

SACHA-ÉLIE AYOUN



Foundations, Implementation, and Applications of
Compositional Symbolic Execution

**Imperial College
London**

Submitted in part fulfilment of the requirements for the degree of
Doctor of Philosophy in Computing of Imperial College London and
the Diploma of Imperial College London, April 11, 2025

Statement of Originality

I, Sacha-Élie Ayoun, confirm that the work presented in this thesis is my own. Where information has been derived from other sources, I confirm it is clearly indicated.

Copyright Declaration

The copyright of this thesis rests with the author. Unless otherwise indicated, its contents are licensed under a Creative Commons Attribution-Non Commercial 4.0 International Licence (CC BY-NC). Under this licence, you may copy and redistribute the material in any medium or format. You may also create and distribute modified versions of the work. This is on the condition that: you credit the author and do not use it, or any derivative works, for a commercial purpose. When reusing or sharing this work, ensure you make the licence terms clear to others by naming the licence and linking to the licence text. Where a work has been adapted, you should indicate that the work has been changed and describe those changes. Please seek permission from the copyright holder for uses of this work that are not included in this licence or permitted under UK Copyright Law.

Abstract

This manuscript presents the mathematical foundations, implementation, instantiation, and evaluation of Gillian, a parametric compositional symbolic execution framework for reasoning about program correctness and incorrectness. Gillian supports three kinds of analyses: whole-program symbolic testing in the style of CBMC; semi-automatic compositional verification in the style of VeriFast or Viper; and automatic specification inference based on bi-abduction in the style of Infer:Pulse.

To instantiate Gillian to a specific target language (TL), a tool developer must implement a *symbolic state model*, providing: for whole-program symbolic testing, a set of *actions* that capture the fundamental behaviours of the TL’s memory; for compositional verification, a set of *core predicates* (building blocks of a separation logic assertion language); and for specification inference, a set of *fixes* describing how errors resulting from missing resource can be fixed. We provide an interface for symbolic state models and an exhaustive list of properties they must satisfy to guarantee the soundness of the analyses.

Gillian currently has four instantiations: real-world languages, namely JavaScript, C, and Rust; and a simple demonstrator language called Wisl. In this manuscript, we focus on the C and Rust instantiations, showing how they each leverage opportunities offered by Gillian’s unique and flexible parametricity. Specifically, Gillian-C encodes objects of the C heap into a novel tree representation that enables efficient compositional byte-level reasoning, and Gillian-Rust automates exotic separation logic reasoning specified by RustBelt and RustHornBelt and usually performed in Iris. The Gillian-C instantiation was used for several real-world case studies, successfully finding bugs in industrial-grade code, including the AWS codebase. Gillian-Rust was designed to enable *hybrid* Rust verification, where safe Rust is verified using the Creusot and the unsafe code is verified using Gillian. Preliminary evaluation shows that this prototype can verify small, non-trivial fragments of the Rust standard library with minor source code modification.

Acknowledgements

The past six years have brought both rewarding and challenging moments, academically and personally. Here, I have the opportunity to thank all the people who have made this work possible, the great times greater, and the difficult times bearable.

First, I thank my supervisor, Philippa, for her guidance throughout my PhD. This journey began when she accepted an enthusiastic student for a small project seven years ago, and she has worked hard ever since so that I could continue this work without worry. I would also like to thank my assistant supervisor and dear friend, Petar, for his unwavering support, both technically and personally. He was there when the proof didn't succeed, when the log files were too long, and, most importantly, when I needed to hear that *this too shall pass*.

I am grateful to Bart Jacobs and John Wickerson for their time and effort in examining this thesis. I am impressed by the quality of the feedback I received from them, and from the level of detail they put into their reviews. They have made this manuscript much better.

Throughout my years at Imperial, I have met and worked with fantastic people. I thank the members of the Verified Software group over the years for the technical discussions, support, and friendships. I am grateful to my co-authors, José, Andreas, Daniele, and Xavier. I am also immensely grateful to Teresa and Amani for their continuous administrative help. In addition, I am lucky to have co-supervised and worked with talented students, with special mentions to Nat and Opale. I conclude this paragraph with a shoutout to Caroline—for her fantastic friendship, and her help in proofreading and improving this manuscript.

When I arrived in London, I had no idea I would be so lucky to meet the friends I made here. There are too many names to list exhaustively, but I would like to mention my ex-flatmates Viet, Pierre, Dominika; and the East London neighbours: Sofiane, Faustine, and Manon. Having you here made London my home. I am also immensely grateful to Manon, for sharing her life with me and supporting me through the end of my PhD.

I am blessed to have been supported by loving friends whom I don't see often enough, as they are in France. I will forever be grateful to David for his friendship and for letting me stay with him when I needed it. I thank Stefan for staying up too late at night fighting monsters so I wouldn't be alone when I couldn't. I thank Amandine, whose friendship has been unwavering for the last 18 years. I thank those who always picked up the phone when I needed to talk, particularly Camille, Maïa and Caroline. I also thank my close friends who have illuminated my life for the last ten years: NLB, Val, Aymeric, Charles, 2BO, Kevin, Thuy, Max, Tom, Victor and Caca. And to all my other friends—if your name is not here, I am still very grateful.

Next, I would like to thank those who taught me. I was fortunate to have had a few fantastic teachers throughout my education, and I would not be here without them.

Finally, last but most importantly, I would like to thank my family, who, in more than one way, fall into the category of those who taught me. Tata and Tonton gave me the creativity and humour to survive a PhD. Mamie for being the best grandmother I could hope for. Zack was my guide in early life, and I have him to thank for my love of computer science. Papa sowed the seed of mathematics and made sure it could grow. And Maman—she gave me everything, and was there every moment of my education.

It is an understatement to say that I wouldn't have done it without you.

Contents

1	Introduction	1
1.1	Overview and contributions	1
1.2	Publications	3
1.3	Collaborations	4
<i>1</i>	<i>The Gillian framework</i>	<i>7</i>
2	A unified framework	9
2.1	Symbolic execution	9
2.2	Compositionality	10
2.3	Parametricity	10
2.4	Incorrectness reasoning	11
2.5	A unified, modular, streamlined formalism	11
3	Overview	13
3.1	Using Gillian	13
3.2	Compositional symbolic execution à la carte	15
3.3	Soundness	16
3.4	Correctness and Incorrectness	18
4	Compositionality and parametricity	21
4.1	Languages and concrete semantics	21
4.2	Partial commutative monoids	23
4.3	Concrete compositional semantics	23
4.4	SIGIL: a parametric intermediate language	25
4.5	Examples of state models	30
5	Parametric Assertion Language and Specification Execution	35
5.1	Parametric assertion language	36
5.2	Producers	37
5.3	Consumers and Matching	38
5.4	Specification semantics	43
5.5	Examples	47
6	Symbolic execution	51
6.1	The symbolic realm	52
6.2	Path conditions	54
6.3	Symbolic abstractions	55
6.4	Approximate solvers	56
6.5	Symbolic execution processes	58
6.6	Interlude: implementation and optimisation	63
6.7	Parametric symbolic execution for SIGIL	66
6.8	Example symbolic state models	68

7	Analyses	75
7.1	OX and UX whole-program symbolic testing	75
7.2	Compositional verification	76
7.3	Automatic UX specification synthesis	78
8	Constructing state models	87
8.1	Product of state models	88
8.2	Exclusive ownership	91
8.3	Agreement state model	92
8.4	Fractional state model	93
8.5	Partial finite maps	94
8.6	Freeable state model	95
8.7	The predicate symbolic transformer	97
8.8	The mutable store, and where it goes wrong	101
8.9	Implementation and code reuse	101
9	Applications: Wisl and JavaScript	103
9.1	Wisl: <u>W</u> hile language for <u>s</u> eparation <u>l</u> ogic	103
9.2	Gillian-JS	105
10	Related work	109
10.1	Compositional symbolic execution tools for verification	109
10.2	Bi-abduction tools	111
10.3	Parametric frameworks for analysis	112
10.4	Combining UX and OX analysis	113
10.5	Monadic symbolic execution	114
II	<i>Gillian-C</i>	115
11	Gillian-C: What and why?	117
11.1	The Gillian-C Infrastructure	118
11.2	Whole-program symbolic testing	119
11.3	Compositional verification	120
11.4	Specification synthesis using bi-abduction	122
12	The Gillian-C symbolic state model	123
12.1	The CompCert memory model	123
12.2	Symbolic block trees: overview	124
12.3	Symbolic block trees: implementation	128
12.4	Symbolic block trees: assertion language	134
12.5	Symbolic block trees: fixes for bi-abduction	138
13	The Gillian-C front-end	139
13.1	Compiling C code	139
13.2	Compiling C assertions	142
14	Evaluation	143
14.1	Collections-C	143
14.2	The AWS Encryption Header case study	148
14.3	Limitations	152

15 Related work	153
15.1 Whole-program symbolic testing	153
15.2 Compositional verification using SL	153
15.3 Infer:Pulse	154
 III Gillian-Rust	 157
16 A challenge	159
17 The Gillian-Rust infrastructure	163
17.1 A hybrid approach: Creusot + Gillian-Rust	163
17.2 Example usage of Gillian-Rust	164
18 Reasoning about the real Rust heap	167
18.1 Layout-independent memory addresses	167
18.2 Objects in the Rust symbolic heap	168
18.3 Specifying the Rust heap: the typed points-to core predicate	170
19 Automating reasoning about mutable borrows	173
19.1 Modelling lifetimes: core predicates	173
19.2 Modelling full borrows: guarded predicates	174
19.3 Proving safety of borrow extraction	176
20 Functional correctness and prophetic reasoning	179
20.1 Representations, parametric prophecies, and observations	179
20.2 Key idea: parametric prophecies and symbolic execution	180
20.3 Value observers and prophecy controllers	181
21 Anatomy of a hybrid proof : Merge Sort	183
21.1 Writing a hybrid proof	183
21.2 Compilation of Creusot specifications	185
21.3 Gillian-Rust in action: <code>LinkedList::push_front</code>	185
22 Evaluation	189
22.1 EvenInt	189
22.2 LinkedList	190
22.3 MiniVec and Vec	191
22.4 Hybrid Verification	192
23 Limitations and Future work	195
23.1 Unimplemented features	195
23.2 Meta-theory simplifications	195
23.3 Unexplored topics	196
24 Related work	199
	203
25 Future work	205
 Appendix	 219
A Compositionality and parametricity	221

B	Parametric Assertion Language	223
B.1	Correctness of the parametric producer	223
B.2	Correctness of the parametric consumer	223
B.3	Soundness of specification execution	227
B.4	Soundness of the specification semantics	229
C	Symbolic Execution	231
C.1	Monad laws for the symbolic execution monad	231
C.2	Composition of symbolic processes	232
D	Analyses	237
D.1	Compositional verification	237
D.2	Specification inference procedure	239
E	Constructing State Models	249
E.1	Product of state models	249
E.2	State model of values	252
E.3	Exclusive ownership	253
E.4	Agreement state model	256
E.5	Fractional state model	259
E.6	Partial maps	264
E.7	Freeable state model	270
E.8	Predicate state model transformer	274
F	Some Gillian-Rust tactics	281
F.1	Freezing existential variables	281
F.2	Borrow extraction with prophecy variables	282

Chapter 1

Introduction

1.1 Overview and contributions

This manuscript presents the mathematical foundations, implementation, instantiation, and evaluation of Gillian, a parametric compositional symbolic execution (CSE) framework for unified reasoning about program correctness and incorrectness. Currently, Gillian supports three kinds of analyses:

1. **Whole-program symbolic testing (WPST)**, where users write symbolic tests, which can be seen as an augmentation of standard concrete tests routinely written by developers with symbolic variables. If a symbolic test passes, WPST provides bounded correctness guarantees, and if it fails, it provides a concrete counter-example expected to trigger the corresponding bug. These guarantees are similar to those provided by a bounded model checker such as CBMC¹.
2. **Semi-automatic compositional verification**, where users annotate code with separation-logic² (SL) specifications and proof tactics. In case of success, this analysis provides unbounded functional correctness guarantees, and in case of failure, it provides a failing symbolic trace. This analysis is similar to the one performed by VeriFast³ or Viper⁴.
3. **Automatic specification inference**, which does not require user-provided annotations of any kind and generates incorrectness separation logic⁵ (ISL) specifications for each of analysed functions, in the style of Infer:Pulse⁶.

Gillian currently has four instantiations: three for real-world languages, JavaScript, C, and Rust, and one for a simple demonstrator programming language called Wisl. We focus here on the C and Rust instantiations in detail, showing how they each leverage opportunities offered by the unique and flexible parametricity of Gillian. This manuscript is split into three parts: [Part I](#) presents the mathematical foundations of Gillian; [Part II](#) focuses on Gillian-C; and [Part III](#) focuses on Gillian-Rust. In the following, we provide an overview of the contributions of each of these parts.

1.1.1 Gillian

The main contribution of [Part I](#) is the formalisation, implementation, and proof of soundness of Gillian; this contribution can be broken down into smaller pieces. First, we introduce a novel formalism for CSE, which, for the first time, uses the “real” semantics of a language as the trusted computing base (TCB) instead of a semantics already specialised for separation-logic reasoning. In addition, this formalism

¹ Clarke et al., “A Tool for Checking ANSI-C Programs”, 2004 [CKL04]

² Reynolds, “Separation Logic: A Logic for Shared Mutable Data Structures”, 2002 [Rey02]

³ Jacobs et al., “VeriFast: A Powerful, Sound, Predictable, Fast Verifier for C and Java”, 2011 [Jac+11]

⁴ Müller et al., “Viper: A Verification Infrastructure for Permission-Based Reasoning”, 2016 [MSS16b]

⁵ Raad et al., “Local Reasoning About the Presence of Bugs: Incorrectness Separation Logic”, 2020 [Raa+20]

⁶ Le et al., “Finding real bugs in big programs with incorrectness logic”, 2022 [Le+22]

is *parametric* on a *symbolic state model*, which allows us to streamline the formalisation of the engine’s core to capture only the essence of CSE. Furthermore, the formalism is designed to be modular so that the proof effort can be broken down into smaller, independent, and more manageable pieces.

To instantiate Gillian to a particular target language (TL), a tool developer has to implement a corresponding *symbolic state model* which then gets “plugged into” the core engine. We formalise the core engine and give a clear interface between the core and symbolic state models. In particular, these state models need to provide:

- **for whole-program symbolic testing**, a set of *actions* that capture the fundamental behaviours of the TL;
- **for compositional verification**, additionally, a set of *core predicates*, which are the building blocks of a separation-logic assertion language appropriate for the TL; and
- **for specification inference**, additionally, a set of *fixes*, which describe how errors resulting from missing resources can be fixed.

For each component we provide an exhaustive list of properties that it must satisfy so that the soundness of the underlying Gillian analyses is guaranteed: for correctness reasoning the absence of false positives and for incorrectness reasoning the absence of false negatives.

As the core engine is minimal, a large share of the design, implementation, and proof burden is delegated to the tool developer creating the symbolic state model. To alleviate this burden, we provide a set of reusable components that can be used to build complex state models, leaving the developer to focus on the specificities of the target language. This approach, inspired by Iris⁷ but novel in the context of CSE, enables a sort of *compositional execution à la carte*. This approach greatly simplifies the implementation and soundness proof of the symbolic state models. For instance, the state model of Gillian-JS (§9.2) can be entirely built from such components and is now only a few lines of code. Furthermore, by removing duplication of code within the state model itself, we reduced the likelihood of errors in the implementation and improved performances by factoring out code and unifying the optimisations accross all actions.

We evaluate the flexibility, expressivity, and real-world applicability of Gillian in the rest of the manuscript.

1.1.2 Gillian-C

Gillian-C is the first compositional symbolic execution tool that supports all three analyses outlined above. Its first key contribution lies at the core of its symbolic state model: a novel tree-based representation of objects in the symbolic heap that automates efficient byte-level reasoning. The use of this representation is facilitated by the unique flexibility of Gillian, which, unlike VeriFast and Viper, does not require the symbolic heap to be represented as a list of chunks.

Its second key contribution is a comprehensive evaluation of Gillian-C on real-world codebases, verifying code that manipulates complex data structures and finding bugs in industrial-grade code. Specifically, we have written and run symbolic tests for Collections-C⁸, a real-world

⁷ Jung et al., “Iris from the ground up: A modular foundation for higher-order concurrent separation logic”, 2018 [Jun+18]

⁸ Panić, *srdja/Collections-C*, 2024 [Pan24]

data structure library for C, and compared the results to those obtained by the state-of-the-art CBMC tool. Moreover, we have verified the correctness of the message header deserialisation module of the AWS Encryption SDK for C⁹, showing Gillian-C’s usability in relatively-large-for-verification-tasks case studies. Finally, we were able to generate ISL specifications for all the functions of Collections-C, demonstrating the scalability of automatic specification inference.

1.1.3 Gillian-Rust

The core contribution of [Part III](#) is the introduction of a *hybrid* approach for the verification of Rust code, based on the use of two tools, one for verifying the safe fragment of the code and one for verifying the unsafe fragment. To demonstrate the feasibility of this approach, we have used the state-of-the-art Creusot¹⁰ safe Rust verifier and developed Gillian-Rust to verify type safety and functional correctness of unsafe code in a way that is compatible with Creusot. To soundly connect Creusot and Gillian-Rust, we have relied on RustBelt¹¹ and RustHornBelt¹² as the underlying meta-theory. This approach required of us to encode complex separation-logic reasoning extracted from the Iris development of RustBelt and RustHornBelt into Gillian. To achieve this, we were again, just as in the case of Gillian-C, able to leverage Gillian’s flexibility, developing novel automation compatible with CSE that can reason about mutable references in unsafe blocks. Some of these automations necessitate the improvements to the compositional symbolic execution framework that we introduce in [Part I](#)¹³. Finally, we have evaluated Gillian-Rust on several small but non-trivial functions of the Rust standard library.

1.2 Publications

The following articles have been published or submitted for publication as part of this PhD:

- *Gillian, part I: A multi-language platform for symbolic execution.*
J. Frago Santos, P. Maksimović, S-É. Ayoun, P. Gardner.
PLDI’20 [\[Fra+20\]](#)
- *Gillian, part II: Real-world verification for JavaScript and C.*
P. Maksimović, S-É. Ayoun, J. Frago Santos, P. Gardner.
CAV’21 [\[Mak+21\]](#).
- *Compositional Symbolic Execution for Correctness and Incorrectness.*
A. Löw, D. Nantes Sobrinho, S-É. Ayoun, C. Cronjäger,
P. Maksimović, P. Gardner.
ECOOP 24, distinguished paper [\[Lö+24a\]](#).
- *Matching plans for Frame Inference in Compositional Reasoning.*
A. Löw, D. Nantes Sobrinho, S-É. Ayoun, P. Maksimović,
P. Gardner.
ECOOP 24 [\[Lö+24b\]](#).
- *A hybrid approach to semi-automated Rust verification.*
S-É.-Ayoun, X. Denis, P. Maksimović, P. Gardner.
PLDI 25 [\[Ayo+25\]](#).

⁹ Amazon Web Services, *aws/aws-encryption-sdk-c*, 2024 [\[Ama24b\]](#)

¹⁰ Denis et al., “Creusot: a Foundry for the Deductive Verification of Rust Programs”, 2022 [\[DJM22\]](#)

¹¹ Jung et al., “RustBelt: securing the foundations of the Rust programming language”, 2017 [\[Jun+17\]](#)

¹² Matsushita et al., “RustHornBelt: a semantic foundation for functional verification of Rust programs with unsafe code”, 2022 [\[Mat+22\]](#)

¹³ Specifically, the guarded predicate transformer defined in [§19.2](#) does not fit into previous formalisations of Gillian where predicates are hard-coded into the engine. In contrast, our modularised engine (where predicates are simply a “transformer” optionally applied by the user) allows for guarded predicates to be “just another transformer”.

1.3 Collaborations

Scientific work is rarely the product of a single mind. I¹⁴ gladly acknowledge that the work presented in this manuscript is not an exception, and outline attribution of this work in this section, in order of appearance in the manuscript¹⁵:

- The original ideas behind Gillian have come from José Fragoso Santos, and the original implementation of Gillian was led by José and supported by Petar Maksimović. This work was done in 2019 and was based on their previous work on JaVerT 2.0. I have been leading the implementation of the tool since 2021, enhancing its performance, expressivity and modularity. I have also closely supervised work that enabled various features of Gillian, such as incremental reasoning and the symbolic execution debugger¹⁶.

	WPST	Comp. Verification	Spec Inference	Parametricity	OX + UX	Soundness Proofs	TCB: full semantics	Symbolic execution monad	Reusable state model components
Fragoso Santos et al. Gillian Part I	✓	×	×	✓	×	✓	×	×	×
Maksimović et al. Gillian Part II	×	✓	×	✓ ⁱ	×	×	×	×	×
Löw et al. CSE	✓	✓	✓	×	✓	✓ ⁱⁱ	×	×	×
Ayoun This manuscript	✓	✓	✓	✓	✓	✓	✓	✓	✓ ⁱⁱⁱ

- The formalisation and soundness proof of parametric non-compositional symbolic execution were published in Gillian Part I, and were led by José and supported by Petar. Petar led the partial formalisation of parametric compositional verification, presented in the Gillian Part II paper, but this presentation did not include a soundness proof. The formalisation and proof of soundness of a non-parametric CSE tool supporting both over- (OX) and under-approximation (UX) were led by Andreas Löw and Daniele Nantes Sobrinho. I implemented support for under-approximate reasoning in the tool and conducted the corresponding evaluation.

This manuscript introduces the following novel features to this line of research, which are the product of my work:

- the soundness of the framework is justified with respect to a *real* (full) semantics, not one that manipulates partial states tailored for separation-logic reasoning;
- I formalise symbolic execution using a *symbolic execution monad*, simplifying the proof and implementation work; and
- I provide a library of reusable components that can be used to build complex state models, enabling what I call *CSE à la*

¹⁴ As this is about my personal contributions, I exceptionally use the first-person pronoun in this section.

¹⁵ Note that this list also includes novel concepts that will be introduced in this manuscript.

¹⁶ Karmios et al., “Symbolic Debugging with Gillian”, 2023 [KAG23]

Table 1.1: Comparison of the contributions of this manuscript and previous partial formalisations of Gillian. I participated in all three pieces of previous work but did not lead the formalisations.

ⁱ In Gillian Part II, the core assertion language has built-in support for pure formulae and user-defined predicates. This manuscript defines these as core predicates, making the approach more modular.

ⁱⁱ In the work led by Löw and Nantes, the proof for compositional verification does not support recursive functions.

ⁱⁱⁱ Opale Sjöstedt implemented these components in Gillian and helped improve their formalisation.

carte. Table 1.1 provides a summary comparison between the formalisation presented in this manuscript and the previously published work on Gillian.

- The design, implementation and evaluation of Gillian-C, are entirely my own contributions.
- During the verification of the AWS Encryption SDK header deserialisation module, Petar Maksimović designed the pure specification of the header, specified and verified the JavaScript implementation, and substantially improved the Gillian first-order solver to enable verification. I specified and verified the C implementation.
- The design, formalisation, and implementation of the Gillian-Rust state model are solely my contributions. The concept of hybrid verification was developed in equal collaboration with Xavier Denis. Additionally, Xavier contributed significantly to the engineering of the Gillian-Rust front-end and was an invaluable source of knowledge about Creusot and RustHornBelt, both of which are fundamental to the introduced approach.

Part I

The Gillian framework

Chapter 2

A unified framework

Well but then you are not solving any problem.

Patrick Cousot about Gillian, to me,
at POPL 2024

Gillian is a *parametric compositional symbolic execution framework for reasoning about program correctness and incorrectness*. This fairly concise description contains five terms that require unpacking, and this chapter traces the origins and developments leading to Gillian, addressing each of these terms and clarifying the gap it fills.

2.1 Symbolic execution

Gillian is a *symbolic execution* framework. Symbolic execution is a technique pioneered in the 1970s¹ that interprets programs using *symbolic values*—expressions that contain variables—rather than concrete values. Since a symbolic value represents a set of concrete values, this approach allows for simultaneous exploration of multiple potential program executions.

In symbolic execution, encountering a conditional statement may lead to the exploration of multiple paths, each of which is characterised by a *path condition*, a first-order formula satisfied by the symbolic variables along this path. A path is said to be *feasible* if its path condition is *satisfiable*, a property determined by an SMT solver². If a feasible path leads to an error, symbolic execution normally reports a bug back to the user.

Symbolic execution flourished in the 2010s, when a great advancement was achieved in the performance of SMT solvers, leading to several tools being extensively used in industry³. However, despite its advantages, symbolic execution is still faced with significant limitations:

- it may lead to *path explosion*, when the number of branching paths exhausts computational resources;
- it is inherently *bounded*; loops and recursive calls must be unrolled a finite number of times, preventing potentially infinite computations;
- it lacks *compositionality* (or *modularity*), in that it does not allow fragments of a program to be analysed in isolation.

The literature is rich with numerous techniques and approaches that aim to address these limitations. In this manuscript, we focus on what it means to do compositional symbolic execution.

¹ King, “Symbolic execution and program testing”, 1976 [Kin76]; and Boyer et al., “SELECT—a formal system for testing and debugging programs by symbolic execution”, 1975 [BEL75]

² De Moura et al., “Z3: an efficient SMT solver”, 2008 [DB08]; and Barbosa et al., “cvc5: A Versatile and Industrial-Strength SMT Solver”, 2022 [Bar+22]

³ Cadar et al., “KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs”, 2008 [CDE08]; and Clarke et al., “A Tool for Checking ANSI-C Programs”, 2004 [CKL04]

2.2 Compositionality

We view compositional symbolic execution (CSE) as symbolic execution that allows the analysis to be performed in segmented, function-by-function fashion. This approach is evidently modular while at the same time reducing the risk of path explosion as traces are limited to the scope of a single function. It also unlocks the ability to obtain unbounded correctness guarantees using various abstraction techniques. In this manuscript, we focus on CSE that uses function specifications written in separation logic.

Separation logic⁴ (SL) is a program logic that allows for *local reasoning* about programs correctness. Its separating conjunction⁵ $P * Q$ asserts that P and Q hold in two *disjoint* regions of the heap. The frame rule⁶, foundational to SL, asserts that adding separate resources does not interfere with execution outcomes, enabling compositional analysis where functions are decoupled from their execution context:

$$\frac{\text{FRAME} \quad \{ P \} e \{ Q \}}{\{ P * R \} e \{ Q * R \}}$$

Compositional symbolic execution traces its roots back to Smallfoot⁷ which pioneered the technique. The legacy of Smallfoot extends to contemporary verification tools like VeriFast⁸, Viper⁹ and JaVerT 2.0¹⁰, and automatic compositional bounded verifiers such as Infer¹¹.

Since its creation, VeriFast has been used to relentlessly explore new verification techniques¹² based on separation logic, while Viper has been used as a basis for enabling and enhancing automation¹³ for the verification of several large-scale projects written in a variety of programming languages¹⁴. Similarly, JaVerT 2.0 was an experiment in applying compositional symbolic execution techniques to a highly-dynamic language that is JavaScript. Infer, on the other hand, has become the industry standard for automatic bug-finding at scale¹⁵.

Gillian is a new entrant in this lineage, unifying non-compositional symbolic execution, compositional verification, and compositional bug finding in a single framework. To further this unification, Gillian introduces a novel parametric approach to the handling of the state and the SL resources that it contains.

2.3 Parametricity

The above-mentioned tools, despite their strengths, are each confined to a single approach to modelling the state. For example, VeriFast represents its resources as lists of “memory chunks”, a highly flexible approach, but one that requires most reasoning to be extrinsically axiomatised and manually performed. Viper employs a similar list of chunks, albeit closely tied to an object-like memory model in which all objects possess a common set of properties. This model has facilitated Viper’s success in verifying programs written using various programming languages; however, it comes with limitations. For instance, Nagini, Viper’s Python front-end, only supports statically typed Python, raising

⁴ Reynolds, “Separation Logic: A Logic for Shared Mutable Data Structures”, 2002 [Rey02]

⁵ This connective is, in fact, characteristic of *bunched logics* [OP99], of which separation logic is an instance.

⁶ The frame rule presented in the original paper had an additional side condition due to the use of a mutable store. We omit it here, and discuss the mutable store in §8.8.

⁷ Berdine et al., “Smallfoot: Modular Automatic Assertion Checking with Separation Logic”, 2006 [BCO06]

⁸ Jacobs et al., “VeriFast: A Powerful, Sound, Predictable, Fast Verifier for C and Java”, 2011 [Jac+11]

⁹ Müller et al., “Viper: A Verification Infrastructure for Permission-Based Reasoning”, 2016 [MSS16b]

¹⁰ Fragoso Santos et al., “JaVerT 2.0: compositional symbolic execution for JavaScript”, 2019 [Fra+19]

¹¹ Calcagno et al., “Infer: An Automatic Program Verifier for Memory Safety of C Programs”, 2011 [CD11]

¹² Jacobs et al., “Expressive modular fine-grained concurrency specification”, 2011 [JP11]; Penninckx et al., “Sound, Modular and Compositional Verification of the Input/Output Behavior of Programs”, 2015 [PJP15]; Agten et al., “Sound Modular Verification of C Code Executing in an Unverified Context”, 2015 [AJP15]; and Jung et al., “The future is ours: prophecy variables in separation logic”, 2020 [Jun+20]

¹³ Müller et al., “Automatic Verification of Iterated Separating Conjunctions Using Symbolic Execution”, 2016 [MSS16a]; and Dardinier et al., “Sound Automation of Magic Wands”, 2022 [Dar+22]

¹⁴ Eilers et al., “Nagini: A Static Verifier for Python”, 2018 [EM18]; Wolf et al., “Gobra: Modular Specification and Verification of Go Programs”, 2021 [Wol+21]; and Blom et al., “The VerCors Tool Set: Verification of Parallel and Concurrent Software”, 2017 [Blo+17]

¹⁵ Distefano et al., “Scaling static analyses at Facebook”, 2019 [Dis+19]

questions about its ability to handle more dynamic language features. Javert 2.0, on the other hand, can handle JavaScript, a highly dynamic language, but its memory model is tailored to the specifics of JavaScript, and could not be used, for example, for verifying C programs.

Gillian introduces a novel parametric way of handling resources. Tool developers using Gillian must provide a *state model*, defining: **a)** the structure of their symbolic states; **b)** a set of basic operations to manipulate these states, called *actions*; and **c)** the building blocks of a separation logic assertion language, called *core predicates*. This parametricity allows for the modelling of a wider variety of programming languages, including static languages such as C and dynamic languages such as JavaScript. In addition, state models tailored for a specific language can make use of domain-specific knowledge to enhance the precision and automation of the analysis¹⁶. Furthermore, this parametricity allows for the exploration of new automations for more exotic separation logic resources, commonly used in projects like Iris, but not widely explored in the context of CSE¹⁷.

Gillian offers this parametricity in the context of another novelty, its support for both correctness and incorrectness reasoning.

2.4 Incorrectness reasoning

Traditional compositional symbolic execution tools primarily focus on over-approximating (OX) verification strategies, which guarantee the *absence of bugs* in case of success. However, the advent of Incorrectness Logic¹⁸ (IL) and Incorrectness Separation Logic¹⁹ (ISL) underscore the importance of under-approximating (UX) analyses for effective bug detection. In fact, Infer’s automatic compositional verification was found to be more useful when trying to find bugs rather than when trying to prove their absence. In turn, the technique used in Infer, called *bi-abduction*, has been refined to guarantee the detection of real, actionable bugs without false positives²⁰.

Gillian is able to host both OX and UX analyses on top of its core engine by flipping a switch depending on the desired mode. Both formalisation and implementation of Gillian are, to our knowledge, the first to exhaustively capture under-approximate CSE (including sound handling of function calls) and also pinpoint precisely how to switch between OX and UX analyses.

2.5 A unified, modular, streamlined formalism

Gillian, in summary, is a compositional symbolic execution framework that is parametric over a state model, supports diverse analyses, and is capable of both OX and UX reasoning, making it the ideal setting in which one could explore the *unifying foundations of CSE*. We take advantage of this by formalising the parametric CSE engine at the core of Gillian in a minimal yet general way. This allows us to extract the essence of CSE and prove soundness of the core without clutter.

The downside of the minimality of the core engine is that a substantial part of the design and soundness proof effort is effectively

¹⁶ The idea of language-tailored automation is explored in [Part II](#), in the context of C analysis.

¹⁷ Exotic Iris-like resources are explored in [Part III](#), in the context of Rust analysis.

¹⁸ O’Hearn, “Incorrectness logic”, 2019 [[OHe19](#)]

¹⁹ Raad et al., “Local Reasoning About the Presence of Bugs: Incorrectness Separation Logic”, 2020 [[Raa+20](#)]

²⁰ Le et al., “Finding real bugs in big programs with incorrectness logic”, 2022 [[Le+22](#)]

delegated to the tool developer in charge of designing the state model. To mitigate this, we create a library of composable building blocks that tool developers can use in the construction of their state models. Each of these components comes with soundness preservation properties which guarantee that the state models that contain them are correct by construction. Importantly, should these components not be sufficient, the tool developer can freely implement their own and combine them with the ones from our library. To assist developers in this task, we provide a succinct interface that should be implemented, together with powerful abstractions that they can use, such as the symbolic execution monad. In addition, should they wish to prove the soundness of these components, Gillian offers a clear blueprint containing a set of simple, concise properties, in contrast to the complex ones currently on offer.

Finally, while our primary goal is to obtain a deep understanding of the essence of CSE tools, we do our best to not over-idealise our formalisation, ensuring that it remains practical and applicable in real-world scenarios. For example, we take into account the imperfection of modern real-world SMT solvers and design an interface that ensures their sound use. We also extensively discuss the various trade-offs and techniques concerning optimisation and enhancing the scalability of analyses, and the framework provides ample flexibility for an exploration of these trade-offs.

Chapter 3

Overview

In this chapter, we provide an overview of our novel formalisation of Gillian. In §3.1, we start by presenting the analyses that Gillian supports and discuss what must be implemented by the tool developer in order for these analyses to be unlocked. In §3.2, we then explain how the core CSE engine can be extended with additional features by applying transformers to the input state model, enabling a sort of *CSE à la carte*. In addition, §3.3 provides a bird’s-eye view of the justification of the framework’s soundness, which is a core contribution of this part of the manuscript. Finally, in §3.4, as our framework supports both over-approximation (OX) and under-approximation (UX), we precisely describe which of its components are approximating and discuss how they can be toggled between OX and UX.

3.1 Using Gillian

Gillian’s implementation performs its analyses on GIL¹, a *parametric* intermediate language. In this manuscript (§4.4), we formalise a **Simplified and Improved GIL (SIGIL)**, which has fewer constructs than GIL and allows for greater flexibility.

To instantiate the framework and use its analyses, a tool developer must provide a compiler from the target language to GIL, together with the OCaml *implementation* of a *symbolic state model*. The latter provides: **a)** the datatype of symbolic states; **b)** *actions* that can be performed on these states; and **c)** *core predicates* that form the building blocks of a separation-logic assertion language.

Running example: linear heap state model and linked lists Consider the linear heap model often used when formalising separation logics². The linear heap is a mapping from natural numbers to values, manipulated using four actions: **load**, **store**, **alloc** and **free**. The main assertion that is used to describe linear heaps is points-to core predicate, $x \mapsto y$, which establishes *ownership* of a heap cell at address x , which contains the value y .

In the linear heap, a linked list is commonly implemented as a collection of nodes, where each node consists of two contiguous heap cells, with the first cell containing the node value and the second cell containing the address of the next node. The end of the list is signalled by the address to the next pair equalling **null**. Figure 3.1 shows an example of a linked list containing three nodes at addresses 0, 4, and 7 containing values 0, 1, and 2. The remaining cells are not relevant and their content is elided for clarity.

¹ GIL stands for Gillian Intermediate Language

² The linear heap is used, for example, in the Separation Logic [Rey02], Incorrectness Separation Logic [Raa+20] and Exact Separation Logic [Mak+23] papers.

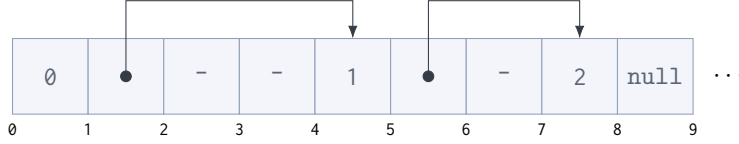


Figure 3.1: A diagram representing the first ten cells of a linear heap containing a singly-linked list of three nodes, respectively at addresses 0, 4, and 7 containing values 0, 1, 2.

Figure 3.2 shows an example of a simple recursive function, written in SIGIL instantiated with the linear heap state model, which computes the length of a singly-linked list. It first checks if we have reached the end of the list (that is, if the address x of the current node equals `null`), and if it does, it evaluates to 0 as the list is empty. Otherwise, it loads the address of the next node using the `load` action (actions are denoted using angular brackets), and assigns this address to the variable z . The function then recursively calls itself using z , adds 1 to the result, and evaluates to the obtained value. We now proceed to show what it means to apply each of the three analyses that Gillian provides to this function.

```
llen(x) {
  if x = null then 0
  else
    let z = ⟨load⟩(x + 1) in
    1 + llen(z)
}
```

Figure 3.2: A list-length function in SIGIL instantiated with the linear heap memory model

Whole-program symbolic testing The first analysis that we present is *whole-program symbolic testing* (WPST), which allows for the execution of *symbolic tests*. Symbolic tests are akin to unit tests but may explore an infinite number of potential executions simultaneously. This analysis unrolls loops and recursive function calls a fixed number of times to avoid any infinite loops. In case of success, WPST guarantees *bounded* correctness, and in case of failure it provides a concrete counter-example. The format of the input symbolic tests and the guarantees provided are similar to those offered by bounded model checkers such as CBMC³.

Figure 3.3 shows a simple symbolic test that proves that the list-length function returns 1 for any list of size 1, independently of the actual content of the (only) node. This is achieved by creating the node one cell at a time using the `store` action, and using the result of the `nondet` function as the node content. This function, when run concretely non-deterministically returns an arbitrary value v , but when run symbolically simply returns a fresh unconstrained symbolic variable, thereby capturing all possible concrete behaviours. Finally, the `assert` function checks that the function returns the expected value, raising a test failure otherwise.⁴

³ Clarke et al., “A Tool for Checking ANSI-C Programs”, 2004 [CKL04]

```
let content = nondet() in
let x = ⟨alloc⟩(2) in
let () = ⟨store⟩(x, content) in
let () = ⟨store⟩(x + 1, null) in
assert(llen(x) == 1)
```

Figure 3.3: A symbolic test for the list-length function of Figure 3.2 which proves that the function is correct for all lists of size 1.

⁴ We will later show that both `nondet` and `assert` can be implemented as an action that is compatible with any state model.

Compositional verification Our second analysis is *compositional verification*, where the user writes separation-logic specifications for functions of the program. Each function is semi-automatically verified to satisfy its specification, in isolation from the rest of the program. If successful, the analysis provides unbounded functional correctness guarantees, and in case of failure gives a failing symbolic trace. It is similar to the analyses performed by VeriFast or Viper.

For example, the list-length function from Figure 3.3 can be specified using the following separation logic triple, where the `list(x, α)` predicate, defined in Figure 3.4, denotes a list starting at address x

$$\begin{aligned} \text{list}(x, \alpha) \triangleq & \\ & (x = \text{null} * \alpha = []) \\ & \vee (\exists y, z, \beta. \\ & \quad x \mapsto y * (x + 1) \mapsto z * \\ & \quad \text{list}(x + 1, \beta) * \beta = y : \alpha) \end{aligned}$$

Figure 3.4: Definition of the `list` predicate that describes a list starting at address x and containing the sequence of values α , where $:$ is the standard list cons operator.

and containing the sequence of values α :

$$\{ \text{list}(\mathbf{x}, \alpha) \} \text{llen}(\mathbf{x}) \{ \text{Ok} : \mathbf{r}. \mathbf{r} = |\alpha| * \text{list}(\mathbf{x}, \alpha) \}$$

This specification states that, starting from a state that contains a list at address \mathbf{x} and holding values α , the $\text{llen}(\mathbf{x})$ function either successfully terminates (as specified by the Ok outcome) or diverges. Furthermore, it states that all terminating branches will return the length of the list (using the dedicated variable \mathbf{r}) and not modify the state. Gillian can prove this specification automatically; we note that some more complex proofs may require additional annotations, such as loop invariants, lemma applications, and predicate manipulation.

Automatic specification synthesis The final type of analysis is *automatic specification synthesis*, which can infer bounded ISL⁵ specifications that are guaranteed sound for a given set of functions. Specifications are inferred using an under-approximate and parametric version of a technique called bi-abduction⁶, which performs *resource inference* when the required resource to execute a program is found to be *missing* from the current state. It is then possible to fully-automatically detect *true* bugs in programs by applying a filter on the synthesised specifications⁷, although Gillian does not currently provide such a filter. This analysis is similar to the analysis performed by Meta’s Infer:Pulse tool.

For example, using the linear heap state model, Gillian can infer the following specification of the llen function, which corresponds to the successful case when the list is exactly of size 1:

$$[\mathbf{x} \mapsto \mathbf{y}, \mathbf{z}] \text{llen}(\mathbf{x}) [\text{Ok} : \mathbf{r}. \mathbf{x} \mapsto \mathbf{y}, \mathbf{z} * \mathbf{z} = \text{null} * \mathbf{r} = 1]$$

where $\mathbf{x} \mapsto \mathbf{y}, \mathbf{z}$ is syntactic sugar for $\mathbf{x} \mapsto \mathbf{y} * (\mathbf{x} + 1) \mapsto \mathbf{z}$.

Now that we have discussed the analyses Gillian can perform, we turn our attention to the philosophy at the heart of our novel formalisation.

3.2 Compositional symbolic execution à la carte

SIGIL and its parametric assertion language are designed to be minimal, delegating most of the reasoning to the input state model. The primary use case of state models is, as the name suggests, to encode resources manipulated by the program, such as the heap, but the interface of state models is sufficiently general to enable other applications.

For instance, CSE tools usually offer a way to define predicates, such as `list`, that can then be used in specifications. Such predicates, however, do not interact well with under-approximate reasoning. Instead of proposing several versions of the engine, with or without support for predicates, we are able to define a *state model transformer* $\overline{\text{Pred}}(\overline{\mathbb{S}}, \mathbf{P})$ ⁸, which extends a given state model $\overline{\mathbb{S}}$ with a set of user-defined predicates \mathbf{P} and the associated reasoning.

Perhaps more surprisingly, we can take a similar approach to enable bi-abduction. Given a symbolic state model compatible with UX reasoning, we can define a $\text{Bi}(\overline{\mathbb{S}})$ state model transformer, which performs resource inference without modifying the symbolic execution engine of

⁵ It is also possible to synthesise bounded SL specifications, though it was found to be less useful than ISL specifications. Therefore, we leave the formalisation of SL specification synthesis out of this presentation.

⁶ Calcagno et al., “Compositional shape analysis by means of bi-abduction”, 2009 [Cal+09]

⁷ Le et al., “Finding real bugs in big programs with incorrectness logic”, 2022 [Le+22]

⁸ We denote “symbolicness” with an overline.

All analyses are then proven sound¹⁰ w.r.t. the semantics obtained by inserting a full state model $\underline{\mathbb{S}}$ ¹¹ in the parametric concrete semantics CS . This semantics, together with the interpretation of the core predicates, forms the *trusted computing base* (TCB) of the framework (i.e. the part assumed to be correct), highlighted in **purple** in the above diagram.

The tool developer is in charge of proving the correctness properties of the symbolic state model $\bar{\mathbb{S}}$ with respect to the TCB. These results, coloured in **blue**, are then lifted to the entire semantics of SIGIL using theorems provided by the framework, coloured in **green**. We provide further details about these two layers of proof below.

Correctness of the symbolic state model First, the tool developer must prove the soundness of the symbolic state model with respect to the concrete state model. To facilitate the proof work, we introduce an additional theoretical component: the compositional state model \mathbb{S} , which still operates on concrete values, but manipulates *fragments* of states, which must form a partial commutative monoid, instead of *full* states. Moreover, the compositional state model defines the satisfiability relation for the core predicates, and must implement producers and consumers for these core predicates.

Next, the tool developer must prove four properties to guarantee the soundness of the analysis. First, the semantics of the actions of the compositional state model must be frame-preserving, and sound with respect to the actions of the concrete state model. Second, the producer and consumer of the compositional state model must be proven valid with respect to the satisfiability relation of the core predicates. Finally, the symbolic state model implemented must be proven sound with respect to the compositional state model.

As it is the state models that handle the majority of the reasoning, we take measures to alleviate the proof effort from the tool developer. For instance, [Chapter 6](#) introduces a monadic formalisation of symbolic execution that simplifies soundness proofs. In addition, the library of state model constructions offered in [Chapter 8](#) has already been proven sound, and state models such as the linear heap can be constructed as described previously without requiring any further proof effort.

Parametric correctness of the symbolic specification semantics The framework performs all three analyses using the symbolic specification semantics (SSS, top-right of the diagram), which the tool developer must instantiate using the symbolic state model. This semantics is proven parametrically sound w.r.t. the concrete semantics of our trusted computing base. In other words, if the symbolic state model $\bar{\mathbb{S}}$ is sound w.r.t. the concrete state model $\underline{\mathbb{S}}$, then the SSS instantiated with $\bar{\mathbb{S}}$ is sound w.r.t. the concrete semantics (CS) instantiated with $\underline{\mathbb{S}}$.



To perform this proof, we introduce two intermediate semantics: the concrete compositional semantics (CCS) and the concrete specification semantics (CSS). First, we show that the soundness property connecting the compositional state model \mathbb{S} and the full state model $\underline{\mathbb{S}}$ lifts to an analogous property that connects the CCS and the CS. Then, we show that, provided validity of the consumer and producers of \mathbb{S} , the CSS—which replaces function calls with an “execution” of their specification,

¹⁰ All proofs in this manuscript are pen-and-paper proofs.



¹¹ We denote “fullness” with an underline.



rendering the analysis compositional—is sound w.r.t. the CCS. Finally, we show that if the symbolic state model $\bar{\mathbb{S}}$ is sound w.r.t. the concrete state model \mathbb{S} , then the SSS is sound w.r.t. the CSS, completing the connection between SSS and CS.

3.4 Correctness and Incorrectness

Throughout our formalisation of Gillian, we precisely identify which components of the framework determine whether the analysis is *over-approximate* (i.e. guaranteeing the absence of false negatives) or *under-approximate* (i.e. ensuring no false positives). We encourage the reader to think of selecting between over-approximation (OX) and under-approximation (UX) as toggling a switch. Each identified component must be set either to  or to , and this setting must be uniformly¹² applied across all components to maintain the integrity of the guarantees. Some components may universally apply to both OX and UX settings, in which case they are said to be *exact*. Although it is uncommon for all components to be exact, achieving exactness in any component enhances the precision of the analysis.

¹² Not doing so would break all guarantees of either absence of false-positives or false-negatives. In practice, one may compromise, for instance using a fast but OX solver for UX analysis. While a few false positives may appear, doing so may enable large-scale analysis. We know from communication with its developer that this is the approach currently taken by Infer Pulse.

<i>Component</i>		
Compositionality (Definition 4.4)	Frame subtraction	Frame addition
Concrete soundness (Definition 4.5)	All full path have a compositional path	All compositional paths have a full path
Core predicates (Definition 5.1)	No restriction	Strictly exact
Logic (Definition 5.11)	SL	ISL
Handling of reasoning errors (Miss & Lfail)	Signal error to the user	Ignore and drop
Solver (Definition 6.10)	In doubt, SAT	In doubt, UNSAT
Symbolic soundness (Definition 6.13)	All concrete paths have a symbolic path	All symbolic paths have a concrete path

We identify seven components influencing whether the analysis is  or . Table 3.1 provides an overview of these seven components, and we discuss each below in detail.

Compositionality There are two flavours of compositionality, one called [Frame subtraction](#), required for over-approximation, and one called [Frame addition](#), required for under-approximation. While most state models satisfy both properties, some exotic ones extracted from Iris (such as the OneShot resource algebra) do not satisfy frame addition.

Concrete soundness The compositional semantics used for analysis can be seen as a reification of a separation logic into semantics. In order

Table 3.1: The seven components that influence whether the analysis is over- or under-approximating.

to guarantee the correct properties, the compositional semantics must be either OX or UX-sound with respect to the (real) full semantics of the language.

Core predicates The core predicates are the building blocks of the separation logic assertion language. To enable under-approximate reasoning, all core predicates used in the analysis must be *strictly exact*, meaning they are satisfiable by at most one state given an interpretation of all logical variables. While this may sound like a strong requirement, it is satisfied by most core predicates used in practice, such as the points-to predicate. In contrast, over-approximate reasoning requires no restriction on the core predicates.

Logic The logic used in the specification semantics is either SL or ISL. The function call specifications are either OX or UX, depending on the employed logic. Note that whole-program symbolic testing does not use specifications and is therefore considered exact for this component. Furthermore, it is possible to use externally proven specifications that lie at the intersection of SL and ISL¹³, in which case function calls are also considered exact.

¹³ Maksimović et al., “Exact Separation Logic: Towards Bridging the Gap Between Verification and Bug-Finding”, 2023 [Mak+23]

Handling of reasoning errors Our framework accurately interprets all potential errors during execution, distinguishing between errors that correspond to a program bug and errors stemming from reasoning discrepancies. Specifically, we identify two categories of reasoning errors: *Miss* and *Lfail*. The former relates to reasoning shortcomings that could be rectified by providing additional resources, whereas the latter represents an incompleteness or a fundamental incompatibility. Neither type of error definitively indicates a bug in the program.

Accordingly, the handling of these errors is based on the analysis type. In scenarios where the analysis must guarantee the absence of false negatives, such errors should be reported to the user. Conversely, these errors should be disregarded and excluded from the results to ensure the absence of false positives. Both are unique to compositional analysis and do not arise during whole-program symbolic testing. Therefore, in the context of whole-program analysis, this framework component is considered exact.

Solver Symbolic execution relies on using an SMT solver. While formalisations of symbolic execution usually assume the solver to be perfect, most theories used during verification are undecidable, breaking this perfection. The SMT-lib¹⁴ standard dictates that an SMT query may return SAT, UNSAT or UNKNOWN, the last indicating a failure to conclude. Furthermore, the encoding of values into SMT may be under-specified (and therefore over-approximating) or over-specified (and hence under-approximating). To account for these imperfections, we propose a definition of *approximate solvers*, which can be compatible with OX or UX. Specifically, OX-approximate solvers may answer SAT when unable to decide on an answer, ensuring that all of the truly feasible paths are considered feasible. Conversely, UX-approximate

¹⁴ Barrett et al., *The Satisfiability Modulo Theories Library (SMT-LIB)*, 2016 [BFT16]

solvers may answer UNSAT in such cases, ensuring that no unfeasible path is considered feasible.

Symbolic soundness Symbolic execution is achieved by *abstracting* sets of concrete values into symbolic variables, inherently supporting approximation. In particular, OX-sound symbolic execution ensures that all concrete paths are covered by at least one symbolic path whereas UX-sound symbolic execution ensures that all symbolic paths must correspond to at least one concrete path. It is also possible to design exact symbolic executions, but only if the structures involved are sufficiently straightforward.

Chapter 4

Compositionality and parametricity

The first part of this manuscript aims to provide solid and extensive theoretical foundations for the soundness of compositional symbolic execution. Such soundness results should be proven with respect to the semantics of the language being analysed.

Many real-world languages provide an official description of their semantics, either in the form of extensive documents written using plain English¹ (e.g. the C standard² or the ECMAScript specifications for JavaScript³) or as a formal operational semantics (e.g. StandardML⁴ and WASM⁵).

On the other hand, the soundness of compositional reasoning based on separation logic is often justified using semantics that already satisfies a form of frame property⁶, which we call **Frame subtraction**. Semantics that satisfy this property are called *state-compositional* (or simply compositional) and describe the effects of programs on fractionable states (defined in §4.2).

However, official descriptions of the semantics of real-world languages are not compositional, leaving a gap in the justification of the soundness of compositional symbolic execution frameworks. In §4.3, we propose a general approach for connecting full semantics to compositional semantics such that results obtained from analysing programs using the latter can be formally transferred to the former.

Our approach builds on that used in JaVerT 2.0⁷, but is more general as it applies to any semantics, not just that of JavaScript⁸. The result is also similar to that obtained using the weakest-precondition approach of Iris⁹, but is formulated as a *semantics* property, instead of a logical one, enabling its integration within compositional symbolic execution. Finally, unlike both JaVerT 2.0 and Iris, our approach supports under-approximate reasoning as well.

While compositionality is proposed as a general construction, the rest of the framework is defined using a simple language called SIGIL with built-in support for function calls, if-else and other base constructs. It is parametric on a *state model*, which provides a set of states and a set of actions that modify the state. SIGIL is defined in §4.4, and basic examples of state models are presented in §4.5.

4.1 Languages and concrete semantics

In this section, we define a generic approach for justifying the soundness of an analysis performed using a *compositional* semantics with respect to the *full* (or *real*) semantics of a language \mathcal{L} . Such compositional semantics enable reasoning based on separation logic, including the use of *fictional separation* and *ghost resources*.

¹ While the problem of interpreting plain-English specifications into formal semantics is a widely studied matter, it is outside the scope of this manuscript.

² International Organization for Standardization, *ISO/IEC 9899:2018 - Information technology – Programming languages – C*, 2018 [Int18]

³ Ecma International, *ECMAScript 2023 Language Specification*, 2023 [Ecm23]

⁴ Milner et al., *The Definition of Standard ML*, 1997 [Mil+97]

⁵ Watt, “Mechanising and verifying the WebAssembly specification”, 2018 [Wat18]

⁶ Yang et al., “A Semantic Basis for Local Reasoning”, 2002 [YO02]

⁷ Fragoso Santos et al., “JaVerT 2.0: compositional symbolic execution for JavaScript”, 2019 [Fra+19]

⁸ the approach in JaVerT 2.0 applies to JSIL, an intermediate language for JavaScript analysis.

⁹ Jung et al., “Iris from the ground up: A modular foundation for higher-order concurrent separation logic”, 2018 [Jun+18]

Our approach applies to any non-deterministic expression language with side effects as long as its semantics is *total*, i.e. it is clearly defined for all possible inputs. In practice, languages such as C have partial semantics, where the evaluation of an expression may be undefined. Here, such *undefined behaviours* should be treated explicitly as errors instead of being left undefined. In fact, without characterising undefined behaviours explicitly, the soundness of an analysis that “finds undefined behaviours” cannot be formally justified.

We now define the class of languages that we consider. In these languages, expressions $e \in \text{Expr}$ are evaluated within a *function context* (or program) γ , a variable substitution θ , and a current state $\underline{\sigma}$. For simplicity, we assume that all languages use a standard countable set of variables $\text{Var} \ni \{x, y, \dots\}$. The result of the evaluation of an expression is a *set* of triples $(\underline{o}, v, \underline{\sigma}')$, where $\underline{o} \in \{\text{Ok}, \text{Err}\}$ is an *outcome*, indicating if the evaluation was successful or erroneous; v is the resulting value; and $\underline{\sigma}'$ is the updated state, which may have been modified, since expression evaluation may be side-effecting. Each element of the set represents a possible evaluation result, as expression evaluation may be non-deterministic.

Definition 4.1 (Expression language: syntax and semantics).

A syntax for an expression language is a pair $\mathcal{L} = (F\text{Ctx}, \text{Expr})$ where:

- $F\text{Ctx} \ni \gamma$, represents the set of function contexts (or programs); and
- $\text{Expr} \ni e$, represents the set of abstract syntax trees for program expressions.

Given an expression language \mathcal{L} , a semantics for that language is a triple $\underline{\mathcal{S}}_{\mathcal{L}} = (\text{Val}, \underline{\Sigma}, \text{eval})$. Its last component is the expression evaluation function, **eval**, which has the following signature¹⁰:

val eval : $F\text{Ctx} \rightarrow \text{Subst} \rightarrow \underline{\Sigma} \rightarrow \text{Expr} \rightarrow (\underline{\mathcal{O}} \times \text{Val} \times \underline{\Sigma}) \text{ set}$

where

- $\text{Subst} = \text{Var} \rightarrow \text{Val} \ni \theta$ is the set of *substitutions*, which map variables to values.
- $\underline{\Sigma} \ni \underline{\sigma}$ represents the set of states on which **eval** operates;
- $\underline{\mathcal{O}} = \{\text{Ok}, \text{Err}\}$ is the set of outcomes indicating the status of execution; and
- $\text{Val} \ni v$ represents the set of values obtainable from evaluation;

Notation. We write $\gamma \vdash \underline{\sigma}, e \Downarrow_{\theta} \underline{o} : (v, \underline{\sigma}')$ to denote $(\underline{o}, v, \underline{\sigma}') \in \text{eval } \gamma \theta \underline{\sigma} e$.

Generality This definition is general enough to capture many real-world languages. While OCaml naturally fits into this class of languages, imperative languages such as C can also be formulated as expression language where evaluating statements constantly evaluate to **unit**.

Furthermore, while we use the term “function context” for elements of $F\text{Ctx}$, the definition does not restrict the content of its elements to be a set of functions that expressions can call. For instance, context can be used to capture definitions of types, addresses of global variables, etc.

Substitution-based language The definition ensures that the semantics of variables is *substitution-based*¹¹. However, it is possible to carry

¹⁰ We write mathematical (meta-theoretical) function signatures and function applications using OCaml syntax to be more homogeneous between the theory and the implementation, even though most non-deterministic semantics are not implementable since resulting sets quickly become infinite. An implementation may, for example, use the sequence type `seq` instead of `set`, allowing for depth-first search over the reachable states or may deterministically select a single outcome instead of branching.

¹¹ A similar choice is made in Iris. Jung provides a thorough argument for this choice in section 7.3 of his thesis [Jun20].

a *mutable variable store* as part of the state $\underline{\sigma}$ ¹².

4.2 Partial commutative monoids

Compositional semantics operate on fragmentable states, i.e. states which can be decomposed into smaller states, and composed together into bigger states. We define the mathematical structure that captures this notion: partial commutative monoids (PCMs)¹³.

Definition 4.2 (Partial commutative monoid (PCM)).

A partial commutative monoid is a triple $(A, 0_A, \bullet_A)$ ¹⁴, where \bullet_A is a partial, associative, and commutative binary operation on A , and $0_A \in A$ is an identity element of this operation.

Notation. We write $a \# a'$, pronounced “ a is a disjoint from a' ”, if $a \bullet a'$ is defined. In addition, we often call *fragments* elements of a PCM.

Lemma 4.3 (PCMs: Induced preorder).

A PCM $(A, 0_A, \bullet)$ induces a preorder \preceq on A as follows:

$$a \preceq b \iff \exists c. b = a \bullet c$$

Commutative monoids and their induced preorder are the canonical structures used for defining heap fragments in the literature.¹⁵

Example The set of partial maps from booleans to booleans $\mathbb{B} \rightarrow \mathbb{B}$ form a PCM. Two partial maps can be composed together if their domains are disjoint and the null element is the empty map. The corresponding Hasse diagram is provided in Figure 4.1.

4.3 Concrete compositional semantics

Equipped with the mathematical tools to define state fragments, we define compositional semantics and how to connect them with their full counterparts.

Definition 4.4 (Concrete compositional semantics).

Let $\mathcal{L} = (FCtx, Expr)$ be the syntax for an expression language. $\mathcal{S}_{\mathcal{L}} = (Val, \Sigma, eval)$ is a *compositional* semantics for \mathcal{L} if Σ is the domain of a partial commutative monoid $(\Sigma, 0, \bullet)$, and *eval* has for signature:

val *eval* : $FCtx \rightarrow Subst \rightarrow \Sigma \rightarrow Expr \rightarrow (\mathcal{O} \times Val \times \Sigma)$ set

where $\mathcal{O} = \underline{\mathcal{O}} \cup \{\text{Miss}\} = \{0k, \text{Err}, \text{Miss}\}$ (the new outcome *Miss* is addressed below). It is said to be *OX-frame-preserving* (resp. *UX-frame-preserving*) if it satisfies the following frame subtraction (resp. frame addition) property:

$$\begin{aligned} \gamma \vdash (\sigma \bullet \sigma_f, e) \Downarrow_{\theta} o : (v, \sigma') &\implies \\ (\exists o', v', \sigma''. \gamma \vdash (\sigma, e) \Downarrow_{\theta} o' : (v', \sigma'') \wedge \\ (o' \neq \text{Miss} \Rightarrow \sigma' = \sigma'' \bullet \sigma_f \wedge o = o' \wedge v = v')) & \\ \text{(Frame subtraction)} \end{aligned}$$

$$\begin{aligned} \gamma \vdash (\sigma, e) \Downarrow_{\theta} o : (v, \sigma') \wedge \sigma' \# \sigma_f \wedge o \neq \text{Miss} & \\ \implies \gamma \vdash (\sigma \bullet \sigma_f, e) \Downarrow_{\theta} o : (v, \sigma' \bullet \sigma_f) & \quad \text{(Frame addition)} \end{aligned}$$

¹² In §8.8, we show how to do so. However, mutable variable stores famously complicate separation-logic reasoning.

¹³ Another structure used in Iris, called *resource algebra*, would be more adapted. However, for our presentation, PCMs are sufficient and easier to define.

¹⁴ When unambiguous, we omit the subscripts and use 0 and \bullet .

¹⁵ Yang et al., “A Semantic Basis for Local Reasoning”, 2002 [YO02]

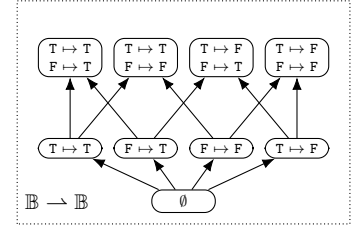



Figure 4.1: Hasse diagram for the partial commutative monoid of boolean partial maps $\mathbb{B} \rightarrow \mathbb{B}$.

We write $\gamma \vdash (\sigma, e) \Downarrow_{\theta} o : (v, \sigma')$ if $(o, v, \sigma') \in eval \ \gamma \ \theta \ \sigma \ e$

Definition 4.5 (Compositional semantics soundness).


The compositional semantics $\mathcal{S}_\mathcal{C}$ is said to be *OX-sound* with respect to the full semantics $\mathcal{S}_\mathcal{L}$ according to the satisfiability relation \models_c if:

$$\begin{aligned} \gamma \vdash (\underline{\sigma}, e) \Downarrow_\theta o : (v, \underline{\sigma}') \wedge \sigma \models_c \underline{\sigma} \implies \\ (\exists o', v', \sigma'. \gamma \vdash (\sigma, e) \Downarrow_\theta o' : (v', \sigma') \wedge \\ (o' \neq \text{Miss} \Rightarrow \sigma' \models_c \underline{\sigma}' \wedge o = o' \wedge v = v')) \end{aligned}$$

( concrete soundness)

Conversly, it is said to be *UX-sound* if:

$$\begin{aligned} \gamma \vdash (\sigma, e) \Downarrow_\theta o : (v, \sigma') \wedge \sigma' \models_c \underline{\sigma}' \wedge o \neq \text{Miss} \\ \implies \exists \underline{\sigma}. \gamma \vdash (\underline{\sigma}, e) \Downarrow_\theta o : (v, \underline{\sigma}') \wedge \sigma \models_c \underline{\sigma} \end{aligned}$$


( concrete soundness)


We now provide a detailed interpretation of these properties, including the meaning of the new missing outcome.

The missing outcome Evaluating an expression using compositional semantics may result in a new outcome, called *missing* and denoted **Miss**. This outcome does not exist in the full semantics and is an artefact of compositionality: it indicates that the state fragment used for evaluation does not contain enough information to evaluate the statement.

A missing outcome does not imply anything about the real execution of the program, but is rather due to an incompleteness in the reasoning when using the compositional semantics.

Compositional semantics for analysis The four rules provided in the definitions above sufficiently justify the use of compositional semantics for bug-finding and verification.

 **concrete soundness** states that, if a transition exists starting from state $\underline{\sigma}$ in the full semantics, then a corresponding transition *must* exist in the compositional semantics, for any compositional state modeling $\underline{\sigma}$. In turn, **Frame subtraction** ensures that if a transition is found in the compositional semantics, then a corresponding transition *must* exist for any smaller state fragment. In both cases, **Miss** acts as an escape hatch: either the execution is preserved, or it is missing, indicating that the state fragment is not sufficient to evaluate the expression. Together, these properties justify the use of state fragments in the compositional semantics for verification: if there is a bug to be found in the full semantics, then it will be found using any fraction of the corresponding state, provided it contains enough information.

Conversly,  **concrete soundness** states that any state reached in the compositional semantics must correspond to a reachable state in the full semantics, while **Frame addition** ensures that outcomes reached using a fraction of a state is also reachable using any larger state containing it, and does not affect the frame. Dually to over-approximating properties, these two properties justify the use of compositional semantics for true bug-finding guaranteeing the absence of false positives.

Compositional semantics do not need to use the same set of states as the full semantics. For instance, compositional states may contain additional information, called *ghost resources*. In [Part III](#), we show

that lifetime tokens, imported from the RustBelt¹⁶ formalism, become ghost resources in the compositional semantics used in Gillian-Rust. As such, compositional semantics can be seen as *logical executions*: where Iris proves all theorems in a logic proven sound with respect to the full semantics, compositional symbolic execution moves this reasoning to the semantics.

Notation. We write $\mathcal{S}_{\mathcal{L}} \stackrel{c}{\sim}_m \underline{\mathcal{S}}_{\mathcal{L}}$, for $m \in \{\text{OX}, \text{UX}\}$, if $\mathcal{S}_{\mathcal{L}}$ is a m -frame-preserving compositional semantics that is m -sound with respect to $\underline{\mathcal{S}}_{\mathcal{L}}$. We also use $m = \text{EX}$ or elide the m when it holds for both $m = \text{OX}$ and $m = \text{UX}$.

Frame is a soundness property Note that the frame conditions are obtained by specialising the soundness conditions to the relation $\models_{\mathcal{C}}^{\sigma_f}$ between two state fragments such that $\sigma \models_{\mathcal{C}}^{\sigma_f} \sigma'$ if $\sigma = \sigma' \cdot \sigma_f$. In other words, the compositional semantics $\mathcal{S}_{\mathcal{L}}$ is m -frame-preserving if $\forall \sigma_f. \mathcal{S}_{\mathcal{L}} \stackrel{c}{\sim}_m \underline{\mathcal{S}}_{\mathcal{L}}$, using $\models_{\mathcal{C}}^{\sigma_f}$.

While this statement has no direct practical implications, it is useful for understanding frame conditions, and hints at a possible generalisation of this approach, which is left for future work.

4.4 SIGIL: a parametric intermediate language

Gillian performs its analyses on an intermediate language called GIL¹⁷ that is *parametric* on a state model. This means that some fundamental operations of the semantics must be provided as parameters. Out of the box, GIL proposes syntactic constructs for GOTO-based control-flow, manipulating a mutable store, returning modes for handling exceptions as well as some legacy features¹⁸ such as ϕ -nodes¹⁹.

For this formalisation, we introduce a simpler (yet more flexible) language, dubbed SIGIL. It offers simpler control flow based on if-else and while loops and does not provide a mutable store²⁰.

In this section, we present SIGIL's syntax and semantics. The semantics of SIGIL is parametric on a set of *actions* $\mathcal{A} \ni \alpha$ which manipulate the state.

4.4.1 Syntax

The syntax of SIGIL is defined as follows:

$$\begin{aligned} z \in \mathbb{Z} \quad & b \in \mathbb{B} \quad & f \in \text{Fid} \quad & \mathbf{x}, \mathbf{y}, \mathbf{z} \in \text{Var} \\ v \in \text{Val} \quad & ::= () \mid z \mid b \mid [\vec{v}] \\ e_p \in \text{PEXpr} \quad & ::= \mathbf{x} \mid v \mid e_p \oplus e_p \mid \ominus e_p \mid [\vec{e}_p] \\ e \in \text{EXpr} \quad & ::= e_p \mid \text{let } \mathbf{x} = e \text{ in } e \mid \text{if } e \text{ then } e \text{ else } e \mid \\ & f(\vec{e}) \mid \langle \alpha \rangle(\vec{e}) \end{aligned}$$

$$\text{FunDef} = \{f(\vec{x}) \{e\} \mid \text{fv}(e) \subseteq \vec{x}\} \quad \gamma \in \text{FCtx} = \text{Fid} \xrightarrow{\text{fin}} \text{FunDef}$$

SIGIL values $v \in \text{Val}$ contain the unit value $()$, integers $z \in \mathbb{Z}$, booleans $b \in \mathbb{B}$, and lists of values $[\vec{v}]$. Pure expressions, $e_p \in \text{PEXpr}$ are guaranteed to be non-side-effecting, and contain variables $\mathbf{x} \in \text{Var}$, values $v \in \text{Val}$, binary operators $e_p \oplus e_p$, unary operators $\ominus e_p$, and

¹⁶ Jung et al., “RustBelt: securing the foundations of the Rust programming language”, 2017 [Jun+17]

¹⁷ GIL stands for *Gillian's Intermediate Language*

¹⁸ These features are used by the Gillian-JS compiler, but are irrelevant to the analyses performed by Gillian

¹⁹ Alpern et al., “Detecting equality of variables in programs”, 1988 [AWZ88]

²⁰ The mutable store can be implemented as part of the state model (see Chapter 8). However, some of its properties are not desirable.

lists of pure expression $[\vec{e}_p]$. Expressions²¹, $e \in Expr$ contain pure expressions, let-bindings, if-else, function calls and action calls where the action name α is between angular brackets. A SIGIL function $f(\vec{x}) \{e\}$ comprises an identifier $f \in Fid$, formal parameters \vec{x} , and a body given by an expression e where all free variables of e (those not bound using **let**) are formal parameters²². A SIGIL program is a set of functions with unique identifiers.

4.4.2 Semantics

We describe the parametric semantics of SIGIL using a combination of OCaml and mathematical syntax instead of using inference rules. In doing so, we aim to provide the reader with an immediate understanding of the corresponding implementation without losing mathematical precision. Furthermore, inference rules can lead to awkward notations when the *absence* of semantic transitions is relevant, whereas a mathematically defined evaluation function may explicitly return the empty set. Finally, it allows for using well-defined syntactic sugar for monadic let-binding, rendering definitions more concise and intuitive.

In this subsection, we present the intuition behind the execution monad that underpins expression evaluation, introduce the let-binding notation in OCaml, and subsequently define the parametric semantics of SIGIL using these newly defined notions.

The execution monad An evaluation function – whether for full or compositional semantics – returns a set of triples, with each triple consisting of an outcome, a value and a state, with the outcome indicating successful or erroneous evaluation. The function returns a *set* of triples rather than a single triple because it accounts for all possible executions under non-determinism. More formally, the signature of an evaluation function is of the form $X \rightarrow (\mathcal{O} \times Y) \text{ set}$, which can be seen as a computation (or Kleisli arrow) in the following *execution monad*^{23,24}:

```
type 'a exec = ( $\mathcal{O} \times 'a$ ) set

(* A computation from 'a to 'b is a function which,
   given a value of type 'a returns an execution of type 'b *)
type ('a, 'b) computation = 'a  $\rightarrow$  'b exec
```

It is often convenient to compose several such computations to define more complex computations. Typically, one would describe the evaluation of a binary operator as: “evaluate the left operand, **then** evaluate the right operand, **then** apply the operator to the results”. A corresponding implementation would be:

```
let eval_binop  $e_1 \oplus e_2$  =
  let  $v_1$  = eval  $e_1$  in
  let  $v_2$  = eval  $e_2$  in
   $v_1 \oplus v_2$ 
```

Unfortunately, this definition is invalid, as it does not account for either operand evaluation failing or branching. However, both effects should be handled consistently during evaluation: erroneous results terminate execution while successful branches continue independently.

It is possible to seamlessly compose computations by defining the function which describes how to apply a new computation to the results

²¹ Note that expressions do *not* contain loops. These (together with potential loop invariants required for compositional verification) could be compiled away to recursive functions or handled directly by adding minor extensions to the formalism.

²² The side condition on free variables being formal parameters trivially avoids the problem of undefined variables. It also has other benefits, specifically when defining the notion of specification in the next chapter.

²³ We do not formally define the notion of monad, as this presentation only aims at explaining monadic composition for the unfamiliar reader. The literature is full of accessible definitions of monad from the category theory or functional programming perspective (e.g. [Mil18]), and providing such a definition would be redundant.

²⁴ The knowledgeable reader may notice that the execution monad is obtained by applying the outcome (or result) monad transformer to the non-determinism monad.

of a first computation. Traditionally, this function is called `bind`:

```
let bind (results: 'a exec) (computation: 'a → 'b exec) : 'b exec =
  ∪res ∈ results {
    match res with
    (* If the previous computation succeeded,
       continue by applying the next computation to the result *)
    | Ok z → computation z
    (* If the previous computation failed
       (with an error or missing outcome),
       applying a new computation has no effect *)
    | Error z → { Error z }
    | Miss z → { Miss z }
  }
```

This `bind` operator allows for a valid definition of binary operator evaluation in terms of sub-computations:

```
let eval_binop e1 ⊕ e2 =
  bind
    (eval e1)
    (fun v1 →
      bind (eval e2)
        (fun v2 → { ok: v1 ⊕ v2 }))
```

Monadic let-binding in OCaml While the definition above uses the `bind` operator to handle non-determinism and errors systematically, reading it is somewhat cumbersome. Throughout this thesis, we make extensive use of the custom `let`-binding syntax extensions of OCaml^{25,26} to make applications of the `bind` operator more readable.

In OCaml, it is possible to define custom `let*` operators, such that

`let* x = e in e'` is desugared to `bind e (fun x → e')`

Using this `let*` operator, evaluation of binary operation finally becomes natural to read and concise to define while seamlessly handling errors and non-determinism:

```
let eval_binop e1 ⊕ e2 =
  let* v1 = eval e1 in
  let* v2 = eval e2 in
  v1 ⊕ v2
```

This definition corresponds to the following rules:

$$\begin{array}{c}
\text{OK} \\
\frac{e_1 \Downarrow \text{Ok} : v_1 \quad e_2 \Downarrow \text{Ok} : v_2}{v_1 \oplus v_2 \Downarrow \text{Ok} : v_3} \\
\hline
e_1 \oplus e_2 \Downarrow \text{Ok} : v_3
\end{array}
\qquad
\begin{array}{c}
\text{LEFT-ERROR} \\
\frac{e_1 \Downarrow o : v_1 \quad o \in \{\text{Err}, \text{Miss}\}}{e_1 \oplus e_2 \Downarrow o : v_1}
\end{array}$$

$$\begin{array}{c}
\text{RIGHT-ERROR} \\
\frac{e_1 \Downarrow \text{Ok} : v_1 \quad e_2 \Downarrow o : v_2 \quad o \in \{\text{Err}, \text{Miss}\}}{e_1 \oplus e_2 \Downarrow o : v_2}
\end{array}
\qquad
\begin{array}{c}
\text{OP-ERROR} \\
\frac{e_1 \Downarrow \text{Ok} : v_1 \quad e_2 \Downarrow \text{Ok} : v_2 \quad v_1 \oplus v_2 \Downarrow o : v_3 \quad o \in \{\text{Err}, \text{Miss}\}}{e_1 \oplus e_2 \Downarrow o : v_3}
\end{array}$$

Finally, we define four utility functions, each corresponding to a simple computation that either yields a single branch of execution²⁷, or no branch at all:

```
let ok z = { Ok z }
let error z = { Err z }
let miss z = { Miss z }
let vanish = ∅
```

²⁵ The OCaml Team, *The OCaml Manual - Ch. 12.23: Binding Operators*, 2023 [The23b]

²⁶ Custom `let`-bindings are akin to the `do`-notation in Haskell.

²⁷ `ok` is the unit operation of our execution monad

Semantics of pure expressions For exhaustiveness, we outline a semantics $\llbracket e_p \rrbracket_\theta$ for pure expressions. Note that any *deterministic* semantics that does not depend on the state would be appropriate. For instance, our treatment of binary operators is not short-circuiting, and both operands are always evaluated regardless of necessity (e.g., in expressions like `true` \wedge ...). However, such design choice is orthogonal to the rest of the framework²⁸.

For the sake of brevity, we detail only a selection of cases, with the omitted cases following analogously. Values are evaluated as themselves, substitutions θ are straightforwardly applied to variables, and binary operators are evaluated without short-circuiting. Note that the successful evaluation of operands does not guarantee the successful application of binary operators, such as in the event of division by zero.

```

let  $\llbracket e_p \rrbracket_\theta =$ 
  match  $e_p$  with
  |  $v \rightarrow \text{ok } v$ 
  |  $x \rightarrow \text{ok } \theta(x)$ 
  |  $e_p \oplus e'_p \rightarrow$ 
    let*  $v = \llbracket e_p \rrbracket_\theta$  in
    let*  $v' = \llbracket e'_p \rrbracket_\theta$  in
     $v \oplus v'$ 
  | ...

let  $v \oplus v' =$ 
  match  $\oplus$  with
  |  $+$   $\rightarrow \text{ok } v + v'$ 
  |  $/$   $\rightarrow$ 
    if  $v' = 0$  then
      error DivByZero
    else
      ok  $v / v'$ 
  | ...

```

The semantics of pure expressions is expressed within the execution monad, despite its deterministic nature. We do this to avoid introducing new notations²⁹.

Parametric semantics The semantics of SIGIL is parametric on the type of states and a set of actions that operate on the states. We define *full state models* here and *compositional state models* in the following subsection to provide these parameters. Using a full state model results in a full semantics, while using a compositional state model results in a compositional semantics.

Full state models A full state model $\underline{\mathbb{S}}$ is a module³⁰ with the following signature signature:

```

module type Full_state_model = sig
  type  $\underline{\Sigma}$  (* Type of states *)
  type  $\mathcal{A}$  (* Type of actions *)
  val eval_action :  $\mathcal{A} \rightarrow \underline{\Sigma} \rightarrow \text{Val list} \rightarrow (\mathcal{Q} \times \text{Val} \times \underline{\Sigma})$  set
end

```

Notation. We write

$$\underline{\sigma}.\alpha(\vec{v}) \xrightarrow{\underline{\mathbb{S}}} o : (v, \underline{\sigma}')$$

if $(o, v, \underline{\sigma}') \in \underline{\mathbb{S}}.\text{eval_action } \alpha \ \sigma \ \vec{v}$, and omit the state model above the arrow when it is clear from the context.

Semantics of expressions A state model $\underline{\mathbb{S}}$ induces a semantics $\mathcal{S}_{\underline{\mathbb{S}}} = (\text{Val}, \underline{\Sigma}, \text{eval } \underline{\mathbb{S}}.\text{eval_action})$ for SIGIL. This induced semantics enables the support of the SIGIL control flow constructions for free, including function calls, and systematically handles pure expression evaluation and variable bindings.

Figure 4.2 defines the parametric semantics of SIGIL. The eval function receives a state model $\underline{\mathbb{S}}$, a function context γ , a substitution θ , a

²⁸ The syntax and semantics of pure expressions could be provided as an additional parameter to the framework. We avoid this approach so as not to clutter the presentation further.

²⁹ The `let*` operator should be defined as the `bind` operator for the outcome monad, which handles errors but not non-determinism.

³⁰ Throughout this presentation, we use the OCaml module syntax to denote mathematical tuples and module signatures to denote the types of these tuples. For instance, the `Full_state_model` signature corresponds to the tuples of the form $(\underline{\Sigma}, \mathcal{A}, \text{eval_action})$ where the last component has the appropriate signature. If $\underline{\mathbb{S}}$ is a full state model, we use module field accessors to denote its components, e.g., $\underline{\mathbb{S}}.\text{eval_action}$

Note Full state models can be seen as *local* languages, with the following language and semantics:

$$\begin{aligned} \mathcal{L} &= (\text{unit}, (\mathcal{A} \times \text{Val list})) \\ \underline{\mathcal{S}} &= (\text{Val}, \underline{\Sigma}, \text{eval_action}) \end{aligned}$$

```

let rec eval $ \gamma \theta \sigma e: (\mathcal{O} \times Val \times \Sigma) set =
  let eval = eval $ \gamma in (* These parameters are constant *)
  match e with
  | ep →
    let (o, v) = [[ep]]\theta in
    { (o, v, \sigma) }
  | let x = e1 in e2 →
    let* (v, \sigma') = eval \theta \sigma e1 in
    eval \theta [x ← v] \sigma' e2
  | if eg then et else ee →
    let* (b, \sigma') = eval \theta \sigma eg in
    let* () = assert_type b \mathbb{B} in
    if b then eval \theta \sigma' et else eval \theta \sigma' ee
  | f(\vec{e}) →
    let* f(\vec{x})\{e\} = \gamma[f] in
    let* (\vec{v}, \sigma') = eval_all \theta \sigma \vec{e} in
    let \theta_f = [\vec{x} → \vec{v}] in
    eval \theta_f \sigma' e
  | \langle \alpha \rangle(\vec{e}) →
    let* (\vec{v}, \sigma') = eval_all \theta \sigma \vec{e} in
    $.\text{eval\_action } \alpha \sigma' \vec{v}

```

state σ and the expression e to evaluate. Pure expressions are evaluated using the previously-defined function; let-binding evaluates the associated expression and updates the substitution, potentially shadowing a previous binding; if-then-else evaluates the guard, performs a dynamic type check³¹, and chooses the corresponding branch according to the result; function call fetches the appropriate function definition from the program, evaluates the arguments³² and executes the function body using the appropriate substitution; and action call evaluates the arguments and feeds them to the action evaluation function.

4.4.3 Compositionality

Compositional state models Compositional state models are similar to full state models, but their action evaluation function must return the `Miss` outcome and be frame-preserving.

```

module type Compositional_state_model = sig
  type \Sigma (* Type of state fragments *)
  type \mathcal{A} (* Type of actions *)
  val eval_action : \mathcal{A} \rightarrow \Sigma \rightarrow Val list \rightarrow (\mathcal{O} \times Val \times \Sigma) set

  (* ... more components, in next chapter *)
end

```

Notation. We write

$$\sigma.\alpha(\vec{v}) \xrightarrow{\$} o : (v, \sigma')$$

if $(o, v, \sigma') \in \$.\text{eval_action } \alpha \sigma \vec{v}$, and omit the state model above the arrow when it is clear from the context.

In addition, a compositional state model $\$$ is m -sound with respect to the full state model $\underline{\$}$ if they share their set of actions ($\$. \mathcal{A} = \underline{\$}. \mathcal{A}$), $\$. \Sigma$, and the action evaluation of $\$$ is m -sound with respect to that of $\underline{\$}$ (as defined by 4.5).

Notation. We write $\$ \stackrel{\mathcal{C}}{\sim}_m \underline{\$}$, $m \in \{\text{OX}, \text{UX}, \text{EX}\}$ if $\$$ is a compositional state model that has m -frame-preserving and m -sound with respect to $\underline{\$}$. We sometimes write $\$ \stackrel{\mathcal{C}}{\sim} \underline{\$}$ as a shorthand for $\$ \stackrel{\mathcal{C}}{\sim}_{\text{EX}} \underline{\$}$.

Figure 4.2: Concrete parametric semantic of SiGIL, the simplified GIL language. It is polymorphic in the type of state and can be used with either a full or a compositional state model.

We pass the state model as an argument, even though this is not valid OCaml. In OCaml, we define a functor that receives the state model as a parameter and yields a module containing the eval function.

³¹ The `assert_type` function checks that a value is of the right type, successfully returns unit if it does, and otherwise yields an error.

³² We use an `eval_all` function, which evaluates a list of arguments from left to right and stops if an error is encountered.

Compositional SIGIL semantics The parametric semantics of SIGIL given in Figure 4.2 is polymorphic on the type of states used. It can be used with a full state model $\underline{\mathbb{S}}$ or a compositional state model \mathbb{S} .

A fundamental property of SIGIL is that its semantics *preserves compositionality and soundness*:

Theorem 4.6 (Preservation of compositionality).

$$\mathbb{S} \underset{m}{\overset{c}{\sim}} \underline{\mathbb{S}} \implies \mathcal{S}_{\mathbb{S}} \underset{m}{\overset{c}{\sim}} \mathcal{S}_{\underline{\mathbb{S}}}$$

Notation. Because the parametric semantics of SIGIL is defined independently of compositionality, we use the down-facing arrow notation without the underline to denote the transition relation. However, when ambiguous, we annotate the arrow with the parameter state model as superscript. Hence, the following two notations correspond respectively to a full semantics (left) and a compositional semantics (right):

$$\gamma \vdash (\underline{\sigma}, e) \Downarrow_{\theta}^{\underline{\mathbb{S}}} \underline{\sigma} : (v, \underline{\sigma}') \quad \gamma \vdash (\sigma, e) \Downarrow_{\theta}^{\mathbb{S}} \sigma : (v, \sigma')$$

4.5 Examples of state models

We now introduce our main state model examples: the pure state model and the linear heap. As we define more concepts, these examples will be reused and extended throughout our presentation of the framework.

4.5.1 The pure state model

Definition The simplest state model one can define corresponds to the trivial monoid, which contains a unique element:

$$\Sigma = \{0\} \cong \text{unit}$$

We call it the *pure* state model, denoted **Pure**, as it is suited to any action that is independent of the state (sometimes called *state-less*). It can be used to model pure languages such as Haskell or composed with other state models to provide state-less actions.

In Figure 4.3, we present four state-less actions: **skip** is a no-op; **nondet** non-deterministically returns any value $v \in \text{Val}$; **assert** errors if a given condition is not true; and **assume** vanishes if a given condition is not true.

Again, the semantics of these actions is described using a combination of mathematics and OCaml syntax. The semantics of each action are separated from the primary `eval_action` function, which is in charge of validating the number and type of arguments. In later examples, we only show the individual implementation of each action and omit the primary `eval_action` function behaving similarly.

In case of error, these actions return special values such as `FailedAssert` or `InvalidArguments`. These values could be encoded using the native values of SIGIL (for example, using lists of integers in ASCII encoding).

Compositionality The unit monoid acts as a set of fragments for itself, and all of the above-defined actions are frame-preserving since they are

```

module Pure =

  (* Unit is the type that only
     contains the value () *)
  type  $\Sigma$  = unit
  val () = ()

  type  $\mathcal{A}$  = skip | nondet | assert | assume

  let skip  $\sigma$  = ok ((),  $\sigma$ )

  let nondet  $\Sigma$  =
    { Ok (v,  $\sigma$ ) | v  $\in$  Val }

  let assert  $\sigma$  b =
    if b then ok ((),  $\sigma$ )
    else error (FailedAssert,  $\sigma$ )

  let assume  $\sigma$  b =
    if b then ok ((),  $\sigma$ )
    else vanish

  let eval_action  $\sigma$   $\alpha$   $\vec{v}$  =
    match  $\alpha$ ,  $\vec{v}$  with
    | skip, []  $\rightarrow$  skip  $\sigma$ 
    | nondet, []  $\rightarrow$  nondet  $\sigma$ 
    | assert, [ b ] when b  $\in \mathbb{B}$   $\rightarrow$  assert  $\sigma$  b
    | assume [ b ] when b  $\in \mathbb{B}$   $\rightarrow$  assume  $\sigma$  b
    | _  $\rightarrow$  error (InvalidArguments,  $\sigma$ )
end

```

Figure 4.3: Formal definition of the pure state model Pure.

independent of the state. Therefore, using equality as the relation \models_c we have that $\text{Pure} \stackrel{c}{\sim} \text{Pure}$.

Generality Because these actions are independent of the state³³, they are compatible with any other state model. In the current implementation of Gillian, they are provided as commands of GIL and supported out-of-the-box for any state model.



³³ In fact, they are *polymorphic* on the type of state.

4.5.2 Linear heap

The linear heap is the heap model classically used in papers describing separation logics, such as the original Reynolds paper³⁴, or the paper introducing incorrectness separation logic.³⁵ We define both the full state model and an EX-sound compositional state model, providing detailed intuition and formal descriptions.

Full states and actions A full linear heap $\underline{\sigma} \in \underline{\Sigma}$ consists of a partial finite map from natural numbers to either a value $v \in \text{Val}$ or a special value \emptyset indicating that the heap address has been freed³⁶:

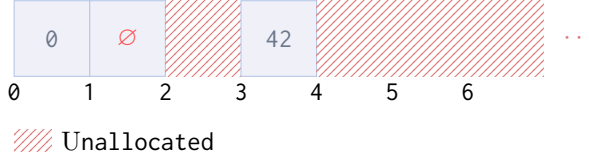
$$\underline{\Sigma} = \mathbb{N} \xrightarrow{\text{fin}} \text{Val}^\emptyset$$

For example, the heap $\underline{\sigma} = [0 \mapsto 0, 1 \mapsto \emptyset, 3 \mapsto 42]$ can be represented as in Figure 4.4, where the cells in blue  have been previously allocated, and the addresses covered with red stripes  are yet to be allocated.

³⁴ Reynolds, “Separation Logic: A Logic for Shared Mutable Data Structures”, 2002 [Rey02]

³⁵ Raad et al., “Local Reasoning About the Presence of Bugs: Incorrectness Separation Logic”, 2020 [Raa+20]

³⁶ $\text{Val}^\emptyset = \text{Val} \uplus \{\emptyset\}$

Figure 4.4: A diagram representing the heap $\underline{\sigma} = [0 \mapsto 0, 1 \mapsto \emptyset, 3 \mapsto 42]$

The heap is manipulated through four actions: **alloc**, **load**, **store**, and **free**. The **alloc** operation non-deterministically selects an unallocated address and initializes its value to the natural number 0. Both the **load** and **store** actions require that the target address has been previously allocated and not yet freed; **load** reads the value at the address, while **store** updates it. The **free** action requires that the targeted cell has been allocated but not freed, and it changes its content to \emptyset .

We provide formal definitions for **alloc** and **load**; the other actions are defined analogously. In the following, we assume that accessing a partial map such as $\underline{\sigma}$ using an index a returns either *Some* v if $\underline{\sigma}(a) = v$, and *None* if $a \notin \text{dom}(\underline{\sigma})$:

```

let alloc  $\underline{\sigma} =$ 
  let  $a = \text{nondet\_in } (\mathbb{N} \setminus \text{dom}(\underline{\sigma}))$  in
  ok ( $a, \underline{\sigma}[a \leftarrow 0]$ )

let load  $\underline{\sigma} (a : \mathbb{N}) =$ 
  match  $\underline{\sigma}(a)$  with
  | Some  $v \rightarrow$  ok ( $v, \underline{\sigma}$ )
  | Some  $\emptyset \rightarrow$  error (UseAfterFree,  $\underline{\sigma}$ )
  | None  $\rightarrow$  error (NotAllocated,  $\underline{\sigma}$ ) (*  $a \notin \text{dom}(\underline{\sigma})$  *)

```

Compositional state model Linear heap fragments are simply linear heaps ($\Sigma = \underline{\Sigma}$) given a partial commutative monoid structure, using the empty map as the unit element, and disjoint union as the composition operator; that is, the composition of two linear heap is only defined if their domains are disjoint. The \models_c relation connecting Σ and $\underline{\Sigma}$ can then simply be defined as equality ($\underline{\sigma} \models_c \sigma \iff \underline{\sigma} = \sigma$).

However, without care, using the same definition of actions as for full heaps would lead to a violation of the frame properties. For instance, trying to load the address 0 in the empty heap \emptyset would yield a **(Err, NotAllocated, \emptyset)** but loading the same address in the heap $[0 \mapsto 0] = [0 \mapsto 0] \uplus \emptyset$ would successfully return the value 0. This directly breaks the **Frame addition** property, which requires that, since the action in the heap \emptyset yields a non-Miss outcome, adding more resources must not interfere with the results.

Effectively, in the compositional state model, absence of an address in the heap cannot distinguish between an unallocated address and absence of information about this address, as in Figure 4.5 where such cells do not convey any information.

Figure 4.5: A diagram representing the heap fragment $\sigma = [0 \mapsto 0, 1 \mapsto \emptyset, 3 \mapsto \emptyset, 4 \mapsto 42]$

To overcome this issue, the load action must return a **Miss** outcome

instead of `Err` when the address is not in the domain of the heap.

```
let load  $\sigma$  (a :  $\mathbb{N}$ ) =
  match  $\underline{\sigma}(a)$  with
  | Some v  $\rightarrow$  ok (v,  $\underline{\sigma}$ )
  | Some  $\emptyset \rightarrow$  error (UseAfterFree,  $\underline{\sigma}$ )
  | None  $\rightarrow$  miss (MissingCell,  $\sigma$ )
```

A form of incompleteness Using this compositional state model, which corresponds to “standard” separation logic, prevents the specification of `NotAllocated` bugs. For compositional bug-finding, presented in [Chapter 7](#), this means that `NotAllocated` bugs cannot soundly be detected. There is no resource capturing that an address has not yet been allocated, and therefore it is impossible to conclusively reach the corresponding erroneous outcome. It would be possible to design a state model which carries a set of non-allocated addresses, but this would significantly increase the complexity of the reasoning.

Generalisation The linear heap is an instance of the partial map state model and can be constructed through combinators defined in [Chapter 8](#). Therefore, we delay the proofs of frame-preservation and soundness to that chapter.

Chapter 5

Parametric Assertion Language and Specification Execution

The previous chapter defined the notion of compositional semantics and SIGIL, a parametric intermediate language. This chapter presents a parametric assertion language for SIGIL and a novel semantics wherein function calls can be substituted with the *execution* of SL or ISL specifications. Specification execution underpins all compositional analyses, allowing each function to be treated independently by leveraging the specifications of other functions. This offers numerous benefits, such as reducing the complexity of the analysis, drastically improving performance and enabling unbounded reasoning.

To perform specification execution, Gillian makes use of the *producer-and-consumer* paradigm originally introduced in VeriFast¹, and later reused in Viper^{2,3} and JaVerT 2.0⁴. Producers are often characterised as “spatial assumes” as they extend a state fragment with the resource corresponding to an assertion. On the other hand, consumers are characterised as “spatial asserts”, as they assert that a state fragment contains the resources corresponding to an assertion and remove it from the state. An important part of consumption is the process of *matching*, which consists in identifying the fragment of the state that corresponds to the assertion being consumed, thereby instantiating quantified and free variables. Executing a specification then corresponds to consuming its pre-condition before producing its post-condition.

Parametricity Since SIGIL is parametric on a state model, the assertion language must also be. Tool developers using Gillian are required to provide a set of *core predicates*, which form the building blocks of this parametric assertion language. For example, the points-to predicate $l \mapsto v$ is a core predicate in the linear heap. Our novel formalisation of Gillian takes parametricity one step further by requiring *all* atoms of the logic to be formulated as core predicates, even pure assertions. This reduces the complexity of the framework’s core, enabling a more modular design.

Tool developers must implement a producer and a consumer as part of the compositional state model:

```
module type Compositional_state_model = sig
  type  $\Sigma$  (* Type of states *)

  type  $\mathcal{A}$  (* Type of actions *)
  val eval_action : (* See previous chapter *)

  type  $\Delta$  (* Type of core predicates, §5.1 *)
  val produce : (* Signature of the producer, §5.2 *)
  val consume : (* Signature of the consumer, §5.3 *)
end
```

The framework then provides a *parametric assertion producer* and

¹ Jacobs et al., “VeriFast: A Powerful, Sound, Predictable, Fast Verifier for C and Java”, 2011 [Jac+11]

² Müller et al., “Viper: A Verification Infrastructure for Permission-Based Reasoning”, 2016 [MSS16b]

³ In Viper, `produce` and `consume` are respectively called `inhale` and `exhale`

⁴ Fragoso Santos et al., “JaVerT 2.0: compositional symbolic execution for JavaScript”, 2019 [Fra+19]

parametric assertion consumer, which are in charge of handling variables and composition. This behaviour mirrors the separation of concerns between actions and the parametric semantics in SIGIL.

Correctness The framework guarantees the soundness of specification execution, provided that the core predicate producer and consumer each satisfy a correctness property. In addition, for UX analysis only, the core predicates must be *strictly exact*, meaning that they are satisfied by at most one state fragment.

Importantly, we formulate the correctness properties of the framework with respect to the semantics of the assertions, i.e. a satisfiability relation of the form $\theta, \sigma \models P$. This approach enables the sound use of specifications proven outside of the framework – for instance, by hand – as long as the external proof uses the same semantics for the assertion language.

5.1 Parametric assertion language

SIGIL proposes a minimal parametric assertion language:

$$P, Q \in \text{Asrt} ::= \langle \delta \rangle(\vec{e}_p^+; \vec{e}_p^-) \mid \exists \mathbf{x}. P \mid P * Q$$

which consists of core predicate assertions, $\langle \delta \rangle(\vec{e}_p^+; \vec{e}_p^-)$, closed under existential quantification and separating conjunction. Core predicate assertions come with a name, δ , and two lists of parameter expressions respectively called *in-parameters* (or *ins*) and *out-parameters* (or *outs*).⁵

This assertion language does not explicitly provide some usual suspects, such as pure assertions or the empty assertion, **emp**. Instead, these can be defined as core predicates compatible with any state model, as illustrated shortly. We view our minimalistic approach to the assertion language as a strength of our framework, as it facilitates highly modular proofs.

Core predicates To instantiate the above-defined parametric assertion language, compositional state models are extended with a set of core predicates, $\Delta \ni \delta$, that are characterised using a satisfiability relation.

Definition 5.1 (Core predicates).

(Δ, \models) is a set of core predicates for Σ if \models is a binary relation connecting states to triples that comprise a core predicate, a list of in-values and a list of out-values:

$$\models \in \Sigma \times (\Delta \times \text{Val list} \times \text{Val list})$$

We write triples $(\delta, \vec{v}_i, \vec{v}_o)$ as $\langle \delta \rangle(\vec{v}_i; \vec{v}_o)$, and say that that σ *satisfies* or *is a model of* $\langle \delta \rangle(\vec{v}_i; \vec{v}_o)$ iff $\sigma \models \langle \delta \rangle(\vec{v}_i; \vec{v}_o)$.

Example. In the linear heap state model, described in §4.5, we can define the points-to assertion $\langle \text{PointsTo} \rangle(a; v)$, pretty-printed $a \mapsto v$, with the relation:

$$\sigma \models a \mapsto v \iff \sigma = [a \mapsto v]$$

In other words, a state fragment satisfies the points-to assertion if it contains a single cell mapping address a to value v .

⁵ The concept of ins and outs is crucial for the matching algorithm, and we explain it in detail when we introduce the algorithm itself in §5.3. Outside of matching, the distinction between ins and outs is irrelevant.

Assertion satisfiability Core predicate satisfiability is then lifted to assertions, using a substitution $\theta \in \text{Subst}$ that maps variables to values.

Definition 5.2 (Assertion satisfiability).

$$\begin{aligned} \theta, \sigma &\models \langle \delta \rangle(\vec{e}_i; \vec{e}_o) \Leftrightarrow \llbracket \vec{e}_i \rrbracket_\theta = \mathbf{0k} \ \vec{v}_i \wedge \llbracket \vec{e}_o \rrbracket_\theta = \mathbf{0k} \ \vec{v}_o \wedge \sigma \models \langle \delta \rangle(\vec{v}_i; \vec{v}_o) \\ \theta, \sigma &\models \exists \mathbf{x}. P \Leftrightarrow \exists v. \theta[\mathbf{x} \leftarrow v], \sigma \models P \\ \theta, \sigma &\models P * Q \Leftrightarrow \exists \sigma_P, \sigma_Q. \theta, \sigma_P \models P \wedge \theta, \sigma_Q \models Q \wedge \sigma = \sigma_P \cdot \sigma_Q \end{aligned}$$

A core predicate assertion $\langle \delta \rangle(\vec{e}_i; \vec{e}_o)$ is satisfied by state fragment σ under substitution θ if all of its arguments successfully evaluate using θ and if σ satisfies the core predicate with the evaluated arguments. For example, a core predicate of the form $\langle \delta \rangle(0; 1/0)$ cannot be satisfied by any state fragment, as $1/0$ cannot be successfully evaluated.⁶

The two remaining cases are standard: $\exists \mathbf{x}. P$ is satisfied by σ under θ if we can extend θ with a value for \mathbf{x} such that σ satisfies P under this extended substitution; and $P * Q$ is satisfied by σ under θ if σ can be split into two disjoint state fragments, σ_P and σ_Q , which respectively satisfy P and Q under θ .

