TOWARDS ELIMINATING EXPERT CREATIVE HELP IN AUTOMATED REASONING

BY

ADITHYA MURALI

DISSERTATION

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois Urbana-Champaign, 2024

Urbana, Illinois

Doctoral Committee:

Professor Madhusudan Parthasarathy, Chair
Professor Mahesh Viswanathan
Assistant Professor Gagandeep Singh
Professor Ranjit Jhala, University of California, San Diego
Professor Swarat Chaudhuri, The University of Texas at Austin

Abstract

The democratization of automated reasoning is a dream for computer scientists like few others. However, despite many years of progress there continue to exist fundamental obstacles in the way. In this thesis we identify one of the key obstacles: the inescapable need for expert help encountered when using automated reasoning tools or algorithms to prove rich sets of properties. This help takes many forms across the many reasoning frameworks that exist, but we argue that in many widely used frameworks the help is a technical intervention that experts are able to come up with by studying a problem, simply using their experience and creativity. In this work we seek to unravel the nature of this creative technical help and take steps towards eliminating it.

Although at first sight the workflow we describe may appear informal or unorganized, our study reveals that it is possible to formally characterize the limits of the reasoning power of various automatic tools and heuristics. Furthermore, we show that the expert help in fact bridges (again in a formal sense) the gap between the limits of the reasoning algorithms and the power needed to prove the properties desired. We dub such gaps in automated reasoning *creativity gaps* and develop new theoretical tools to formally characterize them.

Understanding the role of the expert help in a formal sense allows us to formulate well-defined computational problems to solve in order to bridge creativity gaps. We argue that such problems can be solved effectively using a form of learning we call *logic learning* in this thesis, which refers to the problem of learning logical formulas from rich example structures/logical models. We develop new frameworks and algorithms for logic learning and use them to bridge different creativity gaps.

In a third part of our work, we apply the lens of thinking about the dynamics between expert help and automation to interrogate design principles for new verification paradigms that can minimize the kind of user frustrations we focus on in this work, namely dealing with the opaqueness of creativity gaps and the inability to provide the required help to bridge the gaps without deep verification expertise. We formulate the problem of designing a *predictable verification* framework and develop a new framework for verifying heap manipulating programs that offers a predictable verification experience.

We believe that the contributions of this work take significant steps towards eliminating creativity gaps in automated reasoning, and in doing so pave the way further for progress towards the democratization of automated reasoning.

To Smriti and Loki
You give my life meaning

Table of Contents

Chapter 1: Introduction

The desire for bug-free software is a problem as old as computer science. While testing is the most widely adopted mechanism for assuring software correctness, several infamous incidents in recent years have shown that even the most rigorously tested software systems can violate privacy of personal data [1], compromise financial security [2], or endanger the physical safety [3, 4] of users. As a result, formal verification is being increasingly advocated to serve as the standard for assuring software quality. Verification requires the engineer to describe the desired behavior of a piece of software mathematically and write a computer-checkable proof that the code meets the given specification. On the face of it, this is not a particularly unreasonable ask. Programmers in fact routinely argue (albeit informally) the correctness of thousands of theorems about their code, ranging from simple properties like type-correctness to complex properties like eventual consistency of a blockchain. All formal verification seems to require on top of this is to do the reasoning in a way that computers can check!

However, formal verification in practice is intellectually demanding, requiring experts with many years of training in formal logic and deductive reasoning techniques. This is obviously not desirable, and it makes adoption extremely hard. This tension between the ease of adoption and the strength of guarantees has in practice always resolved towards eschewing verification, with the exception of a few complex software systems. To make verification easier to adopt, we must automate the reasoning involved as much as possible. This argument is not new, and techniques for automated reasoning have made massive strides in the last few decades. These advances have led to many important reliable artifacts that would not have been possible without automation, such as device drivers [5], operating system kernels [6], and large fault-tolerant distributed systems for the cloud [7, 8].

Despite these successes automated verification is at an impasse today, both in terms of technical advances as well as practical adoption. This is because automation for software verification requires crucial technical intervention from verification experts to work, i.e., automated verification isn't really automatic! Indeed, the successes of automated verification cited above have required teams with significant verification expertise. Note that this is *different* from the effort or expertise required to write formal specifications. Even with fixed specifications, current automated verification tools require expert technical help in order to work effectively.

The Problem with Expert Creative Help Existing automated reasoning paradigms practice a two-part approach:

(1) A human, typically an expert, uses their creativity (💡) to formally state their high-level arguments about a system's correctness, and
(2) An automated reasoning engine mechanically (⚙) verifies the stated arguments.

The two-part (💡) + (⚙) view is a lens that applies to many reasoning problems. For example, in automated program verification, the expert breaks down the problem into modular specifications in the form of contracts for individual methods and loop invariants, and then an automated engine like Dafny [9] effectively verifies the methods. Another example is interactive theorem proving in tools like Coq [10], where the expert breaks down a theorem into intermediate lemmas such that SMT solvers [11, 12] can verify that each step follows from the previous steps. Emerging work in the field of AI-driven theorem proving adopts this lens as well [13]. One can even identify these separation of intellectual abilities in applications beyond verification. For example, in mathematical and scientific reasoning, experts come up with axioms to characterize a domain and may then use automatic tools to reason using those laws. As our field moves towards making impact in other scientific disciplines, discovering such laws is a creative task that must be automated.

The above discussion indicates the first problem with the required creative help: it's pervasive and seemingly inescapable. The second problem is that it is not easy to provide. One may be tempted to think that perhaps this creative help is routine for programmers, but is simply hard to automate. After all, we did argue above that programmers argue informally the correctness of many theorems during their development process. However, the technical help required to make automated reasoning tools work is *not* of this nature. It turns out that one needs to know the *internals* of verification tools in order to provide this help. It has less to do with stating arguments in a proof and more to do with convincing the particular verification mechanism being used!

Consider, for example, the following well-studied problem in the literature: verification of programs with specifications written in a rich logic that uses recursively defined predicates. This problem is not recursively enumerable even for loop-free and function call-free basic blocks. In other words, it is impossible for any technique to work 100% of the time. As a result, builders of verification tools often implement heuristics that they believe may work in practice. This brings forth two issues. First, there almost always exist simple natural examples that are not solvable by incomplete heuristics [14]. Second and more importantly, when the heuristics simply give up on a problem and say "I don't know", an average user of the tool is hopelessly stuck! In contrast, an expert knows the heuristics employed by the particular verification tool and can creatively rewrite the problem or provide extra help in such a way that overcomes the limitation of the heuristics and allow the tool to succeed.

Note that the expert's intervention has no formal description: It is simply creative "magic"[1].

This inescapable requirement for crucial and technical creative help creates a **creativity gap** in automated verification.

SUMMARY OF CONTRIBUTIONS

In the sequel we develop novel theory, algorithms, and tools to tackle the problem of creativity gaps. At the broadest level, our work argues the following thesis statement:

> **The crucial reliance on expert creative help in automated reasoning can be effectively mitigated by (a) Precisely and formally identifying the power of automation strategies and heuristics— consequently identifying the role of expert help, and then (b) bridging the creativity gaps using logic learning.**

We now summarize our contributions[2] across three distinct themes explored in this work:

I. **Identifying Creativity Gaps:** The reason experts are forced to turn to their creativity is because verification tools working with incomplete logics automate their reasoning using heuristics. The first theme explores the argument that we need to understand the shortcomings of these heuristics to identify *how* to automate human help. The contributions of this part of the thesis are a set of foundational theoretical tools and results that precisely characterize the reasoning power of several popular heuristics used in program verification.

II. **Bridging Creativity Gaps using Learning:** The second theme explores a rather bold claim: the solution to automating the creativity gap lies in learning-based synthesis. Specifically, the work in this thesis approaches bridging the creativity gaps in automated reasoning using **data-driven logic learning**, which refers to learning logical formulas or expressions from examples. The contributions of this part are (a) a novel algorithmic framework for learning logical formulas from examples in the form of first-order models, (b) accompanying software artifacts that instantiate this framework to address two kinds of creativity gaps in automated reasoning (inductive lemma synthesis and axiom synthesis), and (c) rigorous evaluation of the artifacts which strongly argues the claim explored in this theme.

[1] We in fact study this particular problem in this thesis and provide new theoretical results on the limits of popular verification heuristics as well as a formal characterization of the role of expert creative help (see summary of contributions below).

[2] The contributions in this work have appeared across the following peer-reviewed publications co-authored by the author of this thesis, appearing as citations [15], [16], [17], and [18].

III. **Designing Reasoning Paradigms for the Future:** The final theme uses the (💡) + (⚙) lens on automated reasoning articulated above to explore the design of reasoning paradigms for the future. Specifically, the work on this theme seeks a program verification paradigm that balances the trade-off between the burden on automation and the creative burden on the user such that (a) the user is given a clear description up front of the creative help required from them— which they are able to work out purely logically— *independent* of the verification mechanism used by the automation, and (b) the automation simply confirms whether the provided creative help is sufficient to solve the verification problem at hand. In particular, the automation tool cannot say "I don't know". The contributions of this part are the definition of this new **Predictable Verification** paradigm (as opposed to paradigms with opaque creativity gaps that breed user frustration) and the theoretical development, implementation, and evaluation of a particular predictable verification framework for verifying heap-manipulating programs[3].

Author's Note on Scope for Impact: We believe that the position and contributions of this thesis have the potential to be truly transformational. The need for expert intervention to effectively utilize powerful automated reasoning often makes the field appear arcane to non-experts and outsiders, which is, unfortunately, in direct conflict with the objective of the field to provide mechanisms for producing correct software at large. Worse, the difficulty is sometimes mistaken for juvenescence— this author has heard the unfortunate remark that verification projects don't often "graduate beyond research labs" all too often.

This thesis takes an important first step towards viewing verification paradigms in a more comprehensive manner, *inclusive of the expert help* that makes them tick. We move the arcane expert help from the domain of an art to a rigorous science, showing that it can be studied formally. We also provide supporting evidence for how this awareness bears fruit: one can conceive algorithmic frameworks to automate outright the now formally characterized expert help (as we do in Chapter 3), or perhaps redesign the paradigms themselves so that the required expert help is a first-class concern and has formally predictable impact on the efficacy of automation (which we do in Chapter 4). Other works by this author that are not included in this thesis explore related ideas in this space: the work in [19, 20] explores the design of a logic for specifying properties of heap manipulating programs that is more amenable to automated verification than other contemporary logics for the same problem, and the work in [21] develops automated verification techniques for this logic that have completeness properties similar to the one we show in Chapter 2. The work in [22] identifies

[3]The publication associated with this theme [18] was recognized with the ACM Europe Best Paper Award: https://europe.acm.org/awards

a class of memory-safety specifications and a class of programs where the expressiveness of the class is traded for fully decidable automation with no extra user help; the work in [23] takes this even further and shows that such programs can be synthesized automatically from the limited specifications or even examples.

Finally, we note here the rich and exciting possibilities for advancing this agenda into the future. To design better frameworks for automated verification, we must tackle foundational questions about the power of reasoning algorithms and heuristics, build tools with better automation support that offer more than deductive reasoning (perhaps inductive reasoning based on examples, natural language, experience, etc.), investigate a wide variety of verification frameworks and programming paradigms (concurrency, distributed systems, neural networks, etc.) with an eye on offering a predictable verification experience for the user, and more generally ask ourselves what it takes to bridge the gap between intuitive arguments and formal proofs. We discuss these directions and the approaches the contributions of this thesis indicate for tackling them in Chapter 7.

Detailed Contributions

(A) Identifying Creativity Gaps in Instantiation-Based Verification Practical verification problems involve complex specifications that are stated in *incomplete* logics. This means that no technique can prove all valid programs, so developers of verification tools often end up building a set of complex and opaque heuristics. Consequently, when the heuristics fail, users do not know *why* they fail.

Motivated by this observation, we study a popular heuristic for verifying functional programs called UQFR: Unfolding definitions followed by Quantifier-Free Reasoning (usually performed by SMT solvers). This is a very effective heuristic that has been used in many verification tools [24, 25, 26] over the years. However, despite its efficacy, UQFR fails on very simple problems and users do not understand why this happens.

In Chapter 2, we identify a new logic called FLUID (First-Order Logic Under Inductive Definitions) which provides a sound abstraction of the verification problems posed by users. We find that UQFR is complete for FLUID, i.e., UQFR works *precisely* when the FLUID abstraction is provable, and fails when the abstraction is too weak. Further, we show that when UQFR fails, there always exist spurious counterexamples to the FLUID abstraction called rogue nonstandard models. This shows that the role of human creative help (💡) in aiding tools is to provide hints (in the form of inductive lemmas) that eliminate such spurious counterexamples.

(B) Filling the Creativity Gap: Synthesizing Lemmas for Heap Verification The second major thrust of this thesis is the automation of human creative effort in verification. In this direction, we address the creativity gap that arises when reasoning using UQFR in the context of imperative programs over dynamic heaps [15]. Verification problems in this setting are stated in a very general and powerful logic: First-Order Logic with Recursive Definitions. Prior results in literature [27] on identifying creativity gaps for this problem show results analogous to those shown in Chapter 2, namely that the role of human help is *precisely* the creative task of providing inductive lemmas that bridge the gaps left by the UQFR heuristic. In other words, the verification problem splits into two precise pieces: (1) creatively (💡) come up with inductive lemmas such that (2) UQFR can mechanically (⚙) verify the given program using the lemmas. This motivates the automation of inductive lemma synthesis.

We develop in Chapter 3 a new data-driven logic learning framework called Model-Guided Synthesis that is based on learning quantified first-order formulas from novel counterexample first-order models. Our approach solves several important challenges along the way: we formalize for the first time the idea of counterexample models and create several new kinds of counterexamples beyond the simple positive and negative examples known in the literature. Most importantly, unlike synthesis problems in literature, the salient feature of this creative task is that the synthesis problem has no logical specification! Model-Guided Synthesis handles this by using several different carefully designed algorithmic components in concert with a logic learner to gradually *elicit* valid and useful lemmas. This chapter also contributes a software artifact 'FOSSIL' (First-Order Solver with Synthesis of Inductive Lemmas) for automatically synthesizing inductive lemmas using Model-Guided Synthesis, along with a discussion of its implementation and its evaluation on a suite of verification problems that require inductive lemma synthesis.

(C) Rethinking Verification Paradigms: Predictable Verification using Intrinsic Specifications In the above discourse we primarily used the two-part (💡) + (⚙) lens to frame existing automated reasoning paradigms. However, the lens also embodies the idea that to design reasoning paradigms that are of low cognitive burden for users it is important to explicitly model the role of the user and study it as rigorously as we study the mechanical parts of automated reasoning. This interpretation then raises the following question: what are the trade-offs between the creative and mechanical parts of various reasoning tasks?

