow (\mathrm{aloc}(a), \bot, [\hat{i} \in^? \hat{d}])}$$

$$\frac{\text{ALocPMapGetMatchNotFound} \quad (\hat{h}, \hat{d}) = \mathit{unwrap}(\hat{\sigma}) \qquad a = \texttt{to\_aloc}\ \hat{i} \qquad a \neq \bot \qquad a \notin \mathrm{dom}(\hat{h})}{\texttt{get}(\hat{\sigma}, \hat{i}, \texttt{MATCH}) \rightsquigarrow (\mathrm{aloc}(a), \bot, [\hat{i} \notin \{\mathrm{aloc}(a') : a' \in \mathrm{dom}(\hat{h})\} \wedge \hat{i} \in^? \hat{d}])}$$

$$\frac{\text{ALocPMapGetNoMatchNew} \quad (\hat{h}, \hat{d}) = \mathit{unwrap}(\hat{\sigma}) \qquad a = \texttt{to\_aloc}\ \hat{i} \qquad a = \bot \qquad a' = \texttt{fresh\_aloc ()}}{\texttt{get}(\hat{\sigma}, \hat{i}, \texttt{NO\_MATCH}) \rightsquigarrow (\mathrm{aloc}(a'), \bot, [\hat{i} = \mathrm{aloc}(a') \wedge \hat{i} \in^? \hat{d}])}$$

$$\frac{\text{ALocPMapGetMatchNew} \quad (\hat{h}, \hat{d}) = \mathit{unwrap}(\hat{\sigma}) \qquad a = \texttt{to\_aloc}\ \hat{i} \qquad a = \bot \qquad a' = \texttt{fresh\_aloc ()}}{\texttt{get}(\hat{\sigma}, \hat{i}, \texttt{MATCH}) \rightsquigarrow (\mathrm{aloc}(a'), \bot, [\hat{i} = \mathrm{aloc}(a') \wedge \hat{i} \notin \{\mathrm{aloc}(a'') : a'' \in \mathrm{dom}(\hat{h})\} \wedge \hat{i} \in^? \hat{d}])}$$

Figure 3.20: Extract of rules for `get` in $\text{PMap}_{\text{ALoc}}$

indeed the SMT solver is not needed to match the given abstract location with the locations in the heap: syntactic equality is sufficient. Its set of symbolic elements is very similar to PMap, but with strings as keys (the name of the abstract location):

$$\text{PMap}_{\text{ALoc}}(\mathbb{S}) \triangleq \text{Str} \xrightarrow{fin} \mathbb{S}.\Sigma \times \mathcal{P}(\text{LVal})^?$$

To take advantage of matching, the definition of `get` is modified, adding an argument $m \in \{\texttt{MATCH}, \texttt{NO\_MATCH}\}$. Two rules, in their matching and non-matching form, are shown in Figure 3.20. The only difference between the two is the inclusion in the path condition of $\hat{i} \notin \{\mathrm{aloc}(a'') : a'' \in \mathrm{dom}(\hat{h})\}$, such that the branch is only valid if we're *sure* that the address is not already in the heap. This cannot be included when matching is disabled, because it is too strong a condition: when not matching we already do not attempt to match the abstract location to any of the other indices, so if it results that we skipped a correct branch then the action would vanish (result in no branches), which is not OX sound.

This optimisation is particularly powerful in the case where a value that is not known to be equal to any already present location is accessed: for a heap of size n, PMap would branch $n + 1$ times, whereas $\text{PMap}_{\text{ALoc}}$ does not branch (if not matching), from the rule ALocPMapNoMatchNew. However, this is also too strong an optimisation. Consider the state model PMap(Loc, PMap(Str, Ex(Val))): the state maps locations to objects, which are string-value mappings[9]. It exposes predicates of the form $\langle \texttt{ex} \rangle(a, n; x)$, where a is the address in the heap, n is the name of the attribute, and x its value. Now

[9]This is very similar to the JavaScript state model.

$$\frac{\begin{array}{c}\textsc{ALocPMapSymInterpretation} \\ \forall a \in \mathrm{dom}(\hat{h}).\, [\![\mathrm{aloc}(a)]\!]_{\theta,s} = i \land i \in \mathrm{dom}(h) \land \theta, s, h(i) \models \hat{h}(a) \\ [\![\{\mathrm{aloc}(a) : a \in \mathrm{dom}(\hat{h})\}]\!]_{\theta,s} = \mathrm{dom}(h) \qquad [\![\hat{d}]\!]_{\theta,s} = d\end{array}}{\theta, s, (h, d) \models (\hat{h}, \hat{d})}$$

Figure 3.21: Symbolic state satisfiability for $\text{PMap}_{\text{ALoc}}$

consider the following precondition:

$$\langle \mathsf{ex} \rangle(\hat{a}_1, \text{`val'}; \hat{x}) * \langle \mathsf{ex} \rangle(\hat{a}_2, \text{`len'}; \hat{n})$$

When producing this precondition with PMap, we get two branches: one where $\hat{a}_1 = \hat{a}_2$ and both properties live in the same object, and one where $\hat{a}_1 \neq \hat{a}_2$. However with $\text{PMap}_{\text{ALoc}}$ this only results in one branch: both properties are empty and *the path condition is empty*. In the current definition of symbolic interpretation for $\text{PMap}_{\text{ALoc}}$ (shown in Figure 3.21), this is not OX sound. Substitutions play a role here, because if at a later point in execution the engine finds $\hat{a}_1 = \hat{a}_2$ then it may do a substitution and merge the two states, however this is not guaranteed. A solution to this would be to redefine satisfiability of $\text{PMap}_{\text{ALoc}}$, to take into account the merging of states. Informally, this would mean that a symbolic state models a concrete state if for all concrete addresses, the *composition of all symbolic substates at matching symbolic addresses* is modelled by the concrete substate. For example, the symbolic state $[\hat{a}_1 \mapsto [\text{`val'} \mapsto \hat{x}], \hat{a}_2 \mapsto [\text{`len'} \mapsto \hat{n}]]$ would then satisfy a concrete state $[a \mapsto [\text{`val'} \mapsto x, \text{`len'} \mapsto n]]$[10] for an empty path condition, since we could compose the two substates together (assuming again that composition is defined for symbolic states, which it isn't in the current formalism).

In fact, this unsoundness is also visible when attempting to prove Get OX Soundness, and another similar issue occurs in the proof for Get UX Soundness. In both cases, the issue only occurs when the matching mode is `NO_MATCH`, confirming this is the source of the problems.

In Defense of Abstract Locations

As shown above, there is currently an unsoundness in optimisations using abstract locations, due to the *matching mode*, and in particular when it is disabled. This occurs because the engine eagerly assumes difference, when it shouldn't always. We argue here that this behaviour, while currently unsound, can be made sound, and has a practical justification.

Consider a simple linear heap, $\text{PMap}(\text{Loc}, \text{Ex}(\text{Val}))$, with the core predicate $\langle \mathsf{ex} \rangle(\hat{i}; \hat{x})$, that we denote $\hat{i} \mapsto \hat{x}$. In a heap $\hat{i}_0 \mapsto \hat{x}_0 * \ldots * \hat{i}_n \mapsto \hat{x}_n$, one's intuition is of course that all locations are different, and indeed both $\text{PMap}(\text{Loc}, \text{Ex}(\text{Val}))$ and $\text{PMap}_{\text{ALoc}}(\text{Ex}(\text{Val}))$ will create the same state when producing these assertions. In the optimised case, because each index will be new when producing the associated core predicate, $\text{PMap}_{\text{ALoc}}$ will always generate a new abstract location to associate to it, and consider

[10] We take some liberties when representing the states, for the sake of readability, by omitting the domain sets.

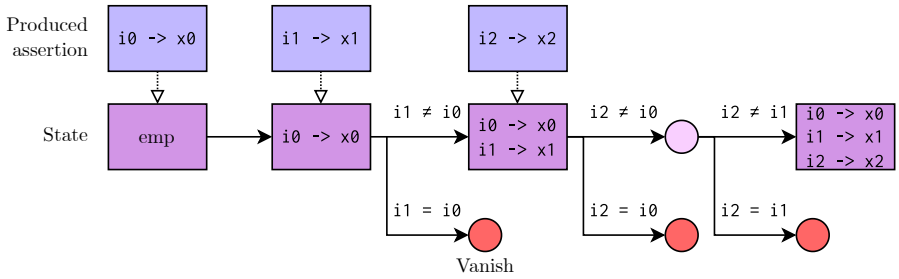


Figure 3.22: Branching when producing cells

that it is different from all other locations. This is done *implicitly* – that is, the path condition is not extended to take this into account. The base version, on the other side, will need to branch for each new assertion. Because these assertions do not already explicitly specify that all addresses are distinct ($\hat{i}_0 \neq \hat{i}_1 \wedge \hat{i}_0 \neq \hat{i}_2 \wedge ...$), the branch $\hat{i}_0 = \hat{i}_1$ is satisfiable; however of course producing $\hat{i}_1 \mapsto \hat{x}_1$ into a state $[\hat{i}_0 \mapsto \hat{x}_0]$ when $\hat{i}_0 = \hat{i}_1$ will vanish. This then happens for each address combination; for n addresses, this results in $\frac{n(n+1)}{2}$ inequalities. This makes the PC grow quadratically, slowing down queries sent to the SMT solver, as well as branching and vanishing right after just as many times (see Figure 3.22).

One could of course specify, along with the assertions, that all addresses are distinct – this is however quite effortful, and detracts from the already complex task of formulating function specifications.