Notice again the parallel between SIGIL and its assertion language. In SIGIL, actions form a variable-free language while the parametric semantics handles variables and control flow. Analogously, here, core predicates form a variable-free assertion language while the parametric satisfiability relation handles variables and composition.

Strict exactness In the context of UX analysis, we require that core predicates be *strictly exact*, meaning that they are satisfied by at most one state fragment.

Definition 5.3 (Strictly exact core predicates).

A core predicate δ is *strictly exact* if:

$$\sigma \models \langle \delta \rangle(\vec{v}_i; \vec{v}_o) \wedge \sigma' \models \langle \delta \rangle(\vec{v}_i; \vec{v}_o) \implies \sigma = \sigma'$$

If all core predicates are strictly exact, then assertions without quantifiers are also strictly exact. As this is the case for any precondition generated by under-approximate bi-abduction, it is not a substantial limitation.

Lemma 5.4 (Strictly exact assertions).

If all core predicates used in a quantifier-free assertion P are strictly exact, then P is strictly exact:

$$\forall \theta, \sigma, \sigma'. \theta, \sigma \models P \wedge \theta, \sigma' \models P \implies \sigma = \sigma'$$

5.2 Producers

Every satisfiability relation on core predicates induces a producer function, which extends a given state fragment with the resources corresponding to a given core predicate.

Definition 5.5 (Core predicate producers).

A set of core predicates, $(\Delta \ni \delta, \models)$, induces a **produce** function, defined axiomatically as:⁷

$$\text{produce } \sigma \ \delta \ \vec{v}_i \ \vec{v}_o = \{ \sigma \cdot \sigma_\delta \mid \sigma_\delta \models \langle \delta \rangle(\vec{v}_i; \vec{v}_o), \sigma \# \sigma_\delta \}$$

⁶ Note that $\langle \delta \rangle(0; 1/0)$ is still a syntactically valid assertion; however, it has no models (i.e. no state fragment satisfies it).

⁷ Observe that there exist two equivalent perspectives: either we start from a satisfiability relation and induce the producer, or provide the producer and induce the satisfiability relation. We choose the former, as it allows us to select a pre-existing satisfiability relation and justify the use of specifications proven outside of the framework (e.g. by hand).

Notation. We write $\sigma.\text{prod}_\delta(\vec{v}_i, \vec{v}_o) \xrightarrow{\mathbb{S}} \sigma'$ for $\sigma' \in \mathbb{S}.\text{produce } \sigma \delta \vec{v}_i \vec{v}_o$, and omit the state model \mathbb{S} when it is clear from the context.

In practice, state models are required to implement the `produce` function, and the satisfiability relation serves only as a meta-theoretical tool to define the semantics of the assertion language. We then lift these producers from core predicates to assertions using the parametric assertion producer, defined in Figure 5.1 below:

```

let rec produce_asrt  $\mathbb{S} \theta \sigma P$ :  $\Sigma$  set =
  match  $P$  with
  |  $\langle \delta \rangle(\vec{e}_i; \vec{e}_o) \rightarrow ($ 
    match  $\llbracket \vec{e}_i \rrbracket_\theta, \llbracket \vec{e}_o \rrbracket_\theta$  with
    |  $\text{Ok } \vec{v}_i, \text{Ok } \vec{v}_o \rightarrow \mathbb{S}.\text{produce } \sigma \delta \vec{v}_i \vec{v}_o$ 
    |  $_ \rightarrow \text{vanish})$ 
  |  $\exists x.Q \rightarrow$ 
    let*  $v = \text{nondet } ()$  in
    produce_asrt  $\mathbb{S} \theta [x \leftarrow v] \sigma Q$ 
  |  $P * Q \rightarrow$ 
    let*  $\sigma' = \text{produce\_asrt } \mathbb{S}.\text{produce } \theta \sigma P$  in
    produce_asrt  $\mathbb{S} \theta \sigma' Q$ 

```

Figure 5.1: Formal definition of the parametric assertion producer.

⁸ In the definition, `let*` corresponds to the `bind` function of the `set` monad, meaning that it handles non-determinism, but not errors:

$$\text{bind } x \ f = \bigcup_{y \in x} f(y)$$

This is sufficient because production cannot throw an error.

The parametric assertion producer delegates the production of core predicates to the state model producer, non-deterministically instantiates existential variables with all possible values, and produces both operands of a separating conjunction in sequence.⁸ Note that the substitution is required to cover all free variables of the assertion (i.e. $\text{fv}(P) \subseteq \text{dom}(\theta)$) for the producer to be well-defined. This is analogous to the semantics of expressions, where the substitution has to cover all variables used in the expression.

Notation. We write $\sigma.\text{prod}_P(\theta) \xrightarrow{\mathbb{S}} \sigma'$ for

$$\sigma' \in \text{produce_asrt } \mathbb{S} \theta \sigma P$$

and omit the state model \mathbb{S} when it is clear from context.

Importantly and straightforwardly, the parametric producer can be proven to extend a given state fragment precisely with the resource corresponding to the provided assertion.

Theorem 5.6 (Correctness of assertion production).

Let Σ be a set of state fragments, Δ an associated set of core predicates, $\mathbb{S}.\text{produce}$ be the producer for Δ , and P be an assertion that only contains core predicates from Δ . Then:

$$\text{produce_asrt } \mathbb{S} \theta \sigma P = \{\sigma \cdot \sigma_P \mid \theta, \sigma_P \models P, \sigma \# \sigma_P\}$$

5.3 Consumers and Matching

Consumers are more complex than producers, as they fulfil two different roles. First, they perform *matching*: the process of *identifying* the fragment of the state and free variables that satisfies the input assertion. Then, they *consume* this state fragment, meaning that it is removed from the output state.

To perform matching without backtracking, Gillian uses an approach based on *parameter modes*⁹: the in-parameters of a core predicate are

⁹ Nguyen et al., “Runtime Checking for Separation Logic”, 2008 [NKC08]

used to *learn* its out-parameters. Before an assertion is consumed, its core predicates are re-ordered into a set of steps. The new ordering of these steps must be such that the variables required to evaluate the in-parameters of a core predicate are known before it is consumed, and sufficient variables are learnt for the next to be consumed.

Analogously to producers, the state model is required to implement a core predicate consumer, while the framework provides a parametric assertion consumer that handles variables and separating conjunction.

5.3.1 Core predicate consumers

A core predicate consumer is a function which, given a state σ , a core predicate δ , and in-parameters \vec{v}_i , returns a *logic outcome* $o \in \mathcal{O}_l^+ = \{\text{Ok}, \text{Miss}, \text{Lfail}\}$, a list of out-parameters \vec{v}_o , and a *frame* state fragment σ' . The outcome is either **Ok** for success, **Miss** when the state fragment does not contain sufficient resources, or **Lfail** for logical failure, described in detail shortly. In case of success, it is guaranteed that the frame state was obtained by removing a state fragment satisfying $\langle \delta \rangle(\vec{v}_i; \vec{v}_o)$ from the input state.

Definition 5.7 (Core predicate consumers).

A core predicate consumer is a function with the following signature

`type` $\mathcal{O}_l^+ = \text{Ok} \mid \text{Miss} \mid \text{LFail}$
`val` $\text{consume} : \Sigma \rightarrow \Delta \rightarrow \text{Val list} \rightarrow (\mathcal{O}_l^+ \times \text{Val list} \times \Sigma)$

It must satisfy the following property, guaranteeing that in case of success, a state fragment satisfying the core predicate has been removed from the state:

$$\begin{aligned} & \sigma.\text{consume}_\delta(\vec{v}_i) \rightarrow \text{Ok} : (\vec{v}_o, \sigma') \\ \implies & \exists \sigma_\delta. \sigma = \sigma' \cdot \sigma_\delta \wedge \sigma_\delta \models \langle \delta \rangle(\vec{v}_i; \vec{v}_o) \end{aligned} \quad (\text{CP Consuming})$$

Consumers may return yet another outcome called *logical failure* in addition to success and missing. A logical failure is either due to a *logical mismatch* or to an incompleteness of the consumer, which we now explain.

Logical mismatch A logical mismatch corresponds to a fundamental incompatibility between the state fragment and the consumed resource. More formally, $\sigma.\text{consume}_\delta(\vec{v}_i) \rightarrow _$ must fail if there is no completion of the state of which a fragment satisfies the core predicate:

$$\forall \vec{v}_o, \forall \sigma' \succeq \sigma. \forall \sigma'' \preceq \sigma'. \sigma'' \not\models \langle \delta \rangle(\vec{v}_i; \vec{v}_o)$$

For instance, take a simplified version of the linear heap, where addresses and values are booleans instead of natural numbers and values. The outcomes of consuming the core predicate¹⁰ $T \mapsto T$ from either of the nine possible state fragments are depicted in Figure 5.2. If the binding $T \mapsto T$ is in the heap, it is removed, and consumption succeeds. If there is no binding for address T , then there exists a state fragment σ_f (namely, $\sigma_f = [T \mapsto T]$) which can be added to the current state fragment σ such that $T \mapsto T$ can be consumed from $\sigma \cdot \sigma_f$. Hence, the corresponding outcome is **Miss**. The last case is the

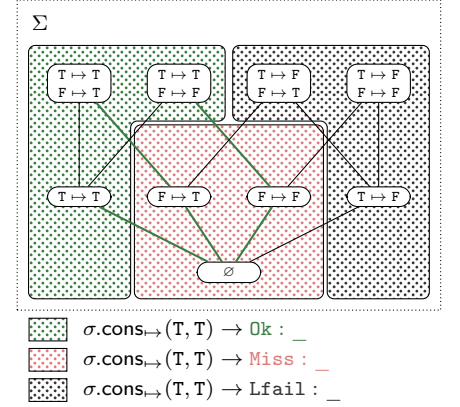


Figure 5.2: Outcomes of consuming the core predicate $T \mapsto T$ in the simplified linear heap with boolean addresses and values, when the consumer is complete and the allocation domain is \perp .

Notation. We write

$$\sigma.\text{consume}_\delta(\vec{v}_i) \xrightarrow{\mathbb{S}} o_l : (v, \sigma')$$

iff

$$\mathbb{S}.\text{consume } \sigma \delta \vec{v}_i = (o_l, v, \sigma')$$

and omit the state model \mathbb{S} when it is clear from the context.

¹⁰ We assume that both address and value are in-parameters for simplicity. Using the core predicate notation, it would be $\langle \text{BPointsTo} \rangle(T, T;)$

“non-fixable” case, where the state contains the binding $T \mapsto F$, which is fundamentally incompatible with the required resource and yields a logical failure.

Completeness Core predicate consumers may fail to consume a resource even when it is present in the state fragment. More formally, consumers may be **incomplete** according to the following definition of completeness:

Definition 5.8 (Completeness of core predicate consumers).

A core predicate consumer is *complete* if it satisfies the following property:

$$\begin{aligned} \sigma_\delta \models \langle \delta \rangle (\vec{v}_i; \vec{v}_o) \wedge \sigma \# \sigma_\delta \\ \implies (\sigma \bullet \sigma_\delta).cons_\delta(\vec{v}_i) \rightarrow Ok : (\vec{v}_o, \sigma) \end{aligned} \quad (\text{CP Completeness})$$

For instance, a consumer that always yields a logical failure is valid according to [Definition 5.7](#), but it is incomplete. In practice, completeness may be limited by the complexity of the state model. Furthermore, since consumers must return a single outcome, core predicates that may have several matches in the heap (called *non-precise predicates* in VeriFast) do not have a complete consumer.

5.3.2 Parametric assertion consumer

The parametric assertion consumer lifts consumption from core predicates to assertions by handling composition. In addition, it performs *matching*: the process of extending the substitution to instantiate the free and existentially quantified variables of the assertion.

The assertion consumer receives a state fragment σ , an assertion P and a partial substitution θ and returns a triple containing a logic outcome, an extended substitution, and a state fragment.

```
val consume_asrt:  $\Sigma \rightarrow Asrt \rightarrow Subst \rightarrow (\mathcal{O}_l^+ \times Subst \times \Sigma)$ 
```

Before presenting the formal definition of the parametric assertion consumer, we introduce *matching plans*, which guide the consumption algorithm.

Matching plans In the consumption algorithm presented in [Figure 5.3](#), core predicates are consumed one by one. Throughout consumption, the substitution must contain enough variables to evaluate all the in-parameters of the core predicate at hand. At each step, the substitution is extended with information *learned* from the outcome of the core predicate consumer.

A matching plan, obtained using a `plan` function of the below signature, is an ordering of the core predicates of an assertion, which allows for the substitution to maintain the required property. Each step of the matching plan indicates *how* to learn new variables from the outcome of the core predicate consumer.

```
val plan : Subst  $\rightarrow Asrt \rightarrow (Asrt \times (Var \times PExpr) \text{ list}) \text{ list result}$ 
```

For example, consider: the assertion $P = y \mapsto 1024 * x \mapsto y + 1$, where $a \mapsto v$ is syntactic sugar for $\langle \text{PointsTo} \rangle(a; v)$; the heap $\sigma = [0 \mapsto$

42, 41 \mapsto 1024]; and substitution $\theta = [x \mapsto 0]$. There is a unique value that can be assigned to y for this assertion to hold, namely 41. The goal of matching is to find this value, and the following matching plan guides this discovery:

$$\text{plan } \theta \ P = \text{Ok} : [(x \mapsto y + 1, [(y, O_1 - 1)]) \\ (y \mapsto 1024, [])]$$

This plan consists of two steps. It indicates that the core predicate $x \mapsto y + 1$ should be consumed first. Doing so is possible since the substitution contains x , meaning it is possible to evaluate the in-parameter of the predicate and call its consumer. The consumer will remove the cell $0 \mapsto 42$ from the heap (since $\theta(x) = 0$) and return the value 42.

Next, the first step indicates that the variable y should be *learned* from $O_1 - 1$, where O_1 is a placeholder variable corresponding to the first (and, here, only) result of the previous consumption. In this example, O_1 takes the value 42, and the substitution is extended with the binding $y \mapsto 41$. Now that the substitution knows y , the second step can be performed, and the assertion is successfully consumed.

Interestingly, matching plans do not have to be “correct” for the consumption algorithm to be sound. The only requirement is that the steps are a permutation of the atoms of the input assertion. For instance, an incorrect matching plan, which assigns O_1 instead of $O_1 - 1$ to y in the first step, would lead to a logical failure but never to an unsound success.

To avoid cluttering an already extensive presentation, we do not provide a detailed formalisation of matching plans and their construction in this manuscript. Should the reader be interested, we refer them to the corresponding paper.¹¹

Note that the plan function is only defined for quantifier-free assertions. We will explain shortly how to handle existential quantifiers.

¹¹ Löw et al., “Matching Plans for Frame Inference in Compositional Reasoning”, 2024 [Löw+24b]

```

let learn  $\theta_o$   $\theta$  (x, e) =
  if  $x \in \text{dom}(\theta)$  then lfail () else
  let*  $v = \llbracket e \rrbracket_{\theta \cup \theta_o}$  in
  ok  $\theta [x \leftarrow v]$ 

let consume_step  $\mathbb{S}$  ( $\theta$ ,  $\sigma$ ) ( $\langle \delta \rangle (\vec{e}_i; \vec{e}_o)$ , learnings) =
  ① let*  $\vec{v}_i = \llbracket \vec{e}_i \rrbracket_{\theta}$  in
  ② let* ( $\vec{v}_o$ ,  $\sigma'$ ) =  $\mathbb{S}.\text{consume } \sigma \ \delta \ \vec{v}_i$  in
  ③ let  $\theta_o = [\vec{O} \mapsto \vec{v}_o]$  in
  ④ let*  $\theta' = \text{fold\_outcome } (\text{learn } \theta_o) \ \theta \ \text{learnings}$  in
  ⑤ let*  $\vec{v}'_o = \llbracket \vec{e}_o \rrbracket_{\theta'}$  in
  ⑥ let* () = assert_all_equal  $\vec{v}_o \ \vec{v}'_o$  in
  ⑦ ok ( $\theta'$ ,  $\sigma'$ )

let consume_asrt  $\mathbb{S}$   $\sigma$   $\theta$   $P$  =
  let* mp = plan  $\theta$   $P$  in
  fold_outcome (consume_step  $\mathbb{S}$ ) ( $\theta$ ,  $\sigma$ ) mp

```

Figure 5.3: Formal description of the parametric assertion consumer.

Formal description Figure 5.3 formally defines the parametric assertion consumer. The `consume_asrt` function calls the `plan` function and iterates over the created matching plan. It applies the `consume_step`

function to each step of the plan, accumulating an extended substitution and a partially consumed state fragment at each step. More formally, if θ_0 and σ_0 are the inputs to the consumption algorithm, the `fold_outcome` operation iteratively applies each step mp_k of the matching plan, obtaining a (θ_k, σ_k) from `consume_step` $\mathbb{S} (\theta_{k-1}, \sigma_{k-1}) \text{mp}_k$. If any step results in a logical failure or a missing outcome, the entire consumption process fails.¹²

The `consume_step` is the main workhorse of the algorithm, and we detail it line by line:

- ① The in-parameters \vec{e}_i of the core predicate are evaluated under the current extension of substitution θ , yielding a set of in-values \vec{v}_i . This requires that the substitution covers all the variables contained in the in-parameters. If evaluation fails, the entire consumption process fails.
- ② The core predicate is consumed using the in-values \vec{v}_i , and yielding a set of out-values \vec{v}_o and a partially consumed state fragment σ' .
- ③ A substitution θ_0 is created, mapping distinct, fresh placeholders to the out-values obtained from the core predicate consumer.
- ④ The substitution is extended: each pair (\mathbf{x}, e) in the `learnings` list is applied to iteratively extend the substitution, using the `learn` function. Given the placeholder substitution θ_0 , the current substitution θ and a pair (\mathbf{x}, e) , a new substitution is obtained by binding \mathbf{x} to the result of evaluating e under the substitution $\theta_0 \cup \theta$. If the variable is already in the substitution or if evaluation fails, the entire consumption process fails.
- ⑤ Using this newly extended substitution, the out-values \vec{e}_o of the core predicate δ to consume are evaluated, yielding a set of out-values \vec{v}'_o .
- ⑥ It is checked that the out-values \vec{v}_o obtained from the core predicate consumer match the values \vec{v}'_o expected by the assertion, according to the learned substitution. This check is crucial for the soundness of the algorithm, ensuring that learning was correctly performed and that uses of the same variables across the assertion are consistent.
- ⑦ If all the previous steps are successful, the extended substitution and the partially consumed state fragment are returned.

Notation. We write

$$\sigma.\text{cons}_P(\theta) \xrightarrow{\mathbb{S}} o : (\theta', \sigma')$$

if `consume_asrt` $\mathbb{S} \sigma P \theta = (o, \theta', \sigma')$, and omit the state model \mathbb{S} when it is clear from the context.

Properties of the assertion consumer The assertion consumer preserves the properties of the core predicate consumers.

Theorem 5.9 (Assertion consumer).

Let $\mathbb{S}.\text{consume}$ be a valid core predicate consumer according to [Definition 5.7](#). Then, `consume_asrt` \mathbb{S} satisfies the following properties:

$$\begin{aligned} \sigma.\text{cons}_P(\theta) \xrightarrow{\mathbb{S}} \text{Ok} : (\theta', \sigma') \\ \implies \exists \sigma_P. \sigma = \sigma' \cdot \sigma_P \wedge \theta', \sigma_P \models P \end{aligned} \quad (\text{Consuming})$$

¹² In other words, the `fold_outcome` function is the `foldM` function in the outcome monad.

$$\begin{aligned} \sigma.\text{cons}_P(\theta) &\xrightarrow{\mathbb{S}} \text{Ok} : (\theta', \sigma') \\ \implies \theta &\subseteq \theta' \end{aligned} \quad (\text{Matching})$$

The **Consuming** property lifts the **CP Consuming** property to the assertion level: if consumption is successful, the removed state fragment σ_P satisfies the consumed assertion P under the extended substitution θ' . This property immediately entails that θ' must cover all free variables of P .

Matching, on the other hand, states that the obtained substitution θ' must be an extension of the input substitution θ . It indicates that the consumer must have found a proper match for the assertion without changing the values of the already known variables.

Existential quantifiers The planning algorithm is only defined for quantifier-free assertions. However, the assertion consumer may still consume assertions with existential quantifiers. This is done by first transforming the assertion into a normal form $\exists \vec{x}. P$, where P is quantifier-free¹³.

Given an assertion in this normal form, the `plan` function creates a matching plan for P . According to the **Consuming** property, in case of success, the consumed fragment σ_P satisfies P under the extended substitution. Therefore, it satisfies $\exists \vec{x}. P$ under the same substitution. Moreover, the extended substitution provides witnesses for all existentially quantified variables.

Completeness The assertion consumer also preserves the completeness of the core predicate consumer.

Theorem 5.10 (Assertion consumer completeness).

Given a complete (according to [Definition 5.8](#)) core predicate consumer `S.consume`, the assertion consumer `consume_asrt S` is also complete.¹⁴ Formally, it satisfies the following property:

$$\begin{aligned} \theta, \sigma_P &\models P \wedge \sigma \# \sigma_P \\ \implies \exists (\sigma \cdot \sigma_P). \text{cons}_P(\theta) &\xrightarrow{\mathbb{S}} \text{Ok} : (\theta, \sigma) \end{aligned} \quad (\text{Consume completeness})$$

5.4 Specification semantics

The primary purpose of consumers and producers is to enable *specification execution*. Instead of executing the body of the called function, the caller function can execute a (valid) specification of the called function. This is a crucial feature of separation logic, often referred to as *function compositionality*.

For the first time, Gillian offers a *unified* engine, allowing for the use of either separation logic for **OX** analysis or incorrectness separation logic for **UX** analysis.

Specifications SL and ISL specifications capture both successful and erroneous executions (but not **Miss** executions) and hence take the form of quadruples, reusing a notation commonly found in the literature¹⁵. Reusing Iris notation, post conditions use an explicit binder **r** to refer

¹³ This is always possible modulo some renaming of existentially quantified variables.

¹⁴ This theorem assumes that the creation a matching plan cannot fail if all the free variables of the assertions are already known, and that, in this case, the list of learnings is always empty.

¹⁵ O'Hearn, "Incorrectness logic", 2019 [OHe19]; Raad et al., "Local Reasoning About the Presence of Bugs: Incorrectness Separation Logic", 2020 [Raa+20]; and Maksimović et al., "Exact Separation Logic: Towards Bridging the Gap Between Verification and Bug-Finding", 2023 [Mak+23]

to the return value of the expression. The semantics of specifications is then canonical.

Definition 5.11 (Specifications: SL/ISL Quadruples).

Given a program γ , an expression e satisfies the separation logic quadruple $\{ P \} e \{ \text{Ok} : r. Q_{\text{Ok}} \} \{ \text{Err} : r. Q_{\text{Err}} \}$ if all executions starting from a state satisfying P either successfully terminate in a state that satisfies Q_{Ok} , erroneously terminate in a state that satisfies Q_{Err} , or diverge:

$$\begin{aligned} \gamma \models \{ P \} e \{ \text{Ok} : r. Q_{\text{Ok}} \} \{ \text{Err} : r. Q_{\text{Err}} \} &\triangleq \forall \sigma, \sigma', \theta, o, v. \\ \theta, \sigma \models P \wedge \gamma \vdash \sigma, e \Downarrow_{\theta} o : (v, \sigma') &\implies \\ (o \neq \text{Miss} \wedge \theta[r \leftarrow v], \sigma' \models Q_o) & \end{aligned}$$

Similarly, expression e satisfies the incorrectness separation logic quadruple $[P] e [\text{Ok} : Q_{\text{Ok}}] [\text{Err} : Q_{\text{Err}}]$ if every state satisfying Q_{Ok} is reachable from a state satisfying P through a successful execution of e , and every state satisfying Q_{Err} is reachable from a state satisfying P through an erroneous execution of e :

$$\begin{aligned} \gamma \models [P] e [\text{Ok} : r. Q_{\text{Ok}}] [\text{Err} : r. Q_{\text{Err}}] &\triangleq \forall \theta, \sigma', o, v. \\ \theta[r \leftarrow v], \sigma' \models Q_o &\implies (\exists \sigma. \theta, \sigma \models P \wedge \gamma \vdash \sigma, e \Downarrow_{\theta} o : (v, \sigma')) \end{aligned}$$

We sometimes use the notation $\langle P \rangle e \langle \text{Ok} : r. Q_{\text{Ok}} \rangle \langle \text{Err} : r. Q_{\text{Err}} \rangle$ to denote a specification independently of its kind. In addition, we use the triple notation $\langle P \rangle e \langle o : Q \rangle$ when the specification describes only a single outcome (i.e. the other outcome has condition **false**) and omit the outcome when it is irrelevant.

Quadruples specify the behaviour of expressions, and we overload the notation to specify the behaviour of functions.

Definition 5.12 (Function specifications).

Given a program γ , a function $f(\vec{x}) \{e\} \in \gamma^{16}$ satisfies the (incorrectness) separation logic quadruple $\langle P \rangle f \langle \text{Ok} : r. Q_{\text{Ok}} \rangle \langle \text{Err} : r. Q_{\text{Err}} \rangle$ if its body satisfies the quadruple:

$$\begin{aligned} \gamma \models \langle P \rangle f(\vec{x}) \langle \text{Ok} : r. Q_{\text{Ok}} \rangle \langle \text{Err} : r. Q_{\text{Err}} \rangle &\triangleq \\ \gamma \models \langle P \rangle e \langle \text{Ok} : r. Q_{\text{Ok}} \rangle \langle \text{Err} : r. Q_{\text{Err}} \rangle & \end{aligned}$$

¹⁶ Remember from the syntax of SIGIL (§4.4) that function definitions require all free variables of the body to be formal arguments.

Specification execution A keystone of our framework is its ability to execute specifications. The algorithm is standard and inspired by VeriFast, Viper and JaVerT 2.0. Figure 5.4 offers a simplified visual representation, ignoring substitutions and outcomes, of executing a specification $\langle P \rangle \langle Q \rangle$. First, the pre-condition P is consumed from the state, leaving the rest of the state, often called the *frame*. Then, the post-condition Q is produced on top of the frame. SL and ISL specifications are executed similarly, though they differ in the guarantees they provide.

The formal definition of the specification execution algorithm is provided in Figure 5.5. Specification execution is always performed in the context of a state model \mathbb{S} , which provides the producer and consumer for the core predicates used in the specification. The return value is non-deterministically initialised to any value and later constrained by

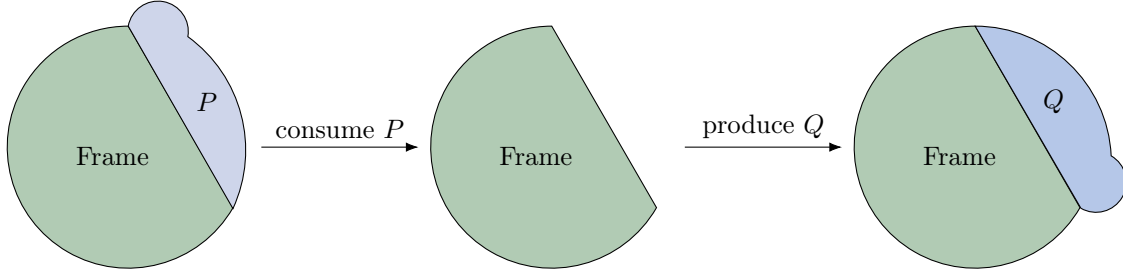


Figure 5.4: Visual representation of the execution of a specification $\langle P \rangle \langle Q \rangle$

the producer; both success and error outcomes are produced. Executing specifications may yield any outcome in $\mathcal{O}_l = \{\text{Ok}, \text{Err}, \text{Miss}, \text{Lfail}\}$, which keep their usual meaning. **Lfail** and **Miss** can occur during the consumption of the pre-condition, while **Ok** and **Err** occur when the specification has been successfully executed.

```

let execute_spec  $\mathbb{S} \ \sigma \ \theta \ S =$ 
  let  $\langle P \rangle \langle \text{Ok} : r. Q_{\text{Ok}} \rangle \langle \text{Err} : r. Q_{\text{Err}} \rangle = S$  in
  let*  $(\theta', \sigma') = \text{consume\_asrt } \mathbb{S}. \text{consume } \theta \ \sigma \ P$  in
  let*  $v = \text{nondet } ()$  in
  let oks =
    let oks_states = produce_asrt  $\mathbb{S}. \text{produce } (\theta' [x \leftarrow v]) \ \sigma' \ Q_{\text{Ok}}$  in
    {  $\text{Ok} : (v, \sigma'') \mid \sigma'' \in \text{oks\_states}$  }
  in
  let errs =
    let err_states = produce_asrt  $\mathbb{S}. \text{produce } (\theta' [x \leftarrow v]) \ \sigma' \ Q_{\text{Err}}$  in
    {  $\text{Err} : (v, \sigma'') \mid \sigma'' \in \text{err\_states}$  }
  in
  oks  $\cup$  errs

```

Figure 5.5: Formal definition of the algorithm used to execute specifications

Notation. We write $\sigma. \text{spec}_S(\theta) \xrightarrow{\mathbb{S}} o_l : (v, \sigma')$ if

$$(o_l : (v, \sigma')) \in \text{execute_spec } \mathbb{S} \ \sigma \ \theta \ S$$

and omit the state model \mathbb{S} when it is clear from the context.

Specification execution of an expression e can always be used instead of executing the expression itself, yielding different guarantees depending on the specification. As expected, SL specifications guarantee no path is dropped, ensuring the absence of false negatives. In contrast, ISL specifications guarantee that no path is added, ensuring the absence of false positives. In both cases, logical failures and missing outcomes are inconclusive: they indicate a failure in the reasoning but guarantee neither the existence nor the absence of a bug. Therefore, during OX analysis, these outcomes should be flagged as potential bugs while ignored and dropped during UX analysis.

Theorem 5.13 (Specification execution: soundness).

Let γ be a program and $S = \{ P \} e \{ \text{Ok} : r. Q_{\text{Ok}} \} \{ \text{Err} : r. Q_{\text{Err}} \}$ be a separation logic quadruple. If S is valid in γ and e can be executed in γ starting from a state σ under a substitution θ , then execution of S in the same state σ and substitution yields at least one result. In

the absence of failures, all paths are preserved:

$$\begin{aligned} \gamma \models S \wedge \gamma \vdash \sigma, e \Downarrow_{\theta} o : (v, \sigma') &\implies \\ (\exists o_S, v_S, \sigma_S. \sigma.\text{spec}_S(\theta) \rightsquigarrow o_S : (v_S, \sigma_S) \\ \wedge (o_S \notin \{\text{Miss}, \text{Lfail}\} \implies (o_S = o \wedge v_S = v \wedge \sigma_S = \sigma'))) & \\ (\text{OX} \bigcirc \text{spec. exec. soundness}) & \end{aligned}$$

If, conversely, $S = [P] e [Ok : r.Q_{Ok}] [Err : r.Q_{Err}]$ is an incorrectness separation logic quadruple, and then all states that are successfully reachable using the specification execution are also reachable using the expression execution:

$$\begin{aligned} \gamma \models S \wedge \sigma.\text{spec}_S(\theta) \rightsquigarrow o : (v, \sigma') \wedge o \notin \{\text{Miss}, \text{Lfail}\} &\implies \\ \gamma \vdash \sigma, e \Downarrow_{\theta} o : (v, \sigma') & \\ (\text{UX} \bigcirc \text{spec. exec. soundness}) & \end{aligned}$$

Note that the guarantees provided by UX specification execution are under the assumption that all core predicates used are strictly exact.

Specification contexts The specification semantics allows for replacing the execution of the callee’s body with the execution of their specifications. While the concrete semantics uses a function context, which registers the implementation of each function, the specification semantics uses a *specification context*, which registers function specifications.

Definition 5.14 (Specification Context).

A specification context $\Gamma \in SCtx = Fid \xrightarrow{fin} Spec$ is a partial finite function that maps function identifiers to specifications (SL or ISL quadruples).

Definition 5.15 (Function Environment).

A function environment (γ, Γ) is a pair comprising an implementation context γ and a specification context Γ . It is m -valid, written $\models^m (\gamma, \Gamma)$ if it contains only specifications of the mode m (i.e. SL if $m = \text{OX}$ and ISL if $m = \text{UX}$), and if every specification in Γ contains an implementation in γ that satisfies it.

Specification semantics We can finally formally provide the specification semantics in Figure 5.6, in the form of an `eval` function that receives an additional Γ parameter.¹⁷ For all cases except the function call, the specification semantics is defined identically to the concrete semantics. In the function-call case, the specification semantics checks if the context includes a specification of the callee function and applies this specification without resorting to the callee’s implementation. If the context does not hold such a specification, the semantics resort to the implementation, as in the concrete semantics.

This approach is in line with Gillian’s implementation. However, it diverges from the approaches adopted by Viper and VeriFast, which restrict execution strictly to specification calls and prohibit defaulting to the execution of the function body.

Notation. We write $\gamma, \Gamma \vdash \sigma, e \Downarrow_{\theta}^{\mathbb{S}} o_l : (v, \sigma)$ if

$$(o_l, v, \sigma) \in \text{eval } \mathbb{S}.\text{eval_action } \gamma \ \Gamma \ \theta \ \sigma \ e$$

and omit the state model \mathbb{S} when it is clear from context.

¹⁷ This new `eval` function behaves exactly like the “non-specification” semantics if given an empty specification context $\Gamma = \emptyset$. Therefore, we do not give it a new name, considering it an extension of `eval`.

```

let rec eval eval_action  $\gamma$   $\Gamma$   $\theta$   $\sigma$   $e$  =
  match  $e$  with
  |  $f(\vec{e}) \rightarrow$ 
    let*  $(\vec{v}, \sigma') = \text{eval\_all } \theta \ \sigma \ \vec{e}$  in
    let*  $f(\vec{x}) \{e\} = \gamma[f]$  in
    let  $\theta_f = [\vec{x} \rightarrow \vec{v}]$  in
    match  $\Gamma(f)$  with
    | Some spec  $\rightarrow$ 
      execute_spec  $\sigma' \ \theta_f \ \text{spec}$ 
    | None  $\rightarrow$ 
      eval  $\theta_f \ \sigma' \ e$ 
  | ...

```

Figure 5.6: Formal definition of the specification semantics

Soundness If the specification context Γ is populated with valid SL specifications, the specification semantics can be soundly used for verification. Similarly, if it is populated with ISL specifications, it can be soundly used for true-bug finding.

Theorem 5.16 (Specification semantics: soundness).

If $\models^m (\gamma, \Gamma)$, then one of the following two properties hold, depending on the mode of execution m :

$$\begin{aligned}
 \models^{\text{OX}} (\gamma, \Gamma) \wedge \gamma \vdash \sigma, e \Downarrow_{\theta} o : (v, \sigma') &\implies (\exists o', \sigma'', v'. \\
 &\gamma, \Gamma \vdash \sigma, e \Downarrow_{\theta} o : (\sigma', v') \\
 &\wedge (o' \notin \{\text{Miss}, \text{Lfail}\} \Rightarrow (o' = o \wedge \sigma'' = \sigma' \wedge v' = v))) \\
 &\text{(Spec Sem. } \text{OX} \text{ soundness)}
 \end{aligned}$$

$$\begin{aligned}
 \models^{\text{UX}} (\gamma, \Gamma) \wedge \gamma, \Gamma \vdash \sigma, e \Downarrow_{\theta} o : (v, \sigma') \wedge o \notin \{\text{Miss}, \text{Lfail}\} \\
 \implies \gamma \vdash \sigma, e \Downarrow_{\theta} o : (v', \sigma') \\
 \text{(Spec Sem. } \text{UX} \text{ soundness)}
 \end{aligned}$$

Again, notice that the guarantees provided by the specification semantics only apply for outcomes $o \in \{\text{Ok}, \text{Err}\}$. Logical failures and missing outcomes correspond to reasoning for errors and do not provide any guarantees: using SL specifications, they are inconclusive and should be flagged as a potential bug. Similarly, they do not prove the presence of a bug when using ISL specifications and should be ignored.

Moreover, missing outcomes obtained while using specifications do not guarantee a missing outcome in the compositional semantics since functions may be over-specified, i.e. the pre-condition might require more resources than needed to execute the body. See, for example, Figure 5.7, where function f is over-specified and would yield a missing outcome when executed without the cell at address $x + 1$, even though this cell is not required for its execution.

```

{ x ↦ 1 * (x + 1) ↦ 0 }
f(x) { store(x, 0) }
{ x ↦ 0 * (x + 1) ↦ 0 }

```

Figure 5.7: An example of over-specification

5.5 Examples

Remember the pure state model and the linear heap presented in §4.5. For both state models, we define core predicates with their consumers and producers.

5.5.1 Pure state model: logic

A simple core predicate The pure state model has a unique state 0, and 4 actions: **skip**, **nondet**, **assume** and **assert**. We define the **Pure**

core predicate, with a unique in-parameter and no out-parameters such that $\langle \text{Pure} \rangle(b;)$ holds if and only if b is true:

$$\sigma \models \langle \text{Pure} \rangle(b;) \iff b = \text{true}$$

Its consumer and producer are defined as follows:

```

module Pure = struct
  type  $\Sigma$  = unit
  let  $\emptyset$  = ()

  (* See §4.5.1 for actions *)
  (* ... *)

  type  $\Delta$  = Pure

  let consume  $\sigma$  Pure [ b ] =
    if b then ok ([],  $\sigma$ )
    else lfail ([],  $\sigma$ )

  let produce  $\sigma$  Pure [ b ] =
    if b then ok  $\sigma$ 
    else vanish
end

```

The producer is defined in the same way as the **assume** action, confirming the intuition that producers are a generalisation of **assume** to spatial resources. Similarly, the consumer is a *spatial assert*, though it yields a logical failure instead of an error when failing.

It can be trivially shown that the producer and consumer defined above satisfy definitions [Definitions 5.5](#) and [5.7](#). Moreover, all of these core predicates are strictly exact, as they are only satisfied by the state 0, making them suitable for UX analysis.

Advanced core predicates Using the above-defined **Pure** core predicate with the planning algorithm can be limiting. Take, for example, the following assertion:

$$\{ \exists z. \langle \text{Pure} \rangle(x + y = z;) * \langle \text{Pure} \rangle(z \neq 0;) \}$$

Consider the case where x and y are in the domain of the initial substitution θ , but z is not. Since **Pure** does not have any out-parameter, there is no way to learn the value of z , and it is impossible to create a matching plan.

However, $x + y$ is the only value of z that satisfies this assertion. Ideally, we would like the consumption algorithm to be able to infer this value. To do so, another core predicate can be used:

$$\langle _ + _ = _ \rangle(v_0, v_1; v_2), \text{ denoted } v_0 + v_1 = v_2$$

such that $0 \models v_0 + v_1 = v_2 \iff v_0, v_1, v_2 \in \mathbb{N} \wedge v_0 + v_1 = v_2$

```

let produce_plus_equal  $\sigma$   $n_0$   $n_1$   $n_2$  = let consume_plus_equal  $\sigma$   $n_0$   $n_1$  =
  if  $n_0 + n_1 = n_2$  then ok  $\sigma$  ok  $n_0 + n_1$ 
  else vanish

```

Using this predicate, we can define the following equivalent but plannable assertion:

$$\{ \exists z. \langle _ + _ = _ \rangle(x, y; z) * \langle \text{Pure} \rangle(z \neq 0;) \}$$

Following the same approach, it is possible to inductively define a family of core predicates that directly match the syntax of simple expressions and leverage algebraic rules to create arbitrarily complex plannable pure core predicates.

The implementation of Gillian In the current implementation of Gillian, pure assertions are part of the core assertion language and need not be implemented as core predicates. Users may write arbitrary expressions, and Gillian implements a set of heuristics to derive in- and out-parameters from each pure assertion while it builds matching plans.

To illustrate, let us reuse the example of the assertion $x + y = z$. When building a matching plan, if $\{x, y\} \subseteq \text{dom}(\theta)$, Gillian will infer that the ins are x and y and that the out is z . On the other hand, if $\{x, z\} \subseteq \text{dom}(\theta)$, it will infer that x and z are ins and y is an out, learning $y = z - x$.

This approach allows for great flexibility and alleviates the burden of declaring the ins and outs of each pure assertion without losing expressivity.

5.5.2 Linear heap: logic

The linear heap, initially presented in §4.5.2, comes with two core predicates:

- the $\langle \text{PointsTo} \rangle(a; v)$ core predicate, also denoted $a \mapsto v$, describes the heap fragment with a single cell at address a , which contains value v ; and
- the $\langle \text{Freed} \rangle(a;)$ core predicates, also denoted $a \mapsto \emptyset$, describes the heap fragment with a single freed cell at address a .

The corresponding satisfiability relation is described below:

$$\begin{aligned} \sigma \models a \mapsto v &\iff \sigma = [a \mapsto v] \\ \sigma \models a \mapsto \emptyset &\iff \sigma = [a \mapsto \emptyset] \end{aligned}$$

For conciseness, we only present the producer and consumer for the points-to assertion, the freed being analogous.

Producer Points-to assertions denote exclusive ownership of a given heap address. Recall that an address a is considered owned if it is already in the domain of the heap, $a \in \text{dom}(\sigma)$. Producing another cell at address a should vanish since no address can be owned twice. In the case where a is not already in the heap, it can be safely added:

```
let produce_cell  $\sigma$   $a$   $v$  =
  match  $\sigma(a)$  with
  | Some  $v \rightarrow$  vanish
  | Some  $\emptyset \rightarrow$  vanish
  | None  $\rightarrow \sigma[a \leftarrow v]$ 
```

Consumer The `PointsTo` core predicate has one in-parameter, the address, and one out-parameter, the contained value. Therefore, its consumer receives only an address and must return the value contained in the consumed cell in case of success. The consumer behaves the same as the load function provided in §4.5.2, except that:

- when the address is owned and not freed, the cell is removed from the heap; and
- when the address is freed or not allocated yet, it yields a logical failure instead of an error due to the mismatch between the required resource and the content of the heap.

```

let consume_cell  $\sigma$   $a$  =
  match  $\sigma(a)$  with
  | Some  $v$ ,  $-$   $\rightarrow$  ok ( $v$ ,  $\sigma \setminus a$ )
  | Some  $\emptyset$ ,  $-$   $\rightarrow$  lfail (UseAfterFree,  $\sigma$ )
  | None  $\rightarrow$  lfail (UseAfterFree,  $\sigma$ )

```

Assertions and state representation At this point, the reader may be wondering why Gillian separates assertion and state representation. In the case of the linear heap, both happen to have almost the same representation (modulo the fact that the state representation is map, and can be looked up more efficiently than an assertion). In [Part II](#), we show that this separation is crucial for the implementation of more complex state models. In Gillian-C, core predicates act as a simple *view* of the state, which can be described by the user through assertions, while the representation of the state is more low-level and enables more automations without requiring knowledge from the user.

Chapter 6

Symbolic execution

In the previous chapters, we introduced compositional execution and a parametric separation logic with an alternative concrete semantics in which function calls can be substituted with specification execution. However, both semantics are **non-deterministic** and often yield a large or infinite number of branches, making their outcomes impossible to search exhaustively.

Symbolic execution addresses this issue by abstracting values using *symbolic values*, i.e. values that depend on *symbolic variables*. Throughout execution, a symbolic interpreter uses an SMT solver¹ to prune out infeasible paths. To illustrate, let us consider a simple example.

Example 6.1. Take the piece of C code provided in Figure 6.1, which computes the absolute value of an integer². Our initial state, at the top of Figure 6.2, has address x pointing to a *symbolic value* \bar{y} (symbolicness are denoted with an overline). The execution also carries an expression called *path condition*, which constrains the value of symbolic variables. It is initially **true**, indicating that the symbolic value \bar{y} is unconstrained and can be any integer.

The symbolic engine starts by executing the if-statement, deciding whether $*x < 0$ evaluates to **true**. Because the symbolic value \bar{y} is unconstrained, it is both feasible that $\bar{y} < 0$ and $\neg(\bar{y} < 0)$. The symbolic engine queries an SMT solver, which answers that both cases are *satisfiable*, which causes the execution to *branch*. Each new branch carries its associated condition, adequately constraining \bar{y} .

When the **assert** is executed, the engine asks the SMT solver whether any interpretation of variable \bar{y} exists such that the assertion does not evaluate to **true**. Here, the SMT solver answers **UNSAT**, meaning that the assertion holds for all interpretations of \bar{y} that satisfy the path condition, guaranteeing the absence of any input that would make the assertion fail. Because the assertion cannot fail, the correctness of the program is verified.

Functional formalism This chapter introduces a somewhat abstract formalisation of symbolic execution involving a symbolic execution monad. The advantages of this approach are twofold. First, it presents a single notion of symbolic process that we can use to formalise symbolic semantics, actions, consumers, and producers. Second, it allows for the definition of syntactic sugar that seamlessly replaces the concrete execution monad with the symbolic execution monad. For instance, the side-by-side snippets provided below show the concrete and symbolic versions of pure expression evaluation related to the specific case of the division operator. Our formalism provides soundness preservation results that guarantee that the symbolic version is sound with respect

¹ De Moura et al., “Z3: an efficient SMT solver”, 2008 [DB08]; and Barbosa et al., “cvc5: A Versatile and Industrial-Strength SMT Solver”, 2022 [Bar+22]

```
if (*x < 0) {
    *x = -(*x);
}
assert(*x >= 0);
```

Figure 6.1: Absolute value computation in a C-like language

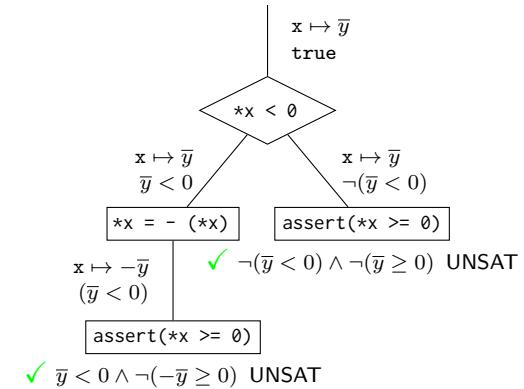


Figure 6.2: A simple symbolic execution tree

² Using machine integers, this implementation is, in fact, unsound since the absolute value of the minimum negative value of a machine integer is not representable using the same type [Ope20]. This example assumes the use of unbounded integers.

to the concrete one when using such a transformation.

<pre> let $\llbracket e_p \rrbracket_\theta =$ match e_p with $e_p \oplus e'_p \rightarrow$ let* $v = \llbracket e_p \rrbracket_\theta$ in let* $v' = \llbracket e'_p \rrbracket_\theta$ in $v \oplus v'$... let $v \oplus v' =$ match \oplus with $/ \rightarrow$ if $v' = 0$ then error DivByZero else ok v / v' ... </pre>	<pre> let $\llbracket e_p \rrbracket_{\bar{\theta}} =$ match e_p with $e_p \oplus e'_p \rightarrow$ let* $\bar{v} = \llbracket e_p \rrbracket_{\bar{\theta}}$ in let* $\bar{v}' = \llbracket e'_p \rrbracket_{\bar{\theta}}$ in $\bar{v} \oplus \bar{v}'$... let $\bar{v} \oplus \bar{v}' =$ match \oplus with $/ \rightarrow$ if%sat $\bar{v}' = 0$ then error DivByZero else ok \bar{v} / \bar{v}' ... </pre>
--	--

In the above, the `let*` operator in the left-hand snippet corresponds to the concrete execution monad presented in §4.4, while the right-hand snippet uses the symbolic execution monad presented in this chapter. The `if%sat` construct is syntactic sugar for the symbolic conditional branching operator, also introduced in this chapter and proven to be sound with respect to the concrete `if` construct. Here, for instance, both the error and the success branches may be explored if its both satisfiable that $\bar{v}' = 0$ and that $\bar{v}' \neq 0$.

Overview The first four sections of this chapter present the essential components of symbolic execution: symbolic *things* (variables, values, ...) in §6.1, path conditions in §6.2, symbolic abstractions in §6.3, and approximate solvers in §6.4. Next, §6.5 introduces *composable* symbolic execution processes. Then, §6.6 presents notes on implementing and optimising symbolic processes and alternative implementations that are more efficient and can seamlessly replace the formal definition. Finally, in §6.7, we apply all this formalism to the parametric symbolic semantics of SIGIL before presenting the symbolic versions of our example state models in §6.8.

6.1 The symbolic realm

Symbolic execution is, at the core, the idea of abstracting values using *symbolic variables*. For instance, in [example 6.1](#), the initial input at address x is abstracted to a variable \bar{y} . The core idea behind the formalisation of symbolic execution is that each variable can be *interpreted* to concrete values. Bugs are detected when an interpretation of the symbolic variables exists for which the concrete execution would lead to an error. Symbolic variables are formalised to an identifier for an unresolved sorted value, where a sort $\tau \in \mathbb{T} = \mathcal{P}(\text{Val}) \setminus \{\emptyset\}$ is a non-empty subset of the values.

Definition 6.1 (Symbolic Variable).

A symbolic variable $\bar{x}, \bar{y} \in \bar{\mathcal{X}}$ is an identifier associated with a sort $\tau \in \mathbb{T}$. We write $\bar{x} : \tau$ if \bar{x} is of the sort τ .

Definition 6.2 (Symbolic Interpretations).

A symbolic interpretation $\varepsilon \in \mathcal{I} = \bar{\mathcal{X}} \xrightarrow{fin} \text{Val}$ is a finite partial function

that maps symbolic variables to a value of the appropriate sort. In other words, it must satisfy

$$\bar{x} : \tau \wedge \bar{x} \in \text{dom}(\varepsilon) \implies \varepsilon(\bar{x}) \in \tau \quad (\text{Interpretation validity})$$

Definition 6.3 (Interpretation extension).

Given two interpretations $\varepsilon_1, \varepsilon_2 \in \mathcal{I}$, we say ε_2 is an *extension* of ε_1 , denoted $\varepsilon_2 \geq \varepsilon_1$, if

$$\varepsilon_1(\bar{x}) = v \implies \varepsilon_2(\bar{x}) = v$$

Extension is a partial order on \mathcal{I} .

In [example 6.1](#), when the value \bar{y} at address \mathbf{x} is negative, it is updated to its negation $-\bar{y}$. This new value is not simply a symbolic variable, but a *symbolic expression*, or *symbolic value* $-\bar{y}$. Given an interpretation ε such that $\varepsilon(\bar{y}) = y$, then the symbolic value $-\bar{y}$ can immediately be interpreted to $-y$. As such, a symbolic value is an object which can be projected to a value given an interpretation of the symbolic variables it contains.

Here, we take a somewhat abstract and categorical approach to formalising these objects. We define the *symbolic world* as the image of the concrete world through the **Sym** functor. Given a set of objects $A \ni a$, this functor provides a set of symbolic objects $\mathbf{Sym}(A) = \bar{A} \ni \bar{a}$. This construction yields our desired definition of a symbolic object \bar{a} , projected to a concrete object a when given an interpretation of all symbolic variables. The set of symbolic values is then straightforwardly defined as the image of the set of values through **Sym**, $\bar{Val} = \mathbf{Sym}(Val)$.

Definition 6.4 (**Sym** functor).

We define the symbolic functor $\mathbf{Sym} = \mathbf{PHom}(\mathcal{I}, -)$ as the covariant **partial-hom-functor** which maps each set X to the set of partial functions $\mathcal{I} \rightarrow X$.

The functor structure provides the ability to project any set into the symbolic realm, and it has other valuable properties.

First, all concrete values can also be considered symbolic. For example, the functor naturally associates the value **true** with the constant symbolic value $\bar{\mathbf{true}} = \lambda\varepsilon. \mathbf{true}$, a total function which does not depend on the interpretation. It can also lift any symbolic variable of sort τ to a symbolic value in $\bar{\tau}$: $\bar{x} = \lambda\varepsilon. \varepsilon(\bar{x})$. Note that we overload the notation \bar{x} to mean both the symbolic variable and the corresponding symbolic value; and that the symbolic value is a partial function since the interpretation of \bar{x} may not be defined.

Furthermore, the **Sym** functor can lift any operation in the concrete world to its symbolic counterpart for free. For example, it lifts the conjunction and negation operators on booleans to operators on symbolic booleans³:

$$\bar{\wedge} : \left\{ \begin{array}{l} \bar{\mathbb{B}} \rightarrow \bar{\mathbb{B}} \rightarrow \bar{\mathbb{B}} \\ \bar{b}_1, \bar{b}_2 \mapsto (\lambda\varepsilon. \bar{b}_1(\varepsilon) \wedge \bar{b}_2(\varepsilon)) \end{array} \right. \quad \bar{\neg} : \left\{ \begin{array}{l} \bar{\mathbb{B}} \rightarrow \bar{\mathbb{B}} \\ \bar{b} \mapsto (\lambda\varepsilon. \neg \bar{b}(\varepsilon)) \end{array} \right.$$

Notation. When unambiguous, we omit the functorial notation and, for example, talk about the symbolic value $\bar{x} + 4$.

For a given set I , the covariant hom-functor $\mathbf{Hom}(I, -) : \mathbf{Set} \rightarrow \mathbf{Set}$ is defined as follows:

For two sets A, B and a function $f : A \rightarrow B$:

$$\begin{aligned} \mathbf{Hom}(I, A) &= I \rightarrow A \\ \mathbf{Hom}(I, f) &: \left\{ \begin{array}{l} \mathbf{Hom}(I, A) \rightarrow \mathbf{Hom}(I, B) \\ g \mapsto f \circ g \end{array} \right. \end{aligned}$$

The partial-hom functor extends this definition to partial function. Formally, it can be constructed by composing the **Option** functor and the covariant hom-functor for I .

³ In these definitions, if any operand is undefined, the result is also undefined. For instance, in empty interpretation, i.e. the interpretation where no symbolic variable is bound, the interpretation of $x \wedge \mathbf{false}$ is undefined.

Partial operator Partial operators become particularly awkward, and cannot be straightforwardly lifted using the **Sym** functor. Consider, for example, the interpretation of the symbolic value $1/\bar{x}$ when $\varepsilon(\bar{x}) = 0$. Deciding that the result is undefined would have subtle implications on the soundness of the symbolic execution, mainly because of the SMT solver’s behaviour when encountering such a value.

In the SMT-LIB standard⁴, all functions must be *total*, and partial functions are extended to total functions by asserting that undefined cases are *under-specified* and may be assigned any value of the right sort. For instance, SMT solvers consider the assertion $a/0 = b$ satisfiable for any value a and b . Therefore, we cannot simply say that $1/\bar{x}$ is undefined for any interpretation, but rather we should account for the fact that it could evaluate to any value.

We do not provide a further formalisation of this behaviour but demonstrate that it is compatible with our formalisation. For instance, the behaviour of division by zero could be formally specified using a special symbolic variable $\overline{\text{ODIV}} : \mathbb{Z}$, and by defining symbolic division as:

$$\overline{\cdot} : \begin{cases} \overline{\mathbb{Z}} \rightarrow \overline{\mathbb{Z}} \rightarrow \overline{\mathbb{Z}} \\ \overline{n_1}, \overline{n_2} \mapsto \lambda \varepsilon. \begin{cases} \varepsilon(\overline{\text{ODIV}}) & \text{if } \overline{n_2}(\varepsilon) = 0 \\ n_1/n_2 & \text{if } \overline{n_1}(\varepsilon) = n_1 \wedge \overline{n_2}(\varepsilon) = n_2 \neq 0 \\ \text{undefined} & \text{if } \varepsilon \notin \text{dom}(\overline{n_1}) \vee \varepsilon \notin \text{dom}(\overline{n_2}) \end{cases} \end{cases}$$

Notation. Given our above handling of partial operators, the only case where the interpretation of a symbolic value for $\overline{v}(\varepsilon)$ is undefined is when \overline{v} makes use of symbolic variables are not defined in ε . We say that ε is *sufficient* for \overline{v} if $\overline{v}(\varepsilon)$ is defined.

6.2 Path conditions

Remember from [example 6.1](#) that a path condition is a logical formula used for constraining the values of symbolic variables and is typically obtained by accumulating the conditions that guard branching expressions. For instance, the path condition $\overline{y} < 0$ restricts the set of interpretations to those for which the interpretation of \overline{y} is less than 0. In other words, the interpretation of the symbolic value $\overline{y} < 0$ must be true. Hence, a path condition is simply a symbolic boolean.

Definition 6.5 (Path conditions).

A path condition $\pi \in \Pi = \overline{\mathbb{B}}$ is a symbolic Boolean.

Since symbolic booleans are defined as $\overline{\mathbb{B}} = \mathcal{I} \rightarrow \mathbb{B}$, this further confirms the intuition that each path condition corresponds to a set of symbolic interpretations (i.e. the set of interpretations such that $\pi(\varepsilon) = \text{true}$).

Furthermore, given two symbolic booleans π_1 and π_2 respectively corresponding to the sets of interpretations \mathcal{I}_1 and \mathcal{I}_2 , the conjunction $\pi_1 \wedge \pi_2$ corresponds to the set of interpretations $\mathcal{I}_1 \cap \mathcal{I}_2$. Therefore, accumulating the path condition using the conjunction monotonically restricts the set of corresponding interpretations.

⁴ Barrett et al., *The Satisfiability Modulo Theories Library (SMT-LIB)*, 2016 [BFT16]

During symbolic execution, an SMT solver determines the satisfiability of the path condition of a branch.

Definition 6.6 (Path condition satisfiability).

A path condition is said to be *satisfiable* if an interpretation exists in which it is true.

$$\text{SAT}(\pi) \iff \exists \varepsilon. \pi(\varepsilon) = \text{true}$$

In addition, to check assert statements, it is necessary to understand if an expression *must* hold under the current path condition. This operation is called an entailment check and relies on the notion of *validity*.

Definition 6.7 (Path condition validity).

A path condition is said to be *valid* if true in all sufficient interpretations.

$$\text{VALID}(\pi) \iff \forall \varepsilon. \pi(\varepsilon) \text{ is undefined} \vee \pi(\varepsilon) = \text{true}$$

Validity checks can be formulated as satisfiability checks as $\text{VALID}(\pi) \iff \text{UNSAT}(\neg\pi)$.

Definition 6.8 (Path condition entailment).

We say that a path condition π_1 entails another path condition π_2 , denoted $\pi_1 \models \pi_2$ if the latter is true in all interpretations in which the former is true. Formally,

$$\begin{aligned} \pi_1 \models \pi_2 &\iff \forall \varepsilon. \pi_1(\varepsilon) = \text{true} \implies \pi_2(\varepsilon) = \text{true} \\ &\iff \text{VALID}(\pi_1) \implies \pi_2 \\ &\iff \text{UNSAT}(\pi_1 \wedge \neg\pi_2) \end{aligned}$$

Note that the above shows how entailment checks can be decided using an SMT solver.

6.3 Symbolic abstractions

Later in this presentation, we define symbolic state models, used as a parameter for the symbolic SIGIL semantics. To define soundness, we need symbolic states to abstract over the set of concrete states Σ . The naive approach would require symbolic states to be elements of $\text{Sym}(\Sigma)$. Unfortunately, this would be too restrictive.

Example 6.2. To illustrate, take the linear heap model $\sigma \in \Sigma = \mathbb{N} \xrightarrow{\text{fin}} \text{Val}$. A common way to define symbolic heaps⁵ is by abstracting each component of the map to their symbolic counterparts, $\bar{\sigma} \in \bar{\Sigma} = \bar{\mathbb{N}} \xrightarrow{\text{fin}} \bar{\text{Val}}$, and each binding of the map is interpreted independently. For instance, using the following interpretation ε , the symbolic heap $\bar{\sigma}$ would be interpreted to the concrete heap σ :

$$\begin{aligned} \varepsilon &= [\bar{x} \mapsto 0, \bar{y} \mapsto 42, \dots] \\ \bar{\sigma} &= [\bar{x} \mapsto \bar{y}, \bar{y} \mapsto \bar{x}] \\ \sigma &= [\varepsilon(\bar{x}) \mapsto \varepsilon(\bar{y}), \varepsilon(\bar{y}) \mapsto \varepsilon(\bar{x})] \\ &= [0 \mapsto 42, 42 \mapsto 0] \end{aligned}$$

However, using the interpretation $\varepsilon' = [\bar{x} \mapsto 0, \bar{y} \mapsto 0, \dots]$, the obtained concrete heap contains the same binding twice and is therefore ill-defined. Instead, we would like to say that $\bar{\sigma}$ yields *no model* under ε' , while it yields *a single model* under ε .

⁵ We provide more details about this choice in §6.8.2

Furthermore, it is possible to define symbolic states with *several models* under a single interpretation.

For these reasons, we define *symbolic abstractions*, which are symbolic objects that yield a *set* of concrete objects given an interpretation. The interpretation of a symbolic heap becomes either a singleton when there are no conflicts in the bindings or the empty set when it would otherwise be ill-defined.

Definition 6.9 (Symbolic abstraction).

Given a set A , its set of symbolic abstractions $\text{Abs}(A) = \mathcal{I} \rightarrow \mathcal{P}(A)$ ⁶.

Notation. When ambiguous, we write symbolic abstractions using both an overline and an asterisk: $\bar{a}^* \in \bar{A}^* = \text{Abs}(A)$. However, when unambiguous, we only use the overline and, for example, denote symbolic states as $\bar{\sigma} \in \bar{\Sigma}$. Furthermore, we write $\varepsilon, a \models \bar{a}^*$ when $a \in \bar{a}^*(\varepsilon)$

We often consider more “concrete” structures than elements of $\text{Abs}(A)$. For instance, the structure of the symbolic linear heap defined above is not strictly an element of $\text{Abs}(A)$, but a partial finite map from $\bar{\mathbb{N}}$ to $\bar{\text{Val}}$. However, by defining a modelling relation $\varepsilon, \sigma \models \bar{\sigma}$, we can *view* the symbolic linear heap a symbolic abstraction, where $\bar{\sigma}(\varepsilon)$ is the set of models σ .

6.4 Approximate solvers

We explained above that SMT solvers are used to decide the satisfiability of a path condition throughout execution. However, SMT solving is generally undecidable, and even when a query is decidable, it may exhaust the time or memory allocated. As such, solvers are *imperfect* and, according to the SMT-LIB specification, may return **SAT**, **UNSAT**, or **UNKNOWN**, indicating that a conclusion could not be reached. The latter is awkward to handle in meta-theory, and other formalisations of symbolic execution usually⁷ assume an ideal solver that never returns **UNKNOWN**.

Another imperfection may arise from the encoding of path conditions into SMT queries. It is common practice for verification tools to under-constrain operations when crafting queries. For example, Viper users may define custom theories and usually declare only the subset of axioms sufficient for verification to succeed. Similarly, Gillian’s encoding of sequences was under-specified until recently. When queries are under-constrained, the solver may answer **SAT** when the right answer is, in fact, **UNSAT**. Effectively, such an encoding leads to the over-approximation of the set of symbolic interpretations corresponding to a path condition. Therefore, while under-constrained encoding should be allowed in **OX** mode, it should be forbidden in **UX** mode. Conversely, solvers used during **UX** analysis should allow over-constraining queries.

Example 6.3 (Under-constrained SMT encoding). In Viper, users have the option to define their own *domains*, which are effectively custom theories that extend the SMT solver encoding. For instance, a user may define a domain for a custom encoding of lists, but only declare the axioms necessary for the verification task at hand.

⁶ The abstraction functor is obtained by composing the covariant power set functor with the symbolic functor

⁷ Featherweight VeriFast [JVP15] does not make this assumption. They define a notion of “sound solver” similar to our notion of OX-approximate solver. We extend the definition to also capture under-approximation.

While we account for solver *imperfection*, our meta-theory cannot account for solver *unsoundness*. Accounting for the latter is impossible without potentially rejecting the entirety of our results. Hence, we must assume that SMT solvers are sound.

```

domain list {
  function Nil(): list
  function Cons(head: T, tail: list): list
  function length(l: list): Int
  axiom { length(Nil()) == 0 }
  axiom { forall l: list :: { length(l) >= 0 } }
}

```

Figure 6.3: A custom under-constrained (over-approximating) domain for lists in Viper

The code in Figure 6.3 encodes lists in Viper through its two constructors, `Nil` and `Cons`. It also axiomatises a `length` function, but only declares two axioms: the length of the empty list is 0, and the length of any list is non-negative. Any verification task performed using this domain as a model of mathematical lists would be sound, and would guarantee the absence of false negatives. However, the encoding is under-constrained as, for instance, `length` could be defined as a constant function that always return 0, and would still satisfy the axioms.

Using this encoding yields a list solver that is strictly OX, as per our definition of *approximate solvers*.

Definition 6.10 (Approximate solver).

An approximate solver with mode $m \in \{\text{OX}, \text{UX}\}$ is a function

$$\text{SAT}_m : \Pi \rightarrow \mathbb{B}$$

with the following property, depending on the mode:

$$\text{SAT}(\pi) \implies \text{SAT}_{\text{OX}}(\pi) \quad (\text{OX} \circ \text{SAT Validity})$$

$$\text{SAT}_{\text{UX}}(\pi) \implies \text{SAT}(\pi) \quad (\circ \text{UX} \text{ SAT Validity})$$

An OX-approximate solver must declare at least all truly satisfiable path conditions as satisfiable. It must never declare unsatisfiable a path condition that has models. However, no constraint prevents such a solver from declaring an unsatisfiable path as satisfiable. Therefore, while an OX-approximate solver may erroneously deem infeasible bugs feasible, leading to false positives, it ensures no true bug is overlooked.

Conversely, for a UX-approximate solver to declare a path condition satisfiable, it must be genuinely satisfiable. In other words, it must never mistakenly label an infeasible path condition as feasible. Hence, while a UX-approximate solver may miss true bugs, it will never raise false positives.

Three theoretical constructs serve as reference points: the trivial SAT_{OX} solver always returns SAT , thus significantly over-approximating; the trivial SAT_{UX} solver always yields UNSAT ; and the ideal solver SAT_{EX} invariably provides the exact answer⁸, but is not computable. Throughout this presentation, the reader is encouraged to use these solvers as conceptual benchmarks to understand the implications of using approximate solvers.

In practice, a solver such as Z3 or CVC5 can be used, and the interpretation of the UNKNOWN answer is modified to obtain an approximate solver. In $\text{OX} \circ$ mode, it becomes SAT , and in $\circ \text{UX}$ mode, it becomes UNSAT :

⁸ $\text{SAT}_{\text{EX}}(\pi) \iff \text{SAT}(\pi)$


```

let solver m query =
  match Z3.solve query with
  | SAT → SAT
  | UNSAT → UNSAT
  | UNKNOWN → match m with
    | OX → SAT
    | UX → UNSAT

```

Existing tools take various approaches when receiving an UNKNOWN answer from the solver. CBMC issues a warning, attesting it cannot determine the absence of a bug. Such a warning is informative but does not provide more theoretical guarantees than an error. JaVerT 2.0 would simply crash, throwing an exception and terminating execution. While not elegant, this approach is still sound and ensures the absence of false negatives. In contrast, VeriFast does not differentiate between UNKNOWN and SAT, adopting precisely the behaviour we prescribe for OX-approximate solvers.

6.5 Symbolic execution processes

Equipped with all the necessary basic definitions, we are ready to define *symbolic execution processes*. Essentially, a symbolic execution process is a function that receives a symbolic abstraction and produces a set of pairs called *branches*, composed of an abstraction and a path condition. They are typically visualised using tree diagrams, as in Figures 6.2 and 6.4.

Symbolic semantics, actions, consumers and producers can all be formalised as symbolic processes. Additionally, simple processes can be composed together to form more complex processes, and the composition operator has soundness preservation properties, allowing for modular soundness proofs.

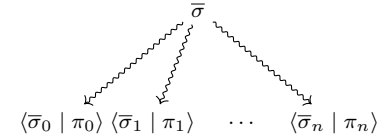


Figure 6.4: A symbolic process

6.5.1 Branches

First, we define the notion of symbolic branch. This definition formalises the use of path conditions to constrain the interpretation of symbolic abstractions.

Definition 6.11 (Symbolic branch).

A symbolic branch is a pair composed of a symbolic abstraction and a path condition and is denoted $\langle \bar{a}^* \mid \pi \rangle \in \langle \bar{A}^* \mid \Pi \rangle$. Symbolic branches themselves are symbolic abstractions over A , where

$$\varepsilon, a \models \langle \bar{a}^* \mid \pi \rangle \iff \varepsilon, a \models \bar{a}^* \wedge \pi(\varepsilon) = \mathbf{true}$$

6.5.2 Symbolic processes

Symbolic processes are functions that return a set of branches given a symbolic abstraction as input. We use them to formalise symbolic semantics, actions, producers and consumers.

Definition 6.12 (Symbolic processes).

A symbolic process from \bar{A}^* to \bar{B}^* is a function of $\bar{A}^* \rightarrow \mathcal{P}(\langle \bar{B}^* \mid \Pi \rangle)$.

For instance, given a fixed program γ , specification context Γ , and expression e , the symbolic semantics is a symbolic process that receives

a symbolic substitution $\bar{\theta} \in \text{Var} \rightarrow \overline{\text{Val}}$ that maps variables to symbolic values and the current symbolic state $\bar{\sigma}$. It returns a set of branches containing a triple $(o, \bar{v}, \bar{\sigma}') \in \mathcal{O} \times \overline{\text{Val}} \times \bar{\Sigma}$ and associated path conditions. Each input pair and resulting triple is a symbolic abstraction, defined as⁹:

$$\begin{aligned} \varepsilon, (\theta, \sigma) \models (\bar{\theta}, \bar{\sigma}) &\iff \bar{\theta}(\varepsilon) = \theta \wedge \varepsilon, \sigma \models \bar{\sigma} \\ \varepsilon, (o, v, \sigma) \models (o', \bar{v}, \bar{\sigma}) &\iff o = o' \wedge \bar{v}(\varepsilon) = v \wedge \varepsilon, \sigma \models \bar{\sigma} \end{aligned}$$

where the interpretation of a symbolic substitution is obtained by interpreting each binding individually

$$\bar{\theta} = [\mathbf{x} \mapsto \bar{v}_1, \mathbf{y} \mapsto \bar{v}_2, \dots] \implies \bar{\theta}(\varepsilon) = [\mathbf{x} \mapsto \bar{v}_1(\varepsilon), \mathbf{y} \mapsto \bar{v}_2(\varepsilon), \dots]$$

Note that symbolic processes are not computable if they return an infinite number of branches. For instance, Gillian’s whole-program symbolic testing could loop infinitely in the presence of recursive functions. The number of branches is artificially capped to prevent this, yielding a *bounded symbolic execution*. In contrast, using specifications for recursive function calls avoids this problem entirely during compositional verification.

Soundness Recall that symbolic execution aims to solve the problem of non-determinism in the concrete semantics yielding an infinite (or large) number of outcomes, rendering the exhaustive search for bugs impossible.

We establish the definition of soundness between symbolic processes and non-deterministic processes, depending on the approximation mode.

\bar{f} is said to be OX-sound with respect to f if all executions of f have a corresponding symbolic path in \bar{f} . More formally, if $f(a) \rightsquigarrow b$ and a is a model of \bar{a}^* under the interpretation ε , then there must exist a symbolic branch $\langle \bar{b}^* \mid \pi \rangle$ produced by $\bar{f}(\bar{a}^*)$ that has b as a model under an extended interpretation $\varepsilon' \geq \varepsilon$. This property guarantees that all executions of f are covered by at least one execution of \bar{f} , ensuring that all concrete erroneous executions will be detected using the symbolic process. Extending the interpretation is necessary to account for the fact that the symbolic process may introduce new symbolic variables¹⁰.

Conversely, \bar{f} is UX-sound with respect to f if every symbolic path corresponds to at least one concrete execution. If $\bar{f}(\bar{a}^*) \rightsquigarrow \langle \bar{b}^* \mid \pi \rangle$, then the path condition π must be feasible under at least one interpretation ε . For any such ε , \bar{b}^* must have at least one model b , and for any such model, there must exist a model a of \bar{a}^* such that $f(a) \rightsquigarrow b$. Note that this implies that it is sufficient to check the satisfiability of the path condition resulting from a UX-sound symbolic process to determine if there is a corresponding concrete execution. This property ensures that all satisfiable erroneous executions produced by \bar{f} are true bugs reachable by f .

Definition 6.13 (Symbolic soundness).

Let \bar{A}^* and \bar{B}^* be sets of symbolic abstractions over A and B . Let $f : A \rightarrow \mathcal{P}(B)$ be a non-deterministic process from A to B and $\bar{f} : \bar{A}^* \rightarrow \mathcal{P}(\langle \bar{B}^* \mid \Pi \rangle)$ be a symbolic execution process from \bar{A}^* to \bar{B}^* .

⁹ In fact, this definition can be algebraically constructed:

- concrete sets, such as the set of outcomes, can be used as their own trivial symbolic abstraction;
- symbolic values and symbolic substitutions are symbolic abstractions that are always interpreted to a singleton; and
- the product of abstraction forms an abstraction which is interpreted point-wise.

Each of these constructions is obtained for free through the definition of the **Abs** functor.

¹⁰ Jacobs et al., “Featherweight VeriFast”, 2015 [JVP15]

\bar{f} is OX-sound (resp. UX-sound) with respect to f if:

$$\begin{aligned} f(a) \rightsquigarrow b \wedge \varepsilon, a \models \bar{a}^* &\implies \\ \exists \bar{b}^*, \pi, \varepsilon' \geq \varepsilon. \bar{f}(\bar{a}^*) \rightsquigarrow \langle \bar{b}^* \mid \pi \rangle \wedge \varepsilon', b \models \langle \bar{b}^* \mid \pi \rangle & \quad (\text{OX} \bullet \text{symbolic soundness}) \end{aligned}$$

$$\begin{aligned} \bar{f}(\bar{a}^*) \rightsquigarrow \langle \bar{b}^* \mid \pi \rangle &\implies \\ (\text{SAT}(\pi) \wedge \forall \varepsilon. \pi(\varepsilon) = \mathbf{true} \Rightarrow \exists b. \varepsilon, b \models \bar{b}^* \wedge & \\ (\forall b. \varepsilon, b \models \bar{b}^* \Rightarrow (\exists a \varepsilon, a \models \bar{a}^* \wedge f(a) \rightsquigarrow b))) & \quad (\bullet \text{UX} \text{symbolic soundness}) \end{aligned}$$

Notation. We write $\bar{f} \stackrel{S}{\sim}_m f$ if \bar{f} is an m -sound symbolic process with respect to the non-deterministic process f , where $m \in \{\text{OX}, \text{UX}, \text{EX}\}$

6.5.3 Composition and monad structure

We have now defined symbolic processes and what it means for a symbolic process to be sound. However, implementing large processes (such as symbolic semantics) and proving their soundness can quickly become cumbersome. To reduce the required effort, we define soundness-preserving composition operators that allow for the construction of large symbolic processes from smaller ones.

To define the composition, we give a monad structure to symbolic branches¹¹. To keep the discussion straightforward and focused, we omit a detailed exposition of the category theory involved, including the formal assertion that this composition adheres to the *monad laws*. However, these details are fully presented and proven in [Appendix C.1](#).

Bind Recall that composition for a monad is defined using a bind operator. In the case of symbolic processes, the bind operator describes how to apply a new symbolic process to the result of a previous one. Since the result of a symbolic process is a set of branches, the bind operator applies the new process to each branch and monotonically accumulates the path conditions using the conjunction operator. Additionally, the bind operator ensures that the resulting branches have a satisfiable path condition. This definition is, therefore, implicitly parametric on an approximate solver SAT_m .

Definition 6.14 (Bind on the symex monad).

Let $\bar{f} : \bar{A}^* \rightarrow \mathcal{P}(\langle \bar{B}^* \mid \Pi \rangle)$ and $r \in \mathcal{P}(\langle \bar{A}^* \mid \Pi \rangle)$. The bind operator for the symbolic execution monad is defined as follows:

```
type 'a symex = ('a × Π) set
let bind (r: 'a symex) (f: 'a → 'b symex) : 'b symex =
  { ⟨ b̄* | π ∧ π' ⟩ | ⟨ ā* | π ⟩ ∈ r ∧ ⟨ b̄* | π' ⟩ ∈ f(ā*) ∧ SATm(π ∧ π') }
```

Figure 6.5 illustrates the composition of two symbolic processes. The first process receives a symbolic abstraction \bar{a}^* and yields n branches $\langle \bar{b}_0^* \mid \pi_0 \rangle$ to $\langle \bar{b}_n^* \mid \pi_n \rangle$. A second process is then applied using the bind operator on the set of branches. When applied to the abstraction \bar{b}_0^* of the first resulting branch, the second symbolic process yields of m new branches $\langle \bar{c}_0^* \mid \pi'_0 \rangle$ to $\langle \bar{c}_m^* \mid \pi'_m \rangle$. The bind operator monotonically accumulates the path conditions using the conjunction operator, and

¹¹ Such that symbolic processes become Kleisli arrows in that monad.

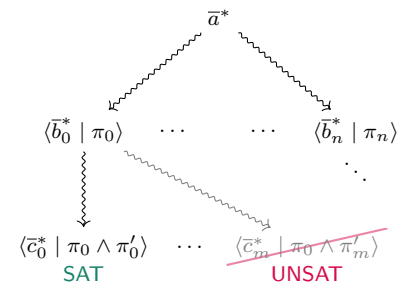


Figure 6.5: Two symbolic processes composed.

the branches resulting from the composition are therefore further constrained using the path condition π_0 – which had been obtained from the first process. After adding this constraint, the branches with a path condition declared unsatisfiable by the solver are filtered out.

Soundness preservation We can now state that the composition of two sound symbolic processes yields a sound symbolic process. To do so elegantly, we first provide the definition of *Kleisli composition*.

Definition 6.15 (Kleisli composition).

Given the `bind` operator of a monad, and two processes f and g , the Kleisli composition of f and g , denoted using the fish operator $f \gg g$, is defined as:

`let` $f \gg g = \text{fun } a \rightarrow \text{bind } (f \ a) \ g$

This definition allows us to compose two symbolic processes \bar{f} and \bar{g} using the above bind operator. It also lets us define the composition of concrete non-deterministic processes using the bind operator that handles non-determinism.¹²

Theorem 6.16 (Process composition: soundness preservation).

$$\bar{f} \stackrel{s}{\sim}_m f \wedge \bar{g} \stackrel{s}{\sim}_m g \implies \bar{f} \gg \bar{g} \stackrel{s}{\sim}_m f \gg g$$

This theorem shows that the composition of any *two* sound symbolic processes is sound. Together with the fact that the induced monad satisfies the monad laws, it ensures that chaining *any number* of such processes yields a sound symbolic process.

Symbolic branching We have defined the bind operator as a means to compose two symbolic processes sequentially. Another equally pivotal composition of symbolic processes is through *conditional branching*. Conceptually, conditional branching for symbolic processes mirrors the behaviour of the `if/else` construct. Given two symbolic processes \bar{f} and \bar{g} , if a guard π is satisfiable, then the first is applied, and in the case where its negation $\neg\pi$ is satisfiable, the second is applied. If both π and its negation are satisfiable, then both branches are explored.

Figure 6.6 illustrates a simple case of conditional branching. Given input abstraction \bar{a}^* , \bar{f} produces a single branch $\langle \bar{b}_f^* \mid \pi_f \rangle$, and \bar{g} produces a single branch $\langle \bar{b}_g^* \mid \pi_g \rangle$. Since the branches are executed *conditionally*, the guard π is added to the path condition of the first branch, and its negation is added to the path condition of the second branch. Moreover, in this example, the obtained path condition $\pi_g \wedge \neg\pi$ is unsatisfiable, and therefore the second branch is dropped.

Formally, branching is not defined using two symbolic processes \bar{f} and \bar{g} but their resulting sets of branches $r_f, r_g \in \mathcal{P}(\langle \bar{B}^* \mid \Pi \rangle)$. The functional definition uses an auxiliary `assume` function, which returns a single branch containing the unit value $()$ and a given path condition. Since branches are checked for satisfiability using the bind operator, this definition implicitly depends on an approximate checker SAT_m .

Definition 6.17 (The branch function).

The branch function is defined as follows using functional notations

¹² In the powerset monad, the `bind` operator is defined as:

$$\text{let bind } s \ f = \bigcup_{x \in s} (f \ x)$$

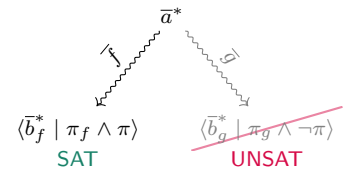


Figure 6.6: A simple symbolic process obtained using the symbolic branching operator.

```

let assume  $\pi$  = if SATm  $\pi$  then {  $\langle () \mid \pi \rangle$  } else vanish

let branch  $\pi$   $r_f$   $r_g$  =
  (bind (assume  $\pi$ ) (fun ()  $\rightarrow r_f$ ))
   $\cup$  (bind (assume  $\neg\pi$ ) (fun ()  $\rightarrow r_g$ ))

```

It was mentioned earlier that conditional branching mirrors the behaviour of **if/else**. In fact, we show that processes resulting from conditional branching are sound with respect to non-deterministic processes constructed from concrete branching.

Theorem 6.18 (Conditional branching: soundness preservation).

Let $f, g : A \rightarrow \mathcal{P}(C)$ and be non-deterministic processes and h be defined as:

```
let h (b, a) = if b then f a else g a
```

Furthermore, let \bar{A}^* and \bar{C}^* be sets of symbolic abstractions for A and C . Let $\bar{f}, \bar{g} : \bar{A}^* \rightarrow \mathcal{P}(\langle \bar{C}^* \mid \Pi \rangle)$ be m -sound symbolic processes with respect to f and g . Finally, let \bar{h} be the symbolic process defined as:

```
let  $\bar{h}$  ( $\pi$ ,  $\bar{a}^*$ ) = branch  $\pi$  ( $\bar{f} \bar{a}^*$ ) ( $\bar{g} \bar{a}^*$ )
```

Then, if the branch operator makes use of an m -approximate solver, then \bar{h}^* is an m -sound symbolic process with respect to h .

The input of \bar{h}^* is a product abstraction defined as

$$\begin{aligned} \varepsilon, (b, a) &\models (\pi, \bar{a}^*) \\ \iff \pi(\varepsilon) = b \wedge \varepsilon, a &\models \bar{a}^* \end{aligned}$$

Error handling and syntactic sugar Our goal is to be able to provide an elegant definition of the parametric symbolic semantics for SIGIL in the next section. So far, we have designed the basic abstractions that allows us to do so. However, the notations can still largely be improved.

First, note that the bind operator defined above for symbolic processes abstracts over non-deterministic processes, but does not yet support error handling as required for defining semantics elegantly. Thankfully, the bind operator can be straightforwardly¹³ modified to handle outcomes, such that execution continues in case of success and terminates in case of error, similarly to the concrete execution monad. This modification does not compromise soundness.¹⁴

We can then assign the updated definition of the bind operator to a **let*** operator, allowing us to seamlessly compose symbolic processes as we did concrete processes in §4.4.

Similarly, we can override the OCaml **if/else** syntax to imply a call to the branch function. In the rest of this presentation, we define the **if%sat**¹⁵ operator such that:

```
if%sat  $\pi$  then  $r_f$  else  $r_g$  is desugared to branch  $\pi$   $r_f$   $r_g$ 
```

These new syntactic sugars are particularly powerful when comparing a concrete semantics and a symbolic semantics side-by-side. To illustrate, here are the concrete and symbolic semantics of the **if/else** expression of SIGIL side by side. The concrete semantics is copied¹⁶ from the definition provided in §4.4.

¹³ To avoid cluttering this presentation, we do not explicitly write its definition. For the familiar reader, it is obtained from applying the Outcome monad transformer to the symbolic execution monad.

¹⁴ The formal justification consists in considering the content of the resulting symbolic branches as pairs (o, \bar{b}^*) , where concrete outcomes o acts as their own abstractions, and pairs are interpreted pointwise.

¹⁵ This syntax is, in fact, valid in OCaml, and defined using a pre-processor extension (or PPX). Gillian's implementation comes with this extension.

¹⁶ To reduce clutter, we omit parameters irrelevant to this specific case

```

let rec eval  $\theta$   $\sigma$  e =
  match e with
  | if  $e_g$  then  $e_t$  else  $e_e \rightarrow$ 
    let* ( $b$ ,  $\sigma'$ ) = eval  $\theta$   $\sigma$   $e_g$  in
    let* () = assert_type  $b$   $\mathbb{B}$  in
    if  $b$  then
      eval  $\theta$   $\sigma'$   $e_t$ 
    else
      eval  $\theta$   $\sigma'$   $e_e$ 

let rec  $\overline{\text{eval}}$   $\overline{\theta}$   $\overline{\sigma}$  e =
  match e with
  | if  $e_g$  then  $e_t$  else  $e_e \rightarrow$ 
    let* ( $\pi$ ,  $\overline{\sigma}'$ ) =  $\overline{\text{eval}}$   $\overline{\theta}$   $\overline{\sigma}$   $e_g$  in
    let* () = assert_type  $\pi$   $\mathbb{B}$  in
    if%sat  $\pi$  then
       $\overline{\text{eval}}$   $\overline{\theta}$   $\overline{\sigma}'$   $e_t$ 
    else
       $\overline{\text{eval}}$   $\overline{\theta}$   $\overline{\sigma}'$   $e_e$ 

```

Just looking at the above and using [Theorems 6.16](#) and [6.18](#), it can be trusted that the symbolic semantics is sound with respect to the concrete one.

More pragmatically, this syntax allows users of the Gillian framework to write symbolic actions, as well as producers and consumers, almost as if they were writing concrete ones. The burden of symbolic execution and error handling is delegated to the monad and hidden behind a familiar syntax.

6.6 Interlude: implementation and optimisation

The abstract framework introduced here is designed to simplify proofs of soundness for the various symbolic processes that must be defined. The symbolic execution monad has been implemented in Gillian¹⁷, demonstrating its practical utility. In this section, we offer insights into its implementation and optimisation. Any optimization applied to the symbolic execution monad is invaluable, given its role in every symbolic operation.

Branch exploration Symbolic processes are defined to return *sets* of branches. In a real implementation, using sets of branches (encoded using binary trees or hashsets) would require superfluous duplicability checks, and yield unpredictable execution orders. In particular, execution order is important in the case where execution does not terminate.

In practice, a symbolic execution processes would yield *lists* of branches, or *sequences*¹⁸. Below, we provide an implementation of the bind operator that depends on a module \mathbb{M} , which should be substituted with `List` or `Seq`¹⁹.

```

let bind x f =
  M.concat_map (fun ( $\overline{v}$ ,  $\pi$ )  $\rightarrow$ 
    M.concat_map (fun ( $\overline{v}'$ ,  $\pi'$ )  $\rightarrow$ 
      if SATm( $\pi \wedge \pi'$ ) then M.singleton ( $\overline{v}'$ ,  $\pi \wedge \pi'$ )
      else M.empty
    ) (f  $\overline{v}$ )
  ) x

```

Using lists, the above implementation explores the symbolic execution tree breadth-first, while the exploration is performed depth-first using sequences.

Breadth-first search is interesting when the depth of the symbolic process tree is infinite, for example, if the process is non-terminating. Using this approach, all nodes are *eventually* explored, while a depth-first search could lead execution to get stuck in a single infinite branch. Therefore, using breadth-first search and an OX engine, all bugs would

¹⁷ For legacy reasons, not all symbolic processes in the Gillian implementation are defined using the monad. It is mainly used in the Gillian-C and Gillian-Rust state models.

¹⁸ Sequence here is defined in the sense of OCaml's `Seq` module and which, according to the OCaml documentation “can be thought of as a delayed list, that is, a list whose elements are computed only when they are demanded by a consumer.”

¹⁹ `concat_map` is an alternative name for the bind operator commonly used for monads that behave as collections such as `List` and `Seq`. Using any other collection monad \mathbb{M} would also work, but render branch exploration less straightforward.

eventually be discovered. This observation has recently been formalised in the context of a mechanised symbolic execution engine²⁰.

In the context of compositional symbolic execution, however, function specifications usually ensure the absence of non-terminating symbolic processes. Therefore, the choice of branch exploration is less critical. In fact, a depth-first search may reduce the memory pressure and can be used to leverage *incremental solvers*, as described in a further paragraph.

Fail-fast execution It is common to terminate execution as soon as an error prevents successful verification during OX analysis so as to provide immediate feedback to the user. Gillian, VeriFast, and Viper all adopt this fail-fast execution strategy.

Our meta-theoretical framework does not allow for the specification of such behaviour, as it is cumbersome to model and of limited theoretical interest. However, in the rest of this presentation, we sometimes write $\bar{f}(\bar{a}^*) \rightsquigarrow \mathbf{Abort}$ to imply that immediate failure is the desired behaviour. By defining \mathbf{Abort} ²¹ to be the symbolic abstraction that invariably captures all concrete elements²², theoretical soundness is maintained.

Access to the current path condition Symbolic processes receive an abstraction as input and must return the set of all possible branches starting from that input. The concatenation of path conditions is then handled when composing the processes. However, previous formalisations of Gillian would present the symbolic semantics as a function receiving both the current state and path condition and returning a set of branches with updated state and path condition. Then, part of proving soundness for under-approximating execution would consist of showing that path conditions may only strengthen.

While our approach immediately guarantees the monotonicity of the path condition, thereby alleviating the burden of proof, it also prevents the symbolic execution engine from inspecting the *current* path condition.

In practice, accessing the current path condition may help with various optimisations, such as the ability to reduce expressions contextually. For instance, if the path condition entails that a variable \bar{x} may only take value v , the variable can be swapped with its value in the current state. Using the concrete value helps to avoid redundant or trivial solver calls.

Furthermore, having access to a current path condition often allows for early cutting of infeasible paths, thereby saving resources. The inability to access the current path condition can be seen as a shortcoming of the symbolic execution monad.

Thankfully, this issue is overcome by introducing an alternative definition of the symbolic execution monad as functions that receive the current path condition and return a set of branches with updated path conditions that must be strictly stronger. Below, the current definition of the monad is provided as a reminder on the left, and the updated version with the ability to access the solver state is presented

²⁰ Correnson et al., “Engineering a Formally Verified Automated Bug Finder”, 2023 [CS23]

²¹ Of course, this corresponds to the maximum of the abstraction lattice from an abstract interpretation point of view.

²² i.e. for $\mathbf{Abort} = \lambda_. A$, such that

$$\forall a, \varepsilon. \varepsilon, a \models \mathbf{Abort}$$

on the right.

$$\text{Symex}(\bar{A}^*) = \mathcal{P}(\langle \bar{A}^* \mid \Pi \rangle) \left| \begin{array}{l} \text{Symex}(\bar{A}^*) = \\ \{f \in \Pi \rightarrow \mathcal{P}(\langle \bar{A}^* \mid \Pi \rangle) \text{ s.t.} \\ \forall \pi. \forall \langle \bar{a}^* \mid \pi' \rangle \in f(\pi). \pi' \Rightarrow \pi\} \end{array} \right.$$

The old monad can be seamlessly swapped with this new monad version without observable differences by defining the appropriate `bind` and `branch` operators. The latter is strictly more expressive and can implement new symbolic processes such as `reduce` function that receives a symbolic value and reduces it using the current path condition.

While we do not explicitly provide the implementations of the basic operations for this alternative monad definition, they can be found in the implementation of Gillian.

Optimising the branch function The branch function presented in the previous section can be optimised by noticing that the following implication holds:

$$\text{UNSAT}(\pi) \Rightarrow \text{SAT}(\neg\pi)$$

If the path condition provided as input of the branch function is unsatisfiable, then the satisfiability check for its negation is guaranteed to be successful, and the second branch can be explored without any SAT check required.

Note that this optimisation requires the above-defined alternative implementation of the symbolic execution monad, which allows for the current path condition to be accessed.

Interestingly, this makes the definition of symbolic processes biased. It encourages developers to put cases that are often unfeasible on the left operand of the branch operator. An example is provided in §6.8.1, where the symbolic `assert` action is implemented with this optimisation in mind.

Incremental solving Modern SMT solvers allow for incremental solving. Users can add checkpoints using the `push` operation and backtrack to the last checkpoint using the `pop` operation, forgetting any new knowledge acquired since. This allows²³ for faster solving times and reduced memory usage.

We propose another implementation of the symbolic execution monad, which leverages incremental solving by using OCaml’s ability to perform side effects and modify the solver state in place. Symbolic processes become iterators over the final states of the process, and the path condition is updated in the background. Every time a process branches, a new checkpoint is added to the solver, and when a leaf is reached, a `pop` operation is performed to backtrack to the last checkpoint and explore the next branch. Figure 6.7 illustrates the structure of symbolic processes using this implementation, and the implementation of the branch operator is provided below.

```
type 'a symex = 'a seq

let branch (π: Π) (t: 'a symex) (e: 'a symex) : 'a symex =
  Seq.append
    (fun () →
```

²³ Some SMT solvers support incremental solvers more efficiently than others.

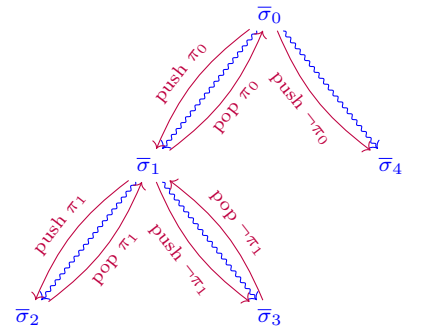


Figure 6.7: Incremental symbolic process, arrows are annotated with the corresponding operation in the solver. States are explored in the order matching their numbering. The root branching is guarded by condition π_0 and the other one is guarded by π_1


```

Solver.push ();
Solver.add_constraints  $\pi$  ;
if Solver.sat () then t () else Seq.empty ()
(fun () →
  Solver.pop ();
  Solver.add_constraint ( $\neg\pi$ );
  if Solver.sat () then e () else Seq.empty ())

```

This new approach allows for implementing all primitive operations of the symbolic execution monad. Therefore, it is a valid alternative to the simpler monad presented in this formalisation.

Although this implementation is not included in Gillian, we have explored it within the scope of a prototype and discovered it to be a promising alternative to the existing methods.

Caching and batching SAT checks Symbolic execution can result in many small checks that are sometimes repeated. A common technique to reduce the overhead of these checks is to use caching and batching. Both these techniques are commonly used in symbolic execution engines and, therefore, deserve to be mentioned here.

Caching consists of storing the results of previous checks and reusing them when the same check is requested again. The overhead of caching is usually low, and it can be very effective when the same queries are repeated often. More advanced caching techniques specialized to symbolic execution have been researched in the past²⁴, but are yet to be experimented with in the context of compositional symbolic execution.

Batching consists of checking satisfiability only when a certain number of constraints have been added to the path condition or when the end of a branch has been reached. While this can mean exploring some unfeasible branches for longer than necessary, it can also reduce the number of queries sent to the solver. Substantial speedups have been observed when batching queries during memory accesses in the Gillian-C memory model.

²⁴ Trabish et al., “Address-Aware Query Caching for Symbolic Execution”, 2021 [TIR21]

6.7 Parametric symbolic execution for SIGIL

Previous sections of this chapter focused on defining the mathematical framework to describe and reason about symbolic execution and its soundness. This section applies this mathematical framework to the symbolic parametric semantics of SIGIL.

Remember from §4.4 that the concrete semantics of SIGIL $\mathcal{S}_{\underline{\mathbb{S}}}$ is parametric on a state model $\underline{\mathbb{S}} = (\mathcal{A}, \Sigma, \text{eval_action})$, which is a triple composed of a set of actions, a set of states and an action evaluation function. Additionally, one may use a compositional state model $\mathbb{S} = (\mathcal{A}, \Sigma, \text{eval_action})$ as parameter for the SIGIL semantics, yielding in a compositional semantics $\mathcal{S}_{\mathbb{S}}$. If the compositional state model \mathbb{S} is m -concrete-sound with the full state model $\underline{\mathbb{S}}$ (i.e. $\mathbb{S} \stackrel{\mathcal{C}}{\sim} \underline{\mathbb{S}}$), then, in turn, the resulting compositional semantics is *sound* with respect to the full semantics, $\mathcal{S}_{\mathbb{S}} \stackrel{\mathcal{C}}{\sim} \mathcal{S}_{\underline{\mathbb{S}}}$. In §5.4, we have also extended the compositional semantics with the ability to execute specifications.

Here, we define the symbolic semantics of SIGIL, which is parametric on a *symbolic compositional state model* $\bar{\mathbb{S}}$. This symbolic semantics

is soundness preserving: if the symbolic state model $\bar{\mathbb{S}}$ is m -sound with respect to a compositional state model \mathbb{S} , denoted $\bar{\mathbb{S}} \stackrel{\mathbb{S}}{\sim} \mathbb{S}$, then the resulting symbolic semantics $\bar{\mathcal{S}}_{\bar{\mathbb{S}}}$ is m -sound with respect to the compositional semantics, $\bar{\mathcal{S}}_{\bar{\mathbb{S}}} \stackrel{\mathbb{S}}{\sim} \mathcal{S}_{\mathbb{S}}$.

Note that the soundness property is orthogonal to that of compositionality. Hence, given a full state model $\underline{\mathbb{S}}$, it is possible to define a full symbolic state model $\bar{\underline{\mathbb{S}}} \stackrel{\underline{\mathbb{S}}}{\sim} \underline{\mathbb{S}}$ and use it as a parameter of the symbolic semantics. This would yield a sound symbolic full semantics $\bar{\mathcal{S}}_{\bar{\underline{\mathbb{S}}}} \stackrel{\underline{\mathbb{S}}}{\sim} \mathcal{S}_{\underline{\mathbb{S}}}$ that can be used for non-compositional symbolic execution. Nevertheless, to limit the scope of our presentation, we focus exclusively on symbolic compositional state models.

6.7.1 Symbolic state models

The signature of a symbolic compositional state model $\bar{\mathbb{S}}$ mirrors that of a concrete compositional state model \mathbb{S} . For $\bar{\mathbb{S}}$ to be m -sound with respect to \mathbb{S} for a mode $m \in \{\text{OX}, \text{UX}\}$, denoted $\bar{\mathbb{S}} \stackrel{\mathbb{S}}{\sim}_m \mathbb{S}$, it must comprise:

- a set of symbolic states $\bar{\Sigma} \subseteq \text{Abs}(\mathbb{S}.\Sigma)$ which are symbolic abstractions over the set of concrete state fragments;
- a symbolic empty state $\bar{0}$ which must have only $\mathbb{S}.0$ as model under any interpretation²⁵;
- a set of actions, which must be identical to those of \mathbb{S} ;
- an action evaluation function which is a symbolic process that receives a symbolic state and a list of symbolic values, returns a set of well-formed branches, and must be sound with respect to the concrete action evaluation function;
- a set of core predicates $\bar{\mathbb{S}}.\Delta = \mathbb{S}.\Delta$; and
- a symbolic producer and consumer that must be sound with respect to the concrete ones.

²⁵ That is, $\bar{0} = \lambda_ . \{\mathbb{S}.0\}$

Below, we provide the complete signature of symbolic state models as they must be implemented by a tool developer using our framework.

Definition 6.19 (Symbolic state model).