In Chapter 4 we make a preliminary incursion into this space by defining the idea of *Predictable Verification*. A predictable program verification paradigm is one where (1) the

user is asked up front to provide a fixed set of annotations (i.e., the creative help) that are *independent* of the underlying verification mechanism, and (2) the verification, given the annotations, can be completely automated. In other words, we want *no more frustrating proof engineering* (e.g., inductive lemmas, instantiation triggers, etc.)! This would yield a framework where the user simply expresses arguments for program correctness[4], and then an automated engine checks their work effectively.

Specifically, we study the problem of verifying programs manipulating heap datastructures, which the reader may remember is relevant to our study in Chapters 2 and 3. In these earlier chapters we formally understood the limitations of unrolling based heuristics in dealing with specifications involving recursive definitions, and used learning-based synthesis to bridge the creative task of coming up with inductive lemmas. In Chapter 4 we eschew this framework entirely, introducing the idea of *Intrinsically Defined Datastructures* and study the problem of verifying programs against specifications involving intrinsically defined datastructures as opposed to recursively defined ones. Intuitively, an intrinsic specification says what a program state looks like when one views it from the 'inside' and is therefore an inherently local definition, whereas recursive definitions offer a 'global' view of the state. This has several useful consequences. Crucially, it turns out that we can model the creative help (💡) required to verify a program against intrinsic specifications as a set of simple, local arguments that talk about how the local view of the state changes as the program modifies the state.

We consequently develop a novel verification methodology based on augmenting programs with creative help in the form of *ghost code* (code which does not execute but provides a proof of the analyzed program in a computational way). We show that the mechanical task (⚙) of verifying programs against intrinsic specifications given the creative help is *decidable*. This then ensures a predictable verification experience in practice, as the decision problem can automated effectively using SMT solvers [11, 12]. We empirically study the expressiveness of intrinsic definitions for several common classes of heap datastructures and evaluate the verification methodology on a suite of programs manipulating the datastructures. In particular, our verification of an overlaid datastructure consisting of a linked list and a binary search tree (see Section 4.7.5) is a case study that offers compelling evidence towards the efficacy of the contributions.

(D) Filling Creativity Gaps by Discovering Laws One of the most beautiful aspects of human intelligence is our ability to formulate abstract rules for navigating new domains. This is especially prominent in mathematical and scientific inquiry, where the scientist may

[4]Given the right language, it may even be enough to state the arguments at a high level. This is a wonderful avenue for further research, which we leave to the future.

even understand the phenomenon at hand precisely, but they want to analyze it at a higher level of abstraction and draw interesting conclusions by operating within the abstraction. For example, doctors have a detailed understanding of the human lung and its dynamics, but when analyzing X-rays of lungs they talk about 'masses' or 'honeycombing' in the images and use these concepts in their diagnosis. Automating this creative aspect of reasoning encompasses a great many technical problems such as acquiring relevant concepts, forming compositional abstractions, learning to reason within the abstract space, and continually learning better abstractions with experience.

Unsurprisingly, the problem of synthesizing axioms is also of great relevance to program verification. We study in Chapter 5 the problem of axiom synthesis for reasoning domains in programming languages like Kleene Algebras. Although the problem of finding axiomatizations is extremely old, we define in this part of the thesis a new formulation of axiom synthesis as a *computational problem*. This definition opens up the use of computational tools for finding axioms in many complex domains, including those where the objects of study may not even have a logical specification. Since the domain under axiomatization may not have a clear logical description, the associated problem of synthesizing axioms has no clear logical description either! We hence develop Learning-based Axiom Synthesis (LAS), a variant of the Model-Guided synthesis framework developed in Chapter 3, for automatically synthesizing axioms. We then we use the framework to automatically synthesize axioms for modal logics and Kleene algebras, finding axioms that were hitherto only known to have been formulated by expert logicians.

Outline The remainder of this thesis is organized as follows. We present the FLUID logic for characterizing creativity gaps in Chapter 2. We then present the Model-Guided Synthesis technique for bridging creativity gaps automatically using logic learning in Chapter 3, and use it to automate the creative task of coming up with inductive lemmas. In Chapter 4 we introduce intrinsically defined datastructures and explore the idea of designing a verification paradigm that offers predictable verification. We then return to the automation of another creative task in Chapter 5, where we use model-guided synthesis to learn axiomatizations. In Chapter 6 we reflect on the design choices we make in our work and discuss the connections to related work. Chapter 7 concludes with a few hopeful thoughts on potential future directions.

Chapter 2: Complete FO Reasoning for Properties of Functional Programs

Several practical tools for automatically verifying functional programs (e.g., LIQUID HASKELL and LEON for Scala programs) rely on a heuristic based on unrolling recursive function definitions followed by quantifier-free reasoning using SMT solvers. In this chapter we uncover foundational theoretical properties of this heuristic, revealing that it can be generalized and formalized as a technique that is in fact *complete* for reasoning with combined First-Order theories of algebraic datatypes and background theories, where background theories support decidable quantifier-free reasoning. The theory developed in this chapter explains the efficacy of these heuristics when they succeed, explain why they fail when they fail, and the precise role that user help plays in making proofs succeed.

2.1 INTRODUCTION

The automation of program verification has been revolutionized with the advent of efficient *logic engines* that check validity of logical formulas over various theories that capture domains that programs work with (arithmetic, strings, arrays, algebraic datatypes, pointer-based heaps, etc.). In particular, *quantifier-free logics* over various theories admit decidable validity checking, and further, permit decision procedures for the combination of theories (Nelson-Oppen style combinations) that have been realized by efficient DPLL(T)-based SMT solvers [11, 28, 29, 30].

However, automation's grip becomes tenuous when it comes to the verification of first-order properties of *functional programs* over *algebraic data types* (ADTs) such as lists or trees over basic types like integers. Functional programs over ADTs can be expressed mathematically using a set of *recursively defined functions* over types. Programs hence can be expressed as a set of first-order definitions of functions *Defs* that are *universally quantified* over their inputs. The goal of verification, then, is to determine whether a particular FO (First-Order) theorem T involving these defined (or interpreted) functions is mathematically valid under a set of definitions *Defs*.

Automation is Impossible in Theory Even though the theorem T that needs to be validated is universally quantified (and hence can be seen as a quantifier-free formula), reasoning about the validity of T under *interpreted definitions Defs* is extremely hard. The validity problem is not decidable (while validity of T under *uninterpreted functions is* typically

The material in this chapter is reproduced from the publication cited as [17] co-authored by the author of this thesis, with minor changes.

decidable). Worse, the problem is not even recursively enumerable (there is no complete proof system nor a semi-decision procedure that is guaranteed to terminate on at least all valid theorems). A simple proof of this fact is that we can define addition and multiplication as defined (interpreted) functions using recursion, and use universal quantification to specify the neither decidable nor recursively enumerable problem of determining the *non-existence* of solutions to Diophantine equations [31].

Automation is Effective in Practice Despite the above hardness, there has been significant progress in systems that provide varying degrees of automation to the process of verifying such theorems. Liquid Haskell (LH) [32] and Leon/Stainless [26, 33] both exploit the automation provided by logic engines for decidable *quantifier-free reasoning* (i.e., SMT solvers) to prove FO theorems. Extrinsic-style verification in LH reduces checking quantifier-free (implicitly universal) properties of functions over ADTs to proving pre- and post-condition contracts that assert those properties in the code ("proofs") written by the verification engineer [1]. The Leon verifier [33] (as well as its successor Stainless [26]) uses a similar style of reasoning for Scala programs with quantifier-free contracts, where the contracts themselves are written using recursively defined pure Scala functions. Leon verifies each function's contract by compiling the body of the function to a *verification condition* (VC), modeling functions called in the body using defined functions and assuming they satisfy their contracts [2]. While LH and Leon provide different mechanisms for users to prove properties via induction and auxiliary lemmas, we observe that they share a common fundamental interface to logic engines: verification is reduced to proving VCs of the form $Defs \rightarrow \varphi$ where φ is universally quantified. LH treats functions defined in *Defs* as *uninterpreted* using a heuristic called *logical evaluation* that finitely unfolds the definitions for terms that appear in φ. Leon's strategy is also to unfold the recursive definitions based on function applications that occur in φ. However, it differs from LH in that it does this *recursively*, unfolding definitions iteratively for larger and larger depths and assuming that such unfolded calls to functions satisfy their contract.

To summarize, both tools automate verification via logical engines by (1) generating VCs of the form $Defs \rightarrow \varphi$ (where φ is a universally quantified FO formula), (2) treating all defined functions as largely uninterpreted, (3) instantiating definitions repeatedly only on certain terms, and (4) dispatching them to an SMT solver that does quantifier-free decidable

[1] LH implements various algorithms including refinement inference. In this work, when we say LH, we refer specifically to extrinsic-style full functional correctness proofs over user-defined functions using methods proposed in [32].

[2] This is an inductive proof (induction on the size of the implicit call stack) that all functions satisfy their contracts.

reasoning. This technique is certainly sound but clearly not a decision procedure: LH just makes a fixed set of instantiations, which may be insufficient; Leon can continuously unfold definitions and may proceed forever (timeout). Yet, despite the hardness results, this heuristic works well in practice, giving predictable results though they may require users to find new inductive lemmas and guidance in proofs!

Why does the heuristic of unfolding recursive definitions followed by quantifier-free reasoning work so well in practice? In this chapter, we establish foundational results that this procedure is in fact a *complete* procedure for the underlying combination of first-order theories. Our results not only explain *when* this heuristic method works well, but also explains when and why they *fail*, and the role of further help asked of the user.

The Standard Model vs Combined Theories

The answer to this question lies in the tension between the theory of the *standard model* and the *combined FO theory* of the various sorts. First-order theorems that express properties of functional programs can be seen as formulas over a *combination of sorts*, in particular sorts that refer to ADTs (e.g. trees) and the base sorts (e.g. integers) that the datatypes are built upon. When a verification engineer wishes to prove a theorem, they want it to be proven for the *fixed* universe (the standard model) consisting of the various sorts. In this universe, the ADT sort is the natural universe of *algebraic terms* of the appropriate type, with constructors and destructors interpreted in the standard manner, and the integer sort and functions over them (e.g. +) are interpreted in the standard manner.

Axiomatized Models In first-order logic, however, we often reason with models that are *axiomatized*: we capture various properties of models using a finite or recursive set of axioms, and reason over *any* model that satisfies the axioms. In particular, ADTs can be axiomatized and the universe of integers with addition can be axiomatized. In fact, they can be individually given *complete axiomatizations*— i.e., all models satisfying the axioms satisfy the same first-order theorems as the standard model [34, 35, 36, 37, 38, 39]. There may be other models, called *nonstandard models*, that are not isomorphic to the standard model (in fact they always exist, say, by the Löwenheim-Skolem Theorem) but one cannot distinguish them using a first-order formula. Nonstandard models are well-known in the literature [36, 40].

Rogue Nonstandard Models that Disagree with the Standard Model However, when we combine universes and their theories, interfacing them with uninterpreted functions,

the combined axioms are no longer powerful enough. More precisely, it is well known that the combined axioms can admit *rogue* nonstandard models that disagree with standard models on first order expressible theorems, and hence the theory entailed by the combined axioms becomes weaker. Rogue nonstandard models are a special case of nonstandard models of the combined theory that disagree with the standard model on some first-order formulas. For example, if we take the complete axiomatization of ADTs and the complete axiomatization of uninterpreted functions (congruence axioms) and combine them, the union of the axioms admits rogue nonstandard models of ADTs that contradict theorems true in the standard model of ADTs with uninterpreted functions.

Nonstandard models exist even for complete theories, but there are no rogue models in such theories since, by the definition of completeness, all nonstandard models agree with the standard model on all FO expressible theorems. Combined theories are incomplete as there are rogue nonstandard models. However, *quantifier-free* formulas over combinations of theories (using the Nelson-Oppen method) do not suffer from such issues, which is why we can think of validity procedures for them as provers for the standard model.

Contributions The key insight behind our result is that the method of unfolding recursive definitions and performing quantifier-free reasoning can *always* prove and *only* prove the subset of theorems that are valid over the *combined theory of ADTs and the background sorts*. Consequently, it *cannot* prove theorems that are valid in the standard model but invalid in a rogue nonstandard model. We develop this insight to explain the unusual effectiveness of unfolding recursive definitions into uninterpreted function applications via four contributions:

1. A FLUID Logic (Section 2.4) Our first contribution is the definition of a logic called FLUID (First-order Logic for Universal properties under Inductive Definitions) that captures the essence of definitions[3] and VCs generated by LH[4] and Leon. FLUID formulas are of the form $Defs \rightarrow \varphi$ where *Defs* are *provably terminating* recursive definitions, and φ is a universally quantified formula. Verification conditions for correctness of many functional programs can be formulated using FLUID formulas; in fact, systems like LH and Leon generate VCs that are in this logical fragment.

2. Completeness of Unfolding followed by Quantifier-Free Reasoning (UQFR) (Section 2.5) UQFR is a technique for proving validity by modeling recursive functions

[3]LH and Leon also support higher-order functions, but such definitions are beyond the scope of this chapter. We provide further discussion on higher-order functions in Chapter 7.

[4]In consultation with the developers of LH, we believe that FLUID captures all VCs generated by LH for extrinsic style proofs using refinement reflection!

as uninterpreted functions, unfolding recursive definitions *Defs* systematically on a class of ground terms, and reasoning with the resulting quantifier-free formulae using decision procedures. Our second contribution is a foundational result that shows that UQFR is a *complete* semi-decision procedure for the validity of FLUID formulas over the combined first-order theory of ADTs and background sorts. Namely, UQFR guarantees to prove all theorems that are valid in the combined FO theory. Consequently, when a theorem that is valid over the standard model is *not* proven using this technique, we are guaranteed that there is a rogue nonstandard model (satisfying the ADT and background theories) where the theorem does not hold. The proof of completeness is nontrivial for two reasons. First, the unfoldings of recursive definitions that UQFR uses (and tools such as LH and LEON use) are *thrifty*; they instantiate definitions of functions only on terms on which they are called, and do not expand instantiations to terms that arise from the underlying axiomatizations of theories. Second, every time a theorem that is valid on the standard model is *not* proven, it is nontrivial to construct a rogue nonstandard model falsifying the theorem. The model construction in the proof of this theorem crucially exploits the fact that FLUID definitions are provably terminating.

3. Completeness in Practice (Sections 2.6 and 2.7) Thus, far from being a whimsical heuristic that happens to work in practice, UQFR is rather a robust procedure whose completeness may explain why this heuristic performs so predictably well. In particular, it does not miss proving theorems that can be proved using pure FO reasoning of the underlying axioms of the theories. Our third contribution shows how this bears out in practice. We explain how LH performs FLUID verification using UQFR (Section 2.6). Crucially, when theorems are not proved valid, we show it is because rogue nonstandard models exist, and that the lemmas and induction hints provided by the user then serve to eliminate such models, all while reasoning within the FLUID fragment. Next, we show how we can use a slightly different FLUID formula to mimic LEON's more sophisticated reasoning which additionally assumes pre/post contracts for functions at each unfolding. Hence, our completeness result also applies to explain the effectiveness of LEON (Section 2.7).