Furthermore, one can't design a general map state model that makes this assumption – while for a simple case like PMap(Loc, Ex(Val)) it is always true that different cells must have different addresses (since the cell is exclusively owned), this is not true for more complex state models. For instance take PMap(Loc, PMap(Str, Val)), where the map contains objects, which are string-value pairs. When describing two objects with non-overlapping keys, if the addresses are not explicitly said to be different then both the branches where they're distinct and where they're separate should be considered. While these two branches should work the same, it results that disabling this abstract location optimisation when producing (i.e. enabling matching on `produce`) results in several functions not verifying with Gillian, in WISL, C and JS. It thus seems that this lack of completeness is at least in part caused by developer intuition, where one "knows" when writing the specification that the locations should be distinct, without formalising it, and instead relying on this unsound behaviour. This problem seems partly caused by recursive user predicates, where addresses are "discovered" as they are unfolder; these locations are, before the unfolding, not named, and as such cannot be stated to be distinct from the others.

This difference between intuition and actual application to a verification setting seems to be caused by "Barendregt Variable Convention" [60], where variables named differently are informally assumed to be different, which can become an issue for an engine that does not properly take it into account. A solution to this would be to better formalise the principles governing abstract locations, for instance using *Nominal Logic* [55, 60]. It provides a sound framework for reasoning about syntactic equivalence of terms within recursive structures, such as recursive user defined predicates.

Chapter 4

Implementation

We now present the implementation of the state models and state model transformers presented. In §4.1, we present the library which exposes these state models, some relevant implementation details, and notable differences with the theory. In §4.2, we then present the instantiations of Gillian for three different target languages using this library.

4.1 State Model Library

All of the presented state models have been implemented in OCaml 5.2 [41] for Gillian [26, 45, 59], a CSE engine that is parametric on the state model. We first present the general interface for state models, and how it connects to the Gillian engine. We then look more in depth into the implementation of two state models: Ex for its canonical simplicity, and PMap for its complexity. Finally, we list some additional "utility" state models that were omitted in the previous chapter but that are frequently used when instantiating state models.

4.1.1 State Model Interface

State models are built as standalone *modules*, while state model transformers are *functors*, that accept one or more state model modules, and return a new state model module. Gillian already provides an interface for state models, `MonadicSMemory`, whose name originates from its extensive use of the `Delayed` monad: a *symbolic execution monad* that allows branching and SMT queries in a sugared syntax that is easy to read and use, as seen in Figure 4.1. The signature of the monad, in OCaml syntax, is `'a Delayed.t = Pc.t -> ('a * Pc.t) list`: given a PC, we return a list of branches, containing the new value and an updated PC. We make extensive use of *let bindings*:[1] `let* v = d in e` is the `bind` operator, passing the value `v` contained in the delayed element `d` to the expression `e`, which must result in another delayed value. `let+` is the `map` operator, which modifies the value without branching. For the code shown,

[1] See https://ocaml.org/manual/5.2/bindingops.html

```
let produce pred s args =
  match (pred, args) with
  | SubPred pred, idx :: args →
    let* ss = validate_index s idx in
    let+ ss' = S.produce pred ss args in
    update_entry s idx ss'
```

Figure 4.1: Example of usage of the `Delayed` monad

we branch when validating the index, and then branch again when calling `S.produce`; we then update the entry with `ss'`, without branching.

The interface provided by Gillian is not satisfactory: it is quite complex, requiring the implementation of 26 different functions whose purpose is at times unclear.[2] We thus create a simpler interface, `MyMonadicSMemory`, reducing the number of required functions and type definitions, and simplifying the definition of some functions (removing arguments when unneeded). Gillian also passes actions and core predicates as strings; while this is manageable for monolithic state models, this is unpractical. We instead require state models to define `action` and `pred` types, along with their `to_string` and `from_string` parsers, ensuring decoupling between parsing and execution. See Table 4.1 for a side-by-side comparison of the interfaces. Because Gillian still requires the state model to satisfy `MonadicSMemory`, we provide a `Make` functor converting from our interface to Gillian's.

Most of the removals are possible because some default function can be used for all state models, under the assumption they behave a particular way; for instance, we remove the `copy` function, because we assume all state models only work on immutable code. Most of the additions are due to the modular nature of our state models, or to enforce a stronger typing of actions and predicates.

A significant difference with the theory presented is that the `execute_action`, `consume` and `produce` functions do not take as input states of type `t option` (to imitate $\hat{\Sigma}^?$), but instead `t`. While both options are possible, using the `option` monad proved to at times be somewhat unweildy (for the same reason it is unpractical to use *wrap* and *unwrap* in the rules) – we instead rely on the `is_empty` function when necessary. We also require a `compose` function, despite it not being justified in the theory, because of substitutions.

One of the most significant improvements of our interface is the re-work of fixes in the engine, to behave precisely like the definition of `fix` in the theory. Gillian uses an intermediary `fix_t` type, that is obtained from an error with `get_fixes`, and must then be applied with `apply_fix: t -> fix_ t -> (t, err_t) result Delayed.t`. Aside from being an unpractical system to use, generating a fix and then applying it separately, it *introduces unsoundness into the bi-abductive engine*. Indeed, applying a fix modifies the state directly; there is no requirement for this fix to be describable in terms of assertions. This means that applying the same fix on the state and the anti-frame may give different results, that do not result in the appropriate change for the function's generated pre-condition.

[2] This is due to the fact Gillian has been in development for several years, and as it grew some unused functions were never cleaned up.

Table 4.1: Difference between Gillian's and our state model interface

Gillian interface	**Our interface**	**Outcome**
apply_fix		Removed
clean_up		Removed
clear		Removed
copy		Removed
get_failing_constraint		Removed
get_init_data		Removed
get_print_info		Removed
is_overlapping_asrt		Removed
mem_constraints		Removed
pp_by_need		Removed
pp_c_fix		Removed
split_further		Removed
sure_is_nonempty		Removed
alocs	alocs	
assertions	assertions	Typed the core predicates
can_fix	can_fix	
consume	consume	Typed the core predicate
execute_action	execute_action	Typed the action
get_fixes	get_fixes	Simplified, use assertions
get_recovery_tactic	get_recovery_tactic	Simplified
init	empty	Renamed
lvars	lvars	
pp	pp	
pp_err	pp_err	
produce	produce	Typed core predicate
substitution_in_place	substitution_in_place	
	action_from_str	Added
	action_to_str	Added
	assertions_others	Added
	compose	Added
	instantiate	Added
	is_concrete	Added
	is_empty	Added
	is_exclusively_owned	Added
	pred_from_str	Added
	pred_to_str	Added

For instance in [2], an example implementation of PMap is given, where executing an action on an empty cell creates a `Miss`, called `MissingBinding`. A naive implementation of PMap for Gillian could resolve this fix by simply adding a binding that points to the empty substate in the heap at the corresponding index. This however is not sound, as it *cannot be expressed as an assertion*: it manipulates the representation of state, but no assertion exists that creates an empty binding, since PMap doesn't define core predicates for bindings – it only lifts the core predicates of the wrapped state model. This would therefore result in a fix that doesn't change the anti-frame or the specification of the function, which is invalid.

We fix this by enforcing `fix_t` to be of type `MyAsrt.t`: strongly typed assertions. The implementation of `apply_fix` then produces all the fixes into the state model. This change thus not only makes state models less prone to error, but also simplifies their code.

4.1.2 Exclusive

We now present the implementation of the Ex state model, step by step (skipping the uninteresting details such as parsing or auxiliary functions). Firstly, we must define the types for the state model: the state type, the errors (which include misses), the actions and core predicates. This is trivial:

```
type t = Expr.t option
type err_t = MissingState
type action = Load | Store
type pred = Ex
```

Here we define states as being an optional expression, where the `None` case stands for $\perp$. There is also one singular error, for misses. We then must define the function of the state model: `execute_action`, `consume` and `produce`, following the rules outlined in **??**:

```
let execute_action action s args =
  match (action, s, args) with
  | _, None, _ → DR.error MissingState
  | Load, Some v, [] → DR.ok (Some v, [ v ])
  | Store, Some _, [ v' ] → DR.ok (Some v', [])
  | _ → failwith "Invalid Ex execute_action"

let consume core_pred s ins =
  match (core_pred, s, ins) with
  | Ex, Some v, [] → DR.ok (None, [ v ])
  | Ex, None, _ → DR.error MissingState
  | _ → failwith "Invalid Ex consume"

let produce core_pred s insouts =
  match (core_pred, s, insouts) with
  | Ex, None, [ v ] → Delayed.return (Some v)
  | Ex, Some _, _ → Delayed.vanish ()
  | _ → failwith "Invalid Ex produce"
```

Again, this is straightforward; a simple pattern matching suffices. `DR.ok` stands for `Delayed.return (Ok -)`, returning the input in a `Result` monad, wrapped in a `Delayed` monad. The outcome is thus `Ok` – for `Err`, `LFail` and `Miss` outcomes, we use `DR.error`; the state model then provides a `can_fix` function that given an error returns whether it is fixable, signifying it's a `Miss`. Experiments were done to create an `Outcome` monad, with variants `Ok | Miss | Err | LFail`, as a replacement for the `Result` monad with

`Ok | Err`. While this allowed getting rid of the `can_fix` function, it was unpractical to use, as it was purely "aesthetic"; the engine still needed to parse everything back to a `Result`. Furthermomre, this made lifting errors quite tedious: instead of simply defining a variant `SubError of S.err_t` for the error type and lifting the errors of the underlying state model through it, one had to handle all three error cases, creating a lot of boilerplate code for little to no added value.

