Scaling the Evolution of Verified Software

KIRAN GOPINATHAN

(B.S., University College London)

A THESIS SUBMITTED
FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

SCHOOL OF COMPUTING
NATIONAL UNIVERSITY OF SINGAPORE

2024

Thesis Advisor:

Associate Professor Ilya Sergey

Jury:

Professor Olivier Gerard Henri Marie Danvy

Dr Assia Mahboubi (Vrije Universiteit Amsterdam, the Netherlands)

Declaration

I, Kiran Gopinathan, hereby declare that this thesis is my original work and it has been written by me in its entirety. I have duly acknowledged all the sources of information which have been used in the thesis.

This thesis has not been submitted for any degree in any university previously.

. . . .

Kiran Gopinathan

September 4, 2024

Contents

Summary

Formal verification is on the cusp of becoming mainstream. The past decade of verification research has convincingly demonstrated the efficacy of verification techniques for certifying large-scale real world software systems, such as compilers, web servers, distributed systems, file systems and even whole operating system kernels. Despite these successes, however, the challenge of *maintaining* such verified artefacts in the face of their inevitable evolution unfortunately remains largely unaddressed and persists as a critical obstacle that must be overcome if verification is ever to achieve wider adoption.

The focus of this thesis is in tackling this challenge of maintaining evolving verified software. To this end, this thesis identifies and develops a suite of techniques that can be deployed and applied in anger to manage the burden of verified software maintenance. The thesis incorporates and adopts a three-fold strategy for proof-maintenance, investigating tackling evolution through: first, 1) classical formal techniques of composition and the use of reduction arguments, then second, 2) through the lens of certified synthesis, drawing from prior works on proof-carrying-code and finally 3) under the umbrella of repair, developing the novel technique of proof-driven-testing to scale proof-repair strategies to operate in a real-world setting. Each technique is motivated using a running case study, provided in an accompanying artefact, that demonstrates the efficacy of the technique in a practical setting.

Through the evaluation of each strategy, this thesis finds that the proposed framework is effective at considerably reducing the maintenance burden across a variety of verified codebases with significant impacts, including, for example, reducing the sizes of manual proof scripts of complex and subtle algorithms by half, or reducing the time spent by proof engineers by hours, or, in some cases, eliminating the maintenance burden entirely and synthesizing all required code and proofs automatically.

Acknowledgements

The past five years that lead to the dissertation you are currently reading has been quite the roller coaster for me, not only in a technical sense but also from a personal perspective — from pivoting away from my initial direction of formalising randomised algorithms to verified software maintenance, to realising my gender identity — I am a trans woman, and being disowned by my parents upon this discovery: I feel confident in saying that this document would not have been possible without help of all the kind and caring friends and colleagues that I've been lucky enough to count on over this journey.

To my parents: To Beena and Jim, my mother and father, I love you both and, in so many ways, this journey wouldn't have even started without your support and influence, so I need to thank you for this. I know that we've had our disagreements, and indeed, my decision to isolate myself over childhood must have undoubtedly been challenging for you both. As your first child, I know that this probably was a new experience for us all, and throwing on gender dysphoria on top of balancing the traditions of our Indian roots wasn't exactly a recipe for smooth sailing. Nevertheless, I'm unbelievably grateful for all the support you've given me throughout my PhD, support that I've certainly taken for granted over the years, and undoubtedly not appreciated enough. Thank you for your support, I love you both.

To my sister: Maya, I couldn't have done this without you! Our daily chats over the years are precious memories that I'll treasure forever. Thank you for lending a listening ear to my troubles at work, and in turn, I hope my modest wisdom on handling school and the stresses of GCSEs and A-levels were similarly helpful in your own journey. Watching films with you in those fleeting movements over the holidays were beacons of joy for me, lighting the way and motivating me throughout the PhD. I really hope that one day we might be able to do that again. Perchance during your own PhD studies? Most of all, I'm so thankful that you were supportive and treated me the same after I came out to you. Thank you so much for accompanying me through this journey, I love you.

To my friends: I unfortunately learnt the importance of support networks outside of work far, far too late. Even prior to starting my PhD, I don't think I really formed deep connections or friendships with others until my "egg" had cracked and I realised that I was trans; so, while I am thankful for the privilege that allowed me to ignore this aspect of human interaction, in some ways I am also grateful for the hardships in this journey that forced me out of my shell. Nevertheless, I am immensely grateful

from the bottom of my heart for the wonderful and brilliant friends that I have been so blessed to have interacted with over this journey: for the close friends that I've made from the lab, Callista, Theo and Andreea, I deeply enjoyed our time together — be technical or personal, I've learnt and grown a lot from our conversations, and I'm happy to have been a part of your own development; for the friends and family I've found through the local transgender support groups in Singapore, the members of Transcend and TransBefrienders, I'm sincerely so grateful to have been able to rely on your support networks when I was so lost and alone in this journey, and I hope that in my time here I was able to reciprocate even a fraction of that support and kindness for others when they needed it. I can't imagine how I'd have reached the end without all your kind and loving support. I love you all.

To my PhD advisor, Ilya Sergey, and VERSE-lab: Of course, it goes without saying, none of this would have been remotely possible without your support, Ilya, for which I am so thankful. I'm glad that I ended up studying under your tutelage, and I've deeply appreciated your support throughout this — both from carefully guiding and helping me to develop as a researcher, through your boundless enthusiasm and belief in my research and giving me the challenges and opportunities to build academic skills, through external reviews and co-writing research grants, but crucially also through your sympathy and understanding of the personal aspects of the PhD journey, from your support in the times my grandparents passed away, to your lenience and acceptance as I chose to explore my gender expression. Naturally, my gratitude also goes out to the fellow students who I've overlapped with, without whom VERSE-lab wouldn't have been the same: Vladimir Gladshtein, Yunjeong Lee, Yuxi Ling, George Pîrlea, Ziyi Yang, Qiyuan Zhao, and interns, Vikram Goyal, Mayank Keoliya, Callista Le, Theodore Leebrant, Koon Wen Lee, Gokul Rajiv, Bryan Tan, Sewen Thy, and Yasunari Watanabe. It has been a joy to share in the journey of developing as scholars with you — I hope that our discussions and conversations on research, paper writing, and the academic life were as fun and helpful for you all as they were for me. I'm so grateful to been able to spend my time in VERSE-lab with you all. Thank you.

To my examiners, Assia Mahboubi and Olivier Danvy: Dear Assia and Olivier, I was overjoyed when I heard that you had agreed to be on my committee, and I am grateful for both your time and the detailed feedback and comments on my thesis and our subsequent insightful discussions. Thank you.

To my institution, NUS, School of Computing: Of course, a PhD is not an act done in isolation, and I would be remiss to not acknowledge how much I have benefited from my graduating institute, the National University of Singapore, School of Computing. From the faculty, the administrative staff and all the way to the cleaning staff and system admins, I am grateful for all the support I've received over the years and helping to ensure an environment conducive for doing research. Thank you.

List of Tables

List of Figures

CHAPTER

1

INTRODUCTION & BACKGROUND

This chapter provides an overview of this thesis, introducing the reader to the motivation of the work, presenting the problem statement and then summarises the main contributions. The chapter ends with an outline of the structure of the thesis and a description of each subsequent chapter.

1.1 Motivation

Can you imagine a world liberated from the pestilence of ubiquitous software errors? This utopian dream, far fetched may it be, has been the driving vision of the field of formal verification since its conception; the art of certifying real programs by means of constructing rigorous machine-checked proofs that reason about semantics in order to establish a program's correctness. In recent years, driven by the tireless efforts of innumerable researchers, this vision has begun to come to fruition.

The past decade has seen a radical paradigm shift in the field of formal verification, as frameworks and tooling have matured and researchers have shifted their efforts to tackle large-scale real-world programs. Notable results in this direction are numerous and easy to list: the CompCert [76] and CakeML [71] compilers for C and SML respectively, verified operating systems such as Sel4 [70] and CertiKOS [57], file servers, such as CrashFS [27] and Daisy-NFS [24], and even complex distributed protocols such as Paxos [59], PBFT [110] and Raft [132]. Sadly, however, this move from toy-programs to real-world systems has not been entirely painless, and researchers have found themselves quickly at odds with a more pernicious and pervasive threat from the world of software engineering: that of code evolution.

Programs are naturally want to change; unfortunately, proofs much less so. The evolution of software systems is a well-known and thoroughly studied phenomenon. A program may want to change for any number of common reasons: for example, the project requirements may have shifted, or external libraries may have updated and changed their implementations and interfaces or simply a developer may have just wanted to optimise the program. Broadly speaking, these modifications typically surface in real code in one of three ways: 1) changes in the specifications of the code itself and thus its external interface, 2) changes in the data-structures used by the program and 3) changes in the implementation of the code itself. Unfortunately, software that happens to be verified is no stranger to these forces, but when such software changes, the maintenance problem is effectively doubled, as developers are also shackled with the task of also updating the corresponding proof alongside the original program.

The lengthy history of research on automating code evolution naturally leads to the question of whether techniques from prior works could be extended to handle verified software. Indeed, a review of the related literature surfaces three general approaches for code evolution that seem likely to generalise to verified software — in particular, those embodied in the strategies of composition, synthesis and repair:

Composition Building composable modular software simplifies the maintenance of said systems as the effects of any changes are localised to their respective components, and untouched components can be reused as is — can we orchestrate verification projects such that they are similarly modular, such that intermediate verification results can be reused between different versions of the same system?

Synthesis Prior work has demonstrated how incorporating automated code-generation and synthesis techniques makes a large-scale system more robust to evolution as it can allow necessary code changes to be automatically generated — in the context of verified software, can synthesis techniques be incorporated to automatically generate verified code in response to software evolution?

Repair An effective means of handling code evolution is to view the problem as a repair task, incorporating techniques from automated program repair to infer and apply the necessary patches to the code — can similar methodologies be used to handle the maintenance of verified systems and incorporate the additional information inherently present in verified software to optimise this process?

Unfortunately, few prior works have investigated the use of these techniques for the evolution of proofs.

Can verified software ever hope to be practical given the turbulent nature of real world systems? The driving motive of this thesis is to answer this question. To this end, the thesis considers three case-studies of verified software that each experiences change, and uses each one to investigate one of the aforementioned strategies for managing and mitigating the burden of verified software evolution. For each case study a methodology is proposed to reduce the effort required to keep their proofs faithful to the underlying code, and the technique is implemented in an accompanying artefact and evaluated on its effectiveness in this regard. Putting it all together, this thesis proposes a unified framework for managing evolution in verified software, paving the path to bring verification to the mainstream.

1.2 State of the art

Of course, the problem of maintaining verified software is by no means a novel observation of this work—in fact, such concerns were already identified as critical challenges even around the inception of the very field [38]—however these issues have only started to grow to prominence in recent years as larger systems have begun to be verified, so research into this topic remains at a nascent state.

The most relevant prior work on this problem is the research by Talia Ringer on *proof-repair* [113, 114, 115]. Notably, Ringer *et al.* adopted a repair based approach and were the first to develop automatic techniques for repairing proofs of programs as they change [115] and also coined the notion proof repair [114] to refer to verified software evolution, and forms much of the conceptual basis upon which this work builds. Ringer's work primarily considers the problem of verified software evolution in the context of changes in data-type definitions; first investigating how user-provided examples of the required changes in one proof when a data-type changes could be systematically generalised to other proofs [115], and then later investigating how user-provided proofs of equivalence between different representations of the same data can be lifted to any proofs that reason over these data-types [114]. While the techniques in these works were evaluated over real-world verified codebases, the focus of the tools to specifically only tackle data-type definitions limits their abilities to handle other kinds of changes and thus leaves the challenge of verified software maintenance as an open problem.

For the remaining approaches, that is, composition and synthesis, while both are well-studied in the literature, few works have considered how they may be exploited for the purposes of proof maintenance.

In the case of composition, the benefits of compositionality and modularity for scaling proofs and analysis are well-known, and much research has been done into extending such reasoning to more complicated domains, such as concurrent code [16], distributed systems [127] or probabilistic programs [123]. However, only the work by Woos *et al.* [132], have specifically considered how compositional proofs can reduce the maintenance cost in updating a verified system. In their work, Woos *et al.* discuss their experiences constructing a formally verified implementation of the Raft distributed consensus protocol. While the protocol itself does not change, their verification effort is nonetheless found to be an iterative process, as they repeatedly have to strengthen their specifications and invariants as they certify more and more of the system in question. Drawing from their experiences, Woos *et al.* develop a set of general guidelines on structuring proofs of large-scale distributed systems such that the effects of any individual change are minimised. While these guidelines are generalised to apply to other verified codebases, they are informal, and do not form a systematic stategy for verified software maintenance.

Similarly, for the case of synthesis, while there has been a long lineage of work investigating the automatic synthesis of verified software, no prior research has considered it for handling verified software evolution. In particular, work on synthesizing certifiying code originates from 1998, considered first by Necula and Lee [98] who developed a certifying compiler which would compile programs in a high-level language to assembly and also construct proofs of correctness for the generated code. In the subsequent decades, many works have built on this idea and considered different approaches to synthesizing certified software, developing more advanced certified compilers, such as the CompCert C compiler [76] or CakeML compiler [71], or integrating using metaprogramming facilities of interactic proof assistant to construct certified programs [36]. While all these works address similar problems as those considered in this work, research in this line considers the synthesis task as a one-shot problem, where the generated program is not expected to change, and are not sufficient to ensure maintainability.

Finally, there have a number of works from research into dependently typed provers on tools for refactoring proofs, considering automated techniques to rearrange and restructure proofs which are also relevant to this thesis. For example, the tool Levity by Bourke *et al.* [20] automates the movement of lemmas between different packages in the Isabelle/HOL theorem prover, ensuring that any definitions relying on those lemmas do not break. Similarly, Tactitian [1] is another tool for refactoring proof scripts in HOL Light and RefactorAgda [131] considers the same problem in the Agda programming language. The Chick tool by Valentin Robert [117] is a framework for refactoring proofs in a bespoke dependently typed language with the capability to be extended to more or less expressive languages, with a case study in OCaml. However, overall, the problem of refactoring is mostly orthogonal to the central problem of verified software maintenance considered by this thesis, that of adapting and updating proofs in large-scale verified software systems in response to changes to their implementations.

1.3 Approach

This work investigates a systematic solution to the problem of maintaining verified software systems over changes in implementations, and to this end, puts forward the following thesis statement:

The problem of maintaining large-scale verified software systems can be effectively solved through a combination of three general strategies: ***composition****,* ***synthesis*** *and* ***repair****.*

To substantiate this claim, the thesis presents three case-studies, one for each strategy, of a representative verified software system that experiences change and investigates and evaluates the use of each respective approach to maintain the relevant proofs for each system over the changes.

The case studies of this work are, in order:

- **Composition** - The first case study investigates the use of composition for proof maintenance, and describes the mechanical verification of a class of complicated probabilistic data-structures known as approximate membership query structures (AMQs) in the Coq proof assistant. By deliberately structuring the verification project in a modular and composable way, this case-study investigates how the mechanisms of proof assistants can be used to *reuse* proofs of complex and nuanced probablistic properties as the definition of these data-structures change, by making use of large-scale compositional reduction arguments to port them across data-structure definitions.
- **Synthesis** - Shifting gears away from the human effort required in manual verification, the second case-study investigates synthesis for proof maintenance, presenting a technique to extend an existing separation-logic based synthesiser, SuSLik by Polikarpova and Sergey [105], to produce executable real world-code accompanied with a corresponding foundational proof of correctness that can be independently machine checked. By making use of an automated synthesis approach, the thesis demonstrates how verified systems can be made to gracefully handle changes in their specifications, as the required code and proofs can simply be automatically generated.
- **Repair** - Finally, the last case study investigates the use of repair for proof maintenance, and presents a novel technique, proof-driven testing, and a mostly automated tool based on this technique that can repair real-world verified OCaml programs over changes in their implementations. In this work, the thesis demonstrates how the inherent information within the constructive proofs used by proof assistants can be exploited to allow verified systems to automatically handle changes in their internal implementations while their specifications remain the same.

Putting it all together, the thesis proposes a single unified framework for handling the maintenance of large-scale verified software, using 1) composition, to handle changes in data-structures, then 2) synthesis, for software components whose specification changes frequently, and finally 3) repair, to handle changes in the implementation of actual code. The thesis ends with a representative application of this framework to a hypothetical verified web page server, motivating its effectiveness.

1.4 Contributions

This main contributions of this thesis can be described as listed below:

Framework for AMQ structures - the results presented in the first case study are part of a general framework for the verification of the popular class of probabilistic data structures: Approximate Membership Query (AMQ) structures. This work also describes the first mechanised proof of the elusive false positive rate property for the Bloom Filter datastructure [17]. The results of this case-study were presented before at the 32nd International Conference on Computer Aided Verification (CAV 2020) [53].

Certified Synthesis backend for C - the second case study describes a novel extension to the SuSLik verifier [105] to produce executable real world C programs alongside with proofs of correctness of said code with respect to the input synthesis specifications in the Coq theorem prover in the foundational Verified Software Toolchain framework [7]. These results were developed as part of a larger framework to translate synthesis proofs from SuSLik to proof scripts for verification frameworks developed with other co-authors. The results of this larger project, including those described in this thesis, have been presented before at the 21nd International Conference of Functional Programming (ICFP 2021) [130]

Proof-driven testing - the final case-study presents the first mostly automated tool for the repair of verified OCaml libraries over changes in their implementation. The tool itself has been evaluated on a number of functions from real-world widely-used OCaml libraries and shown to be efficient and effective at reducing the human effort in maintaining verified versions of these programs. As part of this work, the novel technique of proof-driven testing was invented to allow scaling up the proof-repair process to large-scale real world codebases. The results of this research have been presented before at the 44th Conference on Programming Language Design and Implementation (PLDI 2023) [49].

The remainder of this thesis has the following structure:

- Chapter 2 - Verified Software Maintenance through Composition - This chapter presents the first case study of this work, the verification of Approximate Membership-Query structures in the Coq proof assistant, and investigates how verification efforts can incorporate composition to gracefully handle changes in their implementations. The chapter starts with a discussion of the setting and the objectives motivating this work, before then presenting the technical details of the case study. The case study itself is presented in two parts, with the first part providing a gentle introduction to a particular instance of this class, Bloom filters, and how they can be formally verified, before the second part discusses how the analysis itself can use composition to handle a wider class of data structures. The chapter ends with a reflection on the key insights from this case-study on how composition can be used to handle verified software evolution.

- Chapter 3 - Verified Software Maintenance through Synthesis - This chapter presents the second case study of this work, the implementation of an extension to the SuSLik separation-logic based synthesizer to produce real-world executable C code, and investigates how synthesis can be used to allow verified systems to automate the manual burden of handling changes in specifications. The chapter starts a brief introduction to the wider context of this development, briefly presenting the concept of proof evaluators developed by Watanabe *et al.* [130] to extract synthesized programs from SuSLik into certified programs in verification frameworks and motivates the challenges with extending these techniques to produce real-world executable certified C code. The remainder of the chapter presents the technical details of this extension and describes how the aforementioned challenges were overcome. The chapter ends with a reflection on the insights revealed by the case-study on the role that synthesis plays in alleviating the pains of verified software evolution.

- Chapter 4 - Verified Software Maintenance through Repair - This chapter presents the third case study of this work, the development of a mostly automated tool for repairing verified real-world OCaml programs over their changes, and investigates how repair can be incorporated into software methodologies to simplify maintaining verified software as its implementation evolves. The chapter starts with a discussion of the context of the work and how it relates to the larger goals of the thesis in supporting verified software evolution. The remainder of the chapter describes the technical details of the case-study, presenting the design and implementation of the tool and evaluates its efficacy on real-world verified OCaml programs. The chapter concludes with a reflection on how repair-based processes can be used to support verified software evolution.

- Chapter 5 - A Unified Framework for Verified Software Evolution - Wrapping up the findings of this thesis, this chapter combines the techniques explored in the previous chapters and presents a unifying framework for handling the maintenance of verified software. To motivate this framework, the chapter considers a hypothetical verified static web page server and discusses how each of the prior techniques can be used to gracefully maintain this verified system.

- Chapter 6 - Conclusion and Future work - This chapter concludes the thesis, providing an overview of the work, its main contributions, and directions for future work. The chapter begins with a summary of the previous chapters, reviewing the main contributions of the work and then presents the final conclusions. Finally, the chapter ends by discussing directions for future work.

CHAPTER

2

VERIFIED SOFTWARE MAINTENANCE THROUGH COMPOSITION

This chapter presents the first case study of this work, the verification of approximate membership-query structures in the Coq proof assistant, and through it, investigates the use of composition for verified software maintenance. The chapter starts by discussing the challenges in exploiting composition in verification efforts and proposes the verification of randomised programs as a setting laden with numerous occurrences of these problems. Bloom-filters are introduced as an example of a probabilistic data structure that can serve as a natural case study for investigating the use of composition for maintenance and the thesis then presents the key properties of interest to be verified. The chapter then details the process of encoding Bloom filters within a theorem prover, and follows this with a highlight of the challenging aspects of the actual proof. Zooming out, the chapter presents how the analysis can be scaled up and ported over to a larger class of approximate membership-query structures, by decomposing the analysis into reusable components. Finally, the chapter ends with a review of the main insights and takeaways gained from this investigation.

Composition, and not to mention its many beatitudes for the aspiring proof engineer, are scarcely an idea that needs introduction. Indeed, many of the advances in formal verification over the past decade been discovered through optimising for more compositionality, and researchers have spent considerable time investigating the various ways in which proofs can be structured to be more modular and compositional. Broadly speaking, researchers have identified three general strategies that are sufficient to ensure composition in a verification effort: 1) through type systems and encoding verification conditions as typing constraints, 2) by use of program logics, and reasoning synactically over the program to certify the properties of interest and finally 3) by reasoning from first-principles and exploiting ad-hoc compositional mathematical arguments. By structuring proofs following these patterns, the cost of maintenance is reduced, as when the program changes, the effects are limited to the affected component.

The goals of this chapter are thus not to again follow this well-trodden path but rather to complement these prior works and provide a methodological study into the use of composition from the perspective of verified software maintenance. To this end, the thesis identifies a particular class of randomised data-structures—Approximate Membership Query Structures (AMQs)—whose intricate behaviours and complex probabilistic properties leaves them challenging to verify through program logics or type-systems, and certainly far outside the scope of automated approaches for maintenance such as synthesis or repair. Considering the verification of AMQs as a case-study, the thesis investigates how composition can be employed to aid verified software maintenance, treating each individual AMQ data-structure as a separate verification task, with the goal of maximising proof reuse between instances.

Beyond the methodological study, this chapter makes the following technical contributions:

- A Coq-based mechanised framework Ceramist, specialised for reasoning about AMQs.[1] Implemented as a Coq library, it provides a systematic decomposition of AMQs and their properties in terms of Coq modules and uses these interfaces to to derive certain properties "for free", as well as supporting proof-by-reduction arguments between classes of similar AMQs.
- A library of non-trivial theorems for establishing bounds on the false positive rates of AMQs, including the first formal proof of the closed form for Stirling numbers of the second kind [56].
- A collection of facts and tactics for effective construction of proofs of probabilistic properties in the style of Ssreflect reasoning [47, 86], that expresses its lemmas in terms of rewrites.
- A number of case study AMQs mechanised via Ceramist: ordinary [17] and counting [125] Bloom filters, quotient filters [14, 101], and Blocked AMQs [107].

The resulting mechanised development [52] is entirely *axiom-free*, and is compatible with Coq 8.11.0 [32] and MathComp 1.10 [86]. It relies on the infotheo library [3] for encoding discrete probabilities.

The remainder of this chapter will present the technical details of this case-study, starting with the verification of a particularly well-known example of an AMQ, the Bloom Filter, before generalising these findings to construct a general framework, Ceramist, for verifying a larger class of both classical AMQs and several entirely novel AMQs by making use of composition. The technical content of this chapter is a revision of work that been published at the CAV conference series and can be found here [53].

[1]Ceramist stands for **Cer**tified **A**pproximate **M**embersh**i**p **St**ructures.

2.1 Motivating Example

The main result of this chapter's case study is a framework, Ceramist, for reasoning about AMQ data structures where the underlying randomness arises from the interaction of hashing operations. To motivate the framework, the chapter starts by investigating the classical example of such an algorithm—a Bloom filter [17]—a structure whose properties have been oft mischaracterised, as shall be discussed.

2.1.1 The Basics of Bloom Filters

Bloom filters are probabilistic data structures that provide compact encodings of mathematical sets, trading increased space efficiency for a weaker membership test [17]. Specifically, when testing membership for a value *not* in the Bloom filter, there is a possibility that the query may be answered as positive. Thus a property of direct practical importance is the exact probability of this event, and how it is influenced and affected by the various other parameters of the implementation.

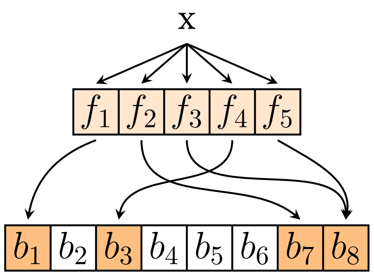


Figure 2.1: Implementation of a Bloom Filter

A Bloom filter bf is implemented as a binary vector of m bits (all initially zeros), paired with a sequence of k hash functions $f_1, \ldots, f_k$, collectively mapping each input value to a vector of k indices from $\{1 \ldots m\}$, the indices determine the bits set to true in the m-bit array Assuming an ideal selection of hash functions, the output of $f_1, \ldots, f_k$ on new values can be treated as a uniformly-drawn random vector. To insert a value x into the Bloom filter, each element of the "hash vector" produced from $f_1, \ldots, f_k$ can be treated as an index into bf and set the corresponding bits to ones. Finally, to test membership for an element x, one need only check that all k bits specified by the hash-vector are raised.

2.1.2 Properties of Bloom Filters

Given this model, there are two properties of importance: that of false positives and of false negatives.

False Negatives. It turns out that these definitions are sufficient to guarantee the lack of false-negatives with complete certainty, *i.e.*, irrespective of the random outcome of the hash functions. This follows from the fact that once a bit is raised, there are no permitted operations that will unset it.

Theorem 2.1.1 (No False Negatives)**.** *If $x \in bf$, then $\Pr[x \in_? bf] = 1$, where $x \in_? bf$ stands for the approximate membership test, while the relation $x \in bf$ means that x has been previously inserted into bf.*

False Positives. This property is more complex as the occurrence of a false positive is entirely dependent on the particular outcomes of the hash functions $f_1, \ldots, f_k$ and one needs to consider situations in which the hash functions happen to map some values to *overlapping* sets of indices. That is, after inserting a series of values xs, subsequent queries for $y \notin xs$ might incorrectly return true.

This leads to subtle dependencies that can invalidate the analysis, and have lead to a number of incorrect probabilistic bounds on the event, including in the analysis by Bloom in his original paper [17]. Specifically, Bloom first considered the probability that inserting l distinct items into the Bloom filter will set a particular bit b_i. From the independence of the hash functions, Bloom was then able to show that the probability of this event has a relatively simple closed-form representation:

Lemma 2.1.2 (Probability of a single bit being set)**.** *If the only values previously inserted into bf are $x_1, \ldots, x_l$, then the probability of a particular single bit at the position i being set is*

$$\Pr\left[i^{\text{th}} \text{ bit in } bf \text{ is set}\right] = 1 - \left(1 - \frac{1}{m}\right)^{kl}.$$

Alas, at this point Bloom made a subtle mistake, and claimed that the probability of a false positive was simply the probability of a single bit being set, raised to the power of k, reasoning that a false positive for an element $y \notin bf$ only occurs when all the k bits corresponding to the hash outputs are set.

Unfortunately, as was later pointed out by Bose *et al.* [19], as the bits specified by $f_1(x), \ldots, f_{k-1}(x)$ may overlap, it is not possible to guarantee the independence that is required for any simple relation between the probabilities. Bose *et al.* rectified the analysis by instead interpreting the bits within a Bloom filter as maintaining a set $\text{bits}(bf) \subseteq \mathbb{N}_{[0,\ldots,m-1]}$, corresponding to the indices of raised bits. With this interpretation, an element y only tests positive if the random set of indices produced by the hash functions on y is such that $\text{inds}(y) \subseteq \text{bits}(bf)$. Therefore, the chance of a positive result for $y \notin bf$ resolves to the chance that the random set of indices from hashing y is a subset of the union of $\text{inds}(x)$ for each $x \in bf$. The probability of this reduced event is described by the following theorem:

Theorem 2.1.3 (Probability of False Positives)**.** *If the only values inserted into bf are $x_1, \ldots, x_l$, then for* *any $y \notin bf$,*

$$\Pr\left[y \in_? bf\right] = \frac{1}{m^{k(l+1)}} \sum_{i=1}^{m} i^k i! \binom{m}{i} \left\{ kl \atop i \right\},$$

where $\left\{ s \atop t \right\}$ *stands for the* Stirling number of the second kind, *capturing the number of surjections from a set of size s to a set of size t.*

The key step in capturing these program properties is in treating the outcomes of hashes as *random variables* and then propagating this randomness to the results of the other operations. A formal treatment of program outcomes requires a suitable semantics, representing programs as distributions of such random variables. Before moving to mechanised proofs, it is necessary to first properly characterise this semantics, formally defining a notion of a probabilistic computation in Coq.

2.2 Encoding Bloom Filters in Coq

Moving on to the encoding of AMQs and their probabilistic behaviours in Coq, consider now how to translate the running example from mathematical notation to Gallina, Coq's language. The rest of this section will introduce each of the key components of this encoding through the lens of Bloom filters.

2.2.1 Probability Monad

In this case study, probabilistic computations are represented using an embedding following the style of the FCF library [102]. The code does not use FCF directly, due to its primary focus on cryptographic proofs, providing little support for proving probabilistic bounds directly, instead prioritising a reduction-based approach of expressing arbitrary computations as compositions of known distributions.

Following the adopted FCF notation, a term of type `Comp` A represents a probabilistic computation returning a value of type A, and is constructed using the standard monadic operators, with an additional primitive `rand` n that allows sampling from a uniform distribution over the range $\mathbb{Z}_n$:

$$\begin{aligned}
\texttt{ret} &: A \rightarrow \texttt{Comp}\ A \\
\texttt{bind} &: \texttt{Comp}\ A \rightarrow (A \rightarrow \texttt{Comp}\ B) \rightarrow \texttt{Comp}\ B \\
\texttt{rand} &: (n : \mathbb{N}) \rightarrow \texttt{Comp}\ (\mathbb{Z}_n)
\end{aligned}$$

It is then possible to implement a Haskell-style `do`-notation over this monad, allowing idiomatic and natural descriptions of probabilistic computations within Gallina. For example, the following seemingly imperative code is used to implement the query operation for the Bloom filter:

```
hash_res <-$ hash_vec_int x hashes; (* hash x using the hash functions *)
let (new_hashes, hash_vec) := hash_res in
(* check if all the corresponding bits are set *)
let qres := bf_query_int hash_vec bf in
(* return the query result and the new hashes *)
    ret (new_hashes, qres).
```

In the above listing, the code passes the queried value `x` along with the hash functions `hashes` to a probabilistic hashing operation `hash_vec_int` to hash `x` over each function in `hashes`. The result of this random operation is then bound to `hash_res` and split into its constituent components—a sequence of hash outputs `hash_vec` and an updated copy `new_hashes` of the hash functions, now incorporating the mapping for `x`. Then, having mapped the input into a sequence of indices, the implementation can simply query the Bloom filter for membership using a corresponding deterministic operation `bf_query_int` to check that all the bits specified by `hash_vec` are set. Finally, the computation is completed by returning the resulting query outcome `qres` and also the updated hash functions `new_hashes` making use of the `ret` operation to lift the result to a probabilistic outcome.

Using the code snippet above, it is possible to define the query operation `bf_query` as a function that maps a Bloom filter, a value to query, and a collection of hash functions to a probabilistic computation returning the query result and an updated set of hash functions. However, because the computation type does not impose any particular semantics on the programs written in it, this result only encodes the *syntax* of the probabilistic query and has no actual meaning without a separate interpretation.

Thus, given a Gallina term of type `Comp` A, any proofs about it must first evaluate it into a distribution over possible results to state properties on the probabilities of its outcomes. To do this, the case-study interprets the aforementioned monadic encoding using Ramsey's probability monad [111], which decomposes a complex distribution into composition of primitive ones bound together via conditional distributions. To capture this interpretation within Coq, the code uses the encoding of this monad from the infotheo library [2, 4], and provides a function `eval_dist : Comp` $A \to$ `dist` A that evaluates computations into distributions by recursively mapping them to the probability monad. Here, `dist A` represents infotheo's encoding of distributions over a finite support `A`, defined as a measure function `pmf` : $A \to \mathbb{R}^{+}$, and a proof that the sum of the measure over the support A produces 1.

This mapping from computations to distributions must be done to a program e (involving, *e.g.*, Bloom filter) before stating its probability bound. Therefore, this evaluation process is hidden behind a notation that allows stating probabilistic properties in a form closer to their mathematical counterparts:

$$\Pr[e = v] \triangleq (\texttt{eval\_dist}\ e)\ v$$
$$\Pr[e] \triangleq (\texttt{eval\_dist}\ e)\ \mathsf{true}$$

Above, v is an arbitrary element in the support of the distribution induced by e. Finally, the code uses a binding operator $\rhd$ to allow concise representation of dependent distributions: $e \rhd f \triangleq \texttt{bind}\ e\ f$.