4. Limits of FLUID (Section 2.8) Our final contribution is a set of results that show why our results on FLUID are unlikely to extend to more expressive logics. We show though the validity problem for FLUID admits complete procedures, it is *undecidable*, hence distinguishing it from several decidable fragments identified in the literature (e.g., [41]). We also show that attempts to generalize FLUID, e.g. by allowing functions whose definitions are required to be terminating (but not *provably* terminating using FO proofs) makes UQFR

not complete. This result also implies that replacing definitions with arbitrary universally quantified formulas makes UQFR an incomplete procedure.

2.2 OVERVIEW

In this section we provide an overview of the key results in this chapter. We illustrate the ideas via example programs over the datatype of lists over integers:

```
data List = Nil | Cons Int List
```

2.2.1 Insertion and Sortedness

Consider the following program that inserts an element into a (sorted) list. We define both `insert`-ion and `sorted`-ness via recursive functions

```
sorted :: List → Bool
sorted Nil                    = True
sorted (Cons h Nil)           = True
sorted (Cons h1 (Cons h2 tl)) = h1 ≤ h2 && sorted (Cons h2 tl)

insert :: List → Int → List
insert Nil k                      = Cons k Nil
insert (Cons x xs) k | x >= k     = Cons k (Cons x xs)
                     | otherwise  = Cons x (insert xs k)
```

Definitions We can encode the above Haskell programs in FOL where each function's definition introduces no new variables, instead using destructors $(head, tail)$ and recognizers $(isNil, isCons)$ to simulate pattern matching. To ensure that destructors are applied sensibly, we *guard* the use of terms of the form $head(t)$ and $tail(t)$ with the recognizer $isCons(t)$.

$$\begin{aligned}
\forall x.\mathsf{List}.\ sorted(x) = \ & \mathsf{ite}(isNil(x), True, \\
& \mathsf{ite}(isNil(tail(x)), True, head(x) \leq head(tail(x)) \wedge sorted(tail(x))))\\
\forall x : \mathsf{List},\ k : \mathsf{Int}.\ insert(x,k) = \ & \mathsf{ite}(isNil(x), Cons(k, Nil), \\
& \mathsf{ite}(head(x) \geq k, Cons(k,x), \\
& Cons(head(x), insert(tail(x), k))))
\end{aligned} \tag{2.1}$$

where we treat *sorted* and *insert* as uninterpreted functions in the signature. We refer to these formulae as the *definitions* of *sorted* and *insert* and denote them by def_{sorted} and def_{insert} respectively.

Verification Conditions Let us consider the example of verifying that inserting an element k into the empty list yields a sorted list. We state this formally as the following *verification condition* (VC):

$$(def_{sorted} \wedge def_{insert}) \rightarrow sorted(insert(Nil, k)) \tag{2.2}$$

Note that this VC is of the form $DEF \rightarrow \varphi$, where DEF is a set of definitions (when it appears in a formula we are referring to the conjunction of the formulas in the set) and φ is quantifier-free, i.e., all variables are implicitly universally quantified. Informally, the VC says that the property φ should hold *assuming* the set of definitions DEF. The FLUID fragment we define (see Section 2.4) consists of such formulas.

Unfolding We prove the above VC valid by *unfolding* the definitions. For a term t, let $def_{sorted}[t]$ denote the quantifier-free formula obtained by instantiating the quantified variable x in def_{sorted} with t. We refer to this as unfolding the definition of *sorted* on t. Similarly we can define the unfolding $insert[\overline{t}]$. To prove the VC valid we simply unfold definitions on arguments that occur in φ, i.e., we attempt to prove

$$(def_{insert}[(Nil, k)] \wedge def_{sorted}[insert(Nil, k)]) \rightarrow sorted(insert(Nil, k)) \tag{2.3}$$

This formula can be dispatched using SMT solvers [11, 12] that use a combination of decision procedures for ADTs and Integers. It is in fact valid because unfolding def_{insert} on (Nil, k) shows that $insert(Nil, k)$ evaluates to $Cons(k, Nil)$, and unfolding def_{sorted} on $Cons(k, Nil)$ shows that $sorted(insert(Nil, k))$ evaluates to $True$.

We generalize this technique of Unfolding definitions followed by Quantifier-Free Reasoning into an algorithm UQFR (Section 2.5), and argue that tools like Liquid Haskell (Section 2.6) and Leon (Section 2.7) perform similar reasoning on such formulas.

2.2.2 Insertion Preserves Sortedness

Next, let us turn to a more interesting theorem, namely that insertion preserves sortedness. Formally, we wish to prove the following *contract* for insertion:

$$\forall x, k. sorted(x) \rightarrow sorted(insert(x, k)) \tag{2.4}$$

The corresponding VC is:

$$VC_{simple} \equiv (def_{sorted} \wedge def_{insert}) \rightarrow (sorted(x) \rightarrow sorted(insert(x, k))) \tag{2.5}$$

Unlike the example in Section 2.2.1, it turns out that there is no set of terms such that unfolding the definitions on these terms can prove the VC valid. Consequently, LH fails to prove the theorem.

Using Contracts Tools like LEON not only unfold definitions but also use contracts for terms generated during unfolding. For example, note that unfolding def_{insert} on (x, k) yields the term $insert(tail(x), k)$. Then, the VC that LEON attempts to prove is not VC_{simple} but rather the following, which we call VC_{LEON}:

$$(def_{sorted} \wedge def_{insert}) \rightarrow \big((x \neq Nil \rightarrow (sorted(tail(x)) \rightarrow sorted(insert(tail(x), k)))) \\ \rightarrow (sorted(x) \rightarrow sorted(insert(x, k))) \big) \tag{2.6}$$

which additionally assumes the contract for $insert(tail(x), k)$ (when $x \neq Nil$). Observe that this VC is also of the form $DEF \rightarrow \varphi$ and is therefore in the FLUID fragment. We show in Section 2.7 that VC_{LEON} can be obtained automatically from the original VC, i.e., VC_{simple}.

We attempt to prove VC_{LEON} using the same technique of unfolding definitions on arguments appearing in the formula (UQFR). This succeeds, and one can verify that unfolding *insert* on $\{(x, k), (tail(x), k)\}$ and *sorted* on $\{x, tail(x), insert(x, k), insert(tail(x), k)\}$ proves VC_{LEON} valid[5]. Observe that the unfolding strategy used in the examples we have seen is *thrifty* in the sense that definitions are unfolded *exactly* on terms that occur as arguments to the corresponding functions.

Using contracts is a more powerful approach. In general, there are theorems whose proofs require even more instantiations of contracts on terms obtained during further unfoldings. We show in Section 2.7 using a reduction that the use of multiple repeated instantiations of definitions as well as contracts can also be viewed as proving FLUID fragment formulas using UQFR. Consequently, our results apply not only to LH but also to tools like LEON.

2.2.3 Membership in a Sorted List

One prominent aspect of program verification in LH or LEON is proof by induction. However, induction is *not* part of UQFR. In this section we discuss the example of checking membership in a sorted list where all the above approaches fail, and explain the role of induction (in the form of explicit user help) in these tools. We first define a function *elems* to capture the set of elements stored in a list and a function *mem* that checks the membership of an element in a sorted list.

$$\forall x : \mathsf{List}.\ elems(x) = \mathsf{ite}(isNil(x), \emptyset, \{head(x)\} \cup elems(tail(x)))$$

$$\forall x : \mathsf{List},\ k : \mathsf{Int}.\ mem(x, k) = \mathsf{ite}(isNil(x), False,\ \mathsf{ite}(k = head(x), True, \\ \mathsf{ite}(k < head(x), False,\ mem(tail(x), k)))) \tag{2.7}$$

[5]The reader may note here that we only argue the validity of VC_{LEON} and not the original goal VC_{simple}. We discuss why validity of the former implies validity of the latter in Section 2.7.

We want to verify that *mem* precisely captures membership for sorted lists. Formally, the contract is: $sorted(x) \rightarrow (mem(x, k) \leftrightarrow k \in elems(x))$

However, the approaches discussed above do not work for this example. They do not succeed even if the definitions are unfolded infinitely and contracts is assumed for all of the infinitely many terms/tuples that occur in the unfoldings.

To see why this is the case, consider what happens when we replace the usual *standard* model of ADTs and Integers we have in our minds with complete axiomatizations for each of the sorts, along with congruence axioms for the function symbols *elems* and *mem*. In this setting, the standard model is only one of the possible models and in general a model of the axiomatized universes may not be identical to the standard model. Validity in the axiomatized setting is an under-approximation to validity in the standard model, and as we show in Section 2.3.3 it is in fact a strict under-approximation. There are theorems that are true on the standard model that do not hold under axiomatization. This is because of the presence of *rogue nonstandard models* where the property we want to prove is not true. A rogue nonstandard model is a model that obeys the axioms but is not identical to the standard model, and further, falsifies the desired theorem. Nonstandard models always exist in the axiomatized setting, but they may satisfy all the same first-order properties as the standard model using a first-order formula. However, *rogue* nonstandard models, when they exist, can disagree with the standard model on a desired first-order theorem.

Our soundness and completeness results in Section 2.5 show that the proving power of unfolding definitions and using contracts is *precisely* that of validity over the axiomatized universe. Therefore, if there is a rogue nonstandard model that falsifies a property, then unfolding based reasoning *cannot* prove it. Indeed, both LH and Leon fail on the above example without extra help.

Rogue Nonstandard Model Let us look at the rogue nonstandard model where our theorem does not hold. The universe U is: $\{s \mid s \text{ is a finite sequence of ints}\} \cup \{(s, i) \mid s \text{ is an infinite sequence of ints}, i \text{ is an int}\}$ The finite sequences correspond to ADT lists of integers as we think of them but the infinite sequences are *nonstandard elements*[6]. *Nil* is interpreted to be the empty sequence and *Cons* behaves as expected on standard elements (prepending an element to a finite sequence). On the nonstandard elements *Cons* is defined by $Cons(j, (s, i)) = (j :: s, i + 1)$ where $j :: s$ denotes prepending j to the sequence s. *head* and *tail* behave as inverses to *Cons* in the usual sense. One can check easily that this model

[6]The standard model of ADTs consists exactly of all terms. *Nonstandard elements* are elements in a nonstandard model that do not correspond to any term. In particular, one cannot destruct them a finite number of times to reach *Nil*. Nonstandard models always have such elements with "infinite tails".

satisfies the usual axioms of ADTs [36, 37].

The meaning of *sorted* on this model is as expected: we define only elements with non-decreasing sequences to be sorted. The definition of $elems(x)$ is as follows

$$elems(x) = \begin{cases} \{v \mid v \text{ is an element of } x\} & \text{for a standard element } x \\ \{v \mid v \text{ is an element of } x\} \cup \{-1\} & \text{for a nonstandard element } (x, i) \end{cases} \tag{2.8}$$

Lastly $mem(x, k)$ holds if and only if k occurs in the longest non-decreasing prefix of the sequence corresponding to x. If x is sorted, the longest non-decreasing prefix of x is x itself.

The above interpretations are consistent with the definitions. Consider the function for *elems*, for example. On standard elements it is consistent with the definition because it has the expected value. It is also consistent on nonstandard elements. Observe that for a nonstandard element x, $tail(x)$ is also a nonstandard element. Therefore, the inclusion of an extraneous element -1 in the *elems* of both x and $tail(x)$ is consistent with the recursion $elems(x) = \{head(x)\} \cup elems(tail(x))$.

Finally, we see this is a rogue nonstandard model as it does not satisfy the property $sorted(x) \rightarrow (mem(x, k) \leftrightarrow k \in elems(x))$. Consider the nonstandard element $x = ([0, 0, 0 \ldots], 0)$. Note that x is sorted since it is a non-decreasing sequence, and $elems(x) = \{0, -1\}$ by the above construction. Hence $-1 \in elems(x)$. However $mem(x, -1) = False$ since -1 does not occur in x.

Role of User Help To prove the above example in LH, one must provide additional hints or *inductive lemmas* (whose proof of the induction step is itself performed using unfolding/UQFR)[7]. We show in Section 2.6 that these lemmas eliminate rogue nonstandard models like the one shown above, and therefore enable the VC to be proven using unfolding techniques.

Rogue Nonstandard Models of Integers It is tempting to think that the above difficulties can be avoided by stating a constraint that lists are finite, i.e., there must exist a non-negative integer corresponding to the length. However, this does not work. This is because there exist *rogue nonstandard models of the integers* containing elements considered 'non-negative' by the model's interpretation but do not correspond to an integer (i.e., decrements do not reach 0). The lengths of infinite lists would be interpreted to such nonstandard numbers, and we would still need user help.

[7] LEON is able to verify *mem*, but it does so using a heuristic for *structural induction* rather than its primary algorithm of instantiating definitions and contracts. There are other examples involving list reversal where LEON also requires lemmas to deal with rogue nonstandard models that fail the theorem (see Section 2.7).

2.3 PRELIMINARIES

In this section we define the general setting of multi-sorted first-order logic over algebraic datatypes (ADTs) and other base types with recursively defined functions. We define FLUID, our logic of study, as a fragment of this logic in Section 2.4.

2.3.1 Syntax and Semantics

The logic we work with is defined over a finite set of disjoint nonempty sorts $\mathcal{S}$. We distinguish certain sorts among these as *foreground* sorts. The foreground sorts support a signature of Algebraic Datatypes (ADTs) which we describe below. The other sorts are referred to as *background* sorts (background sorts could also consist of ADTs).

An ADT signature for a sort σ consists of a finite set of function symbols $ctor_i, 1 \leq i \leq m$ called *constructors*. Each constructor has an arity $r_i \geq 0$ and a signature $\sigma_1 \times \sigma_2 \times \ldots \times \sigma_{r_i} \to \sigma$, where $\sigma_j, \sigma \in \mathcal{S}$. Corresponding to each constructor with the above signature, we also have r_i many *destructors* $dtor_{ij}$ with signature $\sigma \to \sigma_j$ for $1 \leq j \leq r_i$, and *recognizers* is_ctor_i with signature $\sigma \to Bool$.

For example, the algebraic datatype of lists over natural numbers *ListNat* is defined by the nullary constructor $nil : ListNat$ and the binary constructor $Cons : Nat \times ListNat \to ListNat$, with destructors are $head : ListNat \to Nat$ and $tail : ListNat \to ListNat$. The recognizer is_cons identifies elements that correspond to non-nil lists. Note that standard pattern matching idioms for ADTs used in functional programs can be expressed using this vocabulary.

We can also define hierarchical datatypes (e.g., lists of lists of integers), mutually recursive datatypes (e.g., terms corresponding to a context-free grammar), as well as sum (unions) and product types (tuples). We cannot define co-inductive datatypes such as infinite lists in our logic. However, we do not lose generality with respect to the various tools we study; for example, LH's termination checker precludes the creation of values like infinite lists.