Finally here we mention some of the additional functions that the state model must implement:

```
let empty () = None
let is_empty = Option.is_none
let is_exclusively_owned = Option.is_some
let instantiate = function
  | [] → (Some (Expr.Lit Undefined), [])
  | [ v ] → (Some v, [])
  | _ → failwith "Invalid Ex instantiation"
let assertions = function
  | None → []
  | Some v → [ (Ex, [], [ v ]) ]
let can_fix MissingState = true
let get_fixes MissingState =
  [ [ MyAsrt.CorePred (Ex, [], [ LVar (Generators.fresh_svar ()) ]) ] ]
```

These functions are, in order: the constructor for the empty ($\bot$) state, a function to tell if the state is $\bot$, the `is_exclusively_owned` presented before, the `instantiate` function required by PMap, a function to return all the core predicates associated with a state, the function to tell apart `Err` and `LFail` from `Miss`, and the `fix` function presented in chapter 3. For `fix`, we note that specifying the existential for the fix $\exists \hat{x}.\, \langle \mathsf{ex} \rangle(; \hat{x})$ is not needed, as all unbound logical variables in Gillian are existentially quantified by default.

This is – skipping the irrelevant functions – all of the code for a simple state model in Gillian, using our interface `MyMonadicSMemory`. The entire code is 77 lines long, with the biggest function taking 9 lines. This shows that for simple state models, our interface allows for just as simple implementations.

4.1.3 Partial Map

We now present some details of the PMap$(I, \mathbb{S})$ implementation. It is a functor, receiving a `PMapIndex` module and a `MyMonadicSMemory` module, corresponding to the I and $\mathbb{S}$ arguments of the constructor. Both PMap and DynPMap are implemented as one state model; what enables the "dynamic" behaviour is the value `mode : Static | Dynamic` that is contained in the input index. The only difference in behaviour this causes is enabling the `alloc` action in static mode, and catching any `NotAllocated` errors and replacing them with an implicit allocation in dynamic mode. Aside from this, `PMapIndex` has an `is_valid_index`, which returns the "actual" index associated with an input index. This is where the abstract location optimisations plays a role: for the `LocationIndex` module, it is:

```
module LocationIndex : PMapIndex = struct
  let mode = Static

  let make_fresh () =
    let loc = ALoc.alloc () in
    Expr.loc_from_loc_name loc

  let is_valid_index = function
```

```
    | (Expr.Lit (Loc _) | Expr.ALoc _) as l → Delayed.return (Some l)
    | e when not (Expr.is_concrete e) → (
       let* loc = Delayed.resolve_loc e in
       match loc with
       | Some l → Delayed.return (Some (Expr.loc_from_loc_name l))
       | None →
          let loc' = make_fresh () in
          Delayed.return ~learned:[ (loc' #== e) ] (Some loc'))
    | _ → Delayed.return None
end
```

We see here how `Delayed.resolve_loc`, named `to_aloc` in §3.5, may not find a result, in which case we generate a new fresh abstract location to assign to the expression, by extending the PC: "`~learned: [ (loc' #== e) ]`" appends the expression `loc' = e` to it, with `#==` signifying the symbolic equality.

This is only one part of the implementation for what was called `get` in the theory. The rest of it is two functions:[3]

```
let validate_index (h, d) idx =
  let* idx' = I.is_valid_index idx in
  match idx' with
  | None → DR.error (InvalidIndexValue idx)
  | Some idx' → (
     let* match_val = ExpMap.sym_find_opt idx' h in
     match (match_val, d) with
     | Some (idx'', v), _ → DR.ok (idx'', v)
     | None, None → DR.ok (idx', S.empty ())
     | None, Some d →
        if%sat Formula.SetMem (idx', d) then DR.ok (idx', S.empty ())
        else DR.error (NotAllocated idx'))
```

This is the function in PMap responsible for retrieving the substate associated to an index; it calls `ExpMap.sym_find_opt`, which symbolically tries to match an index against all entries of the map, and if that finds no result it either returns an empty substate $\perp$ with `S.empty ()`, or raises an error if a bound exists and $\hat{i}' \notin \hat{d}$. Here `if%sat` is sugared syntax to allow branching according to a symbolic formula, using `Delayed`. If both the formula and its negation can be true then the value it returns evaluates to two branches, extended with the formula or its negation in the PC.

```
module ExpMap = struct
  include Map.Make (Expr)

  let sym_find_opt k m =
    match find_opt k m with
    | Some v → Delayed.return (Some (k, v))
    | None →
       let rec find_match = function
         | [] → Delayed.return None
         | (k', v) :: tl →
            if%sat k' #== k then Delayed.return (Some (k', v))
            else find_match tl
       in
       find_match (bindings m)
end
```

This last function is the symbolic matching function, which checks if $\mathsf{SAT}_{\theta,s}(\hat{i} = \hat{i}')$ for every index $\hat{i}'$ in the map, and creates a branch for every possible match. It also does *syntactic matching*, as seen at the start of the function: it first attempts to find the matching key directly, using `Map.find_opt`, and only if that fails does it proceed to match with every key.

[3] We omit some optimisations for clarity.

The `Delayed` monad and the associated syntactic sugar, initially described in [2], allow us to write branching-heavy code with ease, abstracting the details of the PC away from us. We finally take a brief look at the implementation of `produce`, to show how a state model transformer accesses the wrapped state model via the interface we defined:

```
let produce pred s args =
  match (pred, args) with
  | SubPred pred, idx :: args →
      let*? idx, ss = validate_index s idx in
      let+ ss' = S.produce pred ss args in
      update_entry s idx idx ss'
  | DomainSet, [ d' ] → (
      match s with
      | _, Some _ → Delayed.vanish ()
      | h, None → Delayed.return (h, Some d'))
  | _ → failwith "Invalid PMap produce"
```

Here again, and even for a transformer as complex as PMap$(I, \mathbb{S})$, the code is short and minimal. For the case of a core predicate from $\mathbb{S}$ (`SubPred pred`), we get the substate associated with the index (`let*?` gets the value associated with a `Result` and vanishes if it's an error), we produce the predicate onto the substate, and then update the entry. For the domain set, we simply add it to the state, vanishing if a domain set is already present as it is exclusively owned.

The entire PMap code is 343 lines of code – a sizeable amount, but still rather short given its complexity.

4.1.4 Helper State Models

We now define some additional state model transformers that are often used when one wants to instantiate a state model with the aforementioned library. These transformers focus on applying small, customisable modifications to state models, allowing for an even greater re-use. In particular these utility state model transformers are useful when instantiating our state model with the goal to mimic a pre-existing state model.

ActionAdd$(\mathbb{S}, A)$ is a transformer that equips the given state model with additional actions defined by A, without needing to define any predicates, overriding functions, or re-implementing repetitive functions.

Filter$(\mathbb{S}, \mathcal{A}, \Delta)$ is a transformer that allows filtering the actions and predicates exposed by a state model, for instance to limit the functionality of a state model if one of its provided actions shouldn't exist in the semantics of the language.

Mapper$(\mathbb{S}, F_{\mathcal{A}}, F_{\Delta})$ allows renaming predicates or actions.

Injector$(\mathbb{S}, I)$ adds hooks into `consume`, `produce` and `execute_action`, allowing the developer to apply transformations to the inputs or outputs of these functions. For instance, this makes re-ordering the arguments of actions trivial, or allows treating an out-value as an in-value that makes the branch vanish if it doesn't correspond.

4.2 Instantiations

We now look at how this library of state models can be used to instantiate Gillian. In particular we focus on instantiating Gillian to target languages (TLs) that have already been instantiated for it: WISL, JavaScript and C.

To instantiate a language to Gillian, the developer must provide its state model, and a compiler from the TL to GIL. Because this project focuses on constructing state models, it is thus natural that we picked languages for which the compiler already exists.

We note that all forms of $\textsc{PMap}(\text{Loc}, \mathbb{S})$ implicitly make use of the abstract location optimisation described in §3.5. This is because all of Gillian's instantiations also make use of it, and the goal of our instantiations is to have *parity* with the reference implementation, rather than to have additional completeness. Indeed, while removing the unsoundness caused by ALocs is straightforward in our library, it causes a majority of tests to fail, making any comparison with Gillian impossible.

4.2.1 WISL

WISL (WhIle language for Separation Logic) is an imperative toy language used to teach students the principles of software verification. It uses a simple block-offset memory:

$$\textsc{PMap}(\text{Loc}, \textsc{Freeable}(\textsc{List}(\textsc{Ex}(\text{Val}))))$$

Unsurprisingly for a simple state model like this, its instantiation is also trivial when brought to code:

```
1  module ExclusiveNull = struct
2    include Exclusive
3
4    let instantiate = function
5      | [] → (Some (Expr.Lit Null), [])
6      | [ v ] → (Some v, [])
7      | _ → failwith "ExclusiveNull: instantiate: too many arguments"
8  end
9
10 module BaseMemory = PMap (LocationIndex) (Freeable (MList (ExclusiveNull)))
11
12 module WISLSubst : NameMap = struct
13   let action_substitutions =
14     [
15       ("alloc", "alloc");
16       ("dispose", "free");
17       ("setcell", "store");
18       ("getcell", "load");
19     ]
20
21   let pred_substitutions =
22     [ ("cell", "points_to"); ("freed", "freed"); ("bound", "length") ]
23 end
24
25 module WISLMonadicSMemory = Mapper (WISLSubst) (BaseMemory)
```

This is all the code needed for the instantiation of WISL to match exactly Gillian's original monolithic instantiation.