A symbolic state model $\bar{\mathbb{S}}$ has the following signature:

```
module type Symbolic_state_model = sig
  type  $\bar{\Sigma}$ 
  val  $\bar{0} : \bar{\Sigma}$ 

  type  $\mathcal{A}$ 
  val eval_action :  $\mathcal{A} \rightarrow \bar{\Sigma} \rightarrow \overline{Val} \text{ list} \rightarrow (\mathcal{O} \times \overline{Val} \times \bar{\Sigma}) \text{ symex}$ 

  type  $\Delta$ 
  val produce :  $\Delta \rightarrow \bar{\Sigma} \rightarrow \overline{Val} \text{ list} \rightarrow \overline{Val} \text{ list} \rightarrow \bar{\Sigma} \text{ symex}$ 
  val consume :  $\Delta \rightarrow \bar{\Sigma} \rightarrow \overline{Val} \text{ list} \rightarrow (\mathcal{O}_l^+ \times \overline{Val} \times \bar{\Sigma}) \text{ symex}$ 
end
```

6.7.2 Parametric symbolic semantics of SIGIL

The parametric symbolic semantics of SIGIL is obtained by lifting the concrete specification semantics defined in §5.4. The let-binding operator `let*` is replaced with that which handles the composition of symbolic processes and erroneous outcomes, as defined in the previous section. Additionally, all uses of `if/else` are replaced with symbolic

conditional branching, using the syntactic sugar `if%sat/else`. An excerpt is provided in Figure 6.8.

```

let rec eval  $\bar{S}$   $\gamma$   $\Gamma$   $\bar{\theta}$   $\bar{\sigma}$   $e$  : ( $\mathcal{O} \times \overline{Val} \times \bar{\Sigma}$ ) symex =
  let eval = eval  $\gamma$   $\Gamma$  in (* These arguments do not change *)
  match e with
  | let x = e1 in e2 →
    let* ( $\bar{v}, \bar{\sigma}'$ ) = eval  $\bar{\theta}$   $\bar{\sigma}$  e1 in
    eval  $\bar{\theta}$  [x ←  $\bar{v}$ ]  $\bar{\sigma}'$  e2
  | if eg then et else ee →
    let* ( $\pi, \bar{\sigma}'$ ) = eval  $\bar{\theta}$   $\bar{\sigma}$  eg in
    let* () = assert_type  $\pi$   $\mathbb{B}$  in
    if%sat  $\pi$  then
      eval  $\bar{\theta}$   $\bar{\sigma}'$  et
    else
      eval  $\bar{\theta}$   $\bar{\sigma}'$  ee
  |  $\alpha(\bar{e})$  →
    let* ( $\bar{v}, \bar{\sigma}'$ ) = eval_all  $\bar{\theta}$   $\bar{\sigma}$   $\bar{e}$  in
     $\bar{S}.\text{eval\_action}$   $\alpha$   $\bar{\sigma}'$   $\bar{v}$ 
  | ...

```

Figure 6.8: Excerpt of the parametric symbolic semantics of SIGIL

Notation. Because the overlines make the above difficult to read, we sometimes omit them on function names such as `eval` of `eval_action` when it is clear from the context.

Theorem 6.20 (Semantics: soundness preservation).

If the symbolic state model \bar{S} is m -sound with respect to the concrete compositional state model S , then the induced symbolic semantics \bar{S}_S , using an m -approximate solver, is m -sound with respect to the concrete induced semantics S_S :

$$\bar{S} \stackrel{S}{\sim}_m S \implies \bar{S}_S \stackrel{S}{\sim}_m S_S$$

6.8 Example symbolic state models

We now continue our presentation of the linear heap and pure state model introduced in the previous sections. This time, however, we start by describing the symbolic pure state model, as it captures important base cases for writing symbolic processes.

6.8.1 Symbolic pure state model

Recall that the set of pure states $\Sigma = \text{unit}$ contains a unique element. We define symbolic states also to contain a unique element, which is the abstraction $\bar{0} = \overline{() = \lambda_. \{0\}}$. Since there is a unique symbolic state, we can consider that $\bar{\Sigma} = \text{unit}$.

Its symbolic actions are noteworthy for their coverage of crucial base cases and for offering insights into the distinctions between OX and UX analysis. They also highlight how to apply optimisations discussed in §6.6. Below, we provide their definitions along with an in-depth explanation for each.

```

module Pure : Symbolic_state_model = struct
  type  $\bar{\Sigma}$  = unit (* Type of states *)

  (* Actions are the same as in the concrete state model *)

```

```

type  $\mathcal{A}$  = nondet | assume | assert | skip

let nondet  $\bar{\sigma}$  =
  (* Create a fresh symbolic variable, that has not been used so far *)
  let  $\bar{x}$  = fresh_svar Val in
  ok ( $\bar{x}$ ,  $\bar{\sigma}$ )

let assume  $\bar{\sigma}$   $\pi$  =
  if%sat  $\pi$  then ok ( $\bar{\sigma}$ )
  else vanish

(* Alternative optimised implementation *)
let assume  $\bar{\sigma}$   $\pi$  =
  if SATm  $\pi$  then {  $\langle ok : (\bar{\sigma}) \mid \pi \rangle$  }
  else vanish

let assert  $\bar{\sigma}$   $\pi$  =
  if%sat  $\neg\pi$  then error (FailedAssert,  $\bar{\sigma}$ )
  else ok ( $\bar{\sigma}$ )

let skip  $\bar{\sigma}$  = ok ( $\bar{\sigma}$ )

let eval_action  $\alpha$   $\bar{\sigma}$   $\vec{v}$  = (* ... *)

type  $\Delta$  = Pure

let produce  $\bar{\sigma}$  [  $\pi$  ] =
  let* ( $\bar{\sigma}$ ) = assume  $\bar{\sigma}$   $\pi$  in
   $\bar{\sigma}$ 

let consume  $\sigma$  [  $\pi$  ] =
  if%sat  $\neg\pi$  then Lfail (PureMismatch,  $\bar{\sigma}$ )
  else ok ([],  $\bar{\sigma}$ )

end

```

Nondet In this presentation, symbolic execution has been referred to as a solution to handling non-determinism when it yields too many branches. The **nondet** action embodies this approach: the concrete action yields an infinite number of branches, returning all values $v \in Val$. On the other hand, the symbolic action yields a unique branch with a fresh symbolic variable of sort *Val*. The associated path condition is **true**, rendering the symbolic variable unconstrained and, therefore, satisfying the requirement that it is modelled by all values.

Assume We provide two implementations of **assume**. The first is obtained by lifting the concrete implementation by swapping the concrete execution monad for the symbolic execution monad. However, this implementation leads to an unnecessary check for the satisfiability of $\neg\pi$. Therefore, we propose an alternative approach that only checks for the satisfiability of π and vanishes if it is not. In practice, an **assume** function should be provided directly by the symbolic execution monad library, performing this trick once and for all.

While the first implementation can be straightforwardly proven sound by simply applying [Theorems 6.16](#) and [6.18](#), the latter corresponds to the more classical implementation of **assume** in symbolic execution engines. Both are equivalent, and the corresponding symbolic process is depicted in [Figure 6.9](#).

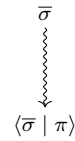


Figure 6.9: The **assume** π symbolic process

Assert The symbolic `assert` is also derived by lifting the concrete implementation. Note that the erroneous case has been deliberately placed as the first operand of the conditional branching. This placement leverages the branch operator optimisation outlined in §6.6. When using this optimisation, if the guard $\neg\pi$ is deemed unsatisfiable by the solver, the satisfiability of π will not be checked to continue execution. Under the assumption that assert statements should be successful more often than not, this decision should reduce the overall number of queries sent to the solver.

Moreover, should the engine adopt an OX fail-fast execution strategy, it would terminate execution the moment the guard is found to be satisfiable, avoiding exploration of the successful case when verification is doomed to fail.


These two optimisations consolidate the process into a unique entailment check: it verifies that π *must* hold by checking that $\neg\pi$ is unsatisfiable. In other OX tools, assert statements are usually executed using a single entailment check. This behaviour is naturally obtained by adding simple optimisations to the lifted implementation.

We occasionally use `if%ent` instead of `if%sat` when the desired behaviour is a single entailment check for the guard π . If this check is successful, the first branch is executed; if it fails, the second branch is executed.

Produce pure The symbolic producer for the pure core predicate is simply a lift of the concrete producer and is identical to the `assume` action. Therefore, it requires no further explanation.

Consume pure As stated in the description of the concrete producer for the pure core predicate, its implementation is identical to that of `assert`, apart from the erroneous case, which yields a logical failure instead of an error. All comments about optimisations made in the context of the `assert` action also hold in the context of the pure consumer.

In addition, recall that in UX execution, logical failures provide no information and are, therefore, ignored. This means the pure consumer can be further optimised to drop the erroneous branch early. By doing so, the implementation of the pure consumer becomes identical to the `assume`. The implementation of Gillian adopts this optimisation and performs an `assert` or an `assume`, depending on the mode.

```
(* -specific implementation *)
let consume_pure  $\bar{\sigma}$   $\pi$  =
  assume  $\bar{\sigma}$   $\pi$ 
```

6.8.2 Symbolic linear heap

Recall that every state in the linear heap state model is a partial finite map from natural integer to values, or \emptyset indicating the address has been freed.

$$\sigma \in \Sigma = \mathbb{N} \xrightarrow{fin} Val^\emptyset$$

Structure of the symbolic heap The art of creating a symbolic state model resides in choosing which components to abstract and which to keep more concrete. For instance, the heap could be entirely abstracted

to a symbolic array²⁶ where each cell contains either a natural number, the special \emptyset value, or a special missing \perp value. However, this approach would require complex universally quantified expressions²⁷ to, for instance, define the empty heap $(\forall i. \sigma(i) = \perp)$

Instead, we propose a simpler abstraction that maintains the concreteness of the freed value and the partial finite map structure, only abstracting the addresses and symbolic values.

$$\bar{\sigma} \in \bar{\Sigma} = \bar{\mathbb{N}} \xrightarrow{\text{fin}} \bar{\mathbb{N}}^\emptyset$$

Note that a partial map could be equivalently represented as a list of its bindings. Such a list of *heap chunks* is the representation employed by Viper or VeriFast²⁸.

The satisfaction relation for the symbolic heap is defined inductively:

$$\begin{aligned} \varepsilon, \emptyset &\models \emptyset \\ \varepsilon, [\bar{a}(\varepsilon) \mapsto \emptyset] \uplus \sigma &\models [\bar{a} \mapsto \emptyset] \uplus \bar{\sigma} \Leftrightarrow \varepsilon, \sigma \models \bar{\sigma} \\ \varepsilon, [\bar{a}(\varepsilon) \mapsto \bar{v}(\varepsilon)] \uplus \sigma &\models [\bar{a} \mapsto \bar{v}] \uplus \bar{\sigma} \Leftrightarrow \varepsilon, \sigma \models \bar{\sigma} \end{aligned}$$

In particular, note that the map $[\bar{x} \mapsto 0; \bar{y} \mapsto 1]$ has no interpretation under ε when $\varepsilon(\bar{x}) = \varepsilon(\bar{y})$, due to the use of the disjoint union operator \uplus .

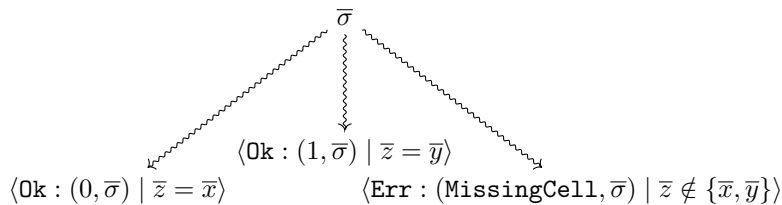
The load action Remember the implementation of the concrete load action, which is copied in the margin. Unfortunately, this implementation cannot soundly be lifted using our composition operators using

```
let load  $\bar{\sigma}$   $\bar{a}$  = match  $\bar{\sigma}(\bar{a})$  with ...
```

The culprit lies in the map access operation $\sigma(a)$ that cannot be lifted to the access $\bar{\sigma}(\bar{a})$. Doing so would perform a *syntactic* equality check instead of a *semantic* equality check when finding the correct binding in the map.

To illustrate, consider the heap $\bar{\sigma} = [\bar{x} \mapsto 0, \bar{y} \mapsto 1]$. What should happen when loading address \bar{z} ? There are three cases to consider, as depicted in Figure 6.10:

- if $\bar{z} = \bar{x}$, the load should return 0 and the state remains unchanged;
- if $\bar{z} = \bar{y}$, the load should return 1 and the state remains unchanged; and
- if $\bar{z} \notin \{\bar{x}, \bar{y}\}$, the load should yield a missing outcome, as the cell is not provably in the heap.



To obtain this behaviour, we must implement a sound symbolic process with respect to concrete map access. We call this process `symbolic_map_get`. This function must iterate through all the bindings of the map, checking for equality with the given address. Below, we provide its implementation:

²⁶ Using the SMT-Lib theory of arrays where each element contains an ADT with three cases.

²⁷ A rule of thumb for improving the performance of symbolic execution is to avoid quantified formulae when possible.

```
type value =
  | Active of Val
  | Freed
```

```
type  $\bar{\Sigma}$  = ( $\bar{\mathbb{N}}$ , value) Map.t
```

²⁸ VeriFast's state representation is a list of *heap chunks*, which do not differentiate between points-to predicates and more complex predicates (e.g. list predicates).

```
let load  $\sigma$   $a$  =
  match  $\sigma(a)$  with
  | Some v  $\rightarrow$ 
    ok (v,  $\sigma$ )
  | Some  $\emptyset \rightarrow$ 
    error (UseAfterFree,  $\sigma$ )
  | None  $\rightarrow$ 
    miss (MissingCell,  $\sigma$ )
```

Figure 6.10: A symbolic load of address \bar{z} in the heap $\bar{\sigma} = [\bar{x} \mapsto 0, \bar{y} \mapsto 1]$

```

let symbolic_map_get  $\bar{\sigma}$   $\bar{a}$  =
  match  $\bar{\sigma}$  with
  |  $\emptyset \rightarrow$  ok None
  |  $[\bar{a}' \mapsto \bar{v}^\emptyset] \uplus \bar{\sigma}' \rightarrow$ 
    if%sat  $\bar{a} = \bar{a}'$  then
      ok (Some  $\bar{v}^\emptyset$ )
    else
      symbolic_map_get  $\bar{\sigma}'$   $\bar{a}$ 

let load  $\bar{\sigma}$   $\bar{a}$  =
  let* looked_up = symbolic_map_get  $\bar{\sigma}$   $\bar{a}$  in
  match looked_up with
  | Some  $\bar{v} \rightarrow$  ok ( $\bar{v}$ ,  $\bar{\sigma}$ )
  | Some  $\emptyset \rightarrow$  error (UseAfterFree,  $\bar{\sigma}$ )
  | None  $\rightarrow$  miss (MissingCell,  $\bar{\sigma}$ )

```

The symbolic load action can then be lifted from the concrete one by using the symbolic map access function, as done above on the right. This implementation can now successfully be proven sound with respect to the concrete implementation by applying [Theorems 6.16](#) and [6.18](#). It will yield the expected three branches in the example case presented above.

Optimisations Although it is generally necessary to iterate through all elements of the map to ensure soundness, there are frequent instances where this can be avoided. Recall that the map only has models if the addresses are disjoint. Moreover, two symbolic expressions that are syntactically equal are also guaranteed to be semantically equal, independently of the path condition. Therefore, if the address looked for is syntactically present in the domain of the heap, then accessing the map directly is sound

Assuming map accesses are performed orders of magnitude faster than SAT checks, it is beneficial always to perform a direct map access and fall back to iterating through the bindings only if it is not. In practice, addresses can be put in a normal form to maximise the chances of a direct hit.

```

let optim_symbolic_map_get  $\bar{\sigma}$   $\bar{a}$  =
  match  $\bar{\sigma}(\bar{a})$  with
  | Some  $\bar{v}^\emptyset \rightarrow$  ok  $\bar{v}^\emptyset$ 
  | None  $\rightarrow$  symbolic_map_get  $\bar{\sigma}$   $\bar{a}$ 

```

Approximations A common approximation consists of preventing the engine from branching when accessing an address in a partial map. Instead of checking which binding *may* correspond to the provided address using a SAT check, the engine checks which binding *must* correspond to the provided address using an entailment check. The corresponding implementation is in the margin with differences from the previous implementation highlighted in **purple**.

If the value cannot be found in the map, the engine cannot determine if the value is missing and must immediately fail, returning **Abort**.

This approach is over-approximating and may lead to false positives. For instance, in the example provided in [Figure 6.11](#), the implementation using entailments as checks would yield a failure when dereferencing z , as it would be unable to prove that address z must be equal to x or must be equal y .

However, this approach is also more predictable as it prevents the engine from branching and is therefore taken by VeriFast and Viper. If a user wants to verify this specification, they must provide additional annotation to force the engine to consider both cases independently. This is a trade-off between automation and predictability and is a choice that must be made by the designer of the verification tool. In particular, the specific case depicted in [Figure 6.11](#) is rare enough that users may be willing to accept the false positive in exchange for more

```

let symbolic_map_get  $\bar{\sigma}$   $\bar{a}$  =
  match  $\bar{\sigma}$  with
  |  $\emptyset \rightarrow$  Abort
  |  $[\bar{a}' \mapsto \bar{v}^\emptyset] \uplus \bar{\sigma}' \rightarrow$ 
    if%ent  $\bar{a} = \bar{a}'$  then ok  $\bar{v}^\emptyset$ 
    else symbolic_map_get  $\bar{\sigma}'$   $\bar{a}$ 

void load_either(int* x, int* y,
  int* z)
  //@ requires  $x \rightarrow 0 * y \rightarrow 0 *$ 
  //@      ( $z == x \mid z == y$ );
  //@ ensures (result == 0);
  { return *z; }

```

Figure 6.11: An example case where using entailment leads to a false positive.

predictability of the engine.

Gillian-C and Gillian-JS use the approximate approach²⁹ when handling object locations, but use the exact approach when handling offsets or properties within an object. This allows for a good balance between automation and predictability and is a design choice that has been found to work well in practice.

The PointsTo core predicate The linear heap state model exposes a core predicate $\bar{x} \mapsto \bar{v}$ that models a cell at address \bar{x} containing value \bar{v} .

The symbolic consumer closely resembles the load action, with the only distinction being that it removes the accessed cell from the heap; hence, we won't elaborate on its implementation here. In the implementation of the producer, we discuss another trade-off between optimisation, approximation, and optimisation.

In UX mode, the symbolic producer must ensure that all branches with a satisfiable path condition have at least one model. Therefore, the path condition must ensure that the produced cell is disjoint from all the other cells in the heap. More formally, the following must always hold, where \equiv is syntactic equality:

$$\pi \models \left(\bigwedge_{\substack{\bar{a}, \bar{a}' \in \text{dom}(\bar{\sigma}) \\ \bar{a} \not\equiv \bar{a}'}} \bar{a} \neq \bar{a}' \right)$$

Therefore, the symbolic producer for the points-to core predicate must construct the disjointness constraint by iterating through the bindings of the map and adding this constraint to the path condition.

Conversely, in OX mode, there is no requirement for the existence of a model, and hence, this invariant need not be enforced. The symbolic producer can add the new cell to the heap without further checking.³⁰:

```
let produce_points_to  $\bar{\sigma}$   $\bar{a}$   $\bar{v}$  =
  ok  $\bar{\sigma}[\bar{a} \mapsto \bar{v}]$ 
```

This implementation is substantially more efficient than the one required in UX mode. However, this performance boost sacrifices automation, as users may be required to prune some infeasible paths manually.

For example, VeriFast does not enforce disjointness of addresses in the heap³¹. As such, it cannot automatically prove the specification provided in Figure 6.12 and instead requires additional annotations. In contrast, Gillian-C can perform the proof automatically, though not as efficiently³².

²⁹ In UX mode when it is impossible to decide, the branch is dropped instead of raising a warning.

```
void nop(int* x, int* y)
//@ requires x -> 0 * y -> 0 *
//@      (x == y);
//@ ensures false;
{ return; }
```

Figure 6.12: An example of valid OX specification that VeriFast cannot verify automatically. The syntax is adapted to match our presentation.

³⁰ If the address is already syntactically in the heap, production should result in an unfeasible heap. This over-approximating implementation would replace the binding, but that is not unsound.

³¹ It does enforce disjointness of fields for objects with identical syntactic addresses, but this is not enough for **OX** soundness.

³² An example further comparing performance and automation of VeriFast and Gillian-C is provided in Chapter 11

Chapter 7

Analyses

The unified parametric compositional execution framework presented in the previous chapters can host three kinds of analysis.




The first one, *whole-program symbolic testing*, symbolically executes the program from scratch to completion, producing results and guarantees similar to tools like CBMC.

The second one, *compositional verification*, verifies each function specification in isolation. This analysis is similar to that proposed by VeriFast or Viper.

The third one, *automatic UX specification synthesis*, automatically generates bounded under-approximate specifications for each function. This analysis is similar to that proposed by Infer:Pulse. However, Infer:Pulse also filters the generated specification to signal only *manifest* bugs (i.e., those that are always reachable in any context). This extra step has not yet been implemented in Gillian and is left as future work.

This chapter describes these analyses, providing additional definitions and succinct examples when required.

7.1 OX and UX whole-program symbolic testing

Gillian’s core symbolic execution engine can be used to implement a non-compositional analysis by disabling the application of function specifications¹. Programs are executed by running the `main` function and a given initial state until execution terminates. It is possible to use either an  or a  engine, which will guarantee the absence of false negatives or false positives. It is also possible to use an  engine, which should find exactly all the bugs if it terminates. However, exactness is often challenging to maintain when programs and semantics grow in complexity.

Of course, this analysis has a few restrictions. First, the analysis user must set up test harnesses, also called symbolic tests. These tests are similar to unit tests, except that developers can use the `nondet` actions to explore many concrete branches using a single symbolic branch.

Second, the symbolic execution engine might enter an infinite loop in the program, meaning that analysis would never terminate. To address this challenge, it is standard practice for whole-program symbolic testing tools to impose an upper bound on the number of iterations a loop (or a recursive function call) may execute. This approach is adopted by Gillian, rendering it a *bounded* symbolic testing tool.

Gillian’s whole-program symbolic testing engine is implemented differently from CBMC², but it provides the same guarantees as bounded model checkers, such as CBMC.

In fact, a version of Gillian-C exists that can substitute CBMC,

¹ Remember that, when executing a function, the specification semantics checks the environment Γ for the existence of a corresponding specification. If there is none, the body of the function is executed instead.

² CBMC compiles the entire program with unrolled loops to one big SAT formula. This process is sometimes called symbolic compilation [LB23]. A similar behaviour could be simulated in our framework by batching all SMT queries until all branches are terminated and creating a single query composed of the disjunctions of the path conditions for all erroneous branches.

analyse the same inputs, and produce similar results. It has been used inside industry as a prototype alternative back-end for the Kani Rust model checker.

Example We illustrate the use of whole-program symbolic testing by re-using [Example 6.1](#). In [Figure 7.1](#), we provide an `abs` function that replaces an integer in place in memory with its absolute value, as well as its test harness, i.e. a `main` function that acts as the entry point for the symbolic execution.

The main function would be compiled to the SIGIL code provided in [Figure 7.2](#), where action calls have been placed between angular brackets to distinguish them from function calls.

Executing this function from the empty state yields an execution similar to the one represented in [Figure 6.2](#), and show that the `assert` statement would always pass, for any value `v`.

7.2 Compositional verification

Compositional verification consists of verifying that a set of user-provided specifications for a program is valid.

The verification procedure In our framework, compositional verification uses the specification semantics with a specific solver and state model compatible with `OX` analysis. Given a program γ , the tool user writes `OX` specifications for each function they wish to verify, creating a specification context Γ . Each function f is then verified using the following procedure which returns `true` if it satisfies its given specification, and `false` otherwise:

```
let verify  $\Gamma$   $\gamma$   $f$  =
  let all_results =
    let  $f(\vec{x}) \{e\} = \gamma(f)$  in
    let {  $P$  }  $f(\vec{x}) \{ \text{Ok} : r. Q_{\text{Ok}} \} \{ \text{Err} : r. Q_{\text{Err}} \} = \Gamma(f)$  in
    let  $\vec{y} = \text{fv}(P)$  in
    ① let  $\bar{\theta} = [\vec{x}, \vec{y} \mapsto \text{fresh\_svar}()]$  in
    ② let*  $\bar{\sigma} = \text{produce\_asrt } \bar{\theta} \ 0 \ P$  in
    ③ let* ( $\underline{o} : \bar{v}, \bar{\sigma}'$ ) = eval  $\gamma$   $\Gamma$   $\bar{\theta}$   $\bar{\sigma}$   $e$  in
    ④ let*  $\bar{\theta}' = \bar{\theta} [r \leftarrow \bar{v}]$  in
    ⑤ consume_asrt  $\bar{\theta}'$   $\bar{\sigma}'$   $Q_{\underline{o}}$ 
  in
   $\forall (o_l : \_ \mid \_) \in \text{all\_branches}. o_l = \text{Ok}$ 
```

To verify that the function with identifier f in program γ satisfies its specification in Γ , step ① creates a substitution that maps each formal argument and free variable of the pre-condition (implicitly universally quantified) to a fresh unconstrained symbolic variable. In step ②, a state $\bar{\sigma}$ is obtained by producing the pre-condition P in the empty state 0. Combined with ①, this step ensures that, if the function successfully verifies, it does so *for all* states that satisfy P . The function body is then executed in step ③, using the initial substitution $\bar{\theta}$ and $\bar{\sigma}$, yielding an outcome $\underline{o} \in \{\text{Ok}, \text{Err}\}$, a result value \bar{v} and an outcome state $\bar{\sigma}'$ ³. Step ④ extends the initial substitution $\bar{\theta}$ with variable r and sets its value to \bar{v} , yielding $\bar{\theta}'$. Using this substitution, step ⑤ matches each resulting branch against the corresponding post-condition Q_{Ok} or Q_{Err} ,

```
void abs(int* x) {
  if (*x < 0) {
    *x = -(*x)
  }
}

int main() {
  int* x = alloc();
  *x = nondet();
  abs(x);
  assert(*x >= 0);
  return 0;
}
```

Figure 7.1: Symbolic test harness for testing the in-place `abs` function

```
main(){
  let x = <alloc>() in
  let v = <nondet>() in
  let _ = <store>(x, v) in
  let _ = abs(x) in
  let y = <load>(x) in
  let _ = <assert>(y >= 0) in
  0
}
```

Figure 7.2: Main function from [Figure 7.1](#) compiled to SIGIL

³ Here, the `let*` operator handles and propagates logical and missing errors, but exposes `Ok` and `Err` outcomes.

depending on its outcome. Verification is successful if all branches terminate without logical failure or missing outcome.

Example Let us provide the detailed steps of verifying the following valid specification⁴ of the `abs` function from Figure 7.1:

$$\{ x \mapsto y \} \text{abs}(x) \{ \text{Ok} : r. \exists z. (r = ()) * x \mapsto z * z \geq 0 \}$$

The substitution $\bar{\theta} = [x \mapsto \bar{x}, y \mapsto \bar{y}]$ created in step ① assigns fresh symbolic variables to the formal argument `x` and the free variable `y`.

Producing the pre-condition in step ② yields a single branch with state $\bar{\sigma} = [\bar{x} \mapsto \bar{y}]$. The producer of the points-to assertion does not enforce any new constraints when the heap is empty before production, and therefore the resulting path condition is `true`:

$$\text{produce_asrt } \bar{\theta} \ 0 \ (x \mapsto y) = \{ \langle [\bar{x} \mapsto \bar{y}] \mid \text{true} \rangle \}$$

Figure 7.3 shows the symbolic execution tree obtained by symbolically executing the `abs` function starting from $\bar{\sigma}$ and the substitution $\bar{\theta}$. We annotate each node in the tree with the evaluated expression, as well as the heap and path condition before execution of the expression.

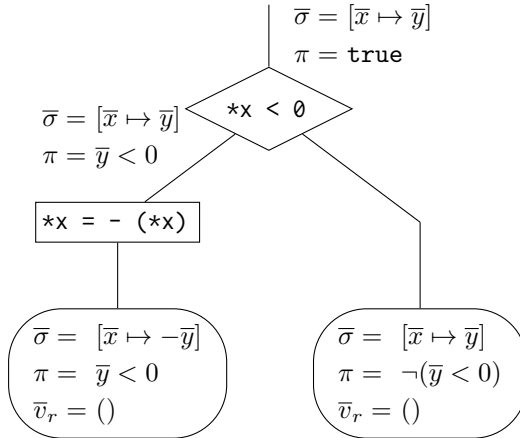


Figure 7.3: Execution tree for the `abs` function starting from

$$\begin{aligned} \bar{\theta} &= [x \mapsto \bar{x}] \\ \bar{\sigma} &= [\bar{x} \mapsto \bar{y}] \\ \pi &= \text{true} \end{aligned}$$

The tree corresponds to a view of the execution of the C code instead of the corresponding SIGIL code. The latter would yield a big tree that is harder to read without providing more information.

The execution above terminates successfully with two branches: if \bar{y} was less than 0, the value at address \bar{x} is updated to its negation, and otherwise, the heap is left unchanged. In both cases, the return value is $\bar{v}_r = ()$, an interpretation of the C `void` type. \bar{v}_r is bound to the return variable `r` during step ④, resulting in the substitution $\bar{\theta}' = \theta[r \leftarrow ()]$.

Finally, step ⑤ consumes the post-condition from the final state using the substitution $\bar{\theta}'$.

We only go through the matching process for the branch where $\bar{y} < 0$, as the other branch works analogously. First, the following matching plan is created:

$$\text{plan}(Q, \bar{\theta}') = [\textcircled{a} (r = (), []) \\ \textcircled{b} (x \mapsto z, [z \leftarrow O_0]) \\ \textcircled{c} (z \geq 0, [])]$$

In step \textcircled{a} of the matching plan, the pure assertion $r = ()$ is consumed independently of the path condition as $\bar{\theta}'[r] = ()$.

⁴ We hide the calls to the `PointsTo` and `Pure` core predicates behind straightforward syntactic sugar.

In step \textcircled{b} , the assertion $\mathbf{x} \mapsto \mathbf{z}$ is consumed. Before consumption, the variable \mathbf{x} is substituted to the symbolic value $\bar{\theta}'[\mathbf{x}] = \bar{x}$. Independently of the path condition, $\bar{x} \mapsto -\bar{y}$ is the only cell at address \bar{x} , and it is successfully removed from the heap, yielding the new heap $\bar{\sigma} = \emptyset$. In addition, the consumer returns the value $-\bar{y}$, which is temporarily bound to the placeholder variable \mathbf{O}_0 . In turn, following the matching plan, the value of \mathbf{O}_0 is bound to \mathbf{z} , resulting in the substitution:

$$\bar{\theta}' = [\mathbf{x} \mapsto \bar{x}, \mathbf{y} \mapsto \bar{y}, \mathbf{r} \mapsto (), \mathbf{z} \mapsto -\bar{y}]$$

Finally, in step \textcircled{c} , the assertion $\mathbf{z} \geq 0$ is consumed. Since $\bar{\theta}'[\mathbf{z}] = -\bar{y}$, and $\pi \models -\bar{y} \geq 0$, consumption is performed successfully.

The verification is successful since the post-condition has been successfully consumed by all branches.

Soundness We show the verification procedure to be sound. The corresponding theorem below states that if a procedure is verified, then it is valid with respect to its specification. The theorem applies if the function is recursive and the specification is used to prove itself. Note that we do not prove that each function satisfies its specification in the classical way but in the *intuitionistic* way (denoted \models^I), where resources are allowed to leak:

$$\gamma \models^I \{ P \} f(\vec{x}) \{ o : r. Q \} \iff \gamma \models \{ P \} f(\vec{x}) \{ o : r. Q * \text{True} \}$$

Gillian, at the moment, does not implement any check to ensure the absence of resource leaks, while, for instance, VeriFast does⁵.

Theorem 7.1 (Compositional verification: soundness).

Let $d = f(\vec{x}) \{e\}$ be a function definition, $s = \{ P \} f(\vec{x}) \{ o : r. Q \}$ be an OX function specification for f , and $\models^I (\gamma, \Gamma)$ be an OX-valid environment, where $f \notin \text{dom}(\gamma)$. Let $\gamma' = \gamma[f \leftarrow d]$ and $\Gamma' = \Gamma[f \leftarrow s]$. If $\text{verify } \Gamma' \gamma' f s = \text{true}$, then $\models^I (\gamma', \Gamma')$

⁵ We could assume the existence of a theoretical leak check to prove the theorem, but decide to be closer to the actual Gillian implementation.

7.3 Automatic UX specification synthesis

As opposed to the previous two analyses, automatic UX specification synthesis does not require any effort from the developer to set up symbolic harnesses or write specifications. Instead, a technique called *bi-abduction* is applied to automatically infer a set of under-approximating specifications for each function in the program.

This analysis also leverages the function-compositionality of the symbolic specification semantics, allowing each function to be analysed in isolation. If the synthesis has previously been used to produce a set of specifications and the code is modified, the analysis does not need to be performed again on the entirety of the code base. Instead, only the functions that have changed and those that depend on them must be re-analysed.

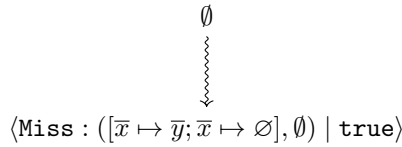
In this section, we describe a state model transformer that enhances any existing state model with the ability to perform bi-abduction, as long as the input state model provides *fixes* for the missing errors.

7.3.1 Fix-from-error bi-abduction

Bi-abduction is a technique that enables incremental discovery of the resources needed to execute a given piece of code. It was first introduced in the over-approximating setting⁶ and later formed the basis of the bug-finding tool⁷ Infer⁸. More recently, it was ported to the UX setting of true bug-finding, yielding a new tool called Infer:Pulse⁹, guaranteeing the absence of false positives.

We use a formalisation of bi-abduction inherited from JaVerT 2.0¹⁰, which we call *fix-from-error bi-abduction*, and adapt it to the parametric and UX setting. For this approach, we require the state models to satisfy an additional condition: missing outcomes obtained from action evaluation and consumption must carry a list of *fixes*. Each of these fixes is a symbolic assertion, that is, an assertion that uses symbolic variables instead of standard variables. These fixes correspond to additional resources that enable execution to continue. Then, when a missing outcome is encountered during bi-abductive execution, the fix is applied to the current state, and execution is restarted.

Example Consider evaluating the expression $\langle \text{store} \rangle(x, z)$, starting from substitution $\bar{\theta} = [x \mapsto \bar{x}, z \mapsto \bar{z}]$ and empty heap $\bar{\sigma} = \emptyset$. Evaluation yields a single missing outcome depicted in Figure 7.4, where the missing outcomes carry two fixes.



⁶ Calcagno et al., “Compositional shape analysis by means of bi-abduction”, 2009 [Cal+09]

⁷ Infer would perform an over-approximating bi-abduction before using heuristics to filter out what seemed to be false positives. The result was a bug finder that guaranteed neither the absence of false positives nor false negatives, though it was still successful.

⁸ Calcagno et al., “Infer: An Automatic Program Verifier for Memory Safety of C Programs”, 2011 [CD11]

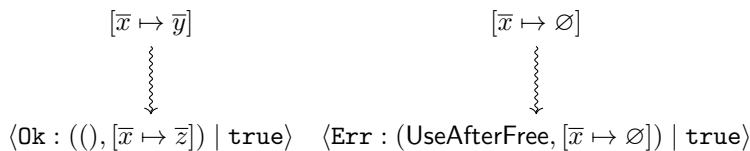
⁹ Raad et al., “Local Reasoning About the Presence of Bugs: Incorrectness Separation Logic”, 2020 [Raa+20]; and Le et al., “Finding real bugs in big programs with incorrectness logic”, 2022 [Le+22]

¹⁰ Fragoso Santos et al., “JaVerT 2.0: compositional symbolic execution for JavaScript”, 2019 [Fra+19]

Figure 7.4: Symbolic execution process resulting from the evaluation of the action $\langle \text{store} \rangle(\bar{x}, \bar{z})$ in the empty heap.

The two fixes are the symbolic assertions $\bar{x} \mapsto \bar{y}$ (where \bar{y} is a fresh symbolic variable) and $\bar{x} \mapsto \emptyset$. By providing these fixes, the state model suggests that, by adding either a valid allocated cell or a freed cell at address \bar{x} , execution would continue without missing error.

Both symbolic assertions are produced¹¹ in the original heap $\bar{\sigma}$ to restart execution. Doing so yields two new executions, depicted in Figure 7.5.



¹¹ Production of symbolic assertion is defined similarly to normal assertions, but does not require a substitution.

Figure 7.5: Executions obtained from applying the fixes obtained in Figure 7.4

On the left, the execution obtained by successfully applying the valid-cell fix terminates, having updated its content to \bar{z} . On the right, the execution obtained by applying the freed-cell fix terminates erroneously with a use-after-free error.

These two executions can then be transformed into valid under-

approximate specifications for the $\langle \text{store} \rangle(x, z)$ expression:

$$\begin{array}{cc} \begin{array}{c} [x \mapsto y] \\ \langle \text{store} \rangle(x, z) \\ [Ok : r. x \mapsto z * (r = ())] \end{array} & \begin{array}{c} [x \mapsto \emptyset] \\ \langle \text{store} \rangle(x, z) \\ [Err : r. x \mapsto \emptyset * \\ (r = \text{UseAfterFree})] \end{array} \end{array}$$

7.3.2 Bi-abductive state model transformer

Instead of keeping track of several executions simultaneously, it is possible to perform a single bi-abductive execution by keeping track of both the current state and the *anti-frame*, the list of fixes applied for this path. The above example becomes a single execution, depicted in Figure 7.6.

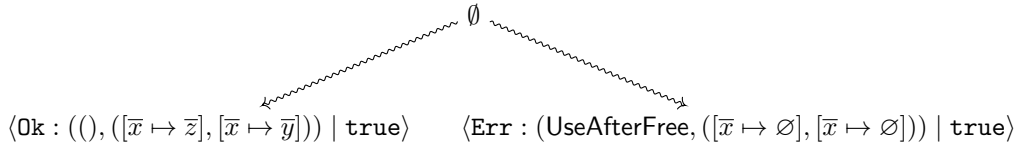


Figure 7.6: An example of bi-abductive execution

Bi-abductive execution returns two branches, each carrying an outcome, a return value, a pair composed of the final heap and the anti-frame, and a path condition.

Note that the signature of bi-abductive execution is much like that of normal symbolic execution, except that it involves a pair of a state and an anti-frame instead of a single state. This similarity suggests the potential to create a bi-abductive state model transformer that takes a state model and produces a new state model for bi-abductive execution. We call this transformer *Bi*.

Given a symbolic state model \bar{S} , the bi-abductive state model $\text{Bi}(\bar{S})$ has the same set \mathcal{A} of actions and Δ of core predicates as \bar{S} . Each symbolic state of $\text{Bi}(\bar{S})$ is a pair that comprises a symbolic state $\bar{\sigma} \in \bar{S}.$ and a symbolic assertion \bar{A} .

When evaluating actions, a first attempt is performed using action evaluation from \bar{S} , i.e. without bi-abduction. If this fails with a **Miss** outcome, fixes are extracted from the return value using the **fixes** function¹² that must be implemented by the symbolic state model \bar{S} . We provide a simplified definition in Figure 7.7, which assumes that the **fixes** function always returns a unique fix as to avoid awkward notations. In reality¹³, the **fixes** function returns a list of fixes, each of which is applied before continuing. If the list is empty, execution vanishes, and nothing is inferred.

Note that the bi-abductive state model only makes a single attempt at fixing each error. If a fix is insufficient and leads to another missing error, no attempt is made to fix the new error to avoid infinite loops. In particular, no constraint on the **fixes** function is required to guarantee the soundness of the analysis. A fix that is too weak will lead to another missing outcome, which will terminate execution without producing any specifications for that fix. In contrast, a fix that is too strong will either yield an infeasible state, which will also terminate execution without producing any specification or will lead to a successful execution with a

¹² As opposed to the high-level description given above, we use the **fixes** function to avoid changing the signature of the action evaluator.

¹³ Its implementation can be found in the open-source Gillian repository in the BiState module.

```

module Bi ( $\bar{S}$  : sig
  include Symbolic_state_model
  val fixes :  $\bar{v} \rightarrow \overline{Asrt}_\Delta$  list
end) = struct

  type  $\bar{\Sigma} = \bar{S}.\bar{\Sigma} \times \overline{Asrt}_\Delta$ 
  let  $\bar{0} = (\bar{S}.\bar{0}, \text{emp})$ 

  type  $\mathcal{A} = \bar{S}.\mathcal{A}$ 

  let eval_action  $\alpha$  ( $\bar{\sigma}_0, \bar{A}$ )  $\vec{v} =$ 
    let* ( $o, \bar{v}, \bar{\sigma}_1$ ) =  $\bar{S}.\text{eval\_action } \alpha \bar{\sigma}_0 \vec{v}$  in
    match  $o$  with
    | Ok | Err  $\rightarrow$  ( $o, \bar{v}, (\bar{\sigma}_1, \bar{A})$ )
    | Miss  $\rightarrow$ 
      let  $[F] = \bar{S}.\text{fixes } \bar{v}$  in
      let  $\bar{\sigma}_2 = \text{produce\_asrt } \bar{S}.\text{produce } F \bar{\sigma}_1$  in
      let* ( $o', \bar{v}', \bar{\sigma}_3$ ) =  $\bar{S}.\text{eval\_action } \alpha \bar{\sigma}_2 \vec{v}$  in
      ( $o', \bar{v}', (\bar{\sigma}_3, \bar{A} * F)$ )

  ...
end

```

Figure 7.7: Formal definition of the bi-abductive state model transformer

sound but over-specified specification. We consider this a strength of our approach, as it allows for the **fixes** to be implemented incrementally by the tool developer, starting with a simple best-effort version. Over time, the quality of the fixes function may be improved, but soundness is never sacrificed.

The consumer for the bi-abductive state is defined similarly to the action evaluation, where missing errors are fixed. On the other hand, the producer cannot fail and is hence defined by simply calling the producer of the underlying state model. Both are omitted from the above definition for brevity.

7.3.3 Specification inference algorithm

The **synthesise** function, which performs specification inference, is defined in Figure 7.8. It receives four arguments: a specification context Γ ; a program γ ; the function identifier f of the function to analyse; and a partial pre-condition P , which can be used to encode statically known information about the function such as the types of the arguments. It returns a set of under-approximating specifications for the function. Below, we give a detailed description of each step of the algorithm.

Step ① creates a substitution $\bar{\theta}_{id}^{\vec{x}, \vec{y}}$ that maps each formal argument and free variable of the pre-condition P (implicitly universally quantified) to a fresh unconstrained symbolic variable of the same name. Step ② produces the partial pre-condition in the empty symbolic state $\bar{0}$ using the producer of the parameter state model \bar{S} and substitution $\bar{\theta}_{id}^{\vec{x}, \vec{y}}$. This production yields an initial state $\bar{\sigma}$, which is combined with the initial anti-frame **emp**¹⁴ to form the initial bi-abductive state. This initial bi-abductive state is used¹⁵, together with the identity substitution $\bar{\theta}_{id}^{\vec{x}, \vec{y}}$, to evaluate the function body e . This yields a set of branches of the form $\langle o : (\bar{v}, (\bar{\sigma}', \bar{A})) \mid \pi \rangle$, where $\bar{\sigma}'$ is the final state and \bar{A} is the final anti-frame.

¹⁴ The symbolic state model returned by Bi transformer supports the **emp** and **Pure** core predicates.

¹⁵ We explicitly use the bind operator of the symbolic execution monad since the **let*** operator is already used as the bind operator of the set monad.


```

let synthesise  $\gamma \Gamma f P =$ 
  let  $f(\vec{x}) \{e\} = \gamma[f]$  in
  let  $\vec{y} = \text{fv}(P)$  in
  ① let  $\bar{\theta}_{id}^{\vec{x}, \vec{y}} = [\vec{x} \mapsto \vec{x}, \vec{y} \mapsto \vec{y}]$  in
  ② let*  $\langle o : (\bar{v}, (\bar{\sigma}', \bar{A})) \mid \pi \rangle =$ 
    Symex.bind
      (produce_asrt  $\bar{S} \bar{\theta}_{id}^{\vec{x}, \vec{y}} \bar{S}.0 P$ )
      (fun  $\bar{\sigma} \rightarrow \text{eval Bi}(\bar{S}) \Gamma \gamma \bar{\theta}_{id}^{\vec{x}, \vec{y}} (\bar{\sigma}, \text{emp}) e$ )
  in
  ③ if  $o \notin \{\text{Ok}, \text{Err}\}$  then vanish else
  ④ let  $P' = P * \bar{\theta}_{id}^{\vec{x}, \vec{y}}(\bar{A})$  in
  ⑤ let  $Q = \text{to\_asrt } \bar{\theta}_{id}^{\vec{x}, \vec{y}} \bar{v} \bar{\sigma}' \pi$  in
  ⑥ return [  $P'$  ]  $f(\vec{x})$  [  $o : r. Q * R$  ]

```

If the outcome of symbolic execution is `Miss` or `Lfail`, it corresponds to a reasoning error and not to a real behaviour of the function, and it is discarded in step ③. Otherwise, step ④ computes the new pre-condition $P' = P * \bar{\theta}_{id}^{\vec{x}, \vec{y}}(\bar{A})$, where $\bar{\theta}_{id}^{\vec{x}, \vec{y}}$ is the inverse substitution that maps the symbolic variables \vec{x} and \vec{y} to the variables \vec{x} and \vec{y} . Subsequently, step ⑥ computes the post-condition by transforming the final symbolic configuration $\langle (\bar{v}, \bar{\sigma}') \mid \pi \rangle$ into an assertion¹⁶, preserving the connection between logical variables and symbolic variables specified by $\bar{\theta}_{id}^{\vec{x}, \vec{y}}$. Finally, the synthesised ISL specification [P'] $f(\vec{x})$ [$o : r. Q * R$] is returned in step ⑥.

7.3.4 Soundness

The specification inference algorithm guarantees the correctness of the under-approximate specifications it infers.

Theorem 7.2 (Specification synthesis: soundness).

$$\models (\gamma, \Gamma) \wedge [P'] f(\vec{x}) [o : r. Q * R] \in \text{synthesise } \Gamma \gamma f P \\ \implies \gamma \models [P'] f(\vec{x}) [o : r. Q * R]$$

The proof of this theorem, provided in [Appendix D.2](#), is challenging and worth discussing here. The main proof is performed by **a)** defining a concrete bi-abductive state model transformer; **b)** proving that the execution performed using this transformer satisfies the property given in [Theorem 7.3](#), described shortly; **c)** proving that the symbolic bi-abductive state model transformer is sound with respect to the concrete bi-abductive state model transformer; and **d)** proving that the synthesis algorithm is sound knowing all of the above. Here, we focus on step **b)** and the below theorem.

Theorem 7.3 (Concrete bi-abductive execution: soundness).

If $\models^{\text{ux}} (\gamma, \Gamma)$, then

$$\gamma, \Gamma \vdash (\sigma, \text{emp}), e \Downarrow_{\theta}^{\text{Bi}(\bar{S})} o : (v, (\sigma', A)) \wedge o \notin \{\text{Miss}, \text{Lfail}\} \\ \implies \exists \sigma_s. \sigma.\text{prod}_A() \rightsquigarrow \sigma_s \wedge \gamma \vdash \sigma_s, e \Downarrow_{\theta}^{\bar{S}} o : (v, \sigma')$$

This theorem states that in a valid environment (γ, Γ) , when using the specification semantics with the bi-abductive state model $\text{Bi}(\bar{S})$, if

Figure 7.8: Formal definition of the specification inference algorithm using bi-abduction. It assumes the use of a bi-abductive state model obtained through the Bi transformer. The `let*` operator corresponds to the bind operator of the set monad.

¹⁶ The existence of such a `to_asrt` function is admitted for this presentation. In practice, it should be implemented by the state model.

the evaluation of any expression e , starting from a partial¹⁷ state σ and an empty anti-frame emp , yields an outcome o a return value v , a final state σ' and an anti-frame A where o is not a reasoning error, then A is a valid fix for this execution. Specifically, there exists a *fixed* state σ_s obtained by producing A in σ , such that evaluating e using σ_s and the original state model \mathbb{S} without bi-abduction or specification execution yields the result $o : (v, \sigma')$.

¹⁷ i.e. with potentially missing resource

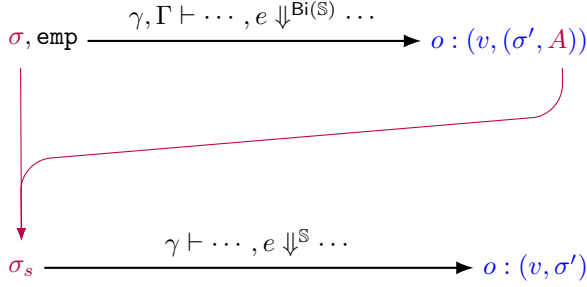


Figure 7.9: Representation of the soundness result from Theorem 7.3.

This property is easy to prove without specification calls since it holds for the base cases and action evaluation and is naturally preserved by sequential composition. However, proving that it also holds for specification calls is a substantially more complex task. The main difficulty comes from the fact that consumption algorithm does not generally preserve the required property.

Let us illustrate this difficulty using an example. Take the specification

$$[0 \mapsto 0 * 1 \mapsto 1] f() [0 \mapsto 1 * 1 \mapsto 2]$$

which is a valid ISL specification for a function f that increments the values at address 0 and 1. Assume that the initial heap is $\sigma = [0 \mapsto 0]$. Executing this specification starts by consuming the pre-condition, which we describe step by step:

- ① The first cell assertion $0 \mapsto 0$ is consumed in heap σ , yielding the new heap $\sigma' = \emptyset$.
- ② The second cell is consumed in σ' , which yields a missing outcome.
- ③ The linear heap state model suggests a fix. It can, for instance, suggest the fix $0 \mapsto 0 * 1 \mapsto 1$, which is successfully produced in σ' to obtain $\sigma'' = [0 \mapsto 0, 1 \mapsto 1]$. Unfortunately, this fix is invalid, as it duplicates the cell at address 0. However, since the cell at this address has already been consumed in step ①, there is no *local* way of detecting, at this step, that this fix is invalid.
- ④ The cell assertion $1 \mapsto 1$ is consumed, yielding a new heap $\sigma_f = [0 \mapsto 0]$, and the final anti-frame $A = 0 \mapsto 0 * 1 \mapsto 1$.

This consumption does not satisfy the property specified by Theorem 7.3 since producing the anti-frame A in the pre-condition does not yield a fixed state σ_s that can be used to replay the successful execution. Specifically, A is not disjoint from the initial heap σ , $\neg(A \# \sigma)$ ¹⁸, and thus its production would vanish. Fortunately, this issue is temporary: producing the post-condition $0 \mapsto 1 * 1 \mapsto 2$ in σ_f will also vanish. Since the theorem requires the whole execution to terminate, this behaviour does not break soundness for the bi-abductive execution of the

¹⁸ We use the assertion A as a state here, since assertions in UX must be strictly exact, and therefore corresponds to a unique state, σ_A .

specification as a whole.

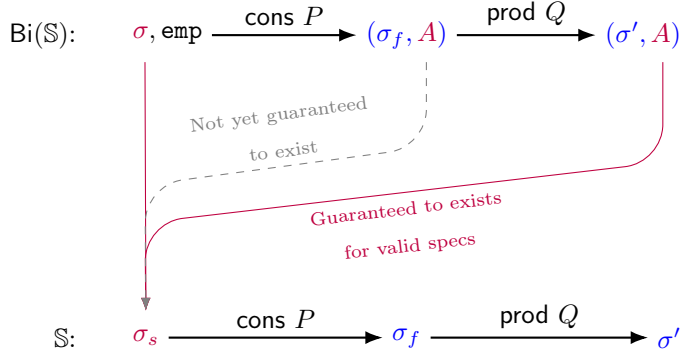


Figure 7.10: The soundness result may not hold in the temporary frame state during specification execution but is guaranteed to be recovered.

The guarantee that production of the post-condition will vanish when invalid fixes are generated during consumption of the pre-condition is crucial for the soundness of the algorithm and holds for any valid ISL specification. Proving this guarantee, however, constitutes the main challenge of the proof of [Theorem 7.3](#).

Let us discuss this challenge further by looking at the Hasse diagram of the states involved, provided in [Figure 7.11](#). Starting from σ and consuming the pre-condition P while performing bi-abduction yields the state σ_f and anti-frame A . The final state σ' is then obtained by producing the post-condition Q in σ_f . We are trying to prove the existence of the state σ_s that is obtained from producing A in σ and lets us consume P to obtain σ_f .

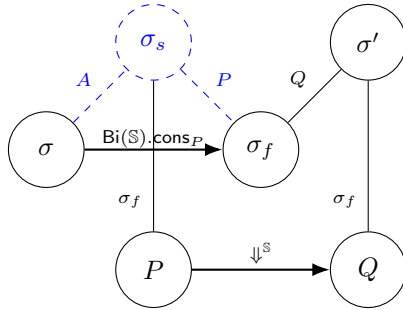


Figure 7.11: Hasse diagram of the states involved in the proof. The existence of $\sigma_s = \sigma \cdot A = \sigma_f \cdot P$ is to be proven. We highlight the corresponding parts in blue.

Since we are executing a valid ISL specification, and Q has a model (since it was produced in σ_f), we know that there exists a state in P and a valid “vanilla” execution (i.e. without specification execution or bi-abduction) of the function body from P to Q . Since this vanilla semantics satisfies [Frame addition](#), and Q is disjoint from σ_f , we learn that P must be disjoint from σ_f and there is a similar valid execution from $\sigma_f \cdot P = \sigma_s$ to $\sigma_f \cdot Q = \sigma'$. The updated Hasse diagram is shown in [Figure 7.12](#).

We have successfully reduced the scope of the problem, although the fact that $\sigma \cdot A = \sigma_s$ remains to be proven.

Next, let us inspect in more detail how σ_f is obtained from σ and P during bi-abductive consumption. We call P_1, \dots, P_n the core predicates of P , and A_1, \dots, A_n the fixes obtained when consuming each core predicate of P ¹⁹

At each step of consumption, the current state σ_i is fixed with

¹⁹ When P_i can be consumed without any fix, we can simply set $A_i = \text{emp}$.

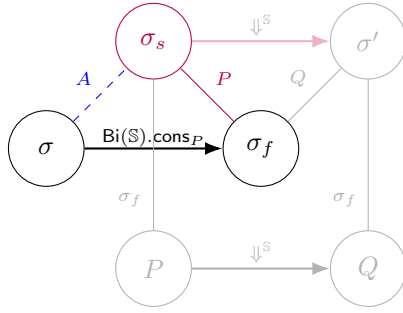


Figure 7.12: Updated Hasse diagram. The information learned so far is in **purple**, and the part left to prove is in **blue**. The scope of the problem has been reduced, and the parts that we do not need anymore are transparent.

A_{i+1} , before being used to consume P_{i+1} , yielding a new state σ_{i+1} , as represented in Figure 7.13.

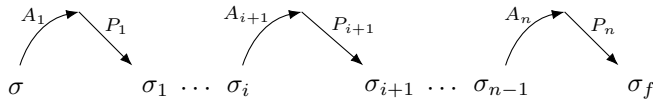


Figure 7.13: Consumption of a core predicate P_i in the state σ_i with the fix A_i yields the state σ_{i+1} .

The last key step of the proof consists of using the fact that $\sigma_s = P \cdot \sigma_f$ to prove by induction that these operations can all be re-ordered to obtain σ_s by producing each if A_i in σ . A visual representation of the induction hypothesis after the last step is shown in Figure 7.14. This completes the last missing connection in Figure 7.12 and hence concludes the proof.

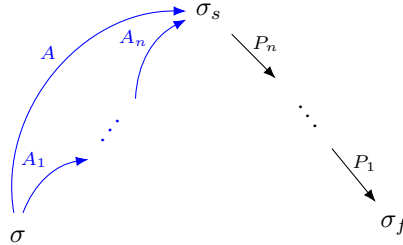


Figure 7.14: Operations can be proven to commute under the assumption that P is disjoint from σ_f .

Chapter 8

Constructing state models

The framework we have described so far enables compositional symbolic execution and various kinds of analyses, requiring only the implementation of a symbolic state model. Unfortunately, the design of state models can blow up in complexity for real-world languages. In particular, since state models are responsible for a large fragment of the semantics, an equally large part of the soundness proof is delegated to the designer of the state model.

Thankfully, state models are often made of several components that are independent of each other. In addition, state models for various languages often share common structures. For instance, while C and JavaScript lie on opposite ends of the abstraction spectrum, their heaps are both modelled as partial finite maps from object location to object, and only the notion of object differs.



Drawing inspiration from the resource algebra (RA) constructions in Iris¹, we define state model constructions that can be reused and composed together to form more complex state models. These constructions are designed to preserve correctness properties from their parameters, thereby largely alleviating the burden of proof from the designer. In addition, some of the state models we propose are directly adapted from Iris resource algebras, which is helpful to encode concepts extracted from projects that make use of Iris, as is the case for several components of the Gillian-Rust state model presented in [Part III](#).

¹ described in Section 4.1 of Jung’s thesis [\[Jun20\]](#)

Comparison with Iris The constructions presented in this chapter are twofoldly novel compared with the similar ones in Iris.

First, state models are formulated using symbolic actions, consumers and producers, enabling their use within semi-automated verification tools. As such, we can discuss various implementations that consider trade-offs between efficiency, approximation, and predictability. In addition, it means that these constructions could potentially be imported into other tools such as VeriFast or Viper².

² Although, to our knowledge, it is not a current goal of the VeriFast project.

Second, the constructions are designed and proven to be compatible with both  and  reasoning, while Iris constructions are only designed to be compatible with the former. In fact, a contribution of this chapter is the identification of Iris constructions that are incompatible with UX reasoning.

It would be interesting to investigate the connection between resource algebras and state models more formally in the future.

The predicate state model The predicate state model presented in this chapter fulfils a slightly different role in that it is purely designed as a trick to encode inductive predicates in the symbolic state. As such,

it has no counterpart in Iris, where inductive predicates are shallowly embedded in Coq. While its design is extracted from other existing tools, its formulation as a state model transformer is novel and hints at the possibility of encoding more complex transformers of the same nature. In particular, the guarded predicate state model transformer presented in Part III is a variation that, to our knowledge, does not exist in any other tool.

8.1 Product of state models

We begin our list of constructions with the product state model, which allows for the composition of two independent state models.

As it is the first construction to be presented, we take the pedantic approach of exhaustively defining each of its layers – full state model, compositional state model, core predicates, and symbolic state model – maximising clarity for the reader.

Full product The product of two full state models is denoted $\mathbb{S}_1 \times \mathbb{S}_2$ and formally defined in Figure 8.1. Its set of states is the cartesian product of the sets of states of \mathbb{S}_1 and \mathbb{S}_2 . All actions of \mathbb{S}_1 and \mathbb{S}_2 apply to the product, and the corresponding set is defined using a tagged union to accommodate overlapping sets. When an action from the first (resp. second) state model is applied, its effect is projected on the first (resp. second) component of the state, leaving the other component unaffected.

```

module  $\mathbb{S}_1 \times \mathbb{S}_2$  : Full_state_model = struct

  type  $\Sigma$  =  $\mathbb{S}_1.\Sigma \times \mathbb{S}_2.\Sigma$ 
  type  $\mathcal{A}$  = | A1 of  $\mathbb{S}_1.\mathcal{A}$  | A2 of  $\mathbb{S}_2.\mathcal{A}$ 

  let eval_action  $\alpha$  ( $\sigma_1$ ,  $\sigma_2$ )  $\vec{v}$  =
    match  $\alpha$  with
    | A1  $\alpha_1$   $\rightarrow$ 
      let* ( $\varrho, v, \sigma'_1$ ) =  $\mathbb{S}_1.\text{eval\_action}$   $\alpha_1$   $\sigma_1$   $\vec{v}$  in
      return ( $\varrho, v, (\sigma'_1, \sigma_2)$ )
    | A2  $\alpha_2$   $\rightarrow$ 
      let* ( $\varrho, v, \sigma'_2$ ) =  $\mathbb{S}_2.\text{eval\_action}$   $\alpha_2$   $\sigma_2$   $\vec{v}$  in
      return ( $\varrho, v, (\sigma_1, \sigma'_2)$ )
end

```

Figure 8.1: Formal definition of the product of two full state models \mathbb{S}_1 and \mathbb{S}_2 . The `let*` operator corresponds to the bind operation of the non-determinism monad (it exposes the returned outcomes).

Compositional product The product of two compositional state models is defined identically to the product of the full state models, i.e., by substituting \mathbb{S}_1 and \mathbb{S}_2 by \mathbb{S}_1 and \mathbb{S}_2 in the above definition.

The only additional definition required is that of the composition operator for the product states. It is straightforwardly defined as follows and is undefined if any of the components is undefined:

$$(\sigma_1, \sigma_2) \cdot (\sigma'_1, \sigma'_2) = (\sigma_1 \cdot \sigma'_1, \sigma_2 \cdot \sigma'_2)$$

Equipped with this operator, it can be trivially proven that $(\mathbb{S}_1.\Sigma \times \mathbb{S}_2.\Sigma, (\mathbb{S}_1.0, \mathbb{S}_2.0), \cdot)$ forms a PCM.

The first major lemma to be proven about the compositional product is preserves m -soundness and m -frame preservation.

Lemma 8.1 (Product state model: concrete soundness).

If $\underline{S}_1 \stackrel{c}{\sim}_m S_1$ according to \models_c^1 and $\underline{S}_2 \stackrel{c}{\sim}_m S_2$ according to \models_c^2 , then $(\underline{S}_1 \times \underline{S}_2) \stackrel{c}{\sim}_m (S_1 \times S_2)$ according to $\models_c^{1 \times 2}$ where

$$(\underline{\sigma}_1, \underline{\sigma}_2) \models_c^{1 \times 2} (\sigma_1, \sigma_2) \Leftrightarrow \underline{\sigma}_1 \models_c^1 \sigma_1 \wedge \underline{\sigma}_2 \models_c^2 \sigma_2$$

In mathematical notations:

$$\underline{S}_1 \stackrel{c}{\sim}_m S_1 \wedge \underline{S}_2 \stackrel{c}{\sim}_m S_2 \Rightarrow (\underline{S}_1 \times \underline{S}_2) \stackrel{c}{\sim}_m (S_1 \times S_2)$$

Core predicates Similarly to actions, any core predicate that applies to one of the components applies to the product. The product uses the tagged union `type` $\Delta = \mid c1 \text{ of } S_1.\Delta \mid c2 \text{ of } S_2.\Delta$ to accommodate overlapping sets. The satisfiability relation is projected on the appropriate component as follows:

$$\begin{aligned} (\sigma_1, \sigma_2) \models \langle C1 \ \delta_1 \rangle(\vec{v}_i; \vec{v}_o) &\Leftrightarrow \sigma_1 \models \langle \delta_1 \rangle(\vec{v}_i; \vec{v}_o) \wedge \sigma_2 = 0 \\ (\sigma_1, \sigma_2) \models \langle C2 \ \delta_2 \rangle(\vec{v}_i; \vec{v}_o) &\Leftrightarrow \sigma_1 = 0 \wedge \sigma_2 \models \langle \delta_2 \rangle(\vec{v}_i; \vec{v}_o) \end{aligned}$$

If both sets of core predicates are strictly exact, then the core predicates of the product are also strictly exact, preserving usability in the context of UX analysis.

Lemma 8.2 (Product state model: strict exactness).

If all core predicates of S_1 and S_2 are strictly exact, then all core predicates of $S_1 \times S_2$ are strictly exact.

Producers and consumers The producer and consumer function for the compositional product, formally defined in Figure 8.2, are also obtained by projecting the operation on the appropriate component of the state depending on which state model the core predicate belongs to (similarly to action evaluation).

Lemma 8.3 (Product state model: producers and consumers).

The producer and consumer of the product compositional state models are valid according to Definition 5.5 and Definition 5.7.

Symbolic product The last layer to describe is the product of the symbolic state models. The formal definition is provided in Figure 8.3. Again, the set of states is the cartesian product of the sets of states of the two symbolic state models. The actions and core predicates are the same as in the compositional product. Action evaluation, producers and consumers are defined by projecting the operation to the appropriate component of the product.

Furthermore, the interpretation of the symbolic states is defined as:

$$\varepsilon, (\sigma_1, \sigma_2) \models (\bar{\sigma}_1, \bar{\sigma}_2) \Leftrightarrow \varepsilon, \sigma_1 \models \bar{\sigma}_1 \wedge \varepsilon, \sigma_2 \models \bar{\sigma}_2$$

Put together, it is now possible to prove that the product symbolic state model preserves the m -soundness of its parameters, for $m \in \{OX, UX\}$.

Lemma 8.4 (Product state model: symbolic soundness).

$$\bar{S}_1 \stackrel{s}{\sim}_m S_1 \wedge \bar{S}_2 \stackrel{s}{\sim}_m S_2 \Rightarrow (\bar{S}_1 \times \bar{S}_2) \stackrel{s}{\sim}_m (S_1 \times S_2)$$

```

module  $\mathbb{S}_1 \times \mathbb{S}_2$  : Compositional_state_model = struct

  (* ... *)

  let 0 = ( $\mathbb{S}_1.0$ ,  $\mathbb{S}_2.0$ )

  type  $\Delta$  = | C1 of  $\mathbb{S}_1.\Delta$  | C2 of  $\mathbb{S}_2.\Delta$ 

  let produce  $\delta$  ( $\sigma_1$ ,  $\sigma_2$ )  $\vec{v}_i$   $\vec{v}_o$  =
    match  $\delta$  with
    | C1  $\delta_1 \rightarrow$ 
      let*  $\sigma'_1 = \mathbb{S}_1.\text{produce } \sigma_1 \delta_1 \vec{v}_i \vec{v}_o$  in
      return ( $\sigma'_1, \sigma_2$ )
    | C2  $\delta_2 \rightarrow$ 
      let*  $\sigma'_2 = \mathbb{S}_2.\text{produce } \sigma_2 \delta_2 \vec{v}_i \vec{v}_o$  in
      return ( $\sigma_1, \sigma'_2$ )

  let consume  $\delta$  ( $\sigma_1$ ,  $\sigma_2$ )  $\vec{v}_i$  =
    match  $\delta$  with
    | C1  $\delta_1 \rightarrow$ 
      let* ( $o_l, \vec{v}_o, \sigma'_1$ ) =  $\mathbb{S}_1.\text{consume } \sigma_1 \delta_1 \vec{v}_i$  in
      return ( $o_l, \vec{v}_o, (\sigma'_1, \sigma_2)$ )
    | C2  $\delta_2 \rightarrow$ 
      let* ( $o_l, \vec{v}_o, \sigma'_2$ ) =  $\mathbb{S}_2.\text{consume } \sigma_2 \delta_2 \vec{v}_i$  in
      return ( $o_l, \vec{v}_o, (\sigma_1, \sigma'_2)$ )
end

```

Figure 8.2: Formal definition of the consumer and producer for the product of \mathbb{S}_1 and \mathbb{S}_2 . The `let*` operator corresponds to the bind operation of the non-determinism monad (it exposes the returned outcomes).

```

module  $\bar{\mathbb{S}}_1 \bar{\times} \bar{\mathbb{S}}_2$  : Symbolic_state_model = struct

  type  $\bar{\Sigma} = \bar{\mathbb{S}}_1.\bar{\Sigma} \times \bar{\mathbb{S}}_2.\bar{\Sigma}$ 
  type  $\bar{\mathcal{A}} = | A1 \text{ of } \bar{\mathbb{S}}_1.\bar{\mathcal{A}} | A2 \text{ of } \bar{\mathbb{S}}_2.\bar{\mathcal{A}}$ 
  val  $\bar{0} = (\bar{\mathbb{S}}_1.\bar{0}, \bar{\mathbb{S}}_2.\bar{0})$ 

  let eval_action  $\alpha$  ( $\bar{\sigma}_1$ ,  $\bar{\sigma}_2$ )  $\vec{v} =$ 
    match  $\alpha$  with
    | A1  $\alpha_1 \rightarrow$ 
      let* ( $o, \vec{v}', \bar{\sigma}'_1$ ) =  $\bar{\mathbb{S}}_1.\text{eval\_action } \alpha_1 \bar{\sigma}_1 \vec{v}$  in
      return ( $o, \vec{v}', (\bar{\sigma}'_1, \bar{\sigma}_2)$ )
    | A2  $\alpha_2 \rightarrow$ 
      let* ( $o, \vec{v}', \bar{\sigma}'_2$ ) =  $\bar{\mathbb{S}}_2.\text{eval\_action } \alpha_2 \bar{\sigma}_2 \vec{v}$  in
      return ( $o, \vec{v}', (\bar{\sigma}_1, \bar{\sigma}'_2)$ )

  type  $\Delta = | C1 \text{ of } \bar{\mathbb{S}}_1.\Delta | C2 \text{ of } \bar{\mathbb{S}}_2.\Delta$ 

  let produce  $\delta$  ( $\bar{\sigma}_1$ ,  $\bar{\sigma}_2$ )  $\vec{v}_i$   $\vec{v}_o =$ 
    match  $\delta$  with
    | C1  $\delta_1 \rightarrow$ 
      let*  $\bar{\sigma}'_1 = \bar{\mathbb{S}}_1.\text{produce } \bar{\sigma}_1 \delta_1 \vec{v}_i \vec{v}_o$  in
      return ( $\bar{\sigma}'_1, \bar{\sigma}_2$ )
    | C2  $\delta_2 \rightarrow$ 
      let*  $\bar{\sigma}'_2 = \bar{\mathbb{S}}_2.\text{produce } \bar{\sigma}_2 \delta_2 \vec{v}_i \vec{v}_o$  in
      return ( $\bar{\sigma}_1, \bar{\sigma}'_2$ )

  let consume  $\delta$  ( $\bar{\sigma}_1$ ,  $\bar{\sigma}_2$ )  $\vec{v}_i =$ 
    match  $\delta$  with
    | C1  $\delta_1 \rightarrow$ 
      let* ( $o_l, \vec{v}_o, \bar{\sigma}'_1$ ) =  $\bar{\mathbb{S}}_1.\text{consume } \bar{\sigma}_1 \delta_1 \vec{v}_i$  in
      return ( $o_l, \vec{v}_o, (\bar{\sigma}'_1, \bar{\sigma}_2)$ )
    | C2  $\delta_2 \rightarrow$ 
      let* ( $o_l, \vec{v}_o, \bar{\sigma}'_2$ ) =  $\bar{\mathbb{S}}_2.\text{consume } \bar{\sigma}_2 \delta_2 \vec{v}_i$  in
      return ( $o_l, \vec{v}_o, (\bar{\sigma}_1, \bar{\sigma}'_2)$ )
end

```

Figure 8.3: Formal definition of the symbolic product state model. The `let*` operator corresponds to the bind operation in the symbolic execution monad as defined in Definition 6.14 (it exposes the outcomes).

8.2 Exclusive ownership

We now present a state model that is directly inspired by an Iris construction. The $\text{Exc}(\tau)$ state model is parametric on a sort $\tau \subseteq \text{Val}$, and its compositional states capture exclusive ownership of a value of that sort. For the rest of this chapter, we take a less pedantic approach and provide fewer details about each construction. All definitions and proofs are available in full in [Appendix E](#).

Full state model of values For this explanation, it is worth introducing the corresponding full state model: the *state model of values* for the sort τ , denoted $\mathbb{V}(\tau)$. Each state in that model is a single value of sort τ , which can be read or updated using the `load` and `store` actions:

```
type  $\Sigma = \tau$ 
let load  $\sigma = \text{ok } (\sigma, \sigma)$ 
let store  $\_ \sigma' = \text{ok } ((), \sigma')$ 
```

$\mathbb{V}(\tau)$ itself does not capture any notion of ownership. In fact, ownership is a property expressed using composition. In particular, *exclusive ownership* of a resource means that two occurrences of that resource may not be composed since only one occurrence may exist simultaneously.

Exclusive compositional state model This is precisely how the exclusive state model is defined. Its state fragments are defined as either a value of sort τ or the special missing value \perp : `type $\Sigma = \tau \mid \perp$` . Its composition is defined such that a state that owns a value may only be composed with a state that does not own one:

$$\sigma \cdot \sigma' = \begin{cases} \sigma & \text{if } \sigma' = \perp \\ \sigma' & \text{if } \sigma = \perp \\ \text{undefined} & \text{otherwise} \end{cases}$$

Loading and storing the value requires its ownership. Therefore, the compositional load actions is defined as follows, yielding a missing outcome when the state is \perp . The store action is similar and elided.

```
let load  $\sigma^\perp =$ 
  match  $\sigma^\perp$  with
  |  $\perp \rightarrow \text{miss } (\text{MissingValue}, \sigma)$ 
  |  $\sigma \rightarrow \text{ok } (\sigma, \sigma)$ 
```

It can be trivially checked that this definition is *m*-sound with respect to the full state model of values $\mathbb{V}(\tau)$, using equality for the relation \models_{c} .

Core predicate We define a unique core predicate Exc , with no in-parameter and a single out-parameter corresponding to the value of the state.

$$\sigma \models \langle \text{Exc} \rangle (; v) \Leftrightarrow \sigma = v$$

The corresponding producer and consumer are straightforwardly defined such that consuming yields a missing error if the state does not own a value and otherwise removes and returns it. The producer van-

ishes if the value is already owned, as the composition is undefined.

```

let consume_cell  $\sigma$  =           let produce_cell  $\sigma$   $v$  =
  match  $\sigma$  with                 match  $\sigma$  with
  |  $\perp$   $\rightarrow$  miss (MissingState,  $\sigma$ )   |  $\perp$   $\rightarrow$  return  $v$ 
  |  $\sigma \rightarrow$  ok ( $\sigma$ ,  $\perp$ )             |  $\_$   $\rightarrow$  vanish

```

Symbolic state model The symbolic exclusive state model is straightforwardly obtained by lifting the concrete implementation. It is only worth noting that the ‘optional emptiness’ is not made symbolic. Instead, the set of symbolic states is defined as `type $\bar{\Sigma} = \bar{\tau} \mid \perp$` . This permits efficient OCaml-native pattern-matching, and no solver has to be queried to determine if a value is missing.

Applications The exclusive state model has little applicability on its own. It is only designed to be composed with other state models. For example, a simplified linear heap can be obtained by using the exclusive state model as the codomain of a partial finite map transformer, defined later in §8.5.

8.3 Agreement state model

The agreement state model, also directly inspired by an Iris construction, allows for *immutable sharing* of a value. Akin to the exclusive state model, the agreement state model is parametric on a sort τ and is **EX**-sound with respect to the full state model of values $\underline{\mathbb{V}}(\tau)$. Its states are values of τ or \perp , as in the exclusive state model, but the composition operator differs. Two non-missing agreement states can be composed together as long as they share the same value:

$$\sigma \cdot \sigma' = \begin{cases} \sigma & \text{if } \sigma' = \perp \\ \sigma' & \text{if } \sigma = \perp \\ \sigma & \text{if } \sigma = \sigma' \\ \text{undefined} & \text{otherwise} \end{cases}$$

This definition forms a valid partial commutative monoid but forbids implementing a store operation, as it would not be frame preserving. Therefore, the agreement state model only comes with a load action.

It can also be trivially checked that this definition is *m*-sound with respect to the full state model of values $\underline{\mathbb{V}}(\tau)$, using equality for the relation \models_{C} .

Core predicate The agreement state model has a single core predicate called **Agree**. Its satisfiability relation is the same as the **Exc** core predicate:

$$\sigma \models \langle \text{Agree} \rangle (; v) \Leftrightarrow \sigma = v$$

However, since composition is defined differently, the producer and consumer functions behave differently. In particular, the producer is idempotent, while the consumer does not modify the state. These two properties are shared with the producer and consumer of all *persistent* core predicates, which are the core predicates that can be freely duplicated.

```

let consume  $\sigma$  Agree [  $v$  ] =
  match  $\sigma$  with
  |  $\perp$   $\rightarrow$  miss (MissingState,  $\sigma$ )
  |  $\sigma \rightarrow$  ok ( $\sigma$ ,  $\sigma$ )

let produce  $\sigma$  Agree [  $v$  ] =
  match  $\sigma$  with
  |  $\perp$   $\rightarrow$  return  $v$ 
  |  $\sigma \rightarrow$ 
    let* () = assume ( $\sigma = v$ ) in
    return  $\sigma$ 

```

8.4 Fractional state model

Fractional permissions³ are widely used in separation logic as a compromise between exclusive ownership and read-only sharing. At the core is the idea that one can own a fraction $0 < q \leq 1$ of a resource. Anyone who owns a fraction of the resource can read the value, but only the owner of the full resource ($q = 1$) can modify it.

State fragments in the fractional state models are then pairs of a value of sort τ and a fraction $0 < q \leq 1$, or \perp :

`type $\Sigma = (\tau \times (0, 1])^\perp$`

```

let load  $\sigma$  =
  match  $\sigma$  with
  |  $\perp$   $\rightarrow$  miss (MissingState,  $\sigma$ )
  | ( $v, q$ )  $\rightarrow$  ok ( $v$ , ( $v, q$ ))

let store  $\sigma$   $v$  =
  match  $\sigma$  with
  |  $\perp$   $\rightarrow$  miss (MissingState,  $\sigma$ )
  | ( $v', q$ )  $\rightarrow$ 
    if  $q = 1$  then ok ( $\perp$ , ( $v, 1$ ))
    else miss ((MissingFrac, 1-q),  $\sigma$ )

```

Composition is defined by adding the fractions together if the values are the same and the sum is less than or equal to 1:

$$\begin{aligned}
 \sigma \cdot \perp &= \sigma \\
 \perp \cdot \sigma &= \sigma \\
 (v, q) \cdot (v', q') &= (v, q + q') \quad \text{if } v = v' \text{ and } q + q' \leq 1 \\
 &\text{undefined} \quad \text{otherwise}
 \end{aligned}$$

Core predicate The fractional state model has a unique core predicate with one in-parameter for the fraction and one out-parameter for the value. Its satisfiability relation is defined straightforwardly as follows, while its producer and consumer are elided as they simply follow the definition of composition.

$$(v, q) \models \langle \text{Frac} \rangle (q; v)$$

Symbolic fractional state model One challenge arises from implementing a symbolic fraction state model: solvers do not generally support symbolic values of sort $(0, 1]$. To enable support for symbolic fractions, the type of symbolic states is instead defined as `type $\bar{\Sigma} = (\bar{\tau} \times \mathbb{R})^\perp$` , where the fraction is a real number. Separately, actions, producer and consumer must enforce the additional well-formedness constraint $\mathcal{W}f((\bar{v}, \bar{q})) = 0 < \bar{q} \leq 1$. To exemplify, we provide the definition of the symbolic producer for the **Frac** core predicate. First, nothing is produced if the core predicate is invalid and specifies a value outside the valid range. Then, if the state already owns a value, fractions are added if their sum is in the required range and if the values agree.

³ Bornat et al., “Permission accounting in separation logic”, 2005 [Bor+05]

```

let produce  $\sigma$  Frac [  $\bar{q}$  ] [  $\bar{v}$  ] =
  let* () = assume ( $0 < \bar{q} \leq 1$ ) in
  match  $\sigma$  with
  |  $\perp \rightarrow$  ok ( $\bar{v}$ ,  $\bar{q}$ )
  | ( $\bar{v}', \bar{q}'$ )  $\rightarrow$ 
    let* () = assume ( $\bar{v} = \bar{v}' \wedge \bar{q} + \bar{q}' \leq 1$ ) in
    ok ( $\bar{v}$ ,  $\bar{q} + \bar{q}'$ )

```

Comparison with Iris Iris offers a resource algebra called *Frac*, which only contains the fraction part of our pairs, that is, an element of $(0, 1]$. An equivalent of our fractional state model can then be implemented in Iris as $\text{Frac} \times \text{Ag}(\tau)$. A similar construction in Gillian would yield a valid state model with the same correctness properties as ours.

However, doing so would introduce two core predicates instead of one: one for the fraction and one for value. While this could be hidden behind syntactic sugar, calling two producers and consumers instead of one at each pair's occurrence could impact performances. In practice, fractions are often used in conjunction with the agreement algebra, and defining a unique state model that captures both is more convenient.

8.5 Partial finite maps

The partial finite map transformer is another fundamental building block for defining state models. All real state models ever defined using Gillian (those of Wisl, Gillian-JS, Gillian-C and Gillian-Rust) contain at least one partial map.

The partial map transformer is parametric on an indexing sort $I \subseteq \text{Val}$, and a codomain state model \mathbb{S} , and its set of full states are partial finite maps of the form $\text{type } \underline{\Sigma} = I \xrightarrow{\text{fin}} \mathbb{S}.\underline{\Sigma}$. For example, the linear heap⁴ is instantiated using natural numbers \mathbb{N} as indexing sort and exclusive ownership $\text{Exc}(\text{Val})$ as codomain.

All actions of the codomain state model apply to the partial map. They receive an additional parameter corresponding to the index, and action evaluation simply applies the codomain's action evaluation to the value at the corresponding index in the map:

```

let eval_action  $\alpha$   $\underline{\sigma}$  (i::r) =
  match  $\underline{\sigma}(i)$  with
  | None  $\rightarrow$  error (InvalidAccess,  $\underline{\sigma}$ )
  | Some  $\underline{\sigma}_c$   $\rightarrow$ 
    let* ( $v$ ,  $\underline{\sigma}'_c$ ) =  $\mathbb{S}.\text{eval\_action } \alpha \underline{\sigma}_c r$  in
    ok ( $v$ ,  $\sigma [i \leftarrow \underline{\sigma}'_c]$ )

```

Compositional states are defined as partial maps and from the index to the compositional codomain: $\text{type } \Sigma = I \xrightarrow{\text{fin}} \mathbb{S}.\Sigma$.

Recall that two linear heaps are composed using a disjoint union of the two maps. The general case is less straightforward, as the decision of how to compose resources at the same index is delegated to the codomain:

$$\sigma_1 \cdot \sigma_2 = \lambda i. \begin{cases} \sigma(i) & \text{if } i \notin \text{dom}(\sigma') \\ \sigma'(i) & \text{if } i \notin \text{dom}(\sigma) \\ \sigma(i) \cdot \sigma'(i) & \text{if } i \in \text{dom}(\sigma) \cap \text{dom}(\sigma') \wedge \sigma(i) \# \sigma'(i) \\ \text{undefined} & \text{otherwise} \end{cases}$$

⁴ We simplify and ignore the fact that cells can be freed. The *Freeable* transformer is introduced in the next section.

Similar to actions, all core predicates from the codomain can be used to characterise the partial map by receiving an additional in-parameter for the index. For example, in the case of the linear heap where the codomain is $\text{Exc}(\text{Val})$, the $\langle \text{Exc} \rangle(;v)$ core predicate is lifted to $\langle \text{Exc} \rangle(i;v)$, which we have been calling `PointsTo` until now.

The producer and consumer of the compositional state model are defined in the expected way and need not be detailed here.

Non-unicity of observably equivalent states and optimisation While the construction presented so far is correct, it potentially introduces multiple representations of the same information, which is a source of complexity. For instance, take the linear heap where the codomain is $\text{Exc}(\text{Val})$. Because \perp is a valid state in the exclusive state model, the linear heap could contain a binding $i \mapsto \perp$. In effect, such a binding is equivalent to the absence of a binding.

This phenomenon can lead to performance issues during symbolic execution. Remember from §6.8.2 that symbolic operations in the partial map must perform access by iterating over all bindings in the map. Each step of the iteration may cause a satisfiability or entailment check, which can be expensive. The presence of empty bindings in the map leads to unnecessary iterations.

Thankfully, this can be fixed by checking if the resulting state is the 0 of the co-domain if it returns true and removing the empty binding when detected.

In turn, the symbolic consumer can be updated to remove empty bindings from the map when detected:

```
let consume  $\sigma$   $\delta$   $\vec{v}_i$  =
  match  $\vec{v}_i$  with
  |  $i :: \vec{v}'_i \rightarrow$  (
    match  $\sigma(i)$  with
    | Some  $\sigma_c \rightarrow$ 
      let*  $\sigma'_c = \mathbb{S}.\text{consume } \sigma_c \delta \vec{v}'_i$  in
      if  $\sigma'_c = \mathbb{S}.0$  then ok ( $\sigma \setminus i$ )
      else ok ( $\sigma[i \leftarrow \sigma'_c]$ )
    | None  $\rightarrow$  miss (MissingBinding,  $\sigma$ )
  | _  $\rightarrow$  lfail (InvalidAccess,  $\sigma$ )
)
```

8.6 Freeable state model

The freeable state model transformer $\text{Freeable}(\mathbb{S})$ adds the ability to dispose of resources to the parameter state model \mathbb{S} . For example, the linear heap defined in earlier chapters maps addresses to memory cells, and each cell may be freed using the `free` action. Once a cell is freed, it cannot be manipulated further.

State fragments of the compositional state model $\text{Freeable}(\mathbb{S})$ are either a state fragment of \mathbb{S} , or the special freed state \emptyset :

```
type  $\Sigma = \mathbb{S}.\Sigma \mid \emptyset$ 
```

This special freed state is incompatible with any state fragment of \mathbb{S} , except for 0:

$$\emptyset \cdot \sigma = \begin{cases} \emptyset & \text{if } \sigma = 0 \\ \text{undefined} & \text{otherwise} \end{cases}$$

All actions of \mathbb{S} can be used on the freeable state model if the state is not freed. It is also possible to perform a free as long as the current state is *exclusively owned*, explained below.

```

module Freeable (S) = struct

  type Σ = S of S.Σ | ∅
  let 0 = S 0.0

  type A = A of S.A | Free

  let free σ =
    match σ with
    | ∅ → error (DoubleFree, σ)
    | S σ →
      if is_exclusively_owned σ then ok ((), ∅)
      else miss (MissingResource, σ)

  let eval_action α σ v̄ =
    match α with
    | A α → (
      match σ with
      | ∅ → error (UseAfterFree, σ)
      | S σ →
        let* (v, σ') = S.eval_action α σ v̄ in
        ok (v, S σ')
      )
    | Free → free σ

  (* ... *)
end

```

Exclusive ownership means that no frame may interfere with the state that is being freed:

$$\text{is_exclusively_owned } \sigma = \forall \sigma'. \sigma' \neq 0 \implies \neg \sigma \# \sigma'$$

For instance, using the fractional state model, one needs to own a fraction $q = 1$ of the resource to be able to free it. Similarly, in real-world languages such as C and Rust, one can only free an entire memory block at once; therefore, one must own the entirety of the block before freeing it. Without this additional condition, the free action would fail to satisfy [Frame subtraction](#).

Comparison with Iris: one-shot RA Iris has a similar construction, called the *one-shot* algebra. However, it differs from **Freeable** in several ways. First, in Iris, the one-shot algebra can be derived from a sum construction, which does not exist in the current version of Gillian. Second, in Iris, the notion of exclusive ownership is slightly simpler and does not have the pre-condition $\sigma' \neq 0$. These two differences are due to the definition of resource algebras in Iris, which is more flexible than that of PCMs and does not require a 0 element, but a *partial core function*⁵. This choice in Iris’s design is explicitly motivated by the ability to define the sum construction, which is impossible in Gillian.

Finally, in Iris, the freed resource is captured by an agreement algebra instead of exclusive ownership: $\text{OneShot} = \text{Frac} + \text{Ag}(\{\emptyset\})$. This implies that, in Iris, $\emptyset \cdot \emptyset = \emptyset$, while it is undefined in Gillian. Interestingly, this definition of composition is *incompatible with under-approximate reasoning*. In particular, it breaks the [Frame addition](#) requirement. To understand why, consider the following specification

⁵ Jung et al., “Iris from the ground up: A modular foundation for higher-order concurrent separation logic”, 2018 [[Jun+18](#)]

of the free action:

$$\{ \langle \text{Frac} \rangle(q; v) \} \text{Free}() \{ \langle \text{Freed} \rangle(;) \}$$

Applying the frame rule of separation logic to this specification and adding the frame $\langle \text{Freed} \rangle(;)$ on both sides, we obtain:

$$\{ \langle \text{Frac} \rangle(q; v) * \langle \text{Freed} \rangle(;) \} \text{Free}() \{ \langle \text{Freed} \rangle(;) * \langle \text{Freed} \rangle(;) \}$$

The pre-condition is now false, as it is impossible to own a fraction of a resource while the resource is freed, but the post-condition is not according to the rules of the agreement resource algebra. Therefore, this specification is now equivalent to:

$$\{ \text{False} \} \text{Free}() \{ \langle \text{Freed} \rangle(;) \}$$

While the above specification is sound in SL, the same reasoning is unsound in ISL since not all states satisfying the post-condition can be reached from the pre-condition.

*Iris*⁶ *sums and UX reasoning*: More generally, Iris updates that transfer the element from one variant to the other require a different property depending on the approximation mode. In the current Iris, which was designed for OX reasoning, the source element must be fully owned to ensure that (a version of) **Frame subtraction** holds. Conversely, if there was a UX Iris, where **Frame addition** must hold, the target element would need to be exclusively owned instead of the source element. Since an element of the agreement element model can never be exclusively owned, using the agreement algebra to capture freed elements is incompatible with UX reasoning.

⁶ This note is intended for readers familiar with Iris.

8.7 The predicate symbolic transformer

Verification based on separation logic is most often performed by making extensive use of inductive predicates. All semi-automated verification tools based on CSE allow users to define inductive predicates and use them in specifications. Usually, symbolic states maintain a list of *closed predicates*, which can be *unfolded* and *folded* manually by the user. In this section, we propose a *general* way of extending a symbolic state model with support for inductive predicates. Before describing this extension, we start by providing a simple example of such a predicate. Then, we justify the soundness of this approach by defining the corresponding compositional state model.

8.7.1 Example and motivation

Example: linked list Consider the linked list predicate, a staple of the separation logic literature, defined as follows in the linear heap⁷:

$$\begin{aligned} \text{list}(x, \alpha) &\triangleq (x = \text{null} * \alpha = []) \\ &\vee (\exists a, z, \beta. x \mapsto a, z * \text{list}(z, \beta) * \alpha = a : \beta) \end{aligned}$$

The above predicate declares two possible cases separated by a disjunction. Either the pointer x is null and the list content α is empty, or the pointer x points to two cells containing value v and pointer z ,

⁷ Pure assertions such as $x = \text{null}$ are syntactic sugar for $\langle \text{Pure} \rangle(; x = \text{null})$, $x \mapsto v$ for $\langle \text{PointsTo} \rangle(x; v)$, and $x \mapsto a, z$ for $x \mapsto a * x + 1 \mapsto z$

respectively. In the latter case, z is also the beginning of a list, as specified by the recursive call to the `list` predicate, and the content of the list α is obtained by consing v to the content β of the list starting at z .

Encoding in the symbolic world Such inductive predicates raise a challenge, as they are difficult to encode in a symbolic state. Recall that symbolic linear heap state $\sigma = (h, d^\perp)$ comprises a partial map from addresses to values h and an optional domain set d^\perp . All concretisations of a linear heap have a *fixed* number of memory cells. In contrast, the list predicate can represent lists of symbolic lengths, which would have a symbolic number of cells. Therefore, the list predicate cannot be directly encoded in the symbolic linear heap.

The solution is to extend the state with a list of *closed predicates*, of the form $\rho(\vec{v})$, where ρ is the name of the predicate and the symbolic values \vec{v} are its parameters. The symbolic linear heap state $\sigma = (h, d^\perp, p)$ becomes a triple comprising the partial map, the domain set, and the list of closed predicates $p \in \text{Str} \times \overline{\text{Val}} \text{ list}$ called *predicate state*.

Take, for instance, the specification of a simple list-length function `llen`:

```
{ list(x, α) } function llen(x) { Ok : r. list(x, α) * r = |α| }
```

Producing its pre-condition using the substitution $\bar{\theta} = [x \mapsto \bar{x}, \alpha \mapsto \bar{\alpha}]$ would yield a symbolic state of the form $\sigma = (\emptyset, \perp, [\text{list}(\bar{x}, \bar{\alpha})])$.

Unfolding and folding At some point, the function would need to dereference the pointer x to access the tail of the list and compute its length. Pointer dereference will be performed by the load action of the linear heap. However, this action requires the cell to be present in the map, which is currently empty. To successfully perform the dereference, we need to *transfer* some of the resources from the predicate state to the heap map.

This is the role of the `unfold` command, which removes a predicate from the predicate state and replaces it with its definition. In the case of the list predicate, unfolding would result in two symbolic states, each corresponding to a definition of the list predicate. The corresponding symbolic process is depicted in Figure 8.4. After unfolding, in the non-null case, it is now possible to access the tail pointer and recursively compute its length.

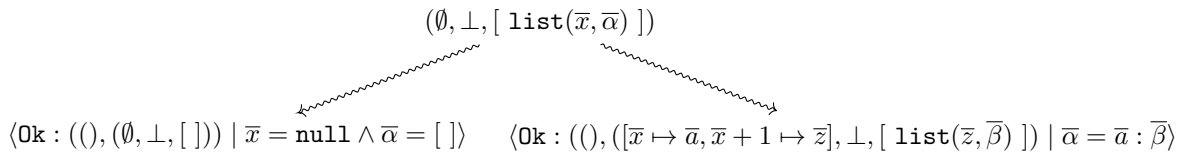


Figure 8.4: Result of unfolding the list predicate, \bar{z} , \bar{a} and $\bar{\beta}$ are fresh symbolic variables.

Folding is the opposite operation, which removes the definition of a predicate from the state and replaces it with the predicate itself. In the list-length example, it is needed when reaching the end of the function,

and the post-condition expects the list predicate to be present in the state again.

8.7.2 Predicate state as a state model transformer

The idea of a predicate state manipulated through unfolding and folding is not new and has been used in VeriFast for over a decade and later in Viper, JaVerT 2.0 and Gillian. Here, we propose a novel formalisation and implementation as a transformer that enables the use of inductive predicates in any state model. The transformer yields state models that can then be seamlessly used with our framework without modifying the symbolic engine. In the current implementation of Gillian, predicates are still part of the core framework, though we intend to extract them in immediate future work.

Extending state models with user-defined predicates Let $\bar{\mathbb{S}}$ be a symbolic state model, and let \mathbf{P} be a set of predicate definitions of the form:

$$\langle \rho \rangle(\vec{x}_i; \vec{x}_o) \triangleq \{Q_1 \vee \dots \vee Q_n\}$$

where \vec{x}_i and \vec{x}_o are the in- and out-parameters of the predicate, and $Q_1, \dots, Q_n \in \text{Asrt}$ are the *case definitions* that must only make use of core predicates $\delta \in \bar{\mathbb{S}}.\Delta$ or user-defined predicates ρ that have their definitions in \mathbf{P} . For example, the list predicate in the linear heap would receive the address \mathbf{x} as an in-parameter and the sequence α as an out-parameter.

We define a new state model transformer $\text{Pred}(\bar{\mathbb{S}}, \mathbf{P})$ that extends $\bar{\mathbb{S}}$ with support for the predicates defined in \mathbf{P} . Its set of states is a pair comprising a symbolic state from $\bar{\mathbb{S}}$ and a list of closed predicates with their name in \mathbf{P} . Each closed predicate carries a list of its in- and out-parameters. User-defined predicates are then used as core predicates of the extended state model:

```
type  $\bar{\Sigma} = \bar{\mathbb{S}}.\bar{\Sigma} \times (\mathbf{P}.\text{names} \times \overline{\text{Val list}} \times \overline{\text{Val list}}) \text{ list}$ 
```

```
type  $\Delta = \mid \text{C of } \bar{\mathbb{S}}.\Delta \mid \cup \text{ of } \mathbf{P}.\text{names}$ 
```

Producer and consumer Producing or consuming a core predicate from the parameter state model delegates the task to its producer and consumer, while a user-defined predicate is simply added or removed from the list of closed predicates.

Three important notes must be made. First, the producer does not enforce any well-formedness invariant. For example, producing a closed predicate with definition $\langle \mathbf{F} \rangle(;) \triangleq \mathbf{False}$ would yield an unfeasible state with a satisfiable path condition. Hence, the predicate state model is *incompatible with UX analysis*, as it cannot check for the satisfiability of the produced states.

Second, the function that removes a predicate from the list of closed predicates must iterate over the list and identify which predicate to remove. This operation may branch if several predicates match the name ρ and the in-parameters. As discussed in §6.8.2, where a similar phenomenon occurs, it is also sound to perform entailment checks

```

let produce  $\delta$  ( $\vec{\sigma}, p$ )  $\vec{v}_i$   $\vec{v}_o$  =
  match  $\delta$  with
  | C  $\delta' \rightarrow$ 
    let*  $\vec{\sigma}' =$ 
       $\vec{\sigma}.$ produce  $\delta' \vec{\sigma} \vec{v}_i \vec{v}_o$ 
    in
    return ( $\vec{\sigma}', p$ )
  | U  $\rho \rightarrow$ 
    return ( $\sigma, (\rho, \vec{v}_i, \vec{v}_o) :: p$ )

let rec remove_pred  $\rho \vec{v}_i p$  =
  match  $p$  with
  | []  $\rightarrow$  Abort (PredNotFound,  $p$ )
  | ( $\rho', \vec{v}_i', \vec{v}_o$ ) ::  $p'$  when  $\rho = \rho' \rightarrow$ 
    if%sat  $\vec{v}_i = \vec{v}_i'$  then ok ( $\vec{v}_o, p'$ )
    else ( $\rho', \vec{v}_i', \vec{v}_o$ ) :: remove_pred  $\rho \vec{v}_i p'$ 
  | ( $\rho', \vec{v}_i', \vec{v}_o$ ) ::  $p'$  when  $\rho \neq \rho' \rightarrow$ 
    ( $\rho', \vec{v}_i', \vec{v}_o$ ) :: remove_pred  $\rho \vec{v}_i p'$ 

let consume  $\delta$  ( $\sigma, p$ )  $\vec{v}_i$  =
  match  $\delta$  with
  | C  $\delta' \rightarrow$ 
    let* ( $\vec{v}_o, \sigma'$ ) =
      with_abort_instead_of_miss
        ( $\vec{\sigma}.$ consume  $\delta' \sigma \vec{v}$ )
    in
    ok ( $\vec{v}_o, (\sigma', p)$ )
  | U  $\rho \rightarrow$ 
    let* ( $\vec{v}_o, p'$ ) =
      remove_pred  $\rho \vec{v}_i p$ 
    in
    ok ( $\vec{v}_o, (\sigma, p')$ )

```

Figure 8.5: Producer and consumer for the predicate state model.

instead of sat checks, preventing branching and gaining predictability at the cost of some automation. This is the choice made by VeriFast.

Finally, the consumer never returns a **Miss** outcome, as it cannot accurately identify if a resource is missing. It explicitly converts all **Miss** outcomes to **Abort** when calling the consumer of \vec{S} and also aborts when a user-defined predicate cannot be found in the list. That is because a resource could be hidden inside another user-defined predicate or would need to be folded. This *information loss* deteriorates the quality of error messages, which is, unfortunately, a necessary side-effect of the expressivity gained.

Unfolding and folding Unfold and fold are then implemented as actions in the predicate state model:

```
type  $\mathcal{A} = \vec{S}.\mathcal{A} \mid \text{Unfold of } \mathbf{P}.\text{names} \mid \text{Fold of } \mathbf{P}.\text{names}$ 
```

Unfold receives the in-parameters of the predicate and removes the corresponding predicate from the state before producing its definitions. The fold operation does the opposite, and consumes the first definition of the predicate that matches the state, and adds its closed form to the state. For conciseness, we elide their definitions.

8.7.3 Recovery mechanisms and automation

Having to unfold and fold predicates manually is a source of complexity for the user. Thankfully, it is possible to implement *error-directed heuristics* to fold and unfold predicates automatically. In particular, if a core predicate $\langle \delta \rangle(\vec{v}_i; \vec{v}_o)$ is declared missing by an action of the parameter state model, the action evaluation function of the predicate state model can find a user-defined predicate closed in the state that shares similar parameters⁸ and automatically unfold it. Similarly, when failing to consume a user-defined predicate, the engine may decide to fold it automatically and try to consume it again.

⁸ In Gillian, actions and consumers return a list of *recovery values* when failing. These values are used to decide what predicates to unfold according to a set of pre-defined heuristics. These heuristics are outside the scope of the current presentation, but are not overly complex.