2.2.2 Representing Properties of Bloom Filters

The state of a Bloom filter (`BF`) itself can be concisely encoded in Coq simply as a binary vector of a fixed length m, shown below using Ssreflect's `m.-tuple` data type to represent the vector of bits:

```
Record BF := mkBF { bloomfilter_state: m.-tuple bool }.
Definition bf_new : BF := (* construct a BF with all bits cleared *).
Definition bf_get_int i : BF → bool := (* retrieve BF's ith bit *).
```

The deterministic components of the implementation can be defined as pure functions taking a `BF` instance and a set of indices assumed to be obtained from a prior call to the associated hash functions:

$$\texttt{bf\_add\_int} : \texttt{BF} \to \texttt{seq}\ \mathbb{Z}_m \to \texttt{BF}$$
$$\texttt{bf\_query\_int} : \texttt{BF} \to \texttt{seq}\ \mathbb{Z}_m \to \mathbb{B}$$

That is, `bf_add_int` takes the Bloom filter state and a sequence of indices to insert and returns a new state with the requested bits also set. Conversely, `bf_query_int` returns true *iff* all the queried indices are set. These pure operations are then called within a probabilistic wrapper that handles hashing the input and the book-keeping associated with hashing to provide the standard interface for AMQs:

$$\texttt{bf\_add} : B \to (\texttt{HashVec}\ B * \texttt{BF}) \to \texttt{Comp}\ (\texttt{HashVec}\ B * \texttt{BF})$$
$$\texttt{bf\_query} : B \to (\texttt{HashVec}\ B * \texttt{BF}) \to \texttt{Comp}\ (\texttt{HashVec}\ B * \mathbb{B})$$

The component `HashVec` B (to be defined in Subsection 2.2.3), parameterised over an input type B, is used to keep track of the *known results* of all the involved hash functions and is provided as an external argument that must be provided to the function rather than being a part of the data structure to reflect typical uses of AMQs, wherein the hash operation is pre-determined and shared by *all* instances.

With these definitions and notation, it is now possible to state the key theorems of interest within Coq:[2]

Theorem 2.2.1 (No False Negatives)**.** *For any Bloom filter state bf, a vector of hash functions hs, after having inserted an element x into bf, followed by a series xs of other inserted elements, the result of query $x \in_? bf$ is always* true*. That is, in terms of probabilities:*

$$\Pr\left[\texttt{bf\_add}\ x\ (hs, bf) \rhd \texttt{bf\_addm}\ xs \rhd \texttt{bf\_query}\ x\right] = 1.$$

Lemma 2.2.2 (Probability of Flipping a Single Bit)**.** *For a vector of hash functions hs of length k, after inserting a series of l distinct values xs, all unseen in hs, into an empty Bloom filter bf, represented by a vector of m bits, the probability of its any index i being set is*

$$\Pr\left[\texttt{bf\_addm}\ xs\ (hs, \texttt{bf\_new}) \rhd \texttt{bf\_get}\ i\right] = 1 - \left(1 - \frac{1}{m}\right)^{kl}.$$

Here, `bf_get` *is a simple embedding of the pure function* `bf_get_int` *into a probabilistic computation.*

Theorem 2.2.3 (Probability of a False Positive)**.** *After having inserted a series of l distinct values xs, all unseen in hs, into an empty Bloom filter bf, for any unseen $y \notin xs$, the probability of a subsequent query $y \in_? bf$ for y returning true is given as*

$$\Pr\left[\texttt{bf\_addm}\ xs\ (hs, \texttt{bf\_new}) \rhd \texttt{bf\_query}\ y\right] = \frac{1}{m^{k(l+1)}} \sum_{i=1}^{m} i^k i! \binom{m}{i} \left\{ {kl \atop i} \right\}.$$

The proof of this theorem required developing *the first axiom-free mechanised proof* of the closed form for Stirling numbers of the second kind [56], a standalone fact that can be reused by future developments.

In the definitions above, the output of the hashing operation is used as the boundary between the deterministic and probabilistic components of the Bloom filter. For instance, in the earlier description of the Bloom filter query operation in Subsection 2.2.1, it was possible to implement the entire operation with the only probabilistic operation being the call `hash_vec_int x hashes`. In general, structuring AMQ operations as manipulations with hash outputs via *pure* deterministic functions allows decomposing the reasoning about the data structure into a series of specialised properties about its deterministic primitives and a separate set of reusable properties on its hash operations, improving modularity.

[2]`bf_addm` is a trivial generalisation of the insertion to multiple elements.

2.2.3 Reasoning about Hash Operations

Hash operations in the development are encoded using a random oracle-based implementation. In particular, in order to keep track of *seen* hashes learnt by hashing previously observed values, the *state* of a hash function from elements of type `B` to a range $\mathbb{Z}_m$ is represented using a finite map to ensure that hashing produces consistent outputs — *i.e.* that previously hashed values produce the same result:

```
Definition HashState B := FixedMap B 'I_m.
```

The hash state is then paired with a randomised function for hashing that generates uniformly random outputs for unseen values, and otherwise returns the value as from its prior invocations:

```
Definition hash value state : Comp (HashState B * B) :=
  match find value state with
  | Some(output) ⇒ ret (state, output)
  | None ⇒ rnd <-$ rand m;
           new_state <- put value rnd state;
           ret (new_state, rnd)
  end.
```

A *hash vector* is a generalisation of this structure to a vector of states of k independent hash functions:

```
Definition HashVec B := k.-tuple HashState B.
```

The corresponding hash operation over the hash vector, `hash_vec_int`, is then defined as a function taking a value and the current hash vector and then returning a pair of the updated hash vector and associated random vector, internally calling out to `hash` to compute individual hash outputs.

This random oracle-based implementation naturally allows formulating several helper theorems for simplifying probabilistic computations using hashes by considering whether the hashed values *have been seen before or not*. For example, if one knows that a value x has not been seen before, then intuitively one can reason that the possibility of obtaining any particular choice of a vector of indices would be equivalent to obtaining the same vector by a draw from a corresponding uniform distribution. This natural intuition can be mechanically formalised in the form of the following theorem:

Theorem 2.2.4 (Uniform Hash Output). *For any two hash vectors hs, hs' of length k, a value x that has not been hashed before, and an output vector ιs of length m obtained by hashing x via hs, if the state of hs' has the same mappings as hs and also maps x to ιs, the probability of obtaining the pair $(hs', \iota s)$ is uniform:* $\Pr\left[\texttt{hash\_vec\_int}\ x\ hs = (hs', \iota s)\right] = \left(\frac{1}{m}\right)^k$

Similarly, there are also cases where a proof requires hashing a value that *has already been seen*. In these cases, if the result a value hashes to is known, it is possible to prove a certainty on the outcome:

Theorem 2.2.5 (Hash Consistency). *For any hash vector hs, a value x, if hs maps x to outputs ιs, then hashing x again will certainly produce ιs and not change hs, that is,* $\Pr[\texttt{hash\_vec\_int}\ x\ hs = (hs, \iota s)] = 1$.

By combining these types of probabilistic properties about hashes with the earlier Bloom filter operations, it was possible to prove the prior theorems about Bloom filters operations by reasoning primarily about the core logical interactions of the *deterministic components* of the data structure rather than the random components. This decomposition is not just applicable to the case of Bloom filters, but can be extended into a general framework for obtaining modular proofs of AMQs, as will be shown in the next section.

2.3 Generalising to AMQs at Large

Zooming out from the prior discussion of Bloom filters, it is time to present the complete Ceramist framework in its full generality, describing how its careful high-level design exploits composition in terms of the various interfaces it requires to instantiate to obtain verified AMQ implementations.

The core of the framework revolves around the decomposition of an AMQ data structure into interfaces for hashing (AMQHash) and state (AMQ), generalising and encapsulating the specific decomposition used for Bloom filters (hash vectors and bit vectors respectively). More specifically, the AMQHash interface captures the generally *reusable* probabilistic properties of the hashing operation, while the AMQ interface captures the *bespoke* deterministic interactions of the state with the hash outcomes.

2.3.1 AMQHash Interface

The AMQHash interface carefully generalises the nuanced behaviours of hash vectors (Subsection 2.2.3) to provide a generic unifying description of the various hashing operations typcially used in AMQs.

The interface abstracts over the specific types used in the prior hashing operations (such as, *e.g.*, `HashVec B`) by treating them as opaque parameters, and using a parameter `AMQHashState` to represent the state of the hash operation. The types `Key` and `Value` are used to encode the hash inputs and outputs respectively, and finally, a deterministic operation `AMQHash_add_internal` is used to encode the basic interaction of the state with the outputs and inputs. As the definition of probability distributions used in the project are defined over finite supports, all types are constrained to also be finite.

```
Parameter AMQHashState : finType.
Parameter Key : finType.
Parameter Value : finType.
Parameter AMQHash_add_internal :
        AMQHashState → Key → Value → AMQHashState.
```

For example, for a single hash function, the state parameter `AMQHashState` would be `HashState B`, while for a hash vector this would instead be `HashVec B` to represent the vector of states.

To invoke this hash and the corresponding hash state from probabilistic computations, the interface then requires a separate probabilistic operation that will take the hash state and randomly generate an output as has been seen before (*e.g.*, `hash` for single hashes and `hash_vec_int` for hash vectors):

```
Parameter AMQHash_hash: Key → AMQHashState → Comp (AMQHash * Value).
```

Then, to abstractly capture the reasoning patterns about outcomes of hash operations as done with Bloom filters in Subsection 2.2.3, the interface assumes two predicates on the hash state about its contents:

```
Parameter AMQHash_hashstate_contains: AMQHashState → Key → Value → bool.
Parameter AMQHash_hashstate_unseen: AMQHashState → Key → bool.
```

These components are then combined together to produce more abstract general formulations of the previous Theorems 2.2.4 and 2.2.5 on hash operation outputs and consistency.

Property 1 (Generalised Uniform Hash Output). *There exists a probability p_{hash}, such that for any two AMQ hash states hs, hs', a value x that is unseen, and an output ιs obtained by hashing x via hs, if the state of hs' has the same mappings as hs and also maps x to ιs, the probability of obtaining the pair $(hs', \iota s)$ is given by:* $\Pr\left[\texttt{AMQHash\_hash}\ x\ hs = (hs', \iota s)\right] = p_{hash}$.

Property 2 (Generalised Hash Consistency). *For any AMQ hash state hs, a value x, if hs maps x to an output ιs, then hashing x again will certainly produce ιs and not change hs:*

$$\Pr\left[\texttt{AMQhash\_hash}\ x\ hs = (hs, \iota s)\right] = 1$$

Proofs of these corresponding properties must also be provided to instantiate the AMQHash interface. Conversely, components operating over this interface can assume their existence, and use them to abstractly perform the same kinds of simplifications as done with Bloom filters, resolving many probabilistic proofs to dealing with deterministic properties on the AMQ states.

2.3.2 The AMQ Interface

Building on top of an abstract AMQHash component, the AMQ interface then provides a unified view of the state of an AMQ and how it interacts with the output type `Value` of a particular hashing operation.

As before, the interface begins by abstracting the specific types and operations of the previous analysis of Bloom filters, first introducing a type `AMQState` to capture the state of the AMQ, and then assuming deterministic implementations of the typical *add* and *query* operations of an AMQ:

```
Parameter AMQ_add_internal: AMQState → Value → AMQState.
Parameter AMQ_query_internal: AMQState → Value → bool.
```

In the case of Bloom filters, these would be instantiated with the `BF`, `bf_add_int` and `bf_query_int` operations (*cf.* Subsection 2.2.2), thereby setting the associated hashing operation to the hash vector (Subsection 2.2.3).

When it comes to reasoning about the behaviours of these operations, the interface diverges slightly from that of the Bloom filter by conditioning on an assumption that the state has sufficient capacity:

```
Parameter AMQ_available_capacity: AMQState → nat → bool.
```

While the Bloom filter has no real deterministic notion of a capacity, this cannot be said of all AMQs in general, such as the Counting Bloom filter or Quotient filter, as will be presented later.

With these definitions in hand, the expected behaviours of well-formed AMQ operations themselves are then characterised in the interface using a corresponding series of associated assumptions:

Property 3 (AMQ insertion validity). *For a state s with sufficient capacity, inserting any hash output ιs into s via* `AMQ_add_internal` *will produce a new state s' for which any subsequent queries for ιs via* `AMQ_query_internal` *will return* true.

Property 4 (AMQ query preservation). *For any AMQ state s with sufficient remaining capacity, if queries for a particular hash output ιs in s via* `AMQ_query_internal` *happen to return* true*, then inserting any further outputs $\iota s'$ into s will return a state for which queries for ιs will* still *return* true.

Though these assumptions seemingly place strict restrictions on the permitted operations, these properties were found to be satisfied by most common AMQ structures. One potential reason for this might be because they are in fact *sufficient* to ensure the No-False-Negatives property standard of most AMQs:

Theorem 2.3.1 (Generalised No False Negatives). *For any AMQ state s, a corresponding hash state hs, after having inserted an element x into s, followed by a series xs of other inserted elements, the result of*

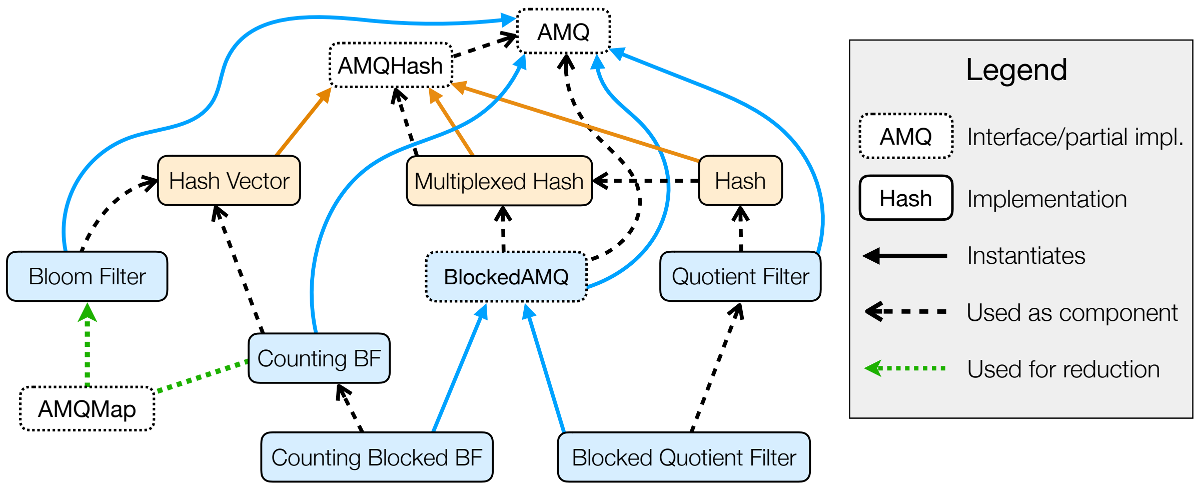


Figure 2.2: Overview of Ceramist and the dependencies between its components.

query for x is always true. *That is,*

$$\Pr\left[\texttt{AMQ\_add}\ x\ (hs, s)\ \rhd\ \texttt{AMQ\_addm}\ xs\ \rhd\ \texttt{AMQ\_query}\ x\right] = 1.$$

Here, `AMQ_add`, `AMQ_addm`, and `AMQ_query` are the straightforward generalisations of the previously seen probabilistic wrappers of Bloom filters (*cf.* Subsection 2.2.1) that both keep track of the bookkeeping associated with hashing and delegate the actual implementation to the internal deterministic operations.

The generalised Theorem 2.3.1 illustrates one of the key facilities provided by the Ceramist framework use of composition, wherein by simply providing components satisfying the AMQHash and AMQ interfaces, it is possible to obtain proofs of standard probabilistic properties or simplifications *for free*.

The diagram in Figure 2.2 provides a high-level overview of the interfaces of Ceramist, their specific instances, and dependencies between them, demonstrating Ceramist's take on compositional reasoning and proof reuse. For example, Bloom filter implementation instantiates the AMQ interface implementation and uses, as a component, hash vectors, which themselves instantiate AMQHash used by AMQ. The Bloom filter is also used as a proof reduction target by Counting Bloom filter. The following sections will elaborate on this and other noteworthy dependencies between interfaces and instances of Ceramist.

2.3.3 Counting Bloom Filters through Composition

To provide a concrete demonstration of the use of the AMQ interface for composition and proof reuse, consider now a new running example—Counting Bloom filters [125]. A Counting Bloom filter is a variant of the Bloom filter in which individual bits are replaced with counters, thereby allowing the removal of elements. The implementation of the structure closely follows the Bloom filter, generalising the logic from bits to counters: insertion increments the counters specified by the hash outputs, while queries treat counters as set if greater than 0. The remainder of this section will discuss the process of encoding and verifying the Counting Bloom filter for the standard AMQ properties using the facilities provided by Ceramist. Furthermore, to demonstrate how new properties and theorems can be built on top of the framework constructions produced by composition, the development also proves two novel domain-specific properties of Counting Bloom filters, to be outlined in Section 2.4.

When applying the Ceramist framework, as the Counting Bloom filter uses the same hashing strategy as the Bloom filter, the hash interface can be instantiated with the Hash Vector structure used for the Bloom filter, entirely reusing the earlier proofs on hash vectors. Next, in order to instantiate the AMQ interface, the state parameter can be defined as a vector of bounded integers, all initially set to 0:

```
Record CF := mkCF { countingbloomfilter_state: m.-tuple ℤp }.
Definition cf_new : CF := (* a new CF with all counters set to 0 *).
```

As mentioned before, the *add* operation is implemented similarly to the Bloom filter but simply increments counters rather than setting bits, and the *query* operation treats non-zero counters as raised.

$$\texttt{cf\_add\_int} : \texttt{CF} \to \texttt{seq}\,\mathbb{Z}_m \to \texttt{CF}$$
$$\texttt{cf\_query\_int} : \texttt{CF} \to \texttt{seq}\,\mathbb{Z}_m \to \mathbb{B}$$

To prevent integer overflows, the counters in the Counting Bloom filter are bounded to some range $\mathbb{Z}_p$, so the overall data structure too has a maximum capacity. It would not be possible to insert any values if doing such would raise any of the counters above their maximum. To account for this, the capacity parameter of the AMQ interface is instantiated with a simple predicate `cf_available_capacity` that verifies that the structure can support l further inserts by ensuring that each counter has at least $k * l$ spaces free (where k is the number of hash functions used by the data structure).

The add operation can be shown to be monotone on any counter when there is sufficient capacity (Property 3). The remaining properties of the operations also trivially follow, thereby completing the instantiation, and allowing the automatic derivation of the No-False-Negatives result via Theorem 2.3.1.

2.3.4 Proofs about False Positive Probabilities by Reduction

As the mechanisms of the Counting Bloom filter closely match those of the Bloom filter, it seems reasonable to expect that the same probabilistic bounds might also apply to the data structure. Drawing inspiration from this intuition, Ceramist provides an AMQMAP interface, which allows reusing proven properties about one data structure to derive probabilistic bounds for a new data structure, internally using reduction arguments to compositionally map properties from one AMQ data structure to another.

The AMQMAP interface is parameterised by two AMQ data structures, AMQ A and B, using the same hashing operation. It is assumed that corresponding bounds on False Positive rates have already been proven for AMQ B, while have not for AMQ A. The interface first assumes the existence of some mapping from the state of AMQ A to AMQ B, which satisfies a number of properties:

```
Parameter AMQ_state_map: A.AMQState → B.AMQState.
```

In the case of the Counting Bloom filter example, this mapping would convert the Counting Bloom filter state to a bit vector by mapping each counter to a raised bit if its value is greater than 0. In order to automatically derive the false positive rate property, the AMQMAP interface then further requires the behaviour of this mapping to be proven to satisfy a number of additional assumptions:

Property 5 (AMQ Mapping Add Commutativity)**.** *Adding a hash output to the AMQ B obtained by applying the mapping to an instance of AMQ A produces the same result as first adding a hash output to AMQ A and then applying the mapping to the result.*

Property 6 (AMQ Mapping Query Preservation)**.** *Applying B's query operation to the result of mapping an instance of AMQ A produces the same result as applying A's query operation directly.*

When reducing Counting Bloom filters (A) to Bloom filters (B), both properties follow from the fact that after incrementing a counter, it will have a value greater than 0 and thus be mapped to a raised bit.

Having instantiated the AMQMAP interface with the corresponding function and proofs about it, it is now possible to exploit composition and automatically show that the false positive rate of Bloom filters holds for Counting Bloom filters entirely for free through the following generalised lemma:

Theorem 2.3.2 (AMQ False Positive Reduction)**.** *For any two AMQs A, B, related by the* AMQMAP *interface, if the false positive rate for B after inserting l items is given by the function f on l, then the false* *positive rate for A is also given by f on l. That is, in terms of probabilities:*

$$\Pr\left[\texttt{B.AMQ\_addm}\ xs\ (hs, \texttt{B.AMQ\_new}) \rhd \texttt{B.AMQ\_query}\ y\right] = f(\texttt{length}\ xs) \implies$$
$$\Pr\left[\texttt{A.AMQ\_addm}\ xs\ (hs, \texttt{A.AMQ\_new}) \rhd \texttt{A.AMQ\_query}\ y\right] = f(\texttt{length}\ xs).$$

2.4 Additional Properties of Counting Bloom Filters

While the No-False-Negatives and false positive rate are practically important aspects of an AMQ, in the case of a Counting Bloom filter, there are a few other behaviours of the structure that are of importance. One such property is the ability to remove elements from a Counting Bloom filter without affecting queries for other ones, by decrementing the counters corresponding to the removed element.

To demonstrate the flexibility of the framework, the development also includes a mechanised proof of the validity of this removal operation, which had not ever been previously formalised:

Theorem 2.4.1 (Counting Bloom filter removal)**.** *For any Counting Bloom filter cf with sufficient capacity and associated hashes hs, removing a previously inserted value x' will not change the query for any other previously inserted value x, that is:*

$$\Pr\left[\texttt{cf\_add}\ x'\ (hs, cf) \rhd \texttt{cf\_add}\ x \rhd \texttt{cf\_remove}\ x' \rhd \texttt{cf\_query}\ x\right] = 1.$$

The operation `cf_remove` from the theorem statement deletes a value from the Counting Bloom filter by decrementing the associated counters, and is provided as a custom operation externally to the other Ceramist components, as removal operations are not a typical operation in AMQ interfaces.

The development also provides a proof of another specialised property of the Counting Bloom filte structure—that inserting any value will increase the total sum of the counters by a fixed amount. This property characterises how the modified state of the Counting Bloom filter allows tracking more detailed information, than just element membership, in terms of the exact number of insertions.

Theorem 2.4.2 (Certainty of Counter Increments)**.** *For any counting Bloom filter cf, a value y that was not previously inserted into cf, if the sum of the values of all counters d_i in cf is l, then after inserting y, the sum of the counters will certainly increment by k, that is:* $\Pr\left[\sum_{d_i \in cf} d_i = l + k\right] = 1.$

2.5 Proof Automation for Probabilistic Sums

The narrative has, until now, avoided discussing the technical details of how properties of probabilistic computations can be composed, and also the specifics of how proofs in the framework itself are structured. As it turns out, most of this process resolves to reasoning about summations over real values as encoded by Ssreflect's bigop library. The development involves extensive manipulations of nested summations, using a tactic library for rewriting under summations by Martin-Dorel and Soloviev [87].

This section outlines the most essential proof principles facilitating the proofs-by-rewriting about probabilistic sums. While most of the provided rewriting primitives are standalone general equality facts, some of the proof techniques are better understood as combining a series of rewritings into a more general rewriting pattern. To delineate these two, the text will use **Pattern** to refer to a general pattern the library supports by means of a dedicated tactic, while **Lemma** will refer to standalone equalities.

2.5.1 The Normal Form for Composed Probabilistic Computations

When stating properties on outcomes of a probabilistic computation (*cf.* Subsection 2.2.1), the computation must first be recursively evaluated into a distribution, where the intermediate results are combined using the probabilistic `bind` operator. Therefore, when decomposing a probabilistic property into smaller subproofs, one must rely on its semantics that is defined for discrete distributions as follows:

$$\texttt{bind\_dist}\ (P : \texttt{dist}\ A)\ (f : A \to \texttt{dist}\ B) \triangleq \sum_{a:\ A} \sum_{b:\ B} P\ a\ \times\ (f\ a)\ b$$

Expanding this definition, one can represent any statement on the outcome of a probabilistic computation in a *normal form* composed of only nested summations over a product of the probabilities of each intermediate computational step. This paramount transformation is captured as the following pattern:

Pattern 2.5.1 (Bind normalisation)**.**

$$\Pr\left[(c_1 \rhd \ldots \rhd c_m) = v\right] = \sum_{v_1} \cdots \sum_{v_{m-1}} \Pr\left[c_1 = v_1\right] \times \cdots \times \Pr\left[c_m\ v_{m-1} = v\right]$$

Here, $c_i\ v_{i-1} = v_i$ is used to denote the event in which the result of evaluating the command $c_i\ v_{i-1}$ is v_i, where v_{i-1} is the result of evaluating the previous command in the chain. This basic transformation then allows decomposing the proof of a given probabilistic property into proving simpler statements on its individual substeps. For instance, consider the implementation of Bloom filter's query operation from Section 2.2.1. When proving properties of the result of a particular query (as in Theorem 2.2.1), this rule is used to decompose the program into its component parts, namely as being the product of a hash invocation $\Pr[\texttt{hash\_vec\_int}\ x\ hs]$ and the deterministic query operation `bf_query_int`. This allows dealing with the hash operation and the deterministic component *separately* by applying subsequent rewritings to each factor on the right-hand side of the above equality.

2.5.2 Probabilistic Summation Patterns

Having resolved a property into its normal form via a tactic implementing Pattern 2.5.1, the subsequent reductions rely on the following straightforward patterns and lemmas.

Sequential composition. When reasoning about the properties of composite programs, it is common for some subprogram e to return a probabilistic result that is then used as the arguments for a probabilistic function f. This composition is encapsulated by the operation $e \rhd f$, as used by Theorems 2.2.1, 2.2.2, and 2.2.3. These programs, once converted to normal form, are characterised by having factors in its product that simply evaluate the probability of the final statement $\texttt{ret}\ v'$ producing a given value v_k:

$$\sum_{v_1} \cdots \sum_{v_{m-1}} \underbrace{\Pr[c_1 = v_1] \times \cdots \Pr[\texttt{ret}\ v' = v_k]}_{e} \underbrace{\cdots \times \Pr[c_m\ v_{m-1} = v]}_{f}$$

Since the return operation is simply defined as a delta distribution with a single peak at the given return value v', the whole statement can be simplified by just removing the entire summation over v_k, and replacing all corresponding occurrences of v_k with v', through the following pattern:

Pattern 2.5.2 (Probability of a Sequential Composition)**.**

$$\sum_{v_1} \cdots \sum_{v_{m-1}} \Pr[\texttt{ret}\ v' = v_1] \cdots \times \Pr[c_m\ v_{m-1} = v]] =$$
$$\sum_{v_2} \cdots \sum_{v_{m-1}} \Pr[[v'/v_1](c_2\ v_1) = v_2] \times \cdots \times \Pr[[v'/v_1]c_m\ v_{m-1} = v]$$

Notice that, without loss of generality, Pattern 2.5.2 assumes that the v'-containing factor is in the head. Ceramist's tactic libraries will implicitly rewrite proof statements into to this form by default.

Plausible statement sequencing. One common issue with the normal form, is that, as each statement is evaluated over the entirety of its support, some of the dependencies between statements are obscured. That is, the outputs of one statement may in fact be constrained to *some subset* of the complete support. To recover these dependencies, the development makes use of the following theorem, that allows reducing computations under the assumption that their inputs are plausible:

Lemma 2.5.3 (Plausible Sequencing). *For any computation sequence $c_1 \rhd c_2$, if it is possible to reduce the computation $c_2\ x$ to a simpler form $c_3\ x$ when x is amongst plausible outcomes of c_1, (*i.e., $\Pr[c_1 = x] \neq 0$ *holds) then it is possible to rewrite c_2 to c_3 without changing the resulting distribution:*

$$\sum_x \sum_y \Pr[c_1 = x] \times \Pr[c_2\ x = y] = \sum_x \sum_y \Pr[c_1 = x] \times \Pr[c_3\ x = y]$$

Plausible outcomes. As seen in the previous paragraph, sometimes proofs may involve knowledge that a particular value v is a plausible outcome for a composite probabilistic computation $c_1 \rhd \ldots \rhd c_m$:

$$\sum_{v_1} \cdots \sum_{v_{m-1}} \Pr[c_1 = v_1] \times \cdots \times \Pr[c_m\ v_{m-1} = v] \neq 0$$

This fact in itself is not particularly helpful as it does not immediately provide any usable constraints on the value v. However, this inequality can now be turned into a conjunction of inequalities for individual probabilities, thus getting more information about the intermediate steps of the computation:

Pattern 2.5.4. *If* $\sum_{v_1} \cdots \sum_{v_{m-1}} \Pr[c_1 = v_1] \times \cdots \times \Pr[c_m\ v_{m-1} = v] \neq 0$, *then there exist* $v_1, \ldots, v_{m-1}$ *such that* $\Pr[c_1 = v_1] \neq 0 \wedge \cdots \wedge \Pr[c_m = v] \neq 0$.

This transformation is possible due to the fact that probabilities are always non-negative, thus if a summation is positive, there must exist at least one element in the summation that is also positive.

Distributions over tuples

When reducing computations to their normal forms, it is common to end up with goals as below:

$$\Pr[c \rhd \lambda v.\ \texttt{ret}\ (P\ v \wedge Q\ v)] =$$
$$\Pr[c \rhd \lambda v.\ \texttt{ret}\ P\ v] \times \Pr[c \rhd \lambda v.\ \texttt{ret}\ Q\ v]$$

For instance, when proving the lack of false negatives (Theorem 2.2.1), the recursive definition of the query operation requires decomposing the event of a successful query for a given set of indices $\iota :: \iota s$ into a conjunction of the facts that the first index ι and that the remaining indices ιs are set.

This property turns out to be equivalent to proving *independence*, which is defined within the framework as the particular situation in which the probability of two or more simultaneous events can be simply decomposed into the product of each individual event independently.

$$\sum_v \Pr[c_1 = v] \times \Pr[c_2 = v] =$$
$$\sum_v \Pr[c_1 = v] \times \sum_v \Pr[c_2 = v]$$

The general approach to proving these kinds of properties would then be to re-index the summand by a *bijective mapping* into disjoint sub-terms, such that each internal statement holds on a separate term.

Pattern 2.5.5 (Split for independent events). *If $f : A \to (A_1, A_2)$ is a bijective mapping that decomposes the outcome of a computation c into two distinct subcomputations c_1, c_2, and predicates P' and Q' are such that $P = P' \circ \cdot_1 \circ f$ and $Q = Q' \circ \cdot_2 \circ f$,*[3] *then*

$$\Pr[c \rhd \lambda v.\ \texttt{ret}\ (P\ v \wedge Q\ v)] =$$
$$\Pr[c_1 \rhd \lambda v_1.\ \texttt{ret}\ (P'\ v_1)] \times \Pr[c_2 \rhd \lambda v_2.\ \texttt{ret}\ (Q'\ v_2)]$$

This reindexing thus allows splitting an event into a product of two summations of the projections of f's images, from which the remainder of the independence proof follows by showing that the factors of the multiplication on either side of the equality are the same. This strategy was also found to act as a good heuristic for event independence: whenever the author was unable to find a suitable bijection, it would often later turn out that the events in question were *not independent* at all.

As a special case, when reasoning about computations returning fixed length sequences, such as the $k.-$ tuples of hashes forming hash vectors, it was found that a common strategy for demonstrating independence of the computations' outcomes was to split the sequence into smaller subsequences.

For simple properties, it suffices to divide the sequence summand into its head and tail:

Lemma 2.5.6 (Distributing sums over cons).

$$\sum_{vs\colon (m+1).-\textit{tuple } A} \Pr[c = vs; P\ vs] = \sum_{v\colon A} \sum_{vs\colon m.-\textit{tuple } A} \Pr[c = v :: vs; P\ (v :: vs)]$$

Other more complex properties sometimes required larger partitions:

[3]The formalisation only requires point-wise predicate equality.

Lemma 2.5.7 (Distributing sums over concatenation)**.**

$$\sum_{vs\colon (m+l).-tuple\ A} \Pr\left[c = vs; P\ vs\right] =$$
$$\sum_{vs\colon m.-tuple\ A} \ \sum_{vs'\colon l.-tuple\ A} \Pr\left[c = vs \ +\!\!+\ vs'; P\ (vs \ +\!\!+\ vs')\right]$$

2.5.3 Implementation Details

For completeness, this section discusses some of the technical details of the encoding used for embedding the properties presented in Sections 2.5.1-2.5.2 into the Coq proof assistant.

Using finite types. The careful reader may have noticed that the prior examples technically only required that the types being reasoned about are countable, however the development itself actually enforces a slightly stronger constraint—that the types must themselves be finite. This primarily arises from the use of infotheo's probability monad for execution, as it limits the definition of distributions to those with finite supports, implemented by means of Ssreflect's `finType` interface.

While enforcing this constraint imposes its own costs, such as necessitating the pervasive use of dependent types, it was found find that constraining types to be finite generally produced more practical results. Specifically, by having to bound all types to finite ranges, the framework was required to ensure that all theorems are stated in relation to constraints on a finite capacity. For example, when the development was used to prove properties on Counting Bloom filters, the author was forced to also consider the possibility of integer overflow by stating the theorems on the condition that all cells in the structure have enough space to not exceed their maximum capacity.

Additionally, as the development separates between the encoding and evaluation of probabilistic computations using the `comp` A and `dist` A types, the core definitions are independent of the formulation of distributions. Future work could potentially lift this constraint by using a different distribution definition and supplying an alternate definition of the `eval_dist` function.

Tactics and automation. Given that the normal form introduces additional summations for each computational step, a large part of the proof effort was in manipulating and reasoning about deeply nested sums. The core difficulty in automating this problem arises from the fact that the body of any summation is implemented as a Coq's lambda-expression. This means that rewriting within a nested summation cannot be performed using the default tactics and either required Coq's `Setoid` rewriting rules[4] or a custom rewrite tactic that implicitly also invokes functional extensionality.

[4] https://coq.inria.fr/refman/addendum/generalized-rewriting.html

To simplify this task, the development adopts the tactic library developed by Martin-Dorel and Soloviev from their work certifying boolean games in Coq [87]. This library provides a transformer tactic `under` which can prefix any other tactic to allow applying the tactic under the context of the summation.