A consequence of our modular approach to state models is functor-heavy code, with many intermediary modules to allow for further constructions. This can be a bit unweidly if not organised properly, which explains why here we take some intermediary steps, instead of writing down one large and confusing functor application.[4]

We now describe, module by module, how the instantiation of WISL is done. We first override the Ex state model, to modify its default value on instantiation, as WISL requires instantiation to `null` and the base implementation of Ex initialises it to `undefined` instead. We then create the intermediary module `BaseMemory`, equivalent to $\textsc{PMap}(\text{Loc}, \textsc{Freeable}(\textsc{List}(\textsc{Ex})))$,[5] which corresponds to the bulk of the state model. We finish by applying the substitutions needed to match the target instantiation, via the `Mapper` module, and voilà! Thanks to our library, what used to be about 800 lines of code is now 25 lines. We point out that our instantiation is also easier to verify the soundness of, as all modules are independent and don't require understanding how the other state model transformers work.

4.2.2 JavaScript

Gillian originated from JaVerT [28, 29], which eventually became Gillian-JS, a JavaScript instantiation for Gillian, which targets ECMAScript 5.1 [23]. Its state model is:

$$\textsc{PMap}(\text{Loc}, \textsc{DynPMap}(\text{Str}, \textsc{Ex}(\text{Val})) \times \textsc{Ag}(\text{Loc}))$$

Here $\textsc{DynPMap}(\text{Str}, \textsc{Ex}(\text{Val}))$ represents objects, and $\textsc{Ag}(\text{Loc})$ is the location of the object's metadata, which itself is also an object. An object's metadata is immutable, and thus shareable, explaining the choice of an agreement.

Its implementation with our library is more involved than for WISL; indeed, Gillian-JS doesn't quite line up with the actions and predicates we defined for our state models. These differences mostly stem from different design, meaning most of the work required to fix them is juggling values to fit the right shape.

For instance in Gillian-JS the `domainset` predicate has one in-value, the domain set, and no out. In our definitions however, the same predicate has no in, and one out, the domain set. To this end, we override `consume` for `domainset`, and vanish when the given in doesn't match the out.

```
1  module MoveInToOut (S : States.MyMonadicSMemory.S) :
2    States.MyMonadicSMemory.S with type t = S.t = struct
3    include S
4
5    let consume pred s ins =
6      match (pred_to_str pred, ins) with
7      | "domainset", [ out ] → (
8          let** s', outs = S.consume pred s [] in
9          match outs with
10         | [ out' ] →
11             if%sat out #== out' then Delayed_result.ok (s', [])
```

[4] It is often not possible to write the whole state model as a single functor application anyways, due to constraints in OCaml abount naming modules – see https://github.com/ocaml/ocaml/issues/6917#issuecomment-626586129.

[5] The List state model transformer is called `MList` to avoid shadowing OCaml's `List` module.

```
            else Delayed.vanish ()
        | _ → Delayed_result.ok (s', outs))
    | _ → consume pred s ins
end
```

Another difference is that for the DynPMap, when accessing a field that didn't exist, Gillian-JS instantiates it to `Nono`, a special value to represent values that are not actually there. When using the `get_domainset` action or consuming the `domainset` predicate, all keys mapping to a `Nono` value must thus be removed:

```
module PatchDomainsetObject (S : sig
  include MyMonadicSMemory
  val is_not_nono : t → Expr.t → bool
end) =
  Injector
    (struct
      include DummyInject (S)

      let post_execute_action a (s, args, rets) =
        match (a, rets) with
        | "get_domainset", [ Expr.EList dom ] →
            let dom = List.filter (S.is_not_nono s) dom in
            Delayed.return (s, args, [ Expr.EList dom ])
        | _ → Delayed.return (s, args, rets)

      let post_consume p (s, outs) =
        match (p, outs) with
        | "domainset", [ Expr.ESet dom ] →
            let dom = List.filter (S.is_not_nono s) dom in
            Delayed.return (s, [ Expr.ESet dom ])
        | _ → Delayed.return (s, outs)
    end)
    (S)
```

This is done via an injection, that receives a state model *enhanced with the* `is_not_nono` *function*; we do this instead of hardcoding the lookup to ease the process of instantiating the JavaScript stack with different optimisations of PMap that may not have the same state type. An example implementation is then:

```
module ObjectBase = struct
  include PMap (StringIndex) (Exclusive)

  let is_not_nono (h, _) idx =
    match ExpMap.find_opt idx h with
    | Some (Some (Expr.Lit Nono)) → false
    | _ → true
end
```

The entire instantiation to JavaScript is 253 lines of code, including all versions of the state model with different optimisations enabled. Most of these lines are used on functor and module definitions, to build up the state one modification at a time. This type of code is hard to read and understand, but this complexity is not caused by the target state model as much as the need for matching the Gillian-JS interface; if the compiler from JavaScript to GIL was reworked to be more aligned with our state model definitions, the instantiation of JavaScript would likely be much simpler and succinct.

4.2.3 C

Gillian-C [59] targets a subset of C, via the CompCert-C verified compiler [40]. Due to the intricacies of the C memory model, its construction uses custom components, not part of the shared library. Its state

model is:

$$\textsc{PMap}(\textsc{Loc}, \textsc{Freeable}(\textsc{BlockTree})) \times \textsc{CGEnv}$$

The first custom state model used is the BlockTree component, which represents chunk of memorys as a binary tree, where each node has a range, a value, and zero or two children. Discussing the behaviour of this state model is out of the scope of this report, and is described in more detail in [2].

The second custom state model is CGEnv, the C global environment. Gillian-C handles function pointers and global variables by constructing an "initial data" when parsing the input code, and then passing this data to the state model on instantiation. While this behaviour was hard-coded in Gillian-C, we generalise it, considering simply that the state is a product of the memory and the global environment.

For the implementation of BlockTree and CGEnv, the code from Gillian-C was reused and adapted where necessary to our interface. A particular improvement that was done to it is that in Gillian-C the notion of Freeable and of BlockTree were coupled, meaning any action executed on the BlockTree had to first check if the memory had been freed before proceeding. Decoupling this behaviour was straightforward, and allowed for simpler code in BlockTree.

The bulk of the new code needed for the C instantiation lies in implementing a new action, that was previously hardcoded in the Gillian-C state model: `move`. This action allows copying some memory from a source into a target location, merging the BlockTrees when appropriate. It is quite inelegant to implement, as it cannot rely on the abstraction provided by state models, since it needs to "cross layers" between the state models: it operates on two different indices of the PMap, but applies a merge operation that only exists for BlockTree. It seems that for such *crossing actions*, our modular state model approach is not sufficient; one instead needs to implement these actions for a specific instantiation, making the proof work much more intricate.

Omitting this addition done via ActionAdd, the rest of the state model is straightforward to instantiate:

```
1  module BaseMemory = PMap (LocationIndex) (Freeable (BlockTree))
2
3  module Wrap (S : C_PMapType) =
4    Product
5      (struct
6        let id1 = "mem_"
7        let id2 = "genv_"
8      end)
9      (ExtendMemory (S))
10     (CGEnv)
11
12 module CMonadicSMemory = Wrap (BaseMemory)
```

Here `ExtendMemory` is the functor responsible for adding the `move` action to the state model. We make use of the `Wrap` functor to avoid code duplication when building the state model with different optimisations of PMap. The module defined on lines 5-8 specifies the suffixes used to distinguish actions and core predicates from the left and right side of the product.

The instantiation of C with our library is a good example of mixing modular, general use state models with specialised state models; one can still make use of this library while implementing a perhaps more performant or more complex state model for some of the construction.

Chapter 5

Evaluation

In this chapter we evaluate the work produced in this document. In §5.2 we evaluate the usability of the theory developed; in §5.2 we assess the ease of use of the developed library; in §5.3 we compare the performance of the developed instantiations with their monolithic equivalent in Gillian; finally in §5.4 we evaluate the performance improvement of the various PMap optimisations.

5.1 Theory Usability

The theory we have presented contrasts with previous work for CSE engines in its use of resource algebras rather than partially commutative monoids or separation algebras. This allows us to easily prove the soundness of the presented constructions.

Still, some aspects of the theory can be at times somewhat unpractical to work with. Representing the empty memory $\bot$ as an element outside of an RA's carrier avoids common mistakes, but comes with the cost of requiring more ceremony when writing the rules for a state model transformers: in particular, we frequently need to define *wrap* and *unwrap* auxiliary functions to allow seamlessly handling the case where the state is $\bot$, which can clutter the rules. We note however that this additional care would also be needed with the PCM approach, though it would not be forced by the framework. The tradeoff here is therefore between a more restrictive notation that avoids some errors, or a more flexible notation that can be error-prone.

The RA constructions presented are relatively primitive; in particular we only focus on simpler forms of ghost state, without exploring more advanced concepts that may make sense in a logic but not in an engine. The presented state models are sufficient to represent JavaScript, but fall short when attempting to model C, requiring specialised state models. For low-level languages where precision is required, it seems like one still needs to create a tailored state model. Other examples of this come to mind: for Gillian-Rust, the state model still needs to define the RustHeap, Lft, Pcy and Obs state models

[2, 3].[1] While some of these are partly derived from other state models (RustHeap uses PMap, Lft uses PMap and Frac), significant effort is needed to represent concepts such as liftetimes, prophecies or observations. While the presented state models all have a use, it seems unavoidable for one to need to define some additional state models when targetting real-world languages. The fact these *sur mesure* state models still use the generic constructions we defined show their versatilty, and that they are still capable of alleviating at least part of the construction of a complex state model.