This is the approach taken by Gillian. In the list-length algorithm example, the engine would automatically unfold the list predicate when the cell is dereferenced and fold it back when it reaches the post-condition. In fact, Gillian verifies the list-length algorithm entirely automatically and requires no manual intervention.

In [Part III](#), we propose an extension of the predicate transformer to model borrows in Rust. These same heuristics are reused to automate reasoning about mutable borrows for the first time.

8.8 The mutable store, and where it goes wrong

While SIGIL does not have a mutable store, it is possible to define a state model that provides one. In the separation logic literature⁹, the mutable store is treated as pure in the assertion language, such that the assertion $x = v * x = v'$ where x is satisfied by a store $[x \mapsto v']$ if and only if $v = v' = v''$.

Using the constructions provided above, the mutable store can be defined as a partial map from strings to values, where values are duplicable: $\text{PMap}(\text{Str}, \text{Ag}(\text{Val}))$. Unfortunately, as mentioned in the section about the agreement state model, mutating values is not frame preserving. This is precisely why using a variable store induces a side condition in the “traditional” frame rule of separation logic. This condition specifies that the frame rule holds only if the frame does not contain any program variable that is modified by the expression:

$$\frac{\text{TRAD-FRAME} \quad \{ P \} e \{ Q \}}{\{ P * R \} e \{ Q * R \}} \text{mod}(e) \cap \text{fv}(R) = \emptyset$$

Additional effort could be made to accommodate this side condition in a general manner by introducing a similar side condition to [Frame addition](#) and [Frame subtraction](#) and allowing state models to declare how to interact with this side condition. However, experience with Gillian shows that the mutable store is a poor model of real-world languages. In systems languages like C and Rust, variables are addressable and hence cannot be kept in a simple mutable store, while in a dynamic language such as JavaScript, the heap is used to emulate an imperfect store¹⁰.

In the current implementation of Gillian, the GIL intermediate language provides a variable store out of the box. Gillian is then in charge of checking that the side condition of the frame rule is never broken. However, this is a source of complexity that brings little benefit.

8.9 Implementation and code reuse

The state model transformers presented in this chapter have been implemented in Gillian and are used to define the symbolic state models of Wisl, JavaScript, and C¹¹. While transformers play a key role modularise and simplifying the soundness proofs of state models in our formalisation, their implementation demonstrates that they also significantly streamline the implementation process.

⁹ Reynolds, “Separation Logic: A Logic for Shared Mutable Data Structures”, 2002 [[Rey02](#)]; Raad et al., “Local Reasoning About the Presence of Bugs: Incorrectness Separation Logic”, 2020 [[Raa+20](#)]; and Maksimović et al., “Exact Separation Logic: Towards Bridging the Gap Between Verification and Bug-Finding”, 2023 [[Mak+23](#)]

¹⁰ Gardner et al., “Towards a program logic for JavaScript”, 2012 [[GMS12](#)]; and Naudziuniene, “An Infrastructure for Tractable Verification of JavaScript Programs”, 2018 [[Nau18](#)]

¹¹ For Wisl and JavaScript, see next chapter; for C, see [Part II](#).

Notably, the Wisl memory model, which previously required 780 lines of code, has been reduced to just 38 lines with transformers. Similarly, the JavaScript state model now consists of 237 lines of code compared to its original 1238. This dramatic reduction highlights the impact of transformers in reducing complexity and improving maintainability.

Beyond code simplification, systematic component reuse enhances reliability and performance. Shared components undergo more thorough scrutiny and testing, and allow optimisations to be ported effortlessly across instantiations. For instance, the new JavaScript state model built using transformers is about 4% faster than its monolithic predecessor. This improvement stems from a small optimisation originally introduced in the C memory model, which was seamlessly integrated into the partial map transformer and automatically applied to JavaScript without additional effort.

Chapter 9

Applications: Wisl and JavaScript

While working on the existing instantiations of Gillian, we carefully considered every choice available in the design and implementation of state models compatible with Gillian, as well as the state model API of Gillian itself. The core insights from this process have already been distilled in the earlier pages of this manuscript. While there remain many observations and novelties in Gillian-C and Gillian-Rust, the state models of Wisl and Gillian-JS have, in an anticlimactic twist, become trivial to describe, thus not meriting separate parts. This chapter outlines both of them as constructions derived from the components introduced in the preceding chapter.

9.1 Wisl: While language for separation logic

The initial version of Wisl (pronounced “weasel”) was developed as a front-end for JaVerT 2.0. The aim was to design a tool to help students learn about semi-automatic verification using separation logic, using the concepts taught in class. This led to the creation of Wisl, a language resembling the small while language from John Reynolds’ seminal paper on separation logic¹, which uses a linear heap. Unlike Gillian, JaVerT 2.0 was not parametric on the state model, necessitating compromises for Wisl to fit a JavaScript-like memory model, notably omitting pointer arithmetic in favour of static objects.

Following its initial creation, Wisl was adapted as a Gillian instantiation the subsequent year, recovering pointer arithmetic and becoming a hub for experimentation. Throughout the years, it enabled us, the Gillian developers, to delve into the effects of various design choices on memory models, significantly contributing to the insights shared in this manuscript.

9.1.1 Overview of the Wisl memory model

At its core, the heap model of Wisl is a (substantially) simplified version of the C heap. The design of the language’s memory management is such that the expression `new(n)` allocates `n` cells of memory (these cells form an *object*) and returns a pointer to the first cell, mimicking the behaviour of `malloc` function in C. Furthermore, objects are considered to exist within separate spaces in memory; pointer arithmetic on a pointer into an object may not yield a pointer to another object. Finally, the statement `free(p)` simulates C’s `free` function by requiring the pointer `p` obtained from a `new` statement and deallocating the entire object at once. Unlike C, however, Wisl does not forbid reading uninitialised memory and instead zero-initialises every cell at allocation time.

This memory model is inspired by the CompCert memory model^{2,3},



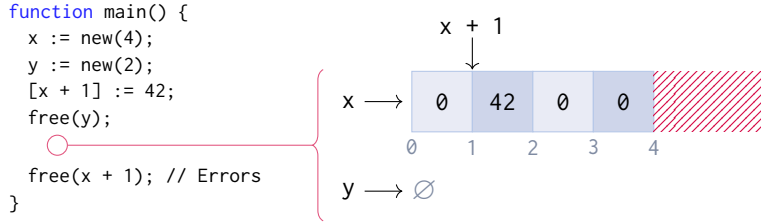
Figure 9.1: The Wisl logo, designed by Valentin Magnat.

¹ Reynolds, “Separation Logic: A Logic for Shared Mutable Data Structures”, 2002 [Rey02]

² Leroy et al., “The CompCert Memory Model, Version 2”, 2012 [Ler+12]

³ Of course, the CompCert memory model is much more complex, as it proposes additional actions to deal with machine integers and keeps track of access permissions, uninitialised memory etc. Gillian-C, presented in Part II, uses a state model that models CompCert much more accurately.

in that it is a map from object locations to objects, where an object is a list of values (bytes in CompCert).



The above figure presents a simple Wisl program which allocates two objects of size 2 and 4 and stores the pointers to their first cell in variables x and y . Then, the program writes 42 in the second cell of the object at address x and frees the first object. Finally, the program attempts to free the second object but provides an invalid pointer – the pointer to the second cell instead of the pointer received from allocation – and the program ends in an error. A diagram of the memory state between the two free statements is presented next to the program’s code.

To support all of the features, we define the following heap model for Wisl:

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

where the List state model captures the representation of the objects. We now describe this new construction in more detail.

9.1.2 The List state model

The Wisl heap model relies on a new $\text{List}(\tau)$ state model to represent the list of values that constitute each object. As expected, a full state in this state model is a list of values of type τ . Loading and storing a value at index i in the list checks that the i is within the list bound before operating.

Defining state fragments to be lists is impossible when, for example, the size of the list is unknown. Instead, we define state fragments as pairs (b, n^\perp) that comprise a partial finite map $b : \mathbb{N} \xrightarrow{\text{fin}} \tau$ (b stands for *block*) from integers to values of the sort τ , and an optional integer n^\perp , called *bound*, corresponding to the size of the list. The bound, if not \perp , specifies that every index greater or equal to is known to be *out-of-bounds*. This is a requirement for the Freeable state model transformer: without the bound, it is impossible to know when a block is exclusively owned.

In Figure 9.2, we give a formal definition of the compositional state model, omitting the store action, as it is defined similarly to the load action.

Wisl symbolic heap and limitation The Wisl symbolic heap also uses partial maps, which are accessed similarly to the symbolic partial map state model. This symbolic heap model is sufficient for analysing

```

module List ( $\tau : \mathbb{T}$ ) = struct

  type  $\Sigma = \left\{ (b, n^\perp) \in (\mathbb{N} \xrightarrow{\text{fin}} \tau) \times \mathbb{N}^\perp \mid \text{dom}(b) \subseteq [0, n^\perp) \right\}$ 

  let load ( $b, n^\perp$ ) i =
    match  $b(i), n^\perp$  with
    |  $v, \_ \rightarrow \text{ok } (v, (b, n^\perp))$ 
    |  $\perp, n$  when  $i \geq n \rightarrow \text{error } (\text{OutOfBounds}, (b, n^\perp))$ 
    |  $\_ \rightarrow \text{miss } (\text{MissingCell}, (b, n^\perp))$ 

  let new  $n = ([i \mapsto 0]_{i=0}^{n-1}, n)$ 

  let is_exclusively_owned ( $b, n^\perp$ ) =
    match  $n^\perp$  with
    |  $\perp \rightarrow \text{false}$ 
    |  $n \rightarrow (\text{dom}(b) = [0, n))$ 

  (* ... *)
end

```

Figure 9.2: Excerpt from the definition of the List state model

many simple examples used in teaching (e.g., doubly-linked lists, trees). However, it also has a strong limitation: it does not allow for the representation of lists of a symbolic size. Since each block is represented as a partial finite map, allocating a block without knowing a concrete number of bindings to insert into the map is impossible. We do not aim at lifting this limitation in the context of Wisl, as we prefer the state model to remain simple and easily understandable by students. However, this limitation is lifted in the context of Gillian-C, which is described in the next part of this manuscript.

9.2 Gillian-JS

Gillian was initially developed by abstracting the heap module specific to JavaScript from JaVerT 2.0 and making the core engine operate as a functor on that module. The state model of JaVerT 2.0, and later Gillian-JS, is extensively described in published papers⁴. Here, we propose a novel, equivalent definition of the Gillian-JS state model in terms of state model constructions.

9.2.1 Overview of the Gillian-JS memory model

Using state model transformers presented in the previous chapter, it is possible to define the Gillian-JS memory model as:

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

where each state is composed of two components. First, the heap is a partial map from location to *dynamic* objects. Each object is a dynamic partial map (described shortly) from strings to values. The string indexes represent property names in JavaScript, which can be dynamically computed at execution time.

The second component of the state is the *metadata* table that efficiently captures each object’s internal properties, providing a significant performance boost when analysing compiled JS code. For each object

⁴ Frago Santos et al., “JaVerT 2.0: compositional symbolic execution for JavaScript”, 2019 [Fra+19]; Frago Santos et al., “Gillian, part I: a multi-language platform for symbolic execution”, 2020 [Fra+20]; and Maksimović et al., “Gillian, Part II: Real-World Verification for JavaScript and C”, 2021 [Mak+21]

location l , its metadata $m(l) = l_m$ is the location of an object containing properties that are used internally by the JavaScript semantics and cannot be directly manipulated by the developer. Since the metadata of each object is immutable, it is captured using the agreement state model, where the resource is duplicable.

The metadata table in the JaVerT 2.0 and Gillian-JS implementations has always been duplicable, facilitating compositional reasoning without having to transfer ownership of the table to callees. However, the corresponding papers described the resource as exclusively owned to avoid cluttering the presentation. Using our state model constructions, defining the metadata table becomes trivial.

9.2.2 Dynamic partial maps

Dynamic partial maps are defined identically to normal partial maps, apart from how “unallocated” cells are handled. In a (full) static partial map, accessing an index not in the map yields an erroneous execution. In JavaScript, however, accessing a property that has not yet been added to the object is allowed and yields a special `undefined` value.

Without care, transferring this behaviour to the compositional world would break frame-preservation. For instance, loading property “ p ” in an empty object would successfully yield the value `undefined`, while loading the same property in the object $[p \mapsto 1]$ would yield the value 1.

This is a well-known problem in JavaScript verification, which was first noticed in 2012⁵. The solution, in the context of compositional symbolic execution, is provided by Naudziūnienė in her PhD thesis⁶, and later in the first version of JaVerT developed by Fragoso Santos et al.⁷. It consists in enhancing the state with a *domain set*, which is a symbolic set which captures all addresses (in the case of JS, properties) which *may* be in the object. All variables outside the domain set are owned and *known to not be in the object*. For instance, creating a new object would yield the empty map and the empty domain set, as the object is known to contain no properties. Furthermore, the pair (obj, d) must satisfy the *heap-domain invariant*: $\text{dom}(\text{obj}) \subseteq d$.

Figure 9.3 shows a representation of the set of property names, distinguishing between properties that in the object (in blue) and properties that are known to not be in the object (in red). The region or properties that are in the domain but not in the object, in white, are *missing* and accessing them yields a `Miss` outcome.

Furthermore, note that the domain set is optional, and can be \perp . In this case, it does not capture any ownership information, and the heap-domain invariant need not be enforced. Any property not in the object’s map is considered missing, as shown in Figure 9.4.

To be frame preserving, the load action of the dynamic partial maps is then defined as follows:

```
let load (obj,  $d^\perp$ ) prop =
  match obj(prop),  $d^\perp$  with
  | Some v, _  $\rightarrow$  ok (v, (obj,  $d^\perp$ ))
  | None,  $d$  when prop  $\notin d \rightarrow$  ok (undefined, (obj,  $d^\perp$ ))
  | None, _  $\rightarrow$  miss (MissingProp, (obj,  $d^\perp$ ))
```

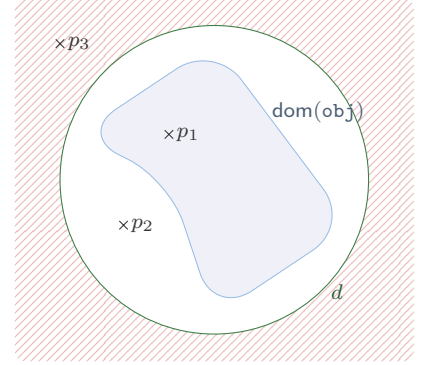


Figure 9.3: Representation of the set of heap addresses, distinguishing the cases $p_1 \in \text{dom}(\text{obj})$, $p_2 \in d$ and $p_3 \notin d$, when $d \neq \perp$.

⁵ Gardner et al., “Towards a program logic for JavaScript”, 2012 [GMS12]

⁶ Naudziuniene, “An Infrastructure for Tractable Verification of JavaScript Programs”, 2018 [Nau18]

⁷ Fragoso Santos et al., “JaVerT”, 2017 [Fra+17]

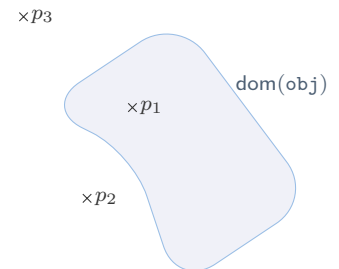


Figure 9.4: Similar to Figure 9.3, but with $d = \perp$

9.2.3 Evaluation

A detailed evaluation of Gillian-JS is outside the scope of this manuscript, and we only provide a brief overview of the results to show the real-world applicability of a JavaScript analyser based on Gillian. First, Gillian-JS is as expressive as JaVerT 2.0⁸ and can reproduce all the results reported in the JaVerT 2.0 paper, apart from those related to bi-abduction⁹. In addition, Gillian-JS was used to verify the header deserialisation module of the AWS Encryption SDK library¹⁰, a larger case study than ever performed with JaVerT 2.0. This case study confirms that migration to the parametric Gillian infrastructure did not restrict the tool’s power.

⁸ Frago Santos et al., “JaVerT 2.0: compositional symbolic execution for JavaScript”, 2019 [[Fra+19](#)]

⁹ When Gillian’s bi-abduction was migrated to UX, the JS fixing interface was not updated accordingly. This is not a limitation, but simply work that remains to be done.

¹⁰ The same module was also verified using Gillian-C, and the corresponding results are presented in [§14.2](#)

Chapter 10

Related work

In this chapter, we review the work related to Gillian. We explain how Gillian and the formalisation presented in this manuscript differ from the work conducted on other CSE verification (§10.1) and bi-abduction (§10.2) tools. We then address other parametric approaches to analysis (§10.3), as well as tools that support both UX and OX (§10.4), two critical features of Gillian. Finally, we describe work that takes a monadic approach to symbolic execution (§10.5) and compare it to our approach in Chapter 6.

10.1 Compositional symbolic execution tools for verification

Immediately related work includes all tools that perform compositional verification using symbolic execution. We first discuss the technique’s inception, followed by VeriFast and Viper, which, alongside Gillian, represent the main tools in this domain with ongoing active development. None of the tools presented in this section are parametric or support whole-program symbolic testing or under-approximate reasoning.

10.1.1 The early days

Compositional symbolic execution was pioneered by the Smallfoot tool¹. It was significant as it provided a formalisation of the CSE methodology, although it lacked a proof of soundness and only worked on a small toy language. Notably, Smallfoot did not employ an SMT solver. Instead, it relied on a bespoke inference engine equipped with hard-coded inductive predicates for reasoning about a few data structures, such as trees and lists. Smallfoot was later formalised in Coq², though the formalisation does not include critical features such as function calls using specifications.

Inspired by Smallfoot, jStar³ emerged as the first compositional symbolic execution tool capable of verifying code written in a real-world language, namely Java. From jStar, coreStar⁴ was developed to create a generic and reusable intermediate language for reasoning with separation logic. Despite having several front-ends, coreStar had a fixed, simple memory model. Like Smallfoot, coreStar required most of the reasoning to be hard-coded directly into the inference engine and did not provide a proof of soundness. Nevertheless, coreStar was the first tool to offer both verification and bi-abduction.

10.1.2 VeriFast

VeriFast⁵, introduced shortly after jStar, is a tool developed at KU Leuven and can be characterised as the first modern CSE tool. It

¹ Berdine et al., “Smallfoot: Modular Automatic Assertion Checking with Separation Logic”, 2006 [BCO06]

² Appel, “VeriSmall”, 2011 [App11b]

³ Distefano et al., “jStar: towards practical verification for Java”, 2008 [DP08]

⁴ Botinčan et al., “coreStar: the Core of jStar”, 2011 [Bot+11]

⁵ Jacobs et al., “A Quick Tour of the VeriFast Program Verifier”, 2010 [JSP10]; and Jacobs et al., “VeriFast: A Powerful, Sound, Predictable, Fast Verifier for C and Java”, 2011 [Jac+11]

pioneered the produce/consume paradigm and employs an SMT solver. VeriFast can verify C, C++ and Java programs using a simple yet expressive state model. It has been successfully used to explore novel techniques for verification using separation logic⁶.

We believe a large part of the VeriFast behaviour could be adapted into a state model for Gillian. In particular, VeriFast encodes all resources into a list of *memory chunks*, analogous to the list of closed predicates carried by the predicate state model discussed in §8.7. Although VeriFast’s encoding of resources is highly expressive, it limits automation. However, it is to be noted that enhancing automation is not a goal of VeriFast. Instead, its design focuses on achieving low verification times and predictable execution, objectives it meets with considerable success.

Moreover, VeriFast incorporates several sophisticated features not currently supported by Gillian’s implementation or the formalisation proposed in this manuscript. Such features include predicate families, predicate constructors, and predicate values. While we believe these features can be implemented within Gillian as state models, further work is required to demonstrate this feasibility.

Featherweight VeriFast Featherweight VeriFast⁷ (FVF) is a mechanised formalisation and soundness proof for a simplified version of VeriFast. To our knowledge, it is the first soundness proof for a compositional symbolic execution engine.

The proof of soundness of FVF employs an intermediate theoretical semantics that manipulates concrete values but handles function calls through specifications. This approach enhances the modularity of its proof and serves as the principal inspiration for the concrete specification semantics introduced in our formalisation. In addition, FVF distinguishes itself by being entirely mechanised in Coq, providing more reliable proofs than our pen-and-paper approach.

That said, the scope of FVF is limited compared to our formalisation. First and foremost, FVF is not parametric. In addition, it uses a concrete compositional semantics as the trusted computing base. As such, it does not establish a link between this compositional semantics and a real non-compositional semantics. Secondly, FVF does not formalise the connection between producers, consumers and the satisfiability relation of an assertion language. Instead, it uses the definition of the consumer and producer to interpret the core predicates. Consequently, FVF does not justify using externally-proven specifications or the external use of specifications proven inside VeriFast. Finally, Gillian’s formalisation captures under-approximate analysis, and FVF does not.

10.1.3 Viper

Viper⁸ is an intermediate language for compositional verification developed at ETH Zurich, and based on a flavour of separation logic called *implicit dynamic frames*⁹ (or IDF). Unlike classic separation logic, IDF allows for the use of heap-dependent expressions.

Viper has many front-ends, including for Rust¹⁰, statically-typed

⁶ Jacobs et al., “Expressive modular fine-grained concurrency specification”, 2011 [JP11]; Penninckx et al., “Sound, Modular and Compositional Verification of the Input/Output Behavior of Programs”, 2015 [PJP15]; Agten et al., “Sound Modular Verification of C Code Executing in an Unverified Context”, 2015 [AJP15]; and Jung et al., “The future is ours: prophecy variables in separation logic”, 2020 [Jun+20]

⁷ Jacobs et al., “Featherweight VeriFast”, 2015 [JVP15]

⁸ Müller et al., “Viper: A Verification Infrastructure for Permission-Based Reasoning”, 2016 [MSS16b]

⁹ Smans et al., “Implicit dynamic frames”, 2012 [SJP12]; and Smans et al., “Heap-Dependent Expressions in Separation Logic”, 2010 [SJP10]

¹⁰ Astrauskas et al., “Leveraging rust types for modular specification and verification”, 2019 [Ast+19]

Python¹¹, Java¹² and Go¹³. However, Viper makes use of a static object-oriented memory model. As such, it remains unclear whether it could be used to model dynamic memory models such as that of JavaScript or untyped Python.

Viper, furthermore, has two back-ends. The first encodes the Viper program – including the permission-based reasoning – into Boogie¹⁴, which allows for the generation of verification conditions. The other, closer to our work, is based on compositional symbolic execution and has been formalised in Schwerhoff’s thesis¹⁵. While this formalisation defines a symbolic semantics for Viper, it does not justify its soundness.

More recently, Zimmerman et al. produced a proof of soundness for a simplified subset of the symbolic semantics¹⁶. This proof covers the soundness of the symbolic execution engine and the verification algorithm. Its main target is to prove the soundness of a *gradual* verification engine, meaning an engine that can work soundly with only partial specifications (which Gillian does not support). Unlike ours, however, their work is neither parametric nor supports under-approximation.

10.1.4 JaVerT 2.0

JaVerT¹⁷, or JavaScript Verification Toolchain, was a CSE tool for verifying JavaScript programs developed at Imperial College London. Its second version, JaVerT 2.0¹⁸, is the ancestor of Gillian and was the first CSE tool to support the three kinds of analysis supported by Gillian and discussed in [Chapter 7](#)¹⁹.

Gillian was initially developed by abstracting the memory-model module specific to JavaScript from JaVerT 2.0 and designing the core engine to operate as a functor on that model. Gillian, therefore, distinguishes itself from JaVerT 2.0 by taking the non-trivial step of incorporating parametricity and under-approximate reasoning.

Finally, throughout the development of Gillian, many improvements have been made to the codebase that enhanced the clarity and modularity of the code and its runtime performances. Compared to the benchmark presented in the JaVerT 2.0 paper, Gillian-JS is now two to three times faster²⁰. These enhancements include for instance, a drastically improved encoding of the path condition into Z3, as well as more efficient matching plans that reduce duplicated work. A detailed description of these changes is outside the scope of this manuscript.

10.2 Bi-abduction tools

The first tool implementing bi-abduction was Abductor²¹, an extension of SpaceInvader, itself a descendant of Smallfoot. Abductor was able to show the scalability and applicability of bi-abduction on a fragment of the C programming language. In addition, it could infer the use of some inductive predicates (such as lists and trees), yielding more expressive over-approximate specifications.

Today, the most famous tool making use of bi-abduction is Infer²², a descendant of Abductor. Infer can perform bi-abduction on Java, C, C++, and Objective-C programs using a simple intermediate language,

¹¹ Eilers et al., “Nagini: A Static Verifier for Python”, 2018 [EM18]

¹² Blom et al., “The VerCors Tool Set: Verification of Parallel and Concurrent Software”, 2017 [Blo+17]

¹³ Wolf et al., “Gobra: Modular Specification and Verification of Go Programs”, 2021 [Wol+21]

¹⁴ Leino, “This is Boogie 2”, 2008 [Lei08]

¹⁵ Schwerhoff, “Advancing Automated, Permission-Based Program Verification Using Symbolic Execution”, 2016 [Sch16]

¹⁶ Zimmerman et al., “Sound Gradual Verification with Symbolic Execution”, 2024 [ZDA24]

¹⁷ Naudziuniene, “An Infrastructure for Tractable Verification of JavaScript Programs”, 2018 [Nau18]

¹⁸ Fragoso Santos et al., “JaVerT 2.0: compositional symbolic execution for JavaScript”, 2019 [Fra+19]

¹⁹ JaVerT 2.0 supports bi-abduction, though in OX mode, instead of UX

²⁰ Fragoso Santos et al., “Gillian, part I: a multi-language platform for symbolic execution”, 2020 [Fra+20]

²¹ Distefano, “Attacking Large Industrial Code with Bi-abductive Inference”, 2009 [Dis09]

²² Calcagno et al., “Infer: An Automatic Program Verifier for Memory Safety of C Programs”, 2011 [CD11]

an extension of the Smallfoot intermediate language (SIL). Infer has been used in industry, inside Meta²³, for over ten years. It is part of Meta’s code review process and analyses commit diffs during the continuous integration pipeline. A few years ago, the maintainers of Infer realised that the over-approximating features of the tool – including the inference of inductive predicates – led to too many false positives, which proved detrimental to the review process. This realisation led to the formalisation of Incorrectness Logic and its separation logic flavour. In turn, it led to a variant of Infer called Infer:Pulse, which is based on an under-approximate implementation of bi-abduction²⁴ without inductive predicates. The formalisations of Incorrectness Separation Logic²⁵ and Pulse, however, make use of the idealised linear heap and do not include a formalisation of function calls.

Other work on bi-abduction includes the body of work about formulating shape analysis as an abstract interpretation²⁶. While this work does not formulate shape analysis parametrically, or as a layer on top of an existing engine, it does describe *relational* shape analysis, which can infer specifications that are more expressive than traditional specifications. For example, the specifications capture whether a mutation happened, even if it was reversed before the end of the function. Furthermore, this work addresses the problem of inferring over-approximate specifications that use inductive predicates, which exceeds the expressivity/power of our work.

10.3 Parametric frameworks for analysis

We now address the literature on parametric frameworks for analysis and verification.

10.3.1 Lithium

One of the most relevant works that takes a parametric approach to analysing several languages is Lithium²⁷. Lithium is a domain-specific programming language shallowly embedded in Coq and aims to write verification tools based on separation logic. To write a verification tool using Lithium, the tool developer writes rules that each handle one verification step. Lithium then comes with an interpreter written in the Ltac tactic language for Coq, allowing automatic proof search. As such, the tool distinguishes itself by being the first compositional verification tool that produced foundational proofs; an impressive feat in our opinion.

Furthermore, Lithium is based on Iris and, as such, enjoys parametricity over the resource model. Its assertion language is minimal, resembling the one presented in §5.1, with the exception that in Lithium, pure assertions are hard-coded in the assertion language since pure assertions use Coq’s `Prop` type. In contrast, our formalisation does not need a special case for pure assertions.

Lithium has several instantiations, including RefinedC²⁸ and RefinedRust²⁹. There, it has proven its applicability to medium-sized (though complex) case studies.

²³ Distefano et al., “Scaling static analyses at Facebook”, 2019 [Dis+19]

²⁴ Le et al., “Finding real bugs in big programs with incorrectness logic”, 2022 [Le+22]

²⁵ Raad et al., “Local Reasoning About the Presence of Bugs: Incorrectness Separation Logic”, 2020 [Raa+20]

²⁶ Chang et al., “Relational inductive shape analysis”, 2008 [CR08]; Illous et al., “A relational shape abstract domain”, 2021 [ILR21]; and Nicole et al., “Lightweight Shape Analysis Based on Physical Types”, 2022 [NLR22]

²⁷ Sammler, “Automated and foundational verification of low-level programs”, 2023 [Sam23]

²⁸ Sammler et al., “RefinedC: automating the foundational verification of C code with refined ownership types”, 2021 [Sam+21]

²⁹ Gäher et al., “RefinedRust: A Type System for High-Assurance Verification of Rust Programs”, 2024 [Gäh+24]

However, foundational proofs come at a price. Proofs performed with Lithium-based tools are slower than those performed by Gillian-based tools and usually require more annotations³⁰. Furthermore, most Lithium proofs require writing some Coq code and, therefore, require more expert knowledge than the CSE tools like Gillian, VeriFast, or Viper.

10.3.2 Semantic frameworks

Semantic frameworks, such as \mathbb{K} ³¹, Ott³², Lem³³ and Redex³⁴, provide specification languages in which users can write the semantics of their target language and automatically generate various tools from this semantics, ranging from interpreters and compilers for multiple back-ends to sophisticated program analysers. Among these frameworks, \mathbb{K} 's agenda is closest to ours, as it automatically generates tools that support various forms of program analysis, including verification. \mathbb{K} has been instantiated to several programming languages, including JavaScript and C, and is used in industry for the symbolic analysis of Ethereum bytecode. However, to our knowledge, \mathbb{K} does not support any compositional analysis.

10.3.3 Symbolic lifting frameworks

Symbolic lifting frameworks, such as Rosette and Chef, automatically lift a concrete interpreter to a symbolic interpreter.

Rosette³⁵ extends Racket³⁶ with solver-aided facilities for creating symbolic values and expressing constraints on those values. With Rosette, a concrete interpreter for the target language is written in Racket and is then symbolically interpreted using Rosette's core symbolic execution engine. Rosette, among other achievements, has been successfully applied to find bugs in parts of the Linux Kernel³⁷. However, an earlier version of the whole-program symbolic testing for JavaScript, currently available in Gillian-JS, was implemented using Rosette³⁸. This implementation was two orders of magnitude slower than the current Gillian-JS³⁹, suggesting that a native implementation, such as that of Gillian, significantly improves efficiency.

Chef⁴⁰ takes a specially packaged, concrete interpreter as input and analyses the target programs by symbolically executing the interpreter's binary. Chef has been applied to dynamic languages, such as Python and Lua. However, the authors comment that its applicability is limited to languages with moderately sized interpreters. Chef would, therefore, not be an appropriate tool for analysis of, for example, JavaScript programs.

10.4 Combining UX and OX analysis

Smash, developed by Godefroid et al.⁴¹, is the most well-known tool that exploits both OX and UX reasoning. It uses first-order logic (as opposed to separation logic). It can *alternate* between OX and UX reasoning to improve the precision and efficiency of the OX analysis. This approach contrasts with Gillian, which can host either OX or UX analyses but has not explored the option of combining their powers. However, according

³⁰ More details about this are given in the related work section of [Part II](#) and [Part III](#)

³¹ Roşu et al., “An overview of the \mathbb{K} semantic framework”, 2010 [[RS10](#)]

³² Sewell et al., “Ott: effective tool support for the working semanticist”, 2007 [[Sew+07](#)]

³³ Mulligan et al., “Lem: reusable engineering of real-world semantics”, 2014 [[Mul+14](#)]

³⁴ Felleisen et al., *Semantics Engineering with PLT Redex*, 2009 [[FFF09](#)]

³⁵ Torlak et al., “Growing solver-aided languages with rosette”, 2013 [[TB13](#)]; and Torlak et al., “A lightweight symbolic virtual machine for solver-aided host languages”, 2014 [[TB14](#)]

³⁶ The Racket Team, *The Racket programming language*, 2024 [[The24](#)]

³⁷ Nelson et al., “Scaling symbolic evaluation for automated verification of systems code with Serval”, 2019 [[Nel+19](#)]

³⁸ Santos et al., “Symbolic Execution for JavaScript”, 2018 [[San+18](#)]

³⁹ This performance comparison is reported in the JaVerT 2.0 paper.

⁴⁰ Bucur et al., “Prototyping symbolic execution engines for interpreted languages”, 2014 [[BKC14](#)]

⁴¹ Godefroid et al., “Compositional May-Must Program Analysis: Unleashing The Power of Alternation”, 2009 [[GNR09](#)]

to Le et al.⁴², who report on personal communication with Godefroid, Smash-style analyses seem to have faced obstacles in practice and were “used in production at Microsoft but are not used by default widely in their deployments, because other techniques were found which were better for fighting path explosion.”

10.5 Monadic symbolic execution

In [Chapter 6](#), we define a symbolic execution monad for writing symbolic execution processes, facilitating soundness proofs. We now discuss similar work.

Mensing et al.⁴³ propose a recipe for deriving a symbolic executor from a definitional interpreter written in Haskell. They compile programs to command trees using a free monad and execute them using a small-step symbolic semantics. However, their approach is only evaluated using a small, simply-typed lambda calculus, and their preliminary evaluation suggests that the resulting engines are several orders of magnitude slower than ours. In addition, their work does not formally justify the soundness of the obtained interpreters. Finally, their approach relies on Haskell-specific features such as lazy evaluation. Our approach, on the other side, is implementable in any functional programming language.

Grisette⁴⁴ is a Haskell library for performing symbolic compilation. As opposed to symbolic execution, symbolic compilation transforms an entire program into a single SMT query in the style of CBMC. The library automatically performs optimisations and allows for state merging. Grisette shows performance improvements over Rosette by an order of magnitude and a reduced generated formula size by a similar scale. However, Grisette has not been applied to the symbolic compilation of real-world languages and provides no soundness result. Nevertheless, it would be interesting to explore a version of Gillian backed by symbolic compilation using Grisette instead of symbolic execution.

μ VeriFast⁴⁵ is a simplified Haskell implementation of VeriFast that makes use of a monadic approach to implement a simplified version of VeriFast. It serves as a case study for a general approach to implement modular effects in Haskell. This implementation does not come with a formalisation of the symbolic execution engine, and the monadic approach is not used to simplify the proofs of soundness.

Finally, Keuchel et al.⁴⁶ formalise symbolic execution in Coq using Kripke specification monads. Their formalisation offers a powerful monad that captures sufficient information to simplify the path condition during execution. In contrast, our formalisation does not allow this, as simplifying the path condition might lead to disconnections between the variables in the state and those in the path condition. In addition, Keuchel et al. fully prove the soundness of their approach in Coq and even perform a case study using separation logic, obtaining a manual CSE tool embedded in Coq. Their approach does not, however, aim at designing symbolic execution tools outside of Coq and is also not automated. As such, it cannot be used in the context of Gillian.

⁴² Le et al., “Finding real bugs in big programs with incorrectness logic”, 2022 [[Le+22](#)]

⁴³ Mensing et al., “From definitional interpreter to symbolic executor”, 2019 [[Men+19](#)]

⁴⁴ Lu et al., “Grisette: Symbolic Compilation as a Functional Programming Library”, 2023 [[LB23](#)]

⁴⁵ Devriese, “Modular Effects in Haskell through Effect Polymorphism and Explicit Dictionary Applications”, 2019 [[Dev19](#)]

⁴⁶ Keuchel et al., “Verified symbolic execution with Kripke specification monads (and no meta-programming)”, 2022 [[Keu+22](#)]

Part II
Gillian-C

Chapter 11

Gillian-C: What and why?

*Be you. Be proud of you. Because
you can be do what we want to do.*

François Hollande, 2015

In 2024, C remains one of the most widely used programming languages in the world, particularly for systems programming. Its low-level nature and (in appearance) permissive fined-grained control make it a popular choice for writing performance-critical software. However, it also comes with many pitfalls. For example, it provides virtually no static check for memory safety, making it easy to introduce bugs that can lead to security vulnerabilities. It is also known for its *undefined behaviours*, which exist in the specification to permit certain compiler optimisations, but can lead to unexpected results in practice. These rather unfortunate properties make C a prime target for dynamic and static analysis tools, including sanitisers¹, fuzzers², concolic execution tools³, bounded model checkers⁴, abstract interpreters⁵, compositional symbolic execution tools⁶, and foundational verifiers⁷. This was our main motivation for exploring using Gillian to analyse C code.

In addition, we felt that Gillian being able to tackle both C and JavaScript would demonstrate its flexibility and the viability of our parametric approach. As detailed shortly, the C state model is practically the antipode of the JS state model: C is low-level, with manual memory management, pointer arithmetic, unsafe operations and byte-level abstraction, whereas JS is fully safe, garbage-collected, and has a high-level object-oriented dynamic memory model.

Importantly, designing Gillian-C has also allowed us to explore and leverage a unique aspect of Gillian’s parametric approach that distinguishes it from other CSE tools, which is that the representation of symbolic states is fully decoupled from the associated assertions. In particular, in Gillian-C, objects are represented in the symbolic heap as trees, capturing the low-level layout of bytes in memory. For instance, the production of both the assertion $p \mapsto_{\text{int8}} 0 * (p + 1) \mapsto_{\text{int8}} 0$ and the assertion $p \mapsto_{\text{int16}} 0$ yields the same symbolic state. This design enables Gillian-C to automate reasoning when manipulating the heap at the byte level. In contrast, tools like VeriFast, which preserve a tight correspondence between the symbolic heap and the assertions, require manual user intervention to convert between these two assertions.

We have evaluated Gillian-C comprehensively on real-world case studies. When it comes to whole-program symbolic testing, we have created a suite of symbolic tests for Collections-C, an open-source library of data structures for C, finding several bugs that were then

¹ Nethercote et al., “Valgrind: a framework for heavyweight dynamic binary instrumentation”, 2007 [NS07]; and GNU GCC developers, *Program Instrumentation Options - GNU GCC*, 2024 [GNU24]

² Haller et al., “Dowsing for overflows: a guided fuzzer to find buffer boundary violations”, 2013 [Hal+13]

³ Cadar et al., “KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs”, 2008 [CDE08]

⁴ Clarke et al., “A Tool for Checking ANSI-C Programs”, 2004 [CKL04]

⁵ Cousot et al., “The ASTREÉ Analyzer”, 2005 [Cou+05]; and Kirchner et al., “Frama-C: A software analysis perspective”, 2015 [Kir+15]

⁶ Jacobs et al., “VeriFast: A Powerful, Sound, Predictable, Fast Verifier for C and Java”, 2011 [Jac+11]

⁷ Sammler et al., “RefinedC: automating the foundational verification of C code with refined ownership types”, 2021 [Sam+21]; and Appel, “Verified Software Toolchain”, 2011 [App11a]

fixed by the library developers. When it comes to verification, we have verified the AWS Encryption SDK message-header deserialisation module, which is stable, critical, industry-grade code (~ 950 lines) that uses advanced language features to manipulate complex data-structures. In addition to identifying potential vulnerabilities, this case study has demonstrated the feasibility of a multi-language verification workflow with Gillian, where a library implemented in both C and JS was verified with the help of the same *pure* abstractions. Finally, when it comes to bi-abduction, we also used the Collections-C library to experiment with automatic synthesis of under-approximate specifications, obtaining promising performance. In this chapter, we give a quick tour of the infrastructure of Gillian-C and showcase its analyses and features.

11.1 The Gillian-C Infrastructure

Gillian-C is a Gillian instantiation, and as such requires a compiler from C to GIL, Gillian’s intermediate language, and a symbolic state model for the Gillian back-end.

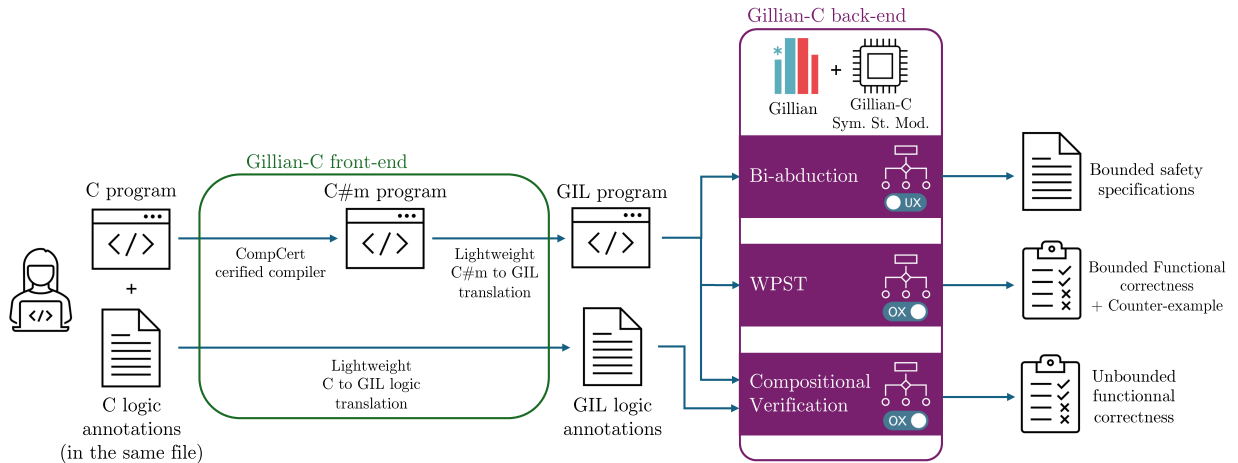


Figure 11.1: Gillian-C: architecture.

When using Gillian-C, the C program to be analysed is first compiled into an intermediate language known as C#minor (read: “C sharp minor”) using the CompCert verified compiler⁸. Subsequently, Gillian-C carries out a lightweight translation from C#minor to GIL, the intermediate language of Gillian. The GIL program is then analysed using the Gillian-C back-end, which is Gillian’s core, parameterised with the Gillian-C symbolic state model. This infrastructure allows Gillian-C to cover a large subset of C features, including pointer arithmetic, structures, first-class function pointers, unstructured control flow using goto statements, and access to the byte-level representation of objects.

Being an instantiation of Gillian, Gillian-C supports three types of analyses: whole-program symbolic testing, compositional verification, and specification generation using bi-abduction. Notably, the Gillian-C state model supports both over- and under-approximation at the flip of a switch, ensuring that compositional verification provides sound functional correctness guarantees while bi-abduction generates sound incorrectness separation logic specifications.

⁸ Leroy et al., “Formal Verification of a C-like Memory Model and Its Uses for Verifying Program Transformations”, 2008 [LB08]; Leroy, “Formal verification of a realistic compiler”, 2009 [Ler09b]; and Leroy, “A Formally Verified Compiler Back-end”, 2009 [Ler09a]

For compositional verification, users are expectedly required to provide logic annotations such as specifications, loop invariants, and lemmas. Gillian-C offers a straightforward separation-logic assertion language tailored for annotating C programs, which translates directly into GIL. As CompCert does not accommodate logic annotations, these are parsed independently by the Gillian-C front-end, translated into GIL separately from the C program, and then integrated into the resulting GIL program.

11.2 Whole-program symbolic testing

Recall that whole-program symbolic testing (WPST) consists of symbolically executing a program starting from its entry point, which is the `main` function in C. For illustrative purposes, we use a simplified version of a test that revealed a bug in the Collections-C library. The corresponding program is given in [Figure 11.2](#), and described below.

```
#define INITIAL_CAPACITY 4

// A dynamic array structure (or vector) of integers.
typedef struct Array { int *buffer; size_t capacity; size_t size } Array;

void array_new(Array *arr) {
    // Creates an array with an initial capacity of 4 and size 0.
    ...
}

void push(Array *ar, int value) {
    // Extends a given array with a given value.
    // If array is full, allocates a new buffer with twice the capacity,
    // copies content of old buffer into new buffer, and frees old buffer.
    ...
}

void remove(Array *ar, size_t index) {
    // Removes the element at a given index from a given array,
    // moving all subsequent values one position to the left.
    if (index != ar->size - 1) {
        size_t bytes_to_move = (ar->size - index) * sizeof(int);
        memmove(
            &(ar->buffer[index]),
            &(ar->buffer[index + 1]),
            bytes_to_move
        );
    }
    ar->size--;
}

int main() {
    FRESH_UINT(init_loops);
    Array arr; array_new(&arr);
    for (unsigned int i = init_loops; i > 0; i--)
        push(&arr, i);
    for (unsigned int i = init_loops; i > 0; i--)
        remove(&arr, 0);
    assert(arr.size == 0);
    return 0;
}
```

Figure 11.2: A small C symbolic test that detects an out-of-bound access.

This symbolic test creates a symbolic integer, `init_loops`, and uses it to push `init_loops` integers into an array, then removes them one

by one. The test checks that the array is empty at the end. When running this test with Gillian-C Using the command `gillian-c wpst array_oob.c --unroll 10 -l disabled`, where `array_oob.c` is the name of the file containing the program, and `--unroll 10` specifies that loops should be unrolled 10 times, and `-l disabled` specifies that exhaustive symbolic trace logging should be disabled., the tool will generate the the following message (with parts elided for clarity):

```
Compilation time: 0.019958s
Total time (Compilation + Symbolic testing): 0.093744s
Errors occurred!
Here's a counter example: [ (#lvar0: 4i) ]
Here is a final symbolic state at error time:
[ . . . ] Procedure 'i__memmove' [ . . . ]
raised: BufferOverrun [ . . . ]
```

The error message states that the symbolic test failed, and provides a concrete value for the generated symbolic variable that would apparently lead to an out-of-bound access. Further, the tool provides a pretty-printing of the symbolic state at the time of the error, which indicates that the error occurred in the `memmove` function while the array had capacity 4 and size 4. In the `remove` function, when index is 0 and size is 4, it is computed that 4 integers should be moved to the left, starting from index 1 (`index + 1`). However, as the capacity of the array is 4, this range extends beyond the end of the array, thus causing an out-of-bound access.

As confirmation, executing the same program with `init_loops = 4` using a sanitiser such as Valgrind does raise a similar error, confirming an out-of-bounds access. The fix, which was accepted by the maintainer of the library⁹, was to modify the computation of `bytes_to_move` to be `(ar->size - index - 1)* sizeof(int)`.

⁹ Ayoun, *fix buffer overflow - Collections-C* - Github, 2019 [Ayo19a]

Comparison with CBMC CBMC is a state-of-the-art bounded model checking tool for C. It takes similar inputs to Gillian-C in WPST mode, and provides similar correctness guarantees. When running this symbolic test with CBMC¹⁰, using the same unrolling bound (10), CBMC is also able to detect this bug, yielding the following message:

```
line 33 memmove source region readable: FAILURE
```

However, CBMC does so within a longer time frame, taking about 16s to complete the analysis, compared to Gillian-C's 0.093s. In Chapter 14, we provide a more extended comparison between Gillian-C and CBMC on the entire Collections-C library.

¹⁰ We use CBMC v5.95.1, released October 2023, with arguments that should provide similar guarantees to Gillian-C. More details in Chapter 14.

11.3 Compositional verification

To illustrate compositional verification we reuse the previous example, focussing this time on the `push` function, the definition of which is given in Figure 11.3, simplified to ignore overflow checks.

When pushing, if the size of the content is equal to the capacity of the array, the function re-allocates a new buffer with twice the capacity, copies the content of the old buffer into the new buffer, and frees the

```

void push(Array *ar, int value) {
  if (ar->size == ar->capacity) {
    ar->capacity *= 2;
    int *new_buffer = malloc(ar->capacity * sizeof(int));
    memcpy(new_buffer, ar->buffer, ar->size * sizeof(int));
    free(ar->buffer);
    ar->buffer = new_buffer;
  }
  ar->buffer[ar->size++] = value;
}

```

Figure 11.3: Definition of push function for the dynamic array example, simplified to ignore overflow checks.

old buffer. The function then adds the new value at the end of the array, which must have sufficient capacity to host it.

To verify correctness of this function, we start by writing a separation-logic predicate `valid_array(ar, content)` that captures the invariant of the array structure. This predicate, given in Figure 11.4 (simplified), states that `ar` is a pointer to a valid array with content `content` if:

- `ar` is a valid pointer to an *owned* region of memory that hosts a structure of type `Array` with fields `#buffer`, `#capacity`, and `#size`¹¹, as specified by the *points-to* predicate;
- the number of elements in the array does not exceed its capacity;
- there is an array of `#size` integers starting from pointer `#buffer`, holding contents captured by the sequence `content`;
- any memory between the end of the array content and the end of the allocated space may be uninitialised; and
- the buffer was allocated with `malloc`, using the capacity of the array.

¹¹ The `#` in front of the variable names, when inside a predicate, indicate implicitly existentially-quantified variables.

```

/*@ pred valid_array(+ar, content) {
  ar -> struct Array { #buffer; #capacity; #size } *
  #size <= #capacity *
  ARRAY(#buffer, int, #size, content) *
  UNINIT(#buffer + #size, (#capacity - #size) * sizeof(int)) *
  MALLOCD(#buffer, #capacity * sizeof(int))
} */

```

Figure 11.4: Definition of the `valid_array` predicate (simplified).

Note also the `+` in front of the `ar` argument of the predicate, indicating that it is an in-parameter, while `content` is an out-parameter.

With this predicate in place, we can write a specification for the `push` function, as given in Figure 11.5. The specification states that, given a pointer to a valid array `ar` with content `content`, the function `push` will modify the array in place such that, at the end, `ar` is a valid array with content `content @ [value]`, where `@` denotes the concatenation of two sequences.

```

/*@ spec push(ar, value) {
  requires: valid_array(ar, content)
  ensures: valid_array(ar, content @ [value])
} */

```

Figure 11.5: Specification of the `push` function (simplified).

Gillian-C is able to fully-automatically verify this specification without any further user intervention. On our machine, which is a MacBook Pro 2019, with a 2.3GHz 8-Core Intel Core i9 processor and 16GB of RAM, the verification takes 0.23s.

Comparison with VeriFast VeriFast is the state-of-the-art compositional verifier for C. Its focus, however, is on performance and predictability, rather than automation. For the verification of the push function, VeriFast requires the user to write 4 additional lines of tactics to guide the proof, while Gillian-C requires none. On the other hand, with these annotations in place, VeriFast verifies the function faster than Gillian-C, in 0.08s.

Comparison with CN More recently, CN¹² has seen a rise in popularity as a compositional verifier for C. CN works similarly to VeriFast, but is based on Cerberus, a thoroughly validated semantics of C. For the verification of the push function, CN requires the user to write 6 lines of annotations, and verifies the function in 35.5s.

¹² Pulte et al., “CN: Verifying Systems C Code with Separation-Logic Refinement Types”, 2023 [Pul+23]

11.4 Specification synthesis using bi-abduction

The third analysis Gillian-C is able to perform is the generation of incorrectness logic specifications using bi-abduction. These specification can then be used to automatically detect bugs in the program that are guaranteed to be true bugs¹³, although this last step has not yet been implemented in Gillian.

¹³ Le et al., “Finding real bugs in big programs with incorrectness logic”, 2022 [Le+22]

To avoid cluttering the presentation, we give an illustrative example simpler than dynamic arrays as the size of the generated specifications grows quickly. Consider the function `deref` given in Figure 11.6: it receives a pointer to an integer, and returns the value contained at the corresponding address.

```
int deref(int *p) {
    return *p;
}
```

Figure 11.6: Implementation of the `deref` function.

When running Gillian-C in bi-abduction mode, the tool will generate one *success* specification and one *bug* specification, both given in simplified form in Figure 11.7¹⁴.

¹⁴ We use mathematical notation, as these specifications are generated at the GIL level, and not at the C level.

$\begin{array}{c} [p = \text{NULL}] \\ \text{int deref}(\text{int } *p) \\ [\text{Err} : r. r = \text{"SegFault"}] \end{array}$	$\begin{array}{c} [p \mapsto_{\text{int}} v] \\ \text{int deref}(\text{int } *p) \\ [\text{Ok} : r. \mapsto_{\text{int}} v * r = v] \end{array}$
---	--

Figure 11.7: Incorrectness separation logic specifications synthesised by Gillian-C using bi-abduction for the `deref` function.

The success specification states that, given a valid pointer `p` to an integer, the function will successfully return that integer. The bug specification captures the case where the pointer is `NULL`, and the function raises a segmentation fault. In Chapter 14, we show that Gillian-C is able to generate specifications for the entire Collections-C library, showing promising performance.

Chapter 12

The Gillian-C symbolic state model

The Gillian-C symbolic state model is based on the CompCert memory model. We give an overview of this memory model, and explain the challenges in implementing a corresponding symbolic state model. In response to these challenges, we introduce a novel symbolic representation of memory blocks called *symbolic block trees*. We present the symbolic state model of Gillian-C, together with its associated actions, core predicates, fixes, and discuss its implementation.

12.1 The CompCert memory model

As the front-end of Gillian-C is built on CompCert, the natural choice for us was to adopt the CompCert memory model as the foundation for the state model of Gillian-C. This choice is further supported by the extensive documentation of the CompCert memory model in academic literature¹ and its implementation in the CompCert Coq development.

CompCert models C memory as collections of memory objects or *blocks*: each block is identified by a *block identifier*, $l \in Loc$, and contains a sequence of bytes. These low-level sequences of bytes are then interpreted at a higher level as the C values that the programmer is familiar with, $v_C \in CVal$ (defined in algebraic datatype form in Figure 12.1). This interpretation is done with the help of memory chunks, $\tau \in Chunk$ (defined in algebraic datatype form in Figure 12.2), which hold information about the size, type (integer or float), signedness (signed or unsigned) and (implicitly) alignment of a value to be loaded from or stored into memory.

The memory model exposes four basic operations:

- **load**(l, o, τ), which receives a block identifier l , an offset $o \in \mathbb{N}$ in the block, the memory chunk $\tau \in Chunk$ describing the size, type, signedness and alignment of the value to be loaded, and returns the appropriate value from memory;
- **store**(l, o, τ, v) which receives a block identifier, offset, memory chunk, as well as the value $v_C \in CVal$ to be stored, and updates the memory accordingly; and
- **alloc**(n), which receives a positive integer n , allocates a new block of n bytes, and returns a pointer to the first byte;
- **free**(l), which receives a block identifier l , and deallocates the corresponding block.

For example, given a block identifier l that holds a sequence of bytes $[0, 1, 2, 3, 4, 5, 6, 7]$, and assuming a big-endian architecture:

- **load**($l, 2, Mint8signed$) returns 2;
- **load**($l, 2, Mint16unsigned$) returns $2 * 256 + 3 = 515$;
- **load**($l, 3, Mint16signed$) fails because of alignment restrictions (a

¹ Leroy et al., “The CompCert Memory Model, Version 2”, 2012 [Ler+12]

```
type CVal =  
  | Vint of  $\mathbb{Z}$   
  | Vlong of  $\mathbb{Z}$   
  | Vfloat of  $\mathbb{R}$  (* Simplified *)  
  | Vptr of  $Loc \times \mathbb{N}$ 
```

Figure 12.1: Definition of C values (simplified).

```
type Chunk =  
  | Mint8signed | Mint8unsigned  
  | Mint16signed | Mint16unsigned  
  | Mint32 | Mint64 | Mfloat32  
  | Mfloat64 | Many32 | Many64
```

Figure 12.2: Definition of C chunks.

16-bit value has to be aligned to a multiple of two); and

- `store(l, 4, Mint32, 117835012)` results in the sequence of bytes `[0, 1, 2, 3, 7, 6, 5, 4]`.

Challenges The above description of the CompCert memory model suggests that a more advanced form of the symbolic state model of Wisl could be useful in the creation of a Gillian-C symbolic state model. Recall from [Chapter 9](#) that the state model of Wisl is also based on a block-offset approach², and can be built as a partial finite map from block identifiers to list of values that are exclusively owned, and where entire list of values can be freed at once: $\bar{S} = \text{PMap}(\text{Loc}, \text{Freeable}(\text{List}))$.

² In fact, the state model of Wisl was designed to be a simplified version of the C state model.

However, the Wisl state model has major limitations when it comes to the real-world applications of C. First, in the List symbolic state model, blocks are modelled as partial finite maps from offsets to values, preventing representation of blocks of unbounded size. Therefore, to reason about such blocks we would have to resorting to inductive predicates, which would come with a substantial automation cost. Second, using the List state model would require of us to read and write symbolic values byte by byte, decomposing and recomposing them at each operation. This would be inefficient and result in having complex symbolic expressions in the state.

12.2 Symbolic block trees: overview

To answer the above challenges, we introduce a novel representation of blocks in memory, called *symbolic block trees* and denoted by `BlockTree`. Symbolic block trees are specifically tailored to the C state model. In particular, they support:

1. representation and reasoning about **blocks of unbounded size**;
2. **abstraction preservation**, in that they allow for efficient higher-level value representation, but also for these values to be decomposed into individual bytes;
3. **variable-sized values**, in that they allow for blocks to contain representations of values of different sizes within the same block;
4. **C-specific behaviour**, such as failure on uninitialised memory access or padding between fields of C structures; and
5. **compositional reasoning**, in that they allow for the modelling of *partially owned* blocks.

With symbolic block trees in place, we define the symbolic state model of Gillian-C as:

$$\bar{S}_C = \text{PMap}(\text{Loc}, \text{Freeable}(\text{BlockTree}))$$

and inherit the four basic operations of the CompCert memory model as the actions of \bar{S}_C .

The basics A symbolic block tree is a binary tree in which each node represents a range within a memory block, captured by two symbolic natural numbers, denoting the start-offset and end-offset of the range. Each node also has its content, which is either: a symbolic C value; a symbolic sequence of values; a special uninitialised value with a

quantity (explained shortly); a special “zero” value which exists only as an optimisation; or an indicator that the node is “not owned”, meaning that its range is not part of the symbolic state. Finally, and expectedly, each node in a symbolic block tree may have zero or two children. The definition of symbolic block trees as an algebraic datatype is as follows:

```

type qty = Partially | Totally

type content =
  | Value of  $\overline{Val} \times Chunk$ 
  | Array of  $\overline{Val} \times Chunk$ 
  | Uninitialised of qty
  | Zeros
  | NotOwned of qty

type node = {
  range:  $\overline{N} \times \overline{N}$ ;
  content: content;
  children: (node * node) option;
}

```

Symbolic block trees come with several well-formedness invariants. First, for each node, the end-offset must be greater than the start-offset. Second, the ranges of children nodes must be disjoint, contiguous and fully cover the range of their parent node. Third, the content of a node must match its size. For instance, a node containing a value of chunk `int32` must have a range spanning 4 bytes. Similarly, a node containing of list of values of chunk `uint16` must have a range spanning twice as many bytes as the length of the sequence value it contains. Finally, the node contents must be sufficient for understanding if a load operation can be performed, independently of its children.

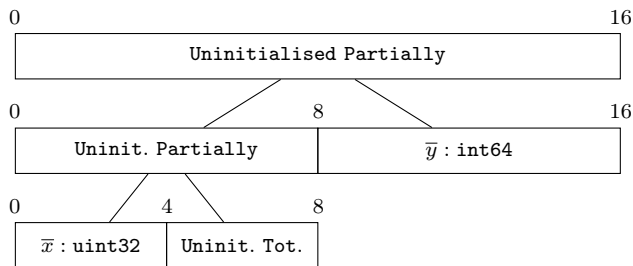
We illustrate this last invariant with a more elaborate example. Take a structure `S` made up of two fields, `x` and `y`, of respective types `int` and `long`, on an architecture where an `int` value is 4 bytes long and a `long` value is 8 bytes long³:

```

struct S { unsigned int x; long y; };

```

A block containing an instance of `S` can be represented using the tree in Figure 12.3. The left-most and right-most leaf nodes, respectively covering the ranges $[0, 4)$ and $[8, 16)$ correspond to the fields `x` and `y` of the structure, as per the C structure layout algorithm. In addition, the leaf node covering the range $[4, 8)$ is marked as uninitialised, as it corresponds to the padding between the fields of the structure. The root node and its left child are both marked as *partially* uninitialised, as they contain uninitialised memory somewhere in their ranges.



This structure makes loading values from memory efficient, as the tree can be traversed to find the appropriate range. Once the range is

³ Gillian-C assumes a fixed architecture that characterises the size and alignment of each type as well as endianness for the values. This architecture can be configured through the command line.

Figure 12.3: Symbolic block tree representing a block containing an instance of `S` where fields `x` and `y` respectively contain symbolic values \bar{x} and \bar{y} .

found, the content of the node can be used directly, without the need to access the children. For example, the operation `load(l, 0, uint32)` on the block in Figure 12.3 would traverse the tree to find the node covering the range $[0, 4)$ and return the symbolic value \bar{x} it contains. Furthermore, if the pointer to the structure were to be cast to an `unsigned long*` pointer, the load operation would use the range $[0, 8)$. In that case, the corresponding node would be found to contain the `Uninitialised Partially` value, and the load would immediately yield an erroneous outcome as uninitialised memory access is identified as undefined behaviour in the C standard.

Further decomposition A more complex operation would consist of loading only the first byte of the structure, which can be done by casting a pointer to its `x` field to a `char*` pointer. In that case, the load operation would decompose the leftmost node into two nodes, one containing the first byte of \bar{x} and the other containing the remaining three bytes, as depicted in Figure 12.4.

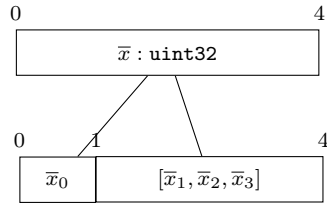


Figure 12.4: Result of reading the first byte of the leftmost node from Figure 12.3. \bar{x}_0 has chunk `uint8`, while the right child is a sequence of chunk `uint8` and length 3. The values of the \bar{x}_i are computed appropriately according to the endianness of the architecture.

Performing the read operation in this way has the advantage of preserving value abstraction. While the node \bar{x} is decomposed further, it still remains in the tree, and if a subsequent read is performed on the range $[0, 4)$, the tree will be traversed to find the node \bar{x} directly, ignoring its children. Furthermore, if the user performed a read operation on the range $[0, 1)$, there is a chance they will either perform another read or write at the same range. As the tree already has the correct shape, this operation can be performed efficiently again.

Storing The store operation is performed similarly to the load operation, but also updates the parents once the corresponding leaf node has been updated. For instance, updating the field `x` of the structure in Figure 12.3 with a value \bar{x}' would not change the parents, since its sibling node is still uninitialised. On the other hand, updating the first byte of \bar{x} would be performed in three steps. First, the first byte would be isolated in its own node as in Figure 12.4. Second, the value of the leftmost leaf node would be updated to consider the value of the new byte \bar{x}'_0 . Finally, the parent node would be reconstructed to the sequence $[\bar{x}'_0, \bar{x}_1, \bar{x}_2, \bar{x}_3]$. The updated tree is given in figure 12.5. When attempting to read this range using the chunk `uint32`, this array would be *decoded* into a value depending on the endianness of the architecture. For instance, on a big-endiann architecture, the array would be decoded into the value $2^{24} \cdot \bar{x}'_0 + 2^{16} \cdot \bar{x}_1 + 2^8 \cdot \bar{x}_2 + \bar{x}_3$. The case of partially updating values and then reading them is rare enough for us to believe that the performance cost of manipulating a complex symbolic value obtained from the recomposition is acceptable.

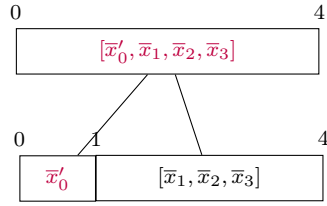


Figure 12.5: Result of updating the first byte of the leftmost node of Figure 12.3. The node is split as in Figure 12.4, the left child is updated, and the parent is then reconstructed. The updated components are highlighted in purple.

Unbounded reasoning Symbolic block trees can model blocks of arbitrary size by making the indexes in the ranges that annotate each node symbolic. The tree invariants, e.g. that the ranges of the children node cover the range of the parent, are preserved by all operations and are entailed by the path condition throughout execution.

Instead of a simple binary search in a tree, each choice could now become a SAT check. While such SAT checks can become expensive in extreme cases, they have been manageable in practice. In particular, in the large majority of cases, all nodes of a symbolic block tree are annotated either with concrete offsets, or with offsets of the form $\bar{i}_b + i_o$, where \bar{i}_b is a base symbolic offset used across all nodes of the tree, and i_o is a concrete offset. When an access is performed with an offset of the same shape, range comparison can be reduced to the comparison of concrete offsets and be decided without using the SMT solver.

Allocation Symbolic block trees come with two flavours of allocation, respectively corresponding to the behaviour of the standard `malloc` and `calloc` functions in C. The C `malloc` function allocates a block of memory of a given (potentially symbolic) size, but does not initialise it: this is the behaviour of our allocation action, which creates a node of a given size with content `Uninitialised Totally`. The C `calloc` function allocates a block of memory of a given size, and initialises it to zero: this we accomplish by first using the allocation action, as in the `malloc` case, and then using another action that zero-initializes entire arrays of data, setting the node content to `Zeros`. In this way, we avoid having to deal with sequences of values constrained by an awkward quantified formula of the form $|\bar{x}| = \bar{n} \wedge \forall i \in [0, n), \bar{x}[i] = 0$.

Block bounds and freeing Recall that the `Freeable` state model presented in Chapter 8 requires its parameter to expose a function indicating if the content is *fully* and *exclusively* owned. Symbolic block trees maintain an additional *block bound* value, which is a symbolic natural number that indicates the maximum size of the block. It has the same use as the bound in the `List` state model, and can also be \perp if the size of the block is not known. In the context of symbolic block trees, a block can be freed if it satisfies the following three conditions:

1. its bound, \bar{n} , does not equal \perp ;
2. its root node span is exactly from 0 to \bar{n} ;
3. its root node must not be of the kind `NotOwned`.

Note that the root node being of the kind `NotOwned` captures the case in which there exists at least one range of the block that is not owned, and therefore the block cannot be freed.

Additional actions The Gillian-C symbolic memory model exposes two further actions: one that performs a memory copy of an array of data; and one that performs pointer validity check for valid comparisons. More detail about the latter is provided in §13.1.3.

12.3 Symbolic block trees: implementation

We provide a high-level presentation of the implementation of symbolic block trees in Gillian-C. We first present a selection of tree-manipulating operations, followed by the implementation of the `load` and `store` actions of the memory model.

12.3.1 Core functions

We present three low-level operations used for manipulating symbolic block trees:

- `split`, which decomposes one node into two nodes;
- `merge`, which composes two nodes into a single one; and
- `frame_range`, which performs frame inference, precisely isolating the specified range and potentially updating the node.

Splitting The `split` operation decomposes a given node into two nodes. In the example of the previous section, it is the operation used to produce the children node of the \bar{x} node in Figure 12.4. It takes a leaf node `node`, and an offset \bar{o} that must be in the range of n , and returns a pair of nodes (n_1, n_2) such that n_1 covers the range from the beginning of n inclusive to \bar{o} exclusive, and n_2 covers the range from \bar{o} inclusive to the end of n :

```
val split : node →  $\bar{N}$  → (node × node) symex
```

Note that this operation is defined within the symbolic execution monad, as it may instantiate new symbolic variables and add constraints to the path condition. Below, we provide a (simplified) implementation for two representative cases:

```
let split node  $\bar{o}$  =
  let ( $\bar{o}_s, \bar{o}_e$ ) = node.range in
  let* left_content, right_content =
    match node.content with
    | Uninitialised Totally →
      Symex.return (Uninitialised Totally, Uninitialised Totally)
    | Value ( $\bar{v}, \tau$ ) →
      let*  $\bar{b}$  = bytes_of_value  $\bar{v} \tau$  in
      let  $\bar{b}_l = \bar{b}_l = \text{subseq}(\bar{b}, 0, \bar{o} - \bar{o}_s)$  in
      let  $\bar{b}_r = \bar{b}_r = \text{subseq}(\bar{b}, \bar{o} - \bar{o}_s, \bar{o}_e - \bar{o})$  in
      Symex.return (Array ( $\bar{b}_l$ , uint8), Array ( $\bar{b}_r$ , uint8))
  | (* ... *)
in
let left_node = {
  range = ( $\bar{o}_s, \bar{o}$ ); content = left_content; children = None;
} in
let right_node = {
  range = ( $\bar{o}, \bar{o}_e$ ); content = right_content; children = None;
} in
Symex.return (left_node, right_node)
```


The first case splits a node containing a totally uninitialised value, which produces two totally uninitialised values.

The second case splits a single value of chunk τ . We explain this case step-by-step using the example of Figure 12.4, where a node spanning the range $[0, 4)$ and containing a single value \bar{x} of chunk `uint32` is split at offset 1. In that case, we have that $\bar{o}_s = 0$, and $\bar{o}_e = 4$.

First, the value \bar{x} is decomposed into bytes using the `bytes_of_value` function, which yields a symbolic value $\vec{b} = [\bar{x}_0, \bar{x}_1, \bar{x}_2, \bar{x}_3]$, which represents the sequence of bytes that encodes \bar{x} under the used architecture. Next, two fresh symbolic values, \vec{b}_l and \vec{b}_r are created: \vec{b}_l is computed as the left subsequence of \vec{b} up to offset o (in our example, $\bar{o} = 1$ and $\bar{o}_s = 0$, so $\vec{b}_l = [\bar{x}_0]$), and \vec{b}_r is computed as right subsequence of \vec{b} starting from offset $\bar{o} - \bar{o}_s = 1$, and of length $\bar{o}_e - \bar{o} = 3$ (that is, $\vec{b}_r = [\bar{x}_1, \bar{x}_2, \bar{x}_3]$). Finally, the function returns two nodes, each containing the corresponding sequence of bytes, using the chunk `uint8` that corresponds to a raw byte.

Merging The `merge` operation is used to combine two nodes. It is used to re-create a parent node after one of its children has been updated. For instance, it is used in our running example to re-create the parent node of Figure 12.5. It receives two nodes that are assumed to be contiguous, and returns a single node covering both their ranges. We provide an implementation of four representative cases:

```

let merge left right =
  let ( $\bar{o}_s^l, \bar{o}_e^l$ ) = left.range in
  let ( $\bar{o}_s^r, \bar{o}_e^r$ ) = right.range in
  let parent_range = ( $\bar{o}_s^l, \bar{o}_e^r$ ) in
  let* parent_content =
    match left.content, right.content with
    ① | NotOwned Totally, NotOwned Totally →
      Symex.return (NotOwned Totally)
    ② | NotOwned _, _ | _, NotOwned _ →
      Symex.return (NotOwned Partially)
    ③ | Value ( $\bar{v}_l, \tau_l$ ), Value ( $\bar{v}_r, \tau_r$ ) when  $\tau_l = \tau_r$  →
      Symex.return (Array ([ $\bar{v}_l, \bar{v}_r$ ],  $\tau_l$ ))
    ④ | Value ( $\bar{v}_l, \tau_l$ ), Value ( $\bar{v}_r, \tau_r$ )
      when is_float  $\tau_l$  && is_int  $\tau_r$  →
      if mode =  then
        let*  $\vec{b}$  = fresh_bytes ( $\bar{o}_e^r - \bar{o}_s^l$ ) in
        Symex.return (Array ( $\vec{b}$ , uint8))
      else
        Symex.vanish ()
    | (* ... *)
  in
  Symex.return {
    range = parent_range;
    content = parent_content;
    children = Some (left, right)
  }

```

Case ① merges two nodes that are totally not owned, which produces a parent node that is totally not owned. Case ② merges two nodes, one of which is at least partially not owned, which produces a parent node that is partially not owned. These two cases take precedence over all the other cases, which is crucial to the behaviour of, for instance, the `free` operation, which can operate only on a block that is entirely owned. Case ③ merges two nodes that contain values of the same

chunk, which produces a parent node that contains the array of the two values of that chunk.

Finally, case ④ is a good example of where over- and under-approximation come up in practice. This case merges two nodes that contain values of different chunks: a float and an integer. At this point, Gillian-C gives up on precision and, depending of the mode, either creates a fresh, unconstrained sequence of bytes of the appropriate length (hence over-approximating), or gives up on that path of execution by vanishing (hence under-approximating). This is an implementation choice of Gillian-C⁴; in particular, we estimate that if such an operation is performed, the user is likely to not need precise knowledge of the result loading the entire range of bytes as a single value.

Range-framing The `frame_range` operation is the core operation of symbolic block trees. In essence, it uses binary search to perform frame inference and isolate a specific range from a given node. Given its importance, we describe this function in more detail and provide a number of illustrative examples both here and later on, in the context of loading and storing. We give explanations in the text, and also annotate the code with further explanations where appropriate. The full signature of the `frame_range` operation is:

```
val frame_range :
  replace_node: (node → node symex) →
  rebuild_parent: (node → node → node → node symex) →
  node →
   $\overline{\mathbb{N}} \times \overline{\mathbb{N}} \rightarrow$ 
  (node × node) symex
```

The operation receives: the root node from which to extract the range; the range of interest; and two functions, called `replace_node` and `rebuild_parent`. The `replace_node` function updates the node that covers the range of interest if appropriate, taking this node as an argument. The `rebuild_parent` function receives the previous parent node and its updated left and right nodes, and creates the new parent node appropriately. The entire operation returns two nodes, the first is the node that covered the range of interest before the operation, and the second is the updated root node after the operation. The code of `frame_range` is as follows:

```
let frame_range replace_node rebuild_parent root range :
  let* new_root = extend_if_needed root range in
  frame_inside replace_node rebuild_parent new_root range
```

The `frame_range` function first calls the auxiliary function `extend_if_needed`, which checks whether or not the range to be framed is contained within the bounds of the provided root node, and if not, extends this root node to contain this range by adding totally-not-owned fragments to the left and/or to the right. For example, if we had a root node with range $[0, 4)$ from which we wanted to frame the range $[0, 8)$, as in Figure 12.6, `extend_if_needed` would return a tree in which the new root would have the range $[0, 8)$ and two children, with the left being the node originally passed to `frame_range` (with range $[0, 4)$), and the right

⁴ In the real implementation of Gillian-C, symbolic block trees have an additional kind of node called **Lazy**. Instead of approximating at the time of merging the nodes, a **Lazy** node is created, and approximation is performed should that node be accessed. In UX, this delays vanishing if the value is never needed, and in OX, this avoids creating symbolic variables and adding unnecessary constraints to the solver.



Figure 12.6: A tree with a single root node with range $[0, 4)$.

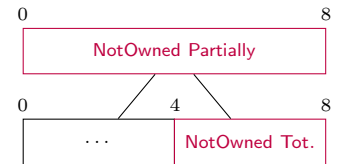


Figure 12.7: Tree from Figure 12.6 extended to the range $[0, 8)$. The added nodes are highlighted in purple.

being a totally-not-owned node with range $[4, 8)$, as in Figure 12.7. Note that adding totally-not-owned fragments in this way is sound as such fragments do not characterise any resource. Having ensured that the targeted range is within the bounds of the root node, control is then passed on to the `frame_inside` function, which performs the actual framing. Its code is shown in Figure 12.8 below:

```
let rec frame_inside replace_node rebuild_parent root range =
  (* Range of root node corresponds to requested range *)
  if%sat Range.is_equal range root.range
  then
    (* Replace node and return *)
    let* new_root = replace_node root in
    Symex.ok (root, new_root)
  else
    (* Range of root node does not correspond to requested range... *)
    match root.children with
    (* ...and root node has children *)
    | Some (left, right) →
      (* Get the split-point of the children *)
      let _, mid = left.range in
      if%sat Range.point_strictly_inside mid range
      (* Split-point is in the requested range *)
      then
        (* Tree rebalancing, elided due to complexity *)
        ...
      (* Split-point is not in the requested range, meaning that the requested
         range is fully contained within either the left or the right child. *)
      else
        if%sat Range.is_inside range left.range
        (* The requested range is contained within the left child *)
        then
          (* Recursively frame inside the left child, ... *)
          let* node, new_left = frame_inside replace_node rebuild_parent left range in
          (* ...rebuild root given new left child, ... *)
          let* new_root = rebuild_parent root new_left right in
          (* ...and return *)
          Symex.ok (node, new_root)
        (* The requested range is contained within the right child *)
        else
          (* Recursively frame inside the right child, ... *)
          let* node, new_right = frame_inside replace_node rebuild_parent right range in
          (* ...rebuild root given new right child, ... *)
          let* new_root = rebuild_parent root left new_right in
          (* ...and return *)
          Symex.ok (node, new_root)
        (* ...and root node has no children *)
      | None →
        (* Split root node to isolate targeted range *)
        let* new_root = split_to_range root range in
        (* Recursively frame inside the obtained new root. This call is guaranteed to find the node with range
           corresponding to the requested range, and will rebuild the parents as it traverses back up the tree. *)
        frame_inside replace_node rebuild_parent new_root range
```

Figure 12.8: Implementation of the `frame_range` function (excerpt)

and we only elide the part of the code related to tree rebalancing, which is illustrated in one of the upcoming examples. The function `Range.is_equal` returns an expression that evaluates to true if the two ranges are equal, and the `if%sat` operator is used to perform the check at the symbolic level, potentially branching, as described in Chapter 6.

12.3.2 Loading and storing

Loading The load operation of symbolic block trees is implemented using the `frame_range` operation. Its implementation, with several details regarding checking read permissions and alignment elided, and instructive commentary inlined, is as follows:

```
(* Load value described by chunk  $\tau$  from given node at offset  $\bar{o}$  *)
let load node  $\bar{o}$   $\tau$  =
  (* The range to be framed starts from offset  $\bar{o}$ 
     and its size is obtained from the chunk  $\tau$  *)
  let range = ( $\bar{o}$ ,  $\bar{o}$  + size_of  $\tau$ ) in
  (* Obtained node does not require modifications for loading *)
  let replace_node = (fun node → Symex.ok node) in
  (* Parents are rebuilt by simply connecting the new children *)
  let rebuild_parent =
    fun parent left right →
      Symex.ok {
        content = parent.content;
        range = parent.range;
        children = Some (left, right)
      }
  in
  (* Obtain framed range representing the value, and the updated node *)
  let* framed, new_node = frame_range node range replace_node
    rebuild_parent in
  (* Compute return value *)
  let* ret_val =
    match framed.content with
    (* If any part of the framed range is not owned,
       signal that there is missing resource *)
    | NotOwned _ → Symex.miss MissingResource
    (* If any part of the framed range is uninitialised,
       signal an appropriate error *)
    | Uninitialised _ → Symex.error UninitialisedAccess
    (* Otherwise, decode the obtained value using
       the information from the chunk  $\tau$  *)
    | content → decode  $\tau$  content in
  (* Return the computed value and the updated node *)
  (ret_val, new_node)
```

Observe, in particular, that the `load` operation is able to differentiate between missing resources and uninitialised access when the load operation cannot be correctly performed. Moreover, note how `load` obtains the desired value by *decoding* the node that covers the range of interest using the specified chunk. In the large majority of cases, the decoding is a no-op, but sometimes, if the value is read using a different chunk than the one used in the heap encoding, it may require *re-encoding* the value (for example, by re-interpreting the value with a different signedness). We elide the implementation of the decode function, but note that its behaviour is what we call *abstraction-preserving*, in that it only reasons at the byte level when absolutely necessary, thereby yielding a more efficient engine.

Storing The store operation of symbolic block trees is also implemented using the `frame_range` operation, as follows:

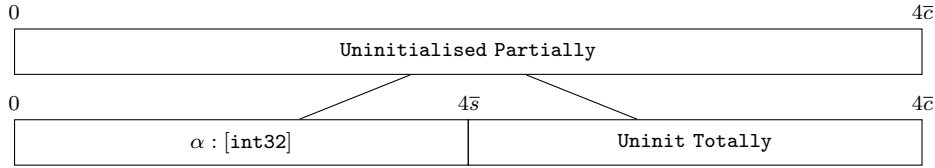
```
(* Store value  $\bar{v}$  described by chunk  $\tau$  into given node at offset  $\bar{o}$  *)
let store node  $\bar{o}$   $\tau$   $\bar{v}$  =
  (* The range to be framed starts from offset  $\bar{o}$ 
     and its size is obtained from the chunk  $\tau$  *)
  let range = ( $\bar{o}$ ,  $\bar{o}$  + size_of  $\tau$ ) in
  (* Obtained node is overwritten by the value to be stored, encoded
     appropriately using the information from the chunk  $\tau$  *)
  let replace_node = (fun _ → Symex.ok (encode  $\bar{v}$   $\tau$ )) in
```

```

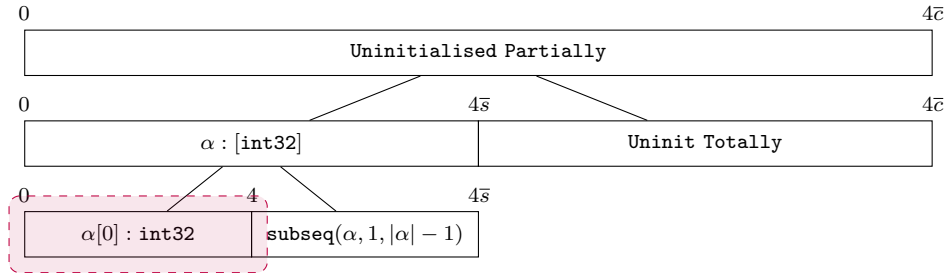
(* Parent nodes are rebuilt by merging the new children,
   disregarding the previous parent entirely *)
let rebuild_parent = fun _ left right → merge left right in
(* Obtain framed range representing the value, and the updated node *)
let* framed, updated = frame_range node range replace_node rebuild_parent
in
match framed.content with
| NotOwned _ → Symex.miss MissingResource
| _ → Symex.ok updated

```

Examples To illustrate all of the above-mentioned operations, we revisit dynamic arrays, our running example from the overview. The tree in the figure below represents a buffer of size \bar{s} , capacity \bar{c} , with content represented by α . We assume for the purposed of this example that the array contains more than one element (i.e., that $|\alpha| > 1$) and that it can be extended without reallocation by more than one element (i.e., that $s + 1 < c$). The array content occupies a contiguous sequence of bytes from 0 to $4\bar{s}$ (as an integer occupies 4 bytes), whereas the remained allocated space for the array occupies a contiguous sequence of bytes from $4\bar{s}$ to $4\bar{c}$.



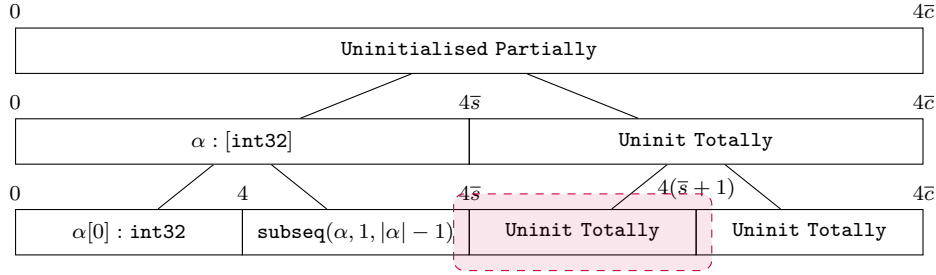
The first operation that we will perform is loading the leading element of the array, resulting in the following tree:



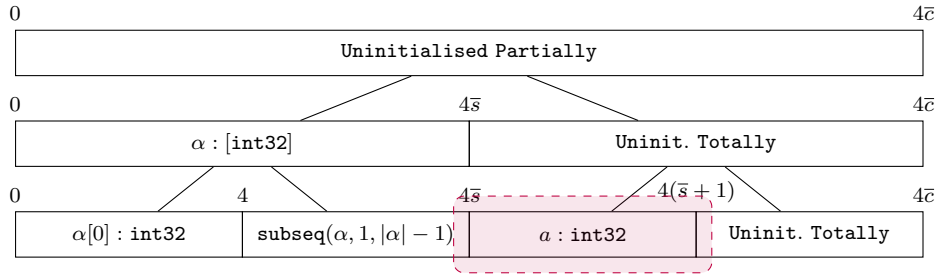
As per the load operation, we first use `frame_range` to isolate a node in the appropriate range (in this case, the range corresponding to the first element of the array, which is $[0, 4)$). This requires of us to split the array content into two children nodes, the left one having the range $[0, 4)$, corresponding to the element being loaded (this is also the node that will be returned and is highlighted in purple), and the right one having the range $[4, 4\bar{s})$, corresponding to the rest of the array. The operation does not modify the returned node, as per the definition of `replace_node`, and also, as per the definition of `rebuild_parent`, does not modify the content of any other nodes.

The second operation that we will perform is to extend the array with an integer value a . As per the `store` operation, the first step is

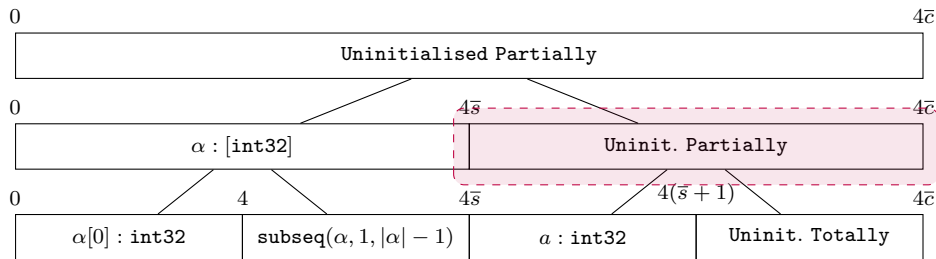
to isolate the appropriate range using `frame_range`:



and this, analogously to the above load example, results in the node that represents the unused part of the array being split into the node corresponding to the requested range (from $4\bar{s}$ to $4(\bar{s} + 1)$), highlighted in **purple**, and the node representing the rest of the unused part. Both of these nodes are created with content `Uninitialised Totally`, as that was the content of their parent node. The next step of `store` uses the `replace_node` function to set the content of the pinpointed node to the given value a , highlighted below in **purple**:



In contrast to `load`, `store` also has to rebuild the parent nodes appropriately. In this case, this means that the node covering the range $[4\bar{s}, 4\bar{c})$ is updated from `Uninitialised Totally` to `Uninitialised Partially`, as it now has a child that is not uninitialised. Updates further up the tree are not required, as the root node is already partially uninitialised. This is illustrated in the diagram below:



We revisit this example further in the next subsection, giving insight into the rebalancing and consumption mechanisms.

12.4 Symbolic block trees: assertion language

Recall from [Chapter 5](#) that Gillian compositional state models are required to provide a set of core predicates, together with a producer

and consumer for each of them. The framework then uses these core predicates as building blocks for the assertion language, by adding support for existential quantifiers, separating conjunction, and variables. The Gillian-C state model exposes five core predicates:

- the typed points-to predicate, $\langle \text{TyPointsTo} \rangle(l, o, \tau; v)$, pretty-printed $(l, o) \mapsto_{\tau} v$, which states that the value v encoded with chunk τ is stored starting from offset o in the block at location l ;
- the array predicate, $\langle \text{Array} \rangle(l, o, \tau, n; v)$, which states that v is an array of n values encoded with chunk τ stored starting from offset o in the block at location l ;
- the uninitialised predicate, $\langle \text{Uninit} \rangle(l, o, n;)$, which states there are n uninitialised bytes starting from offset o in the block at location l ;
- the bound predicate, $\langle \text{Bound} \rangle(l; n)$, which states that the block at location l was allocated with bound n , and that any offsets beyond that bound are to be considered out-of-bounds; and
- the freed core predicate, $\langle \text{Freed} \rangle(l;)$, which states that the block at location l has been freed; this predicate is provided by the **Freeable** state model transformer described in [Chapter 8](#).

We give more detail about the typed points-to predicate, as it is the most important one. The idea of annotating points-to predicates with a type in C is not novel to Gillian-C; it was introduced in a C version of jStar⁵, and has also been used in VeriFast since its creation.

⁵ Parkinson et al., “Separation logic and abstraction”, 2005 [PB05]

We believe that the novelty of Gillian-C is its distinction between the encoding of the symbolic heap and logic assertions, stemming from the inherent flexibility of Gillian. In particular, in VeriFast, the heap is represented as a list of *heap chunks* that is very similar to the structure of the assertions themselves. In Gillian-C, however, the symbolic heap is an expressive data structure, and logic assertions are only used as a syntactic *view* of the heap. For example, take the following C code with a simple specification:

```
//@ requires: p ↦_{uint32} 0 * (p + 1) ↦_{uint32} 0 * aligned(p, 8)
//@ ensures: (ret == 0) * True
long test (int* p)
{
    long* q = (long*) p;
    return *q;
}
```

In the CompCert memory model, where there is no effective typing, this specification is valid. In memory, the regions of memory at p and $p + 1$ are contiguous and contain only zeros. Therefore, reading them using a chunk that covers both regions is valid and also yields zero. VeriFast, however, requires the manual application of a lemma that transforms the two heap chunks into a single one, whereas Gillian-C is able to reason about the heap directly using the symbolic block trees and can automatically perform the required transformation. In particular, producing the pre-condition when $p = (\bar{l}, \bar{o})$ would yield the following symbolic block tree represented in [Figure 12.9](#) at location \bar{l} .

When loading the value at pointer q , which is effectively the same address as pointer p , the root node of this tree is found, and the array $[0, 0] : \text{uint32}$ is decoded using the chunk **uint64** into a single value 0. The post-condition can then be successfully verified automatically.

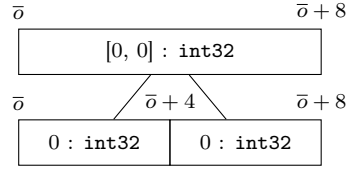


Figure 12.9: Result of producing two contiguous cells of chunk uint32 containing value 0.

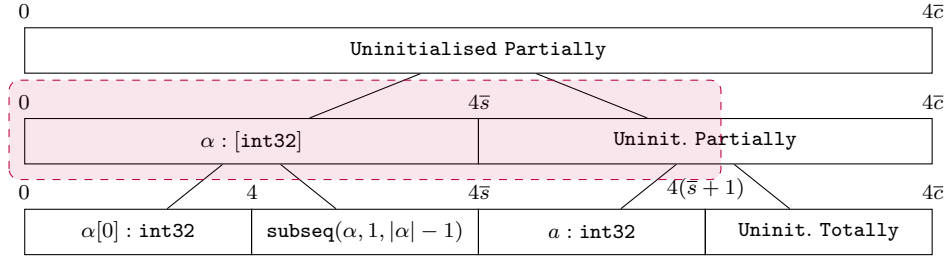
Production and consumption The production and consumption of a single typed points-to predicate are implemented straightforwardly using the `frame_range` function presented in the previous section. Below, we give excerpts of their implementations, with explanations inlined in the code:

```
let produce_points_to node  $\bar{o}$   $\tau$   $\bar{v}$  =
  (* The range to be framed starts from offset  $\bar{o}$ 
     and its size is obtained from the chunk  $\tau$  *)
  let range = ( $\bar{o}$ ,  $\bar{o}$  + size_of  $\tau$ ) in
  (* Production, like storing, updates the content of the obtained
     node appropriately using the information from the chunk  $\tau$  *)
  let replace_node = (fun _ → Symex.ok (encode  $\bar{v}$   $\tau$ )) in
  (* Parent nodes are rebuilt by merging the new children,
     disregarding the previous parent entirely *)
  let rebuild_parent = fun _ left right → merge left right in
  (* Obtain framed range representing the value, and the updated node *)
  let* framed, updated = frame_range node range replace_node rebuild_parent
  in
  match framed.content with
  (* Production succeeds only if the production range was entirely not
     present, returning the updated tree *)
  | NotOwned Totally → Symex.return updated
  (* Otherwise, it vanishes, indicating duplicated resource *)
  | _ → Symex.vanish

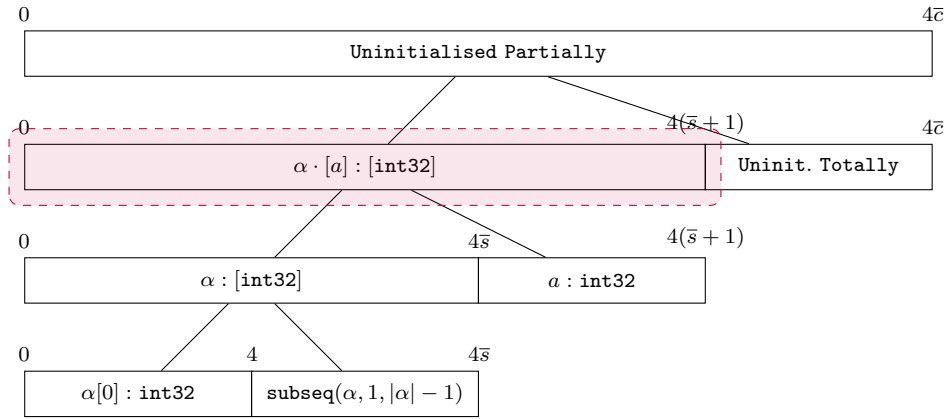
let consume_points_to node  $\bar{o}$   $\tau$   $\bar{v}$  =
  (* The range to be framed starts from offset  $\bar{o}$ 
     and its size is obtained from the chunk  $\tau$  *)
  let range = ( $\bar{o}$ ,  $\bar{o}$  + size_of  $\tau$ ) in
  (* Consumption sets the content of the obtained node to totally
     not owned, effectively removing/consuming the pinpointed resource *)
  let replace_node =
    fun _ → Symex.return {
      content = NotOwned Totally; range; children = None
    }
  in
  (* Parent nodes are rebuilt by merging the new children,
     disregarding the previous parent entirely *)
  let rebuild_parent = fun _ left right → merge left right in
  (* Obtain framed range representing the value, and the updated node *)
  let* framed, updated = frame_range node range replace_node rebuild_parent
  in
  let* ret_val =
    match framed.content with
    (* If any part of the framed range is not owned,
       signal that there is missing resource *)
    | NotOwned _ → Symex.miss MissingResource
    (* If any part of the framed range is uninitialised,
       signal an appropriate error *)
    | Uninitialised _ → Symex.lfail IncompatibleContent
    (* Otherwise, decode the obtained value using
       the information from the chunk  $\tau$  *)
    | content → decode  $\tau$  content in
  (* Return the computed out-value and the updated node *)
  (ret_val, new_node)
```

Finally, we note that the production and consumption of array predicates are similar to those of the typed points-to predicates, with the

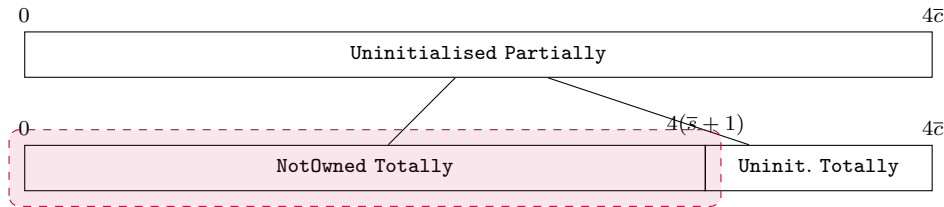
exception that the former use the `decode_array` and `encode_array` functions that are more complex than their single-value counterparts and approximate more often. We illustrate this consumption on the example from the last section, attempting to consume the entire array, highlighted in purple in the diagram below, with the requested range $[0, 4(\bar{s} + 1))$:



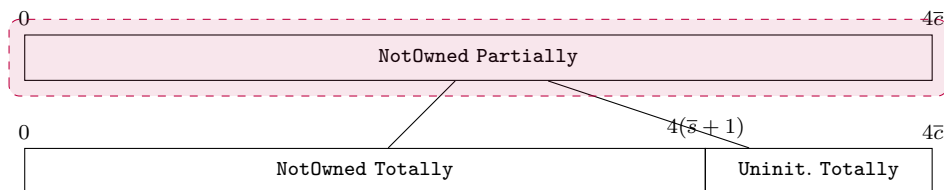
Since in this case the requested range spans both children of the root node, `frame_range` rebalances the tree, creating a new left child of the root that corresponds precisely to the requested range (which is the node that will be returned by `frame_range`), as per the following diagram:



Next, the content of the node that is corresponding to the requested range is set to `NotOwned Totally`, that is, the array is effectively consumed:



and, finally, the root node of the tree is updated accordingly:



12.5 Symbolic block trees: fixes for bi-abduction

Recall that automatic UX specification synthesis, presented in §7.3, is an analysis offered by tools implemented on top of Gillian. To be enabled, this analysis requires the corresponding symbolic memory model to be compatible with under-approximation, and to expose *fixes* that can be used to extend the state when a *missing* outcome occurs.

Symbolic block trees are also designed to make fixes easy to infer in most cases. In particular, when a resource is missing from the state, the `frame_range` function will most commonly return a node of the `NotOwned Totally` kind. If this happens when performing a load operation at address p and using the τ chunk, the fix simply consists of producing the points-to predicate $p \mapsto_{\tau} v$, where v is a fresh symbolic value of the appropriate type.

More complex cases can arise when only part of the returned range is missing, in which case the `frame_range` function will return a node of the `NotOwned Partially` kind. Gillian-C currently does not produce fixes for partially missing resource, as they are rare in practice. However, it is possible to improve the current implementation to produce a fix by identifying the missing part of the range and producing arrays of bytes of the appropriate length.

Drawbacks of using C#minor Gillian-C uses CompCert to compile C to the C#minor intermediate language, and then compiles C#minor to GIL⁶. Unfortunately, C#minor erases most of the type information of the original C program, together with the distinction between pointers and integers. When fixing a missing error of loading a value of chunk `int64`, Gillian-C must then propose three fixes, producing either a value of type `long`, or a valid pointer, or `NULL`.

This behaviour is not ideal, as it leads to the generation of unused specifications, and can cause an explosion in the number of generated specifications. To mitigate this issue, Gillian-C currently implements heuristics that attempt to distinguish between `int64` chunks, and a new `ptr` chunk introduced for this purpose. Both of these chunks behave in the exact same way for all purposes except the inference of fixes. The `int64` chunk is used to detect that a fix should be produced for a value of type `long`, while the `ptr` chunk is used to detect that a fix should be produced for a valid pointer or `NULL`.

⁶ The front-end of Gillian-C is presented in more detail in the next chapter.

Chapter 13

The Gillian-C front-end

In the previous chapter, we have presented the symbolic state model of Gillian-C, which constitutes the main component of the Gillian-C back-end, together with the primary operations it exposes and the associated basic core predicates.

In this chapter, we focus on the front-end of Gillian-C, which is responsible for translating C code into GIL. We also touch on our C assertion language and its compilation into GIL logic annotations.

13.1 Compiling C code

The Gillian-C front-end is based on the CompCert certified compiler, which compiles C code into an low-level structured intermediate language called C#minor (read: “C sharp minor”). C#minor is the second intermediate language of CompCert¹. CompCert first translates C² into a high-level intermediate language called Clight, removing expression side-effects and simplifying various forms of loops into a uniform loop structure. Clight is then translated into C#minor, drastically simplifying expressions, further simplifying loops, and eliminating types.

Gillian-C makes use of a specially-packaged version of CompCert that has been modified to work as a library and expose the appropriate API. This avoids the need to write a printer and parser for C#minor, reducing the complexity of the front-end and the chance of errors.

13.1.1 C to C#minor translation

Figure 13.1 shows an example translation of a simple C function into C#minor by CompCert. The C function, on the left, takes a pointer into an array of integers, and increments the pointer until it finds the value 0 in the array. It then returns the pointer to that value. The corresponding C#minor code, on the right, deserves some explanation.

```
1  int *test(int *p) {
2    while (*p > 0) {
3      p++;
4    }
5    return p;
6  }

1  "test"('p') : long → long ①
2  {
3    loop { ②
4      if (int32['p'] > 0) ③ {
5        /*skip*/;
6      } else {
7        exit 1; ④
8      }
9      'p' = 'p' +1 4LL *1 longofint 1; ⑤
10   }
11   return 'p';
12 }
```

First, note that high-level type information is erased in C#minor. The function signature ① shows that the function receives a parameter

¹ Leroy, *The CompCert C Compiler*, 2023 [Ler23]

² Technically, the subset of C it supports, called “CompCert C”

Figure 13.1: Translation of a simple C function (on the left) into C#minor (on the right) by CompCert.

of type `long` and returns a value of the same type: `C#minor` has erased the distinction between pointers and integers. Then, the while loop has been simplified to an infinite loop ②, with an explicit if-statement to check the condition ③, and an exit statement to break the loop ④ when the condition is false. The exit statement receives a single parameter indicating how many outer blocks to break out of.