Our logics have signatures of the form $\Sigma = (\mathcal{S}, \mathcal{F}, \mathcal{D})$, where:

- $\mathcal{S}$ is a finite non-empty set of sorts as defined above with a partitioning of sorts into a set of foreground ADT sorts and a set of background sorts. We require that there is at least one foreground sort.

- $\mathcal{F}$ is a set of constant, function, and relation symbols over the sorts $\mathcal{S}$. These will be used to model symbols over the sorts that models give interpretations to. These include functions like integer addition or set union, and constructors, destructors, and recognizers over ADT sorts.

- $\mathcal{D}$ is a set of function symbols distinct from $\mathcal{F}$ that will be used to model functions that have *definitions*.

The syntax is standard multi-sorted first-order logic over sorts $\mathcal{S}$ and over symbols $\mathcal{F} \cup \mathcal{D}$. We make two modifications. First, we require that every occurrence of a destructor term $dtor_{ij}(t)$ is *guarded* by the corresponding recognizer $is\_ctor_i(t)$ to ensure that destructor terms are well-defined. We do not lose generality as any formula with well-defined destructor terms can be rewritten to an equivalent one with the appropriate guards. In practice, tools check that $is\_ctor_i(t)$ holds by generating a separate Verification Condition. Second, we allow ite ($if-then-else$) expressions over terms and formulas. The semantics of our formulas is the standard one for FOL. We refer the reader to a standard reference text [42] for the notion of first-order logic, first-order models, syntax, and semantics. Semantics is defined in terms of models (aka structures) that give interpretation to all symbols, including those in $\mathcal{D}$. We use the notation $M \models \varphi$ to denote that a sentence φ evaluates to *true* in a model M, and $\varphi \models \psi$ to denote semantic entailment (all models satisfying φ also satisfy ψ).

Inductive Definitions Intuitively, a *definition* of D (for $D \in \mathcal{D}$) gives a particular interpretation for D. The definition of a function $D \in \mathcal{D}$ of arity r is a quantified formula def_D of the form

$$\forall x_1, x_2, \ldots, x_r.\, D(x_1, x_2, \ldots, x_r) = \rho(x_1, x_2, \ldots, x_r) \tag{2.9}$$

where ρ is a quantifier-free formula over x_1 through x_r called the *body* of the definition. Of course, the body may use other inductively defined symbols $G \in \mathcal{D}$. We require that every function in $\mathcal{D}$ has exactly one definition.

In order to obtain well-defined definitions, we demand a notion of termination. We define this notion using the *standard model* of our logic, which we introduce in the next section.

2.3.2 The Standard Model

The intended standard interpretation of an ADT signature is the initial term algebra where the universe consists of terms that respect the sorts and the interpretation of constructors is that of term application, i.e., $[\![ctor_i]\!](e_1, \ldots, e_{r_i}) = ctor_i(e_1, \ldots, e_{r_i})$.

The destructors are interpreted as $[\![dtor_{ij}]\!](ctor_i(e_1, \ldots, e_{r_i})) = e_j$ and is otherwise interpreted to be identity on other elements [8]. Finally, recognizers are only true on terms constructed with the corresponding constructor: $[\![is\_ctor_i]\!](ctor_i(e_1, e_2, \ldots e_{r_i})) = True$, and is *False* for other elements.

[8] Since we consider only formulas that are guarded to check elements to be of the right sort before applying destructors, the semantics of the formula on other elements is irrelevant.

More generally, our logic is parameterized by a *standard model* $\mathcal{M}_{\mathcal{S},\mathcal{F}}$ of the foreground and background sorts. This is typically true of sorts employed in program verification: ADTs, integers, sets, etc. Note that this model does not give interpretations to functions in $\mathcal{D}$.

We require that inductive definitions are terminating on the standard model using a standard *eager* semantics [43]. Informally, we evaluate a definition on concrete elements over the standard model as follows: (i) we evaluate recursively defined function terms by evaluating the definition on the arguments; (ii) for ite expressions, we evaluate the conditions first and then only evaluate the appropriate branch; (iii) for all other expressions, we first evaluate all recursively defined function terms (with subterms evaluated before their superterms) and then evaluate the expression. A terminating definition is one for which this procedure terminates on all inputs.

The following proposition states that over $\mathcal{M}_{\mathcal{S},\mathcal{F}}$, there exists a unique valuation for the defined functions $\mathcal{D}$ that is consistent with their definition.

Proposition 2.1. Given $(\mathcal{S}, \mathcal{F}, \mathcal{D})$ let $DEF = \{def_D \mid D \in \mathcal{D}\}$ be a set of definitions. There exists a unique model $\mathcal{M}_{\mathcal{S},\mathcal{F},\mathcal{D}}$ such that the interpretation of symbols in $\mathcal{F}$ coincides with $\mathcal{M}_{\mathcal{S},\mathcal{F}}$ and interpretations of symbols in $\mathcal{D}$ satisfy their definitions, i.e., $\mathcal{M}_{\mathcal{S},\mathcal{F},\mathcal{D}} \models def_D$ for every $D \in \mathcal{D}$.

Functional programs can be modeled using the definitions (we only consider terminating programs, of course). Universal FO properties φ of functional programs can be modeled as validity of formulas of the form $DEF \rightarrow \varphi$, where we use DEF to mean the conjunction of formulas in the set of definitions $DEF = \{def_D \mid d \in \mathcal{D}\}$.

However, as discussed in Section 2.1 it is easy to show that the problem of validity of even quantifier-free formulas on $M_{\mathcal{S},\mathcal{F},\mathcal{D}}$ is *not recursively enumerable*.

Proposition 2.2 (Incompleteness Theorem for the Standard Model)**.** There exists a sort σ with an ADT signature $\mathcal{F}$ and defined functions $\mathcal{D}$ such that checking $\mathcal{M}_{\{\sigma\},\mathcal{F},\mathcal{D}} \models \varphi$ is not recursively enumerable for quantifier-free φ.

Note that validity over ADTs without background sorts and definitions is decidable [35] since it has a complete axiomatization [36]. The introduction of definitions (programs) is what leads to incompleteness.

2.3.3 Combinations of Theories, Nonstandard models, and Rogue Nonstandard Models

A primary observation we make in this chapter is that techniques for reasoning based on function unfolding and quantifier-free reasoning (as in Liquid Haskell and Leon) do not

reason with the standard model but rather with a certain *combination of first-order theories*. We will show this in Section 2.5.2. In this section we introduce notation for combined theories.

A *theory* $\mathcal{T}$ for a signature is an entailment closed set of first-order sentences. A model $\mathcal{M}$ satisfies a theory $\mathcal{T}$, denoted $\mathcal{M} \models \mathcal{T}$, if every sentence in the theory holds in the model. A sentence ψ is valid in $\mathcal{T}$, denoted $\mathcal{T} \models \psi$ if ψ belongs to $\mathcal{T}$.

A *theory tuple* for $(\mathcal{S}, \mathcal{F}, \mathcal{D})$ is:

- The first-order theory of ADTs $\mathcal{T}_\sigma$ for each foreground sort σ. This is the precisely the theory of the standard ADT model for σ, which may involve functions over other sorts using which the elements of σ are to be constructed. These other sorts are themselves constrained by theories like Presburger Arithmetic, or an ADT theory.
- A theory $\mathcal{T}_{bg}$ for the combined signature of the background sorts that is recursively enumerable. We require the background sorts in the standard model to satisfy this theory. In practice, this theory is the union of several axiomatized theories, say for arrays, integers, bitvectors, etc.
- Theory of uninterpreted functions with equality for symbols in $\mathcal{D}$.

The combined theory $\mathcal{T}_{comb}$ of a theory tuple is the entailment closure of the union of the theories in the tuple. A model satisfies a theory tuple (and consequently the combined theory) if the projection of the model to each subset of sorts satisfies the theories constraining those sorts. The combined theory $\mathcal{T}_{comb}$ is the set of all FO sentences that hold in all these models. For example, consider the ADT *ListNat* of lists over natural numbers introduced earlier. A theory tuple for this signature could be one that has (a) the theory of ADT lists for the foreground sort, and (b) the theory of Presburger Arithmetic (natural numbers with addition) for the background sort. The combined theory is the entailment closure of the union of the two theories.

Note that the first order theory of ADTs is *complete*. Therefore, the above setting is agnostic to the choice of any complete axiomatization for the ADT sorts! [36, 38]. Consequently, our results are also quite general and agnostic to the choice of axiomatization.

The standard models for each sort satisfy their respective theories. The other models of the individual theories are called *nonstandard models*. The standard model $\mathcal{M}_{\mathcal{S},\mathcal{F},\mathcal{D}}$ is a model of $\mathcal{T}_{comb}$, and other models of $\mathcal{T}_{comb}$ are nonstandard models for the combined theory.

Since the standard model is a model of $\mathcal{T}_{comb}$, it is clearly the case that $\mathcal{T}_{comb}$ is a *subset* of the theory of the standard model $\mathcal{M}_{\mathcal{S},\mathcal{F},\mathcal{D}}$, which we denote by $\mathcal{T}_{std}$. However, the reverse is not true in general, and in fact the combined theory can be *strictly smaller* than the theory

of the standard model. For example, consider the above example of *ListNat* where we extend the logic with the predicate symbol R with the following recursive definition:

$$\begin{aligned} R(x) = ite(&is\_nil(x), False, \\ &ite(is\_nil(tail(x)), head(x) = 1, \\ &\qquad head(x) = head(tail(x)) \wedge R(tail(x)))) \end{aligned} \tag{2.10}$$

One would expect that $R(x)$ holds only for nonempty lists x whose elements are all 1. Indeed, the statement $R(x) \rightarrow head(x) = 1$ is valid on the standard model. However, this sentence is *not valid* in the combined theory as there is a *rogue nonstandard model* that falsifies it. In this work, we define a rogue nonstandard model as a nonstandard model that falsifies a theorem of interest which is valid on the standard model.

An example of a rogue nonstandard model falsifying $R(x) \rightarrow head(x) = 1$ is as follows. It has an element u in the ADT universe that does not correspond to any standard (i.e., finite) ADT term such that $R(u)$ is true and $head(u) = 2$. Destructing u consecutively would proceed forever without reaching *nil* and and all these elements will satisfy R and have their head element to be 2, hence satisfying the recursive equation for R. We had discussed other such examples of rogue nonstandard models in Section 2.2.

Standard and nonstandard models satisfy the same FO properties for ADTs, but the addition of recursively defined functions destroys this. Although the combination of ADTs and recursively defined functions is the primary technical hurdle, we develop completeness results for a theory that also includes background sorts. This is crucial to verify functional programs as they invariably involve background theories.

In this chapter we work with a notion of validity under the combination of theories $\mathcal{T}_{comb}$. We will also henceforth use the extended signature $(\mathcal{S}, \mathcal{F}, \mathcal{D}, \mathcal{T}_{comb})$.

2.3.4 Validity under Defined Functions

Let *DEF* be a set of definitions def_D for each $D \in \mathcal{D}$. We define the validity of a first-order formula φ under definitions *DEF* by considering pairs of the form (DEF, φ).

Definition 2.1 (Validity of FOL Formulae with Defined Functions)**.** Given $(\mathcal{S}, \mathcal{F}, \mathcal{D})$ with definitions *DEF* of functions in $\mathcal{D}$, we say that a formula φ is $\mathcal{T}$-valid under the definitions iff $DEF \rightarrow \varphi$ is $\mathcal{T}$-valid, i.e., is in $\mathcal{T}$. Note that we are using *DEF* in the formula to mean the conjunction of definitions in that set. We denote this by $\mathcal{T} \models (DEF, \varphi)$.

We can utilize the above notion in the case of the theory of the standard model or the

combined theories, writing $\mathcal{T}_{std} \models (DEF, \varphi)$ or $\mathcal{T}_{comb} \models (DEF, \varphi)$ respectively. As before, if $\mathcal{T}_{comb} \models (DEF, \varphi)$ then $\mathcal{T}_{std} \models (DEF, \varphi)$.

2.4 A FLUID LOGIC

In this section we define our first main contribution: the FLUID (First-Order Logic of Universal properties under Inductive Definitions) fragment that captures VCs generated by tools like LH and LEON. The heart of the FLUID fragment is a class of inductive definitions called *provably acyclic* definitions.

Recall that we require definitions to terminate on the standard model. We demand in the FLUID fragment that definitions also satisfy a *provable acyclicity* condition, which is a notion similar to termination. Intuitively, acyclicity means that when definitions are unrolled, there is no cyclic dependency between the recursive calls. Note terminating functions must be acyclic, but acyclic definitions can be non-terminating. For example, the function *forever* on Lists defined by $forever(x) = forever(Cons(0, x))$ does not terminate, but it is acyclic because the recursive calls do not repeat. We demand in the FLUID fragment that the acyclicity property expressed as a first-order formula is *provable* for the recursive definitions[9]. We formulate provable acyclicity below using ranking functions and path conditions, which we first define formally.

An ordered sort $S \in \mathcal{S}$ is one with a binary predicate $<$ such that $<$ forms a strict partial order. Formally, $<$ must satisfy the FO axioms expressing irreflexivity and transitivity under $\mathcal{T}_{comb}$. Note that $<$ need not be well-founded because we only require acyclicity, not termination[10]. ADT sorts are ordered with respect to the (strict) subterm relationship.

For a recursively defined function $D \in \mathcal{D}$, a *ranking function* for D is a recursively defined function $Rank_D \in \mathcal{D}$ from the domain of D to an ordered sort. We require $\mathcal{D}$ is *stratified*. The stratum of a function $D \in \mathcal{D}$ is a natural number denoted by $strat(D)$. Note that multiple functions can have the same strata. We require that every $D \in \mathcal{D}$ with $strat(D) > 0$ has a ranking function $Rank_D$ whose stratum is *strictly lower* than D. When $strat(D) = 0$, we require that D is unary over an ordered sort, and its ranking function is the identity function. Finally, we require that the definition of a function at stratum i can only call functions of strata lower than or equal to i.

We now define path conditions, and then the notion of provable acyclicity.

[9] A subtle point here is that even though terminating functions are acyclic, they need not be *provably* acyclic (see the discussion at the end of this section for an example). Therefore, termination and provable acyclicity are incomparable.

[10] Well-foundedness is not expressible in FOL anyway.

Definition 2.2 (Path Condition)**.** Given a formula ρ[11], we denote by $Path_\rho(\psi, E)$ that the sub-expression (subterm or subformula) E occurs in ρ with path condition ψ. It is the least relation satisfying the following recurrence:

$$
\begin{aligned}
Path_\rho(\mathit{True}, \rho) &\text{ holds} \\
\text{If } Path_\rho(\psi, \mathsf{ite}(\mathit{cond}, E_1, E_2)) &\text{ then } Path_\rho(\psi, \mathit{cond}) \\
\text{If } Path_\rho(\psi, \mathsf{ite}(\mathit{cond}, E_1, E_2)) &\text{ then } Path_\rho(\psi \wedge \mathit{cond}, E_1) \\
\text{If } Path_\rho(\psi, \mathsf{ite}(\mathit{cond}, E_1, E_2)) &\text{ then } Path_\rho(\psi \wedge \neg \mathit{cond}, E_2) \\
\text{If } Path_\rho(\psi, D(E_1 \ldots, E_n)) \text{ for } D \in \mathcal{D} &\text{ then } Path_\rho(\psi, E_j), 1 \leq j \leq n \\
\text{If } Path_\rho(\psi, \oplus(E_1 \ldots, E_n)) \text{ for } \oplus \neq \mathsf{ite}, \oplus \notin \mathcal{D} &\text{ then } Path_\rho(\psi, E_j), 1 \leq j \leq n
\end{aligned}
$$

Informally, the path condition is the conjunction of all the conditions of ite expressions that must be satisfied in order to "reach" the given sub-expression.