Summary of the development. By composing these components together, the resulting development becomes a comprehensive toolbox for effectively reasoning about probabilistic computations. It was found that that the summation patterns end up encapsulating most of the book-keeping associated with the encoding of probabilistic computations, which, combined with the AMQ/AMQHash decomposition from Section 2.3, allows for a fairly straightforward approach for verifying properties of AMQs.

2.5.4 A Simple Proof of Generalised No False Negatives Theorem

To showcase the fluid interaction of the proof principles in action, consider now the proof of the generalised No-False-Negatives Theorem 2.3.1, stating the following:

$$\Pr\left[\underbrace{\texttt{AMQ\_add}\ x\ (hs, s)}_{(a),(b)} \rhd \underbrace{\texttt{AMQ\_addm}\ xs}_{(c)} \rhd \underbrace{\texttt{AMQ\_query}\ x}_{(d),(e)}\right] = 1 \tag{2.1}$$

As with most of probabilistic proofs in the framework, the proof begins by applying the normalisation Pattern 2.5.1 to reduce the computation into its normal form:

$$\sum_{\iota s_0, hs_0} \sum_{s_0} \sum_{s_1, hs_1} \sum_{\iota s_2, hs_2} \begin{pmatrix} (a) & \Pr\left[\texttt{AMQHash\_hash}\ x\ hs = (\iota s_0, hs_0)\right] & \times \\ (b) & \Pr\left[\texttt{ret}\ (\texttt{AMQ\_add\_internal}\ s\ \iota s_0) = s_0\right] & \times \\ (c) & \Pr\left[\texttt{AMQ\_addm}\ xs\ (s_0, hs_0) = (s_1, hs_1)\right] & \times \\ (d) & \Pr\left[\texttt{AMQHash\_hash}\ x\ hs_1 = (\iota s_2, hs_2)\right] & \times \\ (e) & \Pr\left[\texttt{ret}\ (\texttt{AMQ\_query\_internal}\ s_1\ \iota s_2)\right] & \end{pmatrix}$$

The above equation labels the factors to be rewritten as (a)–(e) for the convenience of the presentation, indicating the correspondence to the components of the statement (2.1). From here, as all values are assumed to be unseen, the proof then uses Property 1 in conjunction with the sequencing Pattern 2.5.2 as seen before to reduce the two terms (a) and (b) in the summation as follows:

$$\sum_{\iota s_0} \sum_{s_1, hs_1} \sum_{\iota s_2, hs_2} \begin{pmatrix} (a) & p_{\text{hash}} & \times \\ (c) & \Pr\left[\texttt{AMQ\_addm}\ xs\ ((s \leftarrow_{\text{add}} \iota s_0), (hs \leftarrow_{\text{hash}} (x : \iota s_0))) = (s_1, hs_1)\right] & \times \\ (d) & \Pr\left[\texttt{AMQHash\_hash}\ x\ hs_1 = (\iota s_2, hs_2)\right] & \times \\ (e) & \Pr\left[\texttt{AMQ\_query\_internal}\ s_1\ \iota s_2\right] & \end{pmatrix}$$

Here, p_{hash} is the probability from the statement of Property 1. Here, the presentation uses the notations $s \leftarrow_{\text{add}} \iota s_0$ and $hs \leftarrow_{\text{hash}} (x : \iota s_0)$ to denote the deterministic operations `AMQ_add_internal` and `AMQHash_add_internal` respectively. Then, using Pattern 2.5.4 for decomposing plausible outcomes, it is possible to separately show that any plausible hs_1 from `AMQ_addm` must map x to ιs_0, as hash operations preserve mappings. Combining this fact with Lemma 2.5.3 (plausible sequencing) and Hash Consistency (Property 2), the proof follows by deriving that the execution of `AMQHash_hash` on x in (d) must return ιs_0, simplifying the summation even further:

$$\sum_{\iota s_0} \sum_{s_1, hs_1} \left(\begin{array}{lll} (a) & p_{\text{hash}} & \times \\ (c) & \Pr\left[\texttt{AMQ\_addm}\ xs\ ((s \leftarrow_{\text{add}} \iota s_0), (hs \leftarrow_{\text{hash}} (x : \iota s_0))) = (s_1, hs_1)\right] & \times \\ (e) & \Pr\left[\texttt{AMQ\_query\_internal}\ s_1\ \iota s_0\right] & \end{array} \right)$$

Finally, as s_1 is a plausible outcome from `AMQ_addm` called on $s \leftarrow_{\text{add}} \iota s_0$, it is possible to then show, using Property 4 (query preservation), that querying for ιs_0 on s_1 must succeed. Therefore, the entire summation reduces to the summation of distributions over their support, which is trivially 1.

2.6 Overview of the Development and More Case Studies

The final result of this case study is the Ceramist mechanised framework, implemented as library in the Coq proof assistant [52]. It consists of three sub-parts, each handling a different aspect of constructing and reasoning about AMQs: (*i*) a library of *bounded-length data structures*, enhancing MathComp's [86] support for reasoning about finite sequences of varying lengths; (*ii*) a library of *probabilistic computations*, extending the infotheo probability theory library [4] with definitions of deeply embedded probabilistic computations and a collection of tactics and lemmas on summations described in Section 2.5; and (*iii*) the *AMQ interfaces and instances* representing the core of the framework described in Section 2.3.

Alongside these core components, the development also includes four specific case studies to provide concrete examples of how the library can be used for practical verification. The first two case studies are the mechanisation of the Bloom filter [17] and the Counting Bloom filter[125], as discussed earlier. In proving the false-positive rate for Bloom filters, the proof follows the proof by Bose *et al.* [19], also providing the first mechanised proof of the closed expression for Stirling numbers of the second kind. The third case study provides a mechanised verification of the quotient filter[14]. The final case study is a mechanisation of the Blocked AMQ—a family of AMQs with a common aggregation strategy. Finally, this abstract structure is instantiated with each of the prior AMQs, obtaining, among others, a mechanisation of Blocked Bloom filters [107]. The sizes of each library component, along with the

Section	Size (LOC)	
	Specifications	Proofs
Bounded containers	286	1051
Notation (§2.2.1)	77	0
Summations (§2.5)	742	2122
Hash operations (§2.3.1)	201	568
AMQ framework (§2.3.2)	594	695
Bloom filter (§2.2.2)	322	1088
Counting BF (§2.3.4, §2.4)	312	674
Quotient filter (§2.6.1)	197	633
Blocked AMQ (§2.6.2)	269	522

Table 2.1: Ceramist framework overview

references to the sections that describe them, are given in Table 2.1.

Of particular note, in effect due to the extensive proof reuse supported by Ceramist through its use of compositionality, the proof size for each of the case-studies *progressively decreases*, with around a 50% reduction in the size from the initial proofs of Bloom filters to the final Blocked AMQ case-studies.

2.6.1 Quotient Filter

A quotient filter [14] is a type of AMQ data structure optimised to be more cache-friendly than other typical AMQs. In contrast to the relatively simple internal vector-based states of the Bloom filters, a quotient filter works by internally maintaining a hash table to track its elements.

The operations of a quotient filter build upon the notion of *quotienting*, whereby a single p-bit hash outcome is split into two by treating the upper q-bits (the quotient) and the lower r-bits (the remainder) separately. Whenever an element is inserted or queried, the item is first hashed over a single hash function and then the output quotiented. The operations of the quotient filter then work by using the q-bit quotient to specify a bucket of the hash table, and the r-bit remainder as a proxy for the element, such that a query for an element will succeed if its remainder can be found in the corresponding bucket.

A false positive can occur if the outputs of the hash function happen to exactly collide for two particular values (collisions in just the quotient or remainder are not sufficient to produce an incorrect result). Therefore, it is then possible to reduce the event of a false positive in a quotient filter to the event that at least one in several draws from a uniform distribution produces a particular value. In the framework, quotient filters are encoded by instantiating the AMQHash interface from Subsection 2.3.1 with a *single* hash function, rather than a vector of hash functions, which is used by the Bloom filter variants (Section 2.1). The size of the output of this hashing operation is defined to be $2^q * 2^r$, and a corresponding quotienting operation is defined by taking the quotient and remainder from dividing the hash output by 2^q. With this encoding, the development is then able to provide a mechanised proof of the false positive rate for the quotient filter implemented using a p-bit hash function as being:

Theorem 2.6.1 (Quotient filter False Positive Rate)**.** *For a hash-function hs, after inserting a series of l unseen distinct values xs into an empty quotient filter qf, for any unseen $y \notin xs$, the probability of a query $y \in_? qf$ for y returning true is given by:*

$$\Pr\left[\texttt{qf\_addm}\ xs\ (hs, \texttt{qf\_new}) \rhd \texttt{qf\_query}\ y\right] = 1 - \left(1 - \frac{1}{2^p}\right)^l$$

2.6.2 Blocked AMQ

Blocked Bloom filters[107] are a cache-efficient variant of Bloom filters where a single instance of the structure is composed of a vector of m independent Bloom filters, using an additional "meta"-hash operation to distribute values between the elements. When querying for a particular element, the meta-hash operation would first be consulted to select a particular instance to delegate the query to.

While prior research has only focused on applying this blocking design to Bloom filters, a contribution of this case study was the observation that this strategy is in fact compositional and generic over the choice of AMQ, allowing the development to formalise an abstract Blocked AMQ structure, and later instantiate it for particular choices of "basic" AMQs. As such, this data structure highlights the scalability and proof reuse facilitated by Ceramist through the composition of programs and proofs.

The encoding of Blocked AMQs within Ceramist is done via means of two higher-order modules as in Figure 2.2: (*i*) a *multiplexed-hash* component, parameterised over an arbitrary hashing operation, and (*ii*) a *blocked-state* component, parameterised over some instantiation of the AMQ interface. The multiplexed hash captures the relation between the meta-hash and the hashing operations of the basic AMQ, randomly multiplexing hashes to particular hashing operations of the sub-components. The multiplexed-hash is constructed as a composition of the hashing operation H used by the AMQ in each of the m blocks, and a meta-hash function to distribute queries between the m blocks. The state of this structure is defined as pairing of m states of the hashing operation H, one for each of the m blocks of the AMQ, with the state of the meta-hash function. As such, hashing a value v with this operation produces a *pair* of type $(\mathbb{Z}_m, \texttt{Value})$, where the first element is obtained by hashing v over the meta-hash to select a particular block, and the second element is produced by hashing v over the hash operation H for this selected block. With this hashing operation, the state of the Blocked AMQ is defined as sequence of m states of the AMQ, one for each block. The insertion and query operations work on the output of the multiplexed hash, using the first element to select a particular element of the sequence, and then the second element as the value to be inserted or queried on the selected state.

Having instantiated the data structure as described above, the development proves the following novel generalised result about the false positive rate for blocked AMQs:

Theorem 2.6.2 (Blocked AMQ False Positive Rate). *For any AMQ A with a false positive rate after inserting l elements estimated as $f(l)$, for a multiplexed hash-function hs, after having inserted l distinct values xs, all unseen in hs, into an empty Blocked AMQ filter bf composed of m instances of A, for any unseen $y \notin xs$, the probability of a subsequent query $y \in_? bf$ for y returning* `true` *is given by:*

$$\Pr\left[\texttt{BA\_addm}\ xs\ (hs, \texttt{BA\_new}) \rhd \texttt{BA\_query}\ y\right] = \sum_{i=0}^{l} \binom{l}{i} (\frac{1}{m})^i (1 - \frac{1}{m})^{l-i} f(i)$$

The development instantiates this interface with each of the previously defined AMQ structures, obtaining the standard Blocked Bloom filters, the novel Counting Blocked Bloom filters and Blocked Quotient filter along with proofs of similar properties for them, entirely *for free*.

2.7 Related Work

Proofs about AMQs. While there has been a wealth of prior research into approximate membership query structures and their probabilistic bounds, the prevalence of paper-and-pencil proofs has meant that errors in analysis have gone unnoticed and propagated throughout the literature.

The most notable example is in Bloom's original paper [17], wherein dependencies between setting bits lead to an incorrect formulation of the bound (equation (17)), which has since been repeated in several papers [22, 39, 40, 89] and even textbooks [90]. While this error was identified by Bose *et al.* [19], their analysis was marred by an error in the definition of Stirling numbers of the second kind, resulting in another incorrect bound, corrected two years later by Christensen *et al.* [31], who avoided the error by eliding Stirling numbers, and deriving the bound directly. Furthermore, despite these corrections, many subsequent papers [35, 69, 81, 82, 107, 108, 125] still used Bloom's original incorrect bounds. For example, in Putze *et al.* [107]'s analysis of a Blocked Bloom filter, they derive an incorrect bound on the false positive rate by assuming that the subcomponents are described by Bloom's bound.

Mechanically Verified Probabilistic Algorithms. Past research has also investigated the verification of probabilistic algorithms, and this work builds on the results from several of these developments.

The ALEA library also provides mechanisms to simplify the task of proving properties of probabilistic algorithms [10]. In contrast to the deep embedding used in this case study for encoding probabilistic computations, ALEA uses a shallow embedding through a Giry monad [46], representing probabilistic programs as measures over their outcomes. As ALEA axiomatises a custom type to represent the subset of reals between 0 and 1 for capturing probabilities, they must independently prove any properties on reals required for their theorems, considerably increasing the proof effort.

The Foundational Cryptography Framework (FCF) [102] was developed for proving the security properties of cryptographic programs and provides an encoding for probabilistic algorithms. Rather than developing tooling for solving probabilistic obligations as is done in Ceramist, their library prioritises a proof strategy of proving the probabilistic properties of computations by reducing them to standard "difficult" programs with known distributions. To this end, their library uses an encoding of distributions as lists of possible values and an associated measure, and provides a Probabilistic Relational Hoare logic [11] (PRHL) over their monad to perform the reductions. While this strategy follows the structure of cryptographic proofs, their encoding increases the complexity of proving probabilistic properties.

Tassarotti *et al.*'s Polaris [126] library is a Coq framework for reasoning about probabilistic concurrent algorithms. Polaris uses the same reduction strategy for probabilistic specifications as the FCF library, inheriting some of the same issues with proving standalone bounds. Additionally, unlike this development, Polaris does not use a general monad for encoding probabilistic computations, and instead utilises a specialised monad for representing the interactions of concurrent probabilistic computations.

The author's previous effort on mechanising the probabilistic properties of blockchains also considered the encoding of probabilistic computations in Coq [51]. While that work also relied on infotheo's probability monad, it primarily considered the mechanisation of a restricted form of probabilistic properties (those with complete certainty), and did not deliver reusable tooling for this task.

Though this chapter primarily focuses on verification in the Coq proof assistant, there have been a number of lines of research into verifying probabilistic programs in other theorem provers, such as the HOL or Isabelle theorem prover, that are also relevant to the results of this work. Most such works adopt a measure-theoretic based encoding of probability, as seen with the ALEA library. This allows for reasoning about a larger class of programs, including those that manipulate distributions over infinite or even continuous domains, but imposes a higher verification burden than is required for verifying AMQs, which can be mechanised by reasoning solely about discrete and finite distributions.

The first of such works is Hurd's [65] framework for reasoning about probabilistic programs in the HOL theorem prover, developed in order to verify the Miller-Rabin primality test. His proof represents randomness in a program as an infinite stream of bits, with probabilistic programs producing distributions over such streams. Using this encoding, Hurd can thus reason about programs whose termination depends on probabilistic arguments, however the low-level encoding of randomness makes constructing and reasoning about higher-level primitives such as generating a random natural number more challenging. Subsequent work has built upon this framework to encode the probabilistic guarded command logic (pGCL) program logic and incorporated reasoning about non-determinism [66]. Eberl *et al.* [41] and Lochbihler [83] provide similar measure-theoretic frameworks in the Isabelle/HOL prover and use them to verify randomised binary trees and cryptographic protocols respectively.

Hölzl considered mechanised verification of probabilistic programs in Isabelle/HOL [64]. While Hölzl uses a similar composition of probability and computation monads to encode and evaluate probabilistic programs, his construction defines the semantics of programs as infinite Markov chains, represented as a co-inductive stream of probabilistic outputs. This design makes their encoding unsuitable for capturing terminating programs as done in this development, yet it is the only encoding that the author is aware of that is able to allow reasoning about the properties of non-terminating probabilistic programs.

While Ceramist is the first development, to the best of the author's knowledge, that provides a mechanised proof of the probabilistic properties of Bloom filters, it should be noted that some prior research has considered their deterministic properties. Blot *et al.* [18] provided a mechanised proof of the absence of false negatives for their implementation of a Bloom filter as part of their work on a library for using abstract sets to reason about the bit-manipulations in low-level programs.

Proofs of differential privacy. Another popular motivation to reason about probabilistic computations is for the purposes of demonstrating differential privacy. As a result, some prior research in this area has also considered the mechanised verification of probabilistic programs as this work does.

Barthe *et al.*'s CertiPriv framework [12] extends ALEA to support reasoning using a Probabilistic Relational Hoare logic, and uses this fragment to prove probabilistic non-interference arguments. However, CertiPriv again focuses on proving relational probabilistic properties of coupled computations rather than explicit numerical bounds. More recently, Strub *et al.* [124] have developed a newer mechanisation that supports a more general coupling between distributions, though it should be said that this extension still does not address the problem of directly proving numeric bounds.

2.8 Takeaways and Main Insights

This chapter has presented an overview of the first case study of this thesis, an investigation into the use of composition for verified software evolution through the construction of a general framework for verifying Approximate Membership Query structures. The case study began with the verification of Bloom filters, building up the tooling and definitions in Coq to verify their properties, before using composition to scale up this analysis to handle a larger class of probabilistic data structures, introducing a modular analysis strategy to allow *massive* proof reuse, and sometimes even obtain results for free.

The main contributions of this chapter are the development of a strategy for handling the scalable verification of randomised algorithms and an example of its application to produce the *first* verified implementations of a host of widely used AMQs: Bloom filter [17], Counting Bloom filter [125], the quotient filter [14], and a mechanisation of the novel Blocked AMQ—a generalisation of Blocked Bloom filters to abstract over the constituent elements, which was then instantiated to obtain a mechanisation of Blocked Bloom filters [107]. Finally, when proving the false-positive rate for Bloom filters, the implementation formalises the proof by Bose *et al.* [19], and in the process also provides the first mechanised proof of the closed expression for Stirling numbers of the second kind.

While this case study was effectively able to use compositional reasoning to reuse proofs over changes between the various AMQ data structures, the focus on such complicated data structures limits how representative its results can be of real verification projects. In particular, one of the main such limitations was the direct encoding of randomised algorithms, where the development encoded the semantics of randomised programs directly in terms of probability distributions. While the flexibility of this approach allowed for a greater diversity in proof strategies and thereby allowed effectively reasoning about the behaviours of AMQs, it came at the cost of always having to reason about programs from first principles. Furthermore, even though some of this proof burden was reduced using proof automation, this also lead to complex difficult to maintain proof tactics that took long times to run.[5] In practice, many real programs operate on simpler principles and do not require such bespoke arguments to certify — for the case of such verified software, can the task of maintenance be simplified even further?

[5]The full verification of the development takes around 1 hour to complete, and most of this time is spent in tactics.

CHAPTER

3

VERIFIED SOFTWARE MAINTENANCE THROUGH SYNTHESIS

This chapter presents the second case study of this work, an extension to the SuSLik separation-logic based program synthesiser to produce certified real-world executable C code, and through it, investigates the use of synthesis for verified software maintenance. The chapter starts by introducing certified program synthesis as an alternative to writing manual proofs for verified software maintenance, and focuses on the SuSLik program synthesiser as an example for the case study. The chapter then presents a brief background tutorial on separation logic and deductive synthesis, before introducing the technique of proof interpreters by Watanabe et al. *[130] that allows converting synthesis traces to deductive certificates and upon which this work builds. The chapter then covers the technical details of extending this technique of proof interpreters to produce executable certified C code and evaluates the effectiveness of this extension against SuSLik's benchmark suite. Finally, the chapter ends with a reflection on the main insights from this case study.*

As the reader may have noticed from the several thousands of lines of proofs carefully constructed in the previous chapter, the process of manually certifying a program can be quite a laborious task. In practice however, not all programs make use of such subtle and nuanced probabilistic interactions as seen in Bloom filters that would necessitate this level of complexity in their verification. In fact, many common real world programs rely on far simpler arguments for their correctness, so much so that researchers have found that a substantial number of programs can be automatically proven correct or in some cases the *entire implementation* itself can by synthesised from the specification alone.

As an example, consider now the task of writing a function to deallocate a linked list data structure, a fairly common and ubiquitous operation when writing in low-level programming languages such as C or Rust. The specification for the correctness of such a program might be expressed as follows:

$$\{\mathsf{sll}(\mathtt{x}, S)\}\ \texttt{list\_free(x)}\ \{\mathsf{emp}\} \tag{3.1}$$

The above formula uses classical Separation Logic (SL) [100, 112] to concisely encode constraints on memory locations and asserts that, if the function `list_free` is called on a pointer `x` in a state where `x` points to a singly-linked list with contents S (captured here by the SL predicate $\mathsf{sll}(\mathtt{x}, S)$), then after executing this function, the heap will be empty — *i.e.* this function will indeed deallocate the list at `x`. Passing this specification, along with suitable definitions for the sll predicate to the deductive SL-based synthesiser SuSLik [105] will allow it to automatically generate the program in Figure 3.1 by performing a proof search with the guarantee that the generated program satisfies the specification by construction.

```
void list_free(loc x) {
  if (x == 0) {}
  else {
    let nxt = *(x + 1);
    list_free(nxt);
    free(x);
  }
}
```

Figure 3.1: Automatically synthesised `list_free` implementation

This small experiment reveals a promising alternative approach to verified software maintenance, wherein certain selected components of a larger system may be fully automatically synthesised. In particular, for components in a verified system whose specifications themselves change frequently, designing and invoking bespoke composition arguments manually as was done previously would not be effective, as each change in the specification will inevitably require redoing the entire laborious process of certifying the implementation. Instead, one might envision a workflow in which both the implementation and the verification of these components may be fully outsourced to a program synthesiser like SuSLik, thereby effectively relieving the maintenance burden from the human developer.

There are unfortunately a couple of problems with enacting this plan for verified software maintenance. In particular, firstly, the programs generated by such synthesisers are often written in bespoke domain specific languages tailored for synthesis and not directly executable — for example, the programs generated by SuSLik are by default written in a toy language, SusLang, tailored to capture the tool's encoding of Separation Logic. Secondly, even if these generated programs are transpiled into some executable format, they fail to constitute a *certified* program, as their correctness now depends on both the correctness of the transpilation, which could easily introduce discrepencies in the semantics of the generated programs, and also the correctness of the synthesiser itself, which, for non-trivial program synthesisers such as SuSLik, are typically large and complex codebases in their own right.

The goals of this chapter are to investigate the challenges in constructing real-world executable and certified code from such deductive program synthesisers and through this process demonstrate the efficacy of program synthesis for the purposes of verified software maintenance. To do this, this case study will build upon the technique of proof interpreters introduced by Watanabe *et al.* [130] that provides a unifying framework for translating synthesis trees to verification proofs and extend it to produce executable C code that is verified to be correct according to the initial synthesis specification embedded in the Verified Software Toolchain framework [7] for foundational verification of C code. The main technical contributions of this case study are therefore the design and implementation of an extension to the SuSLik program synthesiser to produce executable and certified real-world C code.

In the remainder of this chapter, the narrative will discuss the technical details of this case-study, starting with a gentle introduction to Separation Logic and how SuSLik performs deductive synthesis, before motivating the high-level intuition behind proof interpreters. The chapter then discusses the key challenges of generating C code from SuSLik and explains how each one was handled, before presenting an overall evaluation of the extension itself by using it to synthesise executable versions of each one of the benchmarks from SuSLik's test suite. The technical content of this chapter is a revision of work that has been published before at the ICFP conference series and can be found here [130].

3.1 A Tutorial on Proof Interpreters

In order to contextualise the results of this chapter, this section provides a brief introduction to technique of *proof interpreters* developed by Watanabe *et al.* [130] to construct proof certificates from the output of a deductive synthesiser such as SuSLik. The section starts with a review of the Synthetic Separation Logic variant [68] used by SuSLik to reason about programs, then motivates how SuSLik uses this formalism to deductively search for programs satisfying a specification by constructing synthesis proof trees. Finally the section presents the key ideas of how proof interpreters use the mechanisms of such derivation trees in SuSLik to map synthesis trees to verification proofs. The reader is directed to Watanabe *et al.*'s paper [130] for more details; a full coverage is outside the scope of this thesis.

3.1.1 Synthetic Separation Logic Primer

Synthetic Separation Logic (SSL) [68] is the extension of classical Separation Logic [112] that underlies the deductive synthesis algorithm implemented by the latest versions of the SuSLik tool [105].

Assertions in (Synthetic) Separation Logic capture constraints over disjoint partitions of the program's heap, called *heaplets*. These constraints can be combined using the *separating conjunction* $*$: the statement $H * H'$ asserts that the heap can be partitioned into two *disjoint* sub-heaps such that H holds on the first, and H' on the second. This document will use the standard notation $a \mapsto e$ to describe a heaplet represented by a memory location with address a and contents e (pronounced "a *points to* e"); emp is an SL assertion satisfied by an empty heap. Using these assertions, one can capture the semantics of an imperative program using a Hoare triple $\{P\}\ c\ \{Q\}$ (*cf.* (3.1)), which asserts the following: if in any state satisfying P, after executing the program c, the resulting heap state must satisfy Q.

Next, in order to reason about non-trivial pointer-based data structures on the heap, separation logics allow users to define new inductive predicates to capture the memory layouts of such entities. For example, in SSL, a singly-linked list might be defined by the following inductive predicate:

$$\begin{aligned} \mathsf{sll}^{\alpha}(x, s) \ &\triangleq\ x = 0 \wedge \{s = \emptyset; \mathsf{emp}\} \\ &\mid\ x \neq 0 \wedge \{s = \{v\} \cup s_1 \wedge \beta < \alpha; [x, 2] * x \mapsto v * (x+1) \mapsto nxt * \mathsf{sll}^{\beta}(nxt, s_1)\} \end{aligned} \tag{3.2}$$

The two clauses in definition (3.2) correspond to the cases of an empty and non-empty list. In the first case, the "head" pointer of the list is null, and the heap allocated for the structure, as well as its payload, represented by the set s, are empty. In the latter case, the head pointer of the list x is non-empty, and the heap structure of the list is represented by two subsequent pointers, starting from x and storing a payload element v and the pointer nxt to the tail, which has the same structure, captured by the recursive occurrence of the same predicate sll. One non-standard aspect of this encoding is the use of mathematical *sets* to represent the linked list's payload instead of more traditional *algebraic lists*, which are an artefact arising from the SuSLik implementation for more streamlined integration with third-party SMT solvers. The other unusual part of this predicate definition are the *cardinality variables* (α, β), as well as the constraints on them ($\beta < \alpha$), which are necessary to reason about termination of synthesised recursive programs and their auxiliary procedures via the mechanism of *cyclic proofs* [118].[1] For the purpose of this work, one can think of cardinalities in inductive predicates as integer variables

[1]The exact usage of cyclic proofs for synthesis of provably terminating recursive programs is orthogonal to this work, and the reader is referred to the paper by Itzhaky et al. [68] for the details.

capturing the fact that the size of a heap, constrained by a recursive occurrence, is strictly smaller than that of the enclosing data structure, as in, *e.g.*, $\beta < \alpha$ in the predicate definition (3.2).

3.1.2 Deductive Synthesis in SSL

Given a specification, SusLik will generate a program in SusLang, a simple C-like language with pointers, function calls, and recursion, whose syntax is given in Figure 3.2. SusLang values include booleans and integers, and a special type `loc` for pointer variables. Pointers are equivalent to unsigned integers with a designated pointer constant `0` for null. Expressions include variables, literal constants, equality checks and logical connectives. The language allows pointer arithmetic in the form $x + \iota$.

$$
\begin{array}{lrcl}
\text{Variable} & x, y & & \text{Alpha-numeric identifiers} \\
\text{Size, offset} & n, \iota & & \text{Non-negative integers} \\
\text{Expression} & e & ::= & 0 \mid \mathsf{True} \mid x \mid e = e \mid e \wedge e \mid \neg e \mid d \\
\mathcal{T}\text{-expr.} & d & ::= & n \mid x \mid d + d \mid n \cdot d \mid \{\} \mid \{d\} \mid \cdots \\
\text{Command} & c & ::= & \texttt{let}\ x = *(x + \iota) \mid *(x + \iota) = e \mid \\
& & & \texttt{let}\ x = \texttt{malloc}(n) \mid \texttt{free}(x) \mid \texttt{err} \mid \\
& & & f(\overline{e_i}) \mid c; c \mid \texttt{if}\ (e)\ \{c\}\ \texttt{else}\ \{c\}
\end{array}
$$

Figure 3.2: SusLang syntax.

To synthesise an implementation of `list_free` in SusLang, the specification (3.1) is first transformed to a synthesis goal of the form $\Gamma; \mathcal{P} \rightsquigarrow \mathcal{Q} | c$, where Γ is the set of currently available program-level and logical variables (in the running example, it was initially just $\{\mathrm{x}, S\}$); $\{\mathcal{P}\}$ and $\{\mathcal{Q}\}$ are the corresponding ascribed pre- and postconditions; and c is an unknown program, yet to be synthesised. In general, both the pre- and postcondition of specifications and goals can feature a *pure* and spatial part, *e.g.*, $\{\mathcal{P}\} = \{\phi, P\}$. The pure part ϕ captures the logical constraints on variables and values involved in the specification. The spatial part P describes the heap shape using standard SL assertions, joined by the separating conjunction connective ($*$): emp for an empty heap, $(x + \iota) \mapsto e$ for an individual address x storing a value e at a (possibly zero) offset ι, a *block assertion* $[x, n]$ for a continuous segment of n elements starting at x, which can be deallocated, and $\mathsf{p}^{\alpha}(\overline{t_i})$ for a heap of size α (omitted when unambiguous from context) described by an occurrence of a predicate p with arguments $\overline{t_i}$.

The synthesis itself proceeds by iteratively applying one of the rules of SSL, building a derivation for the initial goal. Such a derivation will also contain the desired program c as its byproduct which can then be extracted. Figure 3.3 presents a few selected rules of SusLik. Within rules, lowercase Latin letters x, y are used for program variables (taken from the set $\mathsf{ProgVars}$), e, t for program-level terms

$$\textsc{Emp}\ \frac{\vdash \phi \Rightarrow \psi}{\Gamma;\ \{\phi; \mathsf{emp}\} \leadsto \{\psi; \mathsf{emp}\} \,|\, \texttt{skip}} \qquad \textsc{Frame}\ \frac{\{\phi; P\} \leadsto \{\psi; Q\} \,|\, c}{\{\phi; P * R\} \leadsto \{\psi; Q * R\} \,|\, c}$$

$$\textsc{Read}\ \frac{\begin{array}{c}\forall y.\Gamma;\ \{\phi \wedge y = e; (x + \iota) \mapsto e * P\} \leadsto \mathcal{Q} \,|\, c \\ x \in \mathsf{ProgVars} \quad y \in \mathsf{ProgVars} \setminus \mathsf{Vars}(\Gamma)\end{array}}{\Gamma;\ \{\phi; (x + \iota) \mapsto e * P\} \leadsto \mathcal{Q} \,|\, \texttt{let}\ y = *(x + \iota); c}$$

$$\textsc{Write}\ \frac{\begin{array}{c}\Gamma;\ \{\phi; (x + \iota) \mapsto e * P\} \leadsto \{\psi; (x + \iota) \mapsto e * Q\} \,|\, c \\ \mathsf{Vars}(e) \subseteq \mathsf{ProgVars}\end{array}}{\begin{array}{c}\Gamma;\ \{\phi; (x + \iota) \mapsto e' * P\} \leadsto \\ \{\psi; (x + \iota) \mapsto e * Q\} \,|\, *(x + \iota) = e; c\end{array}}$$

$$\textsc{Free}\ \frac{\Gamma;\ \{\phi; P\} \leadsto \{\psi; Q\} \,|\, c \quad x \in \mathsf{ProgVars}}{\Gamma;\ \{\phi; [x, n] * \big((x + i) \mapsto e_i\big)_{0 \le i < n} * P\} \leadsto \{\psi; Q\} \,\big|\, \texttt{free}(x); c}$$

$$\textsc{Call}\ \frac{\begin{array}{c}\forall \overline{x_i}, \overline{\nu_j}.\exists \overline{\omega_k}\,;\ \{\phi'; P\} \leadsto \{\psi'; S\} \,\big|\, f(\overline{x_i}) \qquad \Gamma \mathbin{\dot\cup} \forall \sigma(\overline{\omega_i});\ \{[\sigma]\psi' \wedge \phi; [\sigma]S * R\} \leadsto \mathcal{Q} \,\big|\, c \\ \vdash \phi \Rightarrow [\sigma]\phi' \qquad \mathsf{dom}\,(\sigma) = \{\overline{x_i}, \overline{\nu_j}, \overline{\omega_k}\} \qquad \sigma(x_i) \in e[\Gamma] \qquad \sigma(\nu_j) \in \kappa[\Gamma]\end{array}}{\Gamma;\ \{\phi; [\sigma]P * R\} \leadsto \mathcal{Q} \,|\, f(\sigma(\overline{x_i})); c}$$

$$\textsc{Open}\ \frac{\begin{array}{c}\Gamma \mathbin{\dot\cup} \forall \overline{\omega_{jk}}\,;\ [\overline{t_i/\nu_i}]\{\phi \wedge e_j \wedge \chi_j\ ;\ R_j * P\} \leadsto \mathcal{Q} \,\Big|\, c_j\ \textit{for}\ \underline{\textit{all}}\ j{=}1..r \\ \mathsf{p}^{\alpha}(\overline{\nu_i}) : \overline{\langle e_j, \{\, \overline{\omega_{jk}}, \chi_j\}\rangle\, R_j}_{j=1..r}\ \text{s.t.}\ \omega_{jk} \notin \mathsf{Vars}(\Gamma), \mathsf{GV}(t_i) = \varnothing\end{array}}{\Gamma;\ \{\phi; \mathsf{p}^{\alpha}(\overline{t_i}) * P\} \leadsto \mathcal{Q} \,\Big|\ \begin{array}{l}\texttt{if}\ \ ([\overline{t_i/\nu_i}]e_1)\ \{c_1\} \\ \texttt{else if}\ \ ([\overline{t_i/\nu_i}]e_2)\ \{c_2\}\ \texttt{else}\ \cdots\end{array}}$$

$$\textsc{Close}\ \frac{\begin{array}{c}\Gamma \mathbin{\dot\cup} \exists \overline{\omega_{jk}}\,;\ \mathcal{P} \leadsto [\overline{t_i/\nu_i}]\{\phi \wedge e_j \wedge \chi_j\ ;\ R_j * Q\} \,\Big|\, c_j\ \textit{for}\ \underline{\textit{some}}\ j{\in}1..r \\ \text{Predicate}\ \mathsf{p}^{\alpha}(\overline{\nu_i}) : \overline{\langle e_j, \{\overline{\omega_{jk}}, \chi_j\}\rangle\, R_j}_{j=1..r}\ \text{s.t.}\ \omega_{jk} \notin \mathsf{Vars}(\Gamma)\end{array}}{\Gamma;\ \mathcal{P} \leadsto \{\phi; \mathsf{p}^{\alpha}(\overline{t_i}) * Q\} \,\big|\, c}$$

Figure 3.3: Selected declarative rules of SSL (adapted from [130]).