In C, the guard of the while loop is evaluated by dereferencing the pointer `p` and comparing its value to 0. In `C#minor`, the dereference is performed using an explicit load operation `int32['p']`, using the memory chunk `int32` and the pointer contained by the local variable `p`.

The loop body, a simple increment of the pointer in C, is translated to an explicit update of local variable `p` ⑤. Pointer addition is performed by explicitly adding 4 (the size of an integer on the used architecture) times the value of the summand to the pointer (here, 1). Note that the operators `+1` and `*1` are explicitly typed to apply to values of type `long`, and that casting between basic types is explicitly performed using unary operators, such as `longofint`.

While this function does not make use of structures, note that field access in `C#minor` is also performed through explicit pointer arithmetic. For example, in the structure `struct s { int x; long y; }`, the address of the field `y` of a pointer `p` is obtained through `'p' +1 8LL`, since field `y` is at an offset of 8 bytes from the beginning of the structure.

13.1.2 C#minor to GIL translation

We continue our example by showing the translation of the `C#minor` code from Figure 13.1 into GIL in Figure 13.2. Firstly, note that GIL does not have structured control flow and uses unconditional and conditional goto statements to jump to labels. The load operator is translated to a call to the `mem_load` action of the state model. The action is given the chunk ("`int32`") in string form, together with the address to load from. In GIL, pointers are represented as lists containing two elements, the location and the offset, which are destructured using the list access operator `1-nth`. The result of loading is stored in a temporary local variable `gvar__0`.

In our GIL translation, values are also encoded as lists of two elements, where the first element is a string capturing the type of the value (e.g. "`int`"), and the second is the value itself. This is an unfortunate necessity; in particular, the type information preserved in `C#minor` is not sufficient and requires of us to encode this form of dynamic type checking into the GIL program.

Other operators, such as `longofint`, `1*` and `+1` are also encoded as calls to internal GIL functions performing the corresponding operation. We provide more details on internal functions shortly.

13.1.3 Internal functions and the Gillian-C runtime

As can be seen in the above example, the GIL code generated by the front-end makes use of a number of *internal functions*. At runtime, during all of the supported analyses, Gillian is able to symbolically execute these functions and reason about their effects without requiring

```

1  proc test(p) {
2    loop0:   gvar__0 := [mem_load]("int32", 1-nth(p, 0i), 1-nth(p, 1i));
3            gvar__1 := "i__binop_cmp_gt"(gvar__0, [ "int", 0i ]);
4            gvar__2 := "i__bool_of_value"(gvar__1);
5            goto [gvar__2] then0 else0;
6    then0:   skip;
7            goto endif0;
8    else0:   goto blockend0;
9    endif0:  skip;
10           gvar__3 := "i__unop_longofint"([ "int", 1i ]);
11           gvar__4 := "i__binop_mull"([ "long", 4i ], gvar__3);
12           gvar__5 := "i__binop_addl"(p, gvar__4);
13           p := gvar__5;
14    blockend0: skip;
15            ret := p;
16            return
17  };

```

Figure 13.2: Translation of the C#minor code in Figure 13.1 into GIL.

specifications. Gillian-C currently makes use of 93 internal functions, performing binary and unary operations, as well as some primitive operations such as global variable access. We also use internal functions to encode built-in standard library functions, such as `malloc` or `memmove`.

Let us inspect one particular internal function, `i__binop_cmplu_lt`, which compares two unsigned long integers, returning 1 if the first is less than the second, and 0 otherwise. To explain its implementation, we first present a simplified³ Coq definition of the comparison operator in the mechanised CompCert semantics, given in Figure 13.3.

³ We specialize the code to the operator and architecture at hand.

```

Definition cmplu_lt_bool (v1 v2: val): option bool :=
match v1, v2 with
| Vlong n1, Vlong n2 => Some (Int64.cmpu_le n1 n2)
| Vptr b1 ofs1, Vptr b2 ofs2 =>
  if eq_block b1 b2
    && weak_valid_ptr b1 ofs1
    && weak_valid_ptr b2 ofs2
  then Some (Int.le ofs1 ofs2)
  else None
| _, _ => None
end.

```

Figure 13.3: Simplified Coq definition of the less-than operator for long integers in CompCert.

The function `cmplu_le_bool` takes two values `v1` and `v2`, and returns `Some true` if the values can be compared, and `v1` is less than `v2`, `Some false` if the values can be compared, and `v1` is greater or equal to `v2`, and `None` if the values cannot be compared. The latter case corresponds to an undefined behaviour in C, according to the C standard⁴. In particular, it specifies that pointer comparison using `<` or `>` is only valid if the pointers are part of the same object (that is, in the CompCert memory model, if their blocks are equal), and that both pointers must either be within the object, or one past the end of the object. In Coq, the function `eq_block` checks that two blocks are equal, and `weak_valid_ptr` checks that a pointer is either within the bounds of the object, or one past the end of the object.

⁴ International Organization for Standardization, *ISO/IEC 9899:2018 - Information technology – Programming languages – C*, 2018 [Int18]

The internal function `i__binop_cmplu_lt` is implemented in GIL as in Figure 13.4, and follows the same logic as the Coq definition. Note, in particular, the use of the state model action `mem_weakvalidptr` to efficiently check that the pointers are weakly valid.

Also note how type checking is performed in the GIL code: all values

```

proc i__binop_cmplu_lt(v1, v2) {
  goto [ (l-nth(v1, 0i) = "long") and (l-nth(v2, 0i) = "long") ]
  blon els;
blon: ret := "i__value_of_bool"(l-nth(v1, 1i) < l-nth(v2, 1i));
  return;
els: goto [
  (typeOf(l-nth(v1, 0i)) = Obj)
  and (typeOf(l-nth(v2, 0i)) = Obj)
  and (l-nth(v1, 0i) = l-nth(v2, 0i))
] smbl unde;
smb1: t1 := [mem_weakvalidptr](l-nth(v1, 0i), l-nth(v1, 1i));
  t2 := [mem_weakvalidptr](l-nth(v2, 0i), l-nth(v2, 1i));
  goto [ t1 and t2 ] cmpr unde;
cmpr: ret := "i__value_of_bool"(l-nth(v1, 1i) < l-nth(v2, 1i));
  return;
unde: fail[comparison]("Cannot compare non-comparable values")
};

```

Figure 13.4: Implementation of the internal function `i__binop_cmplu_lt` in GIL.

are expected to be pairs and checking for the first element to be a string indicating the type, or a location (of type `Obj`) is used to perform dynamic type checking.

13.2 Compiling C assertions

Since CompCert does not support an assertion language, we have developed a simple assertion language designed for easy compilation into GIL. Specifications and predicates are parsed separately from the rest of the program, and are compiled into GIL logic annotations in a pipeline separate from the code compilation.

In addition, the Gillian-C front-end generates a user-defined predicate for each structure defined in the program. This predicate captures the layout and basic types for the fields of the structure, including padding between the fields. This enables the use of points-to predicates with structure types in the assertions.

For instance, consider the array structure from the running example of Chapter 11, provided again in Figure 13.5. It contains three fields: a nullable pointer to an array of integers `buffer`, and two size fields `capacity` and `size`.

```

pred struct_Array(+ptr, buffer, capacity, size) :
  ptr == [ #b, #o ] *
  ((#b, #o) -int64-> buffer) * is_ptr_or_null(buffer) *
  ((#b, #o + 8i) -int64-> capacity) * is_long(capacity) *
  ((#b, #o + 16i) -int64-> size) * is_long(size)

```

The generated predicate describes the layout of the `Array` structure in memory and gives basic typing information: it states that an `Array`, starting from the position given by the pointer `ptr`, occupies 24 bytes in memory ($8 + 8 + 8$, given by the type annotation `int64`), with the first 8 bytes taken by `buffer` which is either a pointer or `NULL`, and the other 16 bytes taken by `capacity` and `size`, both of which are of type `long`.

```

struct Array {
  int *buffer;
  size_t capacity;
  size_t size
}

```

Figure 13.5: A dynamic array structure in C.

Chapter 14

Evaluation

The three analyses exposed by Gillian-C have been extensively evaluated. First, we have used the Collections-C library¹, an open-source library of generic data-structure for C programs, to evaluate both whole-program symbolic testing and UX specification synthesis. Second, we have verified the section of the AWS Encryption SDK for C² that implements parsing of the encryption header³. In this chapter, we report on the results of these evaluations. The performance results provided in Table 14.1 have been obtained on a MacBook Pro 2024 with an Apple M4 Max chip and 128Gb of RAM, and all the other performance results MacBook Pro 2019, with a 2.3GHz 8-Core Intel Core i9 processor and 16GB of RAM.

¹ Panić, *srdja/Collections-C*, 2024 [Pan24]

² Amazon Web Services, *aws/aws-encryption-sdk-c*, 2024 [Ama24b]

³ Amazon Web Services, *AWS Encryption SDK message format reference - AWS Encryption SDK*, 2024 [Ama24a]

14.1 Collections-C

Collections-C⁴ is a real-world data-structure library for C with 2.7K stars on Github. The version we analyse⁵ has approximately 5.2K lines of code and uses C-specific constructs and idioms including structures and pointer arithmetic. The data structures it provides include, for example, arrays, lists, treetables, hashtables, ring buffers and queues.

⁴ Panić, *srdja/Collections-C*, 2024 [Pan24]

⁵ Commit hash: 584e113e, December 2019, when the case study was first performed.

Library	Tests	GIL Cmds	Gillian-C Time	CBMC Time	CBMC S/T/F
array	21	109,290	16.61s	131.07s	18/2/ 1
deque	34	106,737	29.2s	340.78s	29/5/0
list	37	730,655	31.60s	172.55s	35/2/0
pqueue	2	15,726	1.99s	2.53s	0/0/ 2
queue	4	39,828	3.63s	4.91s	4/0/0
rbuf	3	27,284	2.66s	3.55s	3/0/0
slist	37	325,383	32.37s	47.39s	37/0/0
stack	2	5,211	1.61s	2.19s	2/0/0
treerset	6	108,583	5.59s	235.93s	2/4/0
reettbl	13	618,326	12.68s	585.00s	0/13/0
Total	159	2,097,023	137.95s	1525.90s	130/26/3

Table 14.1: Results of whole-program symbolic testing of Collections-C, by CBMC and Gillian-C. We used CBMC 5.95.1 with the following arguments:
--bounds-check --pointer-check
--div-by-zero-check
--pointer-primitive-check
--havoc-undefined-functions
--unwind 10 --os macos
--arch x86_64 --function main
--drop-unused-functions

14.1.1 Whole-program symbolic testing

We wrote an extensive symbolic test suite for Collections-C, with results shown in Table 14.1. We report, per data structure: (1) the number of symbolic tests; (2) the number of executed GIL commands; (3) the obtained testing times for CBMC; (4) the obtained testing times for Gillian-C; (5) the testing times for CBMC; and (6) The number of successful Success/Timeout/Failure with CBMC (all tests pass with Gillian-C). The tests were executed on the same machine, and CBMC

was passed a set of arguments that should provide similar guarantees to those of Gillian-C, apart from the detection of uninitialised memory access, which CBMC does not support. Note that the times reported for CBMC include the time spent on tests that timed out.

Performance and comparison with CBMC. Since all the Gillian-C tests run in at most 1.2 seconds, we gave CBMC a timeout of 45s. Using this value, CBMC times out on 26 out of the 161 symbolic tests.

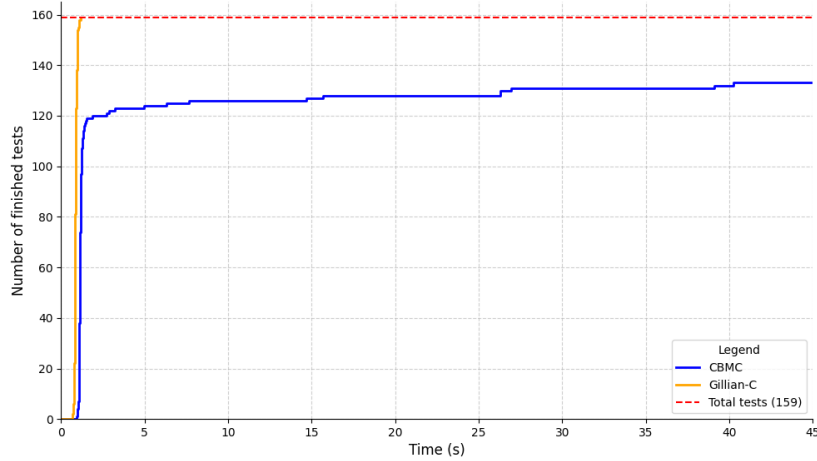


Figure 14.1: Survivor plot for the whole 45s duration

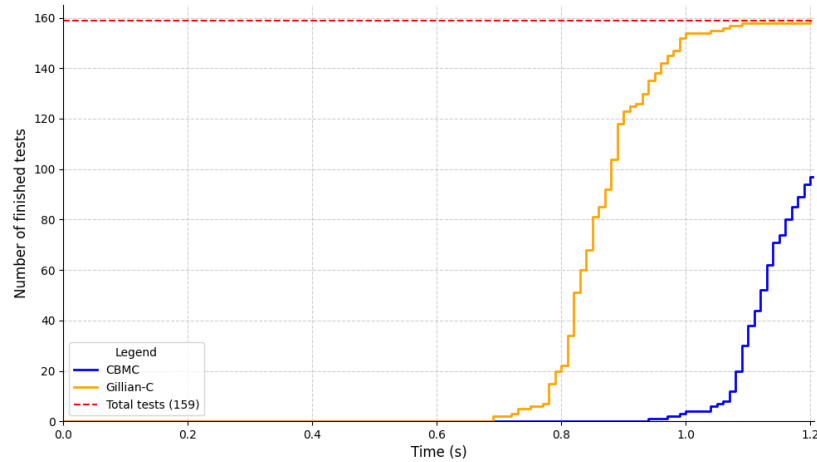


Figure 14.2: Survivor plot for the first 1.2s, until Gillian is done

Because of timeouts, [Table 14.1](#) is difficult to read. To overcome this issue, we draw survivor plots in [Figures 14.1](#) and [14.2](#), providing a more fine-grained visual representation of the performance results. On both plots, the x-axis represents the time in seconds, and the y-axis represents the number of tests that have been completed by that time by Gillian-C and CBMC. The plot on the left shows the results for the full 45s duration, while the plot on the right zooms in on the first 1.2s, which is the time it takes for Gillian-C to complete all the tests. By the time Gillian-C finishes, CBMC has completed 94 out of 159 (60%) tests. In addition, it can be seen that approximately 120 tests run in CBMC almost as fast as in Gillian-C, while the remaining tests seem to run much slower.

The script for running the CBMC tests can be found on Github, in the `collections-c-for-gillian` repository, together with concrete test cases constructed from the counter-examples provided by CBMC for each of the false positives.

CBMC failed to pass three symbolic tests that were successfully checked by Gillian-C. Out of these three, one was a false-positive, and two were due to a bug that was not caught by Gillian-C, revealing an unsoundness in the tool due to a missing overflow check in internal function that performs unsigned division. This confirms the nature of Gillian-C as a prototype in comparison with the battle-tested CBMC.

While the results appear heavily favourable towards Gillian-C, they should be taken with a pinch of salt. The symbolic tests were written with Gillian-C in mind, and CBMC was only employed for comparison after the test cases had been established. It is possible that some of these tests are biased in favour of the Gillian-C engine.

Bugs found by Gillian-C Our testing has revealed the following issues, which have been fixed by the maintainer of Collections-C⁶:

1. a buffer overflow bug in the implementation of dynamic arrays, caused by an off-by-one index;
2. usage of undefined behaviours (pointer comparison) that can lead to buggy behaviours in the presence of compiler optimisations;
3. several bugs in the concrete test suite: in particular, comparing freed pointers, unchecked function returns, and incorrect checks with serendipitously correct values;
4. over-allocation in the ring-buffer data structure (allocating eight times too much memory), but with correct behaviour of the associated functions⁷;
5. a bug in the string hashing function for hashtables that could lead to a performance loss.

14.1.2 Specification generation with bi-abduction

We have used Gillian-C to synthesise specifications for 364 functions of the Collections-C library, producing 5846 specifications in 79.70 seconds. We present the results in [Table 14.2](#), which features, per data structure: the number of associated functions; the number of corresponding GIL instructions (GIL is the intermediate language used by Gillian); the number of success and error specifications; and the analysis time. The results have been obtained with the loop/recursion unrolling bound set to 3, and we believe that they are promising both in terms of performance and number of specifications synthesised.

Examples of Generated Specifications We now give examples of specifications generated by our bi-abduction on one function of the array library. In Collections-C, an array structure contains a size and a capacity, both of type `size_t`. Further, it contains an expansion factor, which specifies how to increase array capacity when the array is full and we need to add a new element. Finally, it contains a pointer to an array of pointers to `void`, an artefact of the lack of polymorphism in C.

⁶ Ayoun, *fix djb2 string hash function - Collections-C - Github*, 2019 [[Ayo19b](#)]; Ayoun, *remove the usage of `cc_comp_ptr` everywhere - Collections-C - Github*, 2019 [[Ayo19e](#)]; Ayoun, *fix over allocation of ring_buffers - Collections-C - Github*, 2019 [[Ayo19c](#)]; Ayoun, *Fix some tests in the list test suite - Collections-C - Github*, 2019 [[Ayo19d](#)]; and Ayoun, *fix buffer overflow - Collections-C - Github*, 2019 [[Ayo19a](#)]

⁷ The over-allocation cannot directly be detected through WPST. It was detected by inspecting the logs of execution that draw a visual representation of the symbolic heap trees. Writing in the ring-buffer would always leave an uninitialised leaf at the right of the tree.

Library	Fcts.	GIL Inst.	Succ. Specs	Err. Specs	Time (s)
array	45	1784	251	260	1.36
deque	47	2312	271	210	2.65
hashset	14	160	7	112	12.01
hashtable	28	1527	31	147	17.01
list	66	2977	454	615	5.63
pqueue	10	557	90	51	4.71
queue	16	85	133	67	1.96
rbuf	9	181	9	17	0.06
slist	52	2269	292	1873	31.79
stack	16	85	136	88	0.57
treerset	17	214	28	106	0.40
treetable	36	1601	144	276	1.51
other	8	139	14	11	0.03
Total	364	13891	1860	3833	66.62

Table 14.2: Aggregated results of synthesising function specifications for the Collections-C library with unrolling bound set to 3.

The structure definition in the original implementation of the library also contained 3 additional function pointers to allow for the users of the library to provide custom implementations of the `malloc`, `calloc` and `free` functions. As this is higher-order in nature⁸, to be able to perform the analysis we removed these pointers from the structure and replaced any calls to these functions with their `stdlib` counterparts. We performed similar changes to any other structure containing allocator function pointers. The approach of Infer:Pulse to this issue is to essentially havoc the return value and the pointer arguments; we could have taken that approach as well.

```

typedef struct array_s {
    size_t size;
    size_t capacity;
    float exp_factor;
    void **buffer;
    // Modified to remove
    // allocator pointers.
} Array;

cc_enum array_get_at(
    Array *ar, size_t index, void **out
)
{
    if (index >= ar->size)
        return CC_ERR_OUT_OF_BOUND;
    *out = ar->buffer[index];
    return CC_OK;
}

```

The function `array_get_at` which retrieves an element from a specific index in the given array at address `ar`; if the index is within the array's bounds, it places the element's address into the output parameter `out` and returns the `CC_OK` success code, otherwise, it returns a code signalling an out-of-bounds error. Gillian generates 9 specifications for this function, 4 successful and 5 erroneous ones. In Table 14.3, we detail 2 successful and 1 erroneous specification, given in readable high-level syntax, using field notation instead of explicit offsets and eliding type annotations from points-to assertions as they are not the focus of this discussion. Recall also that variables in the pre-condition that are not formal arguments are implicitly universally quantified. In all three specifications, we highlight in red the fragments that are modified between the pre- and the post-condition. We also highlight in purple some extraneous conditions bi-abduced by the Gillian-C engine, in the form of `_ = NULL`, which we discuss shortly.

Spec 1 corresponds to a successful execution of the function: the size check passes, and the accessed cell and the `out` pointer are properly allocated. The content pointed to by the `out` pointer is overridden

⁸ Gillian-C supports function pointers, but at call site, the function pointer must be statically resolvable to a concrete function (in the current symbolic execution branch).

Spec 1: Successful access	$\left[\begin{array}{l} \text{ar.size} \mapsto s * \text{ar.buffer} \mapsto s * (s + \text{index}) \mapsto v * \text{out} \mapsto o * o = \text{NULL} \\ \text{cc_enum array_get_at}(\text{Array} * \text{ar}, \text{size_t index}, \text{void} ** \text{out}) \\ \text{Ok} : r. \text{ar.size} \mapsto s * \text{index} < s * \text{ar.buffer} \mapsto s * (s + \text{index}) \mapsto v * \\ \text{out} \mapsto v * o = \text{NULL} * r = \text{CC_OK} \end{array} \right]$
Spec 2: Graceful out-of-bounds	$\left[\begin{array}{l} \text{out} = \text{NULL} * \text{ar.size} \mapsto s \\ \text{cc_enum array_get_at}(\text{Array} * \text{ar}, \text{size_t index}, \text{void} ** \text{out}) \\ \text{Ok} : r. \text{out} = \text{NULL} * \text{a.size} \mapsto s * \text{index} \geq s * r = \text{CC_ERR_OUT_OF_BOUNDS} \end{array} \right]$
Spec 3: NULL dereference	$\left[\begin{array}{l} \text{ar} = \text{NULL} * \text{out} = \text{NULL} \\ \text{cc_enum array_get_at}(\text{Array} * \text{ar}, \text{size_t index}, \text{void} ** \text{out}) \\ \text{Err} : r. \text{ar} = \text{NULL} * \text{out} = \text{NULL} * r = \text{"segmentation fault"} \end{array} \right]$

Table 14.3: Examples of specifications generated by Gillian-C’s bi-abduction.

and the function returns `CC_OK`. Note the minimal footprint, in that the function requires only the `size` and `buffer` fields of the structure and therefore only those fields are bi-abduced.

Spec 2 corresponds to a case where the `size` field of the array structure indicates that the accessed index is out of bounds. The library gracefully handles this by returning the appropriate error code.

Spec 3 corresponds to the case where the input pointer equals `NULL`, which triggers a `NULL` dereference, which Gillian-C detects as an error. This spec is perhaps the most important, as it is the one which, when propagated through the codebase by function calls, would allow a front-end that can filter true bugs to detect an issue.

On Specification Duplication Above, we presented 2 out of the 4 successful specifications generated by Gillian-C; in the other two, the conditions highlighted in purple would be of the form `_! = NULL`. This duplication is an unfortunate by-product of how the state model of Gillian-C currently encodes values in its symbolic heap and the fact that type information lost in compilation from C to GIL: bi-abducting the "shape" of the value (`NULL`, or not `NULL`) becomes necessary to preserve memory well-formedness. This phenomenon leads to an explosion in the number of generated specifications, especially when many different memory cells containing pointers are accessed in a row. In particular, `Collections-C` exposes several iterator structures composed of many pointers, such as `slist_zip_iter`, `hashtable_iter` and `hasht_iter`. For each of these structures, the library also exposes an initialiser function, which assigns each field one by one, leading to a path explosion. In fact, 73% of the execution time in bi-abduction is spent on 3 of the 363 functions, yielding 1640 specifications.

As part of an internship at AWS, the author of this manuscript implemented an alternative front-end for Gillian-C based on CBMC⁹, with an improved state model that could make use of the type information preserved by this new front-end and which encoded symbolic values in memory in such a way that this limitation was effectively lifted. However, we cannot run bi-abduction with this improved back-end as it does not implement fixes; the reason for this is that it was developed

⁹ The goal was to study the potential of Gillian-C as an alternative back-end to Kani, a Rust bounded model checker based on CBMC. While the repository is open-source [Ayo22], further results of this evaluation are not public.

for comparison with CBMC, which does not perform bi-abduction.

14.2 The AWS Encryption Header case study

The AWS Encryption SDK¹⁰ is an open-source encryption library built by Amazon Web Services that provides an interface for manipulation of encrypted data using the standards supported by AWS. In particular, *messages* are encrypted by the library and are then sent between the client and the services. These encrypted messages have a well-defined format¹¹ and comprise a header, a body, and a footer.

As part of the evaluation of Gillian-C and Gillian-JS, we verified the modules of the C and JS SDKs that are responsible for parsing the header of the encrypted messages. To do so, a single *pure* abstraction was written in GIL that describes the format of the header, and two sets of separation logic specifications were written for C and JS using their respective state models and the language-agnostic pure specification. Here, we only report on the results of the verification of the C module; more information on the verification of the JS module can be found in the corresponding paper¹².

¹⁰ Amazon Web Services, *aws/aws-encryption-sdk-c*, 2024 [Ama24b]

¹¹ Amazon Web Services, *AWS Encryption SDK message format reference - AWS Encryption SDK*, 2024 [Ama24a]

¹² Maksimović et al., “Gillian, Part II: Real-World Verification for JavaScript and C”, 2021 [Mak+21]

14.2.1 Description and specification

The AWS Encryption SDK message header is a sequence of bytes (buffer) divided into sections, as illustrated in Figure 14.3; above each section is its length in bytes.

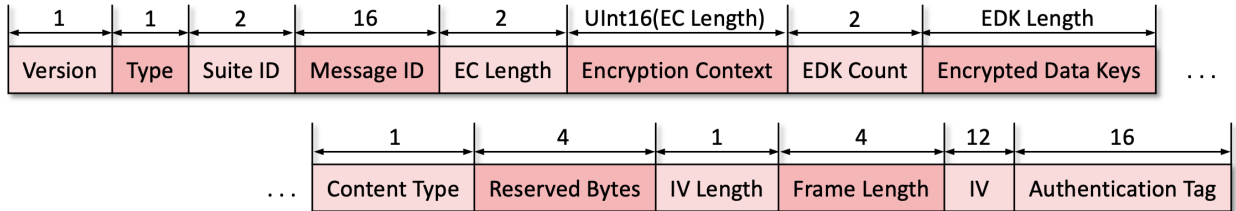


Figure 14.3: Diagram of the Encryption Header structure.

Our approach is to abstract the header contents into a list and formulate pure predicates that describe its structure in a language-independent way. This allows us to then use the same abstractions as part of further, language-dependent, abstractions for both C and JS. Our design of the abstractions was informed by existing code annotations found in the implementations, which describe simple first-order properties of the code and, in the case of C, are checked by CBMC in the AWS CI. However, these annotations are limited by the expressivity of JS and C, particularly when it comes to reflecting on the memory contents. Our predicates have no such limitations.

We narrow down our exposition to the encryption context, as it illustrates well the language-independent and language-dependent aspects of our specification, and is also the section in which we discovered bugs in both implementations.

Pure specification of the Encryption Context The encryption context (EC) is a sequence of bytes that describes a set of key-value pairs. Its structure is given in the diagram in Figure 14.4.

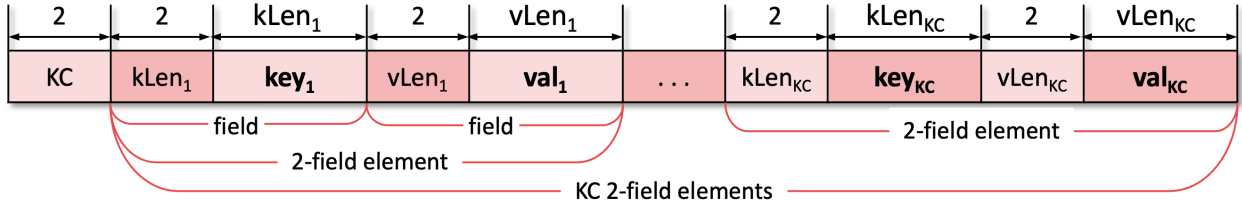


Figure 14.4: Diagram of the Encryption Context structure.

The first two bytes represent the number of key-value pairs, denoted by KC , and the rest describe the KC key-value pairs themselves. Keys and values are represented by sequences of bytes and, as they are of variable length, are serialised by first having two bytes that represent the length, followed by *that many bytes* of the actual key or value; we refer to this pattern as a *field*, and to a sequence of n fields as an n -element. Then, a key-value pair is serialised as a 2-field element, and all of the key-value pairs form a sequence of KC 2-field elements.

We specify the EC by building layers of abstraction, from fields to elements to element sequences to the EC, each of which can either be complete, incomplete (partial, but with correct structure), or malformed (with incorrect structure). In the implementation, these are specified separately and are joined together in appropriate over-arching abstractions. Here, we focus on *complete* variants only.

The $\text{Field}(\text{buf}, \text{pos}, \text{fld}, \text{len})$ predicate, given below, states that the buffer (list of bytes) buf , at index pos , holds a field with contents fld (list of bytes) and total length len :

```
pred Field(buf, pos, fld, len) :
  (0 <= pos) * (#rFL = sub(buf, pos, 2)) *
  UInt16(#rFL, #fL) * (fld = sub(buf, pos+2, #fL)) *
  (len = 2+#fL) * (pos+len <= |buf|)
```

This predicate uses the GIL operator $\text{sub}(l, s, n)$, which returns the sublist of list l starting from index s and of length n , and also the $\text{UInt16}(\text{rn}, n)$ predicate, which states that n is a 16-bit big-endian interpretation of the raw 2-byte list rn .

The $\text{Element}(\text{buf}, \text{pos}, \text{fC}, \text{elem}, \text{len})$ predicate states that buffer buf at index pos holds a sequence of fC fields, with contents elem (a list of the appropriate field contents) and total length len . It is defined similarly to a standard linked-list predicate, with the ‘link’ being the fact that the list members are contiguous in memory.

```
pred Element(buf, pos, fC, elem, len) :
  (fC = 0) * (0 <= pos) * (pos <= |buf|) * (elem = [ ]) * (len = 0);
  (0 < fC) * Field(buf, pos, #fld, #fL) * Element(buf, pos+#fL, fC-1, #rFs,
  #rL) * (elem = #fld :: #rFs) * (len = #fL+#rL)
```

Next, analogously to the Element predicate, we define the $\text{Elements}(\text{buf}, \text{pos}, \text{eC}, \text{fC}, \text{elems}, \text{len})$ predicate, which states that the buffer buf , at index pos , holds a sequence of eC elements, each with fC fields, with contents elems (a list of the appropriate element contents) and of total length len . Finally, the $\text{EncryptionContext}(\text{buf}, \text{KVs})$

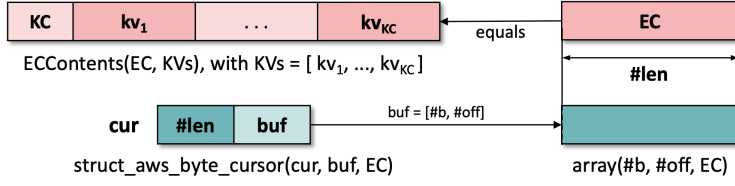


Figure 14.5: Serialised Encryption Context: language-independent pure part (red) and language-specific resource (green).

predicate states that the entire buffer `buf` is an EC with key-value pairs KVs, with all of the keys being unique:

```
pred EncryptionContext(buf, KVs) : (buf = [ ]) * (KVs = [ ]);
  (#rKC = sub(buf, 0, 2)) * UInt16(#rKC, #KC) * (0 < #KC) *
  Elements(buf, 2, #KC, 2, KVs, #len) *
  FirstProj(KVs, #Ks) * Unique(#Ks) * (2 + #len = |buf|)
```

Next, we show how this pure specification of the EC can be connected *without modification* to the state model; its connection to the JS state model is elided and can be found in the corresponding paper.

Encryption Context in C The EC is serialised as a block in memory, and is traversed using an AWS byte cursor. Using the auto-generated `struct_aws_byte_cursor` predicate given in Chapter 13, we define the `aws_byte_cursor(cur, buf, c)` predicate, stating that `cur` points to a byte cursor which has access to an array starting from `buf`, and holding contents `c`, making the length implicit:

```
pred aws_byte_cursor(cur, buf, c) :
  struct_aws_byte_cursor(cur, #len, buf) * (buf = [#b, #off]) *
  array(#b, #off, c) * (#len = |c|)
```

A serialised EC is then described as a byte cursor whose contents represent the EC key-value pairs (cf. Figure 14.5, centre and bottom):

```
pred CSerEC(cur, buf, EC, KVs) :
  aws_byte_cursor(cur, buf, EC) * EncryptionContext(EC, KVs)
```

Finally, we are able to specify the deserialisation function as follows:

```
{ empty_hash_table(ec) * CSerEC(cur, #buf, #EC, #KVs) }
```

```
int aws_cryptosdk_enc_ctx_deserialize(
  struct aws_hash_table *ec, struct aws_byte_cursor *cur)
```

```
{ (ret = 0) * CDeserEC(ec, #KVs) * (#buf = [#b, #off]) *
  array(#b, #off, #EC) * aws_byte_cursor(cur, #buf + p |#EC|, [ ]) }
```

The specification states that the EC is deserialised into an AWS hash table, whose keys and values directly correspond to the key/value pairs of the EC, captured as follows, eliding the internal structure of the hash tables for conciseness:

```
pred CDeserEC(ht, KVs) : valid_hash_table(ht, KVs)
```

It also states that the byte cursor that originally pointed to the EC ends up shifted to the end of the byte buffer, exposing the array underneath the `CSerEC` predicate.

14.2.2 Verification

Using Gillian-JS and Gillian-C, together with the specifications given in the previous subsection, we verify full functional correctness of the

header deserialisation module of the AWS Encryption SDK JS ($\sim 200\text{loc}$) and C¹³ ($\sim 950\text{loc}$) implementations. In particular, we verify that the deserialisation of a complete header is correct, and the deserialisation of an incomplete or a malformed header raises an appropriate error. Here, we only report results about the C-related part of the verification.

Verification Effort and Performance The C verification took 3 person-months: about 1 month for writing the pure specifications characterising the header format, and 2 months to verify the C code itself. The first-order solver of Gillian was substantially improved to reason automatically about complex operations on lists of symbolic length, which are used to reason about the array content of the tree nodes in the C memory model. We created a collection of language-independent predicates and lemmas about their inductive properties ($\sim 1.2\text{kloc}$) that cover the project-specific AWS header, but also re-usable first-order concepts such as list element uniqueness, projections of lists of pairs, conversion from bytes to numbers, and conversion from raw bytes to strings. Similarly, we also had to create language-dependent abstractions and associated lemmas for the C manipulation of the AWS message header ($\sim 1.2\text{kloc}$). Finally, we had to: annotate the code with specifications and loop invariants, with the latter often having more than twenty components; manually apply lemmas to prove numerous complex entailments; and manually unfold user-defined predicates at times (the folding is automated) ($\sim 1.1\text{kloc}$).

On our machine used to perform the experiments, the C verification takes $\sim 221\text{s}$. Interestingly, this is substantially longer than the JS verification (about 45s), in part due to the larger codebase, but mainly due to the complexity of the implementation of the full C memory model, which is able to reason about the heap at the byte-level, while JS is more abstract.

Bugs found We discovered three bugs: one logical error; one undefined behaviour; and one over-allocation.

- The deserialisation of the EC mishandled the case when there is not enough data in the buffer to read it entirely, continuing to read the next part of the header, the EDK, instead of reporting an error. This allows some malformed headers to be parsed as well-formed¹⁴, potentially leading to security vulnerabilities.
- The function `aws_byte_cursor_advance`, when called with a NULL cursor and a length of 0, resulted in `NULL + 0` being computed, which is undefined behaviour, although not problematic for most compilers¹⁵.
- The deserialised EC was stored using `aws_string`, which extends C strings with certain metadata. It is implemented using a structure that includes a flexible array member. We discovered that string creation, in the AWS C standard library, over-allocated this array by 8 bytes. It was detected because our (correct) predicate describing `aws_strings` was not allowing the verification to go through¹⁶, claiming that the expected bound of the block did not match the actual bound of the block.

¹³ Amazon Web Services, *aws/aws-encryption-sdk-c*, 2024 [Ama24b]

¹⁴ Ayoun, *Correctly fail on invalid aad length - aws-encryption-sdk-c - Github*, 2021 [Ayo21a]

¹⁵ Ayoun, *Undefined Behaviour corner case for aws_byte_cursor_advance - aws-c-common - Github*, 2021 [Ayo21c]

¹⁶ Ayoun, *Small over allocation in each aws_string - aws-c-common - Github*, 2021 [Ayo21b]

All of the bugs were (eventually) fixed by the maintainers of the AWS SDK C library¹⁷.

Verification: Caveats Our C verification is correct up to the following caveats. First, we do not use the `aws_byte_cursor_advance_nospec` function, which advances the byte cursor, but also uses complex computation to protect against the Spectre bug. We instead use `aws_byte_cursor_advance`, which has equivalent behaviour, as our specifications are not expressive enough to capture this distinction. Next, we axiomatise the functions of the AWS hash tables and array list libraries, as their verification is of comparable complexity to the entire deserialisation module. Finally, the AWS allocators of the C implementation, which are passed into some of the functions, contain pointers to memory management functions; this is higher-order in nature. In the verification, we assume that those functions are `malloc`, `calloc`, and `realloc`.

14.3 Limitations

While the results of the evaluation are promising, there still exist several limitations that need to be mentioned. Importantly, these limitations are not fundamental to the approach, but rather to the current implementation of Gillian-C.

First, the support of language constructs is intrinsically limited by the support of the CompCert front-end. While CompCert supports a large subset of C, its support for some rare idioms is missing. For instance, switch statements do not support leading commands, restricting the support for, e.g., Duff’s device. The complete list of features unsupported by CompCert is available on the CompCert website¹⁸.

Next, Gillian-C does not soundly support floating point arithmetics, and instead models floats as real numbers. This is a limitation of the current implementation of Gillian’s core and could be lifted by using a more expressive encoding of floating points using bit-vectors.

Next, Gillian-C does not support concurrency, in that its current state model does not implement fractional permissions or any other form of shared-memory reasoning. In addition, while all specifications used are written in separation logic, and hence compatible with exclusive-ownership concurrency, we have never experimented with it in Gillian-C.

Finally, Gillian-C does not support effective types, a feature of the C standard that disallows arbitrary casting of pointers. This is a choice that we made in order to better match the semantics of CompCert, as, in practice, developers often make use of compiler flag that disable this restriction, and those who don’t are likely to be unaware of the consequences of arbitrary pointer casting. However, this could be considered a limitation, as Gillian-C is unable to detect bugs that arise from the misuse of effective types.

¹⁷ The potential security issue was fixed within two weeks, but the over-allocation remained in the codebase for more than 3 years.

¹⁸ Leroy, *The CompCert C Compiler*, 2023 [Ler23]

Chapter 15

Related work

There exists a wide variety of tools that can be applied to C code; here, we focus on those that are able to perform similar analyses to those offered by Gillian-C, and do not address in detail tools that employ techniques such as sanitising¹, fuzzing², and abstract interpretation³. To our knowledge, there is no single tool that offers all three (or any two) analyses provided by Gillian-C.

15.1 Whole-program symbolic testing

There are many mature symbolic execution tools for C⁴ which follows the dynamic-concolic discipline pioneered by Dart⁵. Such tools pair up symbolic execution and concrete execution to allow the symbolic execution to fall back to the concrete whenever it produces symbolic formulae unsupported by the underlying constraint solver. While these techniques make powerful bug-finders, they do not provide the same bounded correctness guarantees as Gillian-C does in WPST mode.

The guarantees provided by Gillian-C are similar to those provided by bounded model checkers like CBMC⁶. Unlike Gillian-C, CBMC performs *symbolic compilation*, where entire programs are compiled into a single boolean SAT formula. This formula is then sent to an SMT solver to check for satisfiability, and if the formula is satisfiable, a counterexample is returned. The comparison between the two tools is given in §14.1.1, but a more detailed evaluation is required to precisely determine the relative strengths and weaknesses of each tool.

15.2 Compositional verification using SL

The history of compositional verification tools for C based on separation logic dates back to 2009 and coreCStar⁷; we focus on the more contemporary tools, starting from VeriFast⁸, first published in 2011.

VeriFast VeriFast by default targets a semantics of C that is closer to the C standard than that targeted by Gillian-C, with support for, e.g., effective typing. It also provides options for disabling such restrictions and target a more relaxed semantics. VeriFast is also more mature and battle-tested than Gillian-C: it has been used for a larger number of case studies, including some that made extensive use of concurrency, although none was larger in size than our AWS case study.

On the other hand, VeriFast does not provide the same level of automation as Gillian-C, aiming instead at performance and predictability. The array push example, from §11.3, illustrates well the trade-off between speed and automation. On that example, Gillian-C is slower

¹ Nethercote et al., “Valgrind: a framework for heavyweight dynamic binary instrumentation”, 2007 [NS07]; and GNU GCC developers, *Program Instrumentation Options - GNU GCC*, 2024 [GNU24]

² Haller et al., “Dowsing for overflows: a guided fuzzer to find buffer boundary violations”, 2013 [Hal+13]

³ Cousot et al., “The ASTREE Analyzer”, 2005 [Cou+05]; and Kirchner et al., “Frama-C: A software analysis perspective”, 2015 [Kir+15]

⁴ Cadar et al., “KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs”, 2008 [CDE08]; Cadar et al., “EXE: Automatically Generating Inputs of Death”, 2008 [Cad+08]; Godefroid et al., “Automated White-box Fuzz Testing”, 2008 [GLM08]; Godefroid et al., “Compositional May-Must Program Analysis: Unleashing The Power of Alternation”, 2009 [GNR09]; and Ramos et al., “Under-Constrained Symbolic Execution: Correctness Checking for Real Code”, 2015 [RE15]

⁵ Godefroid et al., “DART: directed automated random testing”, 2005 [GKS05]

⁶ Clarke et al., “A Tool for Checking ANSI-C Programs”, 2004 [CKL04]

⁷ Botinčan et al., “Separation Logic Verification of C Programs with an SMT Solver”, 2009 [BPS09]

⁸ Jacobs et al., “VeriFast: A Powerful, Sound, Predictable, Fast Verifier for C and Java”, 2011 [Jac+11]

than VeriFast but requires no annotations, while VeriFast requires 4 lines of annotation. When it comes to predictability, our experience confirms that VeriFast is more predictable than Gillian-C. For example, there were cases where *removing* an annotation would make a Gillian-C proof go through substantially faster, as the annotation was working against the built-in automation; this could not happen in VeriFast.

RefinedC RefinedC⁹ is a C verifier based on CompCert, targeting a semantics of C that is close to that of Gillian-C. Based on Lithium¹⁰, it achieves the impressive feat of verifying C code semi-automatically while producing foundational proofs that are checked by Coq. This makes RefinedC proofs more trustworthy than that of Gillian-C and also allows the user to wield the full power of an interactive theorem prover for proving properties about the code. RefinedC, however, requires roughly the same amount of annotations as VeriFast, but these annotations are often much more complex than the ones required by VeriFast or Gillian-C, as they require knowledge of the underlying Coq proof; sometimes, the user is even required to write Coq code directly. Finally, RefinedC is slower than Gillian-C, and has only been applied to a small (~190 loc) case study adapted from real-world code.

CN CN, introduced after Gillian-C, is a tool based on the Cerberus semantics, which is possibly the most accurate ISO C semantics available. Its extensive formalisation takes a different approach, formulating verification as a refinement type-checking problem. Its primary aim is predictability, even staying within the domain of decidable SMT theories, and producing Coq statements that must be manually proven by the user when the tool cannot prove them automatically. While the amount of annotation is similar to VeriFast, CN seems to perform slower than either Gillian-C or VeriFast in terms of speed. CN also provides automation for the manipulation of random-access arrays. To do so, however, CN takes an approach similar to Viper’s iterated separating conjunction¹¹, and hence does not provide automation for byte-level reasoning. CN has been used to verify the pKVM buddy allocator, which comprises 364 lines of complex C code.

VST The Verified Software Toolchain is a Coq-based separation-logic verifier for CompCert C programs. The latest version uses a separation logic based on Iris, and can therefore leverage the full expressivity of Coq and Iris to write and prove specifications. In addition, the entire toolchain provides end-to-end verification, from C to the generated assembly code. However, the proof effort is highly manual, requiring at least an order of magnitude more annotations than required by semi-automatic tools.

15.3 Infer:Pulse

To our knowledge, Infer:Pulse¹² is the only tool able to generate under-approximate specifications for C. It is actively used inside of Meta and runs daily on thousands of commit diffs. Furthermore, Infer:Pulse offers

⁹ Sammler et al., “RefinedC: automating the foundational verification of C code with refined ownership types”, 2021 [Sam+21]

¹⁰ Sammler, “Automated and foundational verification of low-level programs”, 2023 [Sam23]

¹¹ Müller et al., “Automatic Verification of Iterated Separating Conjunctions Using Symbolic Execution”, 2016 [MSS16a]

¹² Calcagno et al., “Infer: An Automatic Program Verifier for Memory Safety of C Programs”, 2011 [CD11]; and Le et al., “Finding real bugs in big programs with incorrectness logic”, 2022 [Le+22]

filtering of the generated specifications, allowing automatic detection of bugs that are reachable from the current codebase (the so-called *manifest* bugs). Gillian-C does not implement this functionality.

On the other hand, we were not able to find a detailed explanation of how real-world memory is handled within Infer/Infer:Pulse, as all of the papers treating the subject use a simple linear heap for illustrative purposes. The memory model presented in this part, in contrast, offers detailed insight into how Gillian-C handles the real-world memory model of C and produces fixes for bi-abduction.

Part III

Gillian-Rust

Chapter 16

A challenge

*Started making it, had a breakdown,
bon appétit.*

James Acaster, 2020

The Rust programming language¹ has seen rapid adoption in recent years, particularly in the field of *systems programming*, to the point where it has become only the second language to be adopted by the Linux kernel². The success of Rust is due to its rejection of false dichotomies between safety and performance: its *ownership type system* and *borrow checker* preserve memory safety while avoiding the need for a garbage collector.

As Rust finds its way into more critical systems, the need for stronger formal guarantees about the *behaviour* of Rust programs grows. In response to this need, the past several years have seen the emergence of a number of tools aimed at the verification of Rust programs, such as Aeneas³, Creusot⁴ and Prusti⁵. These tools all leverage the properties of the Rust type system to simplify verification, but all also share a common limitation: they can only verify *safe* Rust code.

Although most Rust code is written in the safe subset of the language, it is common for it to rely on *unsafe* code to interface with the underlying operating system or to provide low-level abstractions. In unsafe code, the programmer gains access to several ‘superpowers’, such as the ability to dereference raw pointers, cast between types, and even access (potentially) uninitialised memory. Unsafe code is an essential part of Rust’s design, allowing new *safe* abstractions, such as `LinkedList<T>` (the type of doubly-linked lists), to be implemented efficiently in libraries. However, unsafe code also comes with greater responsibility: the programmer is now responsible for ensuring that unsafe code does not exhibit undefined behaviour (UB) and that the corresponding APIs remain observationally safe. In addition, despite representing a fraction of the total codebase, unsafe code is often the most complex and error-prone part of a Rust program, making it the most important to formally verify, which none of the above-mentioned tools is able to accomplish.

In this part of the thesis, we propose a **hybrid** approach to end-to-end Rust verification which, mirroring the differences between safe and unsafe code, leverages Creusot for verification of safe code and a novel instantiation of Gillian for Rust, cleverly dubbed *Gillian-Rust*, for verification of unsafe code, which can be specified but not verified by Creusot.

Understanding the substantial challenges that needed to be overcome

¹ Matsakis et al., “The Rust language”, 2014 [MK14]

² Cook, [GIT PULL] *Rust introduction for v6.1-rc1*, 2022 [Coo22]

³ Ho et al., “Aeneas: Rust verification by functional translation”, 2022 [HP22]

⁴ Denis et al., “Creusot: a Foundry for the Deductive Verification of Rust Programs”, 2022 [DJM22]

⁵ Astrauskas et al., “Leveraging rust types for modular specification and verification”, 2019 [Ast+19]

by the implementation of Gillian-Rust requires a background in the foundational theory underpinning Rust. In 2018, Jung et al. published RustBelt⁶, a theoretical framework that allows for semantic interpretation of Rust’s ownership types using the higher-order separation logic Iris⁷, thereby enabling reasoning about type safety. In 2022, the work on RustHornBelt⁸ extended RustBelt with the ability to reason about functional correctness of unsafe Rust code, allowing for safe functions implemented with unsafe code to be given first-order logic specifications and providing the meta-theory that now underpins Creusot. However, both RustBelt and RustHornBelt operate on λ_{Rust} , a model of Rust that makes many simplifying assumptions relevant to a foundational formalisation and therefore cannot capture the intricacies of real Rust. Moreover, RustHornBelt proofs are manually performed in Coq⁹, on code ported by hand from Rust to λ_{Rust} , and little automation is provided to alleviate the boilerplate of verification. As a result, while blazing the trail for end-to-end functional correctness verification of Rust programs, RustHornBelt cannot bring its approach to real-world Rust programs.

More recently, RefinedRust¹⁰ demonstrated how the techniques developed in Refined-C¹¹ could be adapted to RustBelt to reason about functional correctness of Rust programs in a more automated way. However, RefinedRust remains embedded in Coq and its automation and performance are inherently limited by this choice. We argue that in order for verification to tackle the volume of existing and future unsafe Rust code, more efficient and scalable tooling is needed.

Challenge 1: Real Rust is *really* hard Rust is primarily a systems programming language, and therefore comes with every associated complication, some previously known from the large research effort in C verification—e.g., low-level data representation, byte-level value manipulation, and memory allocator manipulation—and some new—e.g., exotically-sized types, such as zero-sized types, compiler-chosen layouts (whereas C has a standardised layout), and polymorphism.

While these aspects of Rust are invisible to safe Rust code, they become a proper concern when working with unsafe code. For example, it is crucial for a verifier to reason generically over the possible memory layouts of programs so that it could detect any potentially disallowed memory operations. This makes the reuse of the existing Gillian-C memory model difficult and requires development of new techniques to reason automatically and efficiently about real Rust and the way it represents objects in memory.

Challenge 2: Type safety, borrows, and raw pointers The notion of type safety in Rust is much stricter than that found in languages like C. Specifically, the responsibility of a safe function, even an internally unsafe one, is not limited to its own body: it must ensure that no fully-safe program calling it may trigger undefined behaviour. This dramatically increases the complexity of integrating unsafe code into a Rust program.

The main tool employed by Rust to guarantee safety is a strict

⁶ Jung et al., “RustBelt: securing the foundations of the Rust programming language”, 2017 [Jun+17]

⁷ Jung et al., “Iris from the ground up: A modular foundation for higher-order concurrent separation logic”, 2018 [Jun+18]

⁸ Matsushita et al., “RustHornBelt: a semantic foundation for functional verification of Rust programs with unsafe code”, 2022 [Mat+22]

⁹ The Coq Team, *The Coq Proof Assistant*, 2023 [The23a]

¹⁰ Gäher et al., “RefinedRust: A Type System for High-Assurance Verification of Rust Programs”, 2024 [Gäh+24]

¹¹ Sammler et al., “RefinedC: automating the foundational verification of C code with refined ownership types”, 2021 [Sam+21]

and static ownership discipline, wherein each value must always have a unique, exclusive owner. While this alone would be too restrictive, Rust also provides mutable references ($\&\kappa_{\text{mut}}T$) and shared references ($\&\kappa T$) which may *borrow* ownership for a *lifetime* κ . However, even when equipped with references, safe Rust is sometimes too restrictive and prevents the implementation of types such as *doubly-linked lists*, where each node is referenced by two pointers at any time (cf. Figure 17.1, bottom left), breaking the exclusive ownership discipline. In such cases, developers must resort to unsafe code in order to manipulate raw pointers ($*\text{mut } T$) which, unlike references, allow for unrestricted aliasing and do not provide any safety guarantees.

This mixed use of raw pointers and safe references causes the task of verifying type safety of unsafe code to (yet again) increase in complexity, as it requires reasoning about lifetime-dependent safety invariants.

Challenge 3: Scaling safe and unsafe Rust verification, together

While unsafe code is used to perform some of the most complex and primitive operations of Rust programs, it still remains a small fraction of the total codebase¹². Furthermore, safe Rust often uses many advanced features, such as higher-order functions, which are eschewed in unsafe code. In that setting, we believe that it would be extremely challenging to build a tool that both has the required expressivity for reasoning about unsafe code, which makes extensive unrestricted use of raw pointers, and can, at the same time, reason *efficiently* and *automatically* about the higher-level features used in safe Rust.

On the other hand, tools such as Creusot¹³ have demonstrated that safe Rust verification can be performed with impressive automation and simplicity, permitting reasoning about some of the most high-level features of Rust¹⁴, but sacrificing the ability to handle unsafe code. Ideally, one would reuse such a tool for analysing safe code, and use another, more adapted tool, for analysing unsafe Rust, splitting the proof effort appropriately. This approach, however, requires both tools to agree on the semantics of *specifications* given to Rust functions. For example, if Creusot is used for safe code, the other tool has to provide a faithful interpretation of Creusot’s specifications, which use a simple-to-write yet complex-to-interpret prophetic assertion language.

Contributions In this part of the manuscript, we present a *hybrid* approach to verification of Rust programs, which leverages the strengths of specialised tools operating in unison to verify both safe and unsafe Rust code, illustrated in Figure 16.1.

In particular, we combine Creusot, an existing tool for safe Rust verification, with Gillian-Rust, a novel proof-of-concept instance of Gillian for the verification of unsafe Rust code. We manage the boundary between the tools through a shared specification language that can easily be interpreted into either Creusot or Gillian-Rust specifications. To make this possible, Gillian-Rust implements and automates the reasoning of RustBelt and RustHornBelt, which allows it to reason about the *prophetic specifications* of Creusot.

We demonstrate the viability of our approach by verifying *actual*

¹² Astrauskas et al., “How do programmers use unsafe Rust?”, 2020 [Ast+20]

¹³ Denis et al., “Creusot: a Foundry for the Deductive Verification of Rust Programs”, 2022 [DJM22]

¹⁴ Denis et al., “Specifying and Verifying Higher-order Rust Iterators”, 2023 [DJ23]

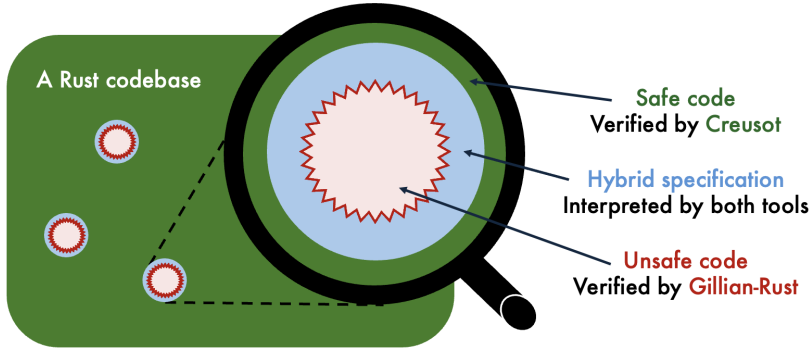


Figure 16.1: Representation of unsafe code in the Rust codebase. The boundary between the tools is managed by a shared specification language that both tools are able to interpret.

code from the Rust standard library—specifically, the `LinkedList` and `Vec` types, along with several other case studies. Our approach performs verification at least two orders of magnitude faster than prior works, made possible by the use of symbolic execution and the efficient memory model of Gillian-Rust.

Building Gillian-Rust on top of Gillian The symbolic state model of Gillian-Rust is tailored to address the challenges of hybrid Rust verification. It is, to this day, the most advanced state model ever built for Gillian, leveraging the unique parametricity of the framework to encode and automate complex reasoning usually performed in Iris.

This state model can be constructed as follows, where \times is the product operator on state models introduced in Chapter 8:

$$\bar{\mathbb{S}}_{\text{Rust}} = \text{GuardedPred}(\text{RustHeap} \times \text{Lft} \times \text{Obs} \times \text{Pcy})$$

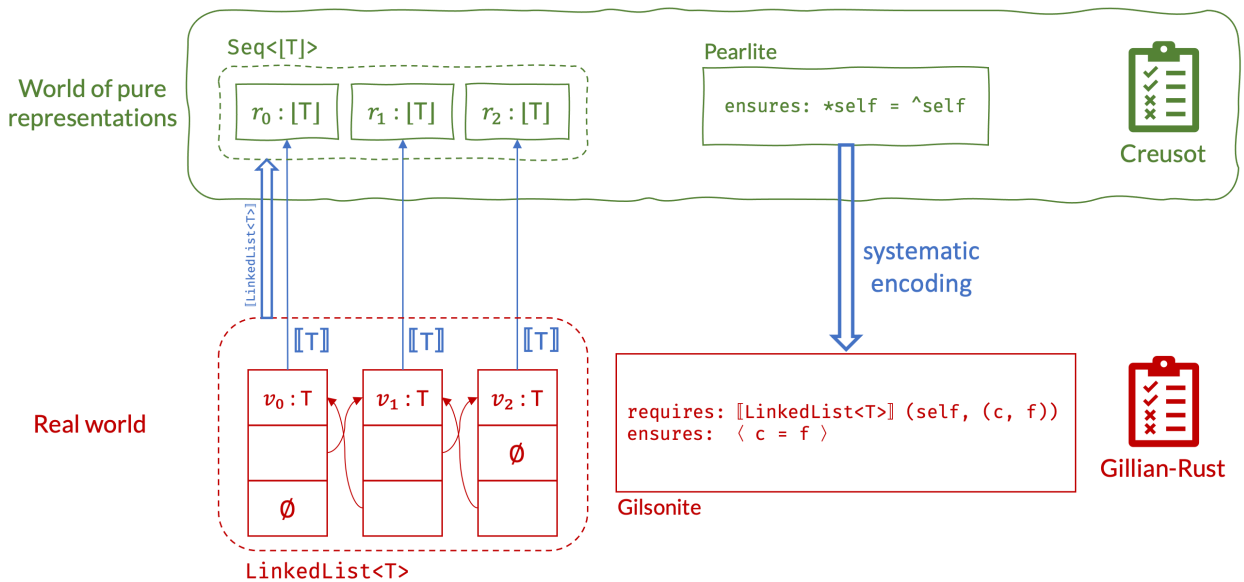
Outline In Chapter 17, we give an overview of our hybrid approach. In Chapter 18, we describe `RustHeap`, a symbolic heap model for Rust capable of both layout-independent reasoning about Rust memory and performing pointer arithmetic and bit-level operations. In Chapter 19, we describe the lifetime context `Lft` and guarded predicate transformer `GuardedPred`, demonstrating how to leverage Gillian’s unique extensibility to encode concepts from the *lifetime logic* of `RustBelt` and obtain a substantial degree of automation, enabling Gillian-Rust to reason about type safety of mutable references. In Chapter 20, we present the observation context `Obs` and the prophecy context `Pcy`, a way of embedding within Gillian-Rust the ability to reason about parametric prophecies as proposed by `RustHornBelt`. In Chapter 21, we describe end-to-end verification of a safe-unsafe Rust program, elaborating on the interpretation of hybrid specifications as both `Creusot` and `Gillian-Rust` specifications and the details of `Gillian-Rust` automation. In Chapter 22, we evaluate Gillian-Rust by verifying type safety and functional correctness of several Rust standard library types and their safe clients, demonstrating the efficiency and scalability of our hybrid approach. Finally, we discuss the current limitations of Gillian-Rust in detail and provide a pathway towards overcoming these limitations (Chapter 23), and place Gillian-Rust in the context of overall related work (Chapter 24).

Chapter 17

The Gillian-Rust infrastructure

We present our hybrid approach in more detail, show how Gillian-Rust can be used for proving a Creusot specification, and describe the structure of Gillian-Rust as an instantiation of Gillian.

17.1 A hybrid approach: Creusot + Gillian-Rust



The unmatched simplicity of Creusot specifications and the extent of its proof automation come from the fact that its proofs do not manipulate the real representation of objects, but an abstract, pure representation instead. Take, for example, doubly-linked lists, which are infamously difficult to implement in Rust, and equally infamous for being difficult to specify without separation logic. Creusot, when performing the proof for a piece of code which uses the Rust `LinkedList` module, does not see its intricate representation but instead models the linked list as a pure sequence of values. This approach, made possible by the guarantees provided by the safe fragment of Rust, sacrifices the ability to reason about the implementation of the `LinkedList` itself, in exchange for an efficient encoding into SMT, a high degree of automation, and no need for separation logic.

RustHornBelt provides a foundational argument for the validity of this approach by connecting the real world to Creusot's world of pure representations. This is done by providing *ownership predicates*¹ for each type T , which describe the safety invariant that the values of this

Figure 17.1: A high-level illustration of the differences and connections between the world of pure representations, observed by Creusot, and the world of real representations, observed by RustHornBelt and Gillian-Rust.

¹ Ownership predicates for Rust types were introduced in RustBelt, but did not connect types to a pure representation

type must uphold and connect it to the associated pure representation of type $[T]$ (cf. Figure 17.1 (left)).

To verify all Rust code, including that which contains unsafe blocks, we propose a hybrid approach where Creusot verifies all of the proof obligations within its reach and delegates the verification of unsafe code to another tool. Such a tool, however, does not yet exist, and building it is a great application for Gillian. We therefore propose a proof-of-concept called Gillian-Rust, which has the ability to perform SL reasoning required for the verification of unsafe code, breaking the abstraction and manipulating ownership predicates directly.

A keystone to this approach is the ability to systematically encode Creusot specification, written in an assertion language called *Pearlite*, into the assertion language of Gillian-Rust, which we dub *Gilsonite*, as represented in Figure 17.1 (right), and detailed in §21.2.

17.2 Example usage of Gillian-Rust

Doubly-linked lists are notoriously difficult to implement in Rust: the presence of back edges violates the strict ownership discipline imposed by the use of mutable references. Instead, one must use mutable *raw pointers* (cf. Figure 17.2, left), making doubly-linked lists a canonical example of a data structure requiring an unsafe implementation. On top, the non-trivial invariant that the list can be integrally traversed in both directions without cycles must be upheld, as otherwise the function in charge of disposing the list would visit a node twice, thereby performing a double-free.

We show the process of using Gillian-Rust to prove a Pearlite specification for the `push_front` function of the Rust standard library, which in-place adds an element to the front of a `LinkedList`.

```

struct Node<T> {
  elem: T,
  next: Option<NonNull<Node<T>>>,
  prev: Option<NonNull<Node<T>>>,
}
struct LinkedList<T> {
  head: Option<NonNull<Node<T>>>,
  tail: Option<NonNull<Node<T>>>,
  len: usize,
}

impl<T : Ownable> Ownable for LinkedList<T> {
  type ReprTy = Seq<T::ReprTy>;
  #[predicate]
  fn own(self, repr: Self::ReprTy) → Gilsonite {
    gilsonite!(
      dllSeg(self.head, None, self.tail, None, repr) *
      (self.len == repr.len())
    )
  }
}

```

Figure 17.2: `LinkedList` structure definition and its ownership predicate. The `dllSeg` predicate is given in §18.3.

Implementing Ownable The first step is to connect the real Rust structure to its pure representation used by Creusot. To do so, users must implement the `Ownable` trait² and define: the type of its representation, `ReprTy` (denoted by $[\cdot]$ in mathematics); and the ownership predicate, `fn own`, which takes two parameters, the structure itself (`self`) and the representation.

In the case of `LinkedList<T>`, its representation type is a sequence, of which each element has type $\tau::\text{ReprTy}$. Note that, in order for this type to be properly defined, τ itself must implement `Ownable`, a constraint specified using a trait bound (the `' : Ownable'` part in `<T : Ownable>`).

² A trait is, akin to a Haskell type-class, a form of interface describing a list of items that can be implemented for a type.

```

#[show_safety] // Expands to:
// #[specification(
//   requires { self.own(_) * e.own(_) }
//   ensures { result.own(_) }
// )]
fn push_front(&mut self, e : T) {
    // Implementation
}

#[requires(self@.len() < usize::MAX)]
#[ensures((^self)@ == (*self).prepend(e))]
fn push_front(&mut self, e : T) {
    // Implementation
    mutref_auto_resolve!(self)
}

```

Figure 17.3: The type safety (left) and Pearlite (right) specifications for `push_front`

Type safety Once the ownership predicate is defined, we can already verify type safety of a function by simply adding the `#[show_safety]` attribute on top. This attribute expands to a Gilsonite specification which requires all input parameters to be owned when entering the function, and ensures that the resulting value be owned when the function returns. The type safety specification (cf. Figure 17.3, left) corresponds to that proposed in RustBelt, which also requires a lifetime token in the pre- and post-condition. This token is added automatically to the pre- and post-condition by the Gillian-Rust compiler, and Gillian-Rust is able to prove this specification fully automatically.

Functional correctness Next, our goal is to specify that the function actually performs the desired operation. This can be elegantly done in Pearlite (cf. Figure 17.3, right), by describing the update performed on the sequence which represents the `LinkedList`: *when the mutable reference expires*, the representation will be the input sequence with the element prepended.

Pearlite, inspired by RustHorn³, uses prophecy variables and the final value operator `^` in order to specify such a property. RustHornBelt provides the theory underpinning this, and we provide a high-level description of the corresponding proof techniques as well as their implementations and automation strategies in Gillian-Rust in Chapter 20. Using our systematic encoding, we can translate this Pearlite specification into a Gilsonite specification: this particular translation is given in §21.2, together with further explanations. Finally, after adding a single line which triggers a semi-automatic tactic during verification, Gillian-Rust is able to perform the proof for this specification.

³ Matsushita et al., “RustHorn: CHC-based Verification for Rust Programs”, 2021 [MTK21]

Chapter 18

Reasoning about the real Rust heap

While RustBelt provides the theoretical framework on which our work is founded, it intentionally avoids the challenge of reasoning about the real Rust heap by instead defining an operational semantics and type system for λ_{Rust} , a small lambda-calculus with a simplified memory model. For example, in λ_{Rust} , all integers are unbounded and take one cell in memory, ignoring the 12 different primitive machine integer types offered by Rust, which take between 1 and 16 bytes in memory.

The literature, from previous work on other systems programming languages such as C, already has ways of reasoning about machine integers, but Rust also comes with challenges currently undealt with. In particular, while C comes with a specific algorithm that describes and decides on the layout of structures in memory and allows for arbitrary pointer arithmetic to access structure fields, the Rust compiler provides fewer guarantees, reserving the right to re-order fields and adjust padding between them. Rust also has features that do not exist in C, such as enums (tagged unions), which offer even fewer guarantees, as Rust may manipulate fields arbitrarily to reduce the overall size of the structure without affecting expressivity, in a process called niche optimization.

Until now, Rust verification tools have been working around these issues. For example, Prusti encodes structures using the object-oriented memory model of Viper, allowing efficient field access but preventing reasoning about pointer arithmetic, and Kani compiles Rust to a C-like representation by choosing a specific layout for each structure, dropping the guarantee that a verified program would be correct had the compiler made different layout choices authorised by the language¹.

In this chapter, we describe the solution provided by Gillian-Rust, which does a best-effort attempt at *maintaining abstraction*—hence preserving field-access efficiency—while still allowing for pointer arithmetic by leveraging Gillian’s ability to implement custom heap models. We show how to encode addresses so that they are layout-independent; describe a novel state model, `RustHeap`², capturing the representation of objects in the heap that allows for efficient automated reasoning; and present the points-to core predicate, which allows for specifying the Rust heap in Gillian-Rust.

18.1 Layout-independent memory addresses

The representation of addresses in Rust constitutes a challenge on its own. Ideally, one would prefer to reuse the one used by Gillian-C, inspired by CompCert³ and also used in RustBelt, where an address is a pair $(l, o) \in Loc \times \mathbb{N}$ of an object location (identifying a unique

¹ Group, *Structs and Tuples - Memory Layout - Unsafe Code Guidelines*, 2023 [Gro23]

² In fact, we present the state model of `RustBlock`, which can be used to construct the heap as

`RustHeap = PMap(Loc, RustBlock)`

using the partial map construction from [Chapter 8](#).

³ Leroy et al., “The CompCert Memory Model, Version 2”, 2012 [Ler+12]

allocation) and an offset. However, because of the above-mentioned challenges, this representation is insufficient, as structure field access may correspond to different offsets depending on the compiler-chosen layout.

To overcome this issue, Gillian-Rust modifies the encoding of offsets by using sequences of *projection elements* forming a *projection* (we reuse the compiler’s internal terminology) instead of a natural number. Specifically, a projection element represents either: an offset of e times the size of the type T , where e is a symbolic integer, denoted by $+^T e$; or the offset of the i -th field of a structure (relative w.r.t. the beginning of the structure), denoted by $.^T i$; or the relative offset of the i -th field of the j -th variant of an enum, denoted by $.^T.j i$.

$$\begin{aligned} l &\in Loc & e &\in \overline{\mathbb{Z}} & i, j &\in \mathbb{N} \\ \text{pr} \in \text{ProjE} & ::= & +^T e & | & .^T i & | & .^T.j i \\ a \in \text{Addr} & ::= & (l, \vec{\text{pr}}) \end{aligned}$$

This representation makes the interpretation of a symbolic address effectively parametric on the layout chosen by the compiler: given a layout which provides a concrete offset for each field of a structure or an enum, and a size to every type, each projection element can be interpreted as a symbolic natural number, and each projection as the sum of the interpretations of its elements.

18.2 Objects in the Rust symbolic heap

Our goal is to represent objects in the symbolic heap in a way that would enable us to efficiently resolve field accesses and perform only layout-independent pointer arithmetic. To this end, we propose a hybrid tree representation featuring two kinds of nodes: *structural nodes*, which represent a region of memory for which we know the structure but not necessarily the layout (such as Rust structures or enums), and on which no pointer arithmetic is allowed; and *laid-out nodes*, which are known to have an array-like layout and admit certain pointer arithmetic. For clarity of presentation, we provide a high-level description of heap objects, focussing on the main functionalities and insights.

Structural nodes Structural nodes are annotated with their type, and may be one of the following:

- a single node containing either: the special value `Uninit`, representing uninitialised memory, which is illegal to read; the special value `Missing`, representing memory that has been framed off; or a symbolic value;
- a tree representing a structure, consisting of: a root (internal) node, which holds no information; and children nodes, which represent its fields; or
- a tree representing an enum with a concrete discriminant⁴, containing: an internal node holding said discriminant; and children nodes representing the fields of the corresponding enum variant.

⁴ A symbolic enum (i.e., an enum with a symbolic discriminant) would be represented as a single node with a symbolic value.

The types annotating the nodes must be sized (that is, must have a size known at compile-time, chosen by the compiler⁵), thereby providing an interpretation for each node. The *load* and *store* primitive operations are provided in the interface of the symbolic heap and must ensure that the validity invariants⁶ of values written in memory are maintained (for example, that booleans are represented only by the bit-patterns `0b0` and `0b1`). They are also responsible for enforcing other important aspects of the Rust semantics, such as that loading a value from memory in the context of a *move* will deinitialise that memory.

In Figure 18.1, we give an example of a structure *S* and its structural node representation, comprising an internal node annotated with type *S* and two single-node children with respective values and types (\bar{x} , `u32`) and (\bar{y} , `u64`). The type of the left child, for example, indicates that it represents a region of 4 bytes in memory, and that the symbolic value \bar{x} is an integer in the range $[0, 2^{32})$. We also show two potential interpretations of a structural node for *S*, depending on the compiler-chosen field ordering: the top interpretation is obtained when the ordering is from-largest-to-smallest, and the bottom when the ordering is from-smallest-to-largest, inserting the appropriate padding when needed. This structural node in particular can only be navigated using `.s0` or `.s1`.

Laid-out nodes While structural nodes facilitate efficient resolution for a large majority of memory accesses, they are not a novel concept. The novelty of our approach lies in combining structural nodes with laid-out nodes, inspired by Gillian-C (Chapter 12), which describe a region of memory with an array-like layout in the sense that it allows for basic indexing pointer arithmetic. For example, Rust arrays, which are at the core of the Rust vector type, are always laid out contiguously such that the *n*-th element of an array of type $[T; N]$ starts at offset $n * \text{size_of}::\langle T \rangle$ w.r.t the beginning of the array, regardless of the layout of the element itself. Similarly, any integer type, say `u32`, can be seen as array-like as it is always represented by contiguous bytes in memory.

⁵ In contrast to unsized types, such as the slice type $[T]$, for which the size is only known at run-time.

⁶ Jung, *Two Kinds of Invariants: Safety and Validity*, 2018 [Jun18]

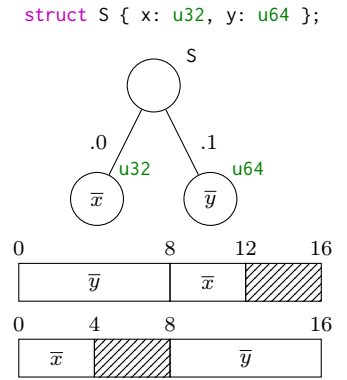


Figure 18.1: A structure *S*, its representation as a structural node, and two of its potential interpretations

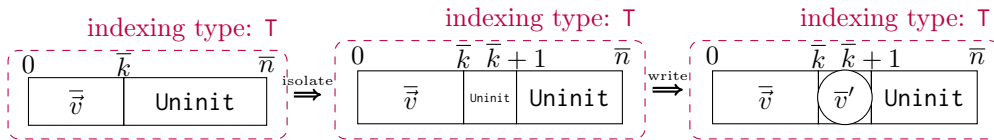


Figure 18.2: Update of a laid-out node corresponding to $n * \text{size_of}::\langle T \rangle$ bytes.

A laid-out node is a pair composed of a sized type (called indexing type) and a list of structural nodes each annotated with the range it occupies in multiples of the size of the indexing type. This indexing type is where Gillian-Rust diverges from Gillian-C. In C, all sizes are known, and can be expressed as a symbolic number. In Rust, however, the size of a type is not always known before compilation, and the indexing type allows us to express the size of the elements in the laid-out node in a way that is independent of the layout chosen by the compiler.

For example, Figure 18.2 (left) shows a laid-out node with indexing type τ and two structural nodes, the first carrying a symbolic list value

\bar{v} occupying the range $[0, \bar{k})$ (note that the \bar{k} is symbolic), and the second capturing uninitialised memory occupying the range $[\bar{k}, \bar{n})$, with $\bar{k} < \bar{n}$.

When resolving pointer arithmetic, Gillian-Rust is able to automatically destruct and reassemble laid-out nodes, allowing for arbitrary range access and manipulation. For example, Figure 18.2 (middle) and (right) show the process of writing a single value of type τ at the \bar{k} -th offset; this corresponds to pushing at the end of a vector with sufficient capacity. Gillian-Rust achieves this by first isolating the region in which the newly added value is going to be written (Figure 18.2, middle), splitting the second node into two, and then overwriting the appropriate region (in this case, from \bar{k} to $\bar{k} + 1$) with a structural node corresponding to the added value (Figure 18.2, right), simplified for this example to be a single node. Importantly, the indexing type does not have to match the type of each individual sub-node. For example, explicit calls to the Rust allocator API will always result in a laid-out node with indexing type `u8` (i.e., single bytes), but can be populated with values of arbitrary other type τ .

18.3 Specifying the Rust heap: the typed points-to core predicate

We focus on the most important core predicate used to specify heap shape with Gilsonite: the typed points-to predicate, $a \mapsto_{\tau} v$, which is satisfied by a heap fragment starting from address a and containing `size_of::()` bytes, which together form a valid representation of the value v . The remaining core predicates are only variations on this theme and are used for specifying, for example, slices or potentially uninitialised memory.

Unlike Gillian-C, the types used to annotate the points-to core predicate can be arbitrarily complex. This is possible because we compile GIL code from the mid-level intermediate representation of Rust (MIR), which still has full type information. This is a significant advantage, as it avoids performing several consumption and production steps for a single structure, hence enhancing performance. It also allows us to write a points-to predicate when the type of the value is a type parameter, enabling support for polymorphism and generic types.

The separation logic induced by the core predicates can be used by the verification engineer to specify a variety of predicates, pre-conditions and post-conditions. For example, the typed points-to predicate is enough to specify the ownership predicate for a doubly-linked list (`LinkedList`):

$$\begin{aligned} \llbracket \text{LinkedList} \langle T \rangle \rrbracket(l, r) &\triangleq \text{dllSeg} \langle T \rangle(l.\text{head}, \text{None}, l.\text{tail}, \text{None}, r) * l.\text{len} = |r| \\ \text{dllSeg} \langle T \rangle(h, n, t, p, r) &\triangleq (h = n * t = p * r = []) \vee \\ &\quad (\exists h', v, z, r'. h = \text{Some}(h') * h' \mapsto_{\text{Node} \langle T \rangle} \{v, z, p\} * \llbracket T \rrbracket(v, r_v) * \\ &\quad \text{dllSeg} \langle T \rangle(z, n, t, h, r') * r = r_v :: r') \end{aligned}$$

The doubly-linked-list-segment predicate, `dllSeg`, is well-known

from SL literature. It receives four optional pointers, h , n , t , and p , and a sequence of values r . The pointers h and p represent, respectively, the head and the tail pointer to the doubly-linked list, while n corresponds to the `next` pointer of the tail node and p to the `prev` pointer of the head node; both p and n equal `None` when the list segment represents the entire linked list. The sequence r contains the values of the nodes in the list, ordered left-to-right. This predicate can be reused in the Rust context with only one adaptation: the value of each node must be owned by the list (captured by the $\llbracket T \rrbracket(v, r_v)$ ownership predicate), effectively making the predicate parametric on the type of the values that the list holds. The predicate can be defined using Gilsonite as follows; note that the \rightarrow arrows need not be annotated with the type, as type inference is performed by the Rust compiler:

```
#[predicate]
fn dll_seg<T: Ownable>(h: Option<NonNull<Node<T>>>, n: Option<NonNull<Node<T>>>,
                      t: Option<NonNull<Node<T>>>, p: Option<NonNull<Node<T>>>, r: Seq<T::ReprTy>) {
    gilsonite!(h == n * t == p * r == Seq::empty());
    gilsonite!(exists hp, z, v, rv. h == Some(hp) * hp  $\rightarrow$  Node { next: z, prev: p, element: v } *
                v.own(rv) * dll_seg(z, n, t, h, r.prepend(rv)))
}
```


Chapter 19

Automating reasoning about mutable borrows

The handling of mutable borrows is one of the main challenges when trying to specify and verify Rust programs in fully-safe and unsafe contexts alike. While RustBelt¹ provides a theoretical framework for reasoning about mutable borrows within Iris and proves its correctness in Coq, this comes at the cost of the reasoning itself being manual and slow. In this section, we show how to leverage the unique flexibility of Gillian to automate reasoning about lifetimes and basic operations on mutable borrows.

¹ Jung et al., “RustBelt: securing the foundations of the Rust programming language”, 2017 [Jun+17]

19.1 Modelling lifetimes: core predicates

In Rust, a lifetime is a type-level variable representing a period of time during which a reference is valid. It is the responsibility of the borrow checker of the compiler to compute sound lifetimes for all references so that the ownership discipline of Rust is maintained.

In RustBelt, lifetimes are encoded as *tokens* in its separation logic: the token $[\kappa]_q$, with $0 < q \leq 1$, represents an alive lifetime κ , while $[\dagger\kappa]$ denotes that the lifetime κ has expired. RustBelt also provides rules to reason about lifetime tokens, some of which are included below for illustrative purposes: e.g., **LFTL-NOT-OWN-END** states that a lifetime cannot be alive and expired at the same time; **LFTL-END-PERSIST** states that an expired lifetime token is persistent (i.e. it can be duplicated); while **LFTL-TOK-FRACT** states that alive lifetime tokens may be split into fractions (for $0 < q, q'$).

LFTL-NOT-OWN-END	LFTL-END-PERSIST	LFTL-TOK-FRACT
$[\kappa]_q * [\dagger\kappa] \Rightarrow \text{False}$	$\text{persistent}([\dagger\kappa])$	$[\kappa]_{q+q'} \Leftrightarrow [\kappa]_q * [\kappa]_{q'}$

For the reader that is familiar with Iris, the above rules are reminiscent of the *one-shot* resource algebra. In Gillian, using variations of the constructions from [Chapter 8](#), we can encode the lifetime context as

$$\text{Lft} = \text{PMap}(\text{Lft}, \text{FreeAg}(\text{Frac}(\text{Unit})))$$

where each state is a partial finite map (constructed using the **PMap** constructor) from lifetimes *Lft* to fractions. The **Unit** state model passed to the **Frac** transformer indicates that the fractional token carries no additional value. The alive lifetime token $[\kappa]_q$ then becomes syntax sugar for the core predicate $\langle \text{Frac} \rangle(\kappa, q; ())$. In addition, each lifetime token can be *freed*, and the freed resources (expired token $[\dagger\cdot]$) form an agreement state model, as captured by the **FreeAg** constructor.

Fun parenthesis: As noted in [Chapter 8](#), duplicable expired tokens are incompatible with under-approximate (UX) reasoning. While this is

not a problem for Gillian-Rust, which does not make use of the support for UX in Gillian, it suggests that reasoning about type *unsoundness* instead of type safety would require novel foundations beyond RustBelt.

A lifetime context ξ is then simply a partial finite map from lifetimes to either a symbolic value ranging over real numbers in the interval $(0, 1]$, corresponding to the currently owned fraction of the lifetime token, or \dagger , capturing that the lifetime has expired.

In Gillian-Rust, both kinds of tokens become core predicates, and we demonstrate how the three RustBelt rules shown above are automated by providing an excerpt of the rules governing their consumers and producers below.² While simple, these rules are illustrative of the relationship between custom consumers/producers and automation. For example, the rule **LFT-PRODUCE-ALIVE-ADD** adds a fraction q of an alive token when a fraction q' is already owned, automating the right-to-left implication of **LFTL-TOK-FRACT**. On the other hand, **LFT-PRODUCE-OWN-END** vanishes (i.e. assumes **False**) when producing an alive token in a context where the lifetime has expired, automating **LFTL-NOT-OWN-END**. Similarly, in the consumer/producer paradigm, a core predicate is made persistent when its producer is idempotent and its consumer does not modify memory. Hence, together, rules **LFT-CONSUME-EXP** and **LFT-PRODUCE-EXP-DUP** automate **LFTL-END-PERSIST**.

$$\xi \in Lctx = Lft \xrightarrow{fin} \mathbb{R}_{(0,1]}^\dagger$$

² In these rules, to avoid clutter: the judgement uses only the lifetime context instead of the entire symbolic state; and the return value is elided because both actions return unit.

$$\begin{array}{c}
\text{LFT-PRODUCE-ALIVE-ADD} \\
\frac{\xi(\kappa') = q' \quad \pi = (\kappa = \kappa' \wedge 0 < q \wedge q + q' \leq 1) \quad \xi' = \xi[\kappa \leftarrow q + q']}{\xi.\text{prod}_{[.]}.(\kappa, q) \rightsquigarrow \langle \xi' \mid \pi \rangle} \\
\\
\begin{array}{cc}
\text{LFT-PRODUCE-OWN-END} & \text{LFT-CONSUME-EXP} \\
\frac{\xi(\kappa) = \dagger}{\xi.\text{prod}_{[.]}.(\kappa, q) \rightsquigarrow \mathbf{vanishes}} & \frac{\xi(\kappa') = \dagger \quad \pi = (\kappa = \kappa')}{\xi.\text{cons}_{[\dagger.]}.(\kappa) \rightarrow \langle 0k : ([], \xi) \mid \pi \rangle} \\
\\
\text{LFT-PRODUCE-EXP-DUP} \\
\frac{\xi(\kappa') = \dagger \quad \pi = (\kappa = \kappa')}{\xi.\text{prod}_{[\dagger.]}.(\kappa) \rightsquigarrow \langle \xi \mid \pi \rangle}
\end{array}
\end{array}$$

19.2 Modelling full borrows: guarded predicates

In Rust, a mutable reference of a value of type T during lifetime κ , denoted by $\&_{\text{mut}}^\kappa T$, corresponds to *temporary* ownership of the reference and the value it points to. To model such a behaviour, RustBelt introduced full borrows, denoted by $\&^\kappa P$, which are higher-order predicates denoting that the resource described by assertion P is borrowed during lifetime κ . In RustBelt, where ownership predicates do not expose a pure representation, the ownership predicate of a mutable reference p and the key rules for manipulating mutable borrows are as follows:

$$\llbracket \&_{\text{mut}}^\kappa T \rrbracket(p) \triangleq \&^\kappa (\exists v. p \mapsto v * \llbracket T \rrbracket(v))$$

$$\begin{array}{c}
\text{LFTL-BORROW-ACC} \\
\&^\kappa P * [\kappa] \equiv * \triangleright P * (\triangleright P \equiv * \&^\kappa P * [\kappa])
\end{array}$$

In particular, **LFTL-BORROW-ACC** states that one may *open a borrow* by temporarily giving up the corresponding lifetime token, and may later *close that borrow* after having reformed the invariant, at which point the token is recovered. Crucially, having to reform the invariant inside a borrow is what ensures that a callee function which is given a borrow may not cause undefined behaviour in the future, and every borrow must eventually be closed, as the lifetime token is required at the time it expires. In Gillian-Rust, the view shift operator present in the **LFTL-BORROW-ACC** rule is realised via guarded predicate unfolding, introduced shortly, whereas the later modality, \triangleright , is omitted; in §23.2, we provide a justification for the soundness of this approach.

Full borrows raise two main challenges for a semi-automated tool such as Gillian: **1)** it needs to reason about higher-order predicates; and **2)** it needs to automatically understand when to open and close borrows in common proof patterns. We now present the two key insights behind the encoding and automation of reasoning about full borrows in Gillian-Rust.

Compiling away higher-orderness While program proofs do make use of higher-order rules such as **LFTL-BORROW-ACC**, they only use them with a specific, finite set of instantiations. For example, when proving `pop_front_node`, one only needs to manipulate the particular borrow predicate corresponding to the ownership predicate $\llbracket \&_{\text{mut}}^{\kappa} \text{LinkedList} \langle T \rangle \rrbracket$. When using the Gilsonite API, a user may instantiate the full borrow assertion using the `#[borrow]` attribute. For instance, the ownership predicate for mutable references is defined as follows in the Gilsonite library:

```
impl<T> Ownable for &mut T {
  #[borrow]
  fn own(self) → Gilsonite {
    gilsonite!(exists v. (self → v) * v.own())
  }
}
```

obtaining an ownership predicate for mutable references of type τ . Note that such predicates can be defined parametrically, using a generic type; when required for a more specific type, such as `LinkedList<T>`, they will be instantiated at compilation time.

Finally, ownership predicates for type parameters are compiled to abstract predicates, that is, predicates that cannot be unfolded, a well-known trick in the world of semi-automated tools. This ensures that if a specification has been proven using a type parameter T , then this type parameter can be instantiated with any other type to obtain a new trusted specification, with the instantiation happening at the call site that requires it.

Leveraging known automations for borrow access The key insight to automating borrow access is the understanding that borrows behave similarly to predicates encoded using the **Pred** symbolic state model transformer presented in Chapter 8. Remember that, using the transformer $\text{Pred}(\mathbb{S}, \mathbf{P})$, the symbolic state model \mathbb{S} is enhanced with support for the user-defined predicates defined in \mathbf{P} . The state then maintains

a list of *folded* predicates, of the form $(\rho, \vec{v}) \in (\mathbf{P.names} \times \text{List}(\overline{Val}))$, where each folded predicate consists of a name ρ and parameters \vec{v} .

Each of these tools also comes with two ghost commands that allow users to manipulate folded predicates: **unfold** and **fold**. In particular, **unfold** removes a predicate stored in its folded form from the state and produces its definition in its place, whereas **fold** is its dual, consuming the predicate’s definition from the state and adding its folded form to the state.

One may notice the similarity between the borrow access rule and the folding and unfolding of predicates: when closed, both borrows and folded predicates act as abstract tokens that can be exchanged for the resource they contain. The only distinction is the “cost” of unfolding: none for predicates, and a lifetime token for borrows.

A guarded predicate context $\bar{\gamma} \in \text{List}(Str \times \textcolor{red}{Lft} \times \text{List}(\overline{Val}))$ is a list of predicates which are annotated with a lifetime such that its token is the cost for their opening. It exposes two actions: **gunfold**/**gfold**, which respectively behave like **unfold**/**fold** apart from the fact that they consume/produce that guarding lifetime token, and produce/consume an additional opaque *closing token*, denoted by $C_\rho(\kappa, q, \vec{x})$, which embodies the closing update $(P \Rightarrow * \&^\kappa P * [\kappa]_q)$.

UNFOLD-GUARDED

$$\frac{\begin{array}{l} \mathbf{P}[\rho] = (\mathbf{1}, \vec{x}, Q) \quad \bar{\sigma}.\text{cons}_{[\cdot]}(\kappa, q) \rightarrow \langle \mathbf{Ok} : ([\cdot], \bar{\sigma}') \mid \pi \rangle \\ \bar{\sigma}' = (\bar{\mu}', \bar{\gamma}') \quad \rho(\kappa, \vec{v}) \in \bar{\gamma}' \quad \bar{\gamma}'' = \bar{\gamma}' \setminus \rho(\kappa, \vec{v}) \quad \bar{\sigma}'' = (\bar{\mu}', \bar{\gamma}'') \\ \textcolor{red}{Q'} = Q * C_\rho(\kappa, q, \vec{v}) \quad \theta = [\vec{x} \mapsto \vec{v}] \quad \bar{\sigma}''.\text{prod}_{Q'}(\theta) \rightsquigarrow \langle \bar{\sigma}''' \mid \pi' \rangle \end{array}}{p \vdash \bar{\sigma}.\text{gunfold}_\rho(\kappa, \vec{v}) \rightsquigarrow \langle \mathbf{Ok} : ([\cdot], \bar{\sigma}''') \mid \pi \wedge \pi' \rangle}$$

The UNFOLD-GUARDED rule describes successful execution of **gunfold**. For clarity, we decompose symbolic states into a pair $(\mu, \bar{\gamma})$, where μ represents the remaining components. In addition, we write **in purple** elements of the rule which are novel with respect to the more classic **unfold** rule. Finally, this command is performed in the context of a program p , where $p.\text{predDefs}$ maps predicates to their definitions.

This encoding of full borrows has one important advantage: Gillian comes with years of experience in automating separation logic proofs, including heuristics that are able to decide when to automatically unfold or fold predicates as required by the analysis. By encoding borrows in the above way, we can immediately leverage those heuristics and allow for automatic opening and closing of full borrows. In particular, proving the type safety of `LinkedList::pop_front` and `LinkedList::push_front` becomes completely automatic once the safety invariants of `LinkedList` has been properly specified as in §18.3.

19.3 Proving safety of borrow extraction

Unfortunately, opening and closing are not the only operations that one needs when working with full borrows. We identify several recurring patterns in unsafe Rust programs and provide ways of instantiating lemmas that allow us to analyse code that uses these patterns.

In particular, borrow extraction—the process of cutting a borrow up into a smaller borrow—is a common pattern in unsafe Rust programming, and every data-structure module of the standard library provides

at least one function that uses this pattern (e.g., `LinkedList::front_mut` or `Vec::get_mut`). In fact, borrow extraction is the most idiomatic way of modifying an element of a collection. Most often, implementing such a function is unsafe, as incorrect borrow extraction could break the safety guarantees of Rust. For example, consider the case in which the `LinkedList` library implementer creates a `first_node_mut` function, which returns a mutable reference not to the first element (`&mut T`), but to the first node (`&mut Node<T>`), which contains the first element as well as `next` and `prev` pointers. Then, **using only safe code**, a client function could modify the `next` pointer to point to the node itself, creating a cycle in the list. As explained in §17.2, this would certainly lead to an undefined behaviour, although not during the execution of `first_node_mut` itself.

On the other hand, returning a mutable reference to the first element (`&mut T`) is perfectly fine, the intuition being that one can *remove* the resource associated with the element and obtain a *remainder*. To that remainder we can then add any other element that satisfies the invariant of `T`, in order to recover a structure satisfying the `LinkedList` invariant. This principle is embodied by the `BORROW-EXTRACT` rule—which we have proven in Iris using RustBelt (it is a trivial corollary of the already existing rules)—where P is the invariant of the `LinkedList`, Q is the invariant of `T`, and $Q \multimap P$ is the remainder. In addition, the rule allows one to add a persistent context if it is required for performing the extraction. For example, in the case of the `LinkedList`, the extraction of the first node is only possible if it is not empty (i.e. if the head pointer is not `None`, which would be captured in that persistent context).

$$\frac{\text{BORROW-EXTRACT} \quad \text{persistent}(F) \quad F * P \Rightarrow Q * (Q \multimap P)}{F * [\kappa]_q * \&^\kappa P \equiv \&^\kappa Q * [\kappa]_q}$$

Using the Gilsonite API, users may instantiate the ghost command that performs the view shift in the conclusion of the `BORROW-EXTRACT` rule by specifying the borrow predicates $\&^\kappa P$ and $\&^\kappa Q$ as well as the persistent assertion F , as illustratively done in Figure 19.2.³ Gillian itself is unable to prove that `BORROW-EXTRACT` holds, or to manipulate borrows using such a rule. Instead, the Gillian-Rust compiler produces two lemmas: one corresponding to the conclusion of the rule, which is marked as trusted and left unproven, and one corresponding to the hypotheses of the rule, which needs to be proven. As we have proven that the rule itself holds in Iris, the meta-theory of Gillian-Rust therefore ensures that if we prove the second lemma, then the first lemma also has to hold.

```
#[extract_lemma( forall head, tail, len, p.
  assuming { head == Some(p) } // F
  from { list_ref_mut_frozen(list, head, tail, len) } // &^\kappa P
  extract { Ownable::own(&mut (*p.as_ptr()).element) } // &^\kappa Q
)]
fn extract_head<T: Ownable>(list: &mut LinkedList<T>); // Implicitly parametric on \kappa
```

To automatically prove this second kind of lemmas, we have extended Gillian with the ability to reason about magic wands, adapting the related work on Viper⁴, to Gillian’s parametric separation logic; the details of this extension are out of scope of this presentation.

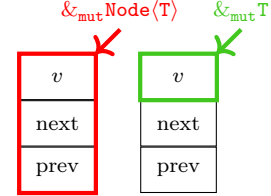


Figure 19.1: An invalid and a valid `LinkedList` mutable reference.

³ In Figure 19.2, `list_ref_mut_frozen` denotes a borrow predicate obtained from the ownership predicate of `&mut LinkedList` by freezing existentials corresponding to the `head`, `tail` and `len` fields of the structure. Freezing existential variables is a common strategy for extracting borrows, supported by the Gilsonite API. To avoid cluttering the main body of this presentation with more constructions, frozen borrows are detailed in Appendix F.

Figure 19.2: Example instantiation of a borrow extraction lemma

⁴ Dardinier et al., “Sound Automation of Magic Wands”, 2022 [Dar+22]

Chapter 20

Functional correctness and prophetic reasoning

While the ability to manipulate full borrows is enough to verify type safety of programs that make use of mutable references, it is not enough to prove functional correctness of these programs. In particular, the rule **LFTL-BORROW-ACC** presented previously enforces that the **same** invariant be used to close the full borrow, effectively losing the information that the value was updated.

Specifying functional correctness of programs manipulating mutable references is, in itself, a challenge, as it requires the ability to specify properties which shall only hold *in the future*, that is, at the time when the borrow expires. Thankfully, this challenge has been addressed by previous work: Prusti¹ introduced pledges and RustHorn² introduced prophecy variables, later used in Creusot. However, only the latter has been given a foundational formalisation in RustHornBelt³, an extension of RustBelt which describes how prophetic specifications interact with full borrows.

In this chapter, we briefly remind the reader of the workings of RustHornBelt, and show how its concepts are encoded in Gillian-Rust. To conclude our technical presentation, we show how Pearlite specifications can be compiled to Gilsonite, hence explaining how unsafe proof goals can be delegated by Creusot to Gillian-Rust.

¹ Astrauskas et al., “Leveraging rust types for modular specification and verification”, 2019 [Ast+19]

² Matsushita et al., “RustHorn: CHC-based Verification for Rust Programs”, 2021 [MTK21]

³ Matsushita et al., “RustHornBelt: a semantic foundation for functional verification of Rust programs with unsafe code”, 2022 [Mat+22]

20.1 Representations, parametric prophecies, and observations

In order to reason about functional correctness within the framework of RustBelt, RustHornBelt extends ownership predicates with an additional parameter corresponding to a pure mathematical *representation* of the value. Given a type T , the type of its representation is denoted by $[T]$. For example, a value of type `LinkedList<T>` is represented by a sequence of which each element is the representation of the element at the corresponding index in the list, i.e. $[LinkedList<T>] = Seq<[T]>$.

Mutable references, on the other hand, are represented as a pair of representations of the inner type (i.e., $[&mut \tau] = [T] \times [T]$), where the first element denotes the value to which the mutable reference currently points, and the second denotes the value it will have at the time it expires.

$$\begin{aligned} \llbracket \&_{mut}^\kappa T \rrbracket(p, r) &\triangleq \exists x \text{ s.t. } r^\star 2 = \uparrow x. VO_x(r^\star 1) * \\ &\&^\kappa(\exists v, a. p \mapsto v * \llbracket T \rrbracket(v, a) * PC_x(a)) \end{aligned}$$

RustHornBelt then proposes an ownership predicate for mutable references which exposes this representation, using a notion of *parametric prophecies*. A prophecy variable x is attached to the mutable reference, and the second element of the representation pair r is the future value

of this prophecy, denoted by $\uparrow x$.

$$\begin{array}{ll} \text{MUT-AGREE} & \text{MUT-UPDATE} \\ \text{VO}_x(a) * \text{PC}_x(a') \vdash a = a' & \text{VO}_x(a) * \text{PC}_x(a) \Rightarrow \text{VO}_x(a') * \text{PC}_x(a') \end{array}$$

In addition, there are two connected resources respectively called *value observer*, denoted by VO_x , and *prophecy controller*, denoted by PC_x , which together provide a solution to the problem of information loss when closing a full borrow. In particular, the observer maintains the last-observed current value and, when the borrow opens, the previously-lost value of the representation a is recovered through the **MUT-AGREE** rule. Before closing a borrow again, the verification engineer may use the **MUT-UPDATE** rule to update the value of the prophecy variable to match the new representation.

Lastly, RustHornBelt introduces *observations*, denoted by $\langle\!\langle\psi\rangle\!\rangle$, where ψ is a pure assertion containing information known about prophecy values. Observations act as a second layer of truth, preventing future information from leaking into the separation logic and creating paradoxes.

20.2 Key idea: parametric prophecies and symbolic execution

In order to encode prophecies into Iris, RustHornBelt wraps the entire execution into a reader monad. In simple terms, execution is performed within a context which preemptively captures an assignment for the future value of each existing prophecy variable (i.e., a *prophecy assignment* map $\text{PcyVar} \rightarrow \text{Val}$). A prophetic value is then an object which, given a prophecy assignment, yields a value, i.e. $\text{ProphAsn} \rightarrow \text{Val}$.

This definition is strikingly similar to the definition of *symbolic values* provided in [Chapter 6](#), with *prophecy* variables replacing *symbolic* variables.

Therefore, parametric prophecies appear to be closer to symbolic variables than they are to prophecy variables formalised by Jung et al.⁴. This intuition suggests that one may use the same process to reason about prophecy variables as for symbolic variables, and ideally fit them into the same framework. In symbolic execution, each state carries a *path condition* π , a pure formula which accumulates all currently-known constraints about the existing symbolic variables, while for prophecy variables, it is the observations that play this role of constraint accumulator. The core idea behind encoding prophecy variables follows from this remark: observations can simply take the shape of a secondary path condition, implemented as a component of the symbolic state, making calls to the solver when required.