Definition 2.3 (Provably Acyclic Definitions)**.** Given a signature with combined theory $\mathcal{T}_{comb}$ and stratified definitions DEF, a definition $def_D \equiv \forall \overline{x}.\, D(\overline{x}) = \rho(\overline{x})$ is provably acyclic if for every $G(\overline{t})$ $(G \in \mathcal{D})$ occurring in ρ with $strat(G) = strat(D)$, $Rank_G$ and $Rank_D$ have the same range sort, and furthermore, for every ψ such that $Path_\rho(\psi, G(\overline{t}))$:

$$
\mathcal{T}_{comb} \models \left(\bigwedge_{strat(H) < strat(D)} def_H \right) \rightarrow \left(\psi \rightarrow Rank_G(\overline{t}) < Rank_D(\overline{x}) \right)
$$

where the overloaded symbol $<$ represents an order predicate in the corresponding sort.

Informally, the above definition says that the arguments to recursive calls must be *provably* (w.r.t $\mathcal{T}_{comb}$) smaller than the input arguments as measured using ranking functions. Although we say 'provable', note that the definition uses semantic entailment ($\models$). However, these two notions are the same since FOL is complete. Provable acyclicity ensures that when a definition is unrolled, there is no cyclic dependency between recursive calls as the arguments will always decrease. We can use the definitions of functions in lower strata and the path condition to the recursive call to establish this property. We give an example below.

Example 2.1 (Sorted List Merge)**.** Consider the usual function $merge(x, y)$ for merging sorted lists. Let its stratum be 1, with its ranking function being the sum of lengths of x and y. The stratum of the length function *length* is 0.

Consider the recursive call $merge(tail(x), y)$. The path condition in this case is $x \neq Nil \wedge y \neq Nil \wedge head(x) < head(y)$. We can show that this call has smaller rank, i.e., $(length(tail(x)) + length(y)) < (length(x) + length(y))$ using the definition of *length* and the

[11] ρ is usually the body of a recursively defined function

path condition ($x \neq Nil$ ensures that the term $tail(x)$ is well-defined). We can show similarly that the other recursive call has smaller arguments, and therefore *merge* is provably acyclic.

We can also show that *length* is provably acyclic. Since its stratum is 0, its ranking function must be identity, therefore we have to show that the arguments to recursive calls must themselves decrease. This is evidently true since $length(x)$ recurses on $tail(x)$, which is smaller according to the ADT subterm ordering.

Aside We note here some subtleties in the definition of provable acyclicity. First, the relation $<$ is a *mathematical* one, and does not need to be part of the signature or logically defined. Consequently, Definition 2.3 can be established by a user/system in any way. For example, if $<$ denotes the subterm ordering on ADT Lists, then a system can trivially deduce that $tail(x) < x$. In particular, a definition that recurses on destructions of the called arguments is immediately provably acyclic. Second, ADTs are an ordered sort regardless of the choice of axiomatization because the subterm relation is an order in any model that satisfies a complete axiomatization of ADTs, including nonstandard models (even rogue ones). Therefore, we do not need ADT signatures/axiomatizations with an explicit subterm predicate [38]. Third, observe that ranks need not be well-founded as we only require acyclicity, not termination. Contrary to the usual ranking functions in literature, ranks need not be lower-bounded. For example, the function *forever* defined above is provably acyclic because we can say $Rank(Cons(0, x)) < Rank(x)$ with the rank being negative of the length, which has no lower bound.

Our fragment is very general and includes most definitions we know that tools use. In practice, provable acyclicity is satisfied when functional programs are proved terminating. This is because, to the best of our knowledge, every tool that proves functional programs terminating uses ranking functions that map arguments to a well-founded order (typically tuples of natural numbers, often associated with the size of ADTs), and shows that (1) the ranking function decreases (according to some order relation $<$) on recursive calls, and (2) the order $<$ on which the ranking function decreases is well-founded. Condition (1) is precisely the property in Definition 2.3!

Intuitively, provable acyclicity generalizes the idea of proving termination. Termination makes sense on a standard model, but in a nonstandard model ADT elements can have "infinite tails" and therefore a function that terminates on the standard model can be nonterminating on a nonstandard model[12]. In contrast, provable acyclicity makes sense on all models, standard and nonstandard. We show that in any $\mathcal{T}_{comb}$ model, provably acyclic definitions

[12]Here we mean nonterminating in the sense that the evaluation procedure described in Section 2.3.2 does not terminate for all inputs drawn from the nonstandard model.

are always satisfiable (though they may not have a unique interpretation). Formally (see Corollary 2.1 in Section 2.5.2 for a proof):

Theorem 2.1. Given a signature $(\mathcal{S}, \mathcal{F}, \mathcal{D}, \mathcal{T}_{comb})$, a set of stratified definitions *DEF* that are provably acyclic, and a model $\mathcal{M}$ of $\mathcal{T}_{comb}$, there exists a model $\mathcal{M}'$ of $\mathcal{T}_{comb}$ such that the interpretation of symbols in $\mathcal{F}$ coincides with $\mathcal{M}$ and interpretations of symbols in $\mathcal{D}$ satisfy their definitions.

We now define the FLUID fragment.

Definition 2.4 (FLUID Fragment)**.** Given a signature $(\mathcal{S}, \mathcal{F}, \mathcal{D}, \mathcal{T}_{comb})$ and a set of stratified definitions *DEF* for the symbols in $\mathcal{D}$, the pair (DEF, φ) is in the FLUID fragment if (1) every definition in *DEF* is provably acyclic, and (2) φ is purely universally quantified.

Discussion on Provable Acyclicity vs. Termination Termination refers only to termination on the standard model. In contrast, provable acyclicity requires that arguments do not repeat when unfolding a definition (on any model), which is established using an order predicate and ranking functions. However, neither one implies the other.

For example, the function $f(x) = f(cons(0, x))$ is a provably acyclic function since the arguments to the function will never repeat across successive recursive calls. However, f does not terminate on the standard model: it simply keeps calling itself on larger and larger lists.

In contrast, the following predicate g is terminating on the standard model but is not provably acyclic:

$$\begin{aligned} \forall x.\, std(x) &= \mathsf{ite}(x = Nil, True, std(tail(x))) \\ \forall x.\, g(x) &= \mathsf{ite}(std(x), True, g(x)) \end{aligned} \tag{2.11}$$

std always terminates on the standard model returning $True$: it continually destructs the element and recursively calls itself until it reaches *Nil*, at which point it returns *True*. Therefore, $g(x)$ also terminates for elements in the standard model as one would simply evaluate the outermost condition (which terminates), and then take the branch corresponding to the success of the condition (which is just *True*).

However, g is not provably acyclic: it recursively calls $g(x)$ which does not decrease the argument. We can't use the fact we used in the termination argument that the *else* branch will never be taken because for provability we have to consider all models, not just the standard model. There are models where *std* does not always evaluate to *True*.

In Section 2.8 we use the fact that *std* cannot be proved to always evaluate to *True* to show incompleteness of UQFR when provably acyclic functions are replaced by terminating functions (see Theorem 2.6).

2.5 COMPLETENESS OF DEFINITION UNFOLDING AND QUANTIFIER-FREE REASONING

In this section we describe the algorithm UQFR, based on Unfolding definitions followed by Quantifier-Free Reasoning, for checking validity of universal properties. We show that the algorithm intrinsically only proves theorems in the combined theory $\mathcal{T}_{comb}$. We then prove our main technical result: the algorithm is *complete* for $\mathcal{T}_{comb}$. Let us fix a signature $(\mathcal{S}, \mathcal{F}, \mathcal{D}, \mathcal{T}_{comb})$ through this section. Recall that $\mathcal{T}_{comb}$ represents the combined theory for the foreground and background sorts, with $\mathcal{D}$ being uninterpreted. Fix also the theory of the standard model $\mathcal{T}_{std}$.

We require for our algorithm that $\mathcal{T}_{comb}$-validity is *decidable* for *quantifier-free formulas*, and that the quantifier-free fragments of $\mathcal{T}_{comb}$ and $\mathcal{T}_{std}$ are identical. We are agnostic to the choice or presence of an axiomatization for the theories and have no other constraints on $\mathcal{T}_{comb}$. This assumption is satisfied for several combined theories, including those that admit Nelson-Oppen combination [30, 44] e.g. ADTs, linear arithmetic, reals, etc. In fact, such theories also admit efficient decision procedures as evidenced by SMT solvers [11, 12]. Checking validity is achieved by negating and checking for unsatisfiability. Note that the *quantified* theories $\mathcal{T}_{std}$ and $\mathcal{T}_{comb}$ are however typically different; see Section 2.3.3.

2.5.1 UQFR Algorithm

The high-level picture of the algorithm is as follows: presented with a set of definitions DEF and a quantifier-free formula φ, UQFR systematically unfolds the definitions on terms on which functions are applied and dispatches the resulting quantifier-free formulas to a decision procedure for satisfiability. We first provide some definitions that are useful in describing the algorithm.

Definition 2.5 ($\mathcal{D}$-Application)**.** A $\mathcal{D}$-application is a pair $(D, \bar{t})$ where $D \in \mathcal{D}$ and $\bar{t} = (t_1, t_2 \ldots, t_r)$ is a tuple of a terms such that $D(\bar{t})$ is well-formed, i.e., D has signature $\sigma_1 \times \sigma_2 \ldots \times \sigma_r \to \sigma$ and t_i is of type σ_i for $1 \leq i \leq r$. A $\mathcal{D}$-application $(D, \bar{t})$ occurs in a formula ψ if $D(\bar{t})$ occurs in ψ.

Definition 2.6 (Definition Unfolding)**.** Let $\varphi \equiv \forall x_1. \forall x_2. \ldots \forall x_n. \psi$ be a universally quantified formula such that ψ is quantifier free. The instantiation of φ with a tuple of terms $\bar{t} \equiv (u_1, u_2, \ldots u_n)$, written $\varphi[\bar{t}]$, is the quantifier-free formula $\psi[u_1/x_1, \ldots u_n/x_n]$.

Given a set of C of $\mathcal{D}$-applications and a set $DEF = \{def_D \mid D \in \mathcal{D}\}$ of definitions we denote $DEF[C] = \{def_D[\bar{t}] \mid (D, \bar{t}) \in C\}$. Informally, $DEF[C]$ is the set of quantifier-free formulas corresponding to unfoldings of functions D on arguments $\bar{t}$ given by C.

Input: (DEF, φ) such that φ is universally quantified, with $DEF = \{def_D \mid D \in \mathcal{D}\}$
Output: *VALID* (when it terminates)
Imports: QFreeSAT for deciding $\mathcal{T}_{std}$-satisfiability of quantifier-free formulas

```
1: procedure UQFR[S; F; D; T_std]
2:     formulas := {¬φ} // Negate the formula and check for satisfiability
3:     while True do
4:         res := QFreeSAT(formulas) // Check sat of ¬φ with current unfoldings
5:         if res = UNSAT then
6:             return VALID // (DEF, φ) is valid
7:         else
8:             // Compute D-applications occurring in formulas
9:             D_applications := {(D, t̄) | D(t̄) occurs in ψ for ψ ∈ formulas}
10:            // Unfold the definitions and add them to formulas
11:            formulas := formulas ∪ DEF[D_applications]
```

Algorithm 2.1: Algorithm for Unfolding Definitions followed by Quantifier-Free Reasoning

Algorithm Description Algorithm 2.1 shows the pseudocode for the UQFR, parameterized by the signature $(\mathcal{S}, \mathcal{F}, \mathcal{D}, \mathcal{T}_{std})$ with the theory of the standard model. It takes as input a set of definitions $DEF = \{def_D \mid D \in \mathcal{D}\}$ and a formula φ such that φ is universally quantified. UQFR attempts to prove validity by establishing unsatisfiability of the negation $DEF \wedge \neg\varphi$ (see Definition 2.1 to see that these are equivalent). Finally, UQFR also assumes access to an external procedure QFreeSAT that checks the satisfiability of quantifier-free formulas with respect to the theory of the standard model $\mathcal{T}_{std}$. It takes as input a set of formulas and outputs *SAT* if the conjunction of the formulas is $\mathcal{T}_{std}$-satisfiable and *UNSAT* otherwise.

The algorithm maintains a set *formulas* of quantifier-free formulas consisting of $\neg\varphi$ along with finitely many unfoldings of the definitions. If this set is unsatisfiable then the formula $\wedge DEF \wedge \neg\varphi$ is unsatisfiable as well, i.e., φ is valid under DEF. Initially the set contains only $\neg\varphi$. Since φ is purely universal, we treat $\neg\varphi$ as a quantifier-free formula by adding the existentially quantified variables as new ground terms (constants) in our signature.

At a general point in the algorithm (line 3), we check the $\mathcal{T}_{std}$-satisfiability of *formulas* using the external procedure QFreeSAT (line 4). Note that although there is a unique valuation for every $D \in \mathcal{D}$ on the standard model consistent with DEF, the set *formulas* only enforces this consistency for finitely many unfoldings of DEF and otherwise treats the symbols in $\mathcal{D}$ as uninterpreted, which is an over-approximation. If *formulas* is unsatisfiable we exit and return *VALID*. Otherwise, we refine our approximation by unfolding the definitions on more terms. We compute the set of $\mathcal{D}$-application terms occurring in *formulas* (line 9), add the corresponding unfoldings of definitions to *formulas* (line 11), and go back to the

beginning of the loop. Observe that if the algorithm is not able to prove the unsatisfiability of $\neg\varphi$ using any amount of unfoldings then it does not terminate.

2.5.2 Soundness and Completeness of UQFR under Combined Theories

In this section we prove the primary contribution of this work, namely that UQFR is complete for $\mathcal{T}_{comb}$-validity of FLUID formulas. We first show the soundness of UQFR.

Theorem 2.2 (Soundness of UQFR w.r.t $\mathcal{T}_{comb}$)**.** If UQFR$(\mathcal{S};\mathcal{F};\mathcal{D};\mathcal{T}_{std})$ terminates on (DEF,φ) then $\mathcal{T}_{comb} \models (DEF,\varphi)$.