(of the syntactic class e in Figure 3.2), Greek letters ν, ω for logical variables, and ϕ, ψ, χ for logical formulas. Assertions are interpreted in an environment Γ in which some of the variables are universally quantified and others existentially quantified, with a prefix of the form $\forall\overline{x}.\exists\overline{y}$. Program variables are *always* included in the universal prefix. Logical variables are split between universal (also called *ghost* variables and denoted $\mathsf{GV}(\Gamma)$) and existential ($\mathsf{Existentials}(\Gamma)$). All quantified variables are denoted as $\mathsf{Vars}(\Gamma) = \{\overline{x}, \overline{y}\}$. The $\dot\cup$ operator, used in some of the rules, joins two environments, so in the resulting environment the quantifiers follow the same $\forall.\exists$ quantifier pattern. The notation uses $e[\Gamma]$ for the set of all possible expressions that can be constructed using program variables in Γ, and $\kappa[\Gamma]$—to denote all logical terms that can be constructed with any variables from Γ. Finally, the notation $[\sigma]P$ is used to denote the application of a substitution σ to all variables in an expression P.

Most of the rules in Figure 3.3 (Read, Write, Alloc, Open, and Call) are *operational*: when read bottom-up, they advance the synthesis by emitting parts of the program and reducing its goal to their premises. Perhaps the most interesting of those is the Call rule, which synthesises procedure calls. The rule combines SL-style framing, with R as the frame, and substitution of actual into formal parameters

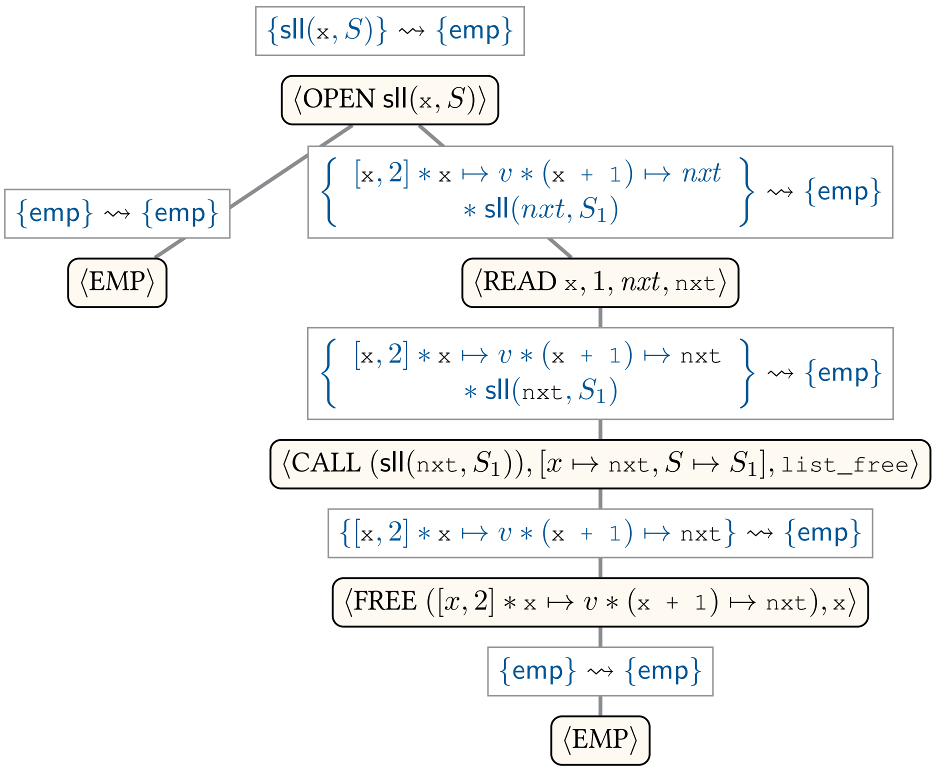

$\{\mathsf{sll}(\mathtt{x}, S)\} \leadsto \{\mathsf{emp}\}$
$\langle \text{OPEN } \mathsf{sll}(\mathtt{x}, S)\rangle$
$\{\mathsf{emp}\} \leadsto \{\mathsf{emp}\}$
$\langle \text{EMP} \rangle$
$\left\{ \begin{array}{c} [\mathtt{x}, 2] * \mathtt{x} \mapsto v * (\mathtt{x} + 1) \mapsto nxt \\ * \, \mathsf{sll}(nxt, S_1) \end{array} \right\} \leadsto \{\mathsf{emp}\}$
$\langle \text{READ } \mathtt{x}, 1, nxt, \mathtt{nxt} \rangle$
$\left\{ \begin{array}{c} [\mathtt{x}, 2] * \mathtt{x} \mapsto v * (\mathtt{x} + 1) \mapsto \mathtt{nxt} \\ * \, \mathsf{sll}(\mathtt{nxt}, S_1) \end{array} \right\} \leadsto \{\mathsf{emp}\}$
$\langle \text{CALL } (\mathsf{sll}(\mathtt{nxt}, S_1)), [x \mapsto \mathtt{nxt}, S \mapsto S_1], \mathtt{list\_free} \rangle$
$\{[\mathtt{x}, 2] * \mathtt{x} \mapsto v * (\mathtt{x} + 1) \mapsto \mathtt{nxt}\} \leadsto \{\mathsf{emp}\}$
$\langle \text{FREE } ([x, 2] * \mathtt{x} \mapsto v * (\mathtt{x} + 1) \mapsto \mathtt{nxt}), \mathtt{x} \rangle$
$\{\mathsf{emp}\} \leadsto \{\mathsf{emp}\}$
$\langle \text{EMP} \rangle$

Figure 3.4: Derivation tree constructed to synthesise `list_free`

via σ, which is also applied to the procedure f's postcondition. Existential variables in the procedure's environment are renamed to fresh ghost variables in the second premise of the rule. Formal parameters x_i of f are mapped to program expressions $e[\Gamma]$ using program variables of Γ, and ghosts ν_j are mapped to logical terms κ using any variables of Γ. The rule Emp is a terminal one, and, when applied, corresponds to a successful synthesis of a program branch: empty heaps in both pre/postconditions mean that there is nothing more for a program to do, assuming the constraints $\phi \Rightarrow \psi$ accumulated in pure parts hold. Finally, the rules Frame and Close are structural ones: they do not emit a part of the program but rather change the shape of the goal, possibly making other rules applicable. For instance, Frame removes similar parts of the symbolic heap from the pre/postcondition, eventually enabling Emp, while Close unfolds a predicate instance in the goal's postcondition, replacing its occurrence $\mathsf{p}^{\alpha}(\overline{t_i})$ by p's j^{th} clause (for some j), thus revealing more information about the structure of the final heap.

To see these rules in action, consider again the synthesis of `list_free`, as referenced in the introduction. Figure 3.4 depicts a simplified derivation tree as constructed by SuSLik when synthesising this function. Initially, the synthesis starts with the goal $\{\mathsf{sll}(\mathtt{x}, S)\} \leadsto \{\mathsf{emp}\} \,|\, c$, that represents the entire specification for the function, as provided by the user. By searching through the space of possible rules to be applied, the synthesiser discovers that for this goal the OPEN rule can be applied to "unfold" the definition of the

$\mathsf{sll}(\mathtt{x}, S)$ predicate in the precondition. This introduces the if condition on `x == 0` seen in the generated program, and produces two subgoals to synthesise the bodies of the branch, one for each case of the inductive predicate. In the first case, where the linked list is empty, the goal is simply $\{\mathsf{emp}\} \rightsquigarrow \{\mathsf{emp}\} \mid c$, indiicating that there *is no more list to free*, and so the the synthesiser is able to easily complete this branch using the EMP rule. In the second case, the list is non-null and the precondition is expanded to reveal the heaplets constituting the head: $\{[\mathtt{x}, 2] * \mathtt{x} \mapsto v * (\mathtt{x} + 1) \mapsto nxt * \mathsf{sll}(nxt, S_1)\} \rightsquigarrow \{\mathsf{emp}\} \mid c$. From here, the synthesiser non-deterministically chooses a READ rule to move the nxt variable from the logical to the program context, and invokes the CALL rule with `list_free` recursively on the tail of the list, $\mathsf{sll}(nxt, S_1)$, to free it, leaving the residual goal as $\{[\mathtt{x}, 2] * \mathtt{x} \mapsto v * (\mathtt{x} + 1) \mapsto nxt\} \rightsquigarrow \{\mathsf{emp}\} \mid c$. At this point, the precondition consists purely of the memory block at `x`, and so the entire synthesis can then be completed by simply applying the FREE rule to `x` and dispatching the trivial remaining obligation with EMP. Finally, SuSLik simply traverses over this derivation tree, mapping each rule to a program statement and thereby produces the correct-by-construction implementation of `list_free`.

3.1.3 From Synthesis to Verification

Can the programs produced by SuSLik really be trusted? Automated program synthesisers are complex, and it is naturally quite hard to guarantee that they are free of bugs. Moreover, a realistic verification project will almost certainly have portions that are beyond the capabilities of any synthesiser, and hence there is a need to make sure that synthesis results will integrate well with parts of the system that are implemented and verified manually. A promising approach to addressing both of these concerns is to make the synthesiser produce *certificates*, allowing for independent checking of its results *w.r.t.* user-ascribed specifications in foundational verification tools, which can be embedded into proof assistants such as Coq and thereby provide the highest assurance guarantees with a minimal trusted code base.

In theory, it should be easy to generate certificates for deductive synthesisers: after all, they synthesise programs together with their "proofs"! In practice, however, this is far from straightforward, because synthesis proofs are fundamentally structured very differently from the proofs used in foundational verifiers. SL verifiers typically work by using symbolic execution to propagate the *symbolic state* from the precondition forward through the program, and only at the end do they check that the final symbolic state entails a given postcondition. On the other hand, a deductive synthesiser is forced to use information from both the pre- and the post-condition, manipulating both in the process, as it cannot rely on the program to guide the search. Aside from this discrepancy, there are many low-level differences in the structure of the proofs between synthesis and verification, as well as between different embeddings, which make generating certificates for a deductive synthesiser a non-trivial task.

To address these issues, Watanabe *et al.* introduced the novel technique of *modular proof interpreters* [130] for certifying the results of deductive synthesis algorithms against custom verification backends. The technique itself is inspired by continuation-passing programming style, and allows proof engineers to write proof translation logic for different verification frameworks in a uniform and modular way.

The core of Watanabe *et al.*'s proof interpreters technique revolves around the following key insights:

- **Deferred Proof Steps** - The naive approach to translating a *source proof* generated by the synthesiser into a *target proof* understood by the verifier would involve mapping each individual application of a source inference rule in to a sequence of applications of target inference rules. This, however, does not work for this application due to the aforementioned non-local differences in the rule application order employed by the synthesiser versus other verifiers.

  The first key idea is to equip a proof translator with a way to *define the order* in which the translated results of synthesis rule applications are executed in the script of the back-end verifier. To do this, Watanabe *et al.* introduce the notion of deferred proof steps, wherein the proof translator will perform the translation from applications of inference rules from one backend to the other, but also simultaneously collect deferred target proof steps as it traverses the source proof tree. Then, when the translator reaches the leaves of any generated target proof tree, it will retroactively apply these deferred steps at the end, bridging differences in application order.

- **Backend Proof Context** - As it turns out, simply deferring certain target proof steps is unfortunately not sufficient: any changes in the target *proof context* since the moment at which certain deferred steps were scheduled, can render those steps invalid at the point of their application.

  The second key idea in Watanabe *et al.*'s work is in allowing their proof translators to define the logic for maintaining a backend-specific proof context, and also to *pass it as an argument* to the deferred proof steps at the time of their applications, making their treatment similar to that of composed continuations in interpreters for programs written in continuation-passing style.

Composing both these insights together, Watanabe *et al.* define a generic and general framework for implementing a certification backend for a deductive synthesiser — in order to translate a synthesis derivation tree into a certificate, the user must provide first, a mapping from synthesis proof steps to a tuple of, firstly, a corresponding program statement in the target language and secondly a sequence of one or more immediate and deferred proof steps for the verification backend. Once again, the reader is

```
void list_free(loc x) {
  if (x == NULL) { return; }
  else {
    loc nxt = READ_LOC(x, 1);
    list_free(nxt);
    free(x);
    return;
  }
}
```

```
Definition list_free_spec :=
  DECLARE _list_free
   WITH x: val, s: (list Z), a: sll_card
   PRE [ (tptr ssl_val) ]
   PROP(is_pointer_or_null(x))
   PARAMS(x)
   SEP ((sll x s a))
   POST[ tvoid ]
   PROP( )
   LOCAL( )
   SEP ().
```

Figure 3.5: Definition of `list_free` in C (left) and corresponding specification in VST (right)

referred to the corresponding paper for more details as a full coverage is outside the scope of this thesis.

3.2 Certifying the Synthesis of C Programs

Returning once more to the overarching aims of this chapter – that is, to automatically synthesise certified executable programs from their specifications – this section presents the main results of this chapter which are to build upon Watanabe *et al.*'s technique of modular proof interpreters and use it to extend SuSLik to produce real-world executable and certified C code as a synthesis output.

In particular, Watanabe *et al.*'s initial case studies looked primarily at verifying SusLang programs according to various "C-like" DSLs within Coq, not actual executable code. The work presented in this chapter builds upon Watanabe *et al.*'s framework and asks whether it be used to produce proofs for the real deal—*executable* C? In this case study, this question is answered in the affirmative, implementing a certification backend for the Verified Software Toolchain [8], using it to translate SusLang programs into C and certify their correctness with regards to a simplified semantics of C.[2] In the rest of this section, the narrative will provide an overview of this extension, focusing on the additional changes that had to be made to make SuSLik and SusLang conform to the constraints of real executable code.

3.2.1 SusLang on Metal: Converting Programs to C and Specifications to VST

Before setting about certifying programs, it is necessary to first translate SusLang functions into C and their specifications to VST. This translation has some subtleties. For instance, Figure 3.5 lists the translated program and specification for the running example `list_free`, which follows closely

[2]Due to limitations of SuSLik's memory model, the implementation makes the simplifying assumption that `malloc` never returns `NULL`.

from the original definitions, but make use of some custom data types (such as `loc`) and operations to manipulate memory (`READ_LOC, free`). As it turns out, these small modifications are actually crucial for faithfully realising the *simplified* memory model assumed by SusLang programs on real hardware.

In particular, a fact that has been glossed over in the prior sections has been the way in which SusLang arrays can freely contain pointer and integer values, side by side, without issue. More generally, by allowing these kinds of constructs, SusLang implicitly makes the assumption that integer and pointer values occupy exactly the same amount of space on the heap, and while, in terms of C code, this might not at all be an uncommon assumption, it is an assumption nonetheless, carelessly introducing unsafe and potentially unexpected implementation-specific behaviour into the generated program.

```
typedef union
    sslval {
  int ssl_int;
  void *ssl_ptr;
} *loc;
```

The solution then is quite natural: this assumption must simply be encoded within the C type system. In the translation, it is enforced that all allocations, reads and writes to and from the heap will be done exclusively to terms of a custom type `sslval`, defined (left) as a union of integer and pointer values. Wrapping this up in a type alias, `typedef union sslval *loc`, and pairing it with corresponding read and write macros that transparently handle the coercion between types (*i.e.*, defining the operation to read locations as a compile-time macro `#define READ_LOC(x,y) (*(x+y)).ssl_ptr`), the generated programs thereby both *syntactically look* and *semantically behave* exactly as the SusLang programs they represent, simplifying the subsequent translation of VST specifications and proofs.

3.2.2 Getting Real: Impedance Mismatching Between SusLang and C Semantics

Having translated SusLang programs to C, the real question is whether these programs can actually be verified as correct using information from SuSLik's synthesis trees. Thankfully, as VST uses the same standard forward-execution style of reasoning as all the other frameworks considered by Watanabe *et al.*, much of the form of these proofs still end up following the same broad strokes, requiring changes only to account for discrepancies in their semantic models. The rest of this section will highlight the most significant areas in which the two semantic models diverged and thus posed issues for the translation.

Ternary Expressions

Consider the following program statement: `*m = (x < y ? 3 : 1);` In programming languages with an absence of uncontrolled side-effects such as SusLang, it would always be safe to treat the evaluation of the entire statement as a single step, *i.e.*, a program operation writing the value of the expression

$(x < y \mathbin{?} 3 : 1)$ directly into the memory at location m. In fact, SSL proofs take this even further, treating ternary expressions as single values within logical specifications, and using them as such within spatial assertions, as in $m \mapsto (x < y \mathbin{?} 3 : 1)$). Switching back to the semantics of C, where expressions can have arbitrary effects, such treatment of ternaries is clearly no longer valid. This is reflected in VST, where the evaluation of a ternary sub-expression is interpreted as an if statement, branching the proof into two separate control flow paths for each case, in contrast to direct execution in SSL proofs. To thereby keep the SSL and VST proof contexts synchronised and avoid divergence, the translation adds additional logic to the proof interpreter to transparently handle such statements. Whenever an ternary expression is evaluated during a SSL proof, the generated VST proof branches on each cases of the ternary but also provides a unifying post-condition to the branch that then joins both cases together immediately afterwards using the fact that the result of the expression is equivalent to a logical ternary expression, using tactics provided by VST to automatically dispatch the generated obligations.

Splitting and Recombining Memory Blocks

Another translation aspect that required special care was in managing the particularly loose treatment of memory in SSL. Recall that a contiguous block of allocated memory in SSL is represented by the spatial assertion $[x, n]$, with the contents of this block captured separately as $(x + i) \mapsto -$ for each element. This encoding of allocations as separate blocks and mappings allows SSL proofs to easily mix between single cells in memory $x \mapsto -$ and individual elements of larger blocks $[x, n] * x \mapsto -$, so that synthesised programs can pass pointers *from the middle of blocks of memory* freely to procedures that expect lone pointers. While mixing these kinds of pointers is valid according to the semantics of C, and this work ran into difficulties when certifying such programs in VST, where blocks of memory and their contents are encoded as a single assertion and can not easily be split. Having experimented with a number of non-trivial logic memory transformations available in VST (*e.g.*, logically splitting memory blocks into individual segments and then recombining them back), this translation opts for a simpler and more principled solution: constraining SuSLik's proof search to reject programs that *mix pointers* from *different-size blocks*, allowing pointers to unify only if their associated blocks are of the same size. This restriction did not prevent any known SuSLik benchmarks from being synthesised.

Integer Semantics and Overflows

As the only backend in Watanabe *et al.*'s framework beholden to the constraints of real-world hardware, VST is uniquely challenged amongst the other instantiations in that it must necessarily reason about overflow semantics. This poses a fundamental problem when translating integer-manipulating programs from SusLang, as SuSLik proofs do not consider overflow, and synthesised programs may sometimes perform unsafe (bounded-)arithmetic operations. In particular, any numbers that arise during the execution of programs, intermediate or otherwise, in VST must always be *guaranteed within the valid ranges* of integers before the proof may continue. This constraint of tracking overflow bounds causes issues even when verifying programs with numeric variables that only rely on comparisons, *i.e.*, having no chance of overflow—as the overflow bounds on these numeric variables must first be established before one can reason about the behaviour of any comparisons over them. While the differences in the overflow semantics between SusLang and C are too large to handle in the general case, by adjusting the synthesised program specs to include assumptions on numeric bounds, and using VST's native tactics to dispatch overflow obligations, the case study were able to certify programs that only rely on arithmetic comparisons, *e.g.*, finding the maximum or the minimum of a list of integers.

3.3 Evaluation

The entire extension for translating SSL proof trees to certified executable C programs has been implemented as an instantiation of Watanabe *et al.*'s framework for SuSLik, written in a combination of Scala (for the individual VST interpreter) and Coq (for VST specific automation and written primarily in Ltac). The table on the right summarises the overall implementation effort of the extension in terms of lines of code, and in comparison to Watanabe *et al.*'s core proof interpreter. The implementation and benchmarks are open source and are publicly available [129]. The Coq automation library for the backend can also be installed via the `opam` package manager.

Component	Scala	Coq
Proof evaluator	1042	-
VST support	1887	166
The rest of SuSLik	5508	-

The translation described in this section works for unaltered SusLang programs, and the synthesis algorithm has been slightly restricted to suit particular patterns in the certification backend (*cf.* Subsection 3.2.2). Bearing this in mind, the aim with the evaluation was to answer the following questions:

1. How efficient is the certification: what are the sizes of the generated Coq specs and proofs, and how long does it take to check them via the corresponding SL embeddings?
2. What design choices in SusLang/SSL and the languages and logic of the VST verification backend might pose obstacles to automated certification of synthesised heap-manipulating programs?

Group	Description	Synthesis Time	VST		
			Spec	Proofs	Time (s)
Integers	max	<0.1	21	20	6.4
	min	<0.1	21	20	79.2
	swap2	<0.1	20	14	132.7
	swap4	<0.1	20	22	649.6
Singly-Linked Lists	length	0.6	-	-	-
	maximum	0.5	21	57	244.8
	minimum	0.5	21	57	242
	append	0.2	23	52	312.9
	copy	0.4	33	63	370.1
	two-element	0.3	34	36	171.5
	dispose	<0.1	31	28	7.8
	singleton	<0.1	34	26	127.4
DLLs	append	2.3	24	97	594.6
	singleton	<0.1	34	27	128.3
Trees	copy	1.3	32	77	516.5
	flatten	0.2	58	76	685.7
	dispose	<0.1	31	32	10.8
	size	0.5	-	-	-

Table 3.1: Statistics for synthesised programs from SuSLik's benchmark suite. Sizes of generated Coq artefacts are in lines of code. Last column reports the checking times for the generated proof scripts.

Table 3.1 summarises the evaluation results on programs manipulating with individual pointers and integers, singly- and doubly-linked lists, and binary trees, taken from SuSLik's benchmark suite. The reported sizes of Coq artefacts do not include translated heap predicates and their inversion lemmas as those are shared between specs of multiple programs. All runtimes are obtained on a 1.90GHz Intel Core i7-8665U machine with 40GB RAM running Ubuntu 18.04 and Coq 8.11.2.

With regard to Question (1), Table 3.1 demonstrates that all generated proofs are relatively concise, typically ranging from around 20 to 100 lines, which are more than in line with the corresponding proofs that a human might have hand-written instead. Secondly, in terms of the checking times for the generated proof scripts, the VST proofs do take considerable time to verify, with a substantial number of scripts taking in the order of minutes to be checked by Coq. The significantly longer checking times for these proofs are primarily due to the generality of VST's `entailer!` tactic [23], which is frequently used in the proof to dispatch heap entailments. This tactic internally performs a sophisticated proof search and has complexity that grows exponentially with the number of variables and theorems in the proof context, expecting that human written proofs will take care to keep the number of these low. Unfortunately, this is not typically the case for synthesised proofs, as SuSLik's proof search mechanism may end up generating scripts with unused variables and lemmas in the proof. In the future, the performance of the proof scripts might be improved by optimising the proof scripts to follow structures and patterns closer to human-written ones (which is not currently the case for the auto-generated ones).

As for Question (2), Table 3.1 indicates a couple benchmarks that failed to verify in the VST backend. In particular, the two case studies that could not be handled in the VST backend were the programs to calculate list length and tree size. This arose because of the discrepancy between SuSLik's logic and VST in their handling of integer overflows. As SuSLik's logical formalism doesn't reason about integer overflows at all, the generated programs for list length and tree size can not be easily translated to any correct implementation in real-world C, where integers may overflow. As a result, the backend is fundamentally unable to produce a proof script that will be accepted by the Coq proof assistant. This shortcoming can be addressed by modifying SusLang and its synthesis rules to account for backend specific constraints, making the synthesiser language closer to an intermediate representation, serving multiple backends. This is an interesting potential extension, and has been left as future work.

3.4 Related Work

Certifying compilers and proof-carrying code The work presented in this chapter is a spiritual successor to a 25 year-long line of research on Proof-Carrying Code (PCC) started by Necula and Lee [97]. The original PCC proposal by Necula [96] was to supply proofs, in a logic embedded into Edinburgh Logical Framework [58], for executable binaries in DEC Alpha assembly language. This would allow the users of the binaries to independently check, that, upon execution, the code does not violate basic type and memory safety properties. In a follow-up work, Necula and Lee [99] have designed a *certifying* compiler, which would automatically generate such proofs when producing low-level assembly from code written in a high-level language. The ideas of PCC and certifying compilation have been studied extensively in the past two decades, in application to, *e.g.*, validation of temporal properties of systems code [62], hardware synthesis [54, 84], static analysis via abstract interpretation [15], refinement types [28], information flow control [13], security policies in software-defined networks [122], and other classes of statically enforceable program properties. Further research has also been conducted into minimising the size of the trusted code base required to validate the corresponding proofs [6].

There has also been a rich line of work that has investigated embedding the certified compilation process entirely within the automation facilities of a theorem prover that is relevant to this chapter. Li *et al.* [78, 79, 80] describe a strategy to compile functions in HOL to low level executable ARM assembly. Their work is based on the idea of embedding compilation steps as manually proven rewrite lemmas and then using automation to search and apply these rules to the initial program to generate and certify executable code automatically. The work by Myreen *et al.* [94] propose a translation-validation approach in HOL instead, first compiling the input to machine-code and then performing a proof-search using a number of user-supplied lemmas to attempt to prove the equivalence of the two programs, a process that may fail. Lammich *et al.* [72, 73] consider a refinement-based approach and compile a functional HOL program into LLVM assembly by using automation to search for a sequence of refinements that map from one to the other. In all these works, while the initial functional program is compiled to an imperative low-level format, in order to facilitate their automation these developments all assume a light-weight compilation process that does not significantly alter the control flow of the input program nor deal with complex heap-based data-structures as in the work in this chapter.

The research on verified extraction as part of the CakeML [71] project also touches on related themes. In particular, in their work, Myreen *et al.* [93] describe a technique for mapping a restricted subset of HOL expressions into the AST of an executable ML language with a proof certificate ensuring the equivalence of the two programs. A subsequent follow up [63] then extends this work to handle monadic functions using IO and state. As the input and output languages are restricted by design to be similar, the translation does not need to significantly alter the program and can be performed mechanically through a recursive traversal of the subterms of the input, with the main challenge being in correctly constructing the proof of equivalence rather than synthesising the resulting program as in this work.

This work presents an application of the ideas of PCC and certifying compilation—coupling generation of code and a machine-checked proof of its safety specification—to the area of automated program synthesis. Unlike the original work on PCC [96], which targeted basic type- and memory safety properties, this proposal instead focuses on a richer class of full functional correctness specifications.

Certified interactive program synthesis The FIAT framework [30, 37] implements a certified *interactive* program synthesiser for abstract data types (in OCaml), by embedding a synthesis procedure directly into the Coq proof assistant. Unlike in SUSLIK, the specifications in FIAT are represented by high-level non-deterministic programs, which are then refined [60], in a step-wise fashion, to more realistic implementations. FIAT facilitates certified synthesis by refinement: it provides tactics for synthesising refined implementations for certain restrictive domains and a library of lemmas that can be used by the clients for verifying derivations of more advanced implementations. In contrast with FIAT's approach to synthesis, which requires one to interactively verify a sequence of semantics-preserving optimisations, SUSLIK's synthesis is based on a fully automated proof search in a domain-specific logic.

3.5 Takeaways and Main Insights

This chapter has presented the second case study of this thesis, an investigation into the use of synthesis for verified software evolution through the design and implementation of an extension to the SuSLik program synthesiser to produce real world executable C code. The chapter began with a gentle tutorial on Watanabe *et al.*'s technique of proof interpreters, providing the appropriate background context and then describing the core idea behind proof interpreters themselves. The chapter then introduced the main contribution of this work, and described how Watanabe *et al.*'s framework was instantiated to make SuSLik produce executable C code and bridge the semantic gap between SusLang and C. Finally, the chapter presented an evaluation of this extension, and highlighted how it is able to produce reasonably sized proof scripts for *almost all* tasks in SuSLik's benchmark suite.

The construction presented in this chapter demonstrates the efficacy of synthesis for verified software maintenance. For a large portion of common programming tasks (*cf.* Table 3.1), one can augment a deductive synthesiser to generate executable implementations *alongside* proofs of correctness. Applying this methodology to functions whose specifications change frequently, these results demonstrate how synthesis can reduce the maintenance burden for verified software, as implementation and verification can be fully automated. The main downside however is the lack of control: auto-generated programs, while functionally correct, may suffer from other non-functional issues, such as making use of counter-intuitive logic or exhibiting poor runtime performance or memory usage, however, with this setting, the verified code cannot be modified without breaking their corresponding proofs — this leads to the natural follow up question: might there be a way to *repair* proofs over such changes?

CHAPTER

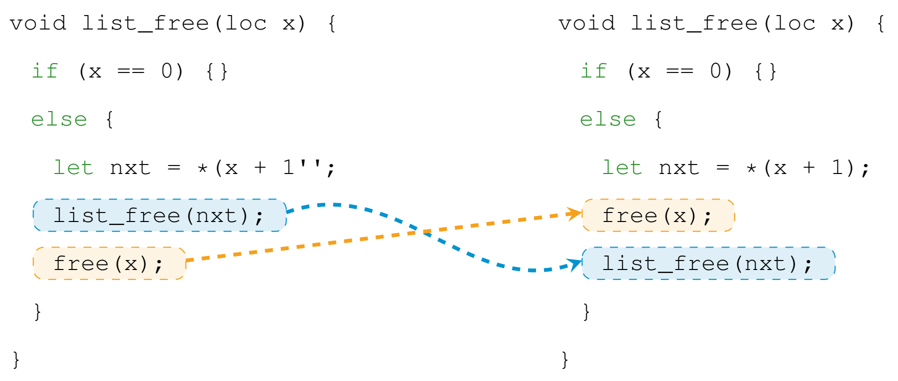
4

VERIFIED SOFTWARE MAINTENANCE THROUGH REPAIR

This chapter presents the last case study of this thesis, the design and implementation of Sisyphus*, a tool to automatically repair proofs of verified OCaml programs over changes in their implementation, and through it, investigates the use of repair for verified software maintenance. The chapter starts by introducing the common problem of maintaining the implementations of verified software, and discusses the challenges with updating verified source code without breaking their corresponding proofs of correctness. As an investigation into how best to mitigate the maintenance burden this can impose, the main technical result of this work,* Sisyphus*, is introduced: an automated tool that utilises the inherent information within proof scripts to automatically repair proofs of OCaml programs over a range of modifications of this kind. The remainder of the chapter presents the technical details of the design and implementation of* Sisyphus*. Finally, the chapter ends with a review of the main insights and takeaways gained from this investigation.*

How can one maintain verified software systems as they experience code evolution and change? The previous two chapters have investigated this problem through a preventative lens, proposing methodologies of structuring verification efforts to minimise the effects of modifications or entirely outsourcing both the verification and implementation of the aforesaid software to program synthesisers. Unfortunately, however, sometimes these preventative solutions are not sufficient, and instead a reactive approach is needed — sometimes, it may be more practical to directly *repair* the proofs themselves.

Consider the automatically generated `list_free` function from the previous chapter (*cf.* Figure 4.1, left). While this implementation is functionally correct — the constructions from the previous chapter were even able to automatically verify it as such — the program itself happens to suffer from a *subtle* limitation that makes it ill-suited for real-world applications: it is not *stack safe*. More specifically, as



```
void list_free(loc x) {
 if (x == 0) {}
 else {
  let nxt = *(x + 1'';
 list_free(nxt);
 free(x);
 }
}
```

```
void list_free(loc x) {
 if (x == 0) {}
 else {
  let nxt = *(x + 1);
 free(x);
 list_free(nxt);
 }
}
```

Figure 4.1: An optimisation to make `list_free` stack safe. Original on left, optimised on right.

the implementation recursively frees the tail of the linked list before freeing the head, the program will require stack frames to be allocated for each element of the linked list, and thus will use stack space linear to the length of its input, potentially causing stack overflows for large lists. Thankfully, this program is not beyond salvage — in fact, the fix is fairly simple: there is no need to wait for the tail to be deallocated before freeing the head, so simply swapping the these two statements (*cf.* Figure 4.1, right), makes the whole program tail recursive, enabling most reasonable compilers to apply tail call optimisations to make the implementation use constant stack space. While intuitively it's fairly clear that this optimisation does not affect the semantics of this implementation at all, unfortunately, this intuition is not sufficient to convince an interactive theorem prover, and thus the associated proofs of correctness will now fail to hold, leaving the developer with the laborious task of manually fixing them.