To this end, we introduce a new symbolic state model, where states consist of only one symbolic expression, called *observation context* and denoted by $\phi \in \text{Obs}$. The observation context may depend on both prophecy variables and symbolic variables. Below, we present three of the rules that apply to observations in RustHornBelt, and two rules that describe the behaviour of the corresponding Gillian-Rust consumer and producer for the successful cases.

⁴ Jung et al., “The future is ours: prophecy variables in separation logic”, 2020 [[Jun+20](#)]

$$\begin{array}{c}
 \text{OBS-MERGE} \\
 \langle \psi \rangle * \langle \psi' \rangle \vdash \langle \psi \wedge \psi' \rangle \\
 \\
 \text{PROPH-SAT} \\
 \langle \psi \rangle \Rightarrow \exists \varepsilon. \varepsilon(\psi) \\
 \\
 \text{PROPH-TRUE} \\
 \frac{\forall \varepsilon. \varepsilon(\psi)}{\langle \psi \rangle} \\
 \\
 \text{OBSERVATION-PRODUCE} \\
 \frac{\pi \wedge \phi \wedge \phi' \text{ SAT}}{\langle \phi, \pi \rangle.\text{prod}_{\langle \cdot \rangle}(\phi') \rightsquigarrow \langle \phi \wedge \phi' \mid \pi \rangle} \\
 \\
 \text{OBSERVATION-CONSUME} \\
 \frac{(\pi \wedge \phi \Rightarrow \phi') \text{ VALID}}{\langle \phi, \pi \rangle.\text{cons}_{\langle \cdot \rangle}(\phi') \rightarrow \langle \text{Ok} : (\cdot), \phi \mid \pi \rangle}
 \end{array}$$

Exceptionally, these rules receive the *current* path condition as an argument, as it is necessary for the reasoning. Gillian-Rust depends on the alternative implementation of the symbolic execution monad presented in §6.6

OBS-MERGE indicates that our model of observations as a single symbolic expression is appropriate, and that framing on a new observation amounts to simply conjuncting it with the current observation. In addition, **PROPH-SAT** tells us that if an observation holds, then at least one prophecy assignment must satisfy it. Together, these rules instruct us how to implement the producer for observations: if the conjunction of the path condition, current observation, and new observation is satisfiable, then we can add the produced observation to our current one (cf. **OBSERVATION-PRODUCE**). Finally, **PROPH-TRUE** states that anything that is true independently of prophecy variables can be captured as an observation, that is, anything that is true outside of the prophetic world is also true within it. With our approach, this means that the path condition can be used seamlessly as part of our observations when needed, embodied in the **OBSERVATION-CONSUME** rule: when checking if an observation ϕ' holds, we check that it is entailed by the current path condition and observation.

20.3 Value observers and prophecy controllers

Value observers and prophecy controllers provide yet another opportunity to leverage the flexibility of Gillian and implement a custom state model. In particular, we entirely automate the **MUT-AGREE** rule by defining a *prophecy context* $\chi = \text{PcyVar} \rightarrow \text{Expr} \times \mathbb{B} \times \mathbb{B}$ as a map that associates each prophecy variable with its current value and two Booleans, which correspond to the ownership of the value observer and of the prophecy controller in the state.

Figure 20.1 provides rules for successfully producing a value observer into the state; the production rules for the prophecy controller are analogous and therefore elided. In particular, producing $\text{VO}_x(a)$ in a prophecy context which does not already contain any binding for the prophecy variable x will bind x to the triple $(a, \text{true}, \text{false})$, thereby encoding that the current value for the prophecy is a , that its value observer is in the context, but not its prophecy controller. On the other hand, if the controller with value a' already exists in the current state, that is, if the prophecy context already has the triple $(a', \text{false}, \text{true})$ bound to x , then the Boolean flag corresponding to the presence of the corresponding value observer is set to true without modifying the current value and we learn that $a = a'$, in the form of an additional constraint added to the path condition.

$$\begin{array}{c}
\text{VOBS-PRODUCE-WITHOUT-CONTROLLER} \\
\frac{x \notin \text{dom}(\chi) \quad \chi' = o[x \leftarrow (a, \text{true}, \text{false})]}{\chi.\text{prod}_{\text{VO}}(x, a) \rightsquigarrow \langle \chi' \mid \text{true} \rangle} \\
\\
\text{VOBS-PRODUCE-WITH-CONTROLLER} \\
\frac{\chi(x) = (a', \text{false}, \text{true}) \quad \chi' = \chi[x \leftarrow (a', \text{true}, \text{true})] \quad \pi = (a = a')}{\chi.\text{prod}_{\text{VO}}(x, a) \rightsquigarrow \langle \chi' \mid \pi \rangle}
\end{array}$$

However, this does not automate the **MUT-UPDATE** rule: after having modified the contents of a mutable reference $p: \&\text{mut } \tau$, one still needs to apply this rule before being able to close the mutable borrow. The current implementation of Gillian does not allow us to fully automate this process, but we are able to provide the **MUT-AUTO-UPDATE** lemma which can be used by the verification engineer by simply writing `p.prophecy_auto_update()`. This lemma updates the current value of the prophecy by automatically choosing the appropriate value that will allow the borrow to be closed again.

Finally, Gillian-Rust also provides a manual way of *resolving* mutable references, as described by **MUTREF-RESOLVE**, which, as proposed by RustHornBelt, allows us to obtain an observation of the equality between the current value of the prophecy and its future value at the time where the corresponding mutable reference expires.

Borrow extraction with prophecies When manipulating the ownership predicate of a mutable reference with prophecies in the style of RustHornBelt, the rule for extracting sub-borrows must be adapted to perform *partial resolution* of the prophecy. The corresponding rule is substantially more complex than **BORROW-EXTRACT**, but it yields the same level of automation and we have proven it correct in the Coq development of RustHornBelt. To avoid cluttering the main body of this presentation, we delay the presentation of this rule to [Appendix F](#).

Figure 20.1: Rules: producer of the value observer

$$\begin{array}{c}
\text{MUT-AUTO-UPDATE} \\
\frac{\llbracket \mathbf{T} \rrbracket(v, a') * \text{VO}_x(a) * \text{PC}_x(a) \Rightarrow \llbracket \mathbf{T} \rrbracket(v, a') * \text{VO}_x(a') * \text{PC}_x(a')}{\llbracket \mathbf{T} \rrbracket(v, a') * \text{VO}_x(a) * \text{PC}_x(a) \Rightarrow \llbracket \mathbf{T} \rrbracket(v, a') * \text{VO}_x(a') * \text{PC}_x(a')} \\
\\
\text{MUTREF-RESOLVE} \\
\llbracket \&_{\text{mut}}^{\kappa} \mathbf{T} \rrbracket(p, (a, a')) \equiv \star \llbracket a = a' \rrbracket
\end{array}$$

Chapter 21

Anatomy of a hybrid proof : Merge Sort

In this section, we present a detailed example of a hybrid proof, showing how we can use Creusot and Gillian-Rust to prove the correctness of a Merge Sort implementation that uses doubly-linked lists. We briefly cover the safe implementation and its verification in Creusot, and then explain how we interface with Gillian-Rust to prove correctness of associated unsafe operations.

21.1 Writing a hybrid proof

Following the approach outlined in [Chapter 17](#), we divide the work as follows: (1) Creusot is responsible for verifying the safe parts of the program (in this case, the Merge Sort algorithm itself), which normally constitute the great majority of the code; while (2) Gillian-Rust is responsible for verifying the unsafe parts (in this case, the doubly-linked list operations), which are normally more low-level and perform more complex but smaller operations such as manipulation of pointers or uninitialised memory. In [Figure 21.1](#), we present a fragment of our Merge Sort implementation. For space reasons, we elide the (standard) implementations of `merge_sort` and `merge`, focusing instead on the `split` function, which takes a mutable borrow to a linked list and splits it into two halves.

```
1  #[pearlite::ensures(sorted(^l)@ && l@.permutation_of(^l)@)]
2  pub fn merge_sort(l: &mut LinkedList<i32>) { // Standard implementation using split and merge }
3
4  #[pearlite::ensures(inp@.permutation_of(result.0@.concat(result.1@)))]
5  fn split(inp: &mut LinkedList<i32>) → (LinkedList<i32>, LinkedList<i32>) {
6      let old_inp = snapshot!(inp);
7      let mut (left, right, push_left) = (LinkedList::new(), LinkedList::new(), true);
8      let mut popped = snapshot! { Seq::EMPTY };
9      #[pearlite::invariant(popped.concat(inp@).ext_eq(old_inp@))]
10     #[pearlite::invariant(popped.permutation_of(left@.concat(right@)))]
11     while let Some(i) = inp.pop_front() {
12         popped = snapshot! { popped.push(i) };
13         snapshot!({perm_right::<i32>; perm_left::<i32>});
14         if push_left { left.push_front(i); } else { right.push_front(i); };
15         push_left = !push_left;
16     }
17     (left, right)
18 }
19
20 #[pearlite::requires(sorted(l@))]
21 #[pearlite::requires(sorted(r@))]
22 #[pearlite::ensures(sorted(result@) && result@.permutation_of(l@.concat(r@)))]
23 fn merge(l: &mut LinkedList<i32>, r: &mut LinkedList<i32>) → LinkedList<i32> { ... }
```

Figure 21.1: A fragment of our Merge Sort algorithm, implemented using doubly-linked lists

In Creusot, unsafe types such as `LinkedList<T>` are treated as *opaque types*, on which no operations can be performed. To reason about them,

Creusot axiomatises their representation function using a `ShallowModel` trait, and the Pearlite¹ specifications of their APIs are assumed as axioms. We can access this shallow model through its associated operator `@`. Using this model operation, we specify the postcondition of the `split` function as per line 4 of Figure 21.1, stating that the concatenation of the two resulting lists is a permutation of the input list. Operations on mutable borrows are specified using the *final* operator `^`, which accesses the prophecy of a mutable reference. In line 1, we specify that the initial value `((*1)@)` of the list is a permutation of its final value `((^1)@)`.

¹ Pearlite is a first-order logic, including the usual connectives for conjunction, disjunction, implication, and quantification, and also support for functions and predicate definitions.

```
pub struct LinkedList<T> { ... }

impl<T : Ownable> LinkedList<T> {
  #[hybrid::ensures(
    forall<x : _> result == Some(x) ==> Seq::singleton(x).concat((^self)@) == (*self)@)]
  #[hybrid::ensures(result == None ==> ^self == *self && self.len() == 0)]
  pub fn pop_front(&mut self) -> Option<T> { ... }

  #[hybrid::requires(self.len() < usize::MAX@)]
  #[hybrid::ensures(Seq::singleton(e).concat((*self)@) == (^self)@)]
  pub fn push_front(&mut self, e: T) { ... }

  #[hybrid::ensures((*self)@.push(e) == (^self)@)]
  pub fn push_back(&mut self, e: T) { ... }
}
```

Figure 21.2: The `LinkedList` library used by our Merge Sort algorithm

```
// Pearlite specification
#[pearlite::requires(self.len() < usize::MAX@)]
#[pearlite::ensures(Seq::singleton(e).concat((*self)@) == (^self)@)]
// Gilsonite specification
#[gilsonite::specification(forall self_repr, e_repr.
  requires { self.own(self_repr) * e.own(e_repr) $ self_repr.0.len() < Int::from(usize::MAX) $ }
  exists ret_repr.
  ensures { ret.own(ret_repr) * $Seq::singleton(e_repr).concat(self_repr.0) == self_repr.1$ }
)]
pub fn push_front(&mut self, e: T) { ... }
```

Figure 21.3: Pearlite and Gilsonite specifications of `push_front`

In Figure 21.2, we present the specification of the `LinkedList` library used by our Merge Sort. We use the `hybrid::requires` and `hybrid::ensures` attributes to specify, respectively, the pre- and post-conditions of the `pop_front`, `push_front`, and `push_back` functions. These attributes act as the bridge between Pearlite and Gilsonite, in that from them, using the compilation mechanism presented in §21.2, we are able to generate the Gilsonite specification expected by Gillian-Rust. For example, for `push_front`, we will end up with the specifications given in Figure 21.3.

Verification of the complete Merge Sort and accompanying Linked List implementation is performed by successively running `cargo creusot` and `cargo gillian` to generate the proof obligations for Creusot and Gillian-Rust, respectively, which are then discharged by running the appropriate back-ends: Why3 for Creusot and the Gillian-Rust back-end for Gillian-Rust.

21.2 Compilation of Creusot specifications

To compile Creusot specifications to Gilsonite, we first need to interpret Creusot's types in Gillian-Rust. Recall that we interpret Rust types using their *representations*, and that `LinkedList<T>` is interpreted via the `Ownable` trait in Gillian-Rust as `gillian_rust::Seq<T::ReprTy>`. In addition, we must interpret the *logical* types of Creusot, which is also done by defining appropriate instances of `Ownable`: in particular, the `creusot::Seq<T>` type of Creusot, just like `LinkedList<T>` or Rust, is interpreted as `gillian_rust::Seq<T::ReprTy>`. Like Creusot and RustHornBelt, we interpret mutable borrows as a pair of the representation of the value and a prophesied value, so that `&mut LinkedList<T>` is interpreted as $(\text{Seq}<T::\text{ReprTy}>, \text{Seq}<T::\text{ReprTy}>)$.

$$\begin{aligned}
 \{P\} \text{fn } f \langle \kappa \rangle (x_1 : T_1, \dots, x_n : T_n) \rightarrow T_r \{Q\} \\
 \implies \\
 \{ (\otimes_{i=1}^n \llbracket T_i \rrbracket (x_i, m_i)) * \langle P[x_i/m_i] \rangle * [\kappa]_q \} \\
 \text{fn } f \langle \kappa \rangle (x_1 : T_1, \dots, x_n : T_n) \rightarrow T_r \\
 \{ \exists m_r. \llbracket T_r \rrbracket (r, m_r) * \\
 \langle Q[x_i/m_i][r/m_r] \rangle * [\kappa]_q \}
 \end{aligned}$$

Specification interpretation is done by *elaboration*, the general schema of which is given above. We require ownership of every function argument, associating each with a representation value, and in the end, we own the result, again associated with a representation value. We then place the preconditions and postconditions into prophecy observations, substituting occurrences of Rust variables with their corresponding representation values. Following this process, we obtain the Gilsonite specification for `pop_front` as given in Figure 21.3.

21.3 Gillian-Rust in action: `LinkedList::push_front`

To complete our tour of this hybrid verification case study, we explain in detail how Gillian-Rust proves the Pearlite specification of `push_front` method of `LinkedList`, leveraging the various features of the tool presented in the previous sections. In Figure 21.4, we give the full implementation of `push_front`, together with the auxiliary `push_front_node` method. We provide a specification only for the former, as Gillian-Rust can simply symbolically execute the latter.

When execution starts, the state contains: **a)** the ownership predicate for a mutable reference to a `LinkedList` at reference `self`, with representation `self_repr`; **b)** the ownership predicate for the element `elt` of type `T`; **c)** an observation that the length of the representation of the linked list is less than `usize::MAX`; and **d)** a lifetime token corresponding to the lifetime of the mutable reference `self`.

First, in line 3, the function allocates a new owned pointer, `Box`, which contains a new node constructed from the element `elt`, with previous and next pointers set to `None`. This pointer is immediately passed to the auxiliary function `push_front_node`.

In line 11, the access to `self.head` requires ownership of the corre-

```

1  #[gilsonite::specification( ... )]
2  pub fn push_front(&mut self, elt: T) {
3      self.push_front_node(Box::new(Node::new(elt)));
4      // Automatically folds the dllSeg predicate twice, applies Mut-Update,
5      // closes the borrow, and applies MutRef-Resolve
6      mutref_auto_resolve!(self); // <- Single additional annotation required
7  }
8
9  fn push_front_node(&mut self, mut node: Box<Node<T>>) { unsafe {
10     // Mutable borrow is automatically opened when accessing self.head on the next line.
11     node.next = self.head; node.prev = None;
12     let node = Some(Box::leak(node).into());
13     // The dllSeg predicate is automatically unfolded to execute the next three lines.
14     match self.head {
15         None => self.tail = node, Some(head) => (*head.as_ptr()).prev = node,
16     }
17     self.head = node;
18     // Symbolic execution will branch on the next line, depending on self.len + 1 overflowing.
19     // The overflow branch will be discarded, by proving contradiction with the precondition.
20     self.len += 1;
21 } }

```

Figure 21.4: Implementation of push_front

sponding location in memory, which is currently hidden in the full borrow contained in the resource **a**). Thanks to the encoding of full borrows presented in §19.2, Gillian-Rust can automatically *open the borrow* by applying the UNFOLD-GUARDED rule, losing ownership of the lifetime token (resource **d**), and obtaining ownership of the value contained at address `self`, together with the ownership of the entire linked list, and the prophecy controller corresponding to its representation.

The following lines of code perform in-place updates to the heap, which are handled fully-automatically by Gillian-Rust, as presented in Chapter 18. Note that the matching of the value of `self.head` in line 14 and its dereferencing to access its `prev` field requires unfolding the `dllSeg` predicate once, which is also done automatically.

Next, in line 20, the `len` field of the list is updated. This operation may overflow the value. The current path condition is not sufficient to prove that the overflow will not happen, and execution branches into two paths: one correct path where the overflow does not happen, and one incorrect path which implicitly calls a panic. Before panicking, Gillian-Rust always checks that the current path condition (here, the overflow condition) does not contradict the observation, using the **PROPH-SAT** rule (which entails that $\langle \text{False} \rangle \Rightarrow \text{False}$). Here, the observation, our resource **c**, contradicts the overflow, and the incorrect path is discarded.

Next, `push_front_node` returns, and the `mutref_auto_resolve!` annotation on line 6 tells Gillian-Rust to apply the **MUT-UPDATE** and **MUTREF-RESOLVE** rules in sequence. The former requires the invariant of the linked list to have been restored, with a new representation. At this point, Gillian-Rust automatically folds the `dllSeg` predicate twice, once to revert the unfolding previously performed, and once to push the newly-added node and its ownership predicate (resource **b**) to its front. Then, Gillian-Rust folds the ownership predicate of the linked-list, checking that the first and final pointer are `None`, and that its length field corresponds to the length of its new representation,

which is `self_repr` with `elt_repr` prepended to it. `MUT-UPDATE` is then successfully applied, updating the prophecy controller and observer to match the new representation.

When applying `MUTREF-RESOLVE`, Gillian-Rust understands that the borrow needs to be closed. Since the invariant of the linked list has been correctly restored, the full borrow is automatically closed, and the lifetime token is recovered. `MUTREF-RESOLVE` then discards the resource corresponding to the mutable reference (including the full borrow), and produces the observation required to prove the postcondition of the function.

Finally, the obtained state is matched against the postcondition, which requires: **1)** ownership of the return value, which is vacuously owned as the return type is `unit`; **2)** the lifetime token that was recovered when closing the borrow; and **3)** the observation obtained by applying `MUT-UPDATE`. As the postcondition is satisfied, the specification is verified, and can be soundly used in Creusot.

Chapter 22

Evaluation

We used our hybrid verification pipeline to perform several case studies. First, we verified `EvenInt`, a tutorial example developed by the authors of `RefinedRust`¹, and compare our results with theirs. Next, we verified type safety (TS) and functional correctness (FC) of `LP`, a “linked-pair” data-structure that we developed as a tutorial example for `Gillian-Rust`. We also verified, for the first time to our knowledge, TS and FC of unsafe code from the Rust standard library, specifically a subset of the `LinkedList` and `Vec` modules (with caveats for the latter), with no or minor modifications to the original source code. We also verified an alternative, smaller implementation of `Vec` called `MiniVec`, also verified using `RefinedRust`, in order to provide a performance comparison. Finally, we proved a merge sort algorithm for linked-lists (cf. [Chapter 21](#)), and a gnome sort algorithm for vectors. For each case study, we provide detailed data for performance, lines of code verified, and annotation overhead. All experiments were performed on a MacBook Pro 2019, with 16GB Memory and a 2.3GHz 9-Core Intel Core i9 processor, noting that `Gillian-Rust` is single-threaded.

[Table 22.1](#) presents the results of our evaluation. For each internally unsafe module analysed, we give: the number of executable lines of code; the number of lines of annotations (specifications/predicate definitions/lemmas/proof tactics); the type of properties verified (with FC subsuming TS); and the time taken to verify the properties. We note that verifying only TS allows for the use of a simpler encoding, which eschews prophecies to track value information.

Test Case	Verified Props.	Exec. LoC	Spec. LoC	Time (s)
<code>EvenInt</code>	TS/FC	47	13	0.04s
<code>LP</code>	TS	32	40	0.03s
<code>LP</code>	FC	43	56	0.04s
<code>LinkedList</code>	TS	130	176	0.24s
<code>LinkedList</code>	FC	130	227	0.45s
<code>MiniVec</code>	FC	140	59	1.35s
<code>Vec</code>	TS	294	44	1.08s
<code>Vec</code>	FC	294	107	2.57s

¹ Gäher et al., “`RefinedRust`: A Type System for High-Assurance Verification of Rust Programs”, 2024 [\[Gäh+24\]](#)

Table 22.1: Evaluation results: Verification of internally unsafe modules using `Gillian-Rust`

22.1 `EvenInt`

We start from a small case study provided as part of the evaluation of the `RefinedRust` paper. `EvenInt` is a structure that only contains a single value of type `i32`, for which the ownership invariant requires

the value to be even. We copy all applicable functions from this case study (eliding those that make use of shared references, as they are not yet supported by Gillian-Rust, cf. [Chapter 23](#)) and verify them within Gillian-Rust, by giving Creusot specifications that correspond to the RefinedRust specifications provided. These functions are:

- `new`, an unsafe function that receives an integer and returns an `EvenInt` without further checks;
- `new_2`, a safe function that receives an integer, checks if it is even, and if it is not, adds or removes one to make it even, and then returns the corresponding `EvenInt`;
- `new_3`, a safe function that receives an integer `i` and returns an `Option<EvenInt>`: `Some(i)` if `i` is even, and `None` otherwise;
- `add`, an unsafe function that increments the `EvenInt` value by one, temporarily breaking the soundness invariant of the structure; and
- `add_two`, a safe function that mutates an `EvenInt` in place, calling `add` twice.

The specifications for `new_2` and `new_3` are only TS specifications, and do not guarantee anything about the value contained by the created object. The specification of `add_two` guarantees both TS and that the value of the `EvenInt` object is incremented by two.

The total verification time for these specifications using Gillian-Rust is **0.04s**, which is several orders of magnitude faster than the **4m36s** of RefinedRust. Furthermore, the Gillian-Rust file contains fewer specifications than RefinedRust, as it does not require specifying the internal unsafe functions `new` and `add`, which only serve as auxiliary functions for the public safe functions we care about. Note that, while we chose not to do so, one could write these specifications in Gillian-Rust, and doing so would not observably increase the verification time, given compositionality of Gillian.

In addition, in the `add_two` function, Gillian-Rust requires a single line of annotation to resolve the prophecy (the one in line 6 of [Figure 21.4](#)). In contrast, RefinedRust requires of the user to manually write a Coq proof that if i is an even integer, then $i + 1 + 1$ is still a even integer.

22.2 LinkedList

We verify, to our knowledge for the first time, TS and FC of a subset of the `LinkedList` API from the Rust standard library, with no or minor modification to the original source code, extracted from commit `ad2b34d0` of the official Rust repository, dated April 12, 2023. Specifically, apart from the added annotations required for verification, the only modification made was to manually inline calls to `Option::map`, as its parameter is a closure, which are not yet supported by the Gillian-Rust compiler. Once closures are added to the compiler, there will be no need for additional annotations, as Gillian-Rust can symbolically execute them like any other function, without requiring a specification.

We use the ownership predicate presented in [Figure 17.2](#) and the `dllSeg` predicate provided in [§18.3](#), and prove FC properties for the following 6 functions: `new`, `push_front`, `pop_front`, `push_back`, `pop_back`, `front_mut`. The total verification time is **0.72s**, which includes the verification of

the auxilliary proofs generated by the `extract_lemma` macro, as well as two additional lemmas required for the proof of `push_back` and `pop_back`. These two lemmas change the direction of `dllSeg`, going from a definition that traverses the list from head to tail to one that traverses the list from tail to head, and vice-versa. These lemmas, importantly, are not Rust-specific, but rather essential primitives for any doubly-linked-list formalisation in SL. Both lemmas are written in Rust, and proven within Gillian-Rust without requiring the use of external tools.

22.3 MiniVec and Vec

MiniVec We verify a subset of the API for MiniVec, a simple implementation of the `Vec` module proposed by RefinedRust as a case study. Using specifications that provide similar guarantees to those proven by RefinedRust, we verify FC of the `new`, `with_capacity`, `push`, `pop`, `get_mut`, and `get_unchecked_mut` functions, as well as a simple client function that is part of the RefinedRust case study. Our hybrid pipeline performs verification in **1.35s**, **1.28s** for Gillian-Rust and **0.07s** for Creusot, in contrast with the **30m40s** of RefinedRust.

Standard library Vec In addition to MiniVec, we verify the `Vec` implementation given in the Rust standard library, taken from the same commit as the `LinkedList` module, and targeting the same functions as for MiniVec. In addition, we also verify `index_mut`, which performs a similar operation to `get_unchecked_mut`, but adds a safety check and performs access in memory through slice indexing instead of raw pointer arithmetics. Verifying both of these functions ensures that we correctly support these two different ways of accessing memory in Rust.

The source code of the standard library `Vec` module is substantially more complex than that of MiniVec. For instance, when pushing a value into a `Vec` with capacity 0 (i.e., a vector that has not yet been allocated), the vector is allocated with a capacity that depends on the size of the value being pushed, using the above code snippet, yielding three different paths of execution that must all be checked.

There exist further examples of optimisations within the code of `Vec`, which we do not simplify. This explains why the verification of this module takes longer than the verification of MiniVec, yielding (in our opinion, a still reasonable) **1.08s** for TS and **2.57s** for FC.

It is important to note that `Vec` performs *untyped* allocations by explicitly providing the size of the allocation in bytes, yielding a raw pointer to an uninitialised array of bytes. The pointer is then cast to a pointer to the vector element type and is used to store the typed values. In this process, the corresponding Gillian-Rust heap object has some nodes indexed using the `u8` type and other nodes indexed using the vector element type (cf. §18.2), showcasing its resilience to low-level operations.

```
const MIN_NON_ZERO_CAP: usize =
  if size_of::() == 1 { 8 }
  else if size_of::() <= 1024 {
    4
  }
  else { 1 };
```

Caveats The verification of `Vec` is performed with several caveats. First, both `Vec` and `MiniVec` are special-cased for when the element type is zero-sized (ZST). For these types, no allocation is performed, the

capacity remains 0, and the vector is simply a counter for the length. The vector ownership invariant, however, still needs to capture `len` times the resource corresponding to the ownership invariant of the type of the elements. As Gillian is untyped, expressing this invariant is difficult, as we are not able to exhibit *the only representative of the ZST type*. Therefore, we disallow ZSTs as types of vector elements. We plan to overcome this limitation by allowing the state model of Gillian-Rust to produce these representatives. This limitation, however, does mean that RefinedRust considers several more paths of execution than Gillian-Rust does for MiniVec.

Furthermore, the borrow extraction lemma required to prove FC of `get_index_mut` and `index_mut` requires the proof of a magic wand that Gillian-Rust is not yet able to automate, and is left unproven for now. We plan to add support for manually specifying the proofs for extract lemmas when the tool is unable to automate them.

Finally, we slightly modified the source code of the standard library `Vec` module, in the following ways. First, the vector type is parametric on an allocator, which we remove and perform all allocation by calling the Gillian-Rust allocator. We also inline calls to functions such as `Result::map`, which receive a closure as parameter, due to the above-mentioned lack of support for closures. Finally, the real implementation of `index_mut` when using `usize` as an index is hidden behind a few layers of trait indirection; we manually inline layers so that `index_mut` is a single function.

22.4 Hybrid Verification

We argue that a *hybrid* approach combining Gillian-Rust with Creusot enables higher performance and flexibility in verification. To validate this, it is essential to answer two questions (1) “Can Gillian-Rust effectively verify Creusot-style specifications?”; and (2) “Can those specifications then be efficiently used from Creusot’s perspective?”.

In [Chapter 21](#), we presented our hybrid macros, which act as a bridge between the two tools, interpreting the specifications appropriately as either Pearlite or Gilsonite. We used these macros to specify and verify the examples presented in [Table 22.1](#), conclusively answering (1). The code generated by the hybrid macros is intuitive and often identical to the raw Gilsonite specification we would write by hand. We have noticed no impact on verification times caused by use of the hybrid macros.

Our answer to (2) comes in two parts. Firstly, we note that Creusot provides the `creusot_contracts` crate, which provides standard, trusted specifications for common Rust types through its `extern_spec!` macro. These specifications are either identical or semantically equivalent to the ones proved by Gillian-Rust and at most a safe wrapper would be required by Creusot to prove the entailment.

Secondly, we implemented and verified several safe programs using the specifications obtained by Gillian-Rust and observed favourable verification times. Specifically, we implemented: **Merge Sort**, consisting of 55 lines of specification, 56 lines of generic lemmas about permu-

tations missing from Creusot’s standard library, and 68 executable lines of code taking 6.3s (wall) 28.7s (user) to verify; **Gnome Sort** consisting of 6 lines of specification and 17 executable lines of code taking 2.6s (wall), 4.6s (user) to verify; and **Right Pad** consisting of 11 lines of specification and 12 executable lines of code taking 0.6s (wall) 0.4s (user) to verify.

Chapter 23

Limitations and Future work

In its present form, our infrastructure is a preliminary proof of concept, demonstrating the feasibility of our hybrid methodology in end-to-end Rust verification. As such, it comes with several limitations, both in the implementation and meta-theory. We identify and discuss these limitations, and outline how they will be addressed in future.

23.1 Unimplemented features

Importantly, the features presented here only pose engineering challenges and mostly require time and engineering effort rather than further scientific insight.

Compiler coverage The Gillian-Rust compiler is missing support for some language constructs, such as closures, but we do not foresee any complications arising from these extensions. In particular, Gillian supports dynamic calls, extensively tested with JavaScript, and therefore would not have problems dealing with closures and other function pointers.

Connection to the Borrow Checker The implementation of the borrow checker in `rustc` makes it difficult to extract values with their lifetimes annotated. This is an intentional erasure performed by `rustc` to avoid introducing accidental dependencies on lifetimes during compilation. This makes it challenging for us to implement specifications and predicates which involve multiple lifetimes, like the ones that would be required for types like `IterMut<'_,>`, an iterator producing mutable borrows to successive elements of a vector. Instead, our implementation is limited to predicates containing only one lifetime, a property which we can easily check and enforce with current compiler APIs. Lifting this limitation will likely require writing a custom borrow-checker pass for our specifications.

23.2 Meta-theory simplifications

The meta-theory of Gillian-Rust presented in this manuscript heavily relies on both `RustBelt` and `RustHornBelt`, but makes two simplifications that need to be formally justified; we believe that this is possible, though it involves a substantial amount of additional work.

Later modalities As `Iris` is a step-indexed logic, original rules from `RustBelt` and `RustHornBelt`, such as `LFTL-BORROW-ACC`, use *later* modalities. We simplify later modalities away, as the meta-theory of

Gillian cannot account for them, given that it is formalised using a non-step-indexed separation logic. We believe that, if Gillian is formalised in Iris, the unfolding and folding ghost commands would be formalised as view shifts which “take a step”, as they are formalised similarly to primitive memory operations. This would be enough to justify the soundness of our approach.

In addition, all described paradoxes that would arise in Iris without step-indexing make extensive use of the impredicativity of the logic. On the other hand, Gillian uses a predicative logic, and it is unclear that any paradoxes could arise even without step-indexing. Unfortunately, providing a more formal version of these arguments would either require formalising the meta-theory of Gillian in Iris, or proving RustBelt and RustHornBelt rules using the meta-theory of Gillian, both of which exceed the scope of the current project.

Prophecy dependencies and type well-formedness Although not presented in the associated paper, the Coq development of RustHornBelt attaches an additional proof obligation to the definitions of Rust types, called `ty_own_proph`, which states that given an ownership predicate, we must be able to extract the tokens for all prophecy variables occurring in the representation. This is a crucial property for the soundness of RustHornBelt as it ensures the absence of ‘causal loops’, prophecies which depend on their own value.

We believe that the current implementation of Gillian-Rust naturally enforces this constraint, thanks to a dataflow requirement imposed on assertion definitions. In Gillian, all predicate parameters must be declared with a mode, `In` or `Out`, such that out-parameters can be learned by the in-parameters. For example, in the core predicate $a \mapsto_T v$, address a and type T are in-parameters, as v can be learned uniquely by querying the heap.

In Gillian-Rust, the ownership predicate `fn own(self, repr: ...)` declares `self` to be an in-parameter and `repr` to be an out-parameter, and Gillian performs an analysis which ensures that the former must be *sufficient* to learn the latter. Because the ownership predicate of a mutable reference is the only way to obtain a representation which depends on prophecy variables, and this predicate provides the associated token, it is impossible to construct an ownership predicate that does not satisfy `ty_own_proph`. However, formally proving this property within RustHornBelt is extremely difficult, as it would require a deep embedding of the assertion language that would enable the formalisation of the dataflow analysis performed by Gillian.

23.3 Unexplored topics

Finally, while our work advances the state-of-the-art when it comes to verification of Rust programs, it does leave several related topics unexplored.

Shared references For the moment, we do not explore shared references and their ownership predicates. The path forward is to introduce

another trait to the Gilsonite API, called `shareable`, which allows one to define the *sharing* predicate for a given type, as defined by RustBelt. Furthermore, we would need to implement a variant of the guarded predicate algebra which behaves according to the rules governing the behaviour of *fractured borrows*, which are the shared counterpart of full borrows for most types. Gillian-Rust would then be able to derive the ownership predicate for shared references of type $\&\tau$, where τ is `shareable`. These enhancements require substantial additional work, which we consider outside the scope of the current presentation, as it aims only at proving the feasibility of our approach. In addition, as prophecies are separate from the representation of shared references, it is likely that the step between supporting them for type safety verification and functional correctness reasoning is minimal.

Concurrency While all of our proven specifications are valid in a concurrent context, we do not explore constructs specific to concurrency. In particular, ownership predicates in RustBelt receive an additional argument corresponding to a thread identifier. Types that are thread-safe—said to be `Send` in Rust—may have an ownership predicate which depends on this identifier. Our approach would have to be extended with thread identifiers in order to prove properties about such types.

Borrows Finally, we do not model `StackedBorrows`¹ or `TreeBorrows`², which are operational semantic models of the aliasing model of Rust. While preliminary research has been conducted on creating a logic to reason about `StackedBorrows`³, symbolic reasoning in the presence of these models has, to our knowledge, not been performed before. Moreover, there is still no foundational framework that marries any of these models with the theory of semantic typing proposed by RustBelt, and we can therefore not model them confidently within Gillian.

¹ Jung et al., “Stacked borrows: an aliasing model for Rust”, 2019 [Jun+19]

² Jung et al., *From Stacks to Trees: A new aliasing model for Rust*, 2023 [JV23]

³ Louwink, *A Separation Logic for Stacked Borrows*, 2021 [Lou21]

Chapter 24

Related work

We provide an overview of the relevant literature on unsafe Rust verification. While there exist many tools other than Creusot for safe Rust verification (such as Prusti¹, Aeneas², or Flux³), we do not address them in detail as our goal is to reason about unsafe Rust. To our knowledge, currently there exist four tools capable of performing this reasoning—RefinedRust⁴, VeriFast⁵, Verus⁶, and Kani⁷—none of which explore the idea of hybrid verification.

RefinedRust In line with RefinedC⁸, RefinedRust⁹ allows users to annotate functions with refinement types, enabling semi-automatic verification of Rust programs with unsafe code. It compiles real-world Rust programs into an enhanced version of RustBelt’s λ_{Rust} before performing a *foundational* proof in Coq: its trusted computing base is smaller than that of Gillian-Rust, only comprising the Rust-to- λ_{Rust} compiler and Coq itself. By building on the strong foundation of RustBelt, RefinedRust can support certain features not present in Gillian-Rust, such as shared references.

RefinedRust extends RustBelt with new techniques for automating and simplifying reasoning about unsafe code, in some instances automating reasoning that requires annotations in Gillian-Rust; we will explore how these techniques can be incorporated into Gillian-Rust.

However, while RefinedRust does allow for the verification of functional correctness of unsafe code, it does not explore the potential synergy between safe and unsafe verification tools, which is the key feature of our approach. Moreover, our preliminary evaluation suggests that Gillian-Rust is several orders of magnitude faster than RefinedRust for verification of unsafe code. Finally, RefinedRust requires the user to write Iris annotations, which we believe to be a higher barrier to entry than the traits and macros proposed by Gillian-Rust.

Importantly, the performance and lower overhead of Gillian-Rust made it possible for us to verify the actual implementations of types such as `LinkedList` and `Vec` from the Rust standard library, whereas the verification of RefinedRust is done on `MiniVec`, a simplified version of `Vec`.

VeriFast for Rust Rahimi Froushaani et al.¹⁰ describe a Rust front-end for VeriFast¹¹ which provides a way of verifying semantic type safety for a fragment of unsafe Rust. By design, this work is similar to ours, as both VeriFast and Gillian rely on a similar theoretical framework of compositional symbolic execution through consumers and producers. However, again¹², VeriFast exists at a different point of the design space, sacrificing some automation that Gillian can provide, in exchange for

¹ Astrauskas et al., “Leveraging rust types for modular specification and verification”, 2019 [Ast+19]

² Ho et al., “Aeneas: Rust verification by functional translation”, 2022 [HP22]

³ Lehmann et al., *Flux: Liquid Types for Rust*, 2022 [Leh+22]

⁴ Gäher et al., “RefinedRust: A Type System for High-Assurance Verification of Rust Programs”, 2024 [Gäh+24]

⁵ Froushaani et al., *Modular Formal Verification of Rust Programs with Unsafe Blocks*, 2022 [FJ22]

⁶ Lattuada et al., “Verus: Verifying Rust Programs using Linear Ghost Types”, 2023 [Lat+23]

⁷ Team, *How Open Source Projects are Using Kani to Write Better Software in Rust / AWS Open Source Blog*, 2023 [Tea23]

⁸ Sammler et al., “RefinedC: automating the foundational verification of C code with refined ownership types”, 2021 [Sam+21]

⁹ Gäher et al., “RefinedRust: A Type System for High-Assurance Verification of Rust Programs”, 2024 [Gäh+24]

¹⁰ Froushaani et al., *Modular Formal Verification of Rust Programs with Unsafe Blocks*, 2022 [FJ22]

¹¹ Jacobs et al., “A Quick Tour of the VeriFast Program Verifier”, 2010 [JSP10]

¹² See the related work of the two previous parts of the manuscript

more speed, predictability, and a robust capacity to explore innovative semi-automatic proof techniques. In particular, VeriFast does not support encoding custom state models, which therefore requires manual application of the rules which Gillian-Rust automates.

The current Rust front-end of VeriFast focuses solely on verifying semantic type safety and not functional correctness—essentially, it encapsulates a portion of RustBelt, but does not extend beyond it to RustHornBelt. Furthermore, it uses an encoding of borrows which requires manual management on the part of the users, while the guarded predicate mechanism of Gillian-Rust facilitates much greater automation. Having said that, the insights of this work and VeriFast itself have contributed to overcoming some of the difficulties encountered during the design and implementation of Gillian-Rust.

Verus As opposed to Gillian-Rust and VeriFast, Verus¹³ does not use SL but rather linear ghost types to encode ownership properties. This approach allows it to leverage the borrow checker of the Rust compiler to drastically improve the encoding into SMT. It also means that writing proofs *feels like* writing Rust code, providing a familiar user experience.

However, Verus does not support traditional raw pointers, meaning that it is not able to verify ‘traditional’ unsafe code. For example, using Verus, one cannot verify the standard library implementation of `LinkedList` in the way that we propose. Instead, Verus developers have verified their own implementation of the `LinkedList` library, keeping track of linear ghost objects (denoted by `PointsTo<T>`, a Verus primitive) to implement links between nodes. In that sense, Verus could be considered a verifier shallowly embedded in an extension of Rust, rather than a Rust verifier.

In addition, since Verus does not use SL, its specification of the heavily pointer-based invariants such as that of doubly-linked list is more verbose than their SL counterparts. Moreover, we have found that, for the verification of heavily pointer-based code, Verus requires many more annotations than Gillian-Rust, as it does not have the ability to use separation logic. Moreover, without separation logic, Verus’s specification of a doubly-linked-list also contains universal quantifiers that are encoded into SMT, while Gillian-Rust’s version of the same specification is entirely quantifier-free.

Finally, Verus cannot currently reason about functions that return mutable references.

Kani Kani¹⁴ is a bounded model checker for Rust, which compiles Rust to the intermediate representation ingested by CBMC¹⁵ for its analysis. While Kani is a well-engineered tool of industrial-strength, which covers an impressive fragment of the existing Rust language, it does not propose solutions to the challenges solved by our work.

First, as it uses CBMC as a back-end, it performs bounded model checking rather than unbounded verification, which is our current task. Kani encodes Rust programs into a C-like representation, instantiating a specific layout for each structure, and performing model checking

¹³ Lattuada et al., “Verus: Verifying Rust Programs using Linear Ghost Types”, 2023 [Lat+23]

¹⁴ Team, *How Open Source Projects are Using Kani to Write Better Software in Rust / AWS Open Source Blog*, 2023 [Tea23]

¹⁵ Clarke et al., “A Tool for Checking ANSI-C Programs”, 2004 [CKL04]

in that context. As a result, layout-sensitive Rust code is beyond the current ability of Kani. Finally, Kani treats all safe and unsafe code alike, and, as such, is unable to leverage any of the safe Rust guarantees to enhance analysis, unlike our hybrid approach.

Chapter 25

Future work

In this manuscript, we have presented a sound, flexible, and expressive formalisation of Gillian, and demonstrated the applicability of the framework to the C and Rust programming languages. Gillian, nonetheless, very much remains an open area of research and, to conclude, we propose several avenues for future exploration.

Firstly, as the development of Gillian-Rust is in its early stages, one immediate objective would be to properly address the limitations that have been identified in [Chapter 23](#).

Second, we have started questioning the necessity of an intermediate language, such as GIL or SIGIL. A potentially more efficient strategy would involve crafting a compositional symbolic interpreter directly tailored to a specific target language or a target-specific intermediate language, such as MIR for Rust. This approach could reduce implementation errors and simplify maintenance. Additionally, it could significantly enhance the user interface by allowing error messages to be directly communicated in the target language, thereby eliminating the cumbersome process of “error message lifting”.

In order to write such compositional symbolic interpreters for diverse languages without repetitive re-implementation, we would need to build a comprehensive library of essential functions. This manuscript constitutes a preliminary step, introducing components such as the symbolic execution monad and a sound parametric specification execution function. However, further work is required to validate the flexibility of this approach and expand the meta-theoretical framework in order to be able to model it.

Thirdly, it could be the case that Resource Algebras à la Iris are more suitable than PCMs for defining compositional state models, leading us to consider a new formalisation of Gillian based directly on Iris. Using RAs could solidify the connections between Gillian-Rust, RustBelt, and RustHornBelt, with direct proofs in Iris. Within such a framework, two strategies would enhance trust in Gillian: extending its engine to *produce Iris proofs* during symbolic execution for external verification or validating the soundness of the engine directly in Coq, thus reducing the TCB to the soundness of Coq and of the SMT solver.

Finally, Gillian could be applied to code written using several languages and foreign function interfaces (FFIs). For example, the Melocoton^{1,2} project introduces Iris rules for reasoning about OCaml code that uses C FFI. The Melocoton logic has distinct points-to predicates for OCaml and C views of the memory, and rules that *convert* between the OCaml and C representations. This conversion process resembles combining smaller contiguous points-to predicates in C into a larger one. Gillian’s unique flexibility for symbolic memory representation could facilitate the automation of Melocoton rules, as it did for C.

¹ Guéneau et al., “Melocoton: A Program Logic for Verified Interoperability Between OCaml and C”, 2023 [Gué+23]

² This idea came after the creators of Melocoton asked us about the potential applicability of Gillian to the automation of the Melocoton Iris rules.

Bibliography

- [AJP15] Pieter Agten, Bart Jacobs, and Frank Piessens. “Sound Modular Verification of C Code Executing in an Unverified Context”. In: *ACM SIGPLAN Notices* 50.1 (Jan. 2015), pp. 581–594. ISSN: 0362-1340. DOI: [10.1145/2775051.2676972](https://doi.org/10.1145/2775051.2676972). URL: <https://dl.acm.org/doi/10.1145/2775051.2676972> (visited on 04/30/2024).
- [Ama24a] Amazon Web Services. *AWS Encryption SDK message format reference - AWS Encryption SDK*. 2024. URL: <https://docs.aws.amazon.com/encryption-sdk/latest/developer-guide/message-format.html#header-structure> (visited on 05/05/2024).
- [Ama24b] Amazon Web Services. *aws/aws-encryption-sdk-c*. original-date: 2018-01-09T18:28:20Z. May 2024. URL: <https://github.com/aws/aws-encryption-sdk-c> (visited on 05/05/2024).
- [App11a] Andrew W. Appel. “Verified Software Toolchain”. en. In: *Programming Languages and Systems*. Ed. by Gilles Barthe. Berlin, Heidelberg: Springer, 2011, pp. 1–17. ISBN: 978-3-642-19718-5. DOI: [10.1007/978-3-642-19718-5_1](https://doi.org/10.1007/978-3-642-19718-5_1).
- [App11b] Andrew W. Appel. “VeriSmall: Verified Smallfoot Shape Analysis”. In: *Certified Programs and Proofs*. Ed. by Jean-Pierre Jouannaud and Zhong Shao. Berlin, Heidelberg: Springer, 2011, pp. 231–246. ISBN: 978-3-642-25379-9. DOI: [10.1007/978-3-642-25379-9_18](https://doi.org/10.1007/978-3-642-25379-9_18).
- [Ast+19] Vytautas Astrauskas, Peter Müller, Federico Poli, and Alexander J. Summers. “Leveraging rust types for modular specification and verification”. In: *Proceedings of the ACM on Programming Languages* 3.OOPSLA (Oct. 2019), 147:1–147:30. DOI: [10.1145/3360573](https://doi.org/10.1145/3360573). URL: <https://doi.org/10.1145/3360573> (visited on 01/05/2023).
- [Ast+20] Vytautas Astrauskas, Christoph Matheja, Federico Poli, Peter Müller, and Alexander J. Summers. “How do programmers use unsafe Rust?” In: *Proceedings of the ACM on Programming Languages* 4.OOPSLA (Nov. 2020), 136:1–136:27. DOI: [10.1145/3428204](https://doi.org/10.1145/3428204). URL: <https://dl.acm.org/doi/10.1145/3428204> (visited on 11/17/2023).
- [AWZ88] B. Alpern, M. N. Wegman, and F. K. Zadeck. “Detecting equality of variables in programs”. In: *Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. POPL ’88. New York, NY, USA: Association for Computing Machinery, Jan. 1988, pp. 1–11. ISBN: 978-0-89791-252-5. DOI: [10.1145/73560.73561](https://doi.org/10.1145/73560.73561). URL: <https://dl.acm.org/doi/10.1145/73560.73561> (visited on 11/30/2023).
- [Ayo+25] Sacha-Élie Ayoun, Xavier Denis, Petar Maksimović, and Philippa Gardner. “A hybrid approach to semi-automated Rust verification”. In: *Proceedings of the 46th ACM SIGPLAN International Conference on Programming Language Design and Implementation*. PLDI 2025. Cham: Association for Computing Machinery, June 2025.
- [Ayo19a] Sacha-Élie Ayoun. *fix buffer overflow - Collections-C - Github*. 2019. URL: <https://github.com/srdja/Collections-C/pull/119> (visited on 05/13/2024).
- [Ayo19b] Sacha-Élie Ayoun. *fix djb2 string hash function - Collections-C - Github*. 2019. URL: <https://github.com/srdja/Collections-C/pull/126> (visited on 05/13/2024).
- [Ayo19c] Sacha-Élie Ayoun. *fix over allocation of ring_buffers - Collections-C - Github*. 2019. URL: <https://github.com/srdja/Collections-C/pull/125> (visited on 05/13/2024).
- [Ayo19d] Sacha-Élie Ayoun. *Fix some tests in the list test suite - Collections-C - Github*. 2019. URL: <https://github.com/srdja/Collections-C/pull/123> (visited on 05/13/2024).

- [Ayo19e] Sacha-Élie Ayoun. *remove the usage of cc_comp_ptr everywhere - Collections-C - Github*. 2019. URL: <https://github.com/srdja/Collections-C/pull/122> (visited on 05/13/2024).
- [Ayo21a] Sacha-Élie Ayoun. *Correctly fail on invalid aad length - aws-encryption-sdk-c - Github*. 2021. URL: <https://github.com/aws/aws-encryption-sdk-c/pull/696> (visited on 05/13/2024).
- [Ayo21b] Sacha-Élie Ayoun. *Small over allocation in each aws_string - aws-c-common - Github*. 2021. URL: <https://github.com/aws/aws-c-common/issues/776> (visited on 05/13/2024).
- [Ayo21c] Sacha-Élie Ayoun. *Undefined Behaviour corner case for aws_byte_cursor_advance - aws-c-common - Github*. 2021. URL: <https://github.com/aws/aws-c-common/issues/771> (visited on 05/13/2024).
- [Ayo22] Sacha-Élie Ayoun. *Kanillian - Github*. 2022. URL: <https://github.com/giltho/kanillian> (visited on 05/13/2024).
- [Bar+22] Haniel Barbosa, Clark Barrett, Martin Brain, Gereon Kremer, Hanna Lachnitt, Makai Mann, Abdalrhman Mohamed, Mudathir Mohamed, Aina Niemetz, Andres Nötzli, Alex Ozdemir, Mathias Preiner, Andrew Reynolds, Ying Sheng, Cesare Tinelli, and Yoni Zohar. “cvc5: A Versatile and Industrial-Strength SMT Solver”. en. In: *Tools and Algorithms for the Construction and Analysis of Systems*. Ed. by Dana Fisman and Grigore Rosu. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2022, pp. 415–442. ISBN: 978-3-030-99524-9. DOI: [10.1007/978-3-030-99524-9_24](https://doi.org/10.1007/978-3-030-99524-9_24).
- [BCO06] Josh Berdine, Cristiano Calcagno, and Peter W. O’Hearn. “Smallfoot: Modular Automatic Assertion Checking with Separation Logic”. en. In: *Formal Methods for Components and Objects*. Ed. by Frank S. de Boer, Marcello M. Bonsangue, Susanne Graf, and Willem-Paul de Roever. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2006, pp. 115–137. ISBN: 978-3-540-36750-5. DOI: [10.1007/11804192_6](https://doi.org/10.1007/11804192_6).
- [BEL75] Robert S. Boyer, Bernard Elspas, and Karl N. Levitt. “SELECT—a formal system for testing and debugging programs by symbolic execution”. In: *Proceedings of the international conference on Reliable software*. New York, NY, USA: Association for Computing Machinery, Apr. 1975, pp. 234–245. ISBN: 978-1-4503-7385-2. DOI: [10.1145/800027.808445](https://doi.org/10.1145/800027.808445). URL: <https://dl.acm.org/doi/10.1145/800027.808445> (visited on 08/18/2023).
- [BFT16] Clark Barrett, Pascal Fontaine, and Cesare Tinelli. *The Satisfiability Modulo Theories Library (SMT-LIB)*. www.SMT-LIB.org. 2016.
- [BKC14] Stefan Bucur, Johannes Kinder, and George Candea. “Prototyping symbolic execution engines for interpreted languages”. In: *ACM SIGARCH Computer Architecture News* 42.1 (Feb. 2014), pp. 239–254. ISSN: 0163-5964. DOI: [10.1145/2654822.2541977](https://doi.org/10.1145/2654822.2541977). URL: <https://dl.acm.org/doi/10.1145/2654822.2541977> (visited on 05/01/2024).
- [Blo+17] Stefan Blom, Saeed Darabi, Marieke Huisman, and Wytse Oortwijn. “The VerCors Tool Set: Verification of Parallel and Concurrent Software”. en. In: *Integrated Formal Methods*. Ed. by Nadia Polikarpova and Steve Schneider. Cham: Springer International Publishing, 2017, pp. 102–110. ISBN: 978-3-319-66845-1. DOI: [10.1007/978-3-319-66845-1_7](https://doi.org/10.1007/978-3-319-66845-1_7).
- [Bor+05] Richard Bornat, Cristiano Calcagno, Peter O’Hearn, and Matthew Parkinson. “Permission accounting in separation logic”. In: *ACM SIGPLAN Notices* 40.1 (Jan. 2005), pp. 259–270. ISSN: 0362-1340. DOI: [10.1145/1047659.1040327](https://doi.org/10.1145/1047659.1040327). URL: <https://dl.acm.org/doi/10.1145/1047659.1040327> (visited on 02/27/2024).
- [Bot+11] Matko Botinčan, Dino Distefano, Mike Dodds, Radu Grigora, Daiva Naudžiūnienė, and Matthew J. Parkinson. “coreStar: the Core of jStar”. In: *Boogie 2011*. 2011.

- [BPS09] Matko Botinčan, Matthew Parkinson, and Wolfram Schulte. “Separation Logic Verification of C Programs with an SMT Solver”. en. In: *Electronic Notes in Theoretical Computer Science*. Proceedings of the 4th International Workshop on Systems Software Verification (SSV 2009) 254 (Oct. 2009), pp. 5–23. ISSN: 1571-0661. DOI: [10.1016/j.entcs.2009.09.057](https://doi.org/10.1016/j.entcs.2009.09.057). URL: <https://www.sciencedirect.com/science/article/pii/S1571066109004113> (visited on 01/05/2023).
- [Cad+08] Cristian Cadar, Vijay Ganesh, Peter M. Pawlowski, David L. Dill, and Dawson R. Engler. “EXE: Automatically Generating Inputs of Death”. In: *ACM Transactions on Information and System Security* 12.2 (Dec. 2008), 10:1–10:38. ISSN: 1094-9224. DOI: [10.1145/1455518.1455522](https://doi.org/10.1145/1455518.1455522). URL: <https://dl.acm.org/doi/10.1145/1455518.1455522> (visited on 05/13/2024).
- [Cal+09] Cristiano Calcagno, Dino Distefano, Peter O’Hearn, and Hongseok Yang. “Compositional shape analysis by means of bi-abduction”. In: *ACM SIGPLAN Notices* 44.1 (Jan. 2009), pp. 289–300. ISSN: 0362-1340. DOI: [10.1145/1594834.1480917](https://doi.org/10.1145/1594834.1480917). URL: <https://doi.org/10.1145/1594834.1480917> (visited on 01/05/2023).
- [CD11] Cristiano Calcagno and Dino Distefano. “Infer: An Automatic Program Verifier for Memory Safety of C Programs”. en. In: *NASA Formal Methods*. Ed. by Mihaela Bobaru, Klaus Havelund, Gerard J. Holzmann, and Rajeev Joshi. Berlin, Heidelberg: Springer, 2011, pp. 459–465. ISBN: 978-3-642-20398-5. DOI: [10.1007/978-3-642-20398-5_33](https://doi.org/10.1007/978-3-642-20398-5_33).
- [CDE08] Cristian Cadar, Daniel Dunbar, and Dawson 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*. OSDI’08. USA: USENIX Association, Dec. 2008, pp. 209–224. (Visited on 08/21/2023).
- [CKL04] Edmund Clarke, Daniel Kroening, and Flavio Lerda. “A Tool for Checking ANSI-C Programs”. en. In: *Tools and Algorithms for the Construction and Analysis of Systems*. Ed. by Kurt Jensen and Andreas Podelski. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2004, pp. 168–176. ISBN: 978-3-540-24730-2. DOI: [10.1007/978-3-540-24730-2_15](https://doi.org/10.1007/978-3-540-24730-2_15).
- [Coo22] Kees Cook. *[GIT PULL] Rust introduction for v6.1-rc1*. <https://lore.kernel.org/lkml/202210010816.1317F2C@keescook>. Accessed: Nov. 16th 2023. Oct. 2022. (Visited on 11/16/2023).
- [Cou+05] Patrick Cousot, Radhia Cousot, Jérôme Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. “The ASTREE Analyzer”. en. In: *Programming Languages and Systems*. Ed. by Mooly Sagiv. Berlin, Heidelberg: Springer, 2005, pp. 21–30. ISBN: 978-3-540-31987-0. DOI: [10.1007/978-3-540-31987-0_3](https://doi.org/10.1007/978-3-540-31987-0_3).
- [CR08] Bor-Yuh Evan Chang and Xavier Rival. “Relational inductive shape analysis”. In: *ACM SIGPLAN Notices* 43.1 (Jan. 2008), pp. 247–260. ISSN: 0362-1340. DOI: [10.1145/1328897.1328469](https://doi.org/10.1145/1328897.1328469). URL: <https://dl.acm.org/doi/10.1145/1328897.1328469> (visited on 03/17/2023).
- [CS23] Arthur Correnson and Dominic Steinhöfel. “Engineering a Formally Verified Automated Bug Finder”. In: *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ESEC/FSE 2023. New York, NY, USA: Association for Computing Machinery, Nov. 2023, pp. 1165–1176. ISBN: 9798400703270. DOI: [10.1145/3611643.3616290](https://doi.org/10.1145/3611643.3616290). URL: <https://dl.acm.org/doi/10.1145/3611643.3616290> (visited on 02/09/2024).
- [Dar+22] Thibault Dardinier, Gaurav Parthasarathy, Noé Weeks, Peter Müller, and Alexander J. Summers. “Sound Automation of Magic Wands”. en. In: *Computer Aided Verification*. Ed. by Sharon Shoham and Yakir Vazel. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2022, pp. 130–151. ISBN: 978-3-031-13188-2. DOI: [10.1007/978-3-031-13188-2_7](https://doi.org/10.1007/978-3-031-13188-2_7).

- [DB08] Leonardo De Moura and Nikolaj Bjørner. “Z3: an efficient SMT solver”. In: *Proceedings of the Theory and practice of software, 14th international conference on Tools and algorithms for the construction and analysis of systems*. TACAS’08/ETAPS’08. Berlin, Heidelberg: Springer-Verlag, Mar. 2008, pp. 337–340. ISBN: 978-3-540-78799-0. (Visited on 08/18/2023).
- [Dev19] Dominique Devriese. “Modular Effects in Haskell through Effect Polymorphism and Explicit Dictionary Applications: A New Approach and the μ VeriFast Verifier as a Case Study”. In: *Proceedings of the 12th ACM SIGPLAN International Symposium on Haskell*. Haskell 2019. New York, NY, USA: Association for Computing Machinery, Aug. 2019, pp. 1–14. ISBN: 978-1-4503-6813-1. DOI: [10.1145/3331545.3342589](https://doi.org/10.1145/3331545.3342589). (Visited on 03/22/2025).
- [Dis+19] Dino Distefano, Manuel Fähndrich, Francesco Logozzo, and Peter W. O’Hearn. “Scaling static analyses at Facebook”. In: *Communications of the ACM* 62.8 (July 2019), pp. 62–70. ISSN: 0001-0782. DOI: [10.1145/3338112](https://doi.org/10.1145/3338112). URL: <https://dl.acm.org/doi/10.1145/3338112> (visited on 04/11/2024).
- [Dis09] Dino Distefano. “Attacking Large Industrial Code with Bi-abductive Inference”. en. In: *Formal Methods for Industrial Critical Systems*. Ed. by María Alpuente, Byron Cook, and Christophe Joubert. Berlin, Heidelberg: Springer, 2009, pp. 1–8. ISBN: 978-3-642-04570-7. DOI: [10.1007/978-3-642-04570-7_1](https://doi.org/10.1007/978-3-642-04570-7_1).
- [DJ23] Xavier Denis and Jacques-Henri Jourdan. “Specifying and Verifying Higher-order Rust Iterators”. en. In: *Tools and Algorithms for the Construction and Analysis of Systems*. Ed. by Sriram Sankaranarayanan and Natasha Sharygina. Lecture Notes in Computer Science. Cham: Springer Nature Switzerland, 2023, pp. 93–110. ISBN: 978-3-031-30820-8. DOI: [10.1007/978-3-031-30820-8_9](https://doi.org/10.1007/978-3-031-30820-8_9).
- [DJM22] Xavier Denis, Jacques-Henri Jourdan, and Claude Marché. “Creusot: a Foundry for the Deductive Verification of Rust Programs”. en. In: Springer Verlag, Oct. 2022. URL: <https://hal.inria.fr/hal-03737878> (visited on 01/03/2023).
- [DP08] Dino Distefano and Matthew J. Parkinson J. “jStar: towards practical verification for Java”. In: *ACM SIGPLAN Notices* 43.10 (Oct. 2008), pp. 213–226. ISSN: 0362-1340. DOI: [10.1145/1449955.1449782](https://doi.org/10.1145/1449955.1449782). URL: <https://dl.acm.org/doi/10.1145/1449955.1449782> (visited on 04/21/2024).
- [Ecm23] Ecma International. *ECMAScript 2023 Language Specification*. 2023. URL: <https://www.ecma-international.org/publications-and-standards/standards/ecma-262/>.
- [EM18] Marco Eilers and Peter Müller. “Nagini: A Static Verifier for Python”. en. In: *Computer Aided Verification*. Ed. by Hana Chockler and Georg Weissenbacher. Cham: Springer International Publishing, 2018, pp. 596–603. ISBN: 978-3-319-96145-3. DOI: [10.1007/978-3-319-96145-3_33](https://doi.org/10.1007/978-3-319-96145-3_33).
- [FFF09] Matthias Felleisen, Robert Bruce Findler, and Matthew Flatt. *Semantics Engineering with PLT Redex*. The MIT Press, 2009. ISBN: 978-0-262-06275-6.
- [FJ22] Nima Rahimi Froushaani and Bart Jacobs. *Modular Formal Verification of Rust Programs with Unsafe Blocks*. arXiv:2212.12976 [cs]. Dec. 2022. DOI: [10.48550/arXiv.2212.12976](https://doi.org/10.48550/arXiv.2212.12976). URL: <http://arxiv.org/abs/2212.12976> (visited on 03/03/2023).
- [Fra+17] José Fragoso Santos, Petar Maksimović, Daiva Naudžiūnienė, Thomas Wood, and Philippa Gardner. “JaVerT: JavaScript verification toolchain”. In: *Proc. ACM Program. Lang.* 2.POPL (Dec. 2017), 50:1–50:33. DOI: [10.1145/3158138](https://doi.org/10.1145/3158138). URL: <https://dl.acm.org/doi/10.1145/3158138> (visited on 09/13/2024).
- [Fra+19] José Fragoso Santos, Petar Maksimović, Gabriela Sampaio, and Philippa Gardner. “JaVerT 2.0: compositional symbolic execution for JavaScript”. In: *Proceedings of the ACM on Programming Languages* 3.POPL (Jan. 2019), 66:1–66:31. DOI: [10.1145/3290379](https://doi.org/10.1145/3290379). URL: <https://doi.org/10.1145/3290379> (visited on 01/12/2023).

- [Fra+20] José Fragoso Santos, Petar Maksimović, Sacha-Élie Ayoun, and Philippa 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*. PLDI 2020. New York, NY, USA: Association for Computing Machinery, June 2020, pp. 927–942. ISBN: 978-1-4503-7613-6. DOI: [10.1145/3385412.3386014](https://doi.org/10.1145/3385412.3386014). URL: <https://doi.org/10.1145/3385412.3386014> (visited on 11/23/2022).
- [Gäh+24] Lennard Gäher, Michael Sammler, Ralf Jung, Robbert Krebbers, and Derek Dreyer. “RefinedRust: A Type System for High-Assurance Verification of Rust Programs”. In: *Proc. ACM Program. Lang.* PLDI (2024). DOI: [10.1145/3656422](https://doi.org/10.1145/3656422). URL: <https://doi.org/10.1145/3656422>.
- [GKS05] Patrice Godefroid, Nils Klarlund, and Koushik Sen. “DART: directed automated random testing”. In: *ACM SIGPLAN Notices* 40.6 (June 2005), pp. 213–223. ISSN: 0362-1340. DOI: [10.1145/1064978.1065036](https://dl.acm.org/doi/10.1145/1064978.1065036). URL: <https://dl.acm.org/doi/10.1145/1064978.1065036> (visited on 05/13/2024).
- [GLM08] Patrice Godefroid, Michael Y Levin, and David Molnar. “Automated Whitebox Fuzz Testing”. en. In: (2008).
- [GMS12] Philippa Anne Gardner, Sergio Maffei, and Gareth David Smith. “Towards a program logic for JavaScript”. In: *ACM SIGPLAN Notices* 47.1 (Jan. 2012), pp. 31–44. ISSN: 0362-1340. DOI: [10.1145/2103621.2103663](https://dl.acm.org/doi/10.1145/2103621.2103663). URL: <https://dl.acm.org/doi/10.1145/2103621.2103663> (visited on 04/21/2024).
- [GNR09] Patrice Godefroid, Aditya Nori, and Sriram Rajamani. “Compositional May-Must Program Analysis: Unleashing The Power of Alternation”. en-US. In: (Jan. 2009). URL: <https://www.microsoft.com/en-us/research/publication/compositional-may-must-program-analysis-unleashing-the-power-of-alternation/> (visited on 01/27/2023).
- [GNU24] GNU GCC developers. *Program Instrumentation Options - GNU GCC*. 2024. URL: <https://gcc.gnu.org/onlinedocs/gcc/Instrumentation-Options.html> (visited on 05/13/2024).
- [Gro23] Unsafe Code Guidelines Working Group. *Structs and Tuples - Memory Layout - Unsafe Code Guidelines*. Accessed: Nov. 16 2019. 2023. URL: <https://github.com/rust-lang/unsafe-code-guidelines/blob/50f8ff4b6892f98740de3b375e4d4bda10b9da9f/reference/src/layout/structs-and-tuples.md>.
- [Gué+23] Armaël Guéneau, Johannes Hostert, Simon Spies, Michael Sammler, Lars Birkedal, and Derek Dreyer. “Melocoton: A Program Logic for Verified Interoperability Between OCaml and C”. In: *Proceedings of the ACM on Programming Languages* 7.OOPSLA2 (Oct. 2023), 247:716–247:744. DOI: [10.1145/3622823](https://dl.acm.org/doi/10.1145/3622823). URL: <https://dl.acm.org/doi/10.1145/3622823> (visited on 12/06/2023).
- [Hal+13] Istvan Haller, Asia Slowinska, Matthias Neugschwandtner, and Herbert Bos. “Dowsing for overflows: a guided fuzzer to find buffer boundary violations”. In: *Proceedings of the 22nd USENIX conference on Security*. SEC’13. USA: USENIX Association, Aug. 2013, pp. 49–64. ISBN: 978-1-931971-03-4. (Visited on 05/13/2024).
- [HP22] Son Ho and Jonathan Protzenko. “Aeneas: Rust verification by functional translation”. In: *Proceedings of the ACM on Programming Languages* 6.ICFP (Aug. 2022), 116:711–116:741. DOI: [10.1145/3547647](https://doi.org/10.1145/3547647). URL: <https://doi.org/10.1145/3547647> (visited on 01/05/2023).
- [ILR21] Hugo Illous, Matthieu Lemerre, and Xavier Rival. “A relational shape abstract domain”. en. In: *Formal Methods in System Design* 57.3 (Sept. 2021), pp. 343–400. ISSN: 1572-8102. DOI: [10.1007/s10703-021-00366-4](https://doi.org/10.1007/s10703-021-00366-4). URL: <https://doi.org/10.1007/s10703-021-00366-4> (visited on 03/17/2023).
- [Int18] International Organization for Standardization. *ISO/IEC 9899:2018 - Information technology – Programming languages – C*. June 2018. URL: <https://www.iso.org/standard/74528.html>.

- [Jac+11] Bart Jacobs, Jan Smans, Pieter Philippaerts, Frédéric Vogels, Willem Penninckx, and Frank Piessens. “VeriFast: A Powerful, Sound, Predictable, Fast Verifier for C and Java”. en. In: *NASA Formal Methods*. Ed. by Mihaela Bobaru, Klaus Havelund, Gerard J. Holzmann, and Rajeev Joshi. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2011, pp. 41–55. ISBN: 978-3-642-20398-5. DOI: [10.1007/978-3-642-20398-5_4](https://doi.org/10.1007/978-3-642-20398-5_4).
- [JP11] Bart Jacobs and Frank Piessens. “Expressive modular fine-grained concurrency specification”. In: *Proceedings of the 38th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. POPL ’11. New York, NY, USA: Association for Computing Machinery, Jan. 2011, pp. 271–282. ISBN: 978-1-4503-0490-0. DOI: [10.1145/1926385.1926417](https://doi.org/10.1145/1926385.1926417). URL: <https://dl.acm.org/doi/10.1145/1926385.1926417> (visited on 04/22/2024).
- [JSP10] Bart Jacobs, Jan Smans, and Frank Piessens. “A Quick Tour of the VeriFast Program Verifier”. en. In: *Programming Languages and Systems*. Ed. by Kazunori Ueda. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2010, pp. 304–311. ISBN: 978-3-642-17164-2. DOI: [10.1007/978-3-642-17164-2_21](https://doi.org/10.1007/978-3-642-17164-2_21).
- [Jun+17] Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. “RustBelt: securing the foundations of the Rust programming language”. In: *Proceedings of the ACM on Programming Languages* 2.POPL (Dec. 2017), 66:1–66:34. DOI: [10.1145/3158154](https://doi.org/10.1145/3158154). URL: <https://doi.org/10.1145/3158154> (visited on 01/05/2023).
- [Jun+18] Ralf Jung, Robbert Krebbers, Jacques-Henri Jourdan, Aleš Bizjak, Lars Birkedal, and Derek Dreyer. “Iris from the ground up: A modular foundation for higher-order concurrent separation logic”. en. In: *Journal of Functional Programming* 28 (2018), e20. ISSN: 0956-7968, 1469-7653. DOI: [10.1017/S0956796818000151](https://doi.org/10.1017/S0956796818000151). URL: https://www.cambridge.org/core/product/identifier/S0956796818000151/type/journal_article (visited on 02/25/2023).
- [Jun+19] Ralf Jung, Hoang-Hai Dang, Jeehoon Kang, and Derek Dreyer. “Stacked borrows: an aliasing model for Rust”. In: *Proceedings of the ACM on Programming Languages* 4.POPL (Dec. 2019), 41:1–41:32. DOI: [10.1145/3371109](https://doi.org/10.1145/3371109). URL: <https://doi.org/10.1145/3371109> (visited on 01/05/2023).
- [Jun+20] Ralf Jung, Rodolphe Lepigre, Gaurav Parthasarathy, Marianna Rapoport, Amin Timany, Derek Dreyer, and Bart Jacobs. “The future is ours: prophecy variables in separation logic”. en. In: *Proceedings of the ACM on Programming Languages* 4.POPL (Jan. 2020), pp. 1–32. ISSN: 2475-1421. DOI: [10.1145/3371113](https://doi.org/10.1145/3371113). URL: <https://dl.acm.org/doi/10.1145/3371113> (visited on 02/23/2023).
- [Jun18] Ralf Jung. *Two Kinds of Invariants: Safety and Validity*. English. Blog. Aug. 2018. URL: <https://www.ralfj.de/blog/2018/08/22/two-kinds-of-invariants.html> (visited on 06/19/2023).
- [Jun20] Ralf Jung. “Understanding and evolving the Rust programming language”. en. Accepted: 2020-09-09T07:57:28Z. doctoralThesis. Saarländische Universitäts- und Landesbibliothek, 2020. DOI: [10.22028/D291-31946](https://publikationen.sulb.uni-saarland.de/handle/20.500.11880/29647). URL: <https://publikationen.sulb.uni-saarland.de/handle/20.500.11880/29647> (visited on 02/08/2023).
- [JV23] Ralf Jung and Neven Villani. *From Stacks to Trees: A new aliasing model for Rust*. Accessed: Nov. 16 2019. June 2023. URL: <https://www.ralfj.de/blog/2023/06/02/tree-borrows.html>.
- [JVP15] Bart Jacobs, Frédéric Vogels, and Frank Piessens. “Featherweight VeriFast”. en. In: *Logical Methods in Computer Science* Volume 11, Issue 3 (Sept. 2015), p. 1595. ISSN: 1860-5974. DOI: [10.2168/LMCS-11\(3:19\)2015](https://doi.org/10.2168/LMCS-11(3:19)2015). URL: <https://lmcs.episciences.org/1595> (visited on 07/26/2023).

- [KAG23] Nat Karmios, Sacha-Élie Ayoun, and Philippa Gardner. “Symbolic Debugging with Gillian”. In: *Proceedings of the 1st ACM International Workshop on Future Debugging Techniques*. DEBT 2023. New York, NY, USA: Association for Computing Machinery, July 2023, pp. 1–2. ISBN: 9798400702457. DOI: [10.1145/3605155.3605861](https://doi.org/10.1145/3605155.3605861). URL: <https://dl.acm.org/doi/10.1145/3605155.3605861> (visited on 10/26/2023).
- [Keu+22] Steven Keuchel, Sander Huyghebaert, Georgy Lukyanov, and Dominique Devriese. “Verified symbolic execution with Kripke specification monads (and no meta-programming)”. en. In: *Proceedings of the ACM on Programming Languages* 6.ICFP (Aug. 2022), pp. 194–224. ISSN: 2475-1421. DOI: [10.1145/3547628](https://doi.org/10.1145/3547628). URL: <https://dl.acm.org/doi/10.1145/3547628> (visited on 07/09/2023).
- [Kin76] James C. King. “Symbolic execution and program testing”. In: *Communications of the ACM* 19.7 (July 1976), pp. 385–394. ISSN: 0001-0782. DOI: [10.1145/360248.360252](https://doi.org/10.1145/360248.360252). URL: <https://dl.acm.org/doi/10.1145/360248.360252> (visited on 08/18/2023).
- [Kir+15] Florent Kirchner, Nikolai Kosmatov, Virgile Prevosto, Julien Signoles, and Boris Yakobowski. “Frama-C: A software analysis perspective”. en. In: *Formal Aspects of Computing* 27.3 (May 2015), pp. 573–609. ISSN: 1433-299X. DOI: [10.1007/s00165-014-0326-7](https://doi.org/10.1007/s00165-014-0326-7). URL: <https://doi.org/10.1007/s00165-014-0326-7> (visited on 05/13/2024).
- [Lat+23] Andrea Lattuada, Travis Hance, Chanhee Cho, Matthias Brun, Isitha Subasinghe, Yi Zhou, Jon Howell, Bryan Parno, and Chris Hawblitzel. “Verus: Verifying Rust Programs using Linear Ghost Types”. In: *Proceedings of the ACM on Programming Languages* 7.OOPSLA1 (Apr. 2023), 85:286–85:315. DOI: [10.1145/3586037](https://doi.org/10.1145/3586037). URL: <https://dl.acm.org/doi/10.1145/3586037> (visited on 05/23/2023).
- [LB08] Xavier Leroy and Sandrine Blazy. “Formal Verification of a C-like Memory Model and Its Uses for Verifying Program Transformations”. en. In: *Journal of Automated Reasoning* 41.1 (July 2008), pp. 1–31. ISSN: 1573-0670. DOI: [10.1007/s10817-008-9099-0](https://doi.org/10.1007/s10817-008-9099-0). URL: <https://doi.org/10.1007/s10817-008-9099-0> (visited on 05/13/2024).
- [LB23] Sirui Lu and Rastislav Bodík. “Grisette: Symbolic Compilation as a Functional Programming Library”. In: *Proceedings of the ACM on Programming Languages* 7.POPL (Jan. 2023), 16:455–16:487. DOI: [10.1145/3571209](https://doi.org/10.1145/3571209). URL: <https://dl.acm.org/doi/10.1145/3571209> (visited on 04/01/2024).
- [Le+22] Quang Loc Le, Azalea Raad, Jules Villard, Josh Berdine, Derek Dreyer, and Peter W. O’Hearn. “Finding real bugs in big programs with incorrectness logic”. In: *Proceedings of the ACM on Programming Languages* 6.OOPSLA1 (Apr. 2022), 81:1–81:27. DOI: [10.1145/3527325](https://doi.org/10.1145/3527325). URL: <https://doi.org/10.1145/3527325> (visited on 01/06/2023).
- [Leh+22] Nico Lehmann, Adam Geller, Niki Vazou, and Ranjit Jhala. *Flux: Liquid Types for Rust*. arXiv:2207.04034 [cs]. Nov. 2022. DOI: [10.48550/arXiv.2207.04034](https://arxiv.org/abs/2207.04034). URL: <http://arxiv.org/abs/2207.04034> (visited on 05/22/2023).
- [Lei08] K. Rustan M. Leino. “This is Boogie 2”. en-US. In: (June 2008). URL: <https://www.microsoft.com/en-us/research/publication/this-is-boogie-2-2/> (visited on 04/30/2024).
- [Ler+12] Xavier Leroy, Andrew W. Appel, Sandrine Blazy, and Gordon Stewart. “The CompCert Memory Model, Version 2”. en. Pages: 26. report. INRIA, June 2012. URL: <https://hal.inria.fr/hal-00703441> (visited on 12/23/2022).
- [Ler09a] Xavier Leroy. “A Formally Verified Compiler Back-end”. en. In: *Journal of Automated Reasoning* 43.4 (Dec. 2009), pp. 363–446. ISSN: 1573-0670. DOI: [10.1007/s10817-009-9155-4](https://doi.org/10.1007/s10817-009-9155-4). URL: <https://doi.org/10.1007/s10817-009-9155-4> (visited on 05/13/2024).

- [Ler09b] Xavier Leroy. “Formal verification of a realistic compiler”. In: *Communications of the ACM* 52.7 (July 2009), pp. 107–115. ISSN: 0001-0782. DOI: [10.1145/1538788.1538814](https://doi.org/10.1145/1538788.1538814). URL: <https://dl.acm.org/doi/10.1145/1538788.1538814> (visited on 04/05/2024).
- [Ler23] Xavier Leroy. *The CompCert C Compiler*. CompCert. Accessed: 2024-05-13. 2023. URL: <https://compcert.org/compcert-C.html>.
- [Löo+24a] Andreas Lööw, Daniele Nantes-Sobrinho, Sacha-Élie Ayoun, Caroline Cronjäger, Petar Maksimović, and Philippa Gardner. “Compositional Symbolic Execution for Correctness and Incorrectness Reasoning”. In: *38th European Conference on Object-Oriented Programming (ECOOP 2023)*. Ed. by Jonathan Aldrich and Guido Salvaneschi. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2024, 12:1–12:28. DOI: [10.4230/LIPIcs.ECOOP.2024.12](https://doi.org/10.4230/LIPIcs.ECOOP.2024.12). URL: <https://drops.dagstuhl.de/entities/document/10.4230/LIPIcs.ECOOP.2024.12>.
- [Löo+24b] Andreas Lööw, Daniele Nantes-Sobrinho, Sacha-Élie Ayoun, Petar Maksimović, and Philippa Gardner. “Matching Plans for Frame Inference in Compositional Reasoning”. In: *38th European Conference on Object-Oriented Programming (ECOOP 2023)*. Ed. by Jonathan Aldrich and Guido Salvaneschi. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2024, 32:1–32:20. DOI: [10.4230/LIPIcs.ECOOP.2024.32](https://doi.org/10.4230/LIPIcs.ECOOP.2024.32). URL: <https://drops.dagstuhl.de/entities/document/10.4230/LIPIcs.ECOOP.2024.32>.
- [Lou21] Daniël Louwrik. *A Separation Logic for Stacked Borrows*. en. Report. Apr. 2021. URL: <https://eprints.illc.uva.nl/id/eprint/1790/> (visited on 01/03/2023).
- [Mak+21] Petar Maksimović, Sacha-Élie Ayoun, José Fragoso Santos, and Philippa Gardner. “Gillian, Part II: Real-World Verification for JavaScript and C”. en. In: *Computer Aided Verification*. Ed. by Alexandra Silva and K. Rustan M. Leino. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2021, pp. 827–850. ISBN: 978-3-030-81688-9. DOI: [10.1007/978-3-030-81688-9_38](https://doi.org/10.1007/978-3-030-81688-9_38).
- [Mak+23] Petar Maksimović, Caroline Cronjäger, Andreas Lööw, Julian Sutherland, and Philippa Gardner. “Exact Separation Logic: Towards Bridging the Gap Between Verification and Bug-Finding”. en. In: *DROPS-IDN/v2/document/10.4230/LIPIcs.ECOOP.2023.19*. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2023. DOI: [10.4230/LIPIcs.ECOOP.2023.19](https://doi.org/10.4230/LIPIcs.ECOOP.2023.19). URL: <https://drops.dagstuhl.de/entities/document/10.4230/LIPIcs.ECOOP.2023.19> (visited on 03/18/2024).
- [Mat+22] Yusuke Matsushita, Xavier Denis, Jacques-Henri Jourdan, and Derek Dreyer. “RustHornBelt: a semantic foundation for functional verification of Rust programs with unsafe code”. In: *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. PLDI 2022. New York, NY, USA: Association for Computing Machinery, June 2022, pp. 841–856. ISBN: 978-1-4503-9265-5. DOI: [10.1145/3519939.3523704](https://doi.org/10.1145/3519939.3523704). URL: <https://doi.org/10.1145/3519939.3523704> (visited on 01/13/2023).
- [Men+19] Adrian D. Mensing, Hendrik van Antwerpen, Casper Bach Poulsen, and Eelco Visser. “From definitional interpreter to symbolic executor”. In: *Proceedings of the 4th ACM SIGPLAN International Workshop on Meta-Programming Techniques and Reflection*. META 2019. New York, NY, USA: Association for Computing Machinery, Oct. 2019, pp. 11–20. ISBN: 978-1-4503-6985-5. DOI: [10.1145/3358502.3361269](https://doi.org/10.1145/3358502.3361269). URL: <https://dl.acm.org/doi/10.1145/3358502.3361269> (visited on 07/09/2023).
- [Mil+97] Robin Milner, Robert Harper, David MacQueen, and Mads Tofte. *The Definition of Standard ML*. en. The MIT Press, May 1997. ISBN: 978-0-262-28700-5. DOI: [10.7551/mitpress/2319.001.0001](https://direct.mit.edu/books/book/2094/The-Definition-of-Standard-ML). URL: <https://direct.mit.edu/books/book/2094/The-Definition-of-Standard-ML> (visited on 03/09/2024).

- [Mil18] Bartosz Milewski. *Category Theory for Programmers*. Blurb, Incorporated, Sept. 2018. ISBN: 978-0-464-82508-1.
- [MK14] Nicholas D. Matsakis and Felix S. Klock. “The Rust language”. In: *ACM SIGAda Ada Letters* 34.3 (Oct. 2014), pp. 103–104. ISSN: 1094-3641. DOI: [10.1145/2692956.2663188](https://doi.org/10.1145/2692956.2663188). URL: <https://dl.acm.org/doi/10.1145/2692956.2663188> (visited on 05/23/2023).
- [MSS16a] Peter Müller, Malte Schwerhoff, and Alexander J. Summers. “Automatic Verification of Iterated Separating Conjunctions Using Symbolic Execution”. en. In: *Computer Aided Verification*. Ed. by Swarat Chaudhuri and Azadeh Farzan. Cham: Springer International Publishing, 2016, pp. 405–425. ISBN: 978-3-319-41528-4. DOI: [10.1007/978-3-319-41528-4_22](https://doi.org/10.1007/978-3-319-41528-4_22).
- [MSS16b] Peter Müller, Malte Schwerhoff, and Alexander J. Summers. “Viper: A Verification Infrastructure for Permission-Based Reasoning”. en. In: *Verification, Model Checking, and Abstract Interpretation*. Ed. by Barbara Jobstmann and K. Rustan M. Leino. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2016, pp. 41–62. ISBN: 978-3-662-49122-5. DOI: [10.1007/978-3-662-49122-5_2](https://doi.org/10.1007/978-3-662-49122-5_2).
- [MTK21] Yusuke Matsushita, Takeshi Tsukada, and Naoki Kobayashi. “RustHorn: CHC-based Verification for Rust Programs”. In: *ACM Transactions on Programming Languages and Systems* 43.4 (Oct. 2021), 15:1–15:54. ISSN: 0164-0925. DOI: [10.1145/3462205](https://doi.org/10.1145/3462205). URL: <https://doi.org/10.1145/3462205> (visited on 01/05/2023).
- [Mul+14] Dominic P. Mulligan, Scott Owens, Kathryn E. Gray, Tom Ridge, and Peter Sewell. “Lem: reusable engineering of real-world semantics”. In: *ACM SIGPLAN Notices* 49.9 (Aug. 2014), pp. 175–188. ISSN: 0362-1340. DOI: [10.1145/2692915.2628143](https://doi.org/10.1145/2692915.2628143). URL: <https://dl.acm.org/doi/10.1145/2692915.2628143> (visited on 05/01/2024).
- [Nau18] Daiva Naudziuniene. “An Infrastructure for Tractable Verification of JavaScript Programs”. PhD thesis. Imperial College London, 2018.
- [Nel+19] Luke Nelson, James Bornholt, Ronghui Gu, Andrew Baumann, Emina Torlak, and Xi Wang. “Scaling symbolic evaluation for automated verification of systems code with Serval”. In: *Proceedings of the 27th ACM Symposium on Operating Systems Principles*. SOSP ’19. New York, NY, USA: Association for Computing Machinery, Oct. 2019, pp. 225–242. ISBN: 978-1-4503-6873-5. DOI: [10.1145/3341301.3359641](https://doi.org/10.1145/3341301.3359641). URL: <https://dl.acm.org/doi/10.1145/3341301.3359641> (visited on 05/01/2024).
- [NKC08] Huu Hai Nguyen, Viktor Kuncak, and Wei-Ngan Chin. “Runtime Checking for Separation Logic”. en. In: *Verification, Model Checking, and Abstract Interpretation*. Ed. by Francesco Logozzo, Doron A. Peled, and Lenore D. Zuck. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2008, pp. 203–217. ISBN: 978-3-540-78163-9. DOI: [10.1007/978-3-540-78163-9_19](https://doi.org/10.1007/978-3-540-78163-9_19).
- [NLR22] Olivier Nicole, Matthieu Lemerre, and Xavier Rival. “Lightweight Shape Analysis Based on Physical Types”. en. In: *Verification, Model Checking, and Abstract Interpretation*. Ed. by Bernd Finkbeiner and Thomas Wies. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2022, pp. 219–241. ISBN: 978-3-030-94583-1. DOI: [10.1007/978-3-030-94583-1_11](https://doi.org/10.1007/978-3-030-94583-1_11).
- [NS07] Nicholas Nethercote and Julian Seward. “Valgrind: a framework for heavyweight dynamic binary instrumentation”. In: *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI ’07. New York, NY, USA: Association for Computing Machinery, June 2007, pp. 89–100. ISBN: 978-1-59593-633-2. DOI: [10.1145/1250734.1250746](https://doi.org/10.1145/1250734.1250746). URL: <https://dl.acm.org/doi/10.1145/1250734.1250746> (visited on 05/13/2024).
- [OHe19] Peter W. O’Hearn. “Incorrectness logic”. In: *Proceedings of the ACM on Programming Languages* 4.POPL (Dec. 2019), 10:1–10:32. DOI: [10.1145/3371078](https://doi.org/10.1145/3371078). URL: <https://doi.org/10.1145/3371078> (visited on 01/05/2023).

- [OP99] Peter W. O’Hearn and David J. Pym. “The Logic of Bunched Implications”. In: *The Bulletin of Symbolic Logic* 5.2 (1999). Publisher: [Association for Symbolic Logic, Cambridge University Press], pp. 215–244. ISSN: 1079-8986. DOI: [10.2307/421090](https://doi.org/10.2307/421090). URL: <https://www.jstor.org/stable/421090> (visited on 04/21/2024).
- [Ope20] Open JDK Team. *JDK Bug 8241805: add Math.absExact*. Mar. 2020. URL: <https://bugs.openjdk.org/browse/JDK-8241805> (visited on 01/30/2024).
- [Pan24] Srđan Panić. *srdja/Collections-C*. original-date: 2014-08-17T14:12:26Z. May 2024. URL: <https://github.com/srdja/Collections-C> (visited on 05/05/2024).
- [PB05] Matthew Parkinson and Gavin Bierman. “Separation logic and abstraction”. en. In: *ACM SIGPLAN Notices* 40.1 (Jan. 2005), pp. 247–258. ISSN: 0362-1340, 1558-1160. DOI: [10.1145/1047659.1040326](https://doi.org/10.1145/1047659.1040326). URL: <https://dl.acm.org/doi/10.1145/1047659.1040326> (visited on 07/24/2023).
- [PJP15] Willem Penninckx, Bart Jacobs, and Frank Piessens. “Sound, Modular and Compositional Verification of the Input/Output Behavior of Programs”. en. In: *Programming Languages and Systems*. Ed. by Jan Vitek. Berlin, Heidelberg: Springer, 2015, pp. 158–182. ISBN: 978-3-662-46669-8. DOI: [10.1007/978-3-662-46669-8_7](https://doi.org/10.1007/978-3-662-46669-8_7).
- [Pul+23] Christopher Pulte, Dhruv C. Makwana, Thomas Sewell, Kayvan Memarian, Peter Sewell, and Neel Krishnaswami. “CN: Verifying Systems C Code with Separation-Logic Refinement Types”. en. In: *Proceedings of the ACM on Programming Languages* 7.POPL (Jan. 2023), pp. 1–32. ISSN: 2475-1421. DOI: [10.1145/3571194](https://doi.org/10.1145/3571194). URL: <https://dl.acm.org/doi/10.1145/3571194> (visited on 08/01/2023).
- [Raa+20] Azalea Raad, Josh Berdine, Hoang-Hai Dang, Derek Dreyer, Peter O’Hearn, and Jules Villard. “Local Reasoning About the Presence of Bugs: Incorrectness Separation Logic”. en. In: *Computer Aided Verification*. Ed. by Shuvendu K. Lahiri and Chao Wang. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2020, pp. 225–252. ISBN: 978-3-030-53291-8. DOI: [10.1007/978-3-030-53291-8_14](https://doi.org/10.1007/978-3-030-53291-8_14).
- [RE15] David A. Ramos and Dawson Engler. “Under-Constrained Symbolic Execution: Correctness Checking for Real Code”. In: *24th USENIX Security Symposium (USENIX Security 15)*. Washington, D.C.: USENIX Association, Aug. 2015, pp. 49–64. ISBN: 978-1-939133-11-3. URL: <https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/ramos>.
- [Rey02] John C. Reynolds. “Separation Logic: A Logic for Shared Mutable Data Structures”. English. In: ISSN: 1043-6871. IEEE Computer Society, July 2002, pp. 55–55. ISBN: 978-0-7695-1483-3. DOI: [10.1109/LICS.2002.1029817](https://doi.org/10.1109/LICS.2002.1029817). URL: <https://www.computer.org/csdl/proceedings-article/lics/2002/14830055/120mNxYbSXN> (visited on 10/26/2023).
- [RS10] Grigore Roşu and Traian Florin Şerbănuță. “An overview of the K semantic framework”. In: *The Journal of Logic and Algebraic Programming*. Membrane computing and programming 79.6 (Aug. 2010), pp. 397–434. ISSN: 1567-8326. DOI: [10.1016/j.jlap.2010.03.012](https://doi.org/10.1016/j.jlap.2010.03.012). URL: <https://www.sciencedirect.com/science/article/pii/S1567832610000160> (visited on 05/01/2024).
- [Sam+21] Michael Sammler, Rodolphe Lepigre, Robbert Krebbers, Kayvan Memarian, Derek Dreyer, and Deepak Garg. “RefinedC: automating the foundational verification of C code with refined ownership types”. In: *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. PLDI 2021. New York, NY, USA: Association for Computing Machinery, June 2021, pp. 158–174. ISBN: 978-1-4503-8391-2. DOI: [10.1145/3453483.3454036](https://doi.org/10.1145/3453483.3454036). URL: <https://dl.acm.org/doi/10.1145/3453483.3454036> (visited on 05/23/2023).

- [Sam23] Michael Joachim Sammler. “Automated and foundational verification of low-level programs”. en. Accepted: 2024-01-05T10:33:27Z. doctoralThesis. Saarländische Universitäts- und Landesbibliothek, 2023. DOI: [10.22028/D291-41316](https://doi.org/10.22028/D291-41316). URL: <https://publikationen.sulb.uni-saarland.de/handle/20.500.11880/37089> (visited on 06/04/2024).
- [San+18] José Fragoso Santos, Petar Maksimović, Théotime Grohens, Julian Dolby, and Philippa Gardner. “Symbolic Execution for JavaScript”. In: *Proceedings of the 20th International Symposium on Principles and Practice of Declarative Programming*. PPDP ’18. New York, NY, USA: Association for Computing Machinery, Sept. 2018, pp. 1–14. ISBN: 978-1-4503-6441-6. DOI: [10.1145/3236950.3236956](https://doi.org/10.1145/3236950.3236956). URL: <https://dl.acm.org/doi/10.1145/3236950.3236956> (visited on 01/29/2024).
- [Sch16] Malte H. Schwerhoff. “Advancing Automated, Permission-Based Program Verification Using Symbolic Execution”. en. Accepted: 2017-10-13T14:23:40Z. Doctoral Thesis. ETH Zurich, 2016. DOI: [10.3929/ethz-a-010835519](https://doi.org/10.3929/ethz-a-010835519). URL: <https://www.research-collection.ethz.ch/handle/20.500.11850/127711> (visited on 09/09/2023).
- [Sew+07] Peter Sewell, Francesco Zappa Nardelli, Scott Owens, Gilles Peskine, Thomas Ridge, Susmit Sarkar, and Rok Strniša. “Ott: effective tool support for the working semanticist”. In: *ACM SIGPLAN Notices* 42.9 (Oct. 2007), pp. 1–12. ISSN: 0362-1340. DOI: [10.1145/1291220.1291155](https://doi.org/10.1145/1291220.1291155). URL: <https://dl.acm.org/doi/10.1145/1291220.1291155> (visited on 05/01/2024).
- [SJP10] Jan Smans, Bart Jacobs, and Frank Piessens. “Heap-Dependent Expressions in Separation Logic”. en. In: *Formal Techniques for Distributed Systems*. Ed. by John Hatchliff and Elena Zucca. Berlin, Heidelberg: Springer, 2010, pp. 170–185. ISBN: 978-3-642-13464-7. DOI: [10.1007/978-3-642-13464-7_14](https://doi.org/10.1007/978-3-642-13464-7_14).
- [SJP12] Jan Smans, Bart Jacobs, and Frank Piessens. “Implicit dynamic frames”. In: *ACM Transactions on Programming Languages and Systems* 34.1 (May 2012), 2:1–2:58. ISSN: 0164-0925. DOI: [10.1145/2160910.2160911](https://doi.org/10.1145/2160910.2160911). URL: <https://dl.acm.org/doi/10.1145/2160910.2160911> (visited on 08/31/2023).
- [TB13] Emina Torlak and Rastislav Bodik. “Growing solver-aided languages with rosette”. In: *Proceedings of the 2013 ACM international symposium on New ideas, new paradigms, and reflections on programming & software*. Onward! 2013. New York, NY, USA: Association for Computing Machinery, Oct. 2013, pp. 135–152. ISBN: 978-1-4503-2472-4. DOI: [10.1145/2509578.2509586](https://doi.org/10.1145/2509578.2509586). URL: <https://dl.acm.org/doi/10.1145/2509578.2509586> (visited on 05/01/2024).
- [TB14] Emina Torlak and Rastislav Bodik. “A lightweight symbolic virtual machine for solver-aided host languages”. In: *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI ’14. New York, NY, USA: Association for Computing Machinery, June 2014, pp. 530–541. ISBN: 978-1-4503-2784-8. DOI: [10.1145/2594291.2594340](https://doi.org/10.1145/2594291.2594340). URL: <https://dl.acm.org/doi/10.1145/2594291.2594340> (visited on 05/01/2024).
- [Tea23] The Kani Team. *How Open Source Projects are Using Kani to Write Better Software in Rust / AWS Open Source Blog*. en-US. Section: Best Practices. Nov. 2023. URL: <https://aws.amazon.com/blogs/opensource/how-open-source-projects-are-using-kani-to-write-better-software-in-rust/> (visited on 11/13/2023).
- [The23a] The Coq Team. *The Coq Proof Assistant*. Accessed: Nov. 16th 2023. 2023. URL: <https://coq.inria.fr/> (visited on 11/17/2023).
- [The23b] The OCaml Team. *The OCaml Manual - Ch. 12.23: Binding Operators*. <https://v2.ocaml.org/manual/bindingops.html>. Accessed: December 4th 2023. 2023.
- [The24] The Racket Team. *The Racket programming language*. 2024. URL: <https://racket-lang.org/> (visited on 05/01/2024).

- [TIR21] David Trabish, Shachar Itzhaky, and Noam Rinetzkky. “Address-Aware Query Caching for Symbolic Execution”. In: *2021 14th IEEE Conference on Software Testing, Verification and Validation (ICST)*. ISSN: 2159-4848. Apr. 2021, pp. 116–126. DOI: [10.1109/ICST49551.2021.00023](https://doi.org/10.1109/ICST49551.2021.00023). URL: <https://ieeexplore.ieee.org/document/9438562> (visited on 02/12/2024).
- [Wat18] Conrad Watt. “Mechanising and verifying the WebAssembly specification”. In: *Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs*. CPP 2018. New York, NY, USA: Association for Computing Machinery, Jan. 2018, pp. 53–65. ISBN: 978-1-4503-5586-5. DOI: [10.1145/3167082](https://doi.org/10.1145/3167082). URL: <https://dl.acm.org/doi/10.1145/3167082> (visited on 11/30/2023).
- [Wol+21] Felix A. Wolf, Linard Arquint, Martin Clochard, Wytse Oortwijn, João C. Pereira, and Peter Müller. “Gobra: Modular Specification and Verification of Go Programs”. en. In: *Computer Aided Verification*. Ed. by Alexandra Silva and K. Rustan M. Leino. Cham: Springer International Publishing, 2021, pp. 367–379. ISBN: 978-3-030-81685-8. DOI: [10.1007/978-3-030-81685-8_17](https://doi.org/10.1007/978-3-030-81685-8_17).
- [YO02] Hongseok Yang and Peter O’Hearn. “A Semantic Basis for Local Reasoning”. en. In: *Foundations of Software Science and Computation Structures*. Ed. by Mogens Nielsen and Uffe Engberg. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2002, pp. 402–416. ISBN: 978-3-540-45931-6. DOI: [10.1007/3-540-45931-6_28](https://doi.org/10.1007/3-540-45931-6_28).
- [ZDA24] Conrad Zimmerman, Jenna DiVincenzo, and Jonathan Aldrich. “Sound Gradual Verification with Symbolic Execution”. In: *Proceedings of the ACM on Programming Languages* 8.POPL (Jan. 2024), 85:2547–85:2576. DOI: [10.1145/3632927](https://doi.org/10.1145/3632927). URL: <https://dl.acm.org/doi/10.1145/3632927> (visited on 04/30/2024).

Appendix

Appendix A

Compositionality and parametricity

Lemma 4.3 (PCMs: Induced preorder).

A PCM $(A, 0_A, \bullet)$ induces a preorder \preceq on A as follows:

$$a \preceq b \iff \exists c. b = a \bullet c$$

Proof. A preorder is a reflexive and transitive relation.

To prove Reflexivity: \preceq is reflexive because $\sigma = 0 \bullet \sigma$ for any σ , and therefore $\sigma \preceq \sigma$.