5.2 Library Usability

The developed library exposes a total of 14 state models, including utility state models and optimised versions. These are all easy to use and combine when little additional functionality is required, for instance with WISL. However, as soon as fine grained modifications are needed, the constructions can become complex, as they require extensive use of OCaml functors, which often cause hard to understand compiler errors. The effort needed is still much lesser than what would be needed when creating the state model from scratch, as most behaviour is already available.

The key improvement our approach has over defining entire state models from scratch is *verifiability*. Because all state models are relatively small (less than 400 lines) and isolated, ensuring a specific construction is sound is *emp*much easier than verifying an entire state model is sound. Modifications to constructions are also relatively easy to verify, as they usually only impact a specific action or predicate, meaning one onle needs to re-verify the soundness of the modified element. This is a major improvement on monolithic approaches, as verifying tightly coupled code spanning thousands of lines is no easy task.

5.3 Comparison with Gillian Instantiations

We now measure the performance of the instantiations created using our library against the performance of the original monolithic Gillian state models.

The evaluation is focused on testing the performance of state models across all modes supported by Gillian: Whole Program Symbolic Testing (WPST), OX Verification and UX Bi-Abduction. The files tested are either small programs used to test the engine, or larger verification targets extracted from real world code. Furthermore, when possible, the tests are also run with different optimisations of PMap: $\text{PMap}_{\text{ALoc}}$ and $\text{PMap}_{\text{Split}}$.

All test logs were verified to ensure full parity between all versions: the instantiations built using state model transformers yield the same results as the original instantiations. All passing tests pass, and all failing tests fail for the same reasons. All tests were run on a 2020 MacBook Pro, with an M1 processor and 8GB of memory.

[1] These state models have not yet been implemented in our library, though they exist as a monolithic construction as part of Gillian-Rust.

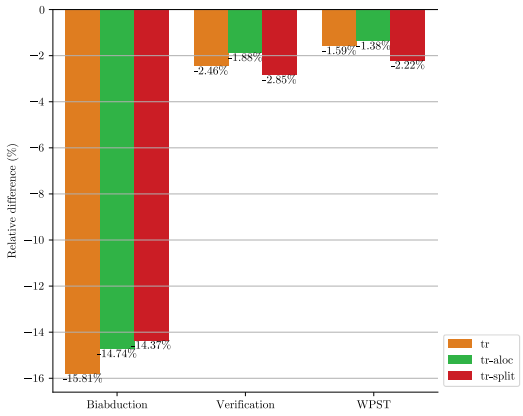


Figure 5.1: Average of the relative difference in execution time for different WISL test suites, per transformer stack

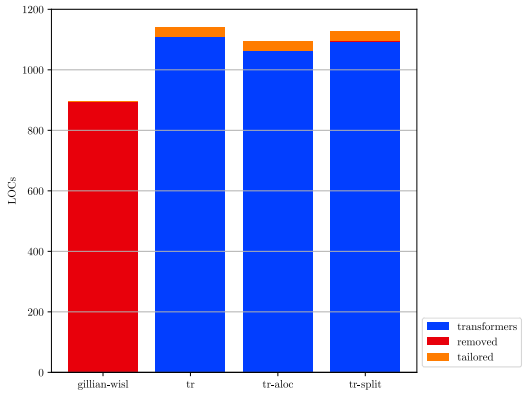


Figure 5.2: Number of lines of code per WISL instantiation

This evaluation is split among the three different stacks originally supported by Gillian: WISL, JavaScript and C. A more in-depth presentation of the instantiations is done in §4.2. We note that all Gillian instantiations make use of the abstract location optimisation internally, which as shown in §3.5 has some unsoundness issues. Therefore, to ensure our instantiations have the same results as the reference implementation, we evaluate our transformer constructions using the ALoc optimisation where relevant.

5.3.1 WISL

The WISL state model is $\textsc{PMap}(\text{Loc}, \textsc{Freeable}(\textsc{List}(\textsc{Ex}(\text{Val}))))$. The Gillian-WISL instantiation uses the abstract location optimisation, so our instantiations also do; using PMap without the optimisation would yield different test outcomes, due to its unsoundness. To measure the performance impact of different optimisations, we build the following transformer stacks:

$$\textsc{PMap}_{\text{ALoc}}(\textsc{Freeable}(\textsc{List}(\textsc{Ex}(\text{Val})))) \quad \text{(TR)}$$

$$\textsc{PMap}^{\text{Str}}_{\text{ALoc}}(\textsc{Freeable}(\textsc{List}(\textsc{Ex}(\text{Val})))) \quad \text{(TR-ALoc)}$$

$$\textsc{PMap}_{\text{ALocSplit}}(\textsc{Freeable}(\textsc{List}(\textsc{Ex}(\text{Val})))) \quad \text{(TR-Split)}$$

Here $\textsc{PMap}_{\text{ALocSplit}}$ represents a PMap with both the ALoc and the split optimisations.

The benchmark is split into three parts: WPST, verification and bi-abduction, which are made up of respectively 1, 6 and 2 files. We note that the benchmark is small because it only has small code samples used for teaching purposes, rather than any real world code (since WISL is used for teaching). For this reason also we will not delve into too much detail for this comparison. The tests were run 50 times each, and the results are shown in Figure 5.1.

The performance here seems to be worsened by the use of our library. This makes sense: WISL has a small toy-like state model, allowing for relatively simple code that can work straightforwardly. Gillian-

WISL also uses an older, more lower level (and deprecated) API for Gillian, allowing it for instance to manipulate the PC without using `Delayed`. While this approach is more error-prone, it works well for small state models, and explains the better performance observed here. In comparison, using state model transformers is probably overkill and the added indirection worsens performance compared to a simple implementation. The difference is however negligible: no more than 3% performance loss in verification and WPST.

What is gained from using transformers is a shorter implementation, as shown in Figure 5.2: only the amount of lines of code needed to instantiate the WISL state model is minuscule. There are more LOCs needed as a whole, but the developer of the state model doesn't need to worry about the size of the library of course.

5.3.2 JavaScript

The JavaScript state model is $\textsc{PMap}(\text{Loc}, \textsc{DynPMap}(\text{Str}, \textsc{Ex}(\text{Val})) \times \textsc{Ag}(\text{Loc}))$. Gillian-JS comes with some optimisations built-in, to improve performance. Firstly, it splits the first level PMap entries between concrete and symbolic *values* (not keys!), avoiding substitutions in the concrete part. Secondly, it uses the abstract location mechanism described in §3.5. Finally, it uses the OCaml `Hashtbl` module, a mutable data structure, rather than the immutable `Map` module; this avoids creating copies of the map on modification. All of these optimisations are important, as the state in JavaScript code tends to be significantly large, with the first PMap regularly reaching 200 to 600 entries when running real-world code: the highest recorded map size reaches 1179 entries for the `set3.gil` file.

To test all combinations of optimisations available via our library, we get the following four transformer stacks:

$$\textsc{PMap}_{\text{ALoc}}(\textsc{DynPMap}(\text{Str}, \textsc{Ex}(\text{Val})) \times \textsc{Ag}(\text{Loc})) \quad \text{(TR)}$$

$$\textsc{PMap}^{\text{Str}}_{\text{ALoc}}(\textsc{DynPMap}(\text{Str}, \textsc{Ex}(\text{Val})) \times \textsc{Ag}(\text{Loc})) \quad \text{(TR-ALoc)}$$

$$\textsc{PMap}_{\text{ALoc}}(\textsc{DynPMap}_{\text{Split}}(\text{Str}, \textsc{Ex}(\text{Val})) \times \textsc{Ag}(\text{Loc})) \quad \text{(TR-Split)}$$

$$\textsc{PMap}^{\text{Str}}_{\text{ALoc}}(\textsc{DynPMap}_{\text{Split}}(\text{Str}, \textsc{Ex}(\text{Val})) \times \textsc{Ag}(\text{Loc})) \quad \text{(TR-ALocSplit)}$$

The last version, TR-ALocSplit, is the closest in terms of applied optimisations to what Gillian-JS does. The biggest difference is the use of `Hashtbl`, as all transformers use immutable data structures.

This benchmark is split into three parts: WPST, verification, and Buckets-JS[2] (in WPST mode), which are made up of respectively 21, 6, and 78 files. All tests were then run 30 times, for each state transformer stack and for Gillian-JS (labelled "base"). The results can be seen in Figure 5.3. The first insight this give us is that transformers seem to, on the whole, outperform the monolithic Gillian-JS, with an improvement ranging from 2.8 to 6.9%. We also see that the ALoc optimisation that uses strings

[2] See https://www.npmjs.com/package/buckets-js.

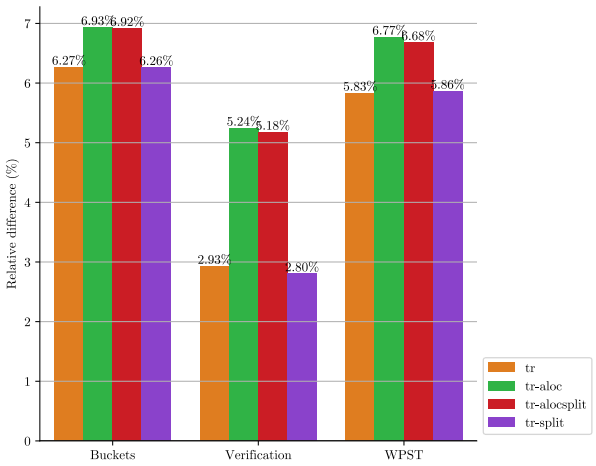


Figure 5.3: Average of the relative difference in execution time for different JavaScript test suites, per transformer stack

as keys is the best improvement; the Split optimisation seems to have no effect, hinting that maybe the cost of splitting the map and checking what is concrete is too high to compensate the time gain on actual substitutions. We also note that the improvement given by the ALoc optimisation is greater when doing verification than with WPST; this is likely caused because Verification uses more non-concrete locations, which implies less syntactic matches when doing lookups.