The focus of this work is on automated techniques for repairing proofs about imperative programs in response to *local* changes in their code while their specifications remain unchanged, and determining a practical solution to this problem. While there has been growing interest within the community into techniques for the repair of proofs, most notably the line of work by Ringer *et al.* [113, 114, 115] who introduced the very term, *proof-repair*, to the best of the author's knowledge, the problem of handling arbitrary changes in the *implementation logic*, has been largely unexplored from this perspective.

To explore this question, this chapter will investigate automating proof repair for verified OCaml programs. In particular, user-facing *libraries* form a significant class of programs whose specifications and data types change infrequently. By their nature, to enable forward-compatibility, library functions are expected to preserve their API and the contracts describing their interaction with the user code. Because of this, verified software libraries [9, 26, 106] are a sweet spot for proof repair: it is not uncommon for library functions to undergo changes in their bodies for the sake of improved performance or readability, whereas conversely modifications in their type definitions and specifications are relatively rare. This work introduces Sisyphus, the first mostly automated tool for performing proof repair for the changes in implementations of imperative OCaml programs verified in the CFML Coq framework, and uses it to demonstrate the efficacy of using repair for the purposes of verified software maintenance.

The main technical contributions of the work presented in this chapter are as follows:

- *Proof-driven testing*—a novel approach to test properties of data and programs for validity by extracting tests from proofs of higher-order facts that rely on those properties. This work provides the intuition and formal description of the proof-driven testing methodology and shows how to use it to prune the search-space of candidate invariants required for proof repair (Section 4.3).
- Sisyphus—a proof repair tool for OCaml programs verified in Coq. This work evaluates Sisyphus on a suite of 14 evolved programs, of which 10 were drawn from popular OCaml libraries, and found that all inferred invariants for new versions were valid and required relatively small amounts of manual effort to prove in comparison to the original programs (Section 4.4).

The remainder of this chapter will present the technical details of this case study, starting with a step by step walk through of the overall repair process, before detailing the individual components of the implementation itself. Finally, the chapter ends with an evaluation of the tool on a representative set of 10 *real-world* OCaml programs. The technical content of this chapter is a revision of work that has been published before at the PLDI conference series and can be found here [49].

4.1 The Labours of Sisyphus

This section presents an overview of Sisyphus by means of an illustrative example: repairing the proof of correctness between two versions of a real-world program from a widely-used OCaml library. The program in question is the function `Seq.to_array`, which converts a lazy sequence to an array, taken from the popular `containers` library between versions 3.6[1] and 3.7,[2] wherein it was updated to improve its performance. The initial version of `to_array` was manually verified in Coq via CFML; Sisyphus was then used to automatically generate a repaired proof for the updated program.

[1] `to_array` in `containers` 3.6: https://github.com/c-cube/ocaml-containers/blob/v3.6/src/core/CCSeq.ml#L397

[2] `to_array` in `containers` 3.7: https://github.com/c-cube/ocaml-containers/blob/v3.7/src/core/CCSeq.ml#L415

```
let to_array s =
  match s () with
  | Nil -> [| |]
  | Cons (hd, _) ->
    let sz = length s in
    let a  = Array.make sz hd in
    iteri
      (fun i vl ->
         a.(i) <- vl) s;
    a
```

Figure 4.2: Original `to_array`

4.1.1 An Initial Verified `Seq.to_array`

Figure 4.2 presents the original implementation of the OCaml library function `to_array`, which converts a sequence into an array, where a sequence of type `'a t` is encoded as a thunked list:

```
type 'a t     = unit -> 'a node
and  'a node = Nil
              | Cons of 'a * (unit -> 'a node)
```

Here, a sequence is represented by a function that, when evaluated, either returns the constructor **`Nil`** for the empty sequence, or returns a **`Cons`** cell with a head element and a tail that can be evaluated on-demand to retrieve the rest of the sequence. This encoding of sequences can even be used to represent lists of infinite length or encode arbitrary side-effects within the thunks of a sequence, however, for the purposes of this work, the narrative will be restricting its focus to the case in which these sequences are finite and side-effect free, since `to_array` has undefined behaviour for other cases.

The implementation of `to_array` is then simply as follows. If the sequence input contains at least one element the function calculates its length using a helper function and allocates a fresh result array with the capacity to store all of the elements of the sequence. The function then calls the *higher-order* iterator function `iteri`, whose argument function successively assigns elements of the sequence to the corresponding slots of the allocated array, which is eventually returned as the result. Now, having stepped through the function, it seems reasonable to believe that `to_array` is correct, and the `containers` library itself comes with an extensive test suite. But if the goal is to truly *ensure* the correctness of this implementation of `to_array`, then one must really formally *prove* its correctness.

Specification of `Seq.to_array`

Before verifying any code, it is necessary to first decide upon a specification to capture what exactly it means for this implementation to be *correct*. In order to faithfully specify the effect of `to_array` on the heap, one can return back to Separation Logic (SL) [100, 112] as discussed in the previous chapter. In this formalism, `to_array` can be ascribed the following SL specification:

$$\forall s\, \ell, \{s \mapsto \texttt{Seq}\, \ell\}\ (\texttt{Seq.to\_array}\ s)\ \exists a, \{a \mapsto \texttt{Array}\, \ell\} \tag{4.1}$$

The precondition, $\{s \mapsto \texttt{Seq}\, \ell\}$, states that the payload of the (universally quantified) input sequence s is modelled by a logical list ℓ, where the predicate `Seq` captures the fact that s is a finite sequence without side-effects. The postcondition, $\exists a, \{a \mapsto \texttt{Array}\, \ell\}$, adopts the CFML framework's convention of using existential quantifiers to encode return values and asserts that the function will return some pointer a that points to an array with contents described by ℓ, where the predicate `Array` encodes the fact that a points to an array on the heap.[3] In other words, the specification (4.1) asserts that `to_array` indeed converts a sequence to an array with the same payload, and now, if proven, provides a meaningful guarantee about the correctness of this function. Finally, note that while the post-condition does not constrain the input sequence s, this does not actually affect the usability of this specification as such sequences have been defined, as mentioned before, to be pure and effect-free, so they are immutable and thus allowed by CFML's affine logic to be duplicated before passing them to the function.

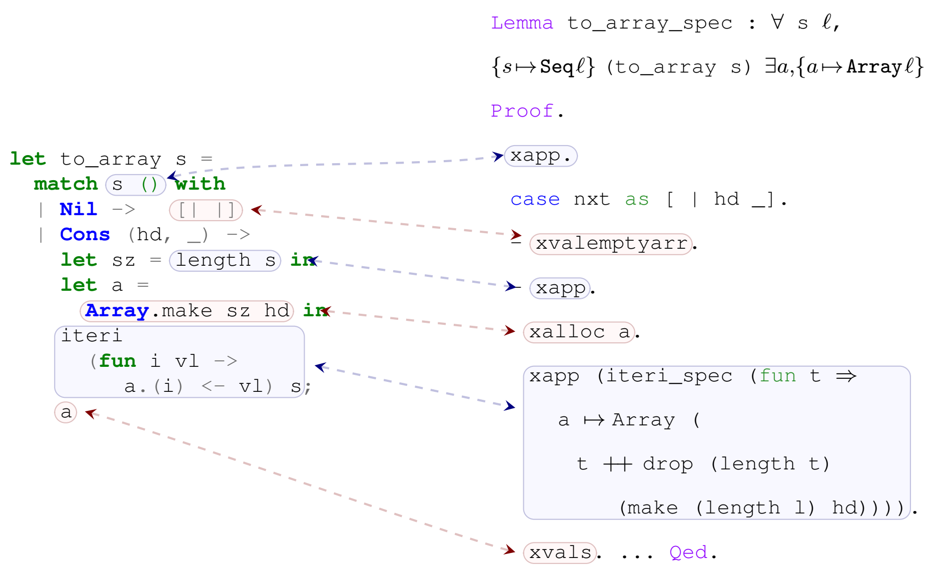
A mechanised proof for `Seq.to_array`

Having introduced the prerequisites, it is now time to verify the original `to_array` function in Coq. When proving properties about these kinds of heap-manipulating programs in a proof assistant, the corresponding proofs follow by stepping through the code using the reasoning rules of the program logic to symbolically evaluate how its individual statements update the symbolic state.

Figure 4.3 shows the correspondence between `to_array` and the Coq proof (done using the CFML embedding of SL) that establishes that the program indeed satisfies the specification (4.1). As is common with such SL implementations, each rule of the logic comes with an associated Coq *tactic* that applies the rule, automatically determining which heaplets are affected by the rule (*i.e.*, its *footprint*). As an example, consider the generic rule XAPP from the CFML framework for verifying an application of a function f with a continuation c (ignoring the highlighted premise for now):

$$\textsc{xapp}\ \frac{\{P\}\ \ (f\ \overline{v})\ \ \exists x, \{Q' x\} \qquad \forall x, \{Q' x\}\ c\ x\ \{Q\}}{\{P\}\ (\texttt{let x} = f\ \overline{v}\ \texttt{in}\ c\ \texttt{x})\ \{\texttt{Q}\}} \tag{4.2}$$

[3]This assumes an OCaml-style memory model, where locations can store arbitrary values, *e.g.*, arrays.



```
let to_array s =
  match s () with
  | Nil -> [| |]
  | Cons (hd, _) ->
    let sz = length s in
    let a =
      Array.make sz hd in
    iteri
      (fun i vl ->
         a.(i) <- vl) s;
    a
```

```
Lemma to_array_spec : ∀ s ℓ,
{s↦Seqℓ} (to_array s) ∃a,{a↦Arrayℓ}.
Proof.
xapp.
 case nxt as [ | hd _].
 xvalemptyarr.
 xapp.
 xalloc a.
 xapp (iteri_spec (fun t ⇒
   a ↦ Array (
     t ++ drop (length t)
       (make (length l) hd)))).
 xvals. ... Qed.
```

Figure 4.3: Proof of `to_array` in CFML.

In a Coq embedding of CFML, this rule is implemented as a tactic `xapp`, which applies a lemma that discharges the conclusion of the rule by emitting verification conditions as per its premise.

```
{s ↦ Seq ℓ}
let a = Array.make sz hd in
{s ↦ Seq ℓ ∗ a ↦ Array (repeat sz hd)}
```

Figure 4.4: Example application of the `xapp` tactic.

Consider the call to **`Array`**`.make` in `to_array` and the corresponding fragment in the proof in Figure 4.3. In the program, this expression allocates an array of size `sz` initialised with the value `hd` in the heap and returns a pointer to the freshly allocated and initialised block of memory. In the corresponding proof step, reasoning about this computation is handled using a tactic `xalloc`, which updates the symbolic program state (and is captured correspondingly by the Coq proof context) to reflect the semantics of the operation, as shown in the proof snippet in Figure 4.4.

Under the hood, the `xalloc` tactic itself operates by applying a corresponding reasoning rule for function application, `xapp` (4.2), to the specification of the library function **`Array`**`.make`:

$$\forall sz\ v, \{sz > 0; \mathsf{emp}\}\ (\texttt{Array.make}\ sz\ v)\ \exists a, \{a \mapsto \texttt{Array}\ (\texttt{repeat}\ sz\ v)\} \tag{4.3}$$

In particular, the spec (4.3) asserts that when calling **`Array`**`.make` with a size sz and value v where $sz > 0$, the return value of the function will be a pointer `a` to an array, whose contents are described by the logical expression `repeat` $sz\ v$; that is, the array has sz copies of v.

The most interesting part of the proof is reasoning about the use of the higher-order function `iteri`, which takes as arguments a possibly-effectful function `f` and a sequence s, and iterates through the sequence, calling `f` on each element. In order to characterise the state of the heap *after* its invocation, the specification (4.4) of `iteri` requires an *invariant* I that must be maintained by `f`:

$$\forall I\ \texttt{f}\ s\ \ell, (\forall t\ v, \{I\ t\}\ \ (\texttt{f}\ v)\ \ \{I\ (t +\!\!+\ [v])\}) \rightarrow \{I\ \texttt{[]} * s \mapsto \texttt{Seq}\ \ell\}\ (\texttt{iteri f}\ s)\ \{I\ \ell * s \mapsto\ \texttt{Seq}\ \ell\} \quad (4.4)$$

Here, invariant I characterises the heap in terms of the *prefix* t of the sequence that has been visited by `iteri`. The premise of the specification asserts that the function call `f` v preserves the invariant I after visiting v, *i.e.*, I now holds over an extended prefix. The conclusion of the specification (in grey) states the effect of `iteri` on the state. Initially, none of the elements of s have been seen, so I must hold on the empty list `[]` in the precondition, constraining the corresponding part of the heap. As per the postcondition, after executing `iteri f` s, every element in the sequence has now been visited, and so the specification asserts that I now holds over *all* elements in the sequence, ℓ.

The proof in Figure 4.3 makes use of the conclusion of the specification (4.4), referred to as `iter_spec` in the Coq code, by providing it as the highlighted premise of the rule (4.2). To do so, it *instantiates* `iter_spec` with a suitable invariant I. This invariant argument *does not* directly follow from the syntax of the program like other components of the proof, but, rather, must be *explicitly* provided by the user. The invariant provided in this proof states that at each iteration of `iteri`, the allocated array `a` will always start with the sequence of elements t that have been visited by the iteration:

$$\texttt{fun}\ t \Rightarrow \texttt{a} \mapsto \texttt{Array}\ (t +\!\!+\ \texttt{drop}\ (\texttt{length}\ t)\ (\texttt{repeat}\ (\texttt{length}\ \ell)\ \texttt{hd})) \quad (4.5)$$

In particular, this invariant asserts that the value `a`, previously returned by the call to `Array.make`, will point to an array with the same length as ℓ, where the contents of the array start with t, the *prefix* of visited elements, and the remaining elements are all `hd`. At the precondition of `iteri`'s specification, the proof instantiates this invariant with the empty list, where `a` points to a list whose length is equal to that of ℓ and whose elements all have the value `hd`. Having executed `iteri`, the invariant in its postcondition is instantiated with the full sequence ℓ, and so it is revealed learn that the final contents of the array `a` must have the same length as ℓ, with its contents starting with the sequence ℓ and followed by an empty suffix. Therefore, it can then be shown that the contents of `a` are exactly the sequence ℓ, thereby satisfying the post-condition of `to_array` and concluding the proof.

```
1  let to_array s =
2    let sz, rls =
3     fold (fun (i,ls) x ->
4       i+1, x::ls) (0, []) s in
5    match rls with
6    | [] -> [| |]
7    | init :: rest ->
8      let a = Array.make sz init in
9      let idx = sz - 2 in
10     let _ = List.fold_left
11         (fun i x -> a.(i) <- x;
12            i - 1) idx rest in a
```

Figure 4.5: New version of `to_array`

4.1.2 A Recipe for Proof Repair

The proof in Figure 4.3 serves as a certificate of correctness for `to_array`, but, alas, only for one particular version of the function: if the implementation of `to_array` were to later change, then this certificate would no longer hold, and the developer would have to prove correctness of the program again. This definition of `to_array` in the `containers` library was later updated to the implementation in Figure 4.5, adopting a radically different, but more efficient implementation that avoids repeated evaluation of the input sequence. In the new implementation, the function first traverses the entire sequence in a single pass (line 3) using the `fold` iterator to fold over the elements of the sequence, using a *pure* function to accumulate a tuple of the length of the sequence and the elements of the sequence in reverse (line 4). Then, having allocated a result array of a suitable length, the program simply iterates over the reversed *list* of elements using `List.fold_left` (line 10), and assigns the elements to its result array in reverse (lines 11-12), gradually increasing the *suffix* of the array that is shared with the input. In this way, while both versions traverse the elements twice, the two implementations differ in the number of times the lazy sequence is "forced": in the original version the sequence is forced twice (by `length` and `iteri`), while the updated implementation only forces it once (using `foldi`). This can have significant performance benefits when the elements themselves capture expensive computations.

While this new version significantly differs from the original implementation of `Seq.to_array` from Figure 4.2, one can observe some striking similarities in their implementations. Most significantly, both programs follow the *same* high-level steps: first (1) to calculate the length of the sequence, then second (2) to allocate a result array with an appropriate size, and finally (3) to populate this array with the elements of the sequence. For example, in the old implementation, step (1) is completed using a dedicated `length` function, while in the new program, this is achieved in parallel with accumulating a reversed list of the elements of the sequence using `fold`. Can these similarities between the versions of the program be somehow used to thus repair the correctness proof from Figure 4.3?

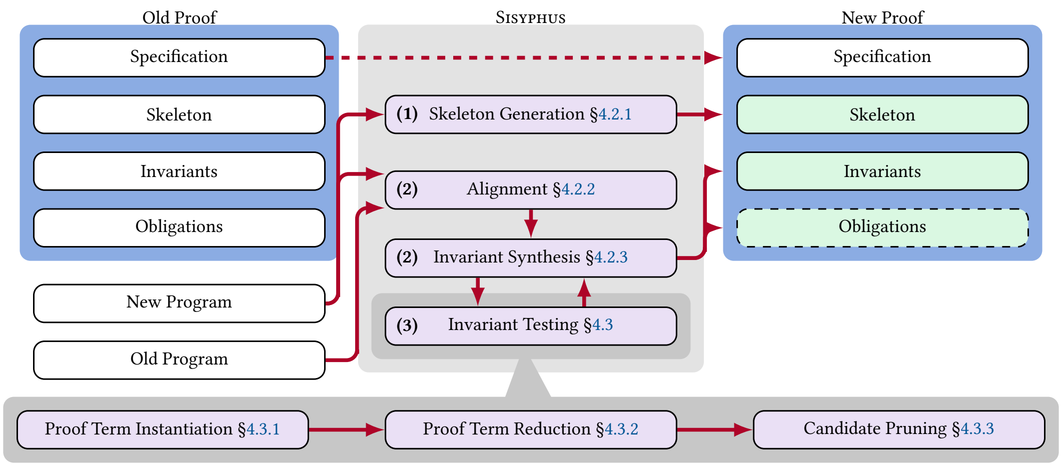


Figure 4.6: Sisyphus overview. White boxes represent user-provided components. Light-green boxes represent outputs generated by the tool with the dashed border around obligations representing a best-effort attempt at dispatching residual goals using `user_solve` or otherwise leaving them as admits. Solid arrows indicate inputs and outputs, and the dashed arrow encodes that specifications are copied.

The remainder of this section describes the tool Sisyphus developed in this work that does exactly that.

Overview of *Sisyphus* The high level overview of Sisyphus is shown in Figure 4.6, which highlights the three main stages in Sisyphus's proof repair process:

1. First, following the syntactic structure of the new program, Sisyphus constructs its *proof skeleton*, with holes for the parts that cannot be immediately inferred (item 4.1.2).

2. Next, Sisyphus compares traces of the old and new programs on the same random inputs in order to recover high-level relations between the individual steps in their implementations. It uses these relations to discover relevant sections of the old proof, which are then used to synthesise candidate expressions to fill holes in the new proof skeleton (Section 4.1.2).

3. Finally, Sisyphus uses a *fast* dynamic test to prune the space of synthesised expressions, instantiating the holes in the proof-skeleton and attempting to dispatch the resulting obligations using user-provided proof automation, thereby completing the proof script (Equation 4.1.2).

In this way, Sisyphus implements a *mostly-automated* proof repair procedure — the tool will generate a proof-skeleton and invariants for the new proof, but residual obligations are handled in a best-effort fashion. In particular, pure domain-specific logical obligations often remain in SL proofs after symbolically reasoning about the program (*e.g.*, proving that a particular integer is a valid index into a list). Sisyphus allows the user to supply a tactic (which is referred to as `user_solve`), that is invoked during repair to dispatch any such obligations. In order to construct such a tactic, one will typically use a mixture of Coq's hint databases and proof search such as `auto`/`eauto`. If the tactic fails to dispatch a

```
Lemma to_array_spec : ∀ s ℓ,
{s ↦ Seq ℓ} (to_array s)
    ∃a, {a ↦ Array ℓ}
Proof.
  (* .. *) xapp (* .. *).
  (* .. *) case ls as [ | init rest].
  - (* .. *) xvalemptyarr. {
      user_solve. }
  - (* .. *) xalloc a. (* .. *)
    xapp (list_fold_left_spec
      (fun (acc: int) (t: list A) ⇒ □)
          ).
      { user_solve. }
    (* .. *) xvals. { user_solve. }
Qed.
```

Figure 4.7: Proof skeleton for new `to_array`

goal, Sisyphus emits an `admit` for that subgoal, and the user should fill it in to complete the proof.

Building a proof skeleton

As has been seen in Equation 4.1.1, when verifying programs using SL program logics in Coq, the structure of the corresponding proof scripts will often mirror the structure of the program being verified. Applying this insight in reverse, Sisyphus starts its own repair process by first constructing an initial skeleton proof script for the new program, traversing the program body and mapping program constructs to the corresponding relevant tactics to symbolically execute them. Any residual logical obligations that remain following this process are delegated to a user-provided solver tactic to dispatch (here, `user_solve`), or admitted and left for the user in cases when the solver fails. Using this strategy, Sisyphus can automatically generate a proof skeleton for the new `to_array` function (Figure 4.7). Each program statement in Figure 4.5 maps to a particular tactic application in the proof skeleton: function applications to `xapp` (lines 4 and 8), array allocation to `xalloc` (line 7), creating an empty array to `xvalemptyarr` (line 6), and branching in the program is mirrored by a case analysis in the proof (line 5).

While this strategy automatically handles a large component of the burden of writing the new proof script, there may still be certain parts of the new proof which cannot be immediately filled in and must be left as holes. For the running example, such a hole must be left when reasoning about the application of **List**.`fold_left` in the program, as its specification (4.6) takes an explicit invariant

I to characterise how the program state is updated and maintained through its execution.

$$\begin{gathered}\forall I\ \mathtt{f}\ s\ acc'\ \ell, (\forall\ acc\ t\ v, \{I\ acc\ t\}\ (\mathtt{f}\ acc\ v)\ \exists res, \{I\ res\ (t +\!\!+ [v])\}) \to \\ \{I\ acc'\ \mathtt{[]} * s \mapsto \mathtt{List}\ \ell\}\ (\mathtt{List.fold\_left\ f}\ s)\ \exists res, \{I\ res\ \ell * s \mapsto \mathtt{List}\ \ell\}\end{gathered} \tag{4.6}$$

The specification (4.6) of `fold_left` is fairly similar to the previously seen specification (4.4) of `iteri`; the key difference between the two being that the invariant used to constrain the behaviour of the user-supplied function `f` now takes two parameters: an accumulator value acc, and a list t. As before, the t parameter represents a *logical* variable, the prefix of the sequence of elements that have been visited so far. The acc parameter represents a program-level value, which is the result accumulated by the fold. The invariant provided to this specification must capture the shape of the heap that evolves over the execution of the fold at each iteration, and it does not easily follow from the program syntax, so Sisyphus leaves a hole (☐) in the proof (Figure 4.7) in place of the invariant body to be filled in later.

Transplanting invariants across proofs

To fill the remaining holes in the proof, Sisyphus builds upon a fairly rudimentary observation: different versions of a particular program will more often than not share similar logical invariants in their proofs of correctness. More precisely, the statements between different versions of a program that perform similar operations (*e.g.*, populating an array), will often in turn share similarities in the assertions used to reason about the state they alter in their corresponding proofs, as in the snippets in Figure 4.8.

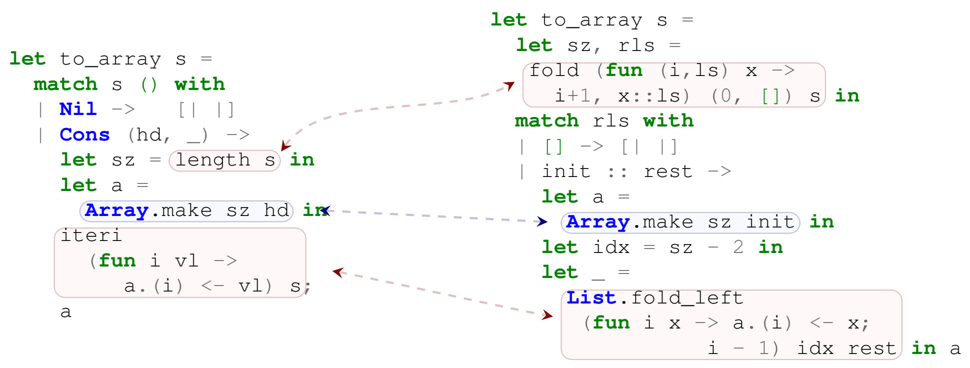
```
(* old: populate prefix of array *)
iteri (fun i vl -> a.(i) <- vl) s

(* new: populate suffix of array *)
List.fold_left (fun i x -> a.(i) <- x; i-1) idx rest
```

Figure 4.8: Old and new operations to populate the buffer in `to_array`

Following this intuition to its conclusion leads to a natural two-step process for instantiating holes in the new proof-skeleton: first, (1) in order to determine which statements in the new program perform similar operations to those from the old program, and then (2) to use relevant invariants from the old proof to guide the generation of invariant candidates to fill the holes in the new proof.

Discovering similar computations Next, in order to instantiate holes in the proof skeleton, a mapping between the high-level steps of old and new programs is required. In other words, what is needed is a *program alignment* between the two programs that will relate individual statements of both programs that correspond to the same high-level steps. These relations between the high-level steps in



```
let to_array s =
  match s () with
  | Nil ->    [| |]
  | Cons (hd, _) ->
  let sz = length s in
  let a =
    Array.make sz hd in
  iteri
    (fun i vl ->
       a.(i) <- vl) s;
  a
```

```
let to_array s =
  let sz, rls =
   fold (fun (i,ls) x ->
     i+1, x::ls) (0, []) s in
  match rls with
  | [] -> [| |]
  | init :: rest ->
    let a =
     Array.make sz init in
    let idx = sz - 2 in
    let _ =
     List.fold_left
      (fun i x -> a.(i) <- x;
                i - 1) idx rest in a
```

Figure 4.9: Alignment between two versions of `to_array`

a given program technically capture deep semantic properties of the implementation that might be hard to determine statically, but, as it turns out, can actually be discovered dynamically using a fairly simple strategy. Consider the old and new versions of `to_array`, which adopt the same general strategy to implement the conversion from a sequence to array (Figure 4.9): (1) compute the length of the input sequence, (2) allocate an array for the result, and (3) populate the elements of the array. While both programs use the same steps, the implementations of these individual steps can significantly differ in execution. For example, when populating the result array in step (3), the original implementation fills the array from the beginning, while the new implementation does it by assigning elements from the end. However, after executing this step in both programs, the result array will contain exactly the elements of the input sequence—a fact that is easy to capture during executions of both programs:

$\{\mathbf{sz} = 3; \mathbf{a} \mapsto \texttt{Array}\ ([1;1;1])\}$

```
iteri (fun i vl -> a.(i) <- vl) s
```

$\left\{\mathbf{sz} = 3; \mathbf{a} \mapsto \texttt{Array}\ (\boxed{[1;2;3]})\right\}$

$\{\mathbf{sz} = 3 \land \mathbf{idx} = 1; \mathbf{a} \mapsto \texttt{Array}\ ([3;3;3])\}$

```
List.fold_left (fun i x -> a.(i) <- x; i - 1) idx rest
```

$\left\{\mathbf{sz} = 3 \land \mathbf{idx} = 1; \mathbf{a} \mapsto \texttt{Array}\ (\boxed{[1;2;3]})\right\}$

In order to exploit this insight for generating new invariants, Sisyphus executes both versions of the program in question on the same randomly generated inputs and records observations of the concrete runtime program states at each program point. These observations between the traces are ranked based on their *similarity* by using certain metrics to compare the values of program variables in each observation. In particular, Sisyphus uses two such metrics: comparing exact matches in runtime values, and barring an exact match, comparing the sizes of collections such as lists, albeit with a lower priority. That is, if many program variables have the same runtime values, then two observations are considered to be highly similar. A statement in one program is considered as similar to a statement in the other program when the observations at points surrounding the respective statements are similar.

In this way, Sisyphus builds up an *alignment* between the individual statements of both respective programs. For example, in the code in Figure 4.9 the call to `iteri` on the left is discovered by Sisyphus to be similar to the call to `fold_left` on the right as both produce arrays with the same payload, even though they do so in different directions, hence both statements are marked as aligned.

Generating invariant candidates When instantiating a hole in the new proof skeleton, Sisyphus first uses the calculated alignment to find the relevant statements of the old program that plausibly correspond to the same high-level step as the current statement in the new program being analysed that needs an explicit invariant in its proof. BY considering the respective proof steps from the old proof that correspond to these statements, Sisyphus then collects any expressions that occur in symbolic states preceding these steps and produces a family of *sketches* that are likely to capture the similarities in the invariants between the old and new proofs by replacing sub-expressions within those invariants with holes (_). For example, when looking to synthesise an invariant for the call to `fold_left` in the proof of the new program, Sisyphus discovers an aligned statement—a call to `iteri`—and the invariant (4.5) used to verify it in the corresponding proof (Figure 4.3). It then extracts from the old invariant a number of sketches, one of them being `(_ ++ drop _ _)`, which captures the prefix/suffix argument from the proof of the old program and contributes the following (simplified) invariant template:

```
fun (acc: int)(t: list A)⇒a ↦ Array (_ ++ drop _ _)
```
(4.7)

Finally, Sisyphus uses this template (amongst others), along with any logical functions and constants in state assertions in the proof of the old and new program, to bootstrap an enumerative synthesis procedure for generating concrete invariant candidates for the current hole in the proof in Figure 4.7.

Validating invariant candidates

The last step in Sisyphus' repair process is to validate the generated invariants and identify suitable candidates to instantiate the holes in the proof skeleton. As the enumerative synthesis based generation strategy can produce quite large numbers of candidates, validating each candidate individually by trying to prove its correctness would quickly become intractable. As such, Sisyphus implements its candidate validation step in two phases, first running a *fast* dynamic test to quickly prune generated candidates, and then using the user-provided solver to actually prove the invariant. The key insight powering Sisyphus is that while such *dynamic* tests for invariants may be challenging to generate from scratch, one can actually *automatically* construct these tests from information hidden within the proofs of the higher-order functions, using a novel technique this work names as *proof-driven testing*.

Consider the invariant candidates for the call to `fold_left` in the new program generated from the

Number	Logical embedding as Coq predicate	Executable embedding as OCaml function	Valid?
(1)	`(fun acc t => a ↦ Array (repeat (acc + 1) init ++ drop (acc + 1) ℓ))`	`(fun acc t -> Array.to_list a = (repeat (acc + 1) init ++ drop (acc + 1) ℓ))`	Yes
(2)	`(fun acc t => a ↦ Array ([] ++ drop 0 (repeat (length t) init)))`	`(fun acc t -> Array.to_list a = ([] ++ drop 0 (repeat (length t) init)))`	No

Table 4.1: Mapping from Coq invariants to OCaml tests

template (4.7) and listed in Table 4.1.[4] The first candidate accurately describes the invariant that holds for one iteration of `fold_left`'s argument function, while the second invariant is incorrect.

Following the intuition presented in the earlier sections, the first invariant asserts that during the execution of `fold_left`, the contents of the array `a` will share an increasing *suffix* with the contents ℓ of the original sequence. The second invariant asserts that the contents of `a` will always be described by `repeat (length t)init`—that is, all elements in the array will always have the same value. The goal is to test these invariants and quickly distinguish between the invalid invariant (2) and the correct invariant (1). To do this, one needs to convert the logical state properties asserted by the invariant and expressed as propositions in Coq's logic (second column of Table 4.1), into executable OCaml tests (third column) that check the program values of the program via a direct syntactic translation. Combining this with a suitable instantiation for the free variables in the expressions (*i.e.*, `a` and ℓ), it would then be possible to construct an executable reference program that could test the invariants.

```
let ℓ     = [ 1; 2; 3 ] in
let a     = [| 3; 3; 3; |] in
let f i x = a.(i) <- x; i-1 in
let len = 3 - 2 in
let inv = (* .. *) in
(* a = [ 3; 3; 3 ] *)
assert (inv len []);
let acc = f len 2 in
(* a = [ 3; 2; 3 ] *)
assert (inv acc [2]);
let acc = f acc 1 in
(* a = [ 1; 2; 3 ] *)
assert (inv acc [2;1])
```

Figure 4.10: A concrete invariant test

Now, suppose one had access to the testing program presented in Figure 4.10. This program encodes a particular *concrete* execution trace of `List.fold_left` within the new implementation of `to_array` called on a sequence with the elements $[1, 2, 3]$, annotated with appropriate assertions (`inv`). Specifically, after each individual step in the trace, the test program includes an additional assertion that an invariant

[4]It is assumed that the invariants are used in an environment where `a` and ℓ are bound, *e.g.*, the proof in Figure 4.7.

```
let rec fold_left (* I *) f acc ls =
  (* assert (I acc []); *)
  let rec loop (* t *) acc ls =
   match ls with
   | [] -> acc
   | hd :: tl ->
     let acc' = f acc hd in
     (* assert (I acc' (t ++ [hd])); *)
     loop (* (t ++ [hd]) *) acc' tl in
 loop (* [] *) acc ls
```

Figure 4.11: Implementation of `fold_left`

function `inv` must hold with the current value of accumulator variable `acc` and the prefix of the elements that have been visited so far. If one were to use the first invariant candidate as the definition of the `inv` function, then this testing function will execute without raising an exception, as each time `inv` is called, the contents of the array `a` will indeed match the state expected by the invariant. Conversely, if this program is executed with the definition of `inv` assigned to the second invariant candidate, then the program will fail to execute to completion, as at the second call to the `inv` function, the contents of the array will be $[3; 2; 3]$ while the invariant will expect all elements to be the same.

The challenge with constructing such a test program is that it manipulates logical variables and performs invariant checks that do not occur in the implementation of `fold_left` (as given in Figure 4.11). Note however that these properties *could* actually be obtained if considering the execution trace of a suitably annotated ideal version of `fold_left` (*cf.* the comments in Figure 4.11) that was parameterised by additional logical parameters (*i.e.*, `t`) and contained explicit invariant checks.