Proof. In each round of the algorithm the set *formulas* is of the form $DEF[C] \cup \{\neg\varphi\}$, where $DEF[C]$ contains unfoldings (i.e., instantiations) of DEF on a set C of $\mathcal{D}$-applications. Therefore, if UQFR terminates then $DEF[C] \wedge \neg\varphi$ is unsatisfiable with respect to $\mathcal{T}_{std}$ (line 4).

Now, QFreeSAT can also be seen as a satisfiability procedure for the combined theory $\mathcal{T}_{comb}$ since the input formulas are *quantifier-free*. We hence have that $DEF[C] \wedge \neg\varphi$ is unsatisfiable with respect to $\mathcal{T}_{comb}$, which yields $DEF \wedge \neg\varphi$ is unsatisfiable with respect to $\mathcal{T}_{comb}$, i.e., $\mathcal{T}_{comb} \models (DEF,\varphi)$. QED.

We showed in Section 2.3.3 that typically $\mathcal{T}_{std}$ is strictly larger than $\mathcal{T}_{comb}$. The above result shows that the proving power of UQFR is in fact bounded by $\mathcal{T}_{comb}$. Therefore, not only are there valid theorems in $\mathcal{T}_{std}$ that are not valid in $\mathcal{T}_{comb}$, but it is also the case that UQFR (and hence systems such as LH and Leon) will never be able to prove those theorems.

We now show that UQFR is complete.

Theorem 2.3 (Completeness of UQFR w.r.t $\mathcal{T}_{comb}$ for FLUID)**.** If (DEF,φ) belongs to the FLUID fragment (Definition 2.4) and $\mathcal{T}_{comb} \models (DEF,\varphi)$, then UQFR$(\mathcal{S};\mathcal{F};\mathcal{D};\mathcal{T}_{std})$ terminates on (DEF,φ) and reports it valid.

We dedicate the rest of this section to the proof of the completeness theorem.

Prologue: Theorem Simplification and Reduction to Model Construction

We make some simplifications for ease of presentation. First, we assume that DEF has only one stratum. We provide a generalization of the argument made here to multiple strata at the end of this section. Second, we assume without loss of generality that the signature $(\mathcal{S},\mathcal{F},\mathcal{D},\mathcal{T}_{comb})$ is such that if a formula Γ is satisfiable in a $\mathcal{T}_{comb}$ model, then it is satisfiable

in a Herbrand model consisting of the terms occurring in Γ and closed under the applications of functions in $\mathcal{F} \cup \mathcal{D}$. This can always be done by Skolemizing $\mathcal{T}_{comb}$ and expanding $\mathcal{F}$ with new function symbols.

We first rewrite the statement of the theorem to an equivalent one. Consider the value of the sets *formulas* and $\mathcal{D}\_applications$ through the algorithm:

$$
\begin{aligned}
formulas_0 &= \{\neg\varphi\} && \text{(initial value)} \\
\mathcal{D}\_applications_i &= \{(D, \bar{t}) \mid D(\bar{t}) \text{ occurs in } \psi \in formulas_{i-1}\} && (i > 0) \\
formulas_i &= formulas_{i-1} \cup DEF[\mathcal{D}\_applications_i] && (i > 0)
\end{aligned}
$$

where the subscript i denotes their values in the i^{th} *round* of the outermost loop on line 3. Observe that $formulas_i \subseteq formulas_j$ and $\mathcal{D}\_applications_i \subseteq \mathcal{D}\_applications_j$ for every $j > i$. The completeness result can then be stated as follows:

Theorem 2.4 (Completeness of UQFR w.r.t $\mathcal{T}_{comb}$)**.** If $\mathcal{T}_{comb} \models (DEF, \varphi)$ then $formulas_i$ is $\mathcal{T}_{comb}$-unsatisfiable for some $i \geq 0$.

Note that the above theorem implies that UQFR is complete for $\mathcal{T}_{comb}$-validity because if for some i we have that $formulas_i$ is $\mathcal{T}_{comb}$-unsatisfiable, then it is also $\mathcal{T}_{std}$-unsatisfiable, therefore the algorithm will terminate in round i. By the soundness theorem (Theorem 2.5.2), (DEF, φ) is $\mathcal{T}_{comb}$-valid.

We show the contrapositive of the above statement. Let us assume that $formulas_i$ is $\mathcal{T}_{comb}$-satisfiable for every $i \in \mathbb{N}$. We show that $DEF \wedge \neg\varphi$ is $\mathcal{T}_{comb}$-satisfiable. Specifically, we construct a $\mathcal{T}_{comb}$ model $\mathcal{N}$ such that $\mathcal{N} \models DEF \wedge \neg\varphi$.

Proof Plan We construct $\mathcal{N}$ in two stages:

1. We first use the assumption that $formulas_i$ is $\mathcal{T}_{comb}$-satisfiable for every $i \in \mathbb{N}$ to construct a model $\mathcal{M}$ (using compactness) that satisfies $\bigcup_{i \geq 1} DEF[\mathcal{D}\_applications_i]$ and $\neg\varphi$. Note that this model need not satisfy *DEF everywhere* (as we have only instantiated definitions for a subset of terms).

2. In this stage we take the model $\mathcal{M}$ and consider a finite set K of pairs of the form $(D, \bar{t})$ such that the interpretation of D in $\mathcal{M}$ does not satisfy the definition of D on $\bar{t}$. We show that we can 'repair' the model so that definition of D now holds on $\bar{t}$ for every $(D, \bar{t}) \in K$. We then show that definitions can be repaired everywhere using a compactness argument. This results in the model $\mathcal{N}$ we seek.

Stage 1: Model of Infinite Instantiations

We recall the compactness theorem for FOL under combinations of theories.

Proposition 2.3 (FOL Compactness with Theories)**.** Given a signature $(\mathcal{S}, \mathcal{F}, \mathcal{D}, \mathcal{T}_{comb})$, a set of formulas Γ (finite or infinite) is $\mathcal{T}_{comb}$-satisfiable if and only if every finite subset of Γ is $\mathcal{T}_{comb}$-satisfiable.

From our assumption we know that $formulas_i$ is $\mathcal{T}_{comb}$-satisfiable for every i. Using compactness and the fact that $formulas_i$ form an increasing sequence w.r.t $\subseteq$, it follows that the infinite set $Inf = \bigcup_{i \in \mathbb{N}} formulas_i$ is $\mathcal{T}_{comb}$-satisfiable. We rewrite this as $Inf = \{\neg\varphi\} \cup \bigcup_{i \geq 1} DEF[\mathcal{D}\_applications_i]$.
Let $\mathcal{M}$ be a $\mathcal{T}_{comb}$-model that satisfies Inf. From our simplifying assumptions, we can assume that $\mathcal{M}$ is a Herbrand model. It satisfies $\neg\varphi$ and satisfies the definitions only on certain tuples, namely for $(D, \bar{t}) \in \bigcup_{i \geq 1} \mathcal{D}\_applications_i$.

Note here that if the model $\mathcal{M}$ happened to be the *standard model* the repair we wish to do would be trivial as def_D is uniquely defined (see Proposition 2.1) for each $D \in \mathcal{D}$ and we can simply 'complete' the model with the correct valuations. However, $\mathcal{M}$ can be a nonstandard model, and this results in the nontrivial aspects of our construction below.

Stage 2: Computational Closure and Model Repair

The reason we can repair $\mathcal{M}$ is because the set $\bigcup_{i \geq 1} \mathcal{D}\_applications_i$ has a special property: it is *computationally closed*. We define this property below.

Definition 2.7 (Computationally Closed Set)**.** Let Γ be a set of quantifier-free formulas. A set C of $\mathcal{D}$-applications is said to be computationally closed with respect to Γ if: (1) if $D(\bar{t})$ occurs in some formula in Γ then $(D, \bar{t}) \in C$, and (2) if $(D, \bar{t}_1) \in C$ and a $\mathcal{D}$-application $(G, \bar{t}_2)$ occurs in $def_D[\bar{t}_1]$ then $(G, \bar{t}_2) \in C$.

Intuitively, for a recursively defined function D, the computational closure of a term $D(t)$ contains all the recursive calls (at any level) made by a call to D on t, where we represent a recursive call to a function G on a term r by the $\mathcal{D}$-application (G, r). The set is called a computational closure because it is the set of calls that occur when 'computing' the value of D on t symbolically. The computational closure of a formula is then the union of the computational closures of all terms of the form $D(t)$ occurring in the formula. For example, consider the length function *length* on Lists. The computational closure of $length(Cons(1, Nil))$ is the

set $\{(\mathit{length}, \mathit{Cons}(1, \mathit{Nil})), (\mathit{length}, \mathit{Nil})\}$. Similarly, the computational closure of $\mathit{length}(x)$ is $\{(\mathit{length}, x), (\mathit{length}, \mathit{tail}(x)), (\mathit{length}, \mathit{tail}(\mathit{tail}(x))), \ldots\}$.

Using the above definition we can see that $\bigcup_{i \geq 1} \mathcal{D}\_\mathit{applications}_i$ is computationally closed for $\neg\varphi$. We now show that we can repair definitions everywhere on a Herbrand model if the definitions are already satisfied on a computationally closed sub-universe. Using this result, we can repair $\mathcal{M}$ so that definitions are satisfied everywhere, which is what we want.

Lemma 2.1 (Finite Repair outside Computational Closure)**.** Let $\mathcal{M}$ be a Herbrand model, Γ a set of quantifier-free formulae, C a computationally closed set for Γ, and K a finite set of $\mathcal{D}$-applications not in C. Let $\mathcal{M}$ satisfy $\mathit{DEF}[C] \cup \Gamma$. Then there exists a model $\mathcal{M}'$ that satisfies $\mathit{DEF}[K] \cup \mathit{DEF}[C] \cup \Gamma$.

Proof. Observe that if K is singleton, say $\{(D, \bar{t})\}$, we can construct $\mathcal{M}'$ by simply 'updating' the interpretation of D on $\bar{t}$ according to the definition. Formally, the model $\mathcal{M}[D(\bar{t}) := [\![\rho(\bar{t})]\!]_{\mathcal{M}}]$ satisfies $\mathit{DEF}[\{(D, \bar{t})\}] \wedge \mathit{DEF}[C] \wedge \Gamma$. Here $\mathcal{M}[(D, \bar{t}) := v]$ denotes an updated model whose interpretation of $D(\bar{t})$ is v but is otherwise identical to $\mathcal{M}$. We also use $[\![\cdot]\!]_{\mathcal{M}}$ to denote the interpretation of $\mathcal{M}$. The correctness of this construction follows from the fact that the definitions over C are satisfied despite the update since C is computationally closed. Consequently the satisfaction of Γ is also unaffected because if $G(\overline{r})$ occurs in Γ then $(G, \overline{r})$ belongs to C.

To show that $\mathit{DEF}[K]$ is satisfiable for an arbitrary finite subset K, we take $\mathcal{M}$ and apply updates as above on each pair in K. However, we have to do this carefully so that each repair does not break any previous repairs. Fix a set K' and a model $\mathcal{M}'$ such that $K' \subseteq K$ and $\mathcal{M}'$ is $\mathcal{M}$ with updated with the fixes for the elements in K'. Initially $K' = K$ and $\mathcal{M}' = \mathcal{M}$. We describe below a mechanism $\mathit{Minimal}(K, K', \mathcal{M}')$ to choose a 'minimal' element in K that has not been fixed yet, and repair it as described above.
$\mathit{Minimal}(K, K', \mathcal{M}')$ is as follows:

1. Pick an arbitrary element $(D, \bar{t}) \in (K \setminus K')$. Let the body of def_D be ρ.
2. We evaluate $\rho(\bar{t})$ on $\mathcal{M}'$ in the following way: subterms must be evaluated before superterms, and for conditionals we evaluate the condition first and then only evaluate the appropriate branch.
3. If the evaluation as described above does not encounter any element in $K \setminus K'$, then return $(D, \bar{t})$.
4. If the evaluation of $\rho(\bar{t})$ encounters a term $G(\overline{r})$ such that $(G, \overline{r}) \in (K \setminus K')$, we recurse, going back to Step (2) and evaluating $\tau(\overline{r})$ where τ is the body of def_G.

Informally, this mechanism has the flavor of an *eager evaluation*, in that we evaluate $\rho(\bar{t})$ eagerly, following the evaluation procedure down (recursively) to a minimal unfixed $\mathcal{D}$-application in K.

Finally, when the procedure returns an element $(H, \overline{u})$, we add it to K' and update $\mathcal{M}'$ with the repair for $(H, \overline{u})$. We then repeat this process of picking a minimal element and repairing the model on it until all elements in K are fixed. This completes our construction, and the model $\mathcal{M}'$ obtained at the end of all the fixes is the model we desire.

A subtle point in the construction is the termination of $\mathit{Repair}(K, K', \mathcal{M}')$ as the choice of minimal element is not well-defined otherwise. If the mechanism does not terminate, it must be because the evaluation of some $D(\bar{t})$ encounters itself. However, this is impossible as definitions are provably acyclic (Definition 2.3). Formally, we have the following proposition:

Proposition 2.4. Let $(D, \bar{t})$ be a $\mathcal{D}$-application in K, ρ be the body of the definition of D, and $\mathcal{M}$ be a model of $\mathcal{T}_{comb}$. Let $(G, \overline{r})$ be a $\mathcal{D}$-application such that $G(\overline{r})$ is a sub-expression of $\rho(\bar{t})$ and the evaluation of $\rho(\bar{t})$ in $\mathcal{M}$ as performed in the *Repair* mechanism above encounters $G(\overline{r})$. Further, let ψ be the path condition of the sub-expression $G(\overline{r})$ that is reached (see Definition 2.2). Then, we have that $\mathcal{M} \models \psi$.

We skip the proof of this proposition as it follows trivially from the description of the evaluation mechanism and Definition 2.2. Now, from the definition of provable acyclicity (Definition 2.3), we know:

$$\mathcal{T}_{comb} \models \left(\bigwedge_{strat(H) < strat(D)} def_H \right) \rightarrow \psi \rightarrow \mathit{Rank}_G(\overline{r}) < \mathit{Rank}_D(\bar{t})$$

Per our assumption we have only one stratum, so the set $\{H\}_{H \in \mathcal{D}, strat(H) < strat(D)}$ is empty. Since $\mathcal{M}$ is a $\mathcal{T}_{comb}$ model that satisfies ψ, we obtain $\mathcal{M} \models \mathit{Rank}_G(\overline{r}) < \mathit{Rank}_D(\bar{t})$. Therefore, if updating $D(\bar{t})$ requires updating $G(\overline{r})$, then the rank of $G(\overline{r})$ is smaller. Therefore, each recursive call of *Repair* is made on a smaller element of K, and therefore the evaluation of $D(\bar{t})$ cannot depend on itself. The mechanism for picking a minimal element is indeed well-defined and we can produce at the end of the procedure a model $\mathcal{M}'$ that satisfies $DEF[K] \cup DEF[C] \cup \Gamma$.

End of proof of Lemma 2.1. QED.