The initial conclusion we can reach from this is that transformer stacks are more performant than the monolithic alternative. One could, of course, imagine that a hyper-optimised monolith could be made with optimisations specific to the state model, which would then out-perform the more generic transformers. However, in practice, the size of such monoliths makes these optimisations hard to apply, as code becomes complex quickly and makes changes harder. In contrast, transformers are very simple to optimise, as they maintain a more generic structure.

Another hypothesis this experiment seems to confirm is that the improvement given by optimisations is highly dependent on the context in which the engine is used: ALoc has a much lesser impact when doing WPST than when doing verification. Transformers thus allow users to tailor the optimisations they use in their stack according to what code is verified and how, by empirically measuring the performance of different alternatives (which are trivial to construct).

We may also take a closer look at how this time is spent within the engine. This is done by measuring the time before and after the entry point of each exposed method of the memory model and summing their difference. By looking at the average time per function call in the Buckets test suite (see Figure 5.4), we see that most memory actions (prefixed with "ea/", shorthand for `execute_action`) are faster than in the base instantiation. Furthermore, copying is orders of magnitude faster with transformers, since no work needs to be done thanks to the use of immutable data structures.

Looking at the average total duration for an execution of the test suite (see Figure 5.5) however, we note that the time taken by copying the state is minimal, as seen by the size of the "other" category.

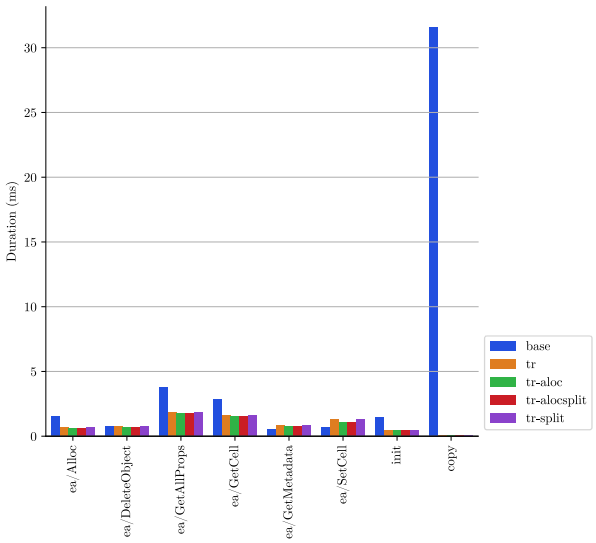


Figure 5.4: Average time spent per 1000 function calls in the Buckets test suite

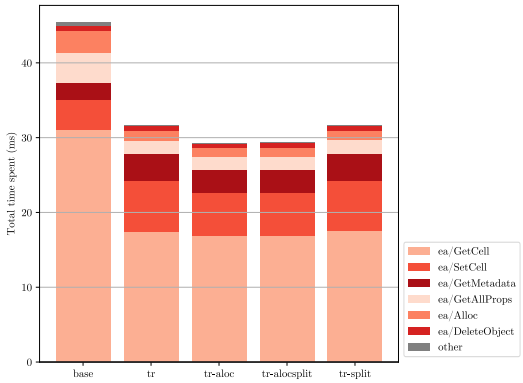


Figure 5.5: Average total time spent per function in an execution of the Buckets test suite

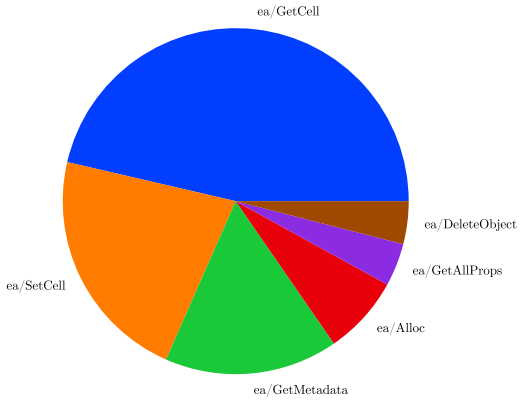


Figure 5.6: Share of function calls for one execution of the Buckets test suite

Instead, most time is spent during the `load` (`GetCell`) action, which is called 11400 times (more than twice as many times as `store`). The most notable improvement from the transformer construction is the reduction of the time spent getting a cell, reducing total time spent by 43% (from 28.55 to 16.18ms). Because `load` dominates the amount of action calls (see Figure 5.6), this is sufficient to make up for the performance loss in `store` and "`GetMetadata`" (`load` on the right side of the product).

If, instead of focusing on WPST with the Buckets code we focus on verification, the picture painted is significantly different (see Figure 5.7). Indeed, while in WPST the memory model is only used for memory actions (as only the core engine is used), verification exercises the memory model quite differently, notably with `produce` and `consume`.

Most notably, this shows us that one of the leading differences in total time between different transformer instantiations is substitutions: TR and TR-Split spend more than twice as much time during substitutions than TR-ALoc and TR-ALocSplit. This is because with the ALoc optimisation, substitutions for the keys (abstract location names) can be filtered: if there is no substitution for abstract

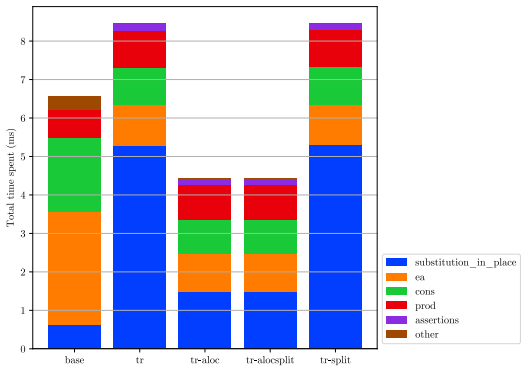


Figure 5.7: Share of grouped function calls for for one execution of the verification test suite

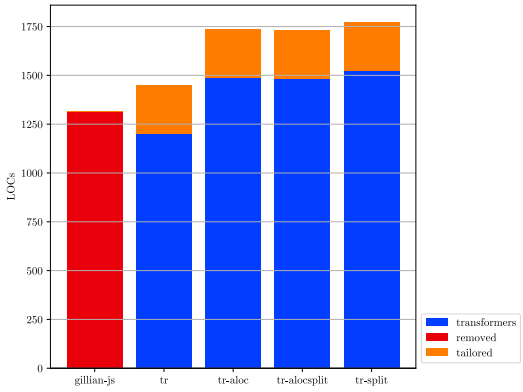


Figure 5.8: Number of lines of code per JS instantiation

locations, we can simply map the values of the heap without worrying about key conflicts. Without this, substitutions need to be applied to both the key and the value, and then all resulting keys need to be compared to compose clashing entries – this is expensive.

While this hasn't been measured, it is possible that the reason why Gillian-JS spends so little time in substitutions is because it uses mutable structures, allowing modifications to be done in-place, rather than creating new copies.

Substitutions aside, we note how significantly less time is spent in `execute_action` and `consume` calls compared to the base state model; about 65% and 48% respectively. Again, this seems to indicate that simpler transformers tend to be more efficient than large monoliths. We have not been able to pinpoint specific reasons for this. An hypothesis is that Gillian-JS overcomplicates its actions; for instance, the implementation of `GetCell` alone is over 100 lines long and is surprisingly intricate.

Finally, we may compare development effort between the two state models. While not a perfect measure of complexity, lines of code (LOCs) are used to compare the amount of code needed to instantiate each state model to get equivalent results. We only measure the lines of code in implementation files (`.ml`), ignoring interface files (`.mli`) and whitespace. The results are shown in Figure 5.8. Here, we note that the amount of tailored code needed for each instantiation (in yellow) is minimal compared to Gillian-JS: only 246 LOCs. Most of the LOCs are in the state model library, which is shared and can be reused for different state models; a user of the engine does not need to worry about them.

5.3.3 C

Gillian-C allows verifying a subset of the C language, via the CompCert-C verified compiler. Its state model can be defined as $\text{PMap}(\text{Loc}, \text{Freeable}(\text{BlockTree})) \times \text{CGEnv}$. To verify that the improvements from the optimised PMap versions are carried between instantiations, we also provide several C instantiations. Again we note Gillian-C already makes use of the ALoc optimisation, so in an effort to

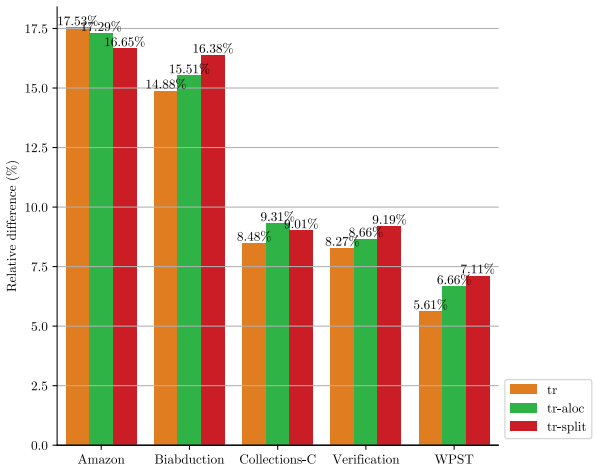


Figure 5.9: Average of the relative difference in execution time for different C test suites, per transformer stack

preserve parity we do not compare with instantiations that do not use it.

$$\textsc{PMap}_{\textsc{ALoc}}(\textsc{Freeable}(\textsc{BlockTree})) \times \textsc{CGEnv} \tag{TR}$$

$$\textsc{PMap}^{\textsc{Str}}_{\textsc{ALoc}}(\textsc{Freeable}(\textsc{BlockTree})) \times \textsc{CGEnv} \tag{TR-ALoc}$$

$$\textsc{PMap}_{\textsc{ALocSplit}}(\textsc{Freeable}(\textsc{BlockTree})) \times \textsc{CGEnv} \tag{TR-Split}$$

This benchmark is split into five parts: WPST, verification, bi-abduction, Collections-C[3] (in WPST mode) and AWS (in verification mode, for the AWS Encryption SDK[4]), each made up of respectively 8, 6, 7, 159, and 10 files. All tests were run 50 times, except the AWS test suite that was executed 10 times[5].