The key insight of Sisyphus is that one *does* have access to these very annotations: they are in fact contained within *the proof of correctness* for `fold_left` with regard to the specification (4.6), which itself has to manipulate and maintain these variables and checks in order to establish that the invariant holds over the course of the entire program. Sisyphus *evaluates* proof terms of specifications for higher-order functions (as well as correctness proofs for rules of a program logic) on concrete inputs, extracting test specifications from the resulting reduced proof terms as presented in Figure 4.10, the aforementioned process this work dubs *proof-driven testing*. Using these generated tests, Sisyphus can quickly prune the generated candidates and suggest valid invariants necessary to complete the repaired proof.

Putting it all together

Recall that the starting point of this section was simply an old program, its SL specification, and a Coq proof of its correctness (Subsection 4.1.1). Taking those artifacts and running them through the

$$
\begin{array}{lll}
v & & \text{variable names} \\
c & ::= & v \mid \texttt{fun}\ \overline{\texttt{v}} \Rightarrow \texttt{c} \mid \texttt{assert c} \\
 & & \mid c\ c \mid \texttt{let v =}\ \texttt{c in c} \\
 & & \mid \texttt{if c then c else c} \\
 & & \mid \texttt{match}\ v\ \texttt{with}\ \overline{C_i\ \overline{v_i} \rightarrow c_i} \\
T & ::= & \texttt{xval.} \mid \texttt{xif.}\ \overline{\{T\}} \\
 & & \mid \texttt{xletval v.} \mid \texttt{xmatch.}\ \overline{\{T\}} \\
 & & \mid \texttt{xapp.} \mid \texttt{xapp v (fun}\ \overline{\texttt{v}} \Rightarrow \square\texttt{).}
\end{array}
$$

Figure 4.12: OCaml programs and Coq tactics

sequence of steps described in item 4.1.2–Equation 4.1.2, Sisyphus produced a nearly complete repaired proof with explicitly-stated high-confidence invariants for calls to higher-order functions and loops. To fully complete the proof, the user has to mechanically establish the validity of the suggested invariants. These residual proof obligations typically boil down to proving new facts about (a) entailment of SL assertions and (b) mathematical properties of involved data types (*e.g.*, a new lemma relating list reversal and concatenation)—all outside of the scope of the original proof, and, therefore, beyond the reach of Sisyphus's proof repair capabilities.

In the remaining sections this chapter will provide detailed descriptions of the outlined algorithms for reconstructing proof skeletons, computing program alignment, and synthesising invariant candidates (Section 4.2), formally define proof-driven testing via Coq proofs (Section 4.3), and elaborate on the utility aspects of Sisyphus, including the proof burden for the residual verification conditions (Section 4.4).

4.2 Proof Reconstruction and Synthesis of Invariant Candidates

This section formal describes the algorithm used by Sisyphus to reconstructing the initial proof skeleton for the updated program, and the invariant generation algorithm used to fill the holes in the skeleton.

4.2.1 Generating Proof Skeletons

Figure 4.12 provides a subset of OCaml terms (ranged over by c) and CFML proof tactics (T), which will be used for the presentation in the remainder of the chapter. For the sake of a uniform treatment, Sisyphus requires loops in OCaml programs to be encoded as applications of higher-order combinators that take a function, representing the loop's body, as their argument. This reduces the problem of inferring loop invariants to inferring invariants for higher-order function applications. In practice, this convention poses little problem: most idiomatic OCaml programs that involve loops are either already implemented using higher-order combinators or can be easily rewritten to do so. Of note in the syntax,

$$\textsc{XIF}\quad \dfrac{\{v = \texttt{true}; P\}\{Q\}\ c_t \rightsquigarrow_{\text{x}} T_t; \varphi_{\texttt{t}} \qquad \{v = \texttt{false}; P\}\{Q\}\ c_f \rightsquigarrow_{\text{x}} T_f; \varphi_{\texttt{f}}}{\{P\}\ \{Q\}; \texttt{if}\ v\ \texttt{then}\ c_t\ \texttt{else}\ c_f \rightsquigarrow_{\text{x}} \texttt{xif.}\ \{T_t\}\{T_f\};\ (\varphi_t, \varphi_f)}$$

$$\textsc{XMATCH}\quad \dfrac{\{v = C_i\ \overline{v_i}; P\}\{Q\}\ c_i \rightsquigarrow_{\text{x}} T_i;\ \varphi_i}{\{P\}\ \{Q\}; \texttt{match}\ v\ \texttt{with}\ \overline{C_i\ \overline{v_i} \Rightarrow c_i} \rightsquigarrow_{\text{x}} \texttt{xmatch.}\ \overline{\{T_i\}};\ \overline{\varphi_i}}$$

$$\textsc{XVAL}\quad \dfrac{}{\{P\}\ \{Q\};\ v \rightsquigarrow_{\text{x}} \texttt{xval.};\ P \vdash Q\ v}$$

$$\textsc{XAPP}\quad \dfrac{\mathcal{E}[f] = \forall\, \overline{x}, \overline{\psi} \to \{P\}\ \ f\ \overline{x}\ \exists x\ \{Q'\ x\} \qquad \{(Q'x)[\overline{v}/\overline{x}]\}\{Q\};\ c\ x \rightsquigarrow_{\text{x}} T;\ \varphi}{\{P\}\ \{Q\}; \texttt{let}\ x = f\ \overline{v}\ \texttt{in}\ c\ x \rightsquigarrow_{\text{x}} \texttt{xapp.}\ T;\ (\overline{\psi}[\overline{v}/\overline{x}], \varphi)}$$

$$\textsc{XAPP-HOF}\quad \dfrac{\mathcal{E}[f] = (s : \forall F, \overline{x}, I, \overline{\psi} \to \{P\}\ f\ F\ \overline{x}\ \exists x\ \{Q'\ x\}) \quad d \text{ is first-order} \qquad \{(Q'\ x)[\overline{v}/\overline{x}, (\texttt{fun}\ \overline{w} \Rightarrow d)/\texttt{F}, (\texttt{fun}\ \overline{q} \Rightarrow \square)/\texttt{I}]\}\{\texttt{Q}\};\ \texttt{c}\ x \rightsquigarrow_{\text{x}} T;\ \varphi}{\{P\}\ \{Q\}; \texttt{let}\ x = f\ (\texttt{fun}\ \overline{w} \Rightarrow d)\ \overline{v}\ \texttt{in}\ c\ x\ \rightsquigarrow_{\text{x}} \texttt{xapp}\ s\ (\texttt{fun}\ \overline{q} \Rightarrow \square).\ T;\ (\overline{\psi}[\overline{v}/\overline{x}, (\texttt{fun}\ \overline{w} \Rightarrow d)/F, (\texttt{fun}\ \overline{q} \Rightarrow \square)/I], \varphi)}$$

Figure 4.13: Transformation from OCaml to CFML proof scripts (sequences of tactics)

the tactic `xapp` that implements the CFML logic rules for function applications has two forms—the former handles first order functions while the latter tackles higher-order functions and requires an explicit invariant, which is initially represented by a "blank" `fun` $\overline{v} \Rightarrow \square$ to be elaborated later.

Figure 4.13 shows Sisyphus' rules for proof skeleton reconstruction. It is implemented as a syntax-based translation $\rightsquigarrow_{\text{x}}$ from OCaml programs to sequences of Coq tactics. It takes the verification goal (*i.e.*, pre-/postcondition of the residual program to be traversed) and emits a list of *residual obligations* $\overline{\varphi}$ that should be discharged separately. For example, the obligation emitted in the conclusion of xval, typically applied at the end of the proof (*cf.* Figure 4.3), is the heap entailment $P \vdash Q\ v$, which is derived immediately from the residual goal $\{P\}\ \{Q\}$. Both rules xapp and xapp-hof deal with function applications (for first-order and higher-order functions respectively), using the function dictionary $\mathcal{E}$ to retrive function specifications and adding their side conditions to the list of residual obligations. In addition to that, xapp-hof generates a "blank" invariant `fun` $\overline{q} \Rightarrow \square$, whose hole $\square$ is going to be filled up later with candidates. The next step in Sisyphus' repair process is to mine the building blocks for constructing such candidates from the old program's proofs using the *program alignment* technique.

4.2.2 Computing Program Alignment

Program alignment is the mapping from program points in the new program to those in the old one that exhibit *similar* program states in their executions—a crucial component in determining possible invariant candidates to fill holes in the reconstructed proof script. Note that alignment is a many-to-one correspondence: many new program points can map to one old program point (but not vice versa). Algorithm 4.2.1 shows the main step of computing the alignment. To measure similarity of two program states, it uses a simple `Score` function that only takes into the account the sizes of the aggregate values (*e.g.*, arrays and lists) present in both program states. Prior to computing the score, the `Normalize` function transforms aggregate data types with known canonical projections to a unified representation. For instance, both arrays and sequences are represented as lists; the user can supply their own projections. This enables SISYPHUS to compare program states bearing the same logical values represented in different OCaml datatypes (*e.g.*, `seq` and `array` in `to_array`).

Algorithm 4.2.1: Dynamic Program Alignment

```
Procedure DPA ($p, p'$)
    Input: Old program $p$, and a new program $p'$
    Output: Map $\alpha$ from program points in $p'$ to $p$
    $R, R' \leftarrow$
     ProgramPoints ($p$), ProgramPoints ($p'$)
    for $input$ in GenRandInputs($p$) do
        $tr, tr' \leftarrow$ Trace ($p, input$), Trace ($p', input$)
        for $\langle \rho, \rho' \rangle$ in $R \times R'$ do
            $ps \leftarrow$ ProgramStateAt($tr, \rho$)
            $ps' \leftarrow$ ProgramStateAt($tr', \rho'$)
            $scores[\rho'][\rho] \leftarrow$ Score($ps, ps'$)
    for $\rho'$ in $R'$ do
        $\alpha[\rho'] \leftarrow$ HighestScore($scores[\rho']$)
    return $\alpha$
```

```
Procedure Score ($ps, ps'$)
    Input: Old and new program state $ps, ps'$
    Output: Integer score
    $H, H' \leftarrow$ Heap ($ps$), Heap ($ps'$)
    $S, S' \leftarrow$ Stack ($ps$), Stack ($ps'$)
    $vals \leftarrow$ Normalize($H \cup S$)
    $vals' \leftarrow$ Normalize($H' \cup S'$)
    $score \leftarrow 0$
    for $val$ in $vals$ do
        if $val \in vals'$ then
            $score \leftarrow score +$ Size ($val$)
    return $score$
```

To compute the alignment α, SISYPHUS generates a series of random inputs and simultaneously executes both old and new versions of the program on them to obtain execution traces tr and tr' correspondingly. During the execution, it records the program state at each program point, computes the similarity to program points in the alternative version via the `Score` function, and finally associates each program point in the new program with the most similar program point in the old program.

4.2.3 Synthesising Invariant Candidates

Algorithm 4.2.2: Synthesis of invariant candidates

```
Procedure MakeTemplate ($\rho', vs$)
    Input: New program point $\rho'$, invariant parameters $vs$
    Output: Invariant template $I_\square$
    $I_\square \leftarrow$ emp
    for $v \mapsto \Pi\, \overline{e_i}$ in Heaplets ($\rho'$) do
        $I_\square \leftarrow I_\square * (v \mapsto \Pi\, \_)$
    for $v$ in $vs$ do
        $I_\square \leftarrow I_\square \wedge (v = \_)$
    return $I_\square$
```

```
Procedure SynthesizeCandidates ($I_\square, \alpha, \mathcal{P}, p', \rho'$)
    Input: Invariant template $I_\square$, DPA $\alpha$, old proof $\mathcal{P}$,
    new program $p'$, new program point $\rho'$
    Output: List of invariant candidates matching $I_\square$
    $I_{old} \leftarrow$ GetInvariant($\alpha(\rho')$)
    $sketches \leftarrow$ GetExpressionSketches($I_{old}$)
    $atoms \leftarrow$ CollectConsts($\mathcal{P}$) $\cup$ CollectFuns($\mathcal{P}$)

    $logvars \leftarrow$ LogicalVars(specification of $\mathcal{P}$)
    return
     EnumSynthesis($sketches, atoms \cup logvars, I_\square$)
```

Algorithm 4.2.2 provides an overview of the algorithm for template-based invariant synthesis. In this approach, invariants are encoded as Coq functions that take one or more parameters and construct a logical proposition that constrains the symbolic SL state (*e.g.*, the invariant (4.5)).

Recall that, as detailed in Subsection 4.2.1, SISYPHUS generates a “blank” invariant for every application of a higher-order function. Initially, this “blank” invariant consists purely of a single empty hole. To aid the synthesis, in its first step (Algorithm 4.2.2, left), the algorithm constructs an *invariant template* $I_\square$ for a program point ρ' in the new program. It does this by collecting all heaplets in the program's symbolic state, obtained by symbolic execution, immediately before ρ'. Each heaplet has the shape $v \mapsto \Pi\ \overline{e_i}$, where v is a logical or program-level variable, and Π is an OCaml data type constructor (*e.g.*, `Array` or `ref`), possibly applied to arguments $\overline{e_i}$ that are being replaced by a hole _. The template heaplets are then conjoined using the SL $*$ connective. Additionally, for every parameter to the invariant (obtained from the type of the corresponding lemma that requires it), the tool generates equality propositions of the form $v = \_$. For example, the template that was elaborated into the form (4.7) is

```
fun acc t ⇒(a ↦ Array _) ∧ (acc = _) ∧ (t = _)
```
(4.8)

After inferring the template $I_\square$ of the desired invariant, Sisyphus constructs concrete candidates by filling it (Algorithm 4.2.2, right). To do so, the algorithm first retrieves the corresponding invariant from the aligned location in the old program and uses it to construct a set of sketches (*i.e.*, expressions with holes). It then collects constants and function symbols from the old proof, as well as logical variables from the ascribed specifications and uses those to fill the holes in the sketches, themselves used to instantiate holes in the template, thus completing the enumerative synthesis of candidates.

▶ 4.3 Proof-Driven Invariant Testing

This section will describe the technique at the heart of Sisyphus, *proof-driven testing*, a means to test the validity of invariant candidates. This goal is achieved by automatically generating tests to check computable properties of program values and states, from proofs of higher-order lemmas.

To build an intuition for this process, it would behoove the reader to start an exploration by considering a simple *computable* property P on natural numbers, defined as $P\ n \triangleq 1 + n = n + 1$. A standard way to prove that P holds on *all* natural numbers is by induction, *i.e.*, by providing proofs of the facts $H_0 : P\ 0$ (*i.e.*, the induction base) and $H_i : \forall i, P\ i \to P\ (i+1)$ (*i.e.*, the induction transition).

H_0 and H_i are *proof terms*: following their types, one can compose them into the expression

$$H_i\ 2\ (H_i\ 1\ (H_i\ 0\ H_0)) : P\ 3 \tag{4.9}$$

whose type will be $P\ 3$, therefore making it a proof term for the fact that the property P holds on the number 3. The proof of $P\ 3$ is, therefore, constructed by applying H_i, an inductive step, to two

$$\begin{array}{lll} v & & \text{variable names} \\ t & ::= & v \mid t\,t' \mid \forall x : t, t' \mid \texttt{fun}\ x : t \Rightarrow t' \\ & & \mid\ C\ \overline{t} \mid \texttt{match}\ t\ \texttt{with}\ \overline{C_i\ \overline{v_i} \Rightarrow t_i} \\ & & \mid\ \texttt{fix}\ f\ \overline{v_i : t_i} : t \Rightarrow t' \end{array}$$

Figure 4.14: Syntax of selected CIC$_\omega$ terms

arguments: a concrete value 2, and a sub-term constructed to have the type $P\ 2$. This sub-term proving $P\ 2$ recursively is also constructed by applying H_i again, but this time to 1 and $(H_i\ 0\ H_0)$. The remarkable observation supported by this example is that this proof, *i.e.*, that P holds on the value 3, *contains* within it the information that P must necessarily also hold on the values 2, 1, and 0—from the fact that within its construction, it contains sub-terms proving $P\ i$ for $i \in \{0, 1, 2\}$.

What does this have to do with testing invariants? Imagine that one *doesn't know* if P holds for all numbers. A natural (pun intended) thing to do in this case would be to first *test* that P holds on *some* numbers, *e.g.*, 0, 1, 2, *etc.*, up to some large number (*e.g.*, 3). On the other hand, *assuming* that P can be proven by induction, it would be known that there *must* be witnesses (*i.e.*, proof terms) H_0 and H_i for statements $P\ 0$ and $\forall i, P\ i \to P\ (i+1)$, correspondingly, such that for any concrete n, $P\ n$ can be derived by repeated application of H_i to smaller numbers and H_0, as demonstrated by (4.9). By combining these two observations, it leads to the main revelation of this work: For any *concrete* n, the proof term t_n for the proposition $P\ n$ *contains* sub-terms of types $P\ i$ for $i < n$ within itself; by traversing t_n, one can identify each of those types $P\ i$ and convert it into a *dynamically checkable assertion* over a boolean expression $P(i)$, which *must not fail* if P holds universally.

As it turns out, the same observation, *i.e.*, that a proof-term for a higher-order lemma about a property can be used for extracting tests for this property, is not just specific to statements about natural numbers! In fact, exactly this very principle can be applied in anger to test the properties of *program executions*, *e.g.*, when P is a guessed *invariant over program states* parameterised by program and logical variables. In these cases, concrete values of both program and logical variables can be effectively supplied by subterms of the corresponding lemma proofs that are partially evaluated on concrete inputs.

4.3.1 Instantiating Proof Terms with Concrete Inputs

For the rest of this section, for simplicity, the narrative will restrict its language to a subset of CIC$_\omega$, the calculus of Coq, its syntax adopted from Timany and Jacobs [128] and given in Figure 4.14.

The first step in this proposed testing process is to instantiate higher-order lemmas with concrete arguments and reduce their proof terms to a head-normal form using rules from Figure 4.15. Once

$$\textsc{red-app}\quad (\texttt{fun}\,x : T, t)\; t' \Downarrow_{\mathrm{r}} t[t'/x]$$

$$\textsc{red-constr}\quad \frac{t_i \Downarrow_{\mathrm{r}} t'_i}{C\,\overline{t_i} \Downarrow_{\mathrm{r}} C\,\overline{t'_i}}$$

$$\textsc{red-match}\quad (\texttt{match}\,C_k\,\overline{t_j}\;\texttt{with}\;\overline{C_i\,\overline{x_j} \to t'_i}) \Downarrow_{\mathrm{r}} t'_k[\overline{t_j}/\overline{x_j}]$$

$$\textsc{red-fun}\quad \frac{t \Downarrow_{\mathrm{r}} t'}{(\texttt{fun}\,x : T \Rightarrow t) \Downarrow_{\mathrm{r}} (\texttt{fun}\,x : T \Rightarrow t')}$$

$$\textsc{red-fix}\quad \frac{t'_0 = C\,\overline{t'} \qquad F = (\texttt{fix}\,f\,\overline{x_i : T_i}\; \Rightarrow t)}{F\,\overline{t'_i} \Downarrow_{\mathrm{r}} t[F/f, \overline{t'_i}/\overline{x_i}]}$$

Figure 4.15: Selected reduction rules for CIC$_\omega$

an expression has been reduced to its head-normal form, the types of its non-reducible subterms will contain the property of interest applied to concrete values, allowing for further test extraction.

As an example, consider instantiating and reducing the induction principle `nat_ind` for natural numbers. The induction principle is implemented as a higher-order lemma whose type is

$$\forall (P : \texttt{nat} \to \texttt{Prop}), P\;0 \to (\forall i, P\;i \to P\;(i+1)) \to \forall (x : \texttt{nat}), P\;x \tag{4.10}$$

and whose proof term is a recursive function that pattern-matches over two constructors of `nat`:

$$\texttt{fun}\;P\;H_0\;H_i \Rightarrow \texttt{fix}\;F\;(n : \texttt{nat}) : P\;n \Rightarrow\; \texttt{match}\;n\;\texttt{with}\;\;|\;0 \Rightarrow H_0\;\;|\;S\;n' \Rightarrow H_i\;n'\;(F\;n') \tag{4.11}$$

According to its (dependent) type (4.10), `nat_ind` takes four parameters, of which the first three are either properties or their proofs (which is indicated by their types), while the fourth one has type `nat`. In the scenario of interest for this work, lemmas are applied to concrete (*i.e.*, non-proposition) arguments, as those correspond to values that can be used for testing the validity of propositions (in this case, passed as arguments to P). Therefore, to formally replicate the example (4.9), `nat_ind` should be instantiated with three *symbolic* values P, H_0, and H_i indicating proof terms or properties whose values are irrelevant for test generation (but whose types are important), and the fourth argument being the *concrete* value 3. By repeatedly applying the rules RED-APP, RED-FIX, and RED-MATCH to this expression, it can be reduced to the following partially-evaluated form:

$$\texttt{nat\_ind}\;P\;H_0\;H_i\;3 \quad \Downarrow^*_{\mathrm{r}} \quad H_i\;2\;(H_i\;1\;(H_i\;0\;H_0)) \tag{4.12}$$

The reduced form on the right reveals the familiar sub-terms whose types contain applications of P to concrete values. The next section will use this result to extract a testable specification for P.

$$\textsc{vanilla-app}\quad \frac{\Gamma; t \rightsquigarrow_{\mathrm{v}} c \qquad \Gamma; t' \rightsquigarrow_{\mathrm{v}} c'}{\Gamma; t\, t' \rightsquigarrow_{\mathrm{v}} c\, c'} \qquad \textsc{vanilla-ind}\quad \frac{C\,\overline{t_i} : \tau \qquad \Gamma; t_i \rightsquigarrow_{\mathrm{v}} c_i}{\Gamma; C\,\overline{t_i} \rightsquigarrow_{\mathrm{v}} C\,\overline{c_i}} \qquad \textsc{vanilla-fun}\quad \frac{x, \Gamma; t \rightsquigarrow_{\mathrm{v}} c}{(\texttt{fun}\ x : T \Rightarrow t) \rightsquigarrow_{\mathrm{v}} (\texttt{fun}\ x \Rightarrow c)}$$

$$\textsc{vanilla-match}\quad \frac{t \rightsquigarrow_{\mathrm{v}} c \qquad \overline{x_i}, \Gamma; t_i' \rightsquigarrow_{\mathrm{v}} c_i}{\begin{array}{c}\Gamma; \texttt{match}\ t\ \texttt{with}\ \overline{C_i\ \overline{x_i : T_i} \to t_i'} \rightsquigarrow_{\mathrm{v}} \\ \texttt{match}\ c\ \texttt{with}\ \overline{C_i\ \overline{x_i} \to c_i}\end{array}} \qquad \textsc{vanilla-erase}\quad \frac{t : \tau \qquad \tau : \texttt{Prop}}{t \rightsquigarrow_{\mathrm{v}} ()} \qquad \textsc{vanilla-fix}\quad \frac{f, \overline{x_i}, \Gamma; t \rightsquigarrow_{\mathrm{v}} c \qquad \Gamma; t_i' \rightsquigarrow_{\mathrm{v}} c_i'}{\begin{array}{c}\Gamma; (\texttt{fix}\ f\ \overline{x_i : T_i} \Rightarrow t)\ \overline{t_i'} \rightsquigarrow_{\mathrm{v}} \\ \texttt{let rec}\ f\ \overline{x_i}\ =\ c\ \texttt{in}\ f\ \overline{c_i'}\end{array}}$$

$$\textsc{extract-prop}\quad \frac{\begin{array}{c}\Gamma \vdash t : \tau \\ \tau = t_1 \ldots t_n \to \tau' \qquad \tau' : \texttt{Prop}\end{array}}{\Gamma; t \rightsquigarrow_{\mathrm{e}} (\texttt{fun}\ x_1 \ldots x_n \Rightarrow ())} \qquad \textsc{extract-test}\quad \frac{\begin{array}{c}\Gamma \vdash t : \tau \qquad \tau : \texttt{Prop} \\ \Gamma \vdash \tau \rightsquigarrow_{\mathrm{t}} p \qquad \Gamma; t \rightsquigarrow_{\mathrm{e}} c\end{array}}{\Gamma; t \rightsquigarrow_{\mathrm{e}} \texttt{assert}\ p; c} \qquad \textsc{extract-gen}\quad \frac{\Gamma; v\,\overline{t_i} \rightsquigarrow_{\mathrm{f}} c}{\Gamma; v\,\overline{t_i} \rightsquigarrow_{\mathrm{e}} c}$$

Figure 4.16: Vanilla extraction rules from CIC$_\omega$ to OCaml (top) and new rules for test extraction (bottom). Highlighted premises are domain-specific and are instantiated for a particular set of properties.

4.3.2 Extracting Tests from Reduced Proof Terms

The top part of Figure 4.16 presents a simplified form of the extraction relation ($\rightsquigarrow_{\mathrm{v}}$) from Coq to OCaml [77]. The key rule that is relevant for this work is VANILLA-ERASE, which handles the removal of logical parameters for the sake of producing efficient executable OCaml code: when a Coq expression t happens to have type in `Prop`, *i.e.* meaning that it is a computationally-irrelevant proof term, then Coq's vanilla extraction simply returns the unit value `()` without even inspecting t's structure. As the goal is to visit the sub-terms of logical properties to generate tests, this rule must be suitably adapted.

The extraction mechanism (Figure 4.16, bottom) developed in this work, updates the relation $\rightsquigarrow_{\mathrm{v}}$ with three additional new rules: EXTRACT-TEST, EXTRACT-PROP, and EXTRACT-GEN (which will be explained in Subsection 4.3.3). The rule EXTRACT-PROP extends the extraction making it traverse terms with type in `Prop`. The key addition is the rule EXTRACT-TEST, which implements the earlier intuition that sub-terms witnessing a property can encode dynamically checkable assertions for which the property must hold. In particular, whenever the extraction process visits a sub-term inhabiting a type τ that can be converted into an executable test `p` (via a *reflection* step $\rightsquigarrow_{\mathrm{t}}$), it emits an assertion that `p` must hold within the extracted computation. The heavy lifting in this translation is done by the $\rightsquigarrow_{\mathrm{t}}$ reflection rule, which is domain-specific and is instantiated by the user for a particular set of decidable properties.

$$\textsc{nat-refl-eq}\quad \frac{\Gamma; t \rightsquigarrow_{\mathrm{v}} c \qquad \Gamma; t' \rightsquigarrow_{\mathrm{v}} c'}{\Gamma; (t = t') \rightsquigarrow_{\mathrm{t}} c = c'}$$

To wrap up this section, consider again the running example. In this case, one can instantiate $\rightsquigarrow_{\mathrm{t}}$ with

$$\text{REFL-EMP}\quad \frac{}{\Gamma; \texttt{emp} \rightsquigarrow_{\mathrm{t}} \texttt{true}} \qquad \text{REFL-SEP}\quad \frac{\Gamma; t \rightsquigarrow_{\mathrm{t}} c \qquad \Gamma; t' \rightsquigarrow_{\mathrm{t}} c'}{\Gamma; t * t' \rightsquigarrow_{\mathrm{t}} c \,\&\&\, c'} \qquad \text{REFL-PTS}\quad \frac{\Gamma; t \rightsquigarrow_{\mathrm{t}} c \qquad \mathcal{R}(f) = F}{\Gamma; v \mapsto f\ t \rightsquigarrow_{\mathrm{t}} F\ v = c}$$

$$\text{EXTRACT-XAPP}\quad \frac{\Gamma; H_c \rightsquigarrow_{\mathrm{e}} c \qquad \Gamma; P \rightsquigarrow_{\mathrm{t}} p}{\Gamma; \texttt{xapp}\ P\ Q'\ Q\ f\ v\ b\ H_f\ H_c \rightsquigarrow_{\mathrm{f}} \texttt{assert}\ p;\ \texttt{let}\ x\ =\ f\ v\ \texttt{in}\ c\ x}$$

Figure 4.17: A reflection for SL properties (top); An extraction rule for XAPP (bottom)

the bespoke reflection function (above) tailored for the property $P\ n \triangleq 1 + n = n + 1$, and use the extraction rules to convert the normalised proof term (4.12) into the test-specification (4.13).

$$H_i\ 2\ (H_i\ 1\ (H_i\ 0\ H_0)) \ \rightsquigarrow_{\mathrm{t}}^{*}$$
$$\texttt{assert}\ (1 + 3 = 3 + 1); \texttt{assert}\ (1 + 2 = 2 + 1); \texttt{assert}\ (1 + 1 = 1 + 1); \texttt{assert}\ (1 + 0 = 0 + 1) \tag{4.13}$$

Notice that the four asserts in the OCaml program above correspond to the subterms $(H_i\ 2\ \ldots)$, $(H_i\ 1\ \ldots)$, $(H_i\ 0\ H_0)$, and H_0, as each one inhabits the type of a concrete instantiation of P.

4.3.3 Testing Properties in Separation Logic

Leaving the world of properties over `nat`, it is now time to tackle the original goal of this section, using *proof-driven testing* to efficiently test separation logic properties, such as invariants. In particular, this section will consider an embedding of CFML in Coq, and present how the previously introduced custom extraction rules for Coq (*cf.* Figure 4.17) can be extended to be used in this domain.

Consider a prototypical CFML rule XAPP for function applications, as seen in Equation 4.1.1. When embedded as a lemma (`xapp`) within Coq, XAPP happens to take the following dependent type:

$$\begin{gathered}\forall (P : \texttt{hprop})\ Q'\ Q\ f\ v\ b,\quad (\{P\}\ (f\ v)\ \exists x, \{Q'\ x\}) \to (\forall x, \{Q'\ x\}\ \ b\ x\ \ \{Q\}) \to \\ \{P\}\ \ (\texttt{let}\ x\ =\ f\ v\ \texttt{in}\ b\ x)\ \{Q\}\end{gathered} \tag{4.14}$$

That is, the `xapp` lemma takes eight arguments, where the first three are properties over the symbolic heap (with type `hprop`), the next three take values, and the last two take SL proofs. Notice that, from the type of `xapp`, for the heap properties that are passed as input (*i.e.*, P, Q', and Q) no witness is passed to the lemma. Rather, the lemma expects proofs of Hoare triples such as $\{P\} \ldots \exists x, \{Q'\ x\}$. This section will not show the proof term of `xapp` here, but it does not construct proof terms inhabiting P, Q, *etc.*, either; instead, it composes the witnesses for the lemma's argument Hoare triples. As such, if one were to apply the extraction rules from *cf.* Figure 4.17 to a concrete instantiation of `xapp`, the

Algorithm 4.3.1: Invariant testing

```
Procedure TestInvariant ($p', f, t_f, I$)
    Input: Program $p'$, HOF $f$, proof term for $f$ $t_f$ : $args \to I \to$ Spec, invariant $I$
    Output: Passes if no assertion is violated
    $st, args \leftarrow$ RunUpto($p', f,$ GenInput($p$))
    $t \leftarrow$ Instantiate($t_f, args, I$)
    Reduce($t \Downarrow_r t'$)
    Extract($t' \rightsquigarrow_e \delta$)
    RunFromState($st, \delta\ ()$)
```

resulting program would *not* include any explicit assertions about properties over the heap.

The goal is to extend the test extraction to test SL properties that hold over the heap. To do this, one can first instantiate the reflection relation $\rightsquigarrow_t$ with the rules REFL-EMP, REFL-SEP, and REFL-PTS (*cf.* Figure 4.17, top) to handle terms in `hprop`, CFML's encoding of heap properties. The key rule in this encoding is REFL-PTS that uses a mapping $\mathcal{R}$ to extract heap predicates (*e.g.*, `Array`) to corresponding OCaml functions that manipulate their logical contents (*e.g.*, `of_list`). Applying these rules, it is possible to reflect logical assertions over the heap into executable OCaml tests similar to the program in Figure 4.10.

To assert these SL properties within SISYPHUS' tests, the extraction procedure must be tuned such that the test programs appropriately update and maintain the heap, testing the predicates derived from the passed heap assertions at the correct respective program points. For this purpose, one can turn to the EXTRACT-GEN rule (*cf.* Figure 4.16), and instantiate it for CFML, extending $\mathcal{F}$ to map CFML's reasoning rules to transformations that capture their semantics. For example, EXTRACT-XAPP (*cf.* Figure 4.17, bottom) presents the corresponding instantiation for XAPP. The essential part of the rule is the bespoke invocation of the $\rightsquigarrow_t$ procedure that converts the precondition P to the assertion **`assert`** `p` installed right before the call to `f`. The rule EXTRACT-XAPP does not convert the other two argument properties of `xapp`, Q' and Q, which both take value arguments, into assertions. This extension is straightforward but would require to generalise $\rightsquigarrow_t$ to handle parameterised properties; in the interest of time this was not implemented in this version of SISYPHUS. The encodings for the remaining rules of CFML follow the same strategy and are not presented here, but can be found in the implementation.

Putting it all together, Algorithm 4.3.1 describes the overall process whereby SISYPHUS uses proof-driven testing to prune candidate invariants when instantiating holes in the generated proof skeleton for a given program. When testing an invariant candidate I for a higher-order function f, whose application occurs in the code of the new program p', `TestInvariant` first executes the enclosing program p' on a randomly generated input and observes both the program state st at the call site and the concrete arguments to f. The concrete arguments and the invariant are passed to t_f to construct an instantiated reduced proof term t, which is extracted into a OCaml test program δ, which is executed in the context st. If δ raises an exception during its execution, then I is invalidated and can be pruned away.

4.4 Implementation and Evaluation

Sisyphus has been implemented in 19k lines of OCaml. The table on the right summarises the distribution of implementation effort, in terms of the approximate lines of code of each component of the development.[5] In order to implement proof-driven testing, the project includes a lightly modified version of Coq's reduction algorithm to allow reduction within proof terms. Note that the CFML instantiation of proof-driven testing only requires around 1600 specific LOC, which is likely a good estimate of the additional effort that one might expect in order to extend Sisyphus to handle proof repair in other Coq embeddings of Separation Logic. In order to dispatch any obligations generated during the repair process, this work implemented a domain-specific solver as a small collection of Ltac-based tactics (∼700 LOC). An initial implementation of the tool used Z3 [34] for this purpose; however, it was found that it was not effective at reasoning about the generated obligations, taking several minutes on even simple goals.