To prove Transitivity: If $\sigma_1 \preceq \sigma_2$ and $\sigma_2 \preceq \sigma_3$, then $\exists \sigma_a. \sigma_2 = \sigma_1 \bullet \sigma_a$ and $\exists \sigma_b. \sigma_3 = \sigma_2 \bullet \sigma_b$, and therefore $\sigma_3 = (\sigma_1 \bullet \sigma_a) \bullet \sigma_b = \sigma_1 \bullet (\sigma_a \bullet \sigma_b)$, by associativity. Hence $\sigma_1 \preceq \sigma_3$. \square

Theorem 4.6 (Preservation of compositionality).

$$\mathbb{S} \stackrel{\mathcal{C}}{\underset{m}{\sim}} \mathbb{S} \implies \mathcal{S}_{\mathbb{S}} \stackrel{\mathcal{C}}{\underset{m}{\sim}} \mathcal{S}_{\mathbb{S}}$$

Proof. This is trivially proven by induction on the structure of the SIGIL syntax tree.

The only language constructs for which the interpretation may differ between the full and the compositional semantics is that of action call, and since $\mathbb{S} \stackrel{\mathcal{C}}{\underset{m}{\sim}} \mathbb{S}$, eval must also be m -sound wrt. eval. \square

Appendix B

Parametric Assertion Language

B.1 Correctness of the parametric producer

Theorem 5.6 (Correctness of assertion production).

Let Σ be a set of state fragments, Δ an associated set of core predicates, $\mathbb{S}.\text{produce}$ be the producer for Δ , and P be an assertion that only contains core predicates from Δ . Then:

$$\text{produce_asrt } \mathbb{S} \ \theta \ \sigma \ P = \{\sigma \cdot \sigma_P \mid \theta, \sigma_P \models P, \sigma \# \sigma_P\}$$

Proof. By induction on the structure of assertions. The base case is proven by directly applying the definition of the core predicate producer. The existential quantification case is proven by observing that

$$\begin{aligned} \sigma' &\in \bigcup_{v \in \text{Val}} \{\sigma \cdot \sigma_P \mid \theta[\mathbf{x} \leftarrow v], \sigma_P \models P, \sigma \# \sigma_P\} \\ &\iff \exists v \in \text{Val}. \theta[\mathbf{x} \leftarrow v], \sigma' \models P \wedge \exists \sigma_P. \sigma' = \sigma \cdot \sigma_P \\ &\iff \theta, \sigma' \models \exists \mathbf{x}. P \wedge \exists \sigma_P. \sigma' = \sigma \cdot \sigma_P \\ &\iff \sigma' \in \{\sigma \cdot \sigma_P \mid \theta, \sigma_P \models \exists \mathbf{x}. P, \sigma \# \sigma_P\} \end{aligned}$$

Finally, the separating conjunction case is proven using the following equality:

$$\begin{aligned} &\bigcup \{\{\sigma' \cdot \sigma_Q \mid \theta, \sigma_Q \models Q, \sigma' \# \sigma_Q\} \mid \sigma' = \sigma \cdot \sigma_P \wedge \theta, \sigma_P \models P, \sigma \# \sigma_P\} \\ &= \{(\sigma \cdot \sigma_P) \cdot \sigma_Q \mid \theta, \sigma_P \models P \wedge \theta, \sigma_Q \models Q, \sigma \# \sigma_P \# \sigma_Q\} \\ &= \{\sigma \cdot (\sigma_P \cdot \sigma_Q) \mid \theta, \sigma_P \models P \wedge \theta, \sigma_Q \models Q, \sigma \# \sigma_P \# \sigma_Q\} \\ &= \{\sigma \cdot \sigma'' \mid \theta, \sigma'' \models P * Q, \sigma \# \sigma''\} \end{aligned}$$

□

B.2 Correctness of the parametric consumer

We start by providing the definition of the parametric consumer in the form of inference rules below. Only the success cases are required for our proofs.

CONSUME-ASRT

$$\frac{\text{plan } \theta \ P = \text{Ok} : \text{mp} \quad \text{consume_mp consume } (\theta, \sigma) \ \text{mp} = \text{Ok} : (\theta', \sigma')}{\text{consume_asrt consume } \sigma \ \theta \ P = \text{Ok} : (\theta', \sigma')}$$

CONSUME-EMPTY-MP

$$\frac{}{\text{consume_mp consume } (\theta, \sigma) \ [\] = \text{Ok} : (\theta, \sigma)}$$

CONSUME-MP-FOLD

$$\frac{\text{consume_step consume } (\theta', \sigma') \ \text{step} = \text{Ok} : (\theta', \sigma') \quad \text{consume_mp consume } (\theta', \sigma') \ \text{rest} = \text{Ok} : (\theta'', \sigma'')}{\text{consume_mp consume } (\sigma, \theta) \ (\text{step} : \text{rest}) = \text{Ok} : (\theta'', \sigma')}$$

CONSUME-STEP

$$\frac{\text{step} = (\langle \delta \rangle (\vec{e}_i; \vec{e}_o), \text{learnings}) \quad \llbracket \vec{e}_i \rrbracket_{\theta} = \text{Ok} : \vec{v}_i \quad \text{consume } \sigma \ \delta \ \vec{v}_i = \text{Ok} : (\vec{v}_o, \sigma') \quad \theta_o = [\vec{O} \mapsto \vec{v}_o] \quad \text{learn_all } \theta_o \ \theta \ \text{learnings} = \text{Ok} : \theta' \quad \llbracket \vec{e}_o \rrbracket_{\theta'} = \text{Ok} : \vec{v}'_o \quad \vec{v}_o = \vec{v}'_o}{\text{consume_step consume } (\theta, \sigma) \ \text{step} = \text{Ok} : (\theta', \sigma')}$$

LEARN-ALL-FOLD

$$\frac{\mathbf{x} \notin \theta \quad \llbracket e \rrbracket_{\theta \cup \theta_o} = \text{Ok} : v \quad \theta' = \theta [\mathbf{x} \leftarrow v] \quad \text{learn_all } \theta_o \ \theta' \ \text{rest} = \text{Ok} : \theta''}{\text{learn_all } \theta_o \ \theta \ ((\mathbf{x}, e) : \text{learnings}) = \text{Ok} : \theta''}$$

LEARN-ALL-EMPTY

$$\frac{}{\text{learn_all } \theta_o \ \theta \ [\] = \text{Ok} : \theta}$$

Theorem 5.9 (Assertion consumer).

Let $\mathbb{S}.\text{consume}$ be a valid core predicate consumer according to [Definition 5.7](#). Then, $\text{consume_asrt } \mathbb{S}$ satisfies the following properties:

$$\begin{aligned} & \sigma.\text{cons}_P(\theta) \xrightarrow{\mathbb{S}} \text{Ok} : (\theta', \sigma') \\ \implies & \exists \sigma_P. \sigma = \sigma' \cdot \sigma_P \wedge \theta', \sigma_P \models P \end{aligned} \quad (\text{Consuming})$$

$$\begin{aligned} & \sigma.\text{cons}_P(\theta) \xrightarrow{\mathbb{S}} \text{Ok} : (\theta', \sigma') \\ \implies & \theta \subseteq \theta' \end{aligned} \quad (\text{Matching})$$

Proof. **Assume**

$$(H1) \text{ consume } \sigma \ \delta \ \vec{v}_i = \text{Ok} : (\vec{v}_o, \sigma') \implies$$

$$\exists \sigma_{\delta}. \sigma = \sigma' \cdot \sigma_{\delta} \wedge \sigma_{\delta} \models \langle \delta \rangle (\vec{v}_i; \vec{v}_o)$$

$$(H2) \text{ consume_asrt consume } \sigma \ \theta \ P = \text{Ok} : (\theta', \sigma')$$

To prove our three main goals, $\exists \sigma_P$.

$$(G1) \ \sigma = \sigma' \cdot \sigma_P$$

$$(G2) \ \theta, \sigma_P \models P$$

$$(G3) \ \theta \subseteq \theta'$$

From (H2), we know that

$$(H3) \text{ plan } \theta \ P = \text{Ok} : \text{mp}$$

$$(H4) \text{ consume_mp consume } (\theta, \sigma) \ \text{mp} = \text{Ok} : (\theta', \sigma')$$

The only property required of the plan function is that, in case of success (H3), it returns a matching plan that is a permutation of P put in normal form:

$$(H5) \text{ mp} = [\text{mp}_k]_{k=1}^n = [(\langle \delta^k \rangle (\vec{e}_i^k; \vec{e}_o^k), \mathbf{1s}^k)]_{k=1}^n$$

$$(H6) P \iff \exists \vec{x}. \bigotimes_{k=1}^n \langle \delta^{g(k)} \rangle (\vec{e}_i^{g(k)}; \vec{e}_o^{g(k)})$$

$$(H7) g \text{ is a permutation of } [1, n]$$

Let $\sigma^0 = \sigma, \theta^0 = \theta, \sigma' = \sigma^n, \theta' = \theta^n$ and $(\theta_{k+1}, \sigma_{k+1}) = \text{consume_step consume}(\theta^k, \sigma^k) \text{ mp}_k$.

We prove by finite induction that, $\forall k \in [0, n]. \exists \sigma_c^k. :$

$$(G4) \theta^k, \sigma_c^k \models \bigotimes_{j=1}^k \langle \delta^j \rangle (\vec{e}_i^j; \vec{e}_o^j)$$

$$(G5) \sigma^0 = \sigma^k \cdot \sigma_c^k$$

$$(G6) \theta^0 \subseteq \theta^k$$

For $k = 0$, we have $\sigma_c^0 = 0$, and the goals are trivially satisfied.

Assume that the goals hold for $k < n$, denoted $(G4)^k, (G5)^k$ and $(G6)^k$. **To prove** that they hold for $k + 1$, denoted $(G4)^{k+1}, (G5)^{k+1}$ and $(G6)^{k+1}$.

Since consumption is successful, we know that there exists $\vec{v}_i^k, \vec{v}_o^k, \vec{u}_o^k$ such that:

Ran out of place to put more superscripts, using \vec{u}_o instead of \vec{v}_o' .

$$(H8) \llbracket \vec{e}_i^k \rrbracket_{\theta^k} = 0k : \vec{v}_i^k$$

$$(H9) \text{ consume } \sigma^k \delta^k \vec{v}_i^k = 0k : (\vec{v}_o^k, \sigma^{k+1})$$

$$(H10) \text{ learn_all } [\vec{O} \mapsto \vec{v}_o^k] \theta^k \mathbf{1s}^k = 0k : \theta^{k+1}$$

$$(H11) \llbracket \vec{e}_o \rrbracket_{\theta^{k+1}} = 0k : \vec{u}_o^k$$

$$(H12) \vec{v}_o^k = \vec{u}_o^k$$

From (H9) and (H1) we have a σ_δ such that:

$$(H13) \sigma^k = \sigma^{k+1} \cdot \sigma_\delta \quad (H14) \sigma_\delta \models \langle \delta^k \rangle (\vec{v}_i^k; \vec{v}_o^k)$$

From (H10), by trivial finite induction on the length of $\mathbf{1s}^k$, and from the fact that learning terminates successfully we learn **(H15)** $\theta^k \subseteq \theta^{k+1}$

From (H11) and (H12), we get **(H16)** $\llbracket \vec{e}_o \rrbracket_{\theta^{k+1}} = 0k : \vec{v}_o^k$ and from (H15) and (H8) we get **(H17)** $\llbracket \vec{e}_i^k \rrbracket_{\theta^{k+1}} = 0k : \vec{v}_i^k$

By [Definition 5.2](#) of assertion satisfiability, using (H17), (H16) and (H14) we get **(H18)** $\theta^{k+1}, \sigma_\delta \models \langle \delta^k \rangle (\vec{v}_i^k; \vec{v}_o^k)$

We can now connect the outcomes of this consumption step to our induction hypothesis.

Let $\sigma_c^{k+1} = \sigma_c^k \cdot \sigma_\delta$.

From (H13) and $(G5)^k$, we get $(G5)^{k+1}$.

Then from (H15) and $(G6)^k$ we get the intermediate result:

$$(H19) \theta^{k+1}, \sigma^k \models \bigotimes_{j=1}^k \langle \delta^j \rangle (\vec{e}_i^j; \vec{e}_o^j)$$

which in turns, combined with (H18) and [Definition 5.2](#) of assertion satisfiability gives us the goal $(G4)^{k+1}$.

Finally, the goal $(G6)^{k+1}$ is obtained using $(G6)^k$ and (H15), concluding our induction, obtaining its three induction goals for $k = n$.

The goal (G3) is just the last induction goal $(G3)^n$. In addition, we note that the separation conjunction is commutative, and that $\theta, \sigma \models Q \implies \theta, \sigma \models \exists x. Q$. Therefore, using (H7), (H6) and our final

induction goals (G4) and (G5)ⁿ, we obtain (G2) and (G1) by setting $\sigma_P = \sigma_c^n$.

□

Theorem 5.10 (Assertion consumer completeness).

Given a complete (according to [Definition 5.8](#)) core predicate consumer $\mathbb{S}.\text{consume}$, the assertion consumer $\text{consume_asrt } \mathbb{S}$ is also complete.¹ Formally, it satisfies the following property:

$$\begin{aligned} & \theta, \sigma_P \models P \wedge \sigma \# \sigma_P \\ \implies & \exists (\sigma \cdot \sigma_P). \text{cons}_P(\theta) \xrightarrow{\mathbb{S}} \text{Ok} : (\theta, \sigma) \end{aligned} \quad (\text{Consume completeness})$$

¹ This theorem assumes that the creation a matching plan cannot fail if all the free variables of the assertions are already known, and that, in this case, the list of learnings is always empty.

Proof. This theorem assumes that creating a matching plan cannot fail if the substitution already covers all the free variables of the assertion.

Assume

$$(H1) \quad \theta, \sigma_P \models P$$

$$(H2) \quad \sigma \# \sigma_P$$

$$(H3) \quad \sigma_\delta \models \langle \delta \rangle (\vec{v}_i; \vec{v}_o) \wedge \sigma_\delta \# \sigma_f \implies (\sigma_\delta \cdot \sigma_f). \text{cons}_\delta(\vec{v}_i) \rightarrow \text{Ok} : (\vec{v}_o, \sigma)$$

To prove (G1) $\exists \theta'. (\sigma \cdot \sigma_P). \text{cons}_P(\theta) \rightarrow \text{Ok} : (\theta', \sigma)$

Since $\theta, \sigma_P \models P$, we know that $\text{fv}(P) \subseteq \text{dom}(\theta)$, and therefore **(H4)** $\text{plan } \theta P = \text{Ok} : \text{mp}$. Since planning is successful, we have the properties of the matching plan, where we assume that there is no learning to be done:

$$(H5) \quad \text{mp} = [\text{mp}_k]_{k=1}^n = [(\langle \delta^k \rangle (\vec{e}_i^k; \vec{e}_o^k), [])]_{k=1}^n$$

$$(H6) \quad P \iff \exists \vec{x}. \otimes_{k=1}^n \langle \delta^{g(k)} \rangle (\vec{e}_i^{g(k)}; \vec{e}_o^{g(k)})$$

$$(H7) \quad g \text{ is a permutation of } [1, n]$$

From all of the above, and the definition of assertion satisfiability, we can deduce that for all k in $[1, n]$, there is a σ_{δ^k} such that **(H8)**^k $\theta, \sigma_{\delta^k} \models \langle \delta^k \rangle (\vec{e}_i^k; \vec{e}_o^k)$ and **(H9)** $\sigma_{\delta_1} \cdot \dots \cdot \sigma_{\delta_n} = \sigma_P$.

For each k , **(H8)**^k and the definition of assertion satisfiability indicates that $\exists \vec{v}_i^k, \vec{v}_o^k$ such that

$$(H10)^k \quad \llbracket \vec{e}_i^k \rrbracket_\theta = \text{Ok} : \vec{v}_i^k \quad (H11)^k \quad \llbracket \vec{e}_o^k \rrbracket_\theta = \text{Ok} : \vec{v}_o^k$$

$$(H12)^k \quad \sigma_{\delta^k} \models \langle \delta^k \rangle (\vec{v}_i^k; \vec{v}_o^k)$$

Each step can be consumed successfully since: evaluation cannot fail, given **(H10)**^k and **(H11)**^k; learning cannot fail, since the learnings are always empty; consumption is complete, and is therefore guaranteed to consume σ_{δ^k} at each step, and to return the \vec{v}_o^k , which is the same list of outs as obtained by evaluating the in-expressions, according to **(H11)**^k, and are therefore guaranteed to pass the final check. At the end, all the steps are consumed, and the final state is σ . This proves our goal. □

B.3 Soundness of specification execution

During our proof, we make use of the equivalent following definition of specification validity. These definitions are equivalent to [Definition 5.11](#), but are more amenable to our proof strategy.

Lemma B.1 (Specification validity with explicit frame).

$$\begin{aligned}
\gamma \models \{ P \} e \{ \text{Ok} : r.Q_{\text{Ok}} \} \{ \text{Err} : r.Q_{\text{Err}} \} &\iff \\
(\forall \theta, \sigma, \sigma', \sigma_f, o. & \\
\theta, \sigma \models P \wedge \gamma \vdash (\sigma \cdot \sigma_f), e \Downarrow_{\theta} o : (v, \sigma') \implies & \\
(o \neq \text{Miss} \wedge \exists \sigma_Q. \sigma' = \sigma_Q \cdot \sigma_f \wedge \theta[r \leftarrow v], \sigma_Q \models Q_o) & \\
\gamma \models [P] e [\text{Ok} : r.Q_{\text{Ok}}] [\text{Err} : r.Q_{\text{Err}}] &\iff \\
(\forall \theta, \sigma', \sigma_f, o, v. & \\
\theta[r \leftarrow v], \sigma' \models Q_o \wedge \sigma' \# \sigma_f \implies & \\
(\exists \sigma. \theta, \sigma \models P \wedge \gamma \vdash (\sigma \cdot \sigma_f), e \Downarrow_{\theta} o : (v, \sigma' \cdot \sigma_f)) &
\end{aligned}$$

We give the definition of the specification execution in inference rule form

$$\begin{array}{c}
\text{EXEC-SPEC-SUCCESS} \\
\frac{
\begin{array}{l}
S = \langle P \rangle _ \langle \text{Ok} : r.Q_{\text{Ok}} \rangle \langle \text{Err} : r.Q_{\text{Err}} \rangle \\
\sigma.\text{cons}_P(\theta) \rightarrow \text{Ok} : (\theta', \sigma_f) \\
v \in \text{Val} \quad \sigma.\text{prod}_{Q_o}(\theta'[r \leftarrow v]) \rightsquigarrow \sigma'
\end{array}
}{
\sigma_f.\text{spec}_S(\theta) \rightsquigarrow o : (v, \sigma')
} \\
\\
\text{EXEC-SPEC-FAIL} \\
\frac{
\begin{array}{l}
S = \langle P \rangle _ \langle \text{Ok} : r.Q_{\text{Ok}} \rangle \langle \text{Err} : r.Q_{\text{Err}} \rangle \\
\sigma.\text{cons}_P(\theta) \rightarrow o : (\theta', \sigma') \quad o \in \{\text{Miss}, \text{Lfail}\}
\end{array}
}{
\sigma.\text{spec}_S(\theta) \rightsquigarrow o : (\text{CannotConsumePre}, \sigma')
}
\end{array}$$

Theorem 5.13 (Specification execution: soundness).

Let γ be a program and $S = \{ P \} e \{ \text{Ok} : r.Q_{\text{Ok}} \} \{ \text{Err} : r.Q_{\text{Err}} \}$ be a separation logic quadruple. If S is valid in γ and e can be executed in γ starting from a state σ under a substitution θ , then execution of S in the same state σ and substitution yields at least one result. In the absence of failures, all paths are preserved:

$$\begin{aligned}
\gamma \models S \wedge \gamma \vdash \sigma, e \Downarrow_{\theta} o : (v, \sigma') &\implies \\
(\exists o_S, v_S, \sigma_S. \sigma.\text{spec}_S(\theta) \rightsquigarrow o_S : (v_S, \sigma_S) & \\
\wedge (o_S \notin \{\text{Miss}, \text{Lfail}\} \implies (o_S = o \wedge v_S = v \wedge \sigma_S = \sigma')) & \\
(\text{OX} \text{ spec. exec. soundness}) &
\end{aligned}$$

If, conversely, $S = [P] e [\text{Ok} : r.Q_{\text{Ok}}] [\text{Err} : r.Q_{\text{Err}}]$ is an incorrectness separation logic quadruple, and then all states that are successfully reachable using the specification execution are also reachable using the expression execution:

$$\begin{aligned}
\gamma \models S \wedge \sigma.\text{spec}_S(\theta) \rightsquigarrow o : (v, \sigma') \wedge o \notin \{\text{Miss}, \text{Lfail}\} &\implies \\
\gamma \vdash \sigma, e \Downarrow_{\theta} o : (v, \sigma') & \\
(\text{UX spec. exec. soundness}) &
\end{aligned}$$

Note that the guarantees provided by UX specification execution are under the assumption that all core predicates used are strictly exact.

Proof. **Assume**

$$(H1) \quad \gamma \models S$$

$$(H2) \quad \gamma \vdash \sigma, e \Downarrow_{\theta} o : (v, \sigma')$$

Case OX soundness:

To prove (G1) $\exists o_S, v_S, \sigma_S. \sigma.\text{spec}_S(\theta) \rightsquigarrow o_S : (v_S, \sigma_S) \wedge (o_S \notin \{\text{Miss}, \text{Lfail}\}) \implies (o_S = o \wedge v_S = v \wedge \sigma_S = \sigma')$

By executing the consumer we get:

$$(H3) \quad \sigma.\text{cons}_P(\theta) \rightarrow o_c : (\theta', \sigma_f)$$

If $o_c \in \{\text{Miss}, \text{Lfail}\}$, then **EXEC-SPEC-FAIL** applies we have our goal (G1). We consider the other case where **(H4)** $o_c = \text{Ok}$.

From (H3) and (H4), and **Theorem 5.9** we have a σ_P such that:

$$(H5) \quad \theta', \sigma_P \models P \quad (H6) \quad \sigma = \sigma_P \cdot \sigma_f \quad (H7) \quad \theta \subseteq \theta'$$

From (H2) and (H7), we have

$$(H8) \quad \gamma \vdash \sigma, e \Downarrow_{\theta'} o : (v, \sigma')$$

From **Lemma B.1**, (H5), (H6), (H7), and (H8), we have $\exists \sigma_Q$.

$$(H9) \quad \sigma' = \sigma_Q \cdot \sigma_f \quad (H10) \quad \theta[r \leftarrow v], \sigma_Q \models Q_o$$

By applying the **Theorem 5.6**, (H9) and (H10), we have

$$(H11) \quad \sigma_f.\text{prod}_{Q_o}(\theta'[r \leftarrow v]) \rightsquigarrow \sigma'$$

From (H3), (H4) and (H11), we can apply **EXEC-SPEC-SUCCESS** to obtain:

$$\sigma.\text{spec}_S(\theta) \rightsquigarrow o : (v, \sigma')$$

The above is our goal (G1).

Case UX soundness:

Assume

$$(H1) \quad \gamma \models S$$

$$(H2) \quad \sigma.\text{spec}_S(\theta) \rightsquigarrow o : (v, \sigma')$$

$$(H3) \quad o \notin \{\text{Miss}, \text{Lfail}\}$$

$$(H4) \quad \theta, \sigma_P \models P \wedge \theta, \sigma'_P \models P \implies \sigma_P = \sigma'_P \text{ (strict exactness of the pre-condition)}$$

To prove (G1) $\gamma \vdash \sigma, e \Downarrow_{\theta} o : (v, \sigma')$

From (H2), (H3) and **EXEC-SPEC-SUCCESS**, we have $\exists \sigma_f$:

$$(H5) \quad \sigma.\text{cons}_P(\theta) \rightarrow \text{Ok} : (\theta', \sigma_f) \quad (H6) \quad \sigma_f.\text{prod}_{Q_o}(\theta'[r \leftarrow v]) \rightsquigarrow \sigma'$$

By validity of the producer (**Theorem 5.6**), we have $\exists \sigma_Q$:

$$(H7) \quad \theta'[r \leftarrow v], \sigma_Q \models Q_o \quad (H8) \quad \sigma' = \sigma_Q \cdot \sigma_f$$

From (H8), we get **(H9)** $\sigma_f \# \sigma_Q$, and from (H9), (H1) and **Lemma B.1**, we have $\exists \sigma_P$:

$$(H10) \quad \theta', \sigma_P \models P \quad (H11) \quad \gamma \vdash \sigma_P \cdot \sigma_f, e \Downarrow_{\theta'} o : (v, \sigma')$$

From (H5), and the validity of consumers ([Theorem 5.9](#)), we have $\exists \sigma'_P$:

$$\textbf{(H12)} \quad \sigma = \sigma'_P \cdot \sigma_f \qquad \textbf{(H13)} \quad \theta', \sigma'_P \models P$$

From (H10), (H13) and (H4) we have **(H14)** $\sigma_P = \sigma'_P$ and therefore, from (H12), (H13), and (H14) we obtain our goal (G1). \square

B.4 Soundness of the specification semantics

Theorem 5.16 (Specification semantics: soundness).

If $\models^m (\gamma, \Gamma)$, then one of the following two properties hold, depending on the mode of execution m :

$$\begin{aligned} \models^{\text{OX}} (\gamma, \Gamma) \wedge \gamma \vdash \sigma, e \Downarrow_{\theta} o : (v, \sigma') &\implies (\exists o', \sigma'', v'. \\ &\gamma, \Gamma \vdash \sigma, e \Downarrow_{\theta} o : (\sigma', v') \\ &\wedge (o' \notin \{\text{Miss}, \text{Lfail}\} \Rightarrow (o' = o \wedge \sigma'' = \sigma' \wedge v' = v))) \\ &\text{(Spec Sem. } \text{OX} \text{ soundness)} \end{aligned}$$

$$\begin{aligned} \models^{\text{UX}} (\gamma, \Gamma) \wedge \gamma, \Gamma \vdash \sigma, e \Downarrow_{\theta} o : (v, \sigma') \wedge o \notin \{\text{Miss}, \text{Lfail}\} \\ \implies \gamma \vdash \sigma, e \Downarrow_{\theta} o : (v', \sigma') \\ \text{(Spec Sem. } \text{UX} \text{ soundness)} \end{aligned}$$

Proof. The proof is performed by induction on the structure of the expression executed. We first prove the OX soundness, and then the UX soundness.

Proposition: OX soundness

Assume that $\models^{\text{OX}} (\gamma, \Gamma)$. We provide a selection of representative cases:

Case Base case: $e = e_p$:

The evaluation of pure expressions is performed identically in both the compositional and specification semantics. Therefore, the base case is trivial.

Case Inductive case: $e = \text{let } x = e_1 \text{ in } e_2$:

Assume

$$\begin{aligned} \models^{\text{OX}} (\gamma, \Gamma) \wedge \gamma \vdash \sigma, e_1 \Downarrow_{\theta} o : (v, \sigma') &\implies (\exists o', \sigma'', v'. \\ \textbf{(IH1)} \quad \gamma, \Gamma \vdash \sigma, e_1 \Downarrow_{\theta} o : (\sigma', v') \\ &\wedge (o' \notin \{\text{Miss}, \text{Lfail}\} \Rightarrow (o' = o \wedge \sigma'' = \sigma' \wedge v' = v))) \end{aligned}$$

$$\begin{aligned} \models^{\text{OX}} (\gamma, \Gamma) \wedge \gamma \vdash \sigma, e_2 \Downarrow_{\theta} o : (v, \sigma') &\implies (\exists o', \sigma'', v'. \\ \textbf{(IH2)} \quad \gamma, \Gamma \vdash \sigma, e_2 \Downarrow_{\theta} o : (\sigma', v') \\ &\wedge (o' \notin \{\text{Miss}, \text{Lfail}\} \Rightarrow (o' = o \wedge \sigma'' = \sigma' \wedge v' = v))) \end{aligned}$$

$$\textbf{(H1)} \quad \gamma \vdash \sigma, e \Downarrow_{\theta} o : (v, \sigma')$$

There are several cases. If the first evaluation fails with **Err**, **Miss** or **Lfail**, then the whole evaluation fails with the same outcome, and the result is trivially obtained from the inductive hypothesis (IH1).

If the first evaluation is successful, then the second evaluation is performed:

$$(H2) \quad \gamma \vdash \sigma, e \Downarrow_{\theta} \text{Ok} : (v_x, \sigma'')$$

$$(H3) \quad \gamma \vdash \sigma'' \Downarrow_{\theta[x \leftarrow v_x]} o : (v, \sigma')$$

From (IH1) and (IH2), we have that: $\exists o', \sigma_f, v'$

$$(H4) \quad \gamma, \Gamma \vdash \sigma, e_1 \Downarrow_{\theta} \text{Ok} : (v_x, \sigma'')$$

$$(H5) \quad \gamma, \Gamma \vdash \sigma'' \Downarrow_{\theta[x \leftarrow v_x]} o' : (v', \sigma_f)$$

$$(H6) \quad o' \notin \{\text{Miss}, \text{Lfail}\} \implies o' = o \wedge \sigma_f = \sigma' \wedge v' = v$$

If $o' \in \{\text{Miss}, \text{Lfail}\}$, then the above is our goal. Otherwise, assume

$$(H7) \quad o' \notin \{\text{Miss}, \text{Lfail}\}$$

From the inductive hypothesis (IH2), (H6) and (H7) we obtain our goal.

Case Inductive case: $e = f(\vec{e})$:

First, it can be proven by induction on the size of \vec{e} that evaluating all of the arguments preserve the soundness. The argument for the inductive case is the same as for the sequential composition case.

Then, for the function call itself, there are two arguments to consider. If the function does not have a specification in Γ , then the evaluation is the same in both semantics. Otherwise, the evaluation is performed by executing the specification, for which we have the soundness from [Theorem 5.13](#).

Proposition: UX soundness

The soundness of the UX semantics is also proven by induction on the structure of the expression, and the proof works like the OX case, so we omit it here.

□

Appendix C

Symbolic Execution

C.1 Monad laws for the symbolic execution monad

We show that the our symbolic execution monad indeed satisfies the monad laws. We first exhaustively define each of its components, which we did not do explicitly in the main text to avoid cluttering the exposition with category theory.

First, it is to be noted that the endofunctor considered is not strictly the one that associates abstractions to sets of symbolic branches, but rather the one that associates abstractions to sets of *feasible* symbolic branches under an approximate solver SAT_m .

Definition C.1 (Symex monad).

Given an approximate solver SAT_m for a mode $m \in \{\text{OX}, \text{UX}, \text{EX}\}$, we define the symbolic execution monad as the triple $(\mathbf{M}, \text{return}, \text{bind})$ where:

$$\begin{aligned} \mathbf{M} &= \overline{A}^* \rightarrow \mathcal{P}(\langle \overline{A}^* \mid \Pi_{\text{SAT}_m} \rangle) \\ \text{return } \overline{a}^* &= \{ \langle \overline{a}^* \mid \text{true} \rangle \} \\ \text{bind } r \ \overline{f} &= \left\{ \langle \overline{b}^* \mid \pi \wedge \pi' \rangle \mid \langle \overline{a}^* \mid \pi \rangle \in r \wedge \langle \overline{b}^* \mid \pi' \rangle \in \overline{f}(\overline{a}^*) \wedge \text{SAT}_m(\pi \wedge \pi') \right\} \end{aligned}$$

and Π_{SAT_m} is the set of feasible branch according to the solver SAT_m .

Lemma C.2 (Symbolic execution monad: Monad Laws).

The symbolic execution induced by the bind operator defined in [Definition C.1](#) satisfies the monad laws.

Proof. **Proposition: Left identity**

The first property that must be proven is that the monad unit is a left identity for the bind operator, that is, for all symbolic abstraction \overline{a}^* and symbolic process \overline{f} :

$$(\mathbf{G1}) \text{ bind } (\text{return } \overline{a}^*) \ \overline{f} = \overline{f}(\overline{a}^*)$$

We have:

$$\begin{aligned} & \text{bind } (\text{return } \overline{a}^*) \ \overline{f} \\ &= \left\{ \langle \overline{b}^* \mid \pi \wedge \pi' \rangle \mid \langle \overline{a}^* \mid \pi \rangle \in \text{return } \overline{a}^* \wedge \langle \overline{b}^* \mid \pi' \rangle \in \overline{f}(\overline{a}^*) \wedge \text{SAT}_m(\pi \wedge \pi') \right\} \\ &= \left\{ \langle \overline{b}^* \mid \text{true} \wedge \pi' \rangle \mid \langle \overline{b}^* \mid \pi' \rangle \in \overline{f}(\overline{a}^*) \wedge \text{SAT}_m(\pi \wedge \pi') \right\} \\ &= \left\{ \langle \overline{b}^* \mid \pi' \rangle \mid \langle \overline{b}^* \mid \pi' \rangle \in \overline{f}(\overline{a}^*) \wedge \text{SAT}_m(\pi') \right\} \\ &= \overline{f}(\overline{a}^*) \end{aligned}$$

The last line holds because all outcomes of \overline{f} must be feasible according to the solver SAT_m .

Proposition: Right identity

The second law is right-identity:

$$(G2) \text{ bind } r \text{ return} = r$$

We have:

$$\begin{aligned} & \text{bind } r \text{ return} \\ &= \left\{ \langle \bar{b}^* \mid \pi \wedge \pi' \rangle \mid \langle \bar{x}^* \mid \pi \rangle \in r \wedge \langle \bar{b}^* \mid \pi' \rangle \in \text{return}(\bar{x}^*) \wedge \text{SAT}_m(\pi \wedge \pi') \right\} \\ &= \left\{ \langle \bar{x}^* \mid \pi \wedge \text{true} \rangle \mid \langle \bar{x}^* \mid \pi \rangle \in r \wedge \text{SAT}_m(\pi \wedge \text{true}) \right\} \\ &= \left\{ \langle \bar{x}^* \mid \pi \rangle \mid \langle \bar{x}^* \mid \pi \rangle \in r \wedge \text{SAT}_m(\pi) \right\} \\ &= r \end{aligned}$$

The last line holds because all outcomes of r must be feasible according to the solver SAT_m .

Proposition: Associativity

The third law is associativity of the composition of symbolic processes:

$$(G3) \text{ bind } (\text{bind } r \bar{f}) \bar{g} = \text{bind } r (\lambda \bar{a}^*. \text{bind } (\bar{f}(\bar{a}^*)) \bar{g})$$

We have:

$$\begin{aligned} & \text{bind } r (\lambda \bar{a}^*. \text{bind } (\bar{f} \bar{a}^*) \bar{g}) \\ &= \left\{ \langle \bar{c}^* \mid \pi \wedge \pi' \rangle \mid \langle \bar{a}^* \mid \pi \rangle \in r^* \wedge \langle \bar{c}^* \mid \pi' \rangle \in \text{bind } (\bar{f} \bar{a}^*) \bar{g} \wedge \text{SAT}_m(\pi \wedge \pi') \right\} \\ &= \left\{ \langle \bar{c}^* \mid \pi \wedge \pi' \wedge \pi'' \rangle \mid \langle \bar{a}^* \mid \pi \rangle \wedge \langle \bar{c}^* \mid \pi'' \rangle \in \bar{g} \bar{b}^* \wedge \langle \bar{b}^* \mid \pi' \rangle \in \bar{f} \bar{a}^* \wedge \text{SAT}_m(\pi' \wedge \pi'') \wedge \text{SAT}_m(\pi \wedge (\pi'' \wedge \pi')) \right\} \\ &= \left\{ \langle \bar{c}^* \mid \pi \wedge \pi' \wedge \pi'' \rangle \mid \langle \bar{a}^* \mid \pi \rangle \wedge \langle \bar{c}^* \mid \pi'' \rangle \in \bar{g} \bar{b}^* \wedge \langle \bar{b}^* \mid \pi' \rangle \in \bar{f} \bar{a}^* \wedge \text{SAT}_m(\pi \wedge \pi') \wedge \text{SAT}_m((\pi \wedge \pi') \wedge \pi'') \right\} \\ &= \left\{ \langle \bar{c}^* \mid \pi'' \wedge \pi''' \rangle \mid \langle \bar{c}^* \mid \pi'' \rangle \in \bar{g} \bar{b}^* \wedge \right. \\ & \quad \left. \langle \bar{b}^* \mid \pi' \rangle \in \left\{ \langle \bar{b}^* \mid \pi \wedge \pi' \rangle \mid \langle \bar{a}^* \mid \pi \rangle \in r \wedge \langle \bar{b}^* \mid \pi' \rangle \in \bar{f} \bar{a}^* \wedge \text{SAT}_m(\pi \wedge \pi') \right\} \right. \\ & \quad \left. \wedge \text{SAT}_m(\pi'' \wedge \pi''') \right\} \\ &= \left\{ \langle \bar{c}^* \mid \pi'' \wedge \pi''' \rangle \mid \langle \bar{c}^* \mid \pi'' \rangle \in \bar{g} \bar{b}^* \wedge \langle \bar{b}^* \mid \pi' \rangle \in \text{bind } r (\bar{f} \bar{a}^*) \wedge \text{SAT}_m(\pi'' \wedge \pi''') \right\} \\ &= \text{bind } (\text{bind } r \bar{f}) \bar{g} \end{aligned}$$

□

Lemma C.3 (Symbolic execution monad: Produce preservation).

Product is preserved by the symbolic execution monad.

C.2 Composition of symbolic processes

Theorem 6.16 (Process composition: soundness preservation).

$$\bar{f} \stackrel{S}{\sim}_m f \wedge \bar{g} \stackrel{S}{\sim}_m g \implies \bar{f} \gg \bar{g} \stackrel{S}{\sim}_m f \gg g$$

Proof. Case OX soundness preservation:

$$\begin{aligned} (H1) \quad & \forall a, b, \bar{a}^* \varepsilon. f(a) \rightsquigarrow b \wedge \varepsilon, a \models \bar{a}^* \\ & \implies \exists \bar{b}^*, \pi, \varepsilon' \geq \varepsilon. \bar{f}(\bar{a}^*) \rightsquigarrow \langle \bar{b}^* \mid \pi \rangle \wedge \varepsilon', b \models \langle \bar{b}^* \mid \pi \rangle \end{aligned}$$

$$\begin{aligned} (H2) \quad & \forall b, c, \bar{b}^*, \varepsilon'. g(b) \rightsquigarrow c \wedge \varepsilon', b \models \bar{b}^* \\ & \implies \exists \bar{c}^*, \pi', \varepsilon'' \geq \varepsilon'. \bar{g}(\bar{b}^*) \rightsquigarrow \langle \bar{c}^* \mid \pi' \rangle \wedge \varepsilon'', c \models \langle \bar{c}^* \mid \pi' \rangle \end{aligned}$$

$$\begin{array}{c}
f(a) \rightsquigarrow b \\
g(b) \rightsquigarrow c \\
\hline
(H3) \quad h(a) \rightsquigarrow c
\end{array}
\quad
\begin{array}{c}
\bar{f}(\bar{a}^*) \rightsquigarrow \langle \bar{b}^* \mid \pi \rangle \\
\bar{g}(\bar{b}^*) \rightsquigarrow \langle \bar{c}^* \mid \pi' \rangle \\
\text{SAT}_{\text{ox}}(\pi \wedge \pi') \\
\hline
(H4) \quad \bar{h}(\bar{a}^*) \rightsquigarrow \langle \bar{c}^* \mid \pi \wedge \pi' \rangle
\end{array}$$

Let $a \in A, c \in C, \varepsilon \in \mathcal{I}$ such that

$$(H5) \quad h(a) \rightsquigarrow c \quad (H6) \quad \varepsilon, a \models \bar{a}^*$$

To prove (G1) $\exists \bar{c}^*, \pi'', \varepsilon'' \geq \varepsilon. \bar{h}(\bar{a}^*) \rightsquigarrow \langle \bar{c}^* \mid \pi'' \rangle \wedge \varepsilon'', c \models \langle \bar{c}^* \mid \pi'' \rangle$

From (H3) and (H5) we get that $\exists b$.

$$(H7) \quad f(a) \rightsquigarrow b \quad (H8) \quad g(b) \rightsquigarrow c$$

From (H4) we have

$$(H9) \quad \forall \bar{b}^*. \langle \bar{b}^* \mid \pi \rangle \in \bar{f}(\bar{a}^*) \implies \bar{g}(\bar{b}^*) \rightarrow -$$

From (H1), (H6) and (H7), $\exists \bar{b}^*, \pi, \varepsilon'$.

$$(H10) \quad \varepsilon' \geq \varepsilon \quad (H11) \quad \bar{f}(\bar{a}^*) \rightsquigarrow \langle \bar{b}^* \mid \pi \rangle \quad (H12) \quad \varepsilon, b \models \langle \bar{b}^* \mid \pi \rangle$$

Then from (H2), (H8), (H12) and (H9), $\exists \bar{c}^*, \pi', \varepsilon''$.

$$(H13) \quad \varepsilon'' \geq \varepsilon' \quad (H14) \quad \bar{g}(\bar{b}^*) \rightsquigarrow \langle \bar{c}^* \mid \pi' \rangle$$

$$(H15) \quad \varepsilon, c \models \langle \bar{c}^* \mid \pi' \rangle$$

Moreover, from (H10), (H13), (H12) and (H15), we have that $\pi(\varepsilon'') = \mathbf{true}$ and $\pi'(\varepsilon'') = \mathbf{true}$, and therefore:

$$(H16) \quad (\pi \wedge \pi')(\varepsilon'') \stackrel{\text{def}}{=} \pi(\varepsilon'') \wedge \pi'(\varepsilon'') = \mathbf{true}$$

Additionally, from (H16) and [Definition 6.10](#) we get

$$(H17) \quad \text{SAT}_{\text{ox}}(\pi \wedge \pi')$$

From (H4), (H11), (H14) and (H17) we obtain

$$(H18) \quad \bar{h}(\bar{a}^*) \rightsquigarrow \langle \bar{c}^* \mid \pi \wedge \pi' \rangle$$

Finally, from (H15), (H16) and (H18) we obtain (G1) □

Case UX soundness preservation:

$$\begin{array}{l}
(H1) \quad \forall \bar{a}^*, \bar{b}^*, \pi. \bar{f}(\bar{a}^*) \rightsquigarrow \langle \bar{b}^* \mid \pi \rangle \implies (\text{SAT}(\pi) \wedge \\
\quad \forall \varepsilon. \pi(\varepsilon) = \mathbf{true} \Rightarrow (\exists b. \varepsilon, b \models \bar{b}^* \wedge \\
\quad \forall b. \varepsilon, b \models \bar{b}^* \Rightarrow (\exists a. \varepsilon, a \models \bar{a}^* \wedge f(a) \rightsquigarrow b)))
\end{array}$$

$$\begin{array}{l}
(H2) \quad \forall \bar{b}^*, \bar{c}^*, \pi. \bar{g}(\bar{b}^*) \rightsquigarrow \langle \bar{c}^* \mid \pi \rangle \implies (\text{SAT}(\pi) \wedge \\
\quad \forall \varepsilon. \pi(\varepsilon) = \mathbf{true} \Rightarrow (\exists c. \varepsilon, c \models \bar{c}^* \wedge \\
\quad \forall c. \varepsilon, c \models \bar{c}^* \Rightarrow (\exists b. \varepsilon, b \models \bar{b}^* \wedge g(b) \rightsquigarrow c)))
\end{array}$$

$$\begin{array}{c}
f(a) \rightsquigarrow b \\
g(b) \rightsquigarrow c \\
\hline
(H3) \quad h(a) \rightsquigarrow c
\end{array}
\quad
\begin{array}{c}
\bar{f}(\bar{a}^*) \rightsquigarrow \langle \bar{b}^* \mid \pi \rangle \\
\bar{g}(\bar{b}^*) \rightsquigarrow \langle \bar{c}^* \mid \pi' \rangle \\
\text{SAT}_{\text{ux}}(\pi \wedge \pi') \\
\hline
(H4) \quad \bar{h}(\bar{a}^*) \rightsquigarrow \langle \bar{c}^* \mid \pi \wedge \pi' \rangle
\end{array}$$

Let $\bar{a}^*, \bar{c}^*, \pi''$ such that

$$(H5) \quad \bar{h}(\bar{a}^*) \rightsquigarrow \langle \bar{c}^* \mid \pi'' \rangle$$

To prove

$$(G1) \quad \forall \varepsilon. \pi(\varepsilon) = \text{true} \Rightarrow (\exists c. \varepsilon, c \models \bar{c}^* \wedge \\ \forall c. \varepsilon, c \models \bar{c}^* \Rightarrow (\exists a. \varepsilon, a \models \bar{a}^* \wedge h(a) \rightsquigarrow b))$$

$$(G2) \quad \text{SAT}(\pi'')$$

Before focusing on either goal, we start by establishing a few facts we can learn from (H4) and (H5): $\exists \bar{b}^*, \pi, \pi'$

$$(H6) \quad \bar{f}(\bar{a}^*) \rightsquigarrow \langle \bar{b}^* \mid \pi \rangle \quad (H7) \quad \bar{g}(\bar{b}^*) \rightsquigarrow \langle \bar{c}^* \mid \pi' \rangle$$

$$(H8) \quad \pi'' = \pi \wedge \pi' \quad (H9) \quad \text{SAT}_{\text{UX}}(\pi \wedge \pi')$$

From (H9) and the definition of approximate UX solvers, we have

$$(H10) \quad \text{SAT}(\pi') \quad (H11) \quad \text{SAT}(\pi)$$

(H8) and (H9) immediately gives (G2). Let us focus on proving (G1).

Let ε such that (H12) $\pi(\varepsilon'') = \text{true}$. Our new goals are:

$$(G3) \quad \exists c. \varepsilon, c \models \bar{c}^*$$

$$(G4) \quad \forall c. \varepsilon, c \models \bar{c}^* \Rightarrow (\exists a. \varepsilon, a \models \bar{a}^* \wedge h(a) \rightsquigarrow b)$$

Goal (G3) is immediate from (H2), (H10) and (H7), so we focus on (G4).

Let c such that (H13) $\varepsilon, c \models \bar{c}^*$.

From (H2), (H10) and (H13), we have $\exists b$.

$$(H14) \quad \varepsilon, b \models \bar{b}^* \quad (H15) \quad g(b) \rightsquigarrow c$$

From (H1), (H6), (H11) and (H14) we also have $\exists a$.

$$(H16) \quad \varepsilon, a \models \bar{a}^* \quad (H17) \quad f(a) \rightsquigarrow b$$

Together, (H15) and (H17) entail (H18) $h(a) \rightsquigarrow c$, which, together with (H16) forms (G4). \square

Theorem 6.18 (Conditional branching: soundness preservation).

Let $f, g : A \rightarrow \mathcal{P}(C)$ and be non-deterministic processes and h be defined as:

let $h(b, a) = \text{if } b \text{ then } f \text{ a else } g \text{ a}$

Furthermore, let \bar{A}^* and \bar{C}^* be sets of symbolic abstractions for A and C . Let $\bar{f}, \bar{g} : \bar{A}^* \rightarrow \mathcal{P}(\langle \bar{C}^* \mid \Pi \rangle)$ be m -sound symbolic processes with respect to f and g . Finally, let \bar{h} be the symbolic process defined as:

let $\bar{h}(\pi, \bar{a}^*) = \text{branch } \pi \ (\bar{f} \ \bar{a}^*) \ (\bar{g} \ \bar{a}^*)$

Then, if the branch operator makes use of an m -approximate solver, then \bar{h}^* is an m -sound symbolic process with respect to h .

Proof. **Case OX soundness preservation:**

$$(H1) \quad \forall a, c, \bar{a}^*, \varepsilon. f(a) \rightsquigarrow c \wedge \varepsilon, a \models \bar{a}^* \\ \implies \exists \bar{c}^*, \pi', \varepsilon' \geq \varepsilon. \bar{f}(\bar{a}^*) \rightsquigarrow \langle \bar{c}^* \mid \pi' \rangle \wedge \varepsilon', c \models \langle \bar{c}^* \mid \pi' \rangle$$

$$(H2) \quad \forall a, c, \bar{a}^*, \varepsilon. g(a) \rightsquigarrow c \wedge \varepsilon, a \models \bar{a}^* \\ \implies \exists \bar{c}^*, \pi', \varepsilon' \geq \varepsilon. \bar{g}(\bar{a}^*) \rightsquigarrow \langle \bar{c}^* \mid \pi' \rangle \wedge \varepsilon', c \models \langle \bar{c}^* \mid \pi' \rangle$$

$$(H3) \quad \frac{f(a) \rightsquigarrow c}{h(\mathbf{true}, a) \rightsquigarrow c} \quad \frac{g(a) \rightsquigarrow c}{h(\mathbf{false}, a) \rightsquigarrow c}$$

$$(H4) \quad \frac{\bar{f}(\bar{a}^*) \rightsquigarrow \langle \bar{c}^* \mid \pi' \rangle \quad \text{SAT}_{\text{ox}}(\pi \wedge \pi')}{\bar{h}(\pi, \bar{a}^*) \rightsquigarrow \langle \bar{c}^* \mid \pi \wedge \pi' \rangle} \quad \frac{\bar{g}(\bar{a}^*) \rightsquigarrow \langle \bar{c}^* \mid \pi' \rangle \quad \text{SAT}_{\text{ox}}(\neg \pi \wedge \pi')}{\bar{h}(\pi, \bar{a}^*) \rightsquigarrow \langle \bar{c}^* \mid \neg \pi \wedge \pi' \rangle}$$

Let $a, b, c, \bar{a}^*, \varepsilon$ such that:

$$(H5) \quad h(b, a) \rightsquigarrow c \quad (H6) \quad \varepsilon, a \models \bar{a}^* \quad (H7) \quad \pi(\varepsilon) = b$$

To prove (G1) $\exists \bar{c}^*, \pi_f, \varepsilon' \geq \varepsilon. \bar{h}(\pi, \bar{a}^*) \rightsquigarrow \langle \bar{c}^* \mid \pi_f \rangle \wedge \varepsilon, c \models \langle \bar{c}^* \mid \pi_f \rangle$

There are two cases to consider: either b is **true** or it is **false**. We only consider the **true** case, the **false** one being analogous.

Assume (H8) $b = \mathbf{true}$

From (H3), (H5) and (H8), we have that necessarily:

$$(H9) \quad f(a) \rightsquigarrow c$$

Using (H8), (H6) and (H7) we can apply (H1) and obtain $\exists \pi', \bar{c}^*, \varepsilon'$

$$(H10) \quad \varepsilon' \geq \varepsilon \quad (H11) \quad \bar{f}(\bar{a}^*) \rightsquigarrow \langle \bar{c}^* \mid \pi' \rangle \quad (H12) \quad \varepsilon', c \models \langle \bar{c}^* \mid \pi' \rangle$$

From (H12) and [Definition 6.11](#), we know that $\pi'(\varepsilon') = \mathbf{true}$, which combined with (H10), (H7) and (H8) gives us **(H13)** $(\pi \wedge \pi')(\varepsilon') = \mathbf{true}$, which naturally implies $\text{SAT}(\pi \wedge \pi')$, and therefore according to [Definition 6.10](#) **(H14)** $\text{SAT}_{\text{ox}}(\pi \wedge \pi')$.

Additionally, from (H12), (H13) and the definition of branch satisfiability, we also have that **(H15)** $\varepsilon', c \models \langle \bar{c}^* \mid \pi \wedge \pi' \rangle$

From (H14) and (H11) we can apply (H4) and obtain **(H16)** $\bar{h}(\pi, \bar{a}^*) \rightsquigarrow \langle \bar{c}^* \mid \pi \wedge \pi' \rangle$

Together, (H15) and (H16) form our goal (G1).

Case UX soundness preservation:

$$(H1) \quad \forall \bar{a}^*, \bar{c}^*, \pi. \bar{f}(\bar{a}^*) \rightsquigarrow \langle \bar{c}^* \mid \pi \rangle \implies (\text{SAT}(\pi) \wedge \\ \forall \varepsilon. \pi(\varepsilon) = \mathbf{true} \Rightarrow (\exists c. \varepsilon, c \models \bar{c}^* \wedge \\ \forall c. \varepsilon, c \models \bar{c}^* \Rightarrow (\exists a. \varepsilon, a \models \bar{a}^* \wedge f(a) \rightsquigarrow c)))$$

$$(H2) \quad \forall \bar{a}^*, \bar{c}^*, \pi. \bar{g}(\bar{a}^*) \rightsquigarrow \langle \bar{c}^* \mid \pi \rangle \implies (\text{SAT}(\pi) \wedge \\ \forall \varepsilon. \pi(\varepsilon) = \mathbf{true} \Rightarrow (\exists c. \varepsilon, c \models \bar{c}^* \wedge \\ \forall c. \varepsilon, c \models \bar{c}^* \Rightarrow (\exists a. \varepsilon, a \models \bar{a}^* \wedge g(a) \rightsquigarrow c)))$$

$$(H3) \quad \frac{f(a) \rightsquigarrow c}{h(\mathbf{true}, a) \rightsquigarrow c} \quad \frac{g(a) \rightsquigarrow c}{h(\mathbf{false}, a) \rightsquigarrow c}$$

$$(H4) \quad \frac{\begin{array}{c} \bar{f}(\bar{a}^*) \rightsquigarrow \langle \bar{c}^* \mid \pi' \rangle \\ \text{SAT}_{\text{UX}}(\pi \wedge \pi') \end{array}}{\bar{h}(\pi, \bar{a}^*) \rightsquigarrow \langle \bar{c}^* \mid \pi \wedge \pi' \rangle} \quad \frac{\begin{array}{c} \bar{g}(\bar{a}^*) \rightsquigarrow \langle \bar{c}^* \mid \pi' \rangle \\ \text{SAT}_{\text{UX}}(\neg \pi \wedge \pi') \end{array}}{\bar{h}(\pi, \bar{a}^*) \rightsquigarrow \langle \bar{c}^* \mid \neg \pi \wedge \pi' \rangle}$$

Let $\bar{a}^*, \pi, \bar{c}^*, \pi_f$ such that

$$(H5) \quad \bar{h}(\pi, \bar{a}^*) \rightsquigarrow \langle \bar{c}^* \mid \pi_f \rangle$$

To prove

$$(G1) \quad \text{SAT}(\pi_f)$$

$$(G2) \quad \forall \varepsilon. \pi_f(\varepsilon) = \text{true} \Rightarrow (\exists c. \varepsilon, c \models \bar{c}^* \wedge \forall c. \varepsilon, c \models \bar{c}^* \Rightarrow (\exists a, b. \varepsilon, a \models \bar{a}^* \wedge \pi(\varepsilon) = b \wedge h(b, a) \rightsquigarrow c)))$$

From (H5), and by inversions on the rules given in (H4), there are two ways the result could have been obtained. We consider the case where the “then” branch was taken, the other being analogous.

Case then branch taken:

$$(H6) \quad \bar{f}(\bar{a}^*) \rightsquigarrow \langle \bar{c}^* \mid \pi' \rangle \quad (H7) \quad \text{SAT}_{\text{UX}}(\pi \wedge \pi')$$

The goal (G1) is immediately achieved by (H7) and the definition of approximate UX solvers. In addition, we also get:

$$(H8) \quad \text{SAT}(\pi') \quad (H9) \quad \text{SAT}(\pi)$$

We now focus on (G2). Let ε such that (H10) $\pi_f(\varepsilon) = \text{true}$. This implies that

$$(H11) \quad \pi(\varepsilon) = \text{true} \quad (H12) \quad \pi'(\varepsilon) = \text{true}$$

From (H1), (H6) and (H12), we have $\exists c$.

$$(H13) \quad \varepsilon, c \models \bar{c}^*$$

We take such a c . From (H1), (H6), (H12) and (H13), we also have $\exists a$.

$$(H14) \quad \varepsilon, a \models \bar{a}^* \quad (H15) \quad f(a) \rightsquigarrow c$$

from (H15) and (H11), we get that

$$(H16) \quad h(b, a) \rightsquigarrow c$$

which concludes the proof of (G2).

Case else branch taken: Analogous to the previous case. □

Theorem 6.20 (Semantics: soundness preservation).

If the symbolic state model $\bar{\mathbb{S}}$ is m -sound with respect to the concrete compositional state model \mathbb{S} , then the induced symbolic semantics $\bar{\mathcal{S}}_{\bar{\mathbb{S}}}$, using an m -approximate solver, is m -sound with respect to the concrete induced semantics $\mathcal{S}_{\mathbb{S}}$:

$$\bar{\mathbb{S}} \stackrel{\mathcal{S}}{\sim}_m \mathbb{S} \implies \bar{\mathcal{S}}_{\bar{\mathbb{S}}} \stackrel{\mathcal{S}}{\sim}_m \mathcal{S}_{\mathbb{S}}$$

Proof. The proof comes naturally from putting the symbolic and the concrete eval function next to each other, and applying [Theorems 6.16](#) and [6.18](#) and whenever processes are composed sequentially or through conditional branching. □

Appendix D

Analyses

D.1 Compositional verification

Theorem 7.1 (Compositional verification: soundness).

Let $d = f(\vec{x}) \{e\}$ be a function definition, $s = \{ P \} f(\vec{x}) \{ o : r. Q \}$ be an OX function specification for f , and $\models^I (\gamma, \Gamma)$ be an OX-valid environment, where $f \notin \text{dom}(\gamma)$. Let $\gamma' = \gamma[f \leftarrow d]$ and $\Gamma' = \Gamma[f \leftarrow s]$. If $\text{verify } \Gamma' \gamma' f s = \text{true}$, then $\models^I (\gamma', \Gamma')$

Proof. We first prove the soundness of a concrete version of the verification procedure. This concrete version is then lifted to the symbolic world, where it satisfies the symbolic soundness property. Putting the two results together, we obtain the soundness of the symbolic verification procedure.

Proposition: Concrete soundness We define the concrete verification procedure as follows:

```
let exec_verify  $\Gamma \gamma f =$ 
  let  $f(\vec{x}) \{e\} = \gamma(f)$  in
  let  $\{ P \} f(\vec{x}) \{ 0k : r. Q_{0k} \} \{ \text{Err} : r. Q_{\text{Err}} \} = \Gamma(f)$  in
  let  $\vec{y} = \text{fv}(P)$  in
  let*  $\vec{v}_x, \vec{v}_y = \text{nondet } ()$  in
  let  $\theta = [\vec{x} \mapsto \vec{v}_x, \vec{y} \mapsto \vec{v}_y]$  in
  let*  $\sigma = \text{produce\_asrt } \theta \ 0 \ P$  in
  let*  $(\underline{o} : \vec{v}, \sigma') = \text{eval } \Gamma \gamma \theta \sigma \ e$  in
  let*  $\theta' = \theta[r \leftarrow \vec{v}]$  in
  consume_asrt  $\theta' \sigma' Q_{\underline{o}}$ 
```

```
let verify_concrete  $\Gamma \gamma f =$ 
   $\forall (o_l : \_) \in \text{exec\_verify } \Gamma \gamma f. o_l = 0k$ 
```

Let $d = f(\vec{x}) \{e\}$ and $s = \{ P \} f(\vec{x}) \{ 0k : r. Q_{0k} \} \{ \text{Err} : r. Q_{\text{Err}} \}$.

Assume

$$(\text{H1}) \models^I (\gamma, \Gamma) \quad (\text{H2}) \text{verify_concrete } \Gamma \gamma f = \text{true}$$

Let $f' \in \text{dom}(\Gamma)$

To prove (G1) $\models^I (\gamma[f \leftarrow d], \Gamma[f \leftarrow s])$

If $f' \in \text{dom}(\Gamma)$, then (G1) is trivially true from (H1). We consider the other case, where $f' = f$. We only have to prove that $\gamma \models s$.

Let $\theta, \sigma, \sigma', o, v$ such that

$$(\text{H3}) \theta, \sigma \models P \quad (\text{H4}) \gamma \vdash \sigma, e \Downarrow_{\theta} o : (v, \sigma')$$

To prove

$$(\text{G2}) o \neq \text{Miss} \wedge \theta[r \leftarrow v], \sigma' \models Q_o * \text{True}$$

From (H3), the fact that $\sigma = 0 \bullet \sigma$ and the validity of the assertion producer (Theorem 5.6), we have that

$$(\text{H5}) 0.\text{prod}_P(\theta) \rightsquigarrow \sigma$$

The difficult part of the proof is to find that evaluation of e within the specification context Γ' finds the trace of the hypothesis (H4), when e might make recursive calls to f . More formally, we want to prove the goal:

$$\begin{aligned} \text{(G3)} \quad & \exists o'', \sigma'', v''. \gamma, \Gamma \vdash \sigma, e \Downarrow_{\theta} o'' : (v'', \sigma'') \wedge \\ & (o'' \notin \{\text{Miss}, \text{Lfail}\} \implies o'' = o \wedge v'' = v \wedge \sigma'' = \sigma') \end{aligned}$$

Thankfully, hypothesis (H4) provides a *finite* trace of execution for e , which *must* have a finite depth of recursive calls to f (recall that over-approximating specifications give no guarantee of termination, and of what happens in case of non-termination). Goal (G1) can therefore be obtained by strong induction¹ on the depth of recursive calls to f in the derivation tree for the evaluation in (H4). In particular, the base case is when there are no recursive calls to f in the derivation tree, in which case the hypothesis (H1) immediately gives goal (G1).

In the inductive case, when the derivation tree contains $n+1$ recursive calls to f and we assume that the inductive hypothesis holds for $k \leq n$. To prove the inductive case, we must, in turn, perform an induction on the structure of the expression e . e cannot be a pure expression, as it could not contain a recursive call.

Then, there are two kinds of cases. First, there is the case of let-binding, if/else, while, action execution or the function call case when the function called is not f or when it is f but the evaluation of its arguments fail. In each of these cases, we must prove that each sub-expression used also satisfies the goal our goal. In each case, either the derivation tree for the expression contains a depth less or equal to n of recursive calls to f , in which case the inductive hypothesis of the outer induction gives us the result, or the depth of recursive calls to f is $n+1$, in which case we can apply the inductive hypothesis of the inner induction.

The only case remaining is when the the function f is successfully called, with the inference rule given below:

$$\frac{(\gamma \vdash \sigma_i, e_i \Downarrow_{\theta} \text{Ok} : (v_i, \sigma_{i+1}))_{i=0}^k \quad \gamma \vdash \sigma_{n+1}, e \Downarrow_{\theta[\vec{x} \mapsto v_0 \dots v_k]} o'' : (v'', \sigma'')}{\gamma \vdash \sigma_0, f(e_0, \dots, e_k) \Downarrow_{\theta} o'' : (v'', \sigma'')}$$

In that case, we can use the outer inductive hypothesis to prove that the evaluation of the body of f satisfies the goal (G1), since it must have a depth of recursive calls to f equal to n . This concludes the proof of the goal (G1).

Now that we have proven (G1), using (H2), we know that the final outcome of the evaluation of e , (H6) $o'' \notin \text{Miss}, \text{Lfail}$, and that therefore (H7) $o'' = o \wedge v'' = v \wedge \sigma'' = \sigma'$.

Finally, also from (H2), we know that there is σ_f, θ_f such that (H8) $\sigma'. \text{cons}_{\theta'}(Q_o) \rightarrow \text{Ok} : (\theta_f, \sigma_f)$. From (H8) and the validity of the assertion consumer (Theorem 5.9), we obtain that a σ_{Q_o} such that

$$\text{(H9)} \quad \sigma' = \sigma_f \cdot \sigma_{Q_o} \qquad \text{(H10)} \quad \theta_f, \sigma_{Q_o} \models Q_o$$

With the definition of assertion satisfiability Definition 5.2, we can

¹ Strong induction, here, refers to the induction technique where the inductive case assumes that the results holds for all $k \leq n$.

conclude that $\theta', \sigma' \models Q_O * \text{True}$, which together with (H6) and (H7) gives us our final goal (G2).

Proposition: Symbolic soundness

Now comes the time to check that the symbolic version of the `verify` function satisfies the theorem. The proof is obtained by simply noticing that the symbolic version of `exec_verify` function is a symbolic process that is sound with respect to the concrete version. In particular, for give Γ, γ and f , the symbolic version is a symbolic process that takes no arguments (conceptually, it receives the symbolic unit argument), and returns a triple composed of a logic outcome, a symbolic substitution and a symbolic state.

The symbolic version of `exec_verify` is sound, as it is obtained from by composing sound processes. All functions composed have a symbolic counterpart, except the construction of the initial substitution, which is done through `nondet` in the concrete setting and allocating a fresh var in the symbolic setting. A fresh symbolic variable is unconstrained, and is therefore modeled by all outcomes of `nondet`, making this process sound.

Now, it must be that for any execution path of the concrete `exec_verify`, there is a corresponding symbolic path. Because the logical outcome stays concrete, then it is enough to check that all symbolic paths of end with `Ok` to ensure that all concrete paths end with `Ok`, which concludes the proof.

□

D.2 Specification inference procedure

To prove the soundness of the specification inference procedure using bi-abduction, we start by formulating a concrete version of the bi-abduction state model and prove that the induced semantics correctly performs resource inference. We then lift this result to the symbolic world using judiciously defined abstractions. Finally, we show that the soundness of specification synthesis, which makes use of the symbolic bi-abductive execution yields valid UX specifications.

D.2.1 Concrete bi-abduction state model

A concrete bi-abductive state is a pair (σ, A) compose of a concrete state fragment and a concrete antiframe, that is, an assertion with no variables. We define action execution, consumption and production for the bi-abductive state model using inference rules. We only provide the rules for `non-Miss` and `non-Lfail` outcomes, as these are the only rules relevant for the soundness of the analysis.

$$\begin{array}{c}
\text{BI-ACTION-OK} \\
\frac{\sigma.\alpha(\vec{v}) \xrightarrow{\mathbb{S}} o : (v', \sigma') \quad o \in \{\mathbf{Ok}, \mathbf{Err}\}}{(\sigma, A).\alpha(\vec{v}) \xrightarrow{\text{Bi}(\mathbb{S})} o : (v', (\sigma', A))}
\end{array}
\qquad
\begin{array}{c}
\text{BI-ACTION-FIX} \\
\frac{\begin{array}{l} \sigma.\alpha(\vec{v}) \xrightarrow{\mathbb{S}} \mathbf{Miss} : (v_{fix}, _) \\ A' \in \mathbf{fixes} \ v_{fix} \quad \sigma.\text{prod}_{A'}() \xrightarrow{\mathbb{S}} \sigma'' \\ \sigma''.\alpha(\vec{v}) \xrightarrow{\mathbb{S}} o : (v, \sigma') \end{array}}{(\sigma, A).\alpha(\vec{v}) \xrightarrow{\text{Bi}(\mathbb{S})} o : (v, (\sigma', A * A'))}
\end{array}$$

$$\begin{array}{c}
\text{BI-CONS-OK} \\
\frac{\sigma.\text{cons}_\alpha(\vec{v}_i) \xrightarrow{\mathbb{S}} \mathbf{Ok} : (\vec{v}_o, \sigma') \quad o = \mathbf{Ok}}{(\sigma, A).\text{cons}_\alpha(\vec{v}_i) \xrightarrow{\text{Bi}(\mathbb{S})} \mathbf{Ok} : (\vec{v}_o, (\sigma', A))}
\end{array}
\qquad
\begin{array}{c}
\text{BI-CONS-FIX} \\
\frac{\begin{array}{l} \sigma.\text{cons}_\delta(\vec{v}_i) \xrightarrow{\mathbb{S}} \mathbf{Miss} : (v_{fix}, _) \\ A' \in \mathbf{fixes} \ v_{fix} \quad \sigma.\text{prod}_{A'}() \xrightarrow{\mathbb{S}} \sigma'' \\ \sigma''.\text{cons}_\delta(\vec{v}_i) \xrightarrow{\mathbb{S}} \mathbf{Ok} : (\vec{v}_o, \sigma') \end{array}}{(\sigma, A).\delta(\vec{v}_i) \xrightarrow{\text{Bi}(\mathbb{S})} \mathbf{Ok} : (\vec{v}_o, (\sigma', A * A'))}
\end{array}$$

$$\begin{array}{c}
\text{BI-PROD} \\
\frac{\sigma.\text{prod}_\delta(\vec{v}_i, \vec{v}_o) \xrightarrow{\mathbb{S}} \sigma'}{(\sigma, A).\text{prod}_\delta(\vec{v}_i, \vec{v}_o) \xrightarrow{\text{Bi}(\mathbb{S})} (\sigma', A)}
\end{array}$$

Lemma D.1 (Concrete bi-abductive actions: soundness).

$$\begin{aligned}
& (\sigma, \mathbf{emp}).\alpha(\vec{v}) \xrightarrow{\text{Bi}} o : (v, (\sigma', A)) \wedge o \in \{\mathbf{Ok}, \mathbf{Err}\} \\
& \implies \exists \sigma_s. \sigma.\text{prod}_A() \rightsquigarrow \sigma_s \wedge \sigma_s.\alpha(\vec{v}) \rightsquigarrow o : (v, \sigma')
\end{aligned}$$

Proof. **Assume**

$$(H1) \ (\sigma, \mathbf{emp}).\alpha(\vec{v}) \xrightarrow{\text{Bi}} o : (v, (\sigma', A))$$

$$(H2) \ o \notin \{\mathbf{Miss}, \mathbf{Lfail}\}$$

To prove (G1) $\exists \sigma_s. \sigma.\text{prod}_A() \rightsquigarrow \sigma_s \wedge \sigma_s.\alpha(\vec{v}) \rightsquigarrow o : (v, \sigma')$

By inversion, there are only two rules that can lead to (H2): **BI-ACTION-OK** and **BI-ACTION-FIX**. We consider the two cases separately.

Case BI-ACTION-OK:

In this case, we have the following new hypotheses:

$$(H3) \ \sigma.\alpha(\vec{v}) \rightsquigarrow o : (v, \sigma')$$

$$(H4) \ A = \mathbf{emp}$$

By selecting $\sigma_s = \sigma$, we trivially obtain (G1) from (H3) and (H4).

Case BI-ACTION-FIX:

In this case, we have the following new hypotheses:

$$(H5) \ \sigma.\alpha(\vec{v}) \rightsquigarrow \mathbf{Miss} : (v_{fix}, _) \qquad (H6) \ A \in \mathbf{fixes} \ v_{fix}$$

$$(H7) \ \sigma.\text{prod}_A() \rightsquigarrow \sigma'' \qquad (H8) \ \sigma''.\alpha(\vec{v}) \rightsquigarrow o : (v, \sigma')$$

$$(H9) \ o \in \{\mathbf{Ok}, \mathbf{Err}\}$$

(H7), and (H8) immediately give the goal (G1) by selecting $\sigma_s = \sigma''$. \square

Lemma D.2 (Concrete bi-abductive consumption: soundness).

If P is a strictly-exact assertion then

$$\begin{aligned} (\sigma, \text{emp}).\text{cons}_P(\theta) &\xrightarrow{\text{Bi}} \text{Ok} : (\theta', (\sigma', A)) \wedge \theta', \sigma_P \models P \wedge \sigma_P \# \sigma_f \\ \implies \exists \sigma_A. \sigma_A &\models A \wedge \sigma_P \cdot \sigma_f = \sigma_A \cdot \sigma \end{aligned}$$

Proof. **Assume**

$$(H1) \quad (\sigma, \text{emp}).\text{cons}_P(\theta) \xrightarrow{\text{Bi}} \text{Ok} : (\theta', (\sigma', A))$$

$$(H2) \quad \theta', \sigma_P \models P$$

$$(H3) \quad \sigma_P \# \sigma_f$$

To prove (G1) $\exists \sigma_A. \sigma_A \models A \wedge \sigma_P \cdot \sigma_f = \sigma_A \cdot \sigma$

Since consumption is successful, we know that P can be decomposed into a list of core predicates $\langle \delta^j \rangle (\vec{e}_i^j; \vec{e}_o^j) \mid_{j=1}^n$ which are successfully consumed one by one according to the plan. We define $\sigma_0 = \sigma$ and $\theta_0 = \theta$ and $A_0 = \text{emp}$. At each step of consumption, we obtain a new state, substitution and antiframe σ_{j+1} , θ_{j+1} and A_{j+1} . We know that the anti-frame is monotonically extended, so that $A_{j+1} = B_{j+1} * A_j$ for some B that is either a fix that allowed consumption to continue, or emp if the consumption was successful without any fix. Finally, we denote $P_j = \otimes_{k=1}^j \langle \delta^k \rangle (\vec{e}_i^k; \vec{e}_o^k)$.

We prove the following goal for all $0 \leq j \leq n$, by induction on j , which implies the goal (G1) for $j = n$:

$$(G2) \quad \exists \sigma_{P_j}, \sigma_{A_j}. \sigma_{A_j} \models A_j \wedge \sigma_{P_j} \cdot \sigma_j = \sigma_{A_j} \cdot \sigma$$

Case Base case, $j = 0$:

$P_0 = \text{emp}$, $A_j = \text{emp}$, $\sigma_0 \cdot 0 = 0 \cdot \sigma$ with $0 \models \text{emp}$. So trivially true.

Case Inductive case, $j + 1$:

We assume the inductive hypothesis:

$$(IH1)_j \quad \exists \sigma_{P_j}, \sigma_{A_j}. \theta_j, \sigma_{P_j} \models P_j \wedge \sigma_{A_j} \models A_j \wedge \sigma_{P_j} \cdot \sigma_j = \sigma_{A_j} \cdot \sigma$$

And aim to prove $(IH1)_{j+1}$.

We have, by inverting the consume-step rule, and because we've already proved of the correctness of the parametric consumer:

$$(H4) \quad \llbracket \vec{e}_i^{j+1} \rrbracket_{\theta_{j+1}} = \vec{v}_i^{j+1}$$

$$(H5) \quad (\sigma_j, A_j).\text{cons}_{\delta^{j+1}}(\vec{v}_i^{j+1}) \xrightarrow{\text{Bi}} \text{Ok} : (\vec{v}_o^{j+1}, (\sigma_{j+1}, B_{j+1} * A_j))$$

$$(H6) \quad \llbracket \vec{e}_o^{j+1} \rrbracket_{\theta_{j+1}} = \vec{v}_o^{j+1}$$

From (H5) by definition of the bi-abductive state model consumer we have $\exists \sigma_{B_{j+1}}$:

$$(H7) \quad \sigma_{B_{j+1}} \models B_{j+1}$$

$$(H8) \quad (\sigma_j \cdot \sigma_{B_{j+1}}).\text{cons}_{\delta^{j+1}}(\vec{v}_i^{j+1}) \xrightarrow{\mathbb{S}} \text{Ok} : (\vec{v}_o^{j+1}, \sigma_{j+1})$$

Using (H4), (H6) and (H10), the validity of consumers for \mathbb{S} , and the definition of satisfiability for assertions, there exists $\sigma_{\delta_{j+1}}$ such that

$$(H9) \quad \theta_{j+1}, \sigma_{\delta_{j+1}} \models \langle \delta^{j+1} \rangle (\vec{v}_i^{j+1}; \vec{v}^{j+1})$$

$$(H10) \quad \sigma_j \cdot \sigma_{B_{j+1}} = \sigma_{\delta_{j+1}} \cdot \sigma_{j+1}$$

From (H3), we know that $\sigma_P \# \sigma_f$. We also know that $\theta', \sigma_P \models P$ and P is strictly exact, so it is the unique model of P under that substitution. Using a decreasing induction, we can deduce that $(\sigma_{P_j} \cdot \sigma_{\delta^{j+1}}) \# \sigma_{j+1}$ at each step, as otherwise the final step would not be disjoint.

By defining $\sigma_{P_{j+1}} = \sigma_{P_j} \cdot \sigma_{\delta^{j+1}}$ and $\sigma_{A_{j+1}} = \sigma_{A_j} \cdot \sigma_{B_{j+1}}$, we can now compute:

$$\begin{aligned} \sigma_{j+1} \cdot \sigma_{P_{j+1}} &= \sigma_{j+1} \cdot (\sigma_{\delta^{j+1}} \cdot \sigma_{P_j}) && \text{by def.} \\ &= (\sigma_{j+1} \cdot \sigma_{\delta^{j+1}}) \cdot \sigma_{P_j} && \text{by assoc} \\ &= (\sigma_j \cdot \sigma_{B_{j+1}}) \cdot \sigma_{P_j} && \text{by (H10)} \\ &= (\sigma_j \cdot \sigma_{P_j}) \cdot \sigma_{B_{j+1}} && \text{assoc. and commut.} \\ &= (\sigma \cdot \sigma_{A_j}) \cdot \sigma_{B_{j+1}} && \text{using (IH1)}_j \\ &= \sigma \cdot (\sigma_{A_j} \cdot \sigma_{B_{j+1}}) && \text{assoc.} \\ &= \sigma \cdot \sigma_{A_{j+1}} && \text{def.} \end{aligned}$$

Which establishes our inductive hypothesis for $j + 1$. □

Lemma D.3 (Concrete bi-abductive specification execution: soundness).

If $S = [P] f(\vec{x}) [o : \mathbf{r}. Q]$ is a valid ISL specification, and $\gamma(f) = f(\vec{x}) \{e\}$

$$\begin{aligned} \gamma \models S \wedge (\sigma, \text{emp}).\text{spec}_S(\theta) &\stackrel{\text{Bi}}{\rightsquigarrow} o : (v, (\sigma', A)) \wedge o \in \{\mathbf{Ok}, \mathbf{Err}\} \\ \implies \exists \sigma_s. \sigma.\text{prod}_A() &\stackrel{\mathbb{S}}{\rightsquigarrow} \sigma_s \wedge \gamma \vdash \sigma_s, e \Downarrow_{\theta}^{\mathbb{S}} o : (v, \sigma') \end{aligned}$$

Proof. let $[P] f(\vec{x}) [o : \mathbf{r}. Q]$ be the specification S .

$$(H1) \quad \gamma \models S$$

$$(H2) \quad (\sigma, \text{emp}).\text{spec}_S(\theta) \stackrel{\text{Bi}}{\rightsquigarrow} o : (v, (\sigma', A))$$

$$(H3) \quad o \in \{\mathbf{Ok}, \mathbf{Err}\}$$

To prove : (G1) $\exists \sigma_s. \sigma.\text{prod}_A() \stackrel{\mathbb{S}}{\rightsquigarrow} \sigma_s \wedge \sigma_s.\alpha(\vec{v}) \stackrel{\mathbb{S}}{\rightsquigarrow} o : (v, \sigma')$

From (H2), (H3) and by inversion on **EXEC-SPEC-SUCCESS**, we get that:

$$(H4) \quad (\sigma, \text{emp}).\text{cons}_P(\theta) \stackrel{\text{Bi}}{\rightsquigarrow} \mathbf{Ok} : (\theta', (\sigma_f, A))$$

$$(H5) \quad (\sigma_f, A).\text{prod}_Q(\theta' [\mathbf{r} \leftarrow v]) \rightsquigarrow (\sigma', A)$$

From (H5), we know that: $\exists \sigma_Q$ such that:

$$(H6) \quad \theta' [\mathbf{r} \leftarrow v], \sigma_Q \models Q$$

$$(H7) \quad \sigma' = \sigma_Q \cdot \sigma_f$$

(H1) tells us that S is a valid ISL specification, and hence, since we have a model of Q ((H6)), there exists a state fragment σ_P and a transition:

$$(H8) \quad \theta', \sigma_P \models P$$

$$(H9) \quad \gamma \vdash \sigma_P, e \Downarrow_{\theta'}^{\mathbb{S}} o : (v, \sigma_Q)$$

where $f(\vec{x})\{e\} = \gamma(f)$.

From (H9), (H7) that indicates that $\sigma_Q \# \sigma_f$, and the fact that the semantics is UX-frame-preserving and satisfies [Frame addition](#), we get:

$$(H10) \quad \gamma \vdash \sigma_P \cdot \sigma_f, e \Downarrow_{\theta'}^{\mathbb{S}} o : (v, \sigma')$$

Then, from (H4), (H8), (H10) (which implies that $\sigma_P \# \sigma_f$), and [Lemma D.2](#), we get that $\exists \sigma_A$:

$$(H11) \quad \sigma_A \models A$$

$$(H12) \quad \sigma_P \cdot \sigma_f = \sigma_A \cdot \sigma$$

Let $\sigma_S = \sigma_A \cdot \sigma$. By validity of producer and (H11), definition of σ_S , (H12) and (H10), we obtain the goal (G1).

□

Theorem 7.3 (Concrete bi-abductive execution: soundness).

If $\models^{\text{ux}} (\gamma, \Gamma)$, then

$$\begin{aligned} \gamma, \Gamma \vdash (\sigma, \text{emp}), e \Downarrow_{\theta}^{\text{Bi}(\mathbb{S})} o : (v, (\sigma', A)) \wedge o \notin \{\text{Miss}, \text{Lfail}\} \\ \implies \exists \sigma_s. \sigma.\text{prod}_A() \rightsquigarrow \sigma_s \wedge \gamma \vdash \sigma_s, e \Downarrow_{\theta}^{\mathbb{S}} o : (v, \sigma') \end{aligned}$$

Proof. First, we introduce an intermediate lemma which can be trivially proven using the rules above:

$$\begin{aligned} (\sigma, A).\alpha(\vec{v}) \rightsquigarrow^{\text{Bi}} o : (v, (\sigma', A')) \implies \\ \exists A''. (A' \Leftrightarrow A * A'') \wedge (\sigma, \text{emp}).\alpha(\vec{v}) \rightsquigarrow^{\text{Bi}} o : (v, (\sigma', A'')) \end{aligned}$$

(Antiframe mono)

In other words, the anti-frame is monotonically increasing, and the input anti-frame is irrelevant and maintained.

We proceed with the proof of our lemma by induction on the structure of e . The only base case is the evaluation of a pure expression, which cannot end in a missing error, and therefore the induction goal trivially holds, similarly to the first case of the proof above.

There are illustrative cases: sequential composition using let-binding and action evaluation. The most important case is sequential composition, as it illustrates how several operations that preserve the property can be composed together.

Case $e = \text{let } x = e_1 \text{ in } e_2$:

Assume Inductive hypotheses:

$$\begin{aligned} \text{(IH1)} \quad & \gamma, \Gamma \vdash (\sigma, \text{emp}), e_1 \Downarrow_{\theta}^{\text{Bi}(\mathbb{S})} o : (v, (\sigma', A)) \wedge o \notin \{\text{Miss}, \text{Lfail}\} \\ & \implies \exists \sigma_s. \sigma.\text{prod}_A() \rightsquigarrow \sigma_s \wedge \gamma \vdash \sigma_s, e_1 \Downarrow_{\theta}^{\mathbb{S}} o : (v, \sigma') \\ \text{(IH2)} \quad & \gamma, \Gamma \vdash (\sigma, \text{emp}), e_2 \Downarrow_{\theta}^{\text{Bi}(\mathbb{S})} o : (v, (\sigma', A)) \wedge o \notin \{\text{Miss}, \text{Lfail}\} \\ & \implies \exists \sigma_s. \sigma.\text{prod}_A() \rightsquigarrow \sigma_s \wedge \gamma \vdash \sigma_s, e_2 \Downarrow_{\theta}^{\mathbb{S}} o : (v, \sigma') \end{aligned}$$

Assume

$$\text{(H1)} \quad \gamma, \Gamma \vdash (\sigma, \text{emp}), e \Downarrow_{\theta}^{\text{Bi}(\mathbb{S})} o : (v, (\sigma', A'))$$

$$\text{(H2)} \quad o \notin \{\text{Miss}, \text{Lfail}\}$$

To prove (G1) $\exists \sigma_s. \sigma.\text{prod}_{A'}() \rightsquigarrow \sigma_s \wedge \gamma \vdash \sigma_s, e \Downarrow_{\theta}^{\mathbb{S}} o : (v, \sigma')$

There are two sub-cases: either the evaluation of e_1 yields a **Err** outcome, or it yields a **Ok** outcome and the subsequent evaluation of e_2 yields a **Ok** or **Err** outcome. In the first case, we can immediately apply the inductive hypothesis (IH1) to obtain the goal. We consider the second case:

$$\text{(H3)} \quad \gamma, \Gamma \vdash (\sigma, \text{emp}), e_1 \Downarrow_{\theta}^{\text{Bi}(\mathbb{S})} \text{Ok} : (v_1, (\sigma_1, A_1))$$

$$\text{(H4)} \quad \gamma, \Gamma \vdash (\sigma_1, A_1), e_2 \Downarrow_{\theta[x \leftarrow v_1]}^{\text{Bi}(\mathbb{S})} o : (v, (\sigma', A'))$$

By (IH1) and (H3), we have that there is a σ_s^1 such that:

$$\text{(H5)} \quad \sigma.\text{prod}_{A_1}() \rightsquigarrow \sigma_s^1$$

$$\text{(H6)} \quad \gamma \vdash \sigma_s^1, e \Downarrow_{\theta}^{\mathbb{S}} \text{Ok} : (v_1, \sigma_1)$$

By using [Antiframe mono](#) on (H4), we obtain that there is a A_2 such that:

$$(H7) \quad A' \Leftrightarrow A_1 * A_2$$

$$(H8) \quad \gamma, \Gamma \vdash (\sigma_1, \mathbf{emp}), e_2 \Downarrow_{\theta[x \leftarrow v_1]}^{\text{Bi}(\mathbb{S})} o : (v, (\sigma', A_2))$$

From (H8) and (IH2), we get that there is σ_s^2 such that:

$$(H9) \quad \sigma_1 \cdot \text{prod}_{A_2}() \rightsquigarrow \sigma_s^2$$

$$(H10) \quad \gamma \vdash \sigma_s^2, e_2 \Downarrow_{\theta[x \leftarrow v_1]}^{\mathbb{S}} o : (v, \sigma')$$

From (H9), we have that $\exists \sigma_{A_1}$ such that:

$$(H11) \quad \sigma_s^2 = \sigma_1 \cdot \sigma_{A_2} \quad (H12) \quad \sigma_{A_2} \models A_2$$

From (H11), (H6) and [Frame addition](#), we have that

$$(H13) \quad \gamma \vdash \sigma_s^1 \cdot \sigma_{A_2}, e \Downarrow_{\theta}^{\mathbb{S}} \mathbf{Ok} : (v_1, \sigma_s^2)$$

By validity of producer and (H12), we have that:

$$(H14) \quad \sigma_1 \cdot \text{prod}_{A_2}() \rightsquigarrow \sigma_s^1 \cdot \sigma_{A_2}$$

From (H5), (H14) and definition production of $A_1 * A_2$, we have that:

$$(H15) \quad \sigma \cdot \text{prod}_{A_1 * A_2}() \rightsquigarrow \sigma_s^1 \cdot \sigma_{A_2}$$

In addition, from (H13) and (H10), and (H11) we have that:

$$(H16) \quad \gamma \vdash \sigma_s^1 \cdot \sigma_{A_2}, e \Downarrow_{\theta}^{\mathbb{S}} o : (v, \sigma')$$

Together, (H15) and (H16) give us the induction goal (G1).

Case $e = \alpha(e_1, \dots, e_k)$:

We do not provide this proof in detail as it can be obtained from composing arguments already exposed. First, all arguments are evaluated left-to-right. It can be proven by a simple induction that leverages the exact same arguments as the proof of the let-binding case that sequentially evaluating all of the arguments preserves the desired property. Then, action evaluation also preserves the desired property, as shown in [Lemma D.1](#). By composing these two results, we obtain our goal.

Case $e = f(e_1, \dots, e_k)$:

This case is similar to the action case, but makes use of [Lemma D.3](#) instead of [Lemma D.1](#).

□

D.2.2 Specification synthesis

Now that we have shown that, concretely, the bi-abduction state model, when execution finishes without reasoning error, produces the anti-frame that allows for this execution, we show that the specification synthesis is correct. We first show that the symbolic version of the bi-abduction state model is sound with respect to the concrete version, and then show that the symbolic specification synthesis is sound.

We provide the rule for successful specification synthesis:

$$\begin{array}{c}
\text{SPEC-SYNTH} \\
f(\vec{x}) \{e\} = \gamma(f) \quad \vec{y} = \text{fv}(P) \quad \bar{\theta}_{id}^{\vec{x}, \vec{y}} = [\vec{x} \mapsto \vec{x}, \vec{y} \mapsto \vec{y}] \\
\bar{0}.\text{prod}_P(\bar{\theta}_{id}^{\vec{x}, \vec{y}}) \rightsquigarrow \bar{\sigma} \\
\gamma, \Gamma \vdash (\bar{\sigma}, \text{emp}), e \Downarrow_{\bar{\theta}_{id}^{\vec{x}, \vec{y}}}^{\text{Bi}(\bar{\sigma})} \langle o : (\bar{v}, (\bar{\sigma}', \bar{A})) \mid \pi \rangle \quad o \in \{\text{Ok}, \text{Err}\} \\
\frac{P' = P * \bar{\theta}_{id}^{\vec{x}, \vec{y}} \bar{A} \quad Q = \text{to_asrt } \bar{\theta}_{id}^{\vec{x}, \vec{y}} \bar{v} \bar{\sigma}' \pi}{[\textcolor{blue}{P'}] f(\mathbf{x}) [\textcolor{blue}{o : r. Q}] \in \text{synthesise } \gamma \Gamma f P}
\end{array}$$

where to_asrt is such that if $Q = \text{to_asrt } \bar{\theta}_{id}^{\vec{x}, \vec{y}} \bar{v} \bar{\sigma} \pi$, then,

$$\begin{aligned}
& \varepsilon, (v, \sigma) \models \langle (\bar{v}, \bar{\sigma}) \mid \pi \rangle \\
& \iff \varepsilon(\bar{\theta}_{id}^{\vec{x}, \vec{y}} [\mathbf{r} \leftarrow v], \sigma \models Q
\end{aligned}$$

Theorem 7.2 (Specification synthesis: soundness).

$$\begin{aligned}
& \models (\gamma, \Gamma) \wedge [\textcolor{blue}{P'}] f(\mathbf{x}) [\textcolor{blue}{o : r. Q * R}] \in \text{synthesise } \Gamma \gamma f P \\
& \implies \gamma \models [\textcolor{blue}{P'}] f(\mathbf{x}) [\textcolor{blue}{o : r. Q * R}]
\end{aligned}$$

Proof.

$$(H1) \models (\gamma, \Gamma)$$

$$(H2) [\textcolor{blue}{P'}] f(\mathbf{x}) [\textcolor{blue}{o : r. Q}] \in \text{synthesise } \gamma \Gamma f P$$

By inversion on **SPEC-SYNTH** for (H2), we have the following new hypotheses, where $\gamma(f) = f(\vec{x}) \{e\}$, $\vec{y} = \text{fv}(P)$ and $\bar{\theta}_{id}^{\vec{x}, \vec{y}} = [\vec{x} \mapsto \vec{x}, \vec{y} \mapsto \vec{y}]$:

$$(H3) \bar{0}.\text{prod}_P(\bar{\theta}_{id}^{\vec{x}, \vec{y}}) \rightsquigarrow \bar{\sigma}$$

$$(H4) \gamma, \Gamma \vdash (\bar{\sigma}, \text{emp}), e \Downarrow_{\bar{\theta}_{id}^{\vec{x}, \vec{y}}}^{\text{Bi}(\bar{\sigma})} \langle o : (\bar{v}, (\bar{\sigma}', \bar{A})) \mid \pi \rangle$$

$$(H5) o \in \{\text{Ok}, \text{Err}\}$$

$$(H6) P' = P * \bar{\theta}_{id}^{\vec{x}, \vec{y}} \bar{A}$$

$$(H7) Q = \text{to_asrt } \bar{\theta}_{id}^{\vec{x}, \vec{y}} \bar{v} \bar{\sigma}' \pi$$

We want to prove that $\gamma \models [\textcolor{blue}{P'}] f(\mathbf{x}) [\textcolor{blue}{o : r. Q}]$. Let θ, o, σ', v such that

$$(H8) \theta [\mathbf{r} \leftarrow v], \sigma' \models Q$$

To prove (G1) $\exists \sigma_s. \theta, \sigma_s \models P' \wedge \gamma \vdash \sigma_s, e \Downarrow_{\theta} o : (v, \sigma')$

Take ε such that $\varepsilon(\bar{\theta}_{id}^{\vec{x}, \vec{y}}) = \theta$. This is possible because $\bar{\theta}_{id}^{\vec{x}, \vec{y}}$ is a substitution with only symbolic variables in the codomain, so ε can be constructed as $[\bar{x}_0 \mapsto \theta(\mathbf{x}_0), \dots]$.

By definition of to_asrt , we have:

$$(H9) \varepsilon, (v, \sigma') \models \langle (\bar{v}, \bar{\sigma}') \mid \pi \rangle$$

Since the anti-frame is an assertion with symbolic values instead of assertions, it can be concretized using ε by concretizing each symbolic value to a concrete value. We define $A = \bar{A}(\varepsilon)$.

By definition of satisfiability for symbolic branches, and using (H9) we have that:

$$(H10) \quad \varepsilon, (o : (v, (\sigma', A))) \models \langle o : (\bar{v}, (\bar{\sigma}', \bar{A})) \mid \pi \rangle$$

From (H1), (H10), and the UX-soundness of bi-abductive execution, we have that $\exists \sigma, \theta'.$:

$$(H11) \quad \gamma \vdash (\sigma, emp), e \Downarrow_{\theta'}^{\text{Bi}(\mathbb{S})} o : (v, (\sigma', A))$$

$$(H12) \quad \varepsilon, \theta' \models \bar{\theta}_{id}^{\bar{x}, \bar{y}}$$

$$(H13) \quad \varepsilon, \sigma \models \bar{\sigma}'$$

Because $\bar{\theta}_{id}^{\bar{x}, \bar{y}}$ may only have one model under ε , using (H12) and the construction of ε , we have (H14) $\theta = \theta'$.

From (H11), (H14) and the soundness of concrete bi-abductive execution, we have that $\exists \sigma_s.$:

$$(H15) \quad \sigma.\text{prod}_A() \rightsquigarrow \sigma_s$$

$$(H16) \quad \gamma \vdash \sigma_s, e \Downarrow_{\theta} o : (v, \sigma')$$

The only property left to prove is that $\theta, \sigma_s \models P'$.

First, from (H15), the definition of $A = \varepsilon(\bar{A})$, and the relationship between $\varepsilon(\bar{\theta}_{id}^{\bar{x}, \bar{y}}) = \theta$, we have that:

$$(H17) \quad \sigma.\text{prod}_{\bar{\theta}_{id}^{\bar{x}, \bar{y}}(\bar{A})}^{\leftarrow}(\theta) \rightsquigarrow \sigma_s$$

and hence, by validity of the producer and (H17), we have $\exists \sigma_A$ such that

$$(H18) \quad \sigma_s = \sigma_A \cdot \sigma \qquad (H19) \quad \theta, \sigma_A \models \bar{\theta}_{id}^{\bar{x}, \bar{y}}(\bar{A})$$

Furthermore, from (H13), (H3), and the UX soundness of the producer, we have $\exists \sigma_0, \theta''.$

$$(H20) \quad \varepsilon, \sigma_0 \models \bar{0}$$

$$(H21) \quad \varepsilon, \theta'' \models \bar{\theta}_{id}^{\bar{x}, \bar{y}}$$

$$(H22) \quad \sigma_0.\text{prod}_P(\theta'') \rightsquigarrow \sigma$$

By definition of $\bar{0}$, and because $\bar{\theta}_{id}^{\bar{x}, \bar{y}}$ has only one model under ε , these three hypotheses together give us:

$$(H23) \quad \theta, \sigma \models P$$

Together, (H19), (H23), (H18) and the definition of assertion satisfiability give us:

$$(H24) \quad \theta, \sigma_s \models P'$$

(H24) was the last require to obtain our (G1). \square

Appendix E

Constructing State Models

E.1 Product of state models

Lemma 8.1 (Product state model: concrete soundness).

If $\underline{\mathbb{S}}_1 \stackrel{\mathcal{C}}{\sim}_m \mathbb{S}_1$ according to $\models_{\mathcal{C}}^1$ and $\underline{\mathbb{S}}_2 \stackrel{\mathcal{C}}{\sim}_m \mathbb{S}_2$ according to $\models_{\mathcal{C}}^2$, then $(\underline{\mathbb{S}}_1 \times \underline{\mathbb{S}}_2) \stackrel{\mathcal{C}}{\sim}_m (\mathbb{S}_1 \times \mathbb{S}_2)$ according to $\models_{\mathcal{C}}^{1 \times 2}$ where

$$(\underline{\sigma}_1, \underline{\sigma}_2) \models_{\mathcal{C}}^{1 \times 2} (\sigma_1, \sigma_2) \Leftrightarrow \underline{\sigma}_1 \models_{\mathcal{C}}^1 \sigma_1 \wedge \underline{\sigma}_2 \models_{\mathcal{C}}^2 \sigma_2$$

In mathematical notations:

$$\underline{\mathbb{S}}_1 \stackrel{\mathcal{C}}{\sim}_m \mathbb{S}_1 \wedge \underline{\mathbb{S}}_2 \stackrel{\mathcal{C}}{\sim}_m \mathbb{S}_2 \Rightarrow (\underline{\mathbb{S}}_1 \times \underline{\mathbb{S}}_2) \stackrel{\mathcal{C}}{\sim}_m (\mathbb{S}_1 \times \mathbb{S}_2)$$

Proof.

Proposition: frame addition We assume that actions of each parameter state models satisfy frame addition.

$$\begin{aligned} \text{(H1)} \quad & \sigma_1. \alpha_1(\vec{v}) \rightsquigarrow o : (v', \sigma'_1) \wedge \sigma'_1 \# \sigma_1^f \wedge o \neq \text{Miss} \\ & \Rightarrow (\sigma_1 \cdot \sigma_1^f). \alpha_1(\vec{v}) \rightsquigarrow o : (v', \sigma'_1 \cdot \sigma_1^f) \end{aligned}$$

$$\begin{aligned} \text{(H2)} \quad & \sigma_2. \alpha_2(\vec{v}) \rightsquigarrow o : (v', \sigma'_2) \wedge \sigma'_2 \# \sigma_2^f \wedge o \neq \text{Miss} \\ & \Rightarrow (\sigma_2 \cdot \sigma_2^f). \alpha_2(\vec{v}) \rightsquigarrow o : (v', \sigma'_2 \cdot \sigma_2^f) \end{aligned}$$

Let $\sigma = (\sigma_1, \sigma_2)$, $\sigma' = (\sigma'_1, \sigma'_2)$ and $\sigma^f = (\sigma_1^f, \sigma_2^f)$.

Let us assume the left-hand side of the frame addition implication:

$$\text{(H3)} \quad \sigma. \alpha(\vec{v}) \rightsquigarrow o : (v', \sigma') \quad \text{(H4)} \quad \sigma^f \# \sigma'$$

Our goal is to prove **(G1)** $(\sigma \cdot \sigma^f). \alpha(\vec{v}) \rightsquigarrow o : (v', \sigma' \cdot \sigma^f)$.

There are two symmetric cases: either $\alpha = \text{A1 } \alpha_1$ or $\alpha = \text{A2 } \alpha_2$. We only consider the first case, the second being analogous.

$$\text{(H5)} \quad \alpha = \text{A1 } \alpha_1$$

According to the definition of action evaluation in the compositional state model, we know from (H3) that:

$$\text{(H6)} \quad \sigma_1. \alpha_1(\vec{v}) \rightsquigarrow o : (v', \sigma'_1)$$

$$\text{(H7)} \quad \sigma'_2 = \sigma_2$$

Furthermore, from the definition of σ^f and (H4), we also learn that:

$$\text{(H8)} \quad \sigma'_1 \# \sigma_1^f \quad \text{(H9)} \quad \sigma'_2 \# \sigma_2^f$$

We can apply (H1) using (H6) and (H8), obtaining

$$\text{(H10)} \quad (\sigma_1 \cdot \sigma_1^f). \alpha_1(\vec{v}) \rightsquigarrow o : (v', \sigma'_1 \cdot \sigma_1^f)$$

By definition of action evaluation in the pair, (H5) and (H9), we can derive

$$(H11) \quad (\sigma_1 \cdot \sigma_1^f, \sigma_2 \cdot \sigma_2^f). \alpha(\vec{v}) \rightsquigarrow o : (v', \sigma_1' \cdot \sigma_1^f, \sigma_2 \cdot \sigma_2^f)$$

Finally, the above (H11) together with (H7) is our goal (G1).