The general results can be seen in Figure 5.9. Here again, we note a significant performance improvement compared to the monolithic Gillian-C, in particular for the AWS and bi-abduction suites. As for the optimisations, we get inconsistent results: they seem to slightly improve performance for all modes but for AWS.

We may now look into the detailed rundown of the time spent; we will focus on Collections-C WPST and the AWS encryption SDK verification, as these correspond to real-world code usage.

For Collections-C, when looking at the time spent in each function for each instantiation (see Figure 5.10), the main takeaway is that the two most common actions, `mem_store` and `mem_load`, are both faster than the original version; about 20% and 25% respectively. This is a crucial improvement, considering more than 75% of memory actions in Collections-C is a load or a store.

The story is however wildly different for AWS code, which uses verification rather than WPST. Here, memory actions only represent a fraction of the total time spent, which is instead dominated by `consume`,

[3]See https://github.com/srdja/Collections-C.
[4]See https://github.com/aws/aws-encryption-sdk-c.
[5]This is because verifying the AWS code takes *significantly* longer than the rest, due to its size.

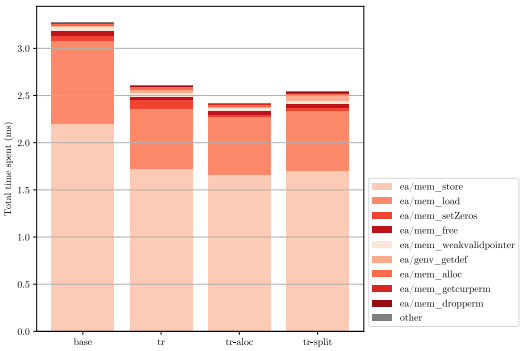


Figure 5.10: Average total time spent per function in an execution of the Collections-C test suite

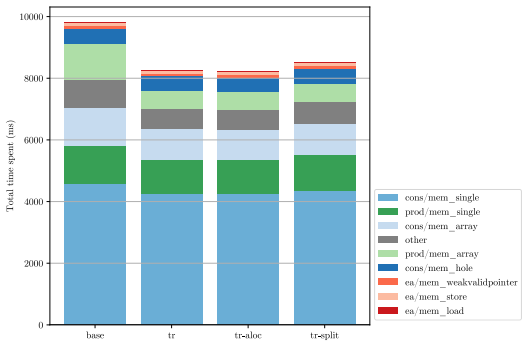


Figure 5.11: Average total time spent per function in an execution of the AWS header encryption SDK test suite

as seen in Figure 5.11. Here we note slight improvements in time for most categories, again seeming to indicate that the more generic approach is generally more performant, though the exact reason as to why is still unclear.

The charts for TR, TR-ALoc and TR-Split are extremely similar, for both Collections-C and AWS code; we can thus hardly make a hypothesis as to which optimisation is better suited. This is in part due to the fact these optimisations excel in larger maps, as was the case for JavaScript. In C, maps are instead rather small; for Collections-C, the map size tends to fluctuate between 15 and 35 elements, with the maximum recorder reaching 52 elements. For AWS, its size is between 10 and 20 elements, maxing at 20. This in particular explains why the optimisations are at times detrimental for the AWS test suite: because the maps are always small, the cost of both optimisations regularly outweighs their improvement.

We now compare the complexity of instantiations, using LOCs, delving into more details on the cause of each line count metric. In particular, we see that the majority of the state model is the same; the modified part representing BlockTree and CGEnv, which are lightly modified to fit into the transformers setup, while the untouched part represents the internal modules used by the C instantiations (for instance, to encode permissions, or memory chunks). Finally, the removed part of the code is trivially replaced with constructions, using PMap and Freeable. Some tailored code is also required, to ensure the construction matches the interface of Gillian-C. This again shows that even for more complex state models that require a significant amount of custom code, using our state model library reduces the amount of code required exclusively for that instantiation, while providing performance gains for free.

5.4 Partial Map Performance

Finally, and to confirm our intuition, we do a brief evaluation of the time taken to symbolically `get` elements from the different PMap optimisations. This is done by measuring the time at the start and end of the `validate_index` function of the various PMap implementations during the execution of the

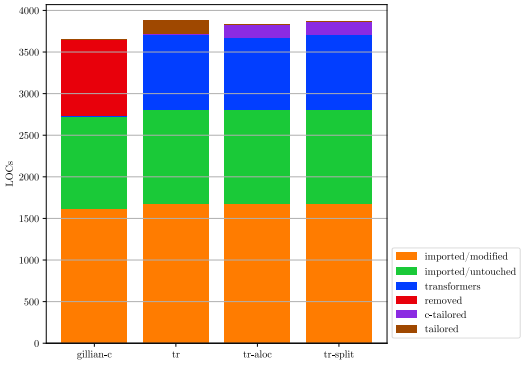


Figure 5.12: Number of lines of code per C instantiation

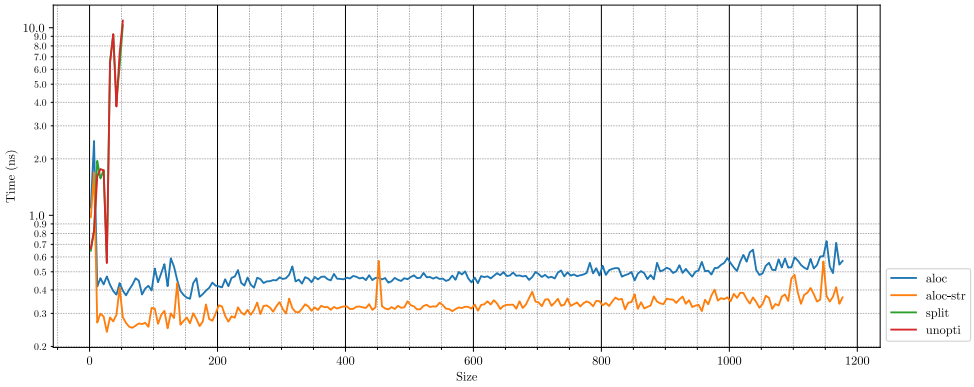


Figure 5.13: Average time taken to access states in PMap, depending on the size of the heap and the optimisations used.

Buckets-JS tests.[6] The results are shown in Figure 5.13. `unopti` is the base PMap implementation with no optimisation, `split` is $\text{PMap}_{\text{Split}}$, `aloc` is $\text{PMap}_{\text{ALoc}}$, and `aloc-str` is $\text{PMap}^{\text{Str}}_{\text{ALoc}}$ – the ALoc optimisation with strings as keys.

We first notice the significant difference between the PMap with and without the ALoc optimisation: while the data only exists for map sizes up to 50 elements, the operation takes up to 10ns without ALocs, while taking on average 0.5ns with them. The use of the split optimisation doesn't seem to have an impact in this regard.

We can also compare performance when using expressions as keys (`aloc`) against using strings (`aloc-str`); here the impact seems to be constant, with an approximate improvement of 30% when using strings. In both cases, the time taken for a lookup seems to logarithmically scale with the size, which makes sense given `Map`, the used data structure, is a balanced binary tree.

[6] We choose to do it over this sample because it represents real-world code, and because JavaScript makes use of more combinations of optimisations, since the PMap for objects does not have a Loc domain.

Chapter 6

Conclusion

6.1 Achievements

Throughout this project we adapted and developed a theory to construct state models, as well as developed a range of different state models to allow for the construction of complex state from easy to understand and modify building blocks. To this end, we introduced the concept of *partial resource algebras*, an adaptation of the resource algebras presented in Iris [34]. We adapted the notion of state models introduced for the Gillian CSE engine [**gillan0**, 2, 43] to use these partial resource algebras, and re-defined the axioms these state models must satisfy. We then introduced a range of different state models and state model transformers, taking inspiration from work done in [2, 34]; *these state models are proven to be OX and UX sound*. These state models and state model transformers can be combined to build more complex state models, allowing real-world memory models to be recreated with ease. These can then be used either when doing manual proofs or in a parametric CSE engine such as Gillian, enabling whole program symbolic testing, verification and true bug finding with bi-abduction. We also demonstrated how to optimise a state model at the theory level, to offer performance improvements in the implementation while preserving soundness.

All presented state models, along with some additional "utility" state models, have been implemented in OCaml, allowing them to be used with Gillian, providing a library of *14 different modules and functors*. These modules are small and independent, easing the process of verification and optimisation.