Component	LOC
Proof-skeleton generator (§4.2.1)	2700
Program alignment (§4.2.2)	1400
Enumerative synthesis (§4.2.3)	1600
Modified Coq reduction (§4.3.1)	7600
Proof-driven test extraction (§4.3.2)	2000
Reflection & extraction for CFML (§4.3.3)	1600
Miscellanea (*e.g.*, logging, stats, etc)	1900
Total	18800

This work conducts an empirical evaluation of Sisyphus to answer the following research questions:

- **RQ1**: Is Sisyphus effective at repairing proofs for real world programs: can it find correct invariants and how much manual effort is required to complete the proof?
- **RQ2**: How efficient is Sisyphus: does it generate invariants in a reasonable amount of time?
- **RQ3**: What are the classes of changes in programs that Sisyphus handles poorly or not at all?

Benchmarks In order to answer these questions, a benchmark suite of 14 evolved OCaml programs was collected for the evaluation, with the details as summarised in Table 4.2. Of these programs, a majority, 10, were drawn from real-world code-bases, found by mining the version control of popular OCaml libraries (*e.g.*, Jane Street's `base`, `containers`, *etc.*—*cf.* the accompanying artefact for their exact provenance), with the remaining ones constructed by the author.

Programs were selected to exhibit the use of a diverse range of refactorings. In particular, the changes

[5]The accompanying artefacts, Sisyphus implementation and benchmarks, are available online [50].

Table 4.2: Categorisation of benchmark programs by the type of change and language features and data structures used (left); Comparison of additional effort required to dispatch the obligations in proof scripts produced by SISYPHUS (right). † indicates when the new version of the program was constructed by the authors, and ‡ indicates when both versions of the programs were constructed by the authors.

Example	Data Structure	Refactoring	Time (old)			Time (new)	# Admits / # Obligations
			Spec	Proof	Total		
seq_to_array	Array, Seq	IterOrd, DataStr	1hrs	1hr	2hrs	17m	3/5
make_rev_list†	Ref	Mutable/Pure	5m	5m	10m	-	0/2
tree_to_array†	Array, Tree	IterOrd, DataStr	4hrs	1hr	5hrs	18m	2/4
array_exists	Array	Mutable/Pure	10m	20m	30m	12m	2/4
array_find_mapi	Array, Ref	Pure/Mutable	30m	1hr	1.5hrs	12m	2/5
array_is_sorted	Array	Pure/Mutable	1hr	3hrs	4hrs	2m	2/5
array_findi	Array	Pure/Mutable	30m	1hr	1.5hrs	9m	3/7
array_of_rev_list	Array	DataStr	5m	1hr	1hr	3m	2/3
array_foldi	Array	Pure/Mutable	10m	5m	15m	-	0/1
array_partition	Array	DataStr	30m	2hrs	2.5hrs	5m	3/3
stack_filter‡	Stack	DataStr	1hr	30m	1.5hrs	11m	3/3
stack_reverse‡	Stack	DataStr	1.5hrs	30m	2hrs	30s	1/1
sll_partition‡	SLL	Mutable/Pure, IterOrd	1hr	1hr	2hrs	-	0/2
sll_of_array‡	Array, SLL	IterOrd	1.5hrs	30m	2hrs	-	0/1

of programs considered in the benchmarks can be classified into four classes: (1) IterOrd, for changes in iteration order (*cf.* the change in `to_array`), (2) DataStr, for changes in intermediate data structures, (3) Mutable/Pure for the transformation of programs using loops with mutation to pure variants, and (4) Pure/Mutable for the converse. The benchmarks being considered manipulate a representative range of common OCaml data-structures: arrays, lazy sequences, mutable singly-linked-lists (SLL), stacks, queues and trees. The main roadblock in tackling larger classes of data structures for the case studies was in the lack of available verified implementations of these data structures. For example, though the CFML development provides a simplified version of the OCaml Map data structure, implemented using a balanced binary tree, it does not verify most of its associated functions.

Methodology The methodology used to evaluate SISYPHUS on these benchmark programs was as follows: For each library function in the benchmark suite, first, the function and its dependencies were extracted into a standalone file, manually replacing the use of **`for`**- and **`while`**-loops with suitable higher-order loop combinators (*cf.* Subsection 4.2.1). Then, in order to construct the initial proofs for each program, the author manually verified the implementations of each of the benchmark programs and recorded the time taken. SISYPHUS was then invoked to construct proofs for the new versions of each benchmark program. Finally, any residual obligations that were left as admits by the tool were proven manually by a co-author of this work and timed. Both the author and the co-author were equally familiar with Coq/CFML before writing any proofs, and the primary purpose of this experiment was to evaluate the manual effort in proving the remaining obligations in relation to the initial proof effort.

Table 4.3: Statistics of Sisyphus when repairing proofs of verified OCaml programs. The first 3 columns list the total invariant candidates that were generated and their breakdown by heap and pure parts, to be tested independently. The last 5 columns describe the time taken by Sisyphus, with the breakdown per individual component: generation of candidates, extraction of tests, running the tests, and the remaining tasks, which include computing program alignment and interaction with the Coq runtime.

Example	Candidates			Time (s)				Total (s)
	Heap	Pure	Total	Generation	Extraction	Testing	Remaining	
seq_to_array	3.4×10^7	1.8×10^4	6.2×10^{11}	28.57	1.95	20.36	5.28	58
make_rev_list	-	30	30	$\leq 10ms$	3.36	$\leq 10ms$	11.95	15
tree_to_array	5.0×10^6	8.2×10^3	4.0×10^{10}	6.75	1.95	2.98	13.32	25
array_exists	-	25	25	$\leq 10ms$	3.30	$\leq 10ms$	13.23	17
array_find_mapi	13	34	442	$\leq 10ms$	2.13	$\leq 10ms$	13.95	17
array_is_sorted	64	70	4.5×10^3	$\leq 10ms$	2.04	$\leq 10ms$	15.38	18
array_findi	4.9×10^3	34	1.7×10^5	$\leq 10ms$	2.13	$\leq 10ms$	19.07	22
array_of_rev_list	1.5×10^6	-	1.5×10^6	1.72	2.82	0.96	15.62	21
array_foldi	24	-	24	$\leq 10ms$	488.89	$\leq 10ms$	15.00	504
array_partition	1.6×10^6	-	1.6×10^6	3.51	69.73	2.62	17.53	95
stack_filter	71	-	71	$\leq 10ms$	81.88	$\leq 10ms$	21.53	104
stack_reverse	22	-	22	$\leq 10ms$	88.42	$\leq 10ms$	16.94	105
sll_partition	630	-	630	$\leq 10ms$	426.62	$\leq 10ms$	16.43	443
sll_of_array	2.4×10^4	-	2.4×10^4	0.02	55.98	0.01	13.33	69

RQ1: Effectiveness and Utility The experiments demonstrate that Sisyphus is effective at repairing the proofs of real world programs. Sisyphus was able to automatically construct new proofs for all programs in the benchmarks: *all* synthesised invariants were valid, and the author was able to manually prove any remaining proof obligations. The fourth to the sixth columns of Table 4.2 describe the comparison of the manual effort, in terms of the time taken, to prove and specify the original programs in comparison to using Sisyphus and manually dispatching remaining obligations. The last column of the table (# Admits / # Obligations) describes the number of verification conditions that Sisyphus was unable to dispatch automatically and were left as admits for the user to prove. It was possible to construct Coq proofs for all such remaining sub-goals manually.

As can be seen from the table, all obligations took fewer than 20 minutes to dispatch, while specifying and proving the original programs took considerably longer, demonstrating the utility of Sisyphus for maintaining verified code-bases. It was found that most of the challenge in completing proofs was in reasoning about properties of the involved functions that were not considered in the original development but relied on by the generated invariants. For example, in the case study for `array_partition`, the generated invariant made use of an expression of the form $\texttt{filter}\ p\ (\texttt{filter}\ p\ \ell)$. Since filtering is idempotent, such repetition is redundant, but no such property had been proven before in the development, so dispatching the remaining obligation required the developer to prove it manually.

RQ2: Efficiency The experiments were conducted on a commodity laptop (3.5 GHz Apple M2 MacBook Air with 8GB RAM). Overall, Sisyphus was found to be reasonably efficient at repairing proofs, taking fewer than 2 minutes to execute on almost all examples in the benchmarks.

In the two cases where Sisyphus takes longer than 2 minutes, `array_foldi` and `sll_partition`, most of the time is spent on performing the extraction of tests itself, rather than generating and pruning candidates. It should be noted that the implementation of proof-driven testing in this work has not been particularly optimised, simply reusing Coq's infrastructure to implement extraction, so further improvements could be obtained by adopting a more specialised implementation. The second to fourth columns of Table 4.3 list the number of generated invariant candidates; Sisyphus uses a lazy generation strategy, so the table only records the number of candidates until Sisyphus finds a suitable invariant or gives up. Note that for many of the benchmarks, Sisyphus is able to use program alignment to considerably reduce the space of candidates, often leaving fewer than 100 candidates. Furthermore, whenever possible Sisyphus conducts testing of pure and heap-related parts of invariant candidates independently, rather than by taking their product, thus contributing to the efficiency of the search.

RQ3: Failure Modes An important assumption of Sisyphus's repair process is that components of the old proof, such as the lemmas and functions that it uses, will be sufficient to prove the new program correct. As has been mentioned, this is not always the case. Recall that in the case of `array_partition`, Sisyphus was unable to fully automatically repair the proof, as the new proof required a lemma about repeated filters that had not been present in the old proof. In cases where the new program requires use of a function not present in the old proof, Sisyphus's search space will not even contain an invariant that can pass proof-driven testing, and the repair process will fail entirely.

```
let batches = (* .. *) in
let res =
   Array.make (* .. *) in
List.iter (fun batch ->
  let dst = (* .. *) in
  Array.copy batch res dst)
  batches
```

Figure 4.18: Batched implementation of `to_array`

For example, consider another version of `to_array` as presented in Figure 4.18 that uses a batching strategy: instead of collecting the elements of the sequence into a list, it accumulates a list of batches of elements and then separately inserts each batch into the result array as on the right. Sisyphus will fail to repair this program as its invariants will require an operation to reason about flattening lists of lists; however, the old proof for `to_array` does have any functions that can capture this operation.

4.5 Related Work

The work presented in this chapter touches upon the three actively developed research themes: SL-based deductive verification of heap-manipulating programs, invariant inference, and proof repair. Below, this section outlines the connections to the most closely related work in those three directions.

Proof repair The work by Ringer [113] focuses on proof repair in two particular scenarios: (i) synthesis-by-example of patches to proofs of theorems whose statements were changed to use new data types [115] and (ii) equivalence-preserving changes in data types used by a verified functional program and its specification [114, 116]. These approaches do not address arbitrary local changes in the code of a verified program, nor do they consider foundational verification of imperative programs.

Techniques from non-dependently typed provers. This work draws some parallels with earlier research in non-dependently typed proof assistants by Matichuk [88] on automatic extraction of function annotations for programs in Isabelle/HOL. Matichuk describes a technique that operates on proofs about stateful programs in monadic Hoare logic to extract intermediate state assertions into a standalone set of function annotations, that can then be re-used to verify other inter-dependent properties. While this approach has similarities to the proof-driven-testing algorithm of this work, there are some constraints arising from the non-constructive setting that limit the generality of this technique.

In particular, Matichuk's approach requires proofs to use a particular monadic Hoare logic implementation that has been adjusted to collect intermediate assertions, while proof-driven testing allows extracting tests from a larger class of theorems in pre-existing logics and is able to obtain this information for free by simply introspecting the proof terms "as is". Furthermore, the machinery in the 2012 paper only considers first-order imperative programs, while proof-driven testing's mechanism enables inferring invariants for higher-order programs with combinators. That said, *assuming* that Matichuk's approach could be extended to proofs about higher-order combinators such that it stored annotations describing how those proofs instantiate the invariants (as in Figure 4.11), it seems likely that such annotations could be used to generate tests similar to the one in Figure 4.10.

KeYTestGen [5] and StaDy [103] take advantage of annotations required by correctness proofs to generate tests with a high degree of coverage. By taking annotated Java and C programs respectively, these tools generate unit tests that exercise the programs' behaviour with a high degree of coverage. On a conceptual level, in contrast with KeYTestGen and StaDy, the approach in this work is inherently *higher-order*, as it extract tests for invariants (*i.e.*, *properties* of programs) from proofs of lemmas that use those invariants, while the mentioned tools target testing of programs themselves.

Invariant inference. Inference of loop invariants for imperative programs using static [45, 61, 109] and dynamic [42, 43] analysis, as well as machine learning techniques [121] is a well-studied research topic.

Magill et al. [85] were amongst the first to describe a heuristic procedure for inferring SL invariants via static analysis relying on predicate abstraction [33, 55] for programs manipulating linked lists. In contrast with Magill et al.'s work, this approach is based on dynamic analysis and does not require a predefined set of predicates, neither heap- nor pure ones, as those are drawn from the old proofs.

The data-driven tools Locust [21] and SLING [74] use dynamic analysis to derive SL formulas describing the shape of a state manipulated by a C program as well as the program's loop invariants. Both of these tools target proofs of memory safety and do not consider correctness *w.r.t.* arbitrary specifications, limiting the language of pure assertions to the first-order logic with arithmetic comparisons. For the validation of inferred invariants, Locust relies on the automated non-foundational verification tool GRASShopper [104], while the authors of SLING report poor experience with SMT solvers and had to resort to manual (*i.e.*, non machine-assisted) invariant checking (*cf.* Sec. 5.3 of Le et al. [74]).

Of the state-of-the-art deductive verification tools, only Why3 [44] allows for any form of invariant inference, in this case over limited class of numeric abstract domains, and does not support arbitrary effectful functions or statements about shape properties of data (*e.g.*, constraining contents of an array). At the time of writing, the author is not aware of any approach comparable to Sisyphus in its ability to automatically infer complex invariants which constrain both heap and data for higher-order imperative programs, and then use these invariants to reconstruct proofs for modified programs.

Automated foundational proofs in Separation Logic. Foundational approaches to program verification encode the meta-theory and the rules of a domain-specific logic (*e.g.*, a version of SL) in terms of the logic of the host verifier (*e.g.*, Coq). This work considered libraries written in a higher-order imperative language and verified in a foundational encoding of SL into Coq, which enabled proof reconstruction and efficient pruning of invariant candidates as described in Section 4.2 and 4.3. The contributions of this work are complementary to efforts in automated foundational verification of sequential [48, 119] and concurrent [91] heap-manipulating programs, as those tools require explicit invariants.

Finally, even though the current implementation of Sisyphus only supports a particular Coq-embedded SL, namely, CFML [25], it seems reasonable to expect that the approach in this work would be applicable to many other foundational SL implementations: HTT [95], Bedrock [29], VST [7], CHL [27], FCSL [120], and the large family of logics based on the Iris framework [67]. As explained in Subsection 4.3.3, to support a custom SL embedding, one would have to elaborate test extraction by implementing reflection procedures for embedding-specific encodings of SL assertions and rules.

▶ 4.6 Takeaways & Main Insights

This chapter has presented the final case study of this thesis, an investigation into the use of repair for verified software evolution through the design and implementation of Sisyphus, a partially-automated tool to repair proofs of verified OCaml programs over changes. The chapter began by motivating the problem of proof repair, illustrating how developers will often need optimise or update verified software while preserving the existing specifications in real world code. The chapter then introduced Sisyphus, the main contribution of this chaptre, as a tool developed to address this problem, and presented a high level overview its repair process, before detailing the technical details of its implementation. Finally, the chapter presented the evaluation of the tool on a benchmark suite of real world OCaml programs and demonstrated how it is able to effectively reduce the human effort involved in software maintenance.

Ultimately, the results from this case-study demonstrate the efficacy of repair-based methodologies for verified software maintenance — in this case, for handling changes in the implementation of certified programs. In particular, the experimental results show how the combination of repair techniques used in Sisyphus makes the tool particularly effective at constraining the search space of repairs, and allows it to produce repaired proofs in only a few minutes for most cases. Furthermore, while the generated proofs are sometimes incomplete, the remaining proof obligations are typically of a reasonable difficulty and pose significantly less challenge than writing the initial proof. While the implementation described in this chapter only supports the CFML embedding of SL, the techniques themselves have are generally parametric over the particular choice of logic, so it seems reasonable to expect that they can be generalised to repairing proofs of programs in other logics and programming languages.

CHAPTER

5

A UNIFIED FRAMEWORK FOR VERIFIED SOFTWARE EVOLUTION

Combining the findings from the previous case studies, this chapter presents a general unified framework for managing the evolution of verified software and demonstrates its use through a hypothetical example – the maintenance of a verified static web page server. The chapter first introduces the overall framework and reviews how each of the prior techniques, that of composition, synthesis and repair, can each be used to tackle different aspects of verified software evolution. The chapter then describes the verified software system used as an example, and details the high-level design and implementation of the web server. The remaining sections in the chapter each consider a different way in which this verified system experiences change and describes how the components of this framework can be used to mitigate the human effort in maintaining the system.

Drawing back from the individual case studies seen earlier in this thesis, it is now time to unify these findings under a overarching umbrella framework. In this chapter, building on insights from each of the three previous case studies, a general framework that incorporates the techniques of composition, synthesis and repair for handling the maintenance of verified software is proposed, and its efficacy demonstrated by means of walking through its application on a representative example software system.

Broadly speaking, the framework proposed by this thesis consists of a series of guidelines for structuring and updating a verified software system to minimise the cost of maintenance. More specifically, this thesis recommends the following general principles for handling changes in a software system:

- **Changes in Data Structure**: This thesis recommends handling changes in data through the use of *composition*, using functors to ensure that the proofs of properties are parametric over the particular representation of their data-structures, and thereby allowing for for reduction arguments to be used to transport old proofs to new data-structures as seen in Chapter 2.
- **Changes in Specifications**: This thesis recommends handling changes in specification through

the use of verified *synthesis*, outsourcing the implementation and verification of components of the system whose specifications change frequently to a certifying synthesiser, and thereby automatically regenerating programs and proofs when specifications change, as in Chapter 3.

- **Changes in Implementation**: Finally, for programs whose implementations rely on complex intricate logic beyond the capabilities of a synthesiser and thus that have to be verified manually, the thesis recommends the direct use of proof repair strategies, utilising information within proofs of existing programs to repair the verified implementation as it changes as in Chapter 4.

In their totality, this thesis conjectures that the combination of these principles will capture the majority of ways in which real-world verified software can evolve and change, and thereby forms a principled and overarching solution to the general problem of verified software maintenance in practice.

To see this framework in action, the remainder of this chapter will provide a high level walk through of applying these rules on a hypothetical verified software system, a verified static web page server, and demonstrate how each of the guidelines can be used to maintain the system as it experiences change.

5.1 A Representative Verified Web Server

In order to illustrate the application of this framework in practice, the following sections in this thesis will consider the maintenance of a hypothetical verified server for serving static web pages. Figure 5.1 presents the core implementation for this application, with state as captured by the variables `pages, stats, bf`, and the method `handle_requests` that handles incoming requests.

This code snippet describes a fairly rudimentary static web page server. The server maintains a list of web pages, in the `pages` variable, and when a client requests a url that matches one of these pages, it returns the contents of the page. The server uses an array, `stats`, mapping each page by its index in `pages` to an integer representing the number of requests it has received, and `bf` is a Bloom Filter that is used as a cache to speed up queries for non-existent pages. For the purposes of clarity, boilerplate code such as for state initialisation, starting the web server, or data type definitions have been omitted.

The function `handle_request` (line 5-32) contains the main implementation of the server, and serves as the target of verification in the subsequent narrative. Upon receiving a request for a particular url, the function handles the query in three steps. First, the server checks if the requested url is present in the cache (line 7-9), using the No False Negatives property of the Bloom Filter to return early if this query returns negative. If the url is reported as present by the cache, the server then calls a helper

```
1  List<Page> pages;
2  Array<int> stats;
3  BloomFilter bf;
4
5  int handle_request(string url) {
6     // check if in cache
7     if(!bf.contains(url)) {
8       return NOT_FOUND;
9     }
10
11    // retrieve respective page
12    Page page =
13      find_oldest_matching_url(url);
14    if (page == null) {
15      return NOT_FOUND;
16    }
17
18    // find page index
19    var ind =
20      pages.find((i, p) =>
21         if(p == page) {
22           return Some(i);
23         } else {
24           return None;
25         }
26      ).unwrap();
27    // update stats
28    stats[ind] += 1;
29
30    write(page.data);
31    return OK;
32 }
```

Figure 5.1: Initial Web Server Implementation

function `find_oldest_matching_url` to retrieve the oldest page with a matching url (line 12-13). If no such page is found, *i.e.* the cache returned a *false positive*, then the server returns early again with `NOT_FOUND` (line 14-16). Next the program invokes a higher-order method `pages.find(f)`, that traverses over the list and returns the first element on which the function `f` returns `Some`, in this case instantiating `f` with a lambda such that it retrieves the index of the first page with a matching url (lines 19-26). This index is then used to update the corresponding element in the stats array (line 28). Finally, the program writes the contents of the page to the client before completing the request (line 30-31).

For the purposes of verification, assuming a suitable embedding of a Separation Logic-based program logic, handle_request is required to satisfy the following SL specification:

$$\forall u \ell, \{\texttt{pages} \mapsto \texttt{List}\ \ell * \texttt{bf} \mapsto \texttt{BF}\ \ell * \texttt{stats} \mapsto \texttt{Array}\ (\ldots)\}$$

$$\texttt{handle\_request}(u)$$

$$\exists res, \{\texttt{pages} \mapsto \texttt{List}\ \ell * \texttt{bf} \mapsto \texttt{BF}\ \ell\ * \texttt{stats}\ \mapsto \texttt{Array}\ (\ldots)\} \wedge (res = \texttt{OK} \Leftrightarrow u \in \ell)$$

In other words, when handle_request is invoked in a state where pages contains a list with contents ℓ, and bf contains a Bloom Filter initialised with the same contents, then the function will leave pages and bf unchanged and return OK if and only if the requested url is indeed in the server's state. To avoid complicating the presentation, the constraints on the stats state variable have been omitted in this description, although it would follow a similar form as seen in previous chapters.

The helper functions used within handle_request are given more precise specifications. While this chapter will not discuss details of the actual verification of this program *w.r.t.* the above specification, the proof and its subsequent repairs will assume that the helper functions have also been verified. For example, the correctness of this program depends on the *no false negatives* property of the Bloom Filter:

$$\forall \texttt{bf}, u, \ell, \{\texttt{bf} \mapsto \texttt{BF}\ \ell\}$$

$$\texttt{bf.contains}(u)$$

$$\exists res, \{\texttt{bf} \mapsto \texttt{Bf}\ \ell\} \wedge (res = \textbf{false} \Leftrightarrow u \notin \ell)$$

That is, the Bloom Filter will return a negative result, iff the element being queried for is not present. Similarly, the helper function find_oldest_matching_url should return a url if present:

$$\forall u, \ell, \{\texttt{pages} \mapsto \texttt{List}\ \ell\}$$

$$\texttt{find\_oldest\_matching\_url}(u)$$

$$\exists res, \{\texttt{pages} \mapsto \texttt{List}\ \ell\} \wedge (res = \textbf{null} \Leftrightarrow u \notin \ell)$$

The above formula, while under-constrained, is sufficient to prove the correctness of handle_request. A stronger specification that strictly constrains the implementation of this function will be shown later.

```
3  BloomFilter bf;
4
5  int handle_request(string url) {
6      // check if in cache
7      if(!bf.contains(url)) {
8        return NOT_FOUND;
9      }
10
11     ...
32 }
```

```
CountingFilter cf;

int handle_request(string url) {
    // check if in cache
    if(!cf.contains(url)) {
      return NOT_FOUND;
    }

    ...
}
```

Figure 5.2: An Example Change in Data Structure

▶ 5.2 Handling Changes in Data Structures

Suppose, for example, the web server were to be extended with page deletion and removal capabilities. This prompts a need to update the caching mechanism of the implementation: as Bloom filters don't support removal of previously inserted elements, the performance of this initial cache would progressively degrade as more and more pages were removed, and more queries produced false positives. A natural solution in this case, is to *change the data structure*, and replace the Bloom filter with a Counting filter, an alternative data structure that does indeed support removing elements (*cf.* Section 2.4).

Figure 5.2 presents the corresponding diff for this modification; the old program on the left, updated version on the right, and red and green highlights to indicate the updated code. The modification to the implementation is fairly minimal, and consists of replacing the `contains` membership query of the Bloom filter at line 7 with a corresponding call to the membership query of the Counting filter. With respect to verification, for the new implementation to remain certified, the user must ensure that this updated membership query still satisfies the *same* No False Negatives property of the Bloom filter:

$$\forall \texttt{cf}, u, \ell, \{\texttt{cf} \mapsto \texttt{CF}\ \ell\} \quad \texttt{cf.contains}(u) \quad \exists res, \{\texttt{cf} \mapsto \texttt{CF}\ \ell\} \wedge (res = \textbf{false} \Leftrightarrow u \notin \ell)$$

As shown earlier, this property still holds for Counting filters and so the system can be verified again. Following the guidelines of the framework, this thesis recommends that changes of such form are handled through the use of composition, using reduction arguments to transport and reuse proofs from the old data structure to the new one. In this case, by developing a reduction from Counting filters to Bloom Filters, mapping non-zero counters to raised bits, as seen in Chapter 2, the proof engineer can easily re-establish the No False Negative property for Counting filters without rewriting the original proof, and thereby the framework is effectively able to minimise the verification burden.

5.3 Handling Changes in Specifications

Having queried the cache, the server handles a request by retrieving a page with a matching url (lines 12-13), breaking ties by some arbitrary heuristic — here, this is done by choosing the oldest such page. This component of the server serves as an excellent source for the second kind of program change, that of *changes in specification*, as such heuristics themselves are frequently changed, and it is not difficult to imagine that a later version of the server might be changed to instead choose the *newest* page instead.

```
 5  int handle_request(string url) {         int handle_request(string url) {
 6      ...                                      ...
 7
11      // retrieve respective page              // retrieve respective page
12      Page page =                              Page page =
13       find_oldest_matching_url(url);           find_newest_matching_url(url);
14
15    ...                                       ...
32  }                                        }
```

Figure 5.3: An Example Change in Specification

Figure 5.3 presents the corresponding diff for implementing this change: as the implementation is entirely encapsulated within the helper method `find_oldest_matching_url`, this change can be handled by simply updating the code to call out to an updated function with the new semantics. In order to gracefully handle this change, following the principles of the framework, this thesis recommends that both the implementation and verification of *both* the old and new versions of this frequently changing function be entirely outsourced to a deductive synthesiser. In particular, in accordance with its oldest-first heuristics, `find_oldest_matching_url` can be assigned the following specification:

$$\forall u, \ell, \{\texttt{pages} \mapsto \texttt{List } \ell\}$$

$$\texttt{find\_oldest\_matching\_url}(u)$$

$$\exists res, \{\texttt{pages} \mapsto \texttt{List } \ell\} \land (res = \max_{\texttt{p.age}} \{p \mid \texttt{p} \in \ell \land \texttt{p.url} = u\} \Leftrightarrow \texttt{p} \in \ell)$$

Passing this more precise specification to a certified deductive synthesiser such as SuSLik will allow it to automatically generate a suitable implementation and *also* prove it correct, as seen in Chapter 3. With this adapted workflow, when the heuristic is inevitably changed, such as to select the newest

```
int handle_request(string url) {
    ...

    // find page index
    var ind =
      pages.find((i, p) =>
         if(p == page) {
            return Some(i);
         } else {
            return None;
         }
      ).unwrap();
    // update stats
    stats[ind] += 1;

    ...
}
```

```
int handle_request(string url) {
    ...

    // find page index
    var ind = -1;
    pages.iter((i, p) =>
       if (p == page) {
          ind = i;
          return false;
       } else {
          return true;
       }
    );
    // update stats
    stats[ind] += 1;

    ...
}
```

Figure 5.4: An Example Change in Implementation

matching page instead, the system can be appropriately updated by simply changing the post-condition:

$$\forall u, \ell, \{\texttt{pages} \mapsto \texttt{List}\ \ell\}$$

$$\texttt{find\_newest\_matching\_url}(u)$$

$$\exists res, \{\texttt{pages} \mapsto \texttt{List}\ \ell\} \wedge (res = \min_{\texttt{p.age}} \{p \mid \texttt{p} \in \ell \wedge \texttt{p.url} = u\} \Leftrightarrow \texttt{p} \in \ell)$$

5.4 Handling Changes in Implementation

To investigate the final kind of software evolution, now consider a *change in implementation* in the web server, for example, such as changing the code for calculating statistics (*cf.* Figure 5.3).

In particular, in this new implementation, instead of using the helper function `pages.find` to retrieve the appropriate index, the code now uses a different method `pages.iter` to iterate over the list of pages instead, using a callback to test for the correct page, and a mutable variable `ind` into which the index is written during the iteration. A practical motivation for this transformation could arise from the fact that `pages.find` requires its callback to return an optional result on each invocation, resulting in unnecessary allocations, whereas the new implementation avoids this by exploiting mutation.

This thesis recommends that such changes in the implementation of the verified server should be automated through proof repair techniques. In this case, the main challenge with the verification of this snippet revolves around the choice of invariant to describe the callbacks to the higher order function. Adopting the techniques of program alignment and proof-driven testing as seen in Chapter 4, an automated tool such as Sisyphus could automatically infer an invariant and repair the proof over these changes, again minimising the maintenance burden on the software engineer.

CHAPTER

6

CONCLUSION & FUTURE WORK

The aim of this thesis was to investigate techniques for scaling the evolution of verified software systems. This chapter concludes the thesis, discussing the main findings of this investigation, the limitations and any directions for future work. The thesis begins with a review of each of the previous chapters, presenting the main ideas and results, and their implications as to verified software evolution. The chapter then summaries the main takeaways from these case studies and the efficacy of the overall framework, considering any limitations and threats to validity to the results. Finally, the chapter ends with a discussion of potential directions for future work.

▶ 6.1 Conclusions

This thesis began with an innocent question: can real-world verified systems be effectively maintained? As researchers have shifted their ambitions towards the verification of larger and more grandiose software systems over the past few years, this very question has only grown in prominence and urgency within the community, with an increasing number of projects brushing up against the pains of preserving their proofs in the face of the inevitable code evolution of real world programs. To avert this oncoming crisis, this thesis has conducted a series of three methodological investigations into different approaches for maintaining a verified system as it experiences change, and through this process developed a general framework for how proof engineers can handle the evolution of their verified software systems.

Chapter 1 presented a general overview of the thesis, introducing the reader to the broad context surrounding the work, motivating how existing techniques fail to appropriately solve the problem of verified software maintenance, and proposing the general three-pronged framework argued by this thesis: that of handling verified software evolution through a combination of composition, synthesis and repair. Chapter 2 introduced the first case study of the thesis, demonstrating the efficacy of composition for verified software maintenance through a methodological investigation into how it could be used to scale up the verification of a class of probabilistic membership-query data structures, AMQs. Chapter 3 then presented the second case study of this work, investigating the use of synthesis for maintaining verified software by developing an extension to the deductive program synthesiser, SuSLik, to automatically produce certified executable code in a popular real-world language, C, when given only a specification in Separation Logic. Finally, Chapter 4 covered the last technique considered by this thesis, and explored the application of repair-based software engineering techniques to verified software, developing the first mostly-automated tool for the repair of verified OCaml program over changes in their implementation and demonstrated its efficacy through an evaluation on to repair the proofs of correctness of a number of real-world programs from widely-used libraries in the OCaml ecosystem. Unifying the findings from each of these prior case studies, Chapter 5 presented the general framework proposed by this thesis to handle verified software evolution, and then demonstrated its efficacy by considering its application on a hypothetical representative example, a verified web server.

The main contribution of this thesis is in this development of a scalable framework for verified software maintenance, capable of tackling the types of code evolution found in real world software systems. In contrast to the existing state of the art on proof engineering which has primarily considered the maintenance of mechanised proofs in isolation, *i.e.* without reference to an executable program being verified, the studies in this thesis have explored this problem taking account of changes in both the proofs *and programs*, taking significant steps towards bringing verified software evolution to practice.

6.2 Limitations & Future Work

Though this thesis purports to provide a general strategy for verified software maintenance, there are a number of limitations that should be noted which constrain the breadth of its applicability. This section details the most pertinent of these and proposes potential follow up directions that could address them.

Proof Repair in Exotic Logics While the first case study into the verification of AMQs encroached upon the realm of probabilistic programming, for the most part, the results of this thesis have been broadly focused on the context of proof maintenance for sequential straight-line single-threaded programs. For larger software systems that rely on more complex and nuanced behaviours, such as concurrent, distributed or randomised programs for example, it is not clear that the same techniques will perform as well, or whether they will require adjustments. As such, one promising direction for further research is into extending the techniques presented in this thesis for such domains — for example, would it be possible to extend a deductive synthesiser to produce certified *concurrent* code? or how can the techniques of proof-driven testing be applied to the verification of randomised systems?

Beyond Constructive Proof Assistants A more pressing and subtle limitation of the framework presented in this research is its implicit focus on software verified in constructive proof assistants, namely, for the purposes of this thesis, the Coq proof assistant. In particular, most of the techniques presented in this document, *i.e.* functors, proof terms, tactic automation, *etc.*, have been based on the facilities provided by the Coq proof assistant, and certain techniques, such as proof-driven testing, have inherently depended upon operating within such a constructive setting, assuming the existence of proof terms for example. While a substantial number of verified software projects have indeed been constructed using such or similar tooling, the same can not be said for all verified software projects and increasingly so. In particular, one alternative approach to formal verification that has seen growing popularity over time has been certifying software using automated verifiers, such as Dafny [75] or Viper [92] — verification tools that use external solvers, such as SMT solvers, to automatically dispatch many proof obligations that arise when verifying a program, and delegate other components that can not be inferred, such as invariants or specifications, to the user to provide as annotations. As such, an interesting direction for future work is to investigate whether the techniques that have been developed in this thesis in the basis of a constructive proof assistant could somehow be extended to work for this automated setting where constructive proof terms are no longer available, and if not and more generally, how verified software maintenance can be supported for these tools.