Repair for All Tuples

We now show that we can repair definitions everywhere, i.e., on arbitrarily large sets of $\mathcal{D}$-applications. We first show some easy results. For a sort σ, consider the set U_σ consisting

of all terms of type σ. Then, the set $DApp$ of all possible $\mathcal{D}$-applications is:

$$DApp = \{(D, (t_1, t_2, \ldots t_r)) \mid D \in \mathcal{D},\ \text{D has signature } \sigma_1 \times \sigma_2 \ldots \sigma_r \to \sigma,\ t_i \in U_{\sigma_i}\}$$

Note that any $\mathcal{D}$-application $(D, \bar{t})$ must belong to $DApp$, and in particular any computationally closed set C, which is a set of $\mathcal{D}$-applications, must be a subset of $DApp$.

Finally, if $\mathcal{N}$ is a Herbrand model of $\mathcal{T}_{comb}$, then its universe for a sort σ is precisely U_σ. Therefore, satisfying definitions *everywhere* on $\mathcal{N}$ simply amounts to satisfying definitions on $DApp$. The following proposition captures this idea:

Proposition 2.5 (Definitions on a Herbrand Model)**.** Let $\mathcal{N}$ be a Herbrand model of $\mathcal{T}_{comb}$. Then, $\mathcal{N} \models DEF$ if and only if $\mathcal{N} \models DEF[DApp]$

We now show the correctness of repairing arbitrarily large sets of $\mathcal{D}$-applications outside a computational closure.

Lemma 2.2 (Definition Completion Lemma)**.** Let DEF be a set of definitions with only one stratum. Let C be a set that is computationally closed with respect to a set of quantifier-free formulas Γ. If $DEF[C] \wedge \Gamma$ is $\mathcal{T}_{comb}$-satisfiable, then $DEF \wedge \Gamma$ is $\mathcal{T}_{comb}$-satisfiable.

The proof is the same as the one given in the main text, but we repeat it here.

Proof. We claim that $DEF[DApp] \cup \Gamma$ is $\mathcal{T}_{comb}$-satisfiable. Since $C \subseteq DApp$, let us rewrite this as $DEF[C] \cup DEF[\overline{C}] \cup \Gamma$, where $\overline{C} = DApp \setminus C$ is the complement of C.

We have from the statement of the theorem that $DEF[C] \cup \Gamma$ is satisfiable. Therefore, to show satisfiability of our desired set by compactness, it is sufficient to show that $DEF[C] \cup \Gamma \cup B$ is satisfiable for an arbitrary finite subset B of $DEF[\overline{C}]$.

Observe that a finite subset of $DEF[\overline{C}]$ is of the form $DEF[K]$ for a finite set $K \in DApp$. We are now done, since we know that $DEF[C] \cup \Gamma \cup DEF[K]$ is satisfiable from Lemma 2.1.

Finally, consider a model $\mathcal{N}$ such that $\mathcal{N} \models DEF[DApp] \cup \Gamma$. Without loss of generality, we can assume that $\mathcal{N}$ is a Herbrand model. Applying Proposition 2.5 gives us that $\mathcal{N} \models DEF \cup \Gamma$, which concludes the proof. *End of proof of Lemma 2.2.* QED.

Epilogue: Generalizing Model Repair to Stratified Definitions

In the above proof we assumed that DEF had only one stratum. For the case of multiple strata, we first begin with the model $\mathcal{M}$ of Inf given to us by Stage 1. We then induct on the stratum number i, with the inductive hypothesis being that definitions for functions from

strata $< i$ are satisfied everywhere. This hypothesis is true for the base case of the lowest stratum $i = 0$ by Lemma 2.1.

Inductively, we assume the hypothesis and then repair the model on definitions in the current strata by applying Lemma 2.2. The arguments for the correctness of repair in this case are identical, i.e., we show the correctness of finite repair and then apply compactness.

However, there is a subtlety involved in showing the correctness of finite repair. The arguments are the same as in the proof of Lemma 2.1, with one exception. When we consider the argument for decreasing ranks, we needed the conjunct $\left(\bigwedge_{strat(H)<strat(D)} def_H\right)$ to be valid. When there is only one stratum, this is trivial since the conjunct is empty. For a general stratum i in our induction proof, the formula demands that the definitions corresponding to defined functions from lower strata are satisfied everywhere. But this is precisely the induction hypothesis! This concludes the proof of correctness of repair for a stratified set of definitions. QED.

As a corollary, we obtain the following result which captures the intuition behind the definition of provable acyclicity:

Corollary 2.1 (Repetition of Theorem 2.1)**.** Given a signature $(\mathcal{S}, \mathcal{F}, \mathcal{D}, \mathcal{T}_{comb})$, a set of stratified definitions *DEF* that are provably acyclic, and a model $\mathcal{M}$ of $\mathcal{T}_{comb}$, there exists a model $\mathcal{M}'$ of $\mathcal{T}_{comb}$ such that the interpretation of symbols in $\mathcal{F}$ coincides with $\mathcal{M}$ and interpretations of symbols in $\mathcal{D}$ satisfy their definitions.

Proof. We simply apply the arguments of the definition completion lemma (Lemma 2.2) to $\mathcal{M}$ for each stratum of the definitions starting with the lowest, setting the set C where definitions are already satisfied as well as the set of formulas Γ to be empty. We can show by induction that when we apply the lemma at stratum i, the resulting model will satisfy definitions at strata $\leq i$ everywhere. *End of proof of Corollary 2.1.* QED.

2.6 FLUID REASONING IN LIQUID HASKELL

Next, let us see how the Liquid Haskell verifier (LH) employs a particular instance of FLUID reasoning referred to by the tool as *reflection* and *proof by logical evaluation* (PLE). We show how a user might use LH to develop a small library of theorems about Peano numbers to illustrate why it can be viewed as FLUID reasoning, why its FLUID-style instantiation heuristics are effective in practice, and, perhaps more importantly, why extra information is *really* required from the user when instantiation fails.

Peano Addition Consider the definition of *Peano* numbers

```
data Peano = Z | S Peano
```

As described in Section 2.3.1, LIQUID HASKELL uses the above definition to generate an ADT `Peano` with (1) two *constructors* Z and S, (2) two *recognizers* isZ and isS, and (3) a single *destructor* $pred$. Next, consider the following function that recursively defines addition

```
plus :: Peano → Peano → Peano
plus Z     m = m
plus (S n) m = S (plus n m)
```

LH generates a *definition* for `plus` which is an "axiom" constraining *plus* [32]

$$def_{plus} \equiv \forall n, m.\ plus(n, m) = ite(isZ(n),\ m,\ S(plus(pred(n), m)) \tag{2.12}$$

2.6.1 Proof by Instantiation

Propositions as Types Suppose we wish to verify that the addition of `Z` is an identity function, i.e. the proposition $\forall n : Peano.\ plus(Z, n) = n$. In LH, a user uses the recipe of "Propositions as Types" to *specify* the property as a type, and *verify* it via a function `zeroL` that inhabits the type:

```
zeroL :: n:Peano → { plus Z n == n }
zeroL n = ()
```

In the above type signature, the *input parameter* has the effect of quantifying over all n, and the *output post-condition* stipulates the particular property that must hold for each n [45].

Programs as Proofs To check this proof, LH generates a VC $def_{plus} \rightarrow \forall n.plus(Z, n) = n$. Next, it uses *logical evaluation* (PLE) [32] to instantiate the definition of *plus* (2.12) at (Z, n) to get the instantiated VC $\forall n.\ def_{plus}[Z, n] \rightarrow plus(Z, n) = n$ using the instantiation

$$def_{plus}[Z, n] \equiv (plus(Z, n) = ite(isZ(Z),\ n,\ S(plus(pred(Z), n)) \tag{2.13}$$

The SMT solver proves the above instantiated VC is valid even when *plus* is uninterpreted, thereby verifying that `plus Z` is an identity function.

2.6.2 Proof by Induction

LH makes no attempt to automate inductive proofs. Instead, the programmer must *explicate induction via recursion*, by writing programs where the induction hypothesis is made explicit in the VC via the asserted post-conditions of recursive calls to smaller inputs.

Constructor			Destructor			Plus		
$\mathcal{I}(S)(i)$	$=$	$i+1$	$\mathcal{I}(pred)(i)$	$=$	$n-1$ if $0<n$	$\mathcal{I}(plus)(i,j)$	$=$	$i+j$
$\mathcal{I}(S)(i')$	$=$	$(i+1)'$	$\mathcal{I}(pred)(i')$	$=$	$(i-1)'$	$\mathcal{I}(plus)(i',j')$	$=$	$(i+j)'$
$\mathcal{I}(Z)$	$=$	0				$\mathcal{I}(plus)(i',j)$	$=$	$(i+j+1)'$
						$\mathcal{I}(plus)(i,j')$	$=$	$(i+j)'$

Figure 2.1: Rogue Nonstandard Model for *Peano* over the universe $\mathcal{U} \equiv \{0,1,2,\ldots\} \cup \{\ldots,-2',-1',0',1',2',\ldots\}$. comprising the naturals and a *primed* version of each integer. The model provides an interpretation for various constructors, destructors and *plus* that respects the ADT axioms, but where *plus* has a nonstandard interpretation on nonstandard ADT elements i': $\mathcal{I}(plus)(i',Z) = (i+1)' \neq i'$, refuting (2.15) , and $\mathcal{I}(plus)(i',j) \neq \mathcal{I}(plus)(j,i')$, refuting (2.14).

As an example, suppose that we wish to verify that the definition of `plus` is commutative. As before, the programmer would start by specifying the above proposition as the type shown at the top of Figure 2.2, and might attempt a direct proof `comm n m = ()`[13] that would yield the FLUID VC

$$def_{plus} \rightarrow (\forall n, m.\ plus(n,m) = plus(m,n)) \tag{2.14}$$

Sadly, PLE does not find any suitable instantiations, and so the SMT solver *cannot* prove the above is valid when *plus* is uninterpreted and hence *rejects* the code on the left.

Rogue Nonstandard Model Did LH simply give up too early — maybe some carefully chosen instantiations would produce a valid instantiated VC? Surprisingly, this is not the case. In fact, verification fails because (2.14) is refuted by a rogue nonstandard model (Figure 2.1) where the interpretation for the constructors and destructors respects the ADT axioms for `Peano` and *plus* satisfies its definition, but there exists an element i' such that $plus(i',Z) \neq plus(Z,i')$ in the model. Let us banish such rogue models by proving that adding `Z` on the *right* is also an identity,

$$\forall n : Peano.\ plus(n,Z) = n \tag{2.15}$$

A direct proof of 2.15 is doomed: it yields the VC below which is refuted by the model in Figure 2.1:

$$def_{plus} \rightarrow \forall n.\ plus(n,Z) = n \tag{2.16}$$

[13] () is a "unit proof" with no extra hints from the user. LH attempts to prove the VC directly given a unit proof.

An Inductive Proof The programmer must spell out an inductive proof as a (recursive) piece of code that yields a VC which *excludes* rogue nonstandard models by explicitly stating the induction hypothesis as an antecedent in the VC. This is achieved via the proof `zeroR` shown on the left in Figure 2.2. First, we split cases (via a pattern match) on the first argument, treating separately the cases where the argument is `Z` or `S n`. Second, the recursive call to `zeroR n` puts the post-condition of `zeroR` for the *smaller input* `n` as a hypothesis for the new VC

$$def_{plus} \rightarrow (\forall n.\ plus(Z, Z) = Z \wedge plus(n, Z) = n \rightarrow plus(S(n), Z) = S(n)) \tag{2.17}$$

PLE instantiates def_{plus} (2.12) at (Z, Z) and $(S(n), Z)$ to get the instantiated VC

$$def_{plus}[Z, Z] \rightarrow def_{plus}[S(n), Z] \rightarrow (plus(Z, Z) = Z \wedge plus(n, Z) = n \rightarrow plus(S(n), Z) = S(n)) \tag{2.18}$$

The instantiated VC is valid even when *plus* is uninterpreted, thus proving (2.15). In essence, the (well-founded) *recursive call* to `zeroR` establishes the *induction hypothesis* for the smaller `n`, thereby eliminating the rogue nonstandard models, letting us verify the proposition for *any* `Peano n`.

2.6.3 Proof by Lemmas

Next, let us see how to use auxiliary lemmas like `zeroR` to eliminate rogue nonstandard models that thwarted the direct proof of the commutativity of `plus`. First, the programmer might attempt an inductive proof (like the `zeroR`) as shown in the middle in Figure 2.2: split cases on whether the first parameter is `Z` or `S n`. In the base case, they would *call* `zeroR m` to eliminate the rogue nonstandard model where $plus(i', 0) \neq plus(0, i')$ (Figure 2.1). In the inductive case, they would recursively invoke the induction hypothesis via recursively calling `comm n m`. This time, LH generates the VC

$$\begin{aligned} def_{plus} \rightarrow (\forall n, m.\ & (plus(m, Z) = m \rightarrow plus(Z, m) = plus(m, Z)) \\ & \wedge\ (plus(n, m) = plus(m, n) \rightarrow plus(S(n), m) = plus(m, S(n)))) \end{aligned} \tag{2.19}$$

Thanks to the equality asserted by the use of the "lemma" `zeroR m`, the first conjunct can proved valid via the instantiation $plus[(m, Z)]$. However, the second conjunct is invalid *despite* the recursive (inductive) call to `comm n m` because of a *different* rogue nonstandard model for *plus* that falsifies the second conjunct!

Rogue Nonstandard Model for `comm` Attempt 1 The following is a rogue nonstandard model for the `Peano` numbers over the same ADT universe as the model in Figure 2.1 and a

different interpretation of `plus` that refutes the second conjunct of (2.19).

$$\begin{array}{lcllcll} \mathcal{I}(plus)(i,j) & = & i+j & \mathcal{I}(plus)(i',j') & = & (i+j+1)' & \text{if } 0 \leq j \\ \mathcal{I}(plus)(i,j') & = & (i+j)' & \mathcal{I}(plus)(i',j') & = & (i+j-1)' & \text{otherwise} \\ \mathcal{I}(plus)(i',j) & = & (i+j)' & & & & \end{array} \tag{2.20}$$

The reader should take a moment to check that the above definition respects def_{plus}. However, even though the induction hypothesis trivially holds at $\mathcal{I}(plus)(-1', -1')$ (as the arguments are the same), $\mathcal{I}(plus)(0', -1') = -2'$ which differs from $\mathcal{I}(plus)(-1', 0') = 0'$!