We have then used this library to instantiate Gillian to three different languages: WISL, JavaScript and C. We measured the difference in performance with preexisting monolithic instantiations of Gillian, showing a considerable performance improvement for JavaScript and particularly C, while greatly reducing the amount of code needed for the instantiations. Finally, we also evaluated the performance offered by the optimisations of the partial map we defined theoretically, showing mixed results.

6.2 Limitations

The presented state models are quite simple in their construction; in particular more ambitious resource algebras such as what Iris calls "higher-order ghost state" [34] have not been replicated. Overall a lot of the logic used is still embedded in Gillian, rather than defined externally via state models and core predicates. More research would be needed to see how more deeply Iris-enabled reasoning can be embedded in Gillian's logic.

The partial map state model transformer, PMap, only supports allocation in a non-concurrent setting. While not a limitation for our use case, since Gillian does not support concurrency, it is a flaw of the theory; more advanced state models could be developed to support parallel allocation while still providing guarantees of OX and UX soundness and allow distinguishing out of bounds from missing errors. This could be done, for instance, via an authoritative resource algebra [34].

Finally, instantiating Gillian using the developed state model library is still a convoluted process at times, requiring functor-heavy code, which is hard to write and understand. The developed interface for state models is also not ideal, requiring a significant amount of functions to be defined; ideally only the core `execute_action`, `consume`, `produce` and `fix` functions should be needed, the rest being optional.

6.3 Future Work

The created instantiations are satisfiable, offering full parity with the monolithic instantiations, as well as performance improvements for JavaScript and C. Future work could attempt to further improve performance, with more agressive optimisations or attempting to tie in the state models with lower level code.

Improving developer experience would also be desirable, simplifying further the implementation of new state models. An idea that comes to mind is the use of OCaml's `ppxlib` library[1] to do meta-programming and simplify some tedious requirements of the state model interface. A general rework of the Gillian engine would also be of interest, to better integrate with our state model interface.

In the future we may also want to expand on the presented concepts, with new state models to allow for higher-order logic or temporal logic. User-defined predicates and bi-abduction, two components of Gillian, can also be formulated via state model transformers [2]; formalising them and implementing them could help lighten the core engine, while facilitating the adoption of these features in new CSE engines.

Finally, we showed the system of abstract locations Gillian uses extensively is currently unsound; finding a sound justification to its use, for instance taking inspiration from nominal logic, would be crucial to reinforce Gillian's claims of soundness.

[1] See https://github.com/ocaml-ppx/ppxlib.

Bibliography

[1] 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.

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

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

[4] 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.

[5] 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.

[6] 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.

[7] 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.

[8] A. Bizjak and L. Birkedal, "On Models of Higher-Order Separation Logic," *Electronic Notes in Theoretical Computer Science*, vol. 336, pp. 57–78, 2018, The Thirty-third Conference on the Mathematical Foundations of Programming Semantics (MFPS XXXIII), ISSN: 1571-0661. DOI: https://doi.org/10.1016/j.entcs.2018.03.016. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1571066118300197.

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

[10] 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.

[11] M. Botincan, D. Distefano, M. Dodds, R. Grigore, D. E, and M. Parkinson, "coreStar: The Core of jStar," May 2012.

[12] 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.

[13] 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.

[14] 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.

[15] C. Calcagno, D. Distefano, P. W. O'Hearn, and H. Yang, "Compositional Shape Analysis by Means of Bi-Abduction," *J. ACM*, vol. 58, no. 6, 2011, ISSN: 0004-5411. DOI: 10.1145/2049697.2049700. [Online]. Available: https://doi.org/10.1145/2049697.2049700.

[16] 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.

[17] Q. Cao, S. Cuellar, and A. W. Appel, "Bringing Order to the Separation Logic Jungle," in *Programming Languages and Systems*, B.-Y. E. Chang, Ed., Cham: Springer International Publishing, 2017, pp. 190–211, ISBN: 978-3-319-71237-6.

[18] T. Coq Development Team, *The Coq Reference Manual, version 8.20*, 2021. [Online]. Available: http://coq.inria.fr/doc.

[19] *CVE-2014-0160.* Available from MITRE, CVE-ID CVE-2014-0160. Dec. 2013. [Online]. Available: https://cve.mitre.org/cgi-bin/cvename.cgi?name=cve-2014-0160.

[20] *CVE-2024-3094.* Available from MITRE, CVE-ID CVE-2024-3094. Mar. 2024. [Online]. Available: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2024-3094.

[21] 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.

[22] 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.

[23] E. Ecma, "ECMAScript® Language Specification," *ECMA (European Association for Standardizing Information and Communication Systems), pub-ECMA: adr,*, Jun. 2011. [Online]. Available: https://262.ecma-international.org/5.1/.

[24] M. Eilers, M. Schwerhoff, and P. Müller, *Verification Algorithms for Automated Separation Logic Verifiers*, 2024. arXiv: 2405.10661 [cs.PL]. [Online]. Available: https://arxiv.org/abs/2405.10661.

[25] F. Farka, A. Nanevski, A. Banerjee, G. A. Delbianco, and I. Fábregas, "On algebraic abstractions for concurrent separation logics," *Proc. ACM Program. Lang.*, vol. 5, no. POPL, 2021. DOI: 10.1145/3434286. [Online]. Available: https://doi.org/10.1145/3434286.

[26] 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.

[27] 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.

[28] 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.

[29] 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.

[30] 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.

[31] 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.

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

[33] R. Jung, R. Krebbers, L. Birkedal, and D. Dreyer, "Higher-order ghost state," *SIGPLAN Not.*, vol. 51, no. 9, pp. 256–269, 2016, ISSN: 0362-1340. DOI: 10.1145/3022670.2951943. [Online]. Available: https://doi.org/10.1145/3022670.2951943.

[34] 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.

[35] R. Jung, D. Swasey, F. Sieczkowski, K. Svendsen, A. Turon, L. Birkedal, and D. Dreyer, "Iris: Monoids and Invariants as an Orthogonal Basis for Concurrent Reasoning," *SIGPLAN Not.*, vol. 50, no. 1, pp. 637–650, 2015, ISSN: 0362-1340. DOI: 10.1145/2775051.2676980. [Online]. Available: https://doi.org/10.1145/2775051.2676980.

[36] 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.

[37] R. Krebbers, R. Jung, A. Bizjak, J.-H. Jourdan, D. Dreyer, and L. Birkedal, "The Essence of Higher-Order Concurrent Separation Logic," Mar. 2017, pp. 696–723, ISBN: 978-3-662-54433-4. DOI: 10.1007/978-3-662-54434-1_26.

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

[39] 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.

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

[41] X. Leroy, D. Doligez, A. Frisch, J. Garrigue, D. Rémy, and J. Vouillon, *The OCaml system: Documentation and user's manual*, May 2024. [Online]. Available: https://ocaml.org/manual/5.2/ocaml-5.2-refman.pdf.

[42] N. Leveson, *Medical Devices: The Therac-25*, 1995. [Online]. Available: https://web.stanford.edu/class/cs240/old/sp2014/readings/therac-25.pdf.

[43] A. Lööw, S. H. Park, D. Nantes-Sobrinho, S.-É. Ayoun, O. Sjöstedt, P. Maksimović, and P. Gardner, "Parametric Compositional Symbolic Execution (working title)."

[44] A. Lööw, D. Sobrinho, S.-É. Ayoun, C. Cronjäger, P. Maksimović, and P. Gardner, "Compositional Symbolic Execution for Correctness and Incorrectness Reasoning," Jul. 2024.

[45] 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.

[46] 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.

[47] W. Mansky, *Bringing iris into the verified software toolchain*, 2022. arXiv: 2207.06574 [cs.PL]. [Online]. Available: https://arxiv.org/abs/2207.06574.

[48] W. Mansky and K. Du, "An Iris Instance for Verifying CompCert C Programs," *Proc. ACM Program. Lang.*, vol. 8, no. POPL, 2024. DOI: 10.1145/3632848. [Online]. Available: https://doi.org/10.1145/3632848.

[49] 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.

[50] H. H. Nguyen, V. Kuncak, and W.-N. Chin, "Runtime Checking for Separation Logic," in *Verification, Model Checking, and Abstract Interpretation*, F. Logozzo, D. A. Peled, and L. D. Zuck, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 203–217, ISBN: 978-3-540-78163-9.

[51] 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.

[52] 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.

[53] 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.

[54] 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/.

[55] A. M. Pitts, "Nominal logic, a first order theory of names and binding," *Information and Computation*, vol. 186, no. 2, pp. 165–193, 2003, Theoretical Aspects of Computer Software (TACS 2001), ISSN: 0890-5401. DOI: https://doi.org/10.1016/S0890-5401(03)00138-X. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S089054010300138X.

[56] 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.

[57] 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.

[58] 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.

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

[60] C. Urban and M. Norrish, "A formal treatment of the barendregt variable convention in rule inductions," in *Proceedings of the 3rd ACM SIGPLAN Workshop on Mechanized Reasoning about Languages with Variable Binding*, ser. MERLIN '05, Tallinn, Estonia: Association for Computing Machinery, 2005, pp. 25–32, ISBN: 1595930728. DOI: 10.1145/1088454.1088458. [Online]. Available: https://doi.org/10.1145/1088454.1088458.

[61] 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.

[62] H. Yang and P. O'Hearn, "A Semantic Basis for Local Reasoning," Mar. 2002, pp. 402–416, ISBN: 978-3-540-43366-8. DOI: 10.1007/3-540-45931-6_28.