Of course, while the work presented in this thesis alone certainly can not hope to solve all the problems of verified software maintenance in their entirety, it is the author's belief that the insights and findings from this research serve as a key step on the path towards the wider goal of bringing scalable and maintainable formal verification to the masses, paving the way for future work to build on.

Bibliography

[1] Mark Adams. 2015. Refactoring Proofs with Tactician. In *SEFM (LNCS, Vol. 9509)*. Springer, 53–67. https://doi.org/10.1007/978-3-662-49224-6_6 Cited on page 15.

[2] Reynald Affeldt and Manabu Hagiwara. 2012. Formalization of Shannon's Theorems in SSReflect-Coq. In *ITP (LNCS, Vol. 7406)*. Springer, 233–249. Cited on page 24.

[3] Reynald Affeldt, Manabu Hagiwara, and Jonas Sénizergues. 2014. Formalization of Shannon's Theorems. *J. Automat. Reason.* 53, 1 (Jun 2014), 63–103. https://doi.org/10.1007/s10817-013-9298-1 Cited on page 20.

[4] Reynald Affeldt, Manabu Hagiwara, and Jonas Sénizergues. 2014. Formalization of Shannon's Theorems. *J. Autom. Reasoning* 53, 1 (2014), 63–103.

[5] Wolfgang Ahrendt, Christoph Gladisch, and Mihai Herda. 2016. Proof-based Test Case Generation. In *Deductive Software Verification - The KeY Book - From Theory to Practice*. LNCS, Vol. 10001. Springer, 415–451. https://doi.org/10.1007/978-3-319-49812-6_12 Cited on page 93.

[6] Andrew W. Appel. 2001. Foundational Proof-Carrying Code. In *LICS*. IEEE Computer Society, 247–256. Cited on page 62.

[7] Andrew W. Appel. 2011. Verified Software Toolchain - (Invited Talk). In *ESOP (LNCS, Vol. 6602)*. Springer, 1–17. https://doi.org/10.1007/978-3-642-19718-5_1 Cited on pages 17, 50, and 94.

[8] Andrew W. Appel, Robert Dockins, Aquinas Hobor, Lennart Beringer, Josiah Dodds, Gordon Stewart, Sandrine Blazy, and Xavier Leroy. 2014. *Program Logics for Certified Compilers*. Cambridge University Press. https://doi.org/10.1017/CBO9781107256552 Cited on page 57.

[9] Andrew W. Appel and David A. Naumann. 2020. Verified sequential Malloc/Free. In *ISMM*. ACM, 48–59. https://doi.org/10.1145/3381898.3397211 Cited on page 66.

[10] Philippe Audebaud and Christine Paulin-Mohring. 2009. Proofs of randomized algorithms in Coq. *Science of Computer Programming* 74, 8 (2009), 568–589. Cited on page 45.

[11] Gilles Barthe, Benjamin Grégoire, and Santiago Zanella Béguelin. 2009. Formal certification of code-based cryptographic proofs. In *POPL*. ACM, 90–101. Cited on page 45.

[12] Gilles Barthe, Boris Köpf, Federico Olmedo, and Santiago Zanella Béguelin. 2012. Probabilistic relational reasoning for differential privacy. In *POPL*. ACM, 97–110. Cited on page 46.

[13] Gilles Barthe, David Pichardie, and Tamara Rezk. 2013. A certified lightweight non-interference Java bytecode verifier. *Math. Struct. Comput. Sci.* 23, 5 (2013), 1032–1081. Cited on page 62.

[14] Michael A. Bender, Martin Farach-Colton, Rob Johnson, Russell Kraner, Bradley C. Kuszmaul, Dzejla Medjedovic, Pablo Montes, Pradeep Shetty, Richard P. Spillane, and Erez Zadok. 2012. Don't thrash: how to cache your hash on flash. *Proc. VLDB Endow.* 5, 11 (Jul 2012), 1627–1637. https://doi.org/10.14778/2350229.2350275 Cited on pages 20, 41, 42, and 47.

[15] Frédéric Besson, Thomas P. Jensen, and David Pichardie. 2006. Proof-carrying code from certified abstract interpretation and fixpoint compression. *Theor. Comput. Sci.* 364, 3 (2006), 273–291. Cited on page 62.

[16] Sam Blackshear, Nikos Gorogiannis, Peter W. O'Hearn, and Ilya Sergey. 2018. RacerD: compositional static race detection. *Proc. ACM Program. Lang.* 2, OOPSLA (2018), 144:1–144:28. https://doi.org/10.1145/3276514 Cited on page 14.

[17] Burton H. Bloom. 1970. Space/Time Trade-offs in Hash Coding with Allowable Errors. *Commun. ACM* 13, 7 (July 1970), 422–426. Cited on pages 17, 20, 21, 22, 41, 44, and 47.

[18] Arthur Blot, Pierre-Évariste Dagand, and Julia Lawall. 2016. From Sets to Bits in Coq. In *FLOPS (LNCS, Vol. 9613)*. Springer, 12–28. Cited on page 46.

[19] Prosenjit Bose, Hua Guo, Evangelos Kranakis, Anil Maheshwari, Pat Morin, Jason Morrison, Michiel Smid, and Yihui Tang. 2008. On the false-positive rate of Bloom filters. *Inform. Process. Lett.* 108, 4 (2008), 210–213. Cited on pages 22, 41, 44, and 47.

[20] Timothy Bourke, Matthias Daum, Gerwin Klein, and Rafal Kolanski. 2012. Challenges and Experiences in Managing Large-Scale Proofs. In *11th International Conference Intelligent Computer Mathematics (CICM) (LNCS, Vol. 7362)*. Springer, 32–48. https://doi.org/10.1007/978-3-642-31374-5_3 Cited on page 15.

[21] Marc Brockschmidt, Yuxin Chen, Pushmeet Kohli, Siddharth Krishna, and Daniel Tarlow. 2017. Learning Shape Analysis. In *SAS (LNCS, Vol. 10422)*. Springer, 66–87. https://doi.org/10.1007/978-3-319-66706-5_4 Cited on page 94.

[22] Andrei Z. Broder and Michael Mitzenmacher. 2003. Survey: Network Applications of Bloom Filters: A Survey. *Internet Mathematics* 1, 4 (2003), 485–509. Cited on page 44.

[23] Qinxiang Cao, Lennart Beringer, Samuel Gruetter, Josiah Dodds, and Andrew W. Appel. 2018. VST-Floyd: A Separation Logic Tool to Verify Correctness of C Programs. *J. Autom. Reason.* 61, 1-4 (2018), 367–422. Cited on page 61.

[24] Tej Chajed, Joseph Tassarotti, M. Frans Kaashoek, and Nickolai Zeldovich. 2019. Verifying concurrent, crash-safe systems with Perennial. In *SOSP*. ACM, 243–258. Cited on page 12.

[25] Arthur Charguéraud. 2020. Separation Logic for Sequential Programs (Functional Pearl). *Proc. ACM Program. Lang.* 4, ICFP (2020), 116:1–116:34. https://doi.org/10.1145/3408998 Cited on page 94.

[26] Arthur Charguéraud, Jean-Christophe Filliâtre, François Pottier, and Mário Pereira. 2017. VOCAL – A Verified OCaml Library. In *ML Family Workshop*. Cited on page 66.

[27] Haogang Chen, Daniel Ziegler, Tej Chajed, Adam Chlipala, M. Frans Kaashoek, and Nickolai Zeldovich. 2015. Using Crash Hoare logic for certifying the FSCQ file system. In *SOSP*. ACM, 18–37. Cited on pages 12 and 94.

[28] Juan Chen, Ravi Chugh, and Nikhil Swamy. 2010. Type-preserving compilation of end-to-end verification of security enforcement. In *PLDI*. ACM, 412–423. Cited on page 62.

[29] Adam Chlipala. 2011. Mostly-automated verification of low-level programs in computational separation logic. In *PLDI*. ACM, 234–245. https://doi.org/10.1145/1993498.1993526 Cited on page 94.

[30] Adam Chlipala, Benjamin Delaware, Samuel Duchovni, Jason Gross, Clément Pit-Claudel, Sorawit Suriyakarn, Peng Wang, and Katherine Ye. 2017. The End of History? Using a Proof Assistant to Replace Language Design with Library Design. In *SNAPL (LIPIcs, Vol. 71)*. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 3:1–3:15. Cited on page 63.

[31] Ken Christensen, Allen Roginsky, and Miguel Jimeno. 2010. A new analysis of the false positive rate of a Bloom filter. *Inform. Process. Lett.* 110, 21 (2010), 944–949. Cited on page 44.

[32] Coq Development Team. 2018. *The Coq Proof Assistant Reference Manual - Version 8.8*. Available at http://coq.inria.fr/. Cited on page 20.

[33] Satyaki Das, David L. Dill, and Seungjoon Park. 1999. Experience with Predicate Abstraction. In *CAV (LNCS, Vol. 1633)*, Nicolas Halbwachs and Doron A. Peled (Eds.). Springer, 160–171. https://doi.org/10.1007/3-540-48683-6_16 Cited on page 94.

[34] Leonardo Mendonça de Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *TACAS (LNCS, Vol. 4963)*. Springer, 337–340. https://doi.org/10.1007/978-3-540-78800-3_24 Cited on page 89.

[35] Biplob Debnath, Sudipta Sengupta, Jin Li, David J Lilja, and David HC Du. 2011. BloomFlash: Bloom filter on flash-based storage. In *2011 31st International Conference on Distributed Computing Systems*. IEEE, 635–644. Cited on page 44.

[36] Benjamin Delaware, Clément Pit-Claudel, Jason Gross, and Adam Chlipala. 2015. Fiat: Deductive Synthesis of Abstract Data Types in a Proof Assistant. In *POPL*. ACM, 689–700. https://doi.org/10.1145/2676726.2677006 Cited on page 15.

[37] Benjamin Delaware, Clément Pit-Claudel, Jason Gross, and Adam Chlipala. 2015. Fiat: Deductive Synthesis of Abstract Data Types in a Proof Assistant. In *POPL*. ACM, 689–700. Cited on page 63.

[38] Richard A. DeMillo, Richard J. Lipton, and Alan J. Perlis. 1977. Social Processes and Proofs of Theorems and Programs. In *POPL*. ACM, 206–214. https://doi.org/10.1145/512950.512970 Cited on page 14.

[39] Sarang Dharmapurikar, Praveen Krishnamurthy, Todd S. Sproull, and John W. Lockwood. 2004. Deep Packet Inspection using Parallel Bloom Filters. *IEEE Micro* 24, 1 (2004), 52–61. Cited on page 44.

[40] Sarang Dharmapurikar, Praveen Krishnamurthy, and David E. Taylor. 2006. Longest prefix matching using Bloom filters. *IEEE/ACM Trans. Netw.* 14, 2 (2006), 397–409. Cited on page 44.

[41] Manuel Eberl, Max W. Haslbeck, and Tobias Nipkow. 2020. Verified Analysis of Random Binary Tree Structures. *J. Autom. Reason.* 64, 5 (2020), 879–910. https://doi.org/10.1007/S10817-020-09545-0

[42] Michael D. Ernst, Jake Cockrell, William G. Griswold, and David Notkin. 1999. Dynamically Discovering Likely Program Invariants to Support Program Evolution. In *ICSE*. ACM, 213–224. https://doi.org/10.1145/302405.302467 Cited on page 94.

[43] Michael D. Ernst, Adam Czeisler, William G. Griswold, and David Notkin. 2000. Quickly detecting relevant program invariants. In *ICSE*. ACM, 449–458. https://doi.org/10.1145/337180.337240 Cited on page 94.

[44] Jean-Christophe Filliâtre and Andrei Paskevich. 2013. Why3 - Where Programs Meet Provers. In *ESOP (LNCS, Vol. 7792)*. Springer, 125–128. https://doi.org/10.1007/978-3-642-37036-6_8 Cited on page 94.

[45] Cormac Flanagan and K. Rustan M. Leino. 2001. Houdini, an Annotation Assistant for ESC/Java. In *FME (LNCS, Vol. 2021)*. Springer, 500–517. https://doi.org/10.1007/3-540-45251-6_29 Cited on page 94.

[46] Michele Giry. 1982. A categorical approach to probability theory. In *Categorical aspects of topology and analysis*. Springer, 68–85. Cited on page 45.

[47] Georges Gonthier, Assia Mahboubi, and Enrico Tassi. 2009. *A Small Scale Reflection Extension for the Coq system*. Technical Report 6455. Microsoft Research – Inria Joint Centre. Cited on page 20.

[48] Georges Gonthier, Beta Ziliani, Aleksandar Nanevski, and Derek Dreyer. 2011. How to make ad hoc proof automation less ad hoc. In *ICFP*. ACM, 163–175. https://doi.org/10.1145/2034773.2034798 Cited on page 94.

[49] Kiran Gopinathan, Mayank Keoliya, and Ilya Sergey. 2023. Mostly Automated Proof Repair for Verified Libraries. *Proc. ACM Program. Lang.* 7, PLDI (2023), 25–49. Cited on pages 17 and 67.

[50] Kiran Gopinathan, Mayank Keoliya, and Ilya Sergey. 2023. *Reproduction Artefact for Article "Mostly Automated Proof Repair for Verified Libraries"*. https://doi.org/10.5281/zenodo.7703886 Cited on page 89.

[51] Kiran Gopinathan and Ilya Sergey. 2019. Towards Mechanising Probabilistic Properties of a Blockchain. In *CoqPL 2019: The Fifth International Workshop on Coq for Programming Languages*. Cited on page 45.

[52] Kiran Gopinathan and Ilya Sergey. 2020. Ceramist: Verified Hash-based Approximate Membership Structures. https://doi.org/10.5281/zenodo.3749474 CAV 2020 Artefact DOI: 10.5281/zenodo.3749474, sources available at https://github.com/certichain/ceramist.. Cited on pages 20 and 41.

[53] Kiran Gopinathan and Ilya Sergey. 2020. Certifying Certainty and Uncertainty in Approximate Membership Query Structures. *Computer Aided Verification* 12225 (Jun 2020), 279. https://doi.org/10.1007/978-3-030-53291-8_16 Cited on pages 17 and 20.

[54] Mike Gordon, Juliano Iyoda, Scott Owens, and Konrad Slind. 2005. Automatic Formal Synthesis of Hardware from Higher Order Logic. In *Proceedings of the 5th International Workshop on Automated Verification of Critical Systems, AVoCS 2005, University of Warwick, UK, September 12-13, 2005 (Electronic Notes in Theoretical Computer Science, Vol. 145).* Elsevier, 27–43. https://doi.org/10.1016/J.ENTCS.2005.10.003

[55] Susanne Graf and Hassen Saïdi. 1997. Construction of Abstract State Graphs with PVS. In *CAV (LNCS, Vol. 1254).* Springer, 72–83. https://doi.org/10.1007/3-540-63166-6_10 Cited on page 94.

[56] Ronald L. Graham, Donald E. Knuth, and Oren Patashnik. 1994. *Concrete Mathematics: A Foundation for Computer Science, 2nd Ed.* Addison-Wesley. Cited on pages 20 and 26.

[57] Ronghui Gu, Zhong Shao, Hao Chen, Xiongnan (Newman) Wu, Jieung Kim, Vilhelm Sjöberg, and David Costanzo. 2016. CertiKOS: An Extensible Architecture for Building Certified Concurrent OS Kernels. In *OSDI.* USENIX Association, 653–669. Cited on page 12.

[58] Robert Harper, Furio Honsell, and Gordon D. Plotkin. 1993. A Framework for Defining Logics. *J. ACM* 40, 1 (1993), 143–184. Cited on page 62.

[59] Chris Hawblitzel, Jon Howell, Manos Kapritsos, Jacob R. Lorch, Bryan Parno, Michael L. Roberts, Srinath T. V. Setty, and Brian Zill. 2015. IronFleet: proving practical distributed systems correct. In *SOSP.* ACM, 1–17. Cited on page 12.

[60] Jifeng He, C. A. R. Hoare, and Jeff W. Sanders. 1986. Data Refinement Refined. In *ESOP (LNCS, Vol. 213).* Springer, 187–196. Cited on page 63.

[61] Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Kenneth L. McMillan. 2004. Abstractions

from proofs. In *POPL*. ACM, 232–244. https://doi.org/10.1145/964001.964021 Cited on page 94.

[62] Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, George C. Necula, Grégoire Sutre, and Westley Weimer. 2002. Temporal-Safety Proofs for Systems Code. In *CAV (LNCS, Vol. 2404)*. Springer, 526–538. Cited on page 62.

[63] Son Ho, Oskar Abrahamsson, Ramana Kumar, Magnus O. Myreen, Yong Kiam Tan, and Michael Norrish. 2018. Proof-Producing Synthesis of CakeML with I/O and Local State from Monadic HOL Functions. In *Automated Reasoning - 9th International Joint Conference, IJCAR 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 14-17, 2018, Proceedings (LNCS, Vol. 10900)*. Springer, 646–662. https://doi.org/10.1007/978-3-319-94205-6_42

[64] Johannes Hölzl. 2017. Markov Processes in Isabelle/HOL. In *CPP*. ACM, 100–111. Cited on page 46.

[65] Joe Hurd. 2003. *Formal verification of probabilistic algorithms*. Technical Report UCAM-CL-TR-566. University of Cambridge, Computer Laboratory. https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-566.pdf

[66] Joe Hurd, Annabelle McIver, and Carroll Morgan. 2005. Probabilistic guarded commands mechanized in HOL. *Theor. Comput. Sci.* 346, 1 (2005), 96–112. https://doi.org/10.1016/J.TCS.2005.08.005

[67] The Iris Project. 2022. Iris: a Higher-Order Concurrent Separation Logic Framework, implemented and verified in the Coq proof assistant. https://iris-project.org/ Online; last accessed 6 November 2022. Cited on page 95.

[68] Shachar Itzhaky, Hila Peleg, Nadia Polikarpova, Reuben Rowe, and Ilya Sergey. 2021. Cyclic Program Synthesis. In *PLDI*. ACM, 944–959. https://doi.org/10.1145/3453483.3454087 Cited on pages 50 and 51.

[69] Chi Jing. 2009. Application and Research on Weighted Bloom Filter and Bloom Filter in Web Cache. In *2009 Second Pacific-Asia Conference on Web Mining and Web-based Application*. 187–191. Cited on page 44.

[70] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. 2009. seL4: formal verification of an OS kernel. In *SOSP*. ACM, 207–220. Cited on page 12.

[71] Ramana Kumar, Magnus O. Myreen, Michael Norrish, and Scott Owens. 2014. CakeML: a verified implementation of ML. In *POPL*. ACM, 179–192. Cited on pages 12, 15, and 63.

[72] Peter Lammich. 2015. Refinement to Imperative/HOL. In *ITP (LNCS, Vol. 9236)*. Springer, 253–269. https://doi.org/10.1007/978-3-319-22102-1_17

[73] Peter Lammich. 2019. Generating Verified LLVM from Isabelle/HOL. In *ITP (LIPIcs, Vol. 141)*. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 22:1–22:19. https://doi.org/10.4230/LIPICS.ITP.2019.22

[74] Ton Chanh Le, Guolong Zheng, and ThanhVu Nguyen. 2019. SLING: Using Dynamic Analysis to Infer Program Invariants in Separation Logic. In *PLDI*. ACM, 788–801. https://doi.org/10.1145/3314221.3314634 Cited on page 94.

[75] K. Rustan M. Leino. 2010. Dafny: An Automatic Program Verifier for Functional Correctness. In *LPAR (LNCS, Vol. 6355)*. Springer, 348–370. https://doi.org/10.1007/978-3-642-17511-4_20

[76] Xavier Leroy. 2006. Formal certification of a compiler back-end or: programming a compiler with a proof assistant. In *POPL*. ACM, 42–54. Cited on pages 12 and 15.

[77] Pierre Letouzey. 2008. Extraction in Coq: An Overview. In *4th Conference on Computability in Europe (CiE) (LNCS, Vol. 5028)*. Springer, 359–369. https://doi.org/10.1007/978-3-540-69407-6_39 Cited on page 86.

[78] Guodong Li, Scott Owens, and Konrad Slind. 2007. Structure of a Proof-Producing Compiler for a Subset of Higher Order Logic. In *ESOP (LNCS, Vol. 4421)*. Springer, 205–219. https://doi.org/10.1007/978-3-540-71316-6_15

[79] Guodong Li and Konrad Slind. 2007. Compilation as Rewriting in Higher Order Logic. In *Automated Deduction - CADE-21, 21st International Conference on Automated Deduction, Bremen, Germany, July 17-20, 2007, Proceedings (LNCS, Vol. 4603)*. Springer, 19–34. https://doi.org/10.1007/978-3-540-73595-3_3

[80] Guodong Li and Konrad Slind. 2008. Trusted Source Translation of a Total Function Language. In *TACAS (LNCS, Vol. 4963)*. Springer, 471–485. https://doi.org/10.1007/978-3-540-78800-3_37

[81] Yun-Zhao Li. 2009. Memory Efficient Parallel Bloom Filters for String Matching. In *2009 International Conference on Networks Security, Wireless Communications and Trusted Computing*, Vol. 1. 485–488. Cited on page 44.

[82] Hyesook Lim, Jungwon Lee, and Changhoon Yim. 2015. Complement Bloom Filter for Identifying True Positiveness of a Bloom Filter. *IEEE Communications Letters* 19, 11 (2015), 1905–1908. Cited on page 44.

[83] Andreas Lochbihler. 2016. Probabilistic Functions and Cryptographic Oracles in Higher Order Logic. In *ESOP (LNCS, Vol. 9632)*. Springer, 503–531. https://doi.org/10.1007/978-3-662-49498-1_20

[84] Andreas Lööw and Magnus O. Myreen. 2019. A proof-producing translator for verilog development in HOL. In *ICSE*. IEEE / ACM, 99–108. https://doi.org/10.1109/FORMALISE.2019.00020

[85] Stephen Magill, Aleksandar Nanevski, Edmund Clarke, and Peter Lee. 2006. Inferring Invariants in Separation Logic for Imperative List-Processing Programs. *The third workshop on Semantics, Program Analysis and Computing Environments for Memory Management (SPACE)* 1, 1, 5–7. Cited on page 94.

[86] Assia Mahboubi and Enrico Tassi. 2017. *Mathematical Components*. Available at https://math-comp.github.io/mcb. Cited on pages 20 and 41.

[87] Erik Martin-Dorel and Sergei Soloviev. 2018. A Formal Study of Boolean Games with Random Formulas as Payoff Functions. In *TYPES 2016 (LIPIcs, Vol. 97)*. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 14:1–14:22. Cited on pages 35 and 40.

[88] Daniel Matichuk. 2012. Automatic Function Annotations for Hoare Logic. In *Proceedings Seventh Conference on Systems Software Verification (SSV) (EPTCS, Vol. 102)*. 46–56. https://doi.org/10.4204/EPTCS.102.6 Cited on page 93.

[89] Michael Mitzenmacher. 2002. Compressed Bloom filters. *IEEE/ACM Transactions on Networking* 10, 5 (2002), 604–612. Cited on page 44.

[90] Michael Mitzenmacher and Eli Upfal. 2017. *Probability and Computing: Randomized Algorithms and Probabilistic Analysis*. Cambridge University Press. ISBN 978-1-107-15488-9, Second Edition. Cited on page 44.

[91] Ike Mulder, Robbert Krebbers, and Herman Geuvers. 2022. Diaframe: automated verification of fine-grained concurrent programs in Iris. In *PLDI*. ACM, 809–824. https://doi.org/10.1145/3519939.3523432 Cited on page 94.

[92] Peter Müller, Malte Schwerhoff, and Alexander J. Summers. 2016. Viper: A Verification Infrastructure for Permission-Based Reasoning. In *VMCAI (LNCS, Vol. 9583)*. Springer, 41–62.

[93] Magnus O. Myreen and Scott Owens. 2012. Proof-producing synthesis of ML from higher-order logic. In *ICFP*. ACM, 115–126. https://doi.org/10.1145/2364527.2364545

[94] Magnus O. Myreen, Konrad Slind, and Michael J. C. Gordon. 2009. Extensible Proof-Producing Compilation. In *Compiler Construction, 18th International Conference, CC 2009, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009, York, UK, March 22-29, 2009. Proceedings (LNCS, Vol. 5501)*. Springer, 2–16. https://doi.org/10.1007/978-3-642-00722-4_2

[95] Aleksandar Nanevski, Viktor Vafeiadis, and Josh Berdine. 2010. Structuring the verification of heap-manipulating programs. In *POPL*. 261–274. https://doi.org/10.1145/1706299.1706331 Cited on page 94.

[96] George C. Necula. 1997. Proof-Carrying Code. In *POPL*. ACM Press, 106–119. Cited on pages 62 and 63.

[97] George C. Necula and Peter Lee. 1996. Safe Kernel Extensions Without Run-Time Checking. In *OSDI*. ACM, 229–243. Cited on page 62.

[98] George C. Necula and Peter Lee. 1998. The Design and Implementation of a Certifying Compiler. In *PLDI*. ACM, 333–344. https://doi.org/10.1145/277650.277752 Cited on page 15.

[99] George C. Necula and Peter Lee. 1998. The Design and Implementation of a Certifying Compiler. In *PLDI*. ACM, 333–344. Cited on page 62.

[100] Peter W. O'Hearn, John C. Reynolds, and Hongseok Yang. 2001. Local Reasoning about Programs that Alter Data Structures. In *CSL (LNCS, Vol. 2142)*. Springer, 1–19. https://doi.org/10.1007/3-540-44802-0_1 Cited on pages 49 and 69.

[101] Anna Pagh, Rasmus Pagh, and S. Srinivasa Rao. 2005. An optimal Bloom filter replacement. In *SODA*. SIAM, 823–829. Cited on page 20.

[102] Adam Petcher and Greg Morrisett. 2015. The Foundational Cryptography Framework. In *POST (LNCS, Vol. 9036)*. Springer, 53–72. Cited on pages 23 and 45.

[103] Guillaume Petiot, Bernard Botella, Jacques Julliand, Nikolai Kosmatov, and Julien Signoles. 2014. Instrumentation of Annotated C Programs for Test Generation. In *14th IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM)*. IEEE Computer Society, 105–114. `https://doi.org/10.1109/SCAM.2014.19` Cited on page 93.

[104] Ruzica Piskac, Thomas Wies, and Damien Zufferey. 2014. GRASShopper - Complete Heap Verification with Mixed Specifications. In *TACAS (LNCS, Vol. 8413)*. Springer, 124–139. `https://doi.org/10.1007/978-3-642-54862-8_9` Cited on page 94.

[105] Nadia Polikarpova and Ilya Sergey. 2019. Structuring the Synthesis of Heap-Manipulating Programs. *PACMPL* 3, POPL (2019), 72:1–72:30. Cited on pages 16, 17, 49, and 51.

[106] Nadia Polikarpova, Julian Tschannen, and Carlo A. Furia. 2018. A fully verified container library. *Formal Aspects Comput.* 30, 5 (2018), 495–523. `https://doi.org/10.1007/s00165-017-0435-1` Cited on page 66.

[107] Felix Putze, Peter Sanders, and Johannes Singler. 2009. Cache-, hash-, and space-efficient bloom filters. *ACM Journal of Experimental Algorithmics* 14 (2009). Cited on pages 20, 42, 43, 44, 45, and 47.

[108] Yan Qiao, Tao Li, and Shigang Chen. 2011. One memory access Bloom filters and their generalization. In *INFOCOM*. IEEE, 1745–1753. Cited on page 44.

[109] Shengchao Qin, Guanhua He, Chenguang Luo, Wei-Ngan Chin, and Xin Chen. 2013. Loop invariant synthesis in a combined abstract domain. *J. Symb. Comput.* 50 (2013), 386–408. `https://doi.org/10.1016/j.jsc.2012.08.007` Cited on page 94.

[110] Vincent Rahli, Ivana Vukotic, Marcus Völp, and Paulo Jorge Esteves Veríssimo. 2018. Velisarios: Byzantine Fault-Tolerant Protocols Powered by Coq. In *ESOP (LNCS, Vol. 10801)*. Springer, 619–650. Cited on page 12.

[111] Norman Ramsey and Avi Pfeffer. 2002. Stochastic lambda calculus and monads of probability distributions. In *POPL*. ACM, 154–165. Cited on page 24.

[112] John C. Reynolds. 2002. Separation Logic: A Logic for Shared Mutable Data Structures. In *LICS*. IEEE Computer Society, 55–74. https://doi.org/10.1109/LICS.2002.1029817 Cited on pages 49, 51, and 69.

[113] Talia Ringer. 2021. *Proof Repair.* Ph. D. Dissertation. University of Washington. Cited on pages 14, 66, and 93.

[114] Talia Ringer, RanDair Porter, Nathaniel Yazdani, John Leo, and Dan Grossman. 2021. Proof repair across type equivalences. In *PLDI*. ACM, 112–127. Cited on pages 14, 66, and 93.

[115] Talia Ringer, Nathaniel Yazdani, John Leo, and Dan Grossman. 2018. Adapting proof automation to adapt proofs. In *CPP*. ACM, 115–129. https://doi.org/10.1145/3167094 Cited on pages 14, 66, and 93.

[116] Talia Ringer, Nathaniel Yazdani, John Leo, and Dan Grossman. 2019. Ornaments for Proof Reuse in Coq. In *ITP (LIPIcs, Vol. 141)*. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 26:1–26:19. https://doi.org/10.4230/LIPIcs.ITP.2019.26 Cited on page 93.

[117] Valentin Robert. 2018. *Front-end tooling for building and maintaining dependently-typed functional programs.* Ph. D. Dissertation. University of California, San Diego, USA. Cited on page 15.

[118] Reuben N. S. Rowe and James Brotherston. 2017. Automatic cyclic termination proofs for recursive procedures in separation logic. In *CPP*. ACM, 53–65. Cited on page 51.

[119] Michael Sammler, Rodolphe Lepigre, Robbert Krebbers, Kayvan Memarian, Derek Dreyer, and Deepak Garg. 2021. RefinedC: automating the foundational verification of C code with refined ownership types. In *PLDI*. ACM, 158–174. https://doi.org/10.1145/3453483.3454036 Cited on page 94.

[120] Ilya Sergey, Aleksandar Nanevski, and Anindya Banerjee. 2015. Mechanized verification of fine-grained concurrent programs. In *PLDI*. ACM, 77–87. https://doi.org/10.1145/2737924.2737964 Cited on page 94.

[121] Xujie Si, Hanjun Dai, Mukund Raghothaman, Mayur Naik, and Le Song. 2018. Learning Loop Invariants for Program Verification. In *NeurIPS*. 7762–7773. Cited on page 94.

[122] Christian Skalka, John Ring, David Darais, Minseok Kwon, Sahil Gupta, Kyle Diller, Steffen Smolka, and Nate Foster. 2019. Proof-Carrying Network Code. In *CCS*. ACM, 1115–1129. Cited on page 62.

[123] Dario Stein and Sam Staton. 2021. Compositional Semantics for Probabilistic Programs with Exact Conditioning. In *36th Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2021, Rome, Italy, June 29 - July 2, 2021*. IEEE, 1–13. https://doi.org/10.1109/LICS52264.2021.9470552 Cited on page 14.

[124] Pierre-Yves Strub, Tetsuya Sato, Justin Hsu, Thomas Espitau, and Gilles Barthe. 2019. Relational ⋆-Liftings for Differential Privacy. *Logical Methods in Computer Science* 15, 4 (2019). Cited on page 47.

[125] Sasu Tarkoma, Christian Esteve Rothenberg, and Eemil Lagerspetz. 2012. Theory and Practice of Bloom Filters for Distributed Systems. *IEEE Communications Surveys and Tutorials* 14, 1 (2012), 131–155. Cited on pages 20, 31, 41, 44, and 47.

[126] Joseph Tassarotti and Robert Harper. 2019. A separation logic for concurrent randomized programs. *PACMPL* 3, POPL (2019), 64:1–64:30. Cited on page 45.

[127] Marcelo Taube, Giuliano Losa, Kenneth L. McMillan, Oded Padon, Mooly Sagiv, Sharon Shoham, James R. Wilcox, and Doug Woos. 2018. Modularity for decidability of deductive verification with applications to distributed systems. In *PLDI*. ACM, 662–677. https://doi.org/10.1145/3192366.3192414 Cited on page 14.

[128] Amin Timany and Bart Jacobs. 2015. First Steps Towards Cumulative Inductive Types in CIC. In *ICTAC (LNCS)*. Springer, 608–617. https://doi.org/10.1007/978-3-319-25150-9_36 Cited on page 84.

[129] Yasunari Watanabe, Kiran Gopinathan, George Pîrlea, Nadia Polikarpova, and Ilya Sergey. 2021. *Certified SuSLik (ICFP 2021 Artifact): Code and Benchmarks*. https://doi.org/10.5281/zenodo.5005829 Cited on page 60.

[130] Yasunari Watanabe, Kiran Gopinathan, George Pirlea, Nadia Polikarpova, and Ilya Sergey. 2021. Certifying the Synthesis of Heap-Manipulating Programs. *Proc. ACM Program. Lang.* 5, ICFP (2021), 1–29. https://doi.org/10.1145/3473589 Cited on pages 10, 17, 18, 48, 50, 53, and 56.

[131] Karin Wibergh. 2019. *Automatic refactoring for Agda*. Master's thesis. Chalmers University of Technology and University of Gothenburg. Cited on page 15.

[132] Doug Woos, James R. Wilcox, Steve Anton, Zachary Tatlock, Michael D. Ernst, and Thomas E. Anderson. 2016. Planning for change in a formal verification of the raft consensus protocol. In *CPP*. ACM, 154–165. https://doi.org/10.1145/2854065.2854081 Cited on pages 12 and 14.