Proposition: frame subtraction

We assume that actions of each parameter state models satisfy frame subtraction.

$$(H1) \quad \begin{aligned} & \sigma_1 \cdot \sigma_1^f. \alpha_1(\vec{v}) \rightsquigarrow o : (v, \sigma_1') \implies \\ & (\exists o', v', \sigma_1''. \sigma_1. \alpha_1(\vec{v}) \rightsquigarrow o' : (v', \sigma_1'') \wedge \\ & \quad (o' \neq \text{Miss} \Rightarrow \sigma_1' = \sigma_1'' \cdot \sigma_1^f \wedge o = o' \wedge v = v')) \\ & \sigma_2 \cdot \sigma_2^f. \alpha_2(\vec{v}) \rightsquigarrow o : (v, \sigma_2') \implies \\ & (H2) \quad (\exists o', v', \sigma_2''. \sigma_2. \alpha_2(\vec{v}) \rightsquigarrow o' : (v', \sigma_2'') \wedge \\ & \quad (o' \neq \text{Miss} \Rightarrow \sigma_2' = \sigma_2'' \cdot \sigma_2^f \wedge o = o' \wedge v = v')) \end{aligned}$$

Assume (H3) $\sigma \cdot \sigma^f. \alpha(\vec{v}) \rightsquigarrow o : (v', \sigma')$

There are two cases, either $\alpha = A1 \ \alpha_1$ or $\alpha = A2 \ \alpha_2$.

Case $\alpha = A1 \ \alpha_1$:

We have that:

$$(H4) \quad \sigma = (\sigma_1, \sigma_2)$$

$$(H5) \quad \sigma^f = (\sigma_1^f, \sigma_2^f)$$

$$(H6) \quad (\sigma_1 \cdot \sigma_1^f). \alpha_1(\vec{v}) \rightsquigarrow o : (v', \sigma_1')$$

$$(H7) \quad \sigma' = (\sigma_1', \sigma_2 \cdot \sigma_2^f)$$

From (H1) and (H6), we can derive that there exist o', v', σ_1'' such that:

$$(H8) \quad \sigma_1. \alpha_1(\vec{v}) \rightsquigarrow o' : (v', \sigma_1'')$$

$$(H9) \quad o' \neq \text{Miss} \Rightarrow \sigma_1' = \sigma_1'' \cdot \sigma_1^f \wedge o = o' \wedge v = v'$$

Since the second element of the state remains unaffected by the action, we also have that, using (H8):

$$(H10) \quad (\sigma_1, \sigma_2). \alpha(\vec{v}) \rightsquigarrow o' : (v', (\sigma_1'', \sigma_2))$$

Furthermore, if $o' \neq \text{Miss}$, from (H9) we get our full goal.

Proposition: concrete soundness The proof goes similarly to that of frame-preservation, we elide it here. \square

Lemma 8.3 (Product state model: producers and consumers).

The producer and consumer of the product compositional state models are valid according to [Definition 5.5](#) and [Definition 5.7](#).

Proof.

Proposition: Validity of the producer

We assume that the producer of each parameter state model is valid:

$$(H1) \quad \mathbb{S}_1.\text{produce } \sigma_1 \delta_1 \vec{v}_i \vec{v}_o = \{\sigma_1 \bullet \sigma'_1 \mid \sigma'_1 \models \langle \delta_1 \rangle(\vec{v}_i; \vec{v}_o)\}$$

$$(H2) \quad \mathbb{S}_2.\text{produce } \sigma_2 \delta_2 \vec{v}_i \vec{v}_o = \{\sigma_2 \bullet \sigma'_1 \mid \sigma'_1 \models \langle \delta_2 \rangle(\vec{v}_i; \vec{v}_o)\}$$

We want to prove that the producer of the product state model is valid:

$$(G1) \quad (\mathbb{S}_1 \times \mathbb{S}_2).\text{produce } \sigma \delta \vec{v}_i \vec{v}_o = \{\sigma \bullet \sigma' \mid \sigma' \models \langle \delta \rangle(\vec{v}_i; \vec{v}_o)\}$$

Let δ be a core predicate of the product state model and $\sigma = (\sigma_1, \sigma_2)$. There are two cases, either $\delta = C1 \delta_1$ or $\delta = C2 \delta_2$. We only consider the first case, the second being analogous

$$(H3) \quad \delta = C1 \delta_1$$

According to the definition of the producer in the product state model, we know that:

$$\begin{aligned} & (\mathbb{S}_1 \times \mathbb{S}_2).\text{produce } \delta (\sigma_1, \sigma_2) \vec{v}_i \vec{v}_o \\ &= \{(\sigma'_1, \sigma_2) \mid \sigma_1 \in \mathbb{S}_1.\text{produce } \delta \sigma_1 \vec{v}_i \vec{v}_o\} && \text{(by (H3))} \\ &= \{(\sigma'_1, \sigma_2) \mid \sigma'_1 \in \{\sigma_1 \bullet \sigma''_1 \mid \sigma''_1 \models \langle \delta_1 \rangle(\vec{v}_i; \vec{v}_o)\}\} && \text{(by (H1))} \\ &= \{(\sigma_1 \bullet \sigma'_1, \sigma_2) \mid \sigma'_1 \models \langle \delta_1 \rangle(\vec{v}_i; \vec{v}_o)\} \\ &= \{(\sigma_1, \sigma_2) \bullet (\sigma'_1, 0) \mid \sigma'_1 \models \langle \delta_1 \rangle(\vec{v}_i; \vec{v}_o)\} && \text{(def. of } \bullet \text{)} \\ &= \{\sigma \bullet \sigma' \mid \sigma' \models \langle \delta \rangle(\vec{v}_i; \vec{v}_o)\} && \text{(by (H3) \& def.)} \end{aligned}$$

The last line of the above derivation is our goal (G1).

Proposition: Validity of the consumer

We elide the proof that the consumer of the product state model does not yield missing on full states (as it is trivially true). We only prove that the consumer of the product state model preserves [CP Consuming](#).

$$\begin{aligned} (H1) \quad & \sigma_1.\text{cons}_{\delta_1}(\vec{v}_i) \rightarrow \text{Ok} : (\vec{v}_o, \sigma'_1) \\ \implies & \exists \sigma_{\delta_1}. \sigma = \sigma'_1 \bullet \sigma_{\delta_1} \wedge \sigma_{\delta_1} \models \langle \delta_1 \rangle(\vec{v}_i; \vec{v}_o) \end{aligned}$$

$$\begin{aligned} (H2) \quad & \sigma_2.\text{cons}_{\delta_2}(\vec{v}_i) \rightarrow \text{Ok} : (\vec{v}_o, \sigma'_2) \\ \implies & \exists \sigma_{\delta_2}. \sigma = \sigma'_2 \bullet \sigma_{\delta_2} \wedge \sigma_{\delta_2} \models \langle \delta_2 \rangle(\vec{v}_i; \vec{v}_o) \end{aligned}$$

Let δ be a core predicate for the product state model.

Assume (H3) $(\sigma_1, \sigma_2).\text{cons}_{\delta}(\vec{v}_i) \rightarrow \text{Ok} : (\vec{v}_o, (\sigma'_1, \sigma'_2))$

We want to prove the following goal:

$$(G1) \quad \exists \sigma_{\delta}. \sigma = (\sigma'_1, \sigma'_2) \bullet \sigma_{\delta} \wedge \sigma_{\delta} \models \langle \delta \rangle(\vec{v}_i; \vec{v}_o)$$

There are two cases, either $\delta = C1 \delta_1$ or $\delta = C2 \delta_2$. We only consider the first case, the second being analogous

$$(H4) \quad \delta = C1 \delta_1$$

From the definition of the consumer in the product state model, we know that:

$$(H5) \quad \sigma_1.\text{cons}_{\delta_1}(\vec{v}_i) \rightarrow \text{Ok} : (\vec{v}_o, \sigma'_1) \quad (H6) \quad \sigma_2 = \sigma'_2$$

From (H5) and (H1), we can derive that there exist a state σ_{δ_1} such that:

$$(H7) \quad \sigma_1 = \sigma'_1 \bullet \sigma_{\delta_1} \quad (H8) \quad \sigma_{\delta_1} \models \langle \delta_1 \rangle(\vec{v}_i; \vec{v}_o)$$

Moreover, by definition of the unit element in a partial commutative monoid, and (H6), we have that **(H9)** $\sigma_2 = \sigma'_2 \cdot 0$.

Furthermore, by definition of core predicate satisfiability in the product state model and (H8) we have that: **(H10)** $(\sigma_{\delta_1}, 0) \models \langle \delta \rangle (\vec{v}_i; \vec{v}_o)$

Finally, (H7) and (H9), together with the definition of composition in the product state model gives us **(H11)** $(\sigma_1, \sigma_2) = (\sigma'_1, \sigma'_2) \cdot (\sigma_{\delta_1}, 0)$.

The above (H11) and (H10) form our goal (G1). \square

Lemma 8.4 (Product state model: symbolic soundness).

$$\bar{\mathbb{S}}_1 \stackrel{\mathbb{S}}{\sim}_m \mathbb{S}_1 \wedge \bar{\mathbb{S}}_2 \stackrel{\mathbb{S}}{\sim}_m \mathbb{S}_2 \Rightarrow (\bar{\mathbb{S}}_1 \times \bar{\mathbb{S}}_2) \stackrel{\mathbb{S}}{\sim}_m (\mathbb{S}_1 \times \mathbb{S}_2)$$

Proof. The proof is performed in two steps. First, it is to be noted that in the compositional state model, all of the operations can be redefined by explicitly applying the product projection operations **fst** and **snd** lifted to the non-determinism monad, i.e., $\mathbf{fst}^* = \lambda(a, b). \{a\}$ and $\mathbf{snd}^* = \lambda(a, b). \{b\}$.

For example, the producer then becomes:

```
let produce  $\delta \ \sigma \ \vec{v}_i \ \vec{v}_o =$ 
  let*  $\sigma_1 = \mathbf{fst}^* \ \sigma$  in
  let*  $\sigma_2 = \mathbf{snd}^* \ \sigma$  in
  match  $\delta$  with
  (* ... *)
```

A similar transformation can be applied to the symbolic state model, lifting the **fst** and **snd** operations to the symbolic execution monad, i.e., $\overline{\mathbf{fst}} = \lambda(\bar{a}^*, \bar{b}^*). \{\langle \bar{a}^* \mid \mathbf{true} \rangle\}$ and $\overline{\mathbf{snd}} = \lambda(\bar{a}^*, \bar{b}^*). \{\langle \bar{b}^* \mid \mathbf{true} \rangle\}$. We trivially have that $\overline{\mathbf{fst}} \stackrel{\mathbb{S}}{\sim} \mathbf{fst}^*$ and $\overline{\mathbf{snd}} \stackrel{\mathbb{S}}{\sim} \mathbf{snd}^*$, easily obtained from the definition of the interpretation of symbolic states.

After proving the above, the main result is obtained by

1. putting the new definitions of the symbolic and compositional products next to each other;
2. performing case analysis on the kind of action (**A1** α_1 or **A2** α_2) or core predicate (**C1** δ_1 or **C2** δ_2); and
3. applying [Theorem 6.16](#) as many times as required.

\square

E.2 State model of values

We provide the definition of the full state model of values for use in proofs.

```
module  $\underline{\mathbb{V}}$  ( $\tau : \mathbb{T}$ ) = struct

  type  $\Sigma = \tau$ 

  type  $\mathcal{A} = \text{Load} \mid \text{Store}$ 

  let eval_action  $\alpha \ \sigma \ \vec{v} =$ 
    match  $\alpha, \vec{v}$  with
    | Load, []  $\rightarrow$  ok ( $\sigma, \sigma$ )
    | Store, [ $\sigma'$ ]  $\rightarrow$  ok ( $() , \sigma'$ )
    | _  $\rightarrow$  error (InvalidArguments,  $\sigma$ )

end
```

E.3 Exclusive ownership

We first provide the full definition of the exclusive ownership state model.

```

module Exc ( $\tau : \mathbb{T}$ ) = struct

  type  $\Sigma = \tau \mid \perp$ 
  val 0 =  $\perp$ 

  type  $\mathcal{A} = \text{Load} \mid \text{Store}$ 
  let eval_action  $\alpha \ \sigma \ \vec{v} =$ 
    match  $\alpha, \vec{v}, \sigma$  with
    | Load,  $\_$ ,  $\perp \rightarrow$  miss (MissingValue,  $\sigma$ )
    | Load,  $[\ ]$ ,  $\sigma \rightarrow$  ok ( $\sigma$ ,  $\sigma$ )
    | Store,  $[\sigma']$ ,  $\perp \rightarrow$  miss (MissingValue,  $\sigma$ )
    | Store,  $[\sigma']$ ,  $\sigma \rightarrow$  ok ( $\_$ ,  $\sigma'$ )
    |  $\_$ ,  $\_$ ,  $\_ \rightarrow$  fail (InvalidArguments,  $\sigma$ )

  type  $\Delta = \text{Exc}$ 

  let produce  $\sigma \ \text{Exc} \ \vec{v}_i \ \vec{v}_o =$ 
    match  $\sigma, \vec{v}_i, \vec{v}_o$  with
    |  $\perp$ ,  $[\ ]$ ,  $[\ \sigma \ ] \rightarrow \sigma$ 
    |  $\_ \rightarrow$  vanish

  let consume  $\sigma \ \text{Exc} \ \vec{v}_i =$ 
    match  $\sigma, \vec{v}_i$  with
    |  $\sigma$ ,  $[\ ] \rightarrow$  ok ( $[\ \sigma \ ], \perp$ )
    |  $\perp$ ,  $[\ ] \rightarrow$  miss (MissingValue,  $\sigma$ )
    |  $\_$ ,  $\_ \rightarrow$  error (InvalidArguments,  $\sigma$ )

end

```

Relationship between the exclusive ownership state model and the value state model is defined as follows:

$$\begin{aligned} \forall v \in \tau. \quad v &\models_c v \\ \perp &\not\models_c v \end{aligned}$$

In addition, composition is defined as:

$$\begin{aligned} \sigma \cdot \perp &= \sigma \\ \sigma \cdot \sigma' &\text{ undefined otherwise} \end{aligned}$$

And predicate satisfiability as:

$$\sigma \models \langle \text{Exc} \rangle(\cdot; \sigma) \iff \sigma \neq \perp$$

Lemma E.1 (Exclusive ownership: concrete soundness).

$$\text{Exc} \stackrel{\text{C}}{\sim}_{\text{EX}} \mathbb{V}$$

Proof. **Proposition: Frame addition Assume**

$$(H1) \ \sigma.\alpha(\vec{v}) \rightsquigarrow o : (v, \sigma')$$

$$(H2) \ \sigma_f \# \sigma'$$

$$(H3) \ o \neq \text{Miss}$$

To prove (G1) $\sigma \cdot \sigma_f.\alpha(\vec{v}) \rightsquigarrow o : (v, \sigma' \cdot \sigma_f)$

There are three difference cases: Successful load, successful store and invalid arguments. **Case Successful load:**

(H4) $\alpha = \text{Load}$

(H5) $o = \text{Ok}$

(H6) $\sigma' = \sigma = v$

(H7) $\sigma \neq \perp$

From (H2) and (H6) we get (H8) $\sigma \# \sigma_f$.

From (H8) and definition of composition, we get that (H9) $\sigma_f = \perp$.

Finally, from (H9) and (H1) we get the goal (G1).

Case Successful store and invalid arguments: Similar to the above case.

Proposition: Frame subtraction

Assume

(H1) $\sigma \cdot \sigma_f \cdot \alpha(\vec{v}) \rightsquigarrow o : (v, \sigma')$

To prove

$$(G1) \quad \begin{aligned} & \exists o', v', \sigma''. \sigma \cdot \alpha(\vec{v}) \rightsquigarrow o' : (v', \sigma'') \wedge \\ & (o' \neq \text{Miss} \Rightarrow \sigma' = \sigma'' \cdot \sigma_f \wedge o = o' \wedge v = v') \end{aligned}$$

There is no case where execution may vanish, we know that $\exists o', v', \sigma''$ such that

$$(H2) \quad \sigma \cdot \alpha(\vec{v}) \rightsquigarrow o' : (v', \sigma'')$$

Assume now that (H3) $o' \neq \text{Miss}$. There remain 3 possible cases: Successful load, successful store and invalid argument. For a change, we consider the successful store first.

Case Successful store:

(H4) $\alpha = \text{Store}$

(H5) $o = \text{Ok}$

(H6) $v' = ()$

(H7) $\vec{v} = [\sigma']$

Then, $\sigma \cdot \sigma_f$ is defined, so either σ or σ_f is \perp . If σ is \perp , then $o' = \text{Miss}$, which goes against (H3). So (H8) $\sigma_f = \perp$. From (H8), we immediately get the require results.

Case Successful load and invalid arguments: Similar to the above case.

Proposition: Concrete UX-soundness

Assume

(H1) $\sigma \cdot \alpha(\vec{v}) \rightsquigarrow o : (v, \sigma')$

(H2) $\sigma' \models \underline{\sigma}'$

(H3) $o \neq \text{Miss}$

To prove (G1) $\exists \underline{\sigma}. \alpha(\vec{v}) \rightsquigarrow o : (v, \underline{\sigma}') \wedge \sigma \models \underline{\sigma}$

There are three difference cases: Successful load, successful store and invalid arguments. **Case Successful load:**

(H4) $\alpha = \text{Load}$

(H5) $o = \text{Ok}$

(H6) $\sigma' = \sigma = v$

(H7) $\sigma \neq \perp$

It is clear that $\underline{\sigma} = \sigma = v$ satisfies (G1).

The other cases are analogous.

Proposition: Concrete OX-soundness This case is similar to UX-soundness: the full and partial states are the same, and the behaviours only differ for the $\sigma = \perp$ case, in which case everything yields **Miss** and trivially satisfies the soundness result. \square

Lemma E.2 (Exclusive ownership: producers and consumers).

The producer and consumer of the exclusive ownership state model are valid according to the definition of [Definition 5.5](#) and [Definition 5.7](#).

Proof. **Proposition: Validity of the producer** The only case where

the producer of the **Exc** core predicate does not vanish is when the input state is \perp . In this case, there must be no in-parameter and a single out-parameter corresponding to the new output state. This corresponds to all states $\sigma = \perp \cdot \sigma$ such that σ models the core predicate $\langle \text{Exc} \rangle (; \sigma)$. Therefore, the producer is valid

Proposition: Validity of the consumer

Assume

(H1) $\sigma.\text{cons}_{\text{Exc}}(\vec{v}_i) \rightarrow \text{Ok} : (\vec{v}_o, \sigma')$

There is a single successful case of consumption, with the following properties:

(H2) $\sigma \neq \perp$

(H3) $\vec{v}_i = []$

(H4) $\vec{v}_o = [\sigma]$

(H5) $\sigma' = \perp$

We know that $\sigma \cdot \perp \models \langle \text{Exc} \rangle (; \sigma)$, so the consumer is valid.

\square

We provide the full symbolic state model of exclusive ownership for use in proofs.

```

module  $\overline{\text{Exc}}$  ( $\tau : \mathbb{T}$ ) = struct

  type  $\Sigma = \bar{\tau} \mid \perp$ 
  val  $\bar{0} = \perp$ 

  type  $\mathcal{A} = \text{Load} \mid \text{Store}$ 
  let eval_action  $\alpha \ \bar{\sigma} \ \vec{v} =$ 
    match  $\alpha, \vec{v}, \bar{\sigma}$  with
    | Load,  $\_$ ,  $\perp \rightarrow$  miss (MissingValue,  $\bar{\sigma}$ )
    | Load,  $[], \bar{\sigma} \rightarrow$  ok ( $\bar{\sigma}, \bar{\sigma}$ )
    | Store,  $[\bar{\sigma}']$ ,  $\perp \rightarrow$  miss (MissingValue,  $\bar{\sigma}$ )
    | Store,  $[\bar{\sigma}']$ ,  $\bar{\sigma} \rightarrow$  ok ( $()$ ,  $\bar{\sigma}'$ )
    |  $\_$ ,  $\_$ ,  $\_ \rightarrow$  error (InvalidArguments,  $\bar{\sigma}$ )

  type  $\Delta = \text{Exc}$ 

  let produce  $\bar{\sigma} \ \text{Exc} \ \vec{v}_i \ \vec{v}_o =$ 
    match  $\bar{\sigma}, \vec{v}_i, \vec{v}_o$  with
    |  $\perp, [], [\bar{\sigma}] \rightarrow \bar{\sigma}$ 
    |  $\_ \rightarrow$  vanish

  let consume  $\bar{\sigma} \ \text{Exc} \ \vec{v}_i =$ 
    match  $\bar{\sigma}, \vec{v}_i$  with
    |  $\bar{\sigma}, [] \rightarrow$  ok ( $[\bar{\sigma}]$ ,  $\perp$ )
    |  $\perp, [] \rightarrow$  miss (MissingValue,  $\bar{\sigma}$ )
    |  $\_$ ,  $\_ \rightarrow$  error (InvalidArguments,  $\bar{\sigma}$ )
end

```

Furthermore, we define satisfiability of the exclusive ownership symbolic states as:

$$\begin{aligned} \varepsilon, \perp &\models \perp \\ \varepsilon, \bar{\sigma}(\varepsilon) &\models \bar{\sigma} \end{aligned}$$

Lemma E.3 (Exclusive ownership: symbolic soundness).

$$\overline{\text{Exc}} \stackrel{\text{S}}{\sim}_{\text{EX}} \text{Exc}$$

Proof. Again, by performing a case analysis on the kind of action or core predicate, the result is obtained by applying [Theorem 6.16](#) as many times as required. \square

E.4 Agreement state model

We first provide the full definition of the agreement state model.

```

module Ag ( $\tau : \mathbb{T}$ ) = struct

  type  $\Sigma = \tau \mid \perp$ 
  val  $0 = \perp$ 

  type  $\mathcal{A} = \text{Load}$ 
  let eval_action  $\alpha \ \sigma \ \vec{v} =$ 
    match  $\alpha, \vec{v}, \sigma$  with
    | Load,  $\_$ ,  $\perp \rightarrow$  miss (MissingValue,  $\sigma$ )
    | Load,  $[], \sigma \rightarrow$  ok ( $\sigma, \sigma$ )
    |  $\_$ ,  $\_$ ,  $\_ \rightarrow$  error (InvalidArguments,  $\sigma$ )

  type  $\Delta = \text{Ag}$ 

  let produce  $\sigma \ \text{Ag} \ \vec{v}_i \ \vec{v}_o =$ 
    match  $\sigma, \vec{v}_i, \vec{v}_o$  with
    |  $\perp, [], [\sigma'] \rightarrow$ 
       $\sigma'$ 
    |  $\sigma, [], [\sigma'] \rightarrow$ 

```

```

    let* = assume ( $\sigma = \sigma'$ ) in
     $\sigma$ 
    |  $\_ \rightarrow$  vanish

let consume  $\sigma$  Ag  $\vec{v}_i =$ 
  match  $\sigma, \vec{v}_i$  with
  |  $\sigma, [] \rightarrow$  ok ( $[\sigma]$ ,  $\sigma$ )
  |  $\perp, [] \rightarrow$  miss (MissingValue,  $\sigma$ )
  |  $\_, \_ \rightarrow$  error (InvalidArguments,  $\sigma$ )

end

```

Relationship between the agreement state model and the value state model is defined as follows:

$$\forall v \in \tau. \quad v \models_c v$$

$$\perp \not\models_c v$$

In addition, composition is defined as:

$$\sigma \cdot \sigma' = \sigma \iff \sigma' = \perp \vee \sigma = \sigma'$$

$$\sigma \cdot \sigma' \text{ undefined otherwise}$$

And predicate satisfiability as:

$$\sigma \models \langle \text{Ag} \rangle (; \sigma) \iff \sigma \neq \perp$$

Lemma E.4 (Agreement state model: compat).

Ag is compatible with the full state model of values without the Store action.

$$\text{Ag} \stackrel{\text{c}}{\underset{\text{EX}}{\sim}} \mathbb{V}_{\uparrow \text{Load}}$$

Proof. **Proposition: Frame addition**

Assume

$$(H1) \quad \sigma.\alpha(\vec{v}) \rightsquigarrow o : (v, \sigma')$$

$$(H2) \quad \sigma_f \# \sigma'$$

$$(H3) \quad o \neq \text{Miss}$$

To prove (G1) $\sigma \cdot \sigma_f.\alpha(\vec{v}) \rightsquigarrow o : (v, \sigma' \cdot \sigma_f)$

There are two difference cases: Successful load, invalid arguments.

Case Successful load:

$$(H4) \quad \alpha = \text{Load}$$

$$(H5) \quad o = \text{Ok}$$

$$(H6) \quad \sigma' = \sigma = v$$

$$(H7) \quad \sigma \neq \perp$$

From (H2) and (H6) we get **(H8)** $\sigma \# \sigma_f$.

From (H8) and definition of composition, we have either that $\sigma_f = \perp$, or $\sigma_f = \sigma$. In both cases, from (H6), we also have that $\sigma' \cdot \sigma_f = \sigma'$, and therefore the goal (G1) is satisfied.

Case Invalid arguments: Similar to the above case.

Proposition: Frame subtraction

Assume

$$(H1) \quad \sigma \cdot \sigma_f.\alpha(\vec{v}) \rightsquigarrow o : (v, \sigma')$$

To prove

$$(G1) \quad \begin{array}{l} \exists o', v', \sigma''. \sigma.\alpha(\vec{v}) \rightsquigarrow o' : (v', \sigma'') \wedge \\ (o' \neq \text{Miss} \Rightarrow \sigma' = \sigma'' \cdot \sigma_f \wedge o = o' \wedge v = v') \end{array}$$

There is no case where execution may vanish, we know that $\exists o', v', \sigma''$ such that

$$(H2) \quad \sigma.\alpha(\vec{v}) \rightsquigarrow o' : (v', \sigma'')$$

Assume now that **(H3)** $o' \neq \text{Miss}$. There remain 2 possible cases: Successful load and invalid argument.

Case Successful load:

$$(H4) \quad \alpha = \text{Load}$$

$$(H5) \quad o = \text{Ok}$$

$$(H6) \quad v = \sigma$$

$$(H7) \quad \vec{v} = []$$

Then, $\sigma \cdot \sigma_f$ is defined, so either σ_f is \perp or $\sigma_f = \sigma$.

In either case, we get the goal immediately.

Case Invalid arguments: Similar to the above case.

Proposition: concrete soundness The of concrete proof is exactly the same as the exclusive case. \square

Lemma E.5 (Agreement state model: producers and consumers).

The producer and consumer of the agreement state model are valid according to the definition of [Definition 5.5](#) and [Definition 5.7](#).

Proof. **Proposition: Validity of the producer** There are two cases

where the producer of the Ag core predicate does not vanish: either the current state is \perp , or the current state is equal to the input value. In both case, the output state is equal to the input value, and hence the producer is valid.

Proposition: Validity of the consumer

Assume

$$(H1) \quad \sigma.\text{cons}_{\text{Ag}}(\vec{v}_i) \rightarrow \text{Ok} : (\vec{v}_o, \sigma')$$

There is a single successful case of consumption, with the following properties:

$$(H2) \quad \sigma \neq \perp$$

$$(H3) \quad \vec{v}_i = []$$

$$(H4) \quad \vec{v}_o = [\sigma]$$

$$(H5) \quad \sigma' = \sigma$$

We know that $\sigma \cdot \sigma \models \langle \text{Ag} \rangle(\sigma)$, so the consumer is valid. \square

We provide the full agreement symbolic state model for use in proofs.

```
module  $\overline{\text{Ag}}$  ( $\tau : \mathbb{T}$ ) = struct

  type  $\Sigma = \bar{\tau} \mid \perp$ 
  val  $\bar{0} = \perp$ 

  type  $\mathcal{A} = \text{Load}$ 
  let eval_action  $\alpha \bar{\sigma} \vec{v} =$ 
    match  $\alpha, \vec{v}, \bar{\sigma}$  with
    | Load,  $\_$ ,  $\perp \rightarrow$  miss (MissingValue,  $\bar{\sigma}$ )
    | Load,  $[\_]$ ,  $\bar{\sigma} \rightarrow$  ok ( $\bar{\sigma}, \bar{\sigma}$ )
    |  $\_$ ,  $\_$ ,  $\_ \rightarrow$  error (InvalidArguments,  $\bar{\sigma}$ )

  type  $\Delta = \text{Ag}$ 

  let produce  $\bar{\sigma} \text{Exc } \vec{v}_i \vec{v}_o =$ 
    match  $\bar{\sigma}, \vec{v}_i, \vec{v}_o$  with
    |  $\perp, [\_]$ ,  $[\bar{\sigma}'] \rightarrow \bar{\sigma}'$ 
    |  $\bar{\sigma}, [\_]$ ,  $[\bar{\sigma}'] \rightarrow$ 
      let* = assume ( $\bar{\sigma} = \bar{\sigma}'$ ) in
       $\bar{\sigma}$ 
    |  $\_ \rightarrow$  vanish

  let consume  $\bar{\sigma} \text{Exc } \vec{v}_i =$ 
    match  $\bar{\sigma}, \vec{v}_i$  with
    |  $\bar{\sigma}, [\_] \rightarrow$  ok ( $[\bar{\sigma}], \bar{\sigma}$ )
    |  $\perp, [\_] \rightarrow$  miss (MissingValue,  $\bar{\sigma}$ )
    |  $\_$ ,  $\_ \rightarrow$  error (InvalidArguments,  $\bar{\sigma}$ )
end
```

Furthermore, we define satisfiability of the agreement symbolic states as:

$$\begin{aligned} \varepsilon, \perp &\models \perp \\ \varepsilon, \bar{\sigma}(\varepsilon) &\models \bar{\sigma} \end{aligned}$$

Lemma E.6 (Agreement state model: symbolic soundness).

$$\overline{\text{Ag}} \stackrel{\text{S}}{\underset{\text{EX}}{\sim}} \text{Ag}$$

Proof. Again, by performing a case analysis on the kind of action or core predicate, the result is obtained by applying [Theorem 6.16](#) as many times as required. \square

E.5 Fractional state model

We first provide the definition of the fractional state model.

```
module Frac ( $\tau : \mathbb{T}$ ) = struct

  type  $\Sigma = \tau \times (0, 1] \mid \perp$ 
  val  $0 = \perp$ 

  type  $\mathcal{A} = \text{Load} \mid \text{Store}$ 
  let eval_action  $\alpha \sigma \vec{v} =$ 
    match  $\alpha, \vec{v}, \sigma$  with
    | Load,  $\_$ ,  $\perp \rightarrow$  miss (MissingValue,  $\sigma$ )
    | Load,  $[\_]$ ,  $(v, q) \rightarrow$  ok ( $v, \sigma$ )
    | Store,  $\_$ ,  $\perp \rightarrow$  miss (MissingValue,  $\sigma$ )
    | Store,  $[v']$ ,  $(v, q) \rightarrow$ 
```

```

    if q = 1 then ok ((), (v', q)) else
    miss (MissingFrac, σ)
  | _, _, _ → error (InvalidArguments, σ)

type Δ = Frac

let produce σ Frac  $\vec{v}_i$   $\vec{v}_o$  =
  match σ,  $\vec{v}_i$ ,  $\vec{v}_o$  with
  | ⊥, [ q ], [ v ] →
    let* = assume (0 < q ≤ 1) in
    return (v, q)
  | (v, q), [ q' ], [ v' ] →
    let* = assume (q < 0 ∧ q + q' ≤ 1 ∧ v = v') in
    return (v, q + q')
  | _ → vanish

let consume σ Frac  $\vec{v}_i$  =
  match σ,  $\vec{v}_i$  with
  | (v, q), [ q' ] →
    if q' = q then ok (v, ⊥)
    else if q' < q then ok (v, (v, q - q'))
    else miss (MissingFrac, σ)
  | ⊥, [] → miss (MissingValue, σ)
  | _, _ → error (InvalidArguments, σ)

end

```

Relationship between the fractional state model and the value state model is defined as follows:

$$\forall v \in \tau. \quad \begin{array}{c} (v, q) \\ \perp \end{array} \models_c v \iff q = 1$$

In addition, composition is defined as:

$$\begin{aligned} (v, q) \cdot (v', q') &= (v, q + q') \iff 0 < q + q' \leq 1 \wedge v = v' \\ \sigma \cdot \perp &= \sigma \\ \sigma \cdot \sigma' &\text{ undefined otherwise} \end{aligned}$$

And predicate satisfiability as:

$$(v, q) \models \langle \text{Frac} \rangle(q; v) \iff 0 < q \leq 1$$

Lemma E.7 (Fractional state model: compat).

Frac is compatible with the full state model of values without the **Store** action.

$$\text{Frac} \underset{\text{EX}}{\overset{\text{C}}{\sim}} \mathbb{V}$$

Proof. **Proposition: Frame addition**

Assume

$$(H1) \quad \sigma.f.\alpha(\vec{v}) \rightsquigarrow o : (v, \sigma')$$

$$(H2) \quad \sigma_f \# \sigma'$$

$$(H3) \quad o \neq \text{Miss}$$

To prove (G1) $\sigma \cdot \sigma_f.\alpha(\vec{v}) \rightsquigarrow o : (v, \sigma' \cdot \sigma_f)$

There are three difference cases: Successful load, Successful store and invalid arguments.

Case Successful load:

(H4) $\alpha = \text{Load}$

(H5) $o = \text{Ok}$

(H6) $\sigma' = \sigma = (v, q)$

From (H2) and (H6) we get (H7) $\sigma \# \sigma_f$.

From (H7) and definition of composition, we have either that $\sigma_f = \perp$, in which case the result is trivially obtained, or $\sigma_f = (v, q')$, in which case $\sigma \cdot \sigma_f = \sigma' \cdot \sigma_f$ and the value is unchanged, so the goal is satisfied.

Case Successful store:

(H8) $\alpha = \text{Store}$

(H9) $o = \text{Ok}$

(H10) $\vec{v} = [v']$

(H11) $\sigma = \sigma' = (v, q)$

If $q < 1$ then the result is **Miss**, which contradicts (H9), so we have that $q = 1$. In this case, it must be that $\sigma_f = \perp$, otherwise it could not be composed with σ' . From there, the goal is immediately satisfied.

Case Invalid arguments: Similar to the above case.

Proposition: Frame subtraction

Assume

(H1) $\sigma \cdot \sigma_f \cdot \alpha(\vec{v}) \rightsquigarrow o : (v, \sigma')$

To prove

$$(G1) \quad \begin{aligned} &\exists o', v', \sigma''. \sigma \cdot \alpha(\vec{v}) \rightsquigarrow o' : (v', \sigma'') \wedge \\ &(o' \neq \text{Miss} \Rightarrow \sigma' = \sigma'' \cdot \sigma_f \wedge o = o' \wedge v = v') \end{aligned}$$

There is no case where execution may vanish, we know that $\exists o', v', \sigma''$ such that

$$(H2) \quad \sigma \cdot \alpha(\vec{v}) \rightsquigarrow o' : (v', \sigma'')$$

Assume now that (H3) $o' \neq \text{Miss}$. There remain 2 possible cases: Successful load, successful store and invalid argument.

Case Successful load:

(H4) $\alpha = \text{Load}$

(H5) $o = \text{Ok}$

(H6) $\sigma = (v, q)$

(H7) $\vec{v} = []$

Then, $\sigma \cdot \sigma_f$ is defined, so either σ_f is \perp or $\sigma_f = (v, q')$ and $q + q' < 1$. In both cases, the state is not modified by load, and the outcome is still $\sigma \cdot \sigma_f$, so the goal is satisfied.

Case Successful store:

(H8) $\alpha = \text{Store}$

(H9) $o = \text{Ok}$

(H10) $\sigma = (v, 1)$

(H11) $\vec{v} = [v']$

Since $\sigma \bullet \sigma_f$ is defined, and the fraction of σ is 1, it must be that $\sigma_f = \perp$, and from there, the goal is trivially obtained.

Case Invalid arguments: Similar to the above case.

Proposition: Concrete soundness It can be trivially checked that if $\sigma = (v, 1)$, then `eval_action` behaves identically in $\underline{\mathbb{V}}$ and `Frac`. Then soundness comes naturally. \square

Lemma E.8 (Fractional state model: producers and consumers).

The producer and consumer of the fractional state model are valid according to the definition of [Definition 5.5](#) and [Definition 5.7](#).

Proof. **Proposition: Validity of the producer** There are two cases

where the producer of the `Frac` core predicate does not vanish: either the current state is \perp , or the current state has a value equal to the out-parameter of the predicate, and the fractions add up to less than 1. In both case, it coincides with the definition of the core predicate satisfiability and composition. Therefore, the producer is valid.

Proposition: Validity of the consumer

Assume

(H1) $\sigma.\text{cons}_{\text{Frac}}(\vec{v}_i) \rightarrow \text{Ok} : (\vec{v}_o, \sigma')$

There are two cases of valid consumption. Either the fraction was entirely consumed, in which case the outcome is `bot`, or the fraction was only partially consumed.

Case Fully consumed:

(H2) $\sigma = (v, q)$

(H3) $\vec{v}_i = [q]$

(H4) $\vec{v}_o = [v]$

(H5) $\sigma' = \perp$

It can be trivially checked that $(v, q) \models \langle \text{Frac} \rangle(q; v)$, and $(v, q) = \perp \bullet (v, q)$ so the consumer is valid.

Case Partially consumed:

(H6) $\sigma = (v, q)$

(H7) $\vec{v}_i = [q']$

$$(H8) \quad 0 < q' < q$$

$$(H9) \quad \vec{v}_o = [v]$$

$$(H10) \quad \sigma' = (v, q - q')$$

Again, in this case, it can be checked that $(v, q - q') \cdot (v, q') = (v, q)$, and $(v, q') \models \langle \text{Frac} \rangle(q'; v)$, so the consumer is valid. \square

We provide the fractional symbolic state model for use in proofs.

```

module  $\overline{\text{Frac}}$  ( $\tau : \mathbb{T}$ ) = struct

  type  $\Sigma = (\tau \times \text{sym}\{\text{nats}\}) \mid \perp$ 
  val  $\bar{0} = \perp$ 

  type  $\mathcal{A} = \text{Load} \mid \text{Store}$ 
  let eval_action  $\alpha \ \bar{\sigma} \ \vec{v} =$ 
    match  $\alpha, \vec{v}, \bar{\sigma}$  with
    | Load,  $\_$ ,  $\perp \rightarrow$  miss (MissingValue,  $\bar{\sigma}$ )
    | Load,  $[\_]$ ,  $(\bar{v}, \bar{q}) \rightarrow$  ok  $(\bar{v}, \bar{\sigma})$ 
    | Store,  $\_$ ,  $\perp \rightarrow$  miss (MissingValue,  $\bar{\sigma}$ )
    | Store,  $[\vec{v}']$ ,  $(\bar{v}, \bar{q}) \rightarrow$ 
      if%sat  $\text{sym}\{q\} = 1$  then ok  $((), (\bar{v}', \bar{q}))$  else
      miss (MissingFrac,  $\bar{\sigma}$ )
    |  $\_$ ,  $\_$ ,  $\_ \rightarrow$  error (InvalidArguments,  $\bar{\sigma}$ )

  type  $\Delta = \text{Frac}$ 

  let produce  $\bar{\sigma} \ \text{Frac} \ \vec{v}_i \ \vec{v}_o =$ 
    match  $\bar{\sigma}, \vec{v}_i, \vec{v}_o$  with
    |  $\perp, [\bar{q}], [\bar{v}] \rightarrow$ 
      let* = assume  $(0 < \bar{q} \leq 1)$  in
      return  $(\bar{v}, \bar{q})$ 
    |  $(\bar{v}, \bar{q}), [\bar{q}'], [\bar{v}'] \rightarrow$ 
      let* = assume  $(\bar{q} < 0 \wedge \bar{q} + \bar{q}' \leq 1 \wedge \bar{v} = \bar{v}')$  in
      return  $(\bar{v}, \bar{q} + \bar{q}')$ 
    |  $\_ \rightarrow$  vanish

  let consume  $\bar{\sigma} \ \text{Frac} \ \vec{v}_i =$ 
    match  $\bar{\sigma}, \vec{v}_i$  with
    |  $(\bar{v}, \bar{q}), [\bar{q}'] \rightarrow$ 
      if  $\bar{q}' = \bar{q}$  then ok  $(\bar{v}, \perp)$ 
      else if  $\bar{q}' < \bar{q}$  then ok  $(\bar{v}, (\bar{v}, \bar{q} - \bar{q}'))$ 
      else miss (MissingFrac,  $\bar{\sigma}$ )
    |  $\perp, [\_] \rightarrow$  miss (MissingValue,  $\bar{\sigma}$ )
    |  $\_$ ,  $\_ \rightarrow$  error (InvalidArguments,  $\bar{\sigma}$ )
end

```

Furthermore, we define satisfiability of the exclusive ownership symbolic states as:

$$\begin{aligned} \varepsilon, \perp &\models \perp \\ \varepsilon, (\sigma, q) &\models (\bar{\sigma}(\varepsilon), \bar{q}(\varepsilon)) \end{aligned}$$

Lemma E.9 (Fraction state model: symbolic soundness).

$$\overline{\text{Frac}} \stackrel{\text{S}}{\underset{\text{EX}}{\sim}} \text{Frac}$$

Proof. Again, by performing a case analysis on the kind of action or core predicate, the result is obtained by applying [Theorem 6.16](#) as many times as required. \square

E.6 Partial maps

We first provide the *full* partial map state mode, then the compositional partial map.

```

module PMap (I: T) (S : Full_state_model) = struct

  type Σ = I  $\xrightarrow{fin}$  S.Σ
  type A = S.A

  let eval_action σ α v̄ =
    match v̄ with
    | i :: v̄' → (
      match σ(i) with
      | Some σc →
        let* (v, σ'c) = S.eval_action σc α v̄' in
        ok (v, σ [i ← σ'c])
      | None → error (InvalidAccess, σ)
    )
    | _ → error (InvalidArguments, σ)
end

module PMap (I: τ) (S: Compositional_state_model) = struct

  type Σ = I  $\xrightarrow{fin}$  S.Σ
  type A = S.A

  let eval_action σ α v̄ =
    match v̄ with
    | i :: v̄' → (
      match σ(i) with
      | Some σc →
        let* (v, σ'c) = S.eval_action σc α v̄' in
        if σ'c = S.0 then ok (v, σ \ {i})
        else ok (v, σ [i ← σ'c])
      | None →
        let* (v, σ'c) = S.eval_action S.0 α v̄' in
        if σ'c = S.0 then ok (v, σ)
        else ok (v, σ [i ← σ'c])
    )
    | _ → error (InvalidArguments, σ)

  type Δ = S.Δ

  let produce σ δ v̄i v̄o =
    match v̄i, v̄o with
    | i :: v̄'i, v̄o → (
      match σ(i) with
      | Some σc →
        let* σ'c = S.produce σc δ v̄'i v̄o in
        if σ'c = S.0 then ok (σ \ {i})
        else ok (σ [i ← σ'c])
      | None →
        let* σ'c = S.produce S.0 δ v̄'i v̄o in
        if σ'c = S.0 then ok (σ)
        else ok (σ [i ← σ'c])
    )
    | _, _ → vanish

  let consume σ δ v̄i =
    match v̄i with
    | i :: v̄'i → (
      match σ(i) with
      | Some σc →
        let* (v̄o, σ'c) = S.consume σc δ v̄'i in
        if σ'c = S.0 then ok (v̄o, σ \ {i})
        else ok (v̄o, σ [i ← σ'c])
      | None →

```

```

    let* ( $\vec{v}_o, \sigma'_c$ ) =  $\mathbb{S}.$ consume  $\mathbb{S}.0 \ \delta \ \vec{v}'_i$  in
    if  $\sigma'_c = \mathbb{S}.0$  then ok ( $\vec{v}_o, \sigma$ )
    else ok ( $\vec{v}_o, \sigma[i \leftarrow \sigma'_c]$ )
  )
  | _, _  $\rightarrow$  lfail (InvalidArguments,  $\sigma$ )

end

```

Note: Admittedly, some of the code above could be factored out for the handling of the case where the return value of the sub-state model is $\mathbb{S}.0$.

Composition is defined as:

$$\sigma_1 \cdot \sigma_2 = \lambda i. \begin{cases} \sigma(i) & \text{if } i \notin \text{dom}(\sigma') \\ \sigma'(i) & \text{if } i \notin \text{dom}(\sigma) \\ \sigma(i) \cdot \sigma'(i) & \text{if } i \in \text{dom}(\sigma) \cap \text{dom}(\sigma') \wedge \sigma(i) \# \sigma'(i) \\ \text{undefined} & \text{otherwise} \end{cases}$$

Lemma E.10 (Partial map state model: concrete soundness).

$$\mathbb{S} \stackrel{\mathcal{C}}{\underset{m}{\simeq}} \underline{\mathbb{S}} \implies \text{PMap}(I, \mathbb{S}) \stackrel{\mathcal{C}}{\underset{m}{\simeq}} \text{PMap}(I, \underline{\mathbb{S}})$$

Proof. **Proposition: propagation of m -soundness** The result is

trivially propagated from the concrete state model.

Proposition: Frame addition

Assume $\forall \sigma_c \in \mathbb{S}$.

$$(H1) \quad \sigma.\alpha_c(\vec{v}) \rightsquigarrow o : (v, \sigma'_c) \wedge \sigma'_c \# \sigma_c^f \wedge o \neq \text{Miss} \implies \\ \sigma \cdot \sigma_c^f.\alpha_c(\vec{v}) \rightsquigarrow o : (v, \sigma'_c \cdot \sigma_c^f)$$

In addition:

$$(H2) \quad \sigma.\alpha(\vec{v}) \rightsquigarrow o : (v, \sigma')$$

$$(H3) \quad \sigma_f \# \sigma'$$

$$(H4) \quad o \neq \text{Miss}$$

If $\vec{v} = []$, then the result is trivially obtained. We consider the case where $\vec{v} = i :: \vec{v}'$. There are two cases to consider, either σ_f has a binding for i , or it does not. If it does not, since the action evaluation can only modify the binding for i in σ , composition will not interfere with this binding i.e.

$$i \notin \text{dom}(\sigma_f) \wedge \sigma_f \# \sigma' \implies \\ (\sigma[i \leftarrow \sigma_c]) \cdot (\sigma_f) = ((\sigma \cdot \sigma_f)[i \leftarrow \sigma_c])$$

The goal is then satisfied. The other case is when $\sigma_f(i) = \sigma_c^f$. There are then four more subcases,

- $i \in \text{dom}(\sigma)$ and $i \in \text{dom}(\sigma')$;
- $i \in \text{dom}(\sigma)$ and $i \notin \text{dom}(\sigma')$;
- $i \notin \text{dom}(\sigma)$ and $i \in \text{dom}(\sigma')$;
- $i \notin \text{dom}(\sigma)$ and $i \notin \text{dom}(\sigma')$.

We handle the first and second case, the other two are similar.

Case $i \in \text{dom}(\sigma)$ and $i \in \text{dom}(\sigma')$:

In this case:

$$(H5) \quad i \in \text{dom}(\sigma) \text{ and } i \in \text{dom}(\sigma')$$

$$(H6) \quad \sigma(i) = \sigma_c$$

$$(H7) \quad \sigma_c \cdot \alpha(\vec{v}') \rightsquigarrow o : (v, \sigma'_c) \text{ with the same outcome } o \neq \text{Miss}.$$

$$(H8) \quad \sigma' = (\sigma [i \leftarrow \sigma'_c])$$

From (H3), (H5) and the definition of composition, we have $\sigma_c^f \# \sigma'_c$.

In turn, from (H7) and (H1), we get

$$(H9) \quad (\sigma_c \cdot \sigma_c^f) \cdot \alpha(\vec{v}') \rightsquigarrow o : (v, \sigma'_c \cdot \sigma_c^f)$$

By definition of the action on the partial map state model, we also get the goal that $\sigma \cdot \sigma^f \cdot \alpha(\vec{v}) \rightsquigarrow o : (v, \sigma' \cdot \sigma^f)$, since every other binding is left untouched.

Case $i \in \text{dom}(\sigma)$ and $i \notin \text{dom}(\sigma')$: In this case, necessarily, the action evaluation yields the 0 of the codomain and the binding gets removed.

$$(H10) \quad i \in \text{dom}(\sigma) \text{ and } i \notin \text{dom}(\sigma')$$

$$(H11) \quad \sigma(i) = \sigma_c$$

$$(H12) \quad \sigma_c \cdot \alpha(\vec{v}') \rightsquigarrow o : (v, \mathbb{S}.0) \text{ with the same outcome } o \neq \text{Miss}.$$

$$(H13) \quad \sigma' = \sigma \setminus \{i\}$$

Because $\mathbb{S}.0$ is necessarily disjoint from any other state, and $o \neq \text{Miss}$, we can apply (H1) and learn that σ_c is disjoint from σ_c^f , and get a transition from $\sigma_c \cdot \sigma_c^f$ tp σ_c^f hence yielding the goal.

Proposition: Frame subtraction

Assume

$$(H1) \quad \begin{aligned} &(\sigma_c \cdot \sigma_c^f) \cdot \alpha_c(\vec{v}) \rightsquigarrow o : (v, \sigma'_c) \implies (\exists o', v', \sigma'' \\ &\sigma_c \cdot \alpha(\vec{v}) \rightsquigarrow o' : (v', \sigma''_c) \wedge \\ &(o' \neq \text{Miss} \Rightarrow \sigma'_c = \sigma''_c \cdot \sigma_c^f \wedge o = o' \wedge v = v')) \end{aligned}$$

In addition: **Assume**

$$(H2) \quad (\sigma \cdot \sigma^f) \cdot \alpha_c(\vec{v}) \rightsquigarrow o : (v, \sigma')$$

Again, we only consider the case where \vec{v} has at least one integer argument i , i.e. $\vec{v} = i :: \vec{v}'$, the other case is trivial. If $i \notin \text{dom}(\sigma_f)$, then σ_f does not interfere with the binding at i , and the goal is satisfied. Otherwise, we denote $\sigma_c^f = \sigma_f(i)$, and there are four cases to consider:

- $i \in \text{dom}(\sigma)$ and $i \in \text{dom}(\sigma')$;
- $i \in \text{dom}(\sigma)$ and $i \notin \text{dom}(\sigma')$;
- $i \notin \text{dom}(\sigma)$ and $i \in \text{dom}(\sigma')$;
- $i \notin \text{dom}(\sigma)$ and $i \notin \text{dom}(\sigma')$.

Since there are no vanishing transitions, we know that there exists o', v', σ'' such that

$$(H3) \quad \sigma.\alpha(\vec{v}) \rightsquigarrow o' : (v', \sigma'')$$

Furthermore, assume that (H4) $o' \neq \text{Miss}$. The case where the list of values is empty is trivial, so we consider the case where $\vec{v} = i :: \vec{v}'$.

Either σ_f has a binding for i , or it does not. If it does not, then the result is trivially obtained, using the same property used for frame addition. Otherwise, we have (H5) $\sigma_f(i) = \sigma_c^f$. Since the outcome is not miss, it must also be that $i \in \text{dom}(h)$, as otherwise it would need to be in the domain and that would clash with σ_f . From there, we have

$$(H6) \quad h(i) = \sigma_c$$

$$(H7) \quad (\sigma_c \cdot \sigma_c^f).\alpha(\vec{v}) \rightsquigarrow o' : (v', \sigma'_c)$$

$$(H8) \quad \sigma' = (h[i \leftarrow \sigma'_c], d^\perp)$$

From (H7), (H1) and (H4) we have that

$$(H9) \quad \sigma_c.\alpha(\vec{v}') \rightsquigarrow o' : (v', \sigma''_c)$$

$$(H10) \quad \sigma'_c = \sigma''_c \cdot \sigma_c^f$$

These two last properties put together with the definition of composition of maps give us the goal. \square

Lemma E.11 (Partial map state model: producers and consumers).
If the producer and consumer of the codomain \mathbb{S} are valid, then the producer and consumer of the partial map state model $\text{PMap}(I, \mathbb{S})$ are valid.

Proof.

Proposition: Validity of the producer

Assume

(H1) The producer of \mathbb{S} is valid.

There are two cases, core predicate is δ with in-parameters $\vec{v}_i = i :: \vec{v}'_i$ and the case with an invalid number of arguments. The latter case is trivial and not considered here.

First, recall that the only case for satisfying the core predicate δ is

$$[i \mapsto \sigma_c] \models \langle \delta \rangle(i :: \vec{v}'_i; \vec{v}_o) \iff \sigma_c \models \langle \delta \rangle(\vec{v}'_i; \vec{v}_o)$$

Such a state can only be composed with σ if

- $i \notin \text{dom}(\sigma)$, i.e. the binding i is entirely missing from the state; or
- $i \in \text{dom}(\sigma)$ and σ_c can be composed with the current binding at i , otherwise it should vanish.

In the first case, the goal holds because $\mathbb{S}.\text{produce}$ is in charge of producing all states that can be composed, and the right arguments are passed (\vec{v}'_i and \vec{v}_o).

The second case is where the binding is not in the map. In this case, since 0 is the identity, all states satisfying $\langle \delta \rangle(\vec{v}'_i; \vec{v}_o)$ can be obtained calling the producer from the state $\mathbb{S}.0$, in which case the binding is updated.

Since all other cases are vanishing, the producer is valid.

Proposition: Validity of the consumer

Again, we ignore the case of invalid arguments which is trivial, since it returns `Lfail`.

Assume

$$(H1) \quad \vec{v}_i = i :: \vec{v}'_i$$

$$(H2) \quad \sigma.\text{cons}_\delta(\vec{v}_i) \rightarrow \text{Ok} : (\vec{v}_o, \sigma')$$

From (H2) and the definition of the consumer we have two cases to consider: either the binding for i is in the heap, or it is not. We only consider the first case, the other is analogous with σ_c replaced with 0.

$$(H3) \quad \sigma(i) = \sigma_c$$

$$(H4) \quad \sigma_c.\text{cons}_\delta(\vec{v}'_i) \rightarrow \text{Ok} : (\vec{v}_o, \sigma'_c)$$

From (H4) and the fact that the consumer of \mathbb{S} is valid, we get that

$$(H5) \quad \sigma_c = \sigma'_c \cdot \sigma_c^\delta \text{ where } (H6) \quad \sigma_c^\delta \models \langle \delta \rangle (\vec{v}'_i; \vec{v}_o).$$

There are two cases of succesful consumption, either $\sigma_c = \mathbb{S}.0$ or $\sigma_c \neq \mathbb{S}.0$.

Case Fully consumed:

$$(H7) \quad \sigma'_c = \mathbb{S}.0$$

$$(H8) \quad \sigma' = \sigma \setminus \{i\}$$

Given (H7) and (H5), we have that $\sigma_c = \sigma_c^\delta$ and thererfore, given $\sigma = \sigma' [i \leftarrow \sigma_c]$, which will add the binding to the heap, and validates our goal.

Case Partially consumed:

$$(H9) \quad \sigma'_c \neq \mathbb{S}.0$$

$$(H10) \quad \sigma' = \sigma [i \leftarrow \sigma'_c]$$

Given (H9) and (H5), we have that $\sigma_c = \sigma_c^\delta$ and thererfore, given $\sigma = \sigma' [i \leftarrow \sigma_c^\delta]$, where the binding for i will the composition of σ_c^δ and σ'_c , which is σ_c , and validates our goal. \square

We now give the definition of the symbolic partial map state model transformer.

```

let symbolic_map_get  $\bar{\sigma} \ \bar{i} =$ 
match  $\bar{\sigma}$  with
|  $\emptyset \rightarrow$  ok None
|  $[\bar{i}' \mapsto \sigma_c] \uplus \bar{\sigma}' \rightarrow$ 
  if%sat  $\bar{i} = \bar{i}'$  then
    ok  $\sigma_c$ 
  else
    symbolic_map_get  $\bar{\sigma}' \ \bar{i}$ 

```

```

module  $\overline{\text{PMap}}$  (I:  $\tau$ ) ( $\bar{\mathbb{S}}$ : Symbolic_state_model) = struct
  type  $\bar{\Sigma} = \bar{I} \xrightarrow{fm} \bar{\mathbb{S}}.\bar{\Sigma}$ 

```

```

type  $\mathcal{A} = \overline{\mathbb{S}}.\mathcal{A}$ 

let  $\overline{\text{eval\_action}} \ \overline{\sigma} \ \alpha \ \vec{v} =$ 
  match  $\vec{v}$  with
  |  $\bar{i} :: \vec{v}' \rightarrow$  (
    let*  $\text{opt\_i} = \text{symbolic\_map\_get} \ \overline{\sigma} \ \bar{i}$  in
    match  $\text{opt\_i}$  with
    | Some  $\overline{\sigma}_c \rightarrow$ 
      let*  $(\overline{v}, \overline{\sigma}'_c) = \overline{\mathbb{S}}.\overline{\text{eval\_action}} \ \overline{\sigma}_c \ \alpha \ \vec{v}'$  in
      if%sat  $\overline{\sigma}'_c = \overline{\mathbb{S}}.\overline{0}$  then ok  $(\overline{v}, \overline{\sigma} \setminus \{\bar{i}\})$ 
      else ok  $(\overline{v}, \overline{\sigma} [\bar{i} \leftarrow \overline{\sigma}'_c])$ 
    | None  $\rightarrow$ 
      let*  $(\overline{v}, \overline{\sigma}'_c) = \overline{\mathbb{S}}.\overline{\text{eval\_action}} \ \overline{\mathbb{S}}.\overline{0} \ \alpha \ \vec{v}'$  in
      if%sat  $\overline{\sigma}'_c = \overline{\mathbb{S}}.\overline{0}$  then ok  $(\overline{v}, \overline{\sigma})$ 
      else ok  $(\overline{v}, \overline{\sigma} [\bar{i} \leftarrow \overline{\sigma}'_c])$ 
  )
  | _  $\rightarrow$  error (InvalidArguments,  $\overline{\sigma}$ )

type  $\Delta = \overline{\mathbb{S}}.\Delta$ 

let produce  $\overline{\sigma} \ \delta \ \vec{v}_i \ \vec{v}_o =$ 
  match  $\vec{v}_i$  with
  |  $\bar{i} :: \vec{v}'_i \rightarrow$  (
    let*  $\text{opt\_i} = \text{symbolic\_map\_get} \ \overline{\sigma} \ \bar{i}$  in
    match  $\text{opt\_i}$  with
    | Some  $\overline{\sigma}_c \rightarrow$ 
      let*  $\overline{\sigma}'_c = \overline{\mathbb{S}}.\text{produce} \ \overline{\sigma}_c \ \delta \ \vec{v}'_i \ \vec{v}_o$  in
      if%sat  $\overline{\sigma}'_c = \overline{\mathbb{S}}.\overline{0}$  then ok  $(\overline{\sigma} \setminus \{\bar{i}\})$ 
      else ok  $(\overline{\sigma} [\bar{i} \leftarrow \overline{\sigma}'_c])$ 
    | None  $\rightarrow$ 
      let*  $\overline{\sigma}'_c = \overline{\mathbb{S}}.\text{produce} \ \overline{\mathbb{S}}.\overline{0} \ \delta \ \vec{v}'_i \ \vec{v}_o$  in
      if  $\overline{\sigma}'_c = \overline{\mathbb{S}}.\overline{0}$  then ok  $\overline{\sigma}$ 
      else ok  $(\overline{\sigma} [\bar{i} \leftarrow \overline{\sigma}'_c])$ 
  )
  | _  $\rightarrow$  vanish

let consume  $\overline{\sigma} \ \delta \ \vec{v}_i =$ 
  match  $\vec{v}_i$  with
  |  $\bar{i} :: \vec{v}'_i \rightarrow$  (
    let*  $\text{opt\_i} = \text{symbolic\_map\_get} \ \overline{\sigma} \ \bar{i}$  in
    match  $\text{opt\_i}$  with
    | Some  $\overline{\sigma}_c \rightarrow$ 
      let*  $\overline{\sigma}'_c = \overline{\mathbb{S}}.\text{consume} \ \overline{\sigma}_c \ \delta \ \vec{v}'_i$  in
      if  $\overline{\sigma}'_c = \overline{\mathbb{S}}.\overline{0}$  then ok  $(\overline{\sigma} \setminus \{\bar{i}, \bar{d}^\perp\})$ 
      else ok  $(\overline{\sigma} [\bar{i} \leftarrow \overline{\sigma}'_c], \bar{d}^\perp)$ 
    | None  $\rightarrow$ 
      let*  $\overline{\sigma}'_c = \overline{\mathbb{S}}.\text{consume} \ \overline{\mathbb{S}}.\overline{0} \ \delta \ \vec{v}'_i$  in
      if  $\overline{\sigma}'_c = \overline{\mathbb{S}}.\overline{0}$  then ok  $(\overline{\sigma})$ 
      else ok  $(\overline{\sigma} [\bar{i} \leftarrow \overline{\sigma}'_c])$ 
  )
  | _  $\rightarrow$  lfail (InvalidArguments,  $\overline{\sigma}$ )

end

```

Lemma E.12 (Partial map state model: symbolic soundness).

$$\overline{\mathbb{S}} \stackrel{\mathbb{S}}{\sim}_m \mathbb{S} \implies \overline{\text{PMap}}(I, \overline{\mathbb{S}}) \stackrel{\mathbb{S}}{\sim}_m \text{PMap}(I, \mathbb{S})$$

Proof. Again the proof is performed by case analysis and lifting using the symbolic execution monad. There is one function call that needs a bit more attention: we need to prove that `symbolic_map_get` is indeed a sound abstraction of the map lookup function. This can be again

trivially proven by noting that map access can be written

```
let  $\sigma(i)$  =
  match  $\sigma$  with
  |  $\emptyset \rightarrow$  None
  |  $[i' \mapsto \sigma_c] \uplus \sigma' \rightarrow$ 
    if  $i = i'$  then
      ok  $\sigma_c$ 
    else
       $\sigma'(i)$ 
```

This is lifted directly using our symbolic execution monad to obtain `symbolic_map_get`, and, in turn, we get our result. \square

E.7 Freeable state model

```
module Freeable ( $\mathbb{S}$ : Full_state_model) = struct
```

```
  type  $\underline{\Sigma}$  = S of  $\mathbb{S}.\underline{\Sigma}$  |  $\emptyset$ 
```

```
  type  $\mathcal{A}$  = A of  $\mathbb{S}.\mathcal{A}$  | Free
```

```
  let eval_action  $\underline{\sigma}$   $\alpha$   $\vec{v}$  =
    match  $\alpha$ ,  $\vec{v}$  with
    | Free, []  $\rightarrow$  (
      match  $\underline{\sigma}$  with
      |  $\emptyset \rightarrow$  error (DoubleFree,  $\underline{\sigma}$ )
      | S  $\underline{\sigma} \rightarrow$  ok ((),  $\emptyset$ )
    )
    | A  $\alpha$ , _  $\rightarrow$  (
      match  $\underline{\sigma}$  with
      |  $\emptyset \rightarrow$  error (UseAfterFree,  $\underline{\sigma}$ )
      | S  $\underline{\sigma} \rightarrow$  (
        let* ( $v, \sigma'$ ) =  $\mathbb{S}.\text{eval\_action } \underline{\sigma} \alpha \vec{v}$  in
        ok ( $v$ , S  $\underline{\sigma}'$ )
      )
    )
  )
```

```
end
```

```
module Freeable ( $\mathbb{S}$ : Compositional_state_model) = struct
```

```
  type  $\Sigma$  = S of  $\mathbb{S}.\Sigma$  |  $\emptyset$ 
```

```
  let  $\emptyset$  =  $\mathbb{S}.\emptyset$ 
```

```
  type  $\mathcal{A}$  = A of  $\mathbb{S}.\mathcal{A}$  | Free
```

```
  let eval_action  $\sigma$   $\alpha$   $\vec{v}$  =
    match  $\alpha$ ,  $\vec{v}$  with
    | Free, []  $\rightarrow$  (
      match  $\sigma$  with
      |  $\emptyset \rightarrow$  error (DoubleFree,  $\sigma$ )
      | S  $\sigma \rightarrow$ 
        if is_exclusively_owned  $\sigma$  then ok ( $\emptyset$ ,  $\sigma$ )
        else miss (MissingResource,  $\sigma$ )
    )
    | A  $\alpha$ , _  $\rightarrow$  (
      match  $\sigma$  with
      |  $\emptyset \rightarrow$  error (UseAfterFree,  $\sigma$ )
      | S  $\sigma \rightarrow$  (
        let* ( $v, \sigma'$ ) =  $\mathbb{S}.\text{eval\_action } \sigma \alpha \vec{v}$  in
        ok ( $v$ , S  $\sigma'$ )
      )
    )
  )
```

```

type  $\Delta = \text{C of } \mathbb{S}.\Delta \mid \text{Freed}$ 

let produce  $\sigma \delta \vec{v}_i \vec{v}_o =$ 
  match  $\delta, \vec{v}_i, \vec{v}_o$  with
  | C  $\delta, \vec{v}_i, \vec{v}_o \rightarrow$  (
    match  $\sigma$  with
    |  $\emptyset \rightarrow$  vanish
    | S  $\sigma \rightarrow$  (
      let*  $\sigma' = \mathbb{S}.\text{produce } \sigma \delta \vec{v}_i \vec{v}_o$  in
      return (S  $\sigma'$ )
    )
  )
  | Freed, [], []  $\rightarrow$  (
    match  $\sigma$  with
    |  $\emptyset \rightarrow$  vanish
    | S  $\sigma \rightarrow$ 
      if  $\sigma = \mathbb{S}.0$  then ok  $\emptyset$ 
      else vanish
  )
  | _  $\rightarrow$  vanish

let consume  $\sigma \delta \vec{v}_i =$ 
  match  $\delta, \vec{v}_i$  with
  | C  $\delta, \vec{v}_i \rightarrow$  (
    match  $\sigma$  with
    |  $\emptyset \rightarrow$  lfail (UseAfterFree,  $\sigma$ )
    | S  $\sigma \rightarrow$  (
      let*  $(\vec{v}_o, \sigma') = \mathbb{S}.\text{consume } \sigma \delta \vec{v}_i$  in
      ok  $(\vec{v}_o, \text{S } \sigma')$ 
    )
  )
  | Freed, []  $\rightarrow$  (
    match  $\sigma$  with
    |  $\emptyset \rightarrow$  ok ([],  $\emptyset$ )
    | _  $\rightarrow$  lfail (NotFreed,  $\sigma$ )
  )
  | _, _  $\rightarrow$  lfail (InvalidArguments,  $\sigma$ )

end

```

Lemma E.13 (Freeable state model: concrete soundness).

If $\mathbb{S}.\Sigma$ is a cancellative monoid, then:

$$\mathbb{S} \underset{m}{\overset{\text{C}}{\sim}} \underline{\mathbb{S}} \implies \text{Freeable} \underset{m}{\overset{\text{C}}{\sim}} \underline{\text{Freeable}}$$

Proof.

Proposition: Frame addition Assume

(H1) $\sigma.\alpha(\vec{v}) \rightsquigarrow o : (v, \sigma')$

(H2) $\sigma_f \# \sigma'$

(H3) $o \neq \text{Miss}$

To prove (G1) $\sigma \cdot \sigma_f.\alpha(\vec{v}) \rightsquigarrow o : (v, \sigma' \cdot \sigma_f)$

We perform case analysis on the kind of action. If the action is inherited from \mathbb{S} , then the result is trivial, as the behavior is unchanged. We consider the free action: (H4) $\alpha = \text{free}$. There are two cases, either $\sigma = \emptyset$ or $\sigma \in \mathbb{S}.\Sigma$. **Case** $\sigma = \emptyset$:

(H5) $\sigma = \sigma' \emptyset$

(H6) $\vec{v} = \text{DoubleFree}$

From (H5), (H1) and the definition of composition, we know that **(H7)** $\sigma_f = 0$. From there, the result is trivial, since $\sigma \cdot \sigma_f = \sigma$.

Case $\sigma = S \sigma_s$:

Since the outcome is not **Miss**, it must be that

(H8) $\sigma' = \emptyset$

Therefore, $\sigma_f = 0$ and the result is also trivially obtained.

Proposition: Frame subtractionAssume

(H1) $\sigma \cdot \sigma_f \cdot \alpha(\vec{v}) \rightsquigarrow o : (v, \sigma')$

To prove

(G1) $\exists o', v', \sigma''. \sigma \cdot \alpha(\vec{v}) \rightsquigarrow o' : (v', \sigma'') \wedge$
 $(o' \neq \text{Miss} \Rightarrow \sigma' = \sigma'' \cdot \sigma_f \wedge o = o' \wedge v = v')$

There are no vanishing transitions, so we know that there exists o', v', σ'' such that

(H2) $\sigma \cdot \alpha(\vec{v}) \rightsquigarrow o' : (v', \sigma'')$

Now, assume that **(H3)** $o' \neq \text{Miss}$.

Again, the case of inherited actions is trivial, so we consider the case of the free action. There are two cases, either $\sigma \cdot \sigma_f = \emptyset$ or $\sigma \cdot \sigma_f \in \mathbb{S} \cdot \Sigma$.

Case $\sigma \cdot \sigma_f = \emptyset$:

In this case, since \emptyset can only be obtained by compositing 0 with \emptyset , so from (H3), we know that

(H4) $\sigma_f = 0$

(H5) $\sigma = \emptyset$

From there, the result is trivially obtained.

Case $\sigma \cdot \sigma_f \in \mathbb{S} \cdot \Sigma$:

In this case, again, according to (H3), we know that

(H6) $\text{is_exclusively_owned } \sigma$

Therefore, since $\sigma \# \sigma_f$, it must be that $\sigma_f = 0$, and the result is trivially obtained. \square

Lemma E.14 (Freeable: producers and consumers).

If the producer and consumer of the codomain \mathbb{S} are valid, then the producer and consumer of the partial map state model $\text{Freeable}(\mathbb{S})$ are valid.

Proof. This property is trivial given the simplicity of composition: \emptyset is not disjoint from any state except 0. \square

```

module  $\overline{\text{Freeable}}$  ( $\overline{\mathbb{S}}$ : Symbolic_state_model) = struct

  type  $\overline{\Sigma} = \text{S of } \overline{\mathbb{S}}.\overline{\Sigma} \mid \emptyset$ 
  let  $\overline{0} = \overline{\mathbb{S}}.\overline{0}$ 

  type  $\mathcal{A} = \text{A of } \overline{\mathbb{S}}.\mathcal{A} \mid \text{Free}$ 
  let eval_action  $\overline{\sigma} \alpha \vec{v} =$ 
    match  $\alpha, \vec{v}$  with
    | Free, []  $\rightarrow$  (
      match  $\overline{\sigma}$  with
      |  $\emptyset \rightarrow$  error (DoubleFree,  $\overline{\sigma}$ )
      |  $\text{S } \overline{\sigma} \rightarrow$ 
        if is_exclusively_owned  $\overline{\sigma}$  then ok ( $\emptyset, \overline{\sigma}$ )
        else miss (MissingResource,  $\overline{\sigma}$ )
      )
    | A  $\alpha, \_ \rightarrow$  (
      match  $\overline{\sigma}$  with
      |  $\emptyset \rightarrow$  error (UseAfterFree,  $\overline{\sigma}$ )
      |  $\text{S } \overline{\sigma} \rightarrow$  (
        let* ( $\vec{v}, \overline{\sigma}'$ ) =  $\overline{\mathbb{S}}.\text{eval\_action } \overline{\sigma} \alpha \vec{v}$  in
        ok ( $\vec{v}, \text{S } \overline{\sigma}'$ )
      )
    )
  )

  type  $\Delta = \text{C of } \overline{\mathbb{S}}.\Delta \mid \text{Freed}$ 

  let produce  $\overline{\sigma} \delta \vec{v}_i \vec{v}_o =$ 
    match  $\delta, \vec{v}_i, \vec{v}_o$  with
    | C  $\delta, \vec{v}_i, \vec{v}_o \rightarrow$  (
      match  $\overline{\sigma}$  with
      |  $\emptyset \rightarrow$  vanish
      |  $\text{S } \overline{\sigma} \rightarrow$  (
        let*  $\overline{\sigma}' = \overline{\mathbb{S}}.\text{produce } \overline{\sigma} \delta \vec{v}_i \vec{v}_o$  in
        return (S  $\overline{\sigma}'$ )
      )
    )
  | Freed, [], []  $\rightarrow$  (
    match  $\overline{\sigma}$  with
    |  $\emptyset \rightarrow$  vanish
    |  $\text{S } \overline{\sigma} \rightarrow$ 
      if  $\overline{\sigma} = \overline{\mathbb{S}}.\overline{0}$  then ok  $\emptyset$ 
      else vanish
    )
  | _  $\rightarrow$  vanish

  let consume  $\overline{\sigma} \delta \vec{v}_i =$ 
    match  $\delta, \vec{v}_i$  with
    | C  $\delta, \vec{v}_i \rightarrow$  (
      match  $\overline{\sigma}$  with
      |  $\emptyset \rightarrow$  lfail (UseAfterFree,  $\overline{\sigma}$ )
      |  $\text{S } \overline{\sigma} \rightarrow$  (
        let* ( $\vec{v}_o, \overline{\sigma}'$ ) =  $\overline{\mathbb{S}}.\text{consume } \overline{\sigma} \delta \vec{v}_i$  in
        ok ( $\vec{v}_o, \text{S } \overline{\sigma}'$ )
      )
    )
  | Freed, []  $\rightarrow$  (
    match  $\overline{\sigma}$  with
    |  $\emptyset \rightarrow$  ok ([],  $\emptyset$ )
    | _  $\rightarrow$  lfail (NotFreed,  $\overline{\sigma}$ )
  )
  | _, _  $\rightarrow$  lfail (InvalidArguments,  $\overline{\sigma}$ )
end

```

Lemma E.15 (Partial map state model: symbolic soundness).

$$\overline{\mathbb{S}} \stackrel{\mathbb{S}}{\sim}_m \mathbb{S} \implies \overline{\text{Freeable}(\overline{\mathbb{S}})} \stackrel{\mathbb{S}}{\sim}_m \text{Freeable}(\mathbb{S})$$

Proof. The proof is free from the symbolic execution monad results. \square

E.8 Predicate state model transformer

The justification of soundness or the predicate state model is more convoluted than that of other transformers, and requires an additional intermediate proof step. We first define a concrete transformer $\text{Pred}(\mathbb{S})$ which we can be used in our framework, providing the same guarantees as \mathbb{S} itself. We then define the symbolic $\overline{\text{Pred}}(\overline{\mathbb{S}})$ which is shown to be symbolically sound with respect to $\text{Pred}(\mathbb{S})$.

We provide the full definition of the concrete and symbolic predicate state transformer:

```
module rec Pred (S) (P) = struct

  type Σ = S.Σ × (P.names × Val list × Val list) list

  type A =
    | A of S.A
    | Unfold of P.names
    | Fold of P.names

  let rec remove_pred ρ v̄_i p =
    match p with
    | [] → lfail (PredicateNotFound, p)
    | (ρ', v̄'_i, o)::p' when ρ = ρ' →
      if v̄_i = v̄'_i then ok (v̄_o, p')
      else remove_pred ρ v̄_i p'
    | (ρ', _)::p' when ρ ≠ ρ' →
      remove_pred ρ v̄_i p'

  let unfold σ ρ v̄_i =
    let (σ_s, p) = σ in
    let v̄_o, p' = remove_pred ρ v̄_i p in
    let ⟨ρ⟩(x̄_i; x̄_o) ≜ ∨ P̄ = P[ρ] in
    let* P = P̄ in (* For each definition *)
    let θ = [x̄_i ↦ v̄_i, x̄_o ↦ v̄_o] in
    (* This is recursive on the module itself *)
    let* σ' = produce_asrt (P̄ S P) (σ_s, p') θ P in
    ok ((), σ')

  let rec first_success P̄ θ σ =
    match P̄ with
    | [] → Abort
    | P :: P̄' →
      let consumption_result =
        let* θ', (σ'_s, p') = S.consume_asrt (P̄ S P) σ θ P in
        let v̄_i, v̄_o = θ'[x̄_i], θ'[x̄_o] in
        ok ((), (σ'_s, (ρ, v̄_i, v̄_o) :: p))
      in
      match outcome_result with
      | (Ok, _, _) →
        (* If predicate was successfully consumed we return this outcome *)
        outcome_result
      | _ →
        (* Otherwise we try the next predicate definition *)
        first_success P̄' θ σ

  let fold σ ρ v̄_i =
```



```

let ( $\sigma_s, p$ ) =  $\sigma$  in
let  $\langle \rho \rangle(\vec{x}_i; \vec{x}_o) \triangleq \bigvee \vec{P} = \mathbf{P}[\rho]$  in
let  $\theta = [\mathbf{x}_i \mapsto \vec{v}_i]$  in
first_success  $\vec{P} \theta \sigma$ 

let eval_actions  $\alpha \sigma \vec{v} =$ 
  match  $\alpha, \vec{v}$  with
  | Unfold  $\rho, \vec{v}_i \rightarrow$  unfold  $\sigma \rho \vec{v}_i$ 
  | Fold  $\rho, \vec{v}_i \rightarrow$  fold  $\sigma \vec{v}_i$ 
  | A  $\alpha_s \rightarrow$ 
    let ( $\sigma_s, p$ ) =  $\sigma$  in
    let* ( $\vec{v}, \sigma'_s$ ) =
      with_abort_instead_of_miss
      ( $\mathbb{S}.\text{eval\_action } \alpha_s \sigma_s \vec{v}$ )
    in
    ok ( $\vec{v}, (\sigma'_s, p)$ )

let  $\Delta = \mid \text{C of } \mathbb{S}.\Delta \mid \text{U of } \mathbf{P}.\text{names}$ 

let produce  $\sigma \delta \vec{v}_i \vec{v}_o =$ 
  let ( $\sigma_s, p$ ) =  $\sigma$  in
  match  $\delta$  with
  | C  $\delta_s \rightarrow$ 
    let*  $\sigma'_s = \mathbb{S}.\text{produce } \sigma_s \delta_s \vec{v}_i \vec{v}_o$  in
    return ( $\sigma'_s, p$ )
  | U  $\rho \rightarrow$  return ( $\sigma_s, \langle \rho \rangle(\vec{v}_i; \vec{v}_o) :: p$ )

let consume  $\delta (\sigma, p) \vec{v}_i =$ 
  match  $\delta$  with
  | C  $\delta' \rightarrow$ 
    let* ( $\vec{v}_o, \sigma'$ ) =
      with_lfail_instead_of_miss
      ( $\sigma.\text{consume } \delta' \sigma \vec{v}$ )
    in
    ok ( $\vec{v}_o, (\sigma', p)$ )
  | U  $\rho \rightarrow$ 
    let* ( $\vec{v}_o, p'$ ) =
      remove_pred  $\rho \vec{v}_i p$ 
    in
    ok ( $\vec{v}_o, (\sigma, p')$ )

end

module rec  $\overline{\text{Pred}}$  ( $\mathbb{S}$ ) ( $\mathbf{P}$ ) = struct

  type  $\overline{\Sigma} = \mathbb{S}.\overline{\Sigma} \times (\mathbf{P}.\text{names} \times \overline{\text{Val}} \text{ list} \times \overline{\text{Val}} \text{ list}) \text{ list}$ 

  type  $\mathcal{A} =$ 
    | A of  $\mathbb{S}.\mathcal{A}$ 
    | Unfold of  $\mathbf{P}.\text{names}$ 
    | Fold of  $\mathbf{P}.\text{names}$ 

  let rec remove_pred  $\rho \vec{v}_i p =$ 
    match  $p$  with
    | []  $\rightarrow$  Abort
    | ( $\rho', \vec{v}'_i, \vec{v}_o$ )::p' when  $\rho = \rho' \rightarrow$ 
      if%sat  $\vec{v}_i = \vec{v}'_i$  then ok ( $\vec{v}_o, p'$ )
      else remove_pred  $\rho \vec{v}_i p'$ 
    | ( $\rho', \_$ )::p' when  $\rho \neq \rho' \rightarrow$ 
      remove_pred  $\rho \vec{v}_i p'$ 

  let unfold  $\overline{\sigma} \rho \vec{v}_i =$ 
    let ( $\overline{\sigma}_s, p$ ) =  $\overline{\sigma}$  in
    let  $\vec{v}_o, p' = \text{remove\_pred } \rho \vec{v}_i p$  in
    let  $\langle \rho \rangle(\vec{x}_i; \vec{x}_o) \triangleq \bigvee \vec{P} = \mathbf{P}[\rho]$  in

```

```

let* P =  $\vec{P}$  in (* For each definition *)
let  $\vec{\theta} = [x_i \mapsto \vec{v}_i, x_o \mapsto \vec{v}_o]$  in
(* This is recursive on the module itself *)
let*  $\vec{\sigma}' = \text{produce\_asrt } (\overline{\text{Pred}} \ \vec{\mathbb{S}} \ \mathbf{P}) \ (\vec{\sigma}_s, p') \ \vec{\theta} \ P \text{ in}$ 
ok ((),  $\vec{\sigma}'$ )

let rec first_success  $\vec{P} \ \vec{\theta} \ \vec{\sigma} =$ 
  match  $\vec{P}$  with
  | []  $\rightarrow$  Abort
  |  $P :: \vec{P}' \rightarrow$ 
    (* Only works if all outcomes are ok *)
    try%sym
      (* Recursive on the module itself *)
      let*  $\theta', (\vec{\sigma}'_s, p') = \vec{\mathbb{S}}.\text{consume\_asrt } (\overline{\text{Pred}} \ \vec{\mathbb{S}} \ \mathbf{P}) \ \vec{\sigma} \ \vec{\theta} \ P \text{ in}$ 
      let  $\vec{v}_i, \vec{v}_o = \theta'[\vec{x}_i], \theta'[\vec{x}_o]$  in
      ok ((), ( $\vec{\sigma}'_s, (\rho, \vec{v}_i, \vec{v}_o) :: p$ ))
    with
    | _  $\rightarrow$  first_success  $\vec{P}' \ \vec{\theta} \ \vec{\sigma}$ 

let fold  $\vec{\sigma} \ \rho \ \vec{v}_i =$ 
  let ( $\vec{\sigma}_s, p$ ) =  $\vec{\sigma}$  in
  let  $\langle \rho \rangle(\vec{x}_i; \vec{x}_o) \triangleq \bigvee \vec{P} = \mathbf{P}[\rho]$  in
  let  $\vec{\theta} = [x_i \mapsto \vec{v}_i]$  in
  first_success  $\vec{P} \ \vec{\theta} \ \vec{\sigma}$ 

let eval_actions  $\alpha \ \vec{\sigma} \ \vec{v} =$ 
  match  $\alpha, \vec{v}$  with
  | Unfold  $\rho, \vec{v}_i \rightarrow$  unfold  $\vec{\sigma} \ \rho \ \vec{v}_i$ 
  | Fold  $\rho, \vec{v}_i \rightarrow$  fold  $\vec{\sigma} \ \vec{v}_i$ 
  | A  $\alpha_s \rightarrow$ 
    let ( $\vec{\sigma}_s, p$ ) =  $\vec{\sigma}$  in
    let* ( $\vec{v}, \vec{\sigma}'_s$ ) =
      with_abort_instead_of_miss
      ( $\vec{\mathbb{S}}.\text{eval\_action } \alpha_s \ \vec{\sigma}_s \ \vec{v}$ )
    in
    ok ( $\vec{v}, (\vec{\sigma}'_s, p)$ )

let  $\Delta = | \text{C of } \vec{\mathbb{S}}.\Delta \ | \cup \text{ of } \mathbf{P}.\text{names}$ 

let produce  $\vec{\sigma} \ \delta \ \vec{v}_i \ \vec{v}_o =$ 
  let ( $\vec{\sigma}_s, p$ ) =  $\vec{\sigma}$  in
  match  $\delta$  with
  | C  $\delta_s \rightarrow$ 
    let*  $\vec{\sigma}'_s = \vec{\mathbb{S}}.\text{produce } \vec{\sigma}_s \ \delta_s \ \vec{v}_i \ \vec{v}_o \text{ in}$ 
    return ( $\vec{\sigma}'_s, p$ )
  | U  $\rho \rightarrow$  return ( $\vec{\sigma}_s, \langle \rho \rangle(\vec{v}_i; \vec{v}_o) :: p$ )

let consume  $\delta \ (\sigma, p) \ \vec{v}_i =$ 
  match  $\delta$  with
  | C  $\delta' \rightarrow$ 
    let* ( $\vec{v}_o, \sigma'$ ) =
      with_lfail_instead_of_miss
      ( $\vec{\sigma}.\text{consume } \delta' \ \sigma \ \vec{v}$ )
    in
    ok ( $\vec{v}_o, (\sigma', p)$ )
  | U  $\rho \rightarrow$ 
    let* ( $\vec{v}_o, p'$ ) =
      remove_pred  $\rho \ \vec{v}_i \ p$ 
    in
    ok ( $\vec{v}_o, (\sigma, p')$ )

end

```

The fold function makes use of an intermediate function called first_success which will attempt to consume each predicate in the list

of disjunctions, until one fully succeeds (it has on `Ok` outcomes). This is performed using the `try%sym ... with ...` construct, which overrides the try-with syntax of OCaml and instead checks that all results of the symbolic execution monad within the block are `Ok`. If any of them are not, the block is aborted and the next predicate is tried.

Definition E.16 (Concrete predicate state satisfiability).

We define the satisfiability relation $\sigma \models \langle \rho \rangle(\vec{v}_i; \vec{v}_o)$ as:

$$\begin{aligned} \sigma \models \langle \rho \rangle(\vec{v}_i; \vec{v}_o) &\Leftrightarrow \theta, \sigma \models Q_1 \vee \dots \vee \theta, \sigma \models Q_n \\ &\text{where } \left(\langle \rho \rangle(\vec{x}_i; \vec{x}_o) \triangleq \bigvee_{i=1}^n Q_i \right) \in \mathbf{P} \\ &\text{and } \theta = [\vec{x}_i \mapsto \vec{v}_i, \vec{x}_o \mapsto \vec{v}_o] \end{aligned}$$

We define a relation \models_p between concrete states of \mathbb{S} and states of $\text{Pred}(\mathbb{S}, \mathbf{P})$ as:

$$\begin{aligned} \sigma \models_p [] &\Leftrightarrow \sigma = 0 \\ \sigma \models_p \langle \rho \rangle(\vec{v}_i; \vec{v}_o) :: p' &\Leftrightarrow \exists \sigma_1, \sigma_2. \sigma = \sigma_1 \cdot \sigma_2 \\ &\quad \wedge \sigma_1 \models \langle \rho \rangle(\vec{v}_i(\varepsilon); \vec{v}_o(\varepsilon)) \\ &\quad \wedge \varepsilon, \sigma_2 \models_p p' \\ \varepsilon, \sigma \models_p (\sigma_s, p) &\Leftrightarrow \exists \sigma_p. \sigma = \sigma_s \cdot \sigma_b \cdot \sigma_p \\ &\quad \wedge \varepsilon, \sigma_b \models \bar{\sigma} \\ &\quad \wedge \varepsilon, \sigma_p \models_p p \end{aligned}$$

Lemma E.17 (Concrete predicate abstraction).

The concrete predicate state model is an over-approximate abstraction over its input. More formally, let \mathbb{S} be a concrete compositional state model, and \mathbf{P} be a set of user-defined predicates that make use of the core predicates of \mathbb{S} . Then the following properties hold, where $\mathbb{S}_{\mathbf{P}} = \text{Pred}(\mathbb{S}, \mathbf{P})$:

$$\begin{aligned} \sigma. \alpha(\vec{v}) &\xrightarrow{\mathbb{S}} o : (v, \sigma') \wedge \sigma \models_p (\sigma_s, p) \\ \Rightarrow (\sigma_s, p). \alpha(\vec{v}) &\xrightarrow{\mathbb{S}_{\mathbf{P}}} o' : (v', (\sigma'_s, p')) \\ &\quad \wedge (o' \neq \text{Miss} \Rightarrow (o = o' \wedge v = v' \wedge \sigma' \models_p (\sigma'_s, p'))) \\ &\quad \text{(Predicate action soundness)} \end{aligned}$$

$$\begin{aligned} \sigma \models (\sigma_s, p) \wedge (\sigma_s, p). \text{unfold}_{\rho}(\vec{v}) &\xrightarrow{\mathbb{S}_{\mathbf{P}}} \text{Ok} : (v, (\sigma'_s, p')) \\ \Rightarrow \sigma \models_p (\sigma'_s, p') & \\ &\quad \text{(Unfold abstracts no-op)} \end{aligned}$$

$$\begin{aligned} \sigma \models (\sigma_s, p) \wedge (\sigma_s, p). \text{fold}_{\rho}(\vec{v}) &\xrightarrow{\mathbb{S}_{\mathbf{P}}} \text{Ok} : (v, (\sigma'_s, p')) \\ \Rightarrow \sigma \models_p (\sigma'_s, p') & \\ &\quad \text{(Fold abstracts no-op)} \end{aligned}$$

$$\begin{aligned} (\sigma_s, p). \text{cons}_{\delta}(\vec{v}_i) &\xrightarrow{\mathbb{S}_{\mathbf{P}}} o : (\vec{v}_o, (\sigma'_s, p')) \wedge \sigma \models_p (\sigma_s, p) \\ \Rightarrow \exists \sigma', \sigma_{\delta}. \sigma' \models (\sigma'_s, p') \wedge \sigma &= \sigma' \cdot \sigma_{\delta} \wedge \sigma_{\delta} \models \langle \delta \rangle(\vec{v}_i; \vec{v}_o) \\ &\quad \text{(Predicate consumer validity)} \end{aligned}$$

$$\begin{aligned} \sigma \models (\sigma_s, p) \wedge \sigma_{\delta} \models \langle \delta \rangle(\vec{v}_i; \vec{v}_o) \wedge \sigma \# \sigma_{\delta} & \\ \Rightarrow \exists (\sigma'_s, p'). (\sigma_s, p). \text{prod}_{\delta}(\vec{v}_i, \vec{v}_o) &\xrightarrow{\mathbb{S}_{\mathbf{P}}} (\sigma'_s, p') \wedge \sigma \cdot \sigma_{\delta} \models (\sigma'_s, p') \\ &\quad \text{(Predicate producer validity)} \end{aligned}$$

Proof.

Proposition: Predicate action soundness

Actions that are inherited from \mathbb{S} satisfy [Predicate action soundness](#) trivially, by application of [Frame subtraction](#). Specifically, if there is a transition from σ to σ' with action α and $\sigma \models_p (\sigma_s, p)$, then $\sigma = \sigma_s \cdot \sigma_p$ where $\sigma_p \models_p p$. Using frame subtraction, we can learn that a transition exists from σ_s , and either too much information has been removed, in which case a **Miss** occurs, or the exact same transition exists to σ'_s where $\sigma' = \sigma'_s \cdot \sigma_p \models (\sigma'_s, p)$.

Proposition: Unfold abstracts no-op

Unfold is only successful if it found the same predicate in the state, in which case the predicate is removed and its definitions are produced in the state. By definition of \models_p , if $\sigma \models_p (\sigma_s, p)$, then $\sigma = \sigma_s \cdot \sigma_p$ where $\sigma_p \models_p p$. If a predicate $\langle \rho \rangle(\vec{v}_i; \vec{v}_o)$ is found, then it is removed from the state, and we have a "frame state" σ_s, p_f where p_f is p with the predicate removed. Therefore, again by definition of \models_p , there is some $\sigma_f = \sigma_s \cdot \sigma_p^f$ such that $\sigma_p^f \models_p p_f$, and $\sigma = \sigma_f \cdot \sigma_p$ where $\sigma_u p \models \langle \rho \rangle(\vec{v}_i; \vec{v}_o)$. Satisfying $\langle \rho \rangle(\vec{v}_i; \vec{v}_o)$ is equivalent to satisfying one of its definition, so using [Predicate producer validity](#), we obtain the result that $\sigma \models_p (\sigma'_s, p')$, and unfold is a no-op.

Proposition: Fold abstracts no-op

This is similar to the above property, but in reverse. The proof leverages [Predicate consumer validity](#), instead of [Predicate producer validity](#).

Proposition: Predicate consumer validity

There are two kinds of core predicates. User defined predicates are simply removed from the list of predicates on match, and if successful, by definition of \models_p , guarantee that the property holds. On the other hand, core predicates inherited from \mathbb{S} are consumed only from σ_s , and the property is obtained by applying validity of the consumer for \mathbb{S} .

Proposition: Predicate producer validity

The proof is similar to the above. User defined predicates are added to the list of predicate and the result is trivially obtained through definition of \models_p , while core predicates inherited from \mathbb{S} are produced only in σ_s , and the result is obtained by applying the validity of the producer for \mathbb{S} .

□

We now show that specification calls in the concrete predicate state model are sound with respect to the concrete state model without predicates. This result could be generalised to any "abstraction" state which preserves the above properties.

Lemma E.18 (Soundness of specification calls in concrete abstraction).

Let γ be a program and $S = \{ P \} e \{ \text{Ok} : r. Q_{\text{Ok}} \} \{ \text{Err} : r. Q_{\text{Err}} \}$

be a separation logic quadruple. Then:

$$\begin{aligned} \gamma \models S \wedge \gamma \vdash \sigma, e \Downarrow_{\theta}^{\mathbb{S}} o : (v, \sigma') \wedge \sigma \models_p (\sigma_s, p) &\implies \\ (\exists o', v', \sigma', p'. (\sigma_s, p). \text{spec}_S(\theta) \overset{\mathbb{S}^P}{\rightsquigarrow} o' : (v', (\sigma', p')) & \\ \wedge (o' \notin \{\text{Miss}, \text{Lfail}\} \Rightarrow (o' = o' \wedge v' = v \wedge \sigma' \models_p (\sigma'_s, p')))) & \\ \text{(Specification call soundness with predicates)} & \end{aligned}$$

Proof. This proof is almost identical to that of [Theorem 5.13](#), but it is slightly more general, and requires the indirection through \models_p . However, this indirection is trivial, and we do not repeat the full proof here. \square

The results from [Lemma E.18](#), together with [Predicate action soundness](#) naturally lift to the whole specification semantics, ensuring OX-soundness of the analysis performed using a predicate state model. The final piece of the puzzle is the soundness of the symbolic predicate state model with respect to the concrete predicate state model.

Lemma E.19 (Predicate state model: soundness).

$$\overline{\mathbb{S}} \overset{\mathbb{S}}{\underset{m}{\sim}} \mathbb{S} \implies \overline{\text{Pred}}(\overline{\mathbb{S}}, \mathbf{P}) \overset{\mathbb{S}}{\underset{m}{\sim}} \text{Pred}(\mathbb{S}, \mathbf{P})$$

Proof. The proof is straightforward by lifting through the symbolic execution monad. \square

Some Gillian-Rust tactics

F.1 Freezing existential variables

¹ Jung, “Understanding and evolving the Rust programming language”, 2020 [Jun20]

For instance, consider the borrow $\&^\kappa(\exists y, z. x \mapsto y * y \mapsto z)$. If this borrow corresponds to a mutable reference, one could provide a sub-borrow $\&^\kappa(y \mapsto z)$. However, it is not possible to do so without saying that y cannot change anymore. For this reason, one must start by first freezing y , obtaining $\&^\kappa(\exists z. x \mapsto y * y \mapsto z)$. Then, one can split the borrow in two, thereby obtaining $\exists y. \&^\kappa(x \mapsto y) * \&^\kappa(\exists z. y \mapsto z)$, before discarding the first part.

```
#[with_freeze_lemma(
  lemma_name = freeze_y,
  predicate_name = some_borrow_frozen,
  frozen_variables = [ y ]
)]
#[borrow]
fn some_borrow(x: *mut *mut i32) {
  gilsonite!(exists y: *mut i32, z: i32. x → y * y → z)
}
```

[illegible]

F.2 Borrow extraction with prophecy variables

Recall that, when proving type safety, the borrow-extract rule is as follows:

$$\frac{\text{BORROW-EXTRACT} \quad \text{persistent}(F) \quad F * P \Rightarrow Q * (Q \multimap P)}{F * [\kappa]_q * \&^\kappa P \equiv \multimap \&^\kappa Q * [\kappa]_q}$$

We have proven this rule in Iris RustBelt development. Gillian-Rust provides a macro to instantiate a trusted lemma corresponding to the update in the conclusion of the rule, together with a proof obligation corresponding to the premise.

The premise says that, in the context of a persistent assertion F , assertion Q can be derived from P , together with a wand $Q \multimap P$, which ensures that, should the invariant Q be restored, the assertion P can be derived again. While this rule is sufficient to prove type safety of functions that perform borrow extraction, it is not enough to prove their functional correctness.

To prove functional correctness, one needs, in addition, to describe the relationship between the value contained in the original borrow, and that contained in the extracted borrow. This is done by introducing a function $f(a, b)$, where a is the value of the original borrow, and b is the value of the extracted borrow. The corresponding rule is the following:

$$\frac{\text{BORROW-EXTRACT-PROPH} \quad \text{persistent}(F) \quad f(a, -) \text{ injective} \quad F * P(a) \Rightarrow Q(b) * a = f(a, b) * (Q(b') \multimap P(f(a, b')))}{F * [\kappa]_q * \&^\kappa (\exists a. P(a) * \text{PC}_x(a)) * \text{VO}_x(a) \equiv \multimap \&^\kappa (\exists b. Q(b) * \text{PC}_y(b)) * \text{VO}_y(b) * (\uparrow x = f(a, \uparrow y)) * (a = f(a, b)) * [\kappa]_q}$$

Let us walk through the rule step by step. The first premise is the same as in the previous rule. It allows us to perform extraction within the context of a persistent assertion F . For instance, when extracting the first node of a linked list, F is the pure assertion that states that the list is not empty.

The second premise requires that the function $\lambda b. f(a, b)$ is injective. For instance, again in the case of extracting a borrow to the first element of a linked list, the function f connects the representation of the list to the representation of the first element: $f(a, b) = b :: (\text{tail } a)$. It is easy to check that this function is injective.

The final premise is a generalisation of the premise of BORROW-EXTRACT. It states that, if the invariant P holds for a value a , then the invariant Q holds for a value b such that $a = f(a, b)$, and for any b' , if $Q(b')$ holds, then it is possible to recover the invariant P for the value $f(a, b')$. In the case of the linked list, this states that, given the entire linked list with representation a , one can extract a pointer to its first element with representation b , such that the entire list a is the tail of a with b prepended. In addition, if the pointer to the first element is returned with representation b' , then the invariant for the entire linked-list is recovered with representation $f(a, b')$, that is, the tail of a with b' prepended.

The conclusion is an update which requires the context F to hold, together with a lifetime token $[\kappa]_q$, and resource that has *the same shape as the ownership predicate of a mutable reference*, with invariant P , prophecy variable x and representation a . Such resource is usually either directly the ownership predicate of a mutable reference or a predicate obtained by freezing variable in the full borrow it contains. The update does not modify the lifetime token, and produces a new mutable-reference-like resource with invariant Q , prophecy variable y and representation b . In addition, it *partially resolves* the prophecy variable x , stating that the future value of x (at the time when the borrow expires), denoted by $\uparrow x$, shall be $f(a, \uparrow y)$, where $\uparrow y$ is the future value of y . Finally, it states the the current representation a is equal to $f(a, b)$.

In the case of the linked list, the observations respectively state that the current value of the entire linked list is obtained by prepending the current value of the obtained pointer to the first element to the tail of the current list; and that the future value of the entire list is obtained by prepending the future value of the pointer to the first element to the tail of the current list. This is true since, when using the function `first_mut`, nothing other than the first element of the list can be modified until the borrow expires, as enforced by the borrow checker.

Similarly to the case without prophecies, Gillian-Rust generates the obligation corresponding to the separation between the extracted resource and the magic wand. In addition, it generates a second obligation for the injectivity of the function $f(a, -)$, which is usually trivially discharged.