Proof of `comm` Attempt 2 The peculiar property of the rogue nonstandard model from attempt 1 is that it introduces a *discontinuity* at $0'$ that violates Again $\forall n, m : Peano.\ plus(n, S(m)) = S(plus(n, m))$. We separately specify and inductively prove this property via a lemma `succR`:

```
succR :: n:Peano → m:Peano → { plus n (S m) = S (plus n m) }
```

The proof of `succR` is similar to `zeroR`:

```
succR :: n:Peano → m:Peano → { plus n (S m) = S (plus n m) }
succR Z     _ = ()
succR (S n) m = succR n m
```

Now, we can use both helper lemmas `zeroR` and `succR` to eliminate the rogue nonstandard models that thwarted our previous attempts, by the proof shown on the right in Figure 2.2. First, we replace the body of `comm Z m` with `comm z m = zeroR m` which add the post-condition of `zeroR` as a lemma. Second, we strengthen the body of `comm (S n) m` with a call `succR n m`. Together, these lemmas yield a VC with the strengthened antecedents that preclude the above rogue nonstandard models

$$\begin{array}{l} def_{plus} \rightarrow (\forall n, m.\ (plus(m, Z) = m \rightarrow plus(Z, m) = plus(m, Z)) \\ \qquad \wedge\ (plus(n, m) = plus(m, n) \rightarrow plus(m, S(n)) = S(plus(m, n)) \rightarrow \\ \qquad\quad plus(S(n), m) = plus(m, S(n)))) \end{array} \tag{2.21}$$

This time, PLE instantiates def_{plus} at (Z, m) and $(S(n), m)$ to yield an instantiated VC that the SMT solver validates even when *plus* is uninterpreted. Note that as with induction, LH makes no attempt to automate the creation and use of such lemmas: the programmer must explicitly spell them out by defining and proving them, and then "calling" the lemmas inside the theorem body to appropriately "instantiate" them at the relevant values, thereby yielding a VC that can be automatically discharged by FLUID reasoning.

```
-- Succeeds
zeroR :: n:_ →
  {plus n Z == n}
zeroR Z     = ()
zeroR (S n) = zeroR
   n
```

```
-- Fails
comm :: n:_ → m:_ →
  {plus n m == plus m
     n}
comm Z     m = zeroR m
comm (S n) m = comm n
     m
```

```
-- Succeeds
comm :: n:_ → m:_  →
  {plus n m == plus m n}
comm Z     m = zeroR m
comm (S n) m = comm n m
             && succR m n
```

Figure 2.2: Proof of the commutativity of *Peano* addition: The explicit case-splitting, recursion and "lemma application" are needed to eliminate rogue nonstandard models.

```
get :: Map k v → k → Maybe
   v
get (Node k v l r) key
 | key == k   = Just v
 | key <  k   = get l key
 | otherwise = get r key
get Leaf _    = Nothing
```

```
set :: Map k v → k → v → Map k v
set (Node k v l r) key val
 | key == k   = Node key val l r
 | key <  k   = Node k v (set l key val)
     r
 | otherwise = Node k v l (set r key
    val)
set Leaf k v = Node k v Leaf Leaf
```

Figure 2.3: Implementations of `get` and `set` functions for Binary Search Tree.

2.6.4 Rogue Nonstandard Models in Proofs about Data Structures

We use the simple `Peano` datatype to illustrate how LH implements FLUID reasoning, and how direct proofs can fail due to rogue nonstandard models which can be eliminated via explicit induction (recursion) and lemmas (function calls). Similar phenomena occur when verifying more complicated properties. Consider the datatype of finite maps from keys (`k`) to values (`v`)

```
data Map k v = Leaf | Node k v (Map k v) (Map k v)
```

Figure 2.3 shows the code for two functions that respectively `get` the value of a `key` from a tree, and `set` the value of a `key` to some new `val` leaving the values of all other keys unchanged. The following proposition is one of McCarthy's two laws that characterize finite maps

$$\forall m, k, v.\ get(set(m, k, v), k) = Just(v)$$

Attempt 1: Direct Proof In LH, we could try to specify and verify the above law as

```
getEq :: m:_ → k:_ → v:_ → { get (set m k v) k = Just v }
getEq m k v = ()
```

which would yield the VC $(def_{get} \wedge def_{set}) \rightarrow \forall m, k, v.\ get(set(m, k, v), k) = Just(v)$ Unfortunately, no (finite) instantiation can prove the above VC, and we describe here an intuitive rogue nonstandard model that falsifies the theorem.

Rogue Nonstandard Model Let the universe $\mathcal{U}$ be the set of *finite trees* and *infinite non-regular trees*[14] over (k, v) pairs. The interpretation for $get(m, k)$ on a tree m follows the usual path in a binary search tree to find k, even on infinite trees. If the k is found, *get* returns the corresponding value, and if the path ends or continues forever, then *get* returns `Nothing`. The interpretation for *set* is similar, except that on an infinite computation it returns the input tree.

To see why this model refutes the VC, consider the *infinite binary tree* m that is infinite on all paths, and every node of which has key 0 and value 0. Let us call `set`, to set key 1 to the value 1 and try to *get* the value of key 1 after that, i.e. consider $get(set(m, 1, 1), 1)$ By the above interpretation $set(m, 1, 1) = m$, as all paths in m are infinite, and further $get(m, 1) =$ `Nothing` thereby refuting the proposition despite being a model of the ADT theory and the definitions of `get` and `set`. Intuitively, `set` loses the update entirely, and hence `get` returns `Nothing`.

Attempt 2: Inductive Proof We cannot prove the law directly. Instead, we need an inductive proof as in Figure 2.2.

```
getEq (Node key _ l r) k v
  | k == key    = ()
  | k <  key    = getEq l k v
  | otherwise   = getEq r k v
getEq Leaf _ _ = ()
```

The successful proof splits cases on constructor for `m` and then on the ordering of the two keys, and recursively invokes `getEq` (i.e. applies the induction hypothesis) on the left and right subtrees appropriately. The induction hypotheses in the antecedents of the VC for the above, as in `zeroR` eliminates the rogue nonstandard models, and allows PLE to find suitable instantiations such that the resulting instantiated VC can be validated by the SMT solver.

2.7 FLUID REASONING AND REASONING IN LEON

The LEON system and its successor STAINLESS [26] reason with functional programs [33, 41, 46] using techniques broadly similar to LH and UQFR. LEON reasons about Scala programs

[14] A non-regular tree is one that is not isomorphic to any of its proper subtrees. This is a technical condition we require to ensure that the ADTs are *acyclic*, i.e., it is not possible to reach a term by destructing itself.

with quantifier-free pre/post conditions, and the recursively defined functions occurring in annotations are written in Scala as terminating functions. It also automates certain induction proofs, including induction by "stack-height" (akin to Hoare-style reasoning), as well as structural induction on ADTs. LEON caters to other aspects of development as well, including techniques similar to bounded model-checking for finding errors. We do not discuss these aspects here as they are not relevant to our work.

The first observation is that verification conditions for LEON programs can also be modeled as (DEF, φ) in the FLUID fragment. Specifically, the property φ is quantifier-free (implicitly universally quantified) and definitions are proven terminating.

While reasoning in LEON also involves unfolding definitions followed by SMT solving, there are important differences in comparison to LH or UQFR. First, whereas LH typically unfolds definitions only once, LEON continually unfolds definitions over multiple rounds similar to UQFR. Second, LEON asserts the contract of a function along with its definition on the given input arguments during the unfolding. In theory, it does so for *every* unfolding, *ad infinitum*. Observe that when expressing a problem as (DEF, φ), contracts cannot be assumed for all subsequent function calls, as that would require universally quantified assumptions. Assuming contracts for subsequent function calls is also strictly more powerful than simply unfolding definitions, as we illustrated in Section 2.2.2.

Reduction to UQFR and Completeness for LEON We show that the reasoning mechanism in LEON can in fact be captured in the FLUID framework and proven using UQFR. More formally, given a pair (DEF, φ) in the FLUID fragment with contracts $\{(pre_D, post_D)\}_{D \in \mathcal{D}}$ for the functions, we construct an effectively computable instance (DEF', φ') in the FLUID fragment such that running UQFR on (DEF', φ') mimics assuming the contracts in addition to unfolding definitions. The key idea is to construct, for every $D \in \mathcal{D}$, an additional recursively defined predicate $Contract_D$ with the same input signature that returns a Boolean value indicating whether the pre/post condition holds for the input parameters as well as all recursive calls D makes in the computation on these parameters. We then check validity of verification conditions that further assume that the immediate calls to other functions D have $Contract_D$ evaluate to *true*. UQFR applied on this formula mimics the procedure that LEON does and Theorem 2.3 argues the completeness with respect to the underlying combined theory.

We assume for simplicity an ADT universe with two destructors d_1 and d_2 that map to the ADT sort and other destructors that map to background sorts, and a single nullary constructor Nil. We also assume that there is only one recursively defined function f that takes arguments $(x, \overline{y})$ where x is of an ADT sort. Let the ADT sort have a single nullary

constructor *Nil*. Without loss of generality, let us also suppose that the definition of f on arguments $(x, \overline{y})$ has two recursive calls with the arguments being $(d_1(x), \overline{t_1}(x, \overline{y}))$ and $(d_2(x), \overline{t_2}(x, \overline{y}))$ where $\overline{t_1}$ and $\overline{t_2}$ are tuples of terms over $x, \overline{y}$. This definition is in the FLUID fragment, i.e., provably acyclic as the first parameter decreases in the subterm ordering on the ADT sort. Finally, let us assume a postcondition $post_f(x, \overline{y}, \rho)$ for f, where x of type ADT and $\overline{y}$ are the input parameters for f and ρ is a variable denoting the return value of f on $(x, \overline{y})$. We assume that there is no precondition for ease of exposition.

Let φ be a property that we want to prove. The VC is then $DEF \rightarrow \varphi$ where DEF contains the definition of f. Without loss of generality, let f occur in φ as the term $f(z, \overline{w})$. In order to model LEON's procedure, when proving φ valid we need to be able to assume that $post_f(x, \overline{y}, f(x, \overline{y}))$ holds on arguments obtained by unfolding the definition of f on $(z, \overline{w})$ arbitrarily many times.

Let us define a new recursive predicate $Contract(x, \overline{y})$ with input parameters identical to f and the following definition:

$$\begin{aligned} \forall x.\ \ Contract(x, \overline{y}) = (&post_f(x, \overline{y}, f(x, \overline{y})) \wedge \\ &\big(x \neq Nil \rightarrow (Contract(d_1(x), \overline{t_1}(x, \overline{y})) \wedge Contract(d_2(x)), \overline{t_2}(x, \overline{y}))\big)\big) \end{aligned} \tag{2.22}$$

The above declares $Contract(x, \overline{y})$ to be true iff f satisfies its contract on the input $x, \overline{y}$ and further, if x is not *Nil*, *Contract* also holds for the recursive calls $(d_1(x), \overline{t_1}(x, \overline{y}))$ and $(d_2(x), \overline{t_2}(x, \overline{y}))$ that occur in the definition of f. Therefore, asserting $Contract(x, \overline{y})$ can be seen as asserting that the contract of f holds for $(x, \overline{y})$ as well as all the tuples that occur when unfolding the definition of f on $(x, \overline{y})$ *ad infinitum*, i.e., the computational closure of $(x, \overline{y})$ (see Section 2.5.2).

Finally, in order to simulate LEON-style reasoning we want to assert contracts for all tuples occurring in the unfolding of $f(z, \overline{w})$ (but not the contract for the arguments themselves). We do this by explicitly asserting *Contract* for the 'first level' of terms in the unfolding of f and adding *Contract* to the *DEF*:

$$DEF' \rightarrow \big(\big(z \neq Nil \rightarrow Contract(d_1(z), \overline{t_1}(z, \overline{w})) \wedge Contract(d_2(z), \overline{t_2}(z, \overline{w}))\big) \rightarrow \varphi\big) \tag{2.23}$$

where DEF' is the union of *DEF* and the definition for *Contract* as above. As argued earlier, given the definition of *Contract*, asserting *Contract* on the first level tuples amounts to asserting the contract for all tuples that occur in the unfolding of $f(z, \overline{w})$.

Observe that the new VC is also in the FLUID fragment. Furthermore, when we unfold def-

initions, unfolding the definition of *Contract* naturally leads to assuming (in the instantiated quantifier-free formula) that $post_f$ holds on the arguments that occur when unfolding the definition of f on $(z, \overline{w})$. Hence applying UQFR to the above constructed formula essentially simulates the procedure that LEON performs. It is easy to see that the above construction can be generalized to multiple ADT sorts with different signatures as well as multiple mutually recursively defined functions with their respective contracts.

As far as we know the above result is new. Prior literature on LEON [33, 41, 46] shows soundness of the procedure. Restricted fragments [41] involving certain kinds of "measures" (functions from ADTs to background sorts) have been shown to admit complete unfolding based reasoning with respect to the *standard model*, with a *decidable* validity problem. In contrast, we show completeness (i.e., recursively enumerable procedures) for validity with respect to the *combined theory* for a more general class of functions. Further, our logic is *undecidable* (see Section 2.8), which shows that it is fundamentally different from decidable subclasses reported in prior art [41] (see also Chapter 6).

Our results also show that when theorems are not provable in LEON, there ought to be rogue nonstandard models. We considered a few such examples and were indeed able to construct rogue nonstandard models. For example, LEON fails to prove $rev(rev(x)) = x$ automatically, where *rev* reverses a list. It has a rogue nonstandard model that is eliminated by an inductive lemma provided by the user.

2.8 EXPRESSIVENESS RESULTS ON THE FLUID FRAGMENT

We show some technical results pertaining to the FLUID fragment.

Undecidability of the FLUID Fragment We show undecidability of the FLUID fragment even when the combined theory admits decision procedures for quantifier-free reasoning. In other words, the validity problem for the FLUID fragment, for which we proved UQFR is a semi-decision procedure in Section 2.5.2, does not admit any decision procedures.

Theorem 2.5. The validity problem for FLUID formulas is undecidable.

We provide a reduction from the non-halting problem for two-counter machines. A two-counter machine [47] is a machine with two registers that can contain unbounded integers. The machine can only increment or decrement these counters, or check whether they are equal to zero. Two-counter machines are computationally equivalent to Turing machines [47], and checking the halting/non-halting of a two-counter machine is undecidable (assuming, without loss of generality, an initial configuration where counters are set to zero).

It is tempting to try to find a simple reduction that encodes executions of the machine using ADTs (say, as lists of configurations), defining a recursive predicate that identifies *halting* executions (which are finite), and stating the theorem that no ADT element encodes a halting execution of the machine. However, note that we are seeking validity with respect to the *combined theory* and not validity in the standard model. In fact, since validity over the combined theory is recursively enumerable, we cannot reduce non-halting problem of two counter machines (which is co-r.e. hard) to it. Our reduction reduces the non-halting problem to the complement of validity, i.e., satisfiability. We provide the proof below.

Proof. Let us fix a two-counter machine M. Let us consider ADTs that are lists of triples of integers: ADT List, with two constructors Nil and Cons:

```
data List = Nil | Cons (state : Int) (fst : Int) (snd : Int) (tail
    : List)
```

Each element of the list represents a configuration of a two-counter machine– the state of the two-counter machine and the value of the two counters. We can write quantifier-free logical formulae $\mathsf{init}(x)$ representing the initial configuration, $\mathsf{halt}(x)$ representing any halting configuration, and $\mathsf{nextconfig}(x, y)$ representing that y is the successor configuration of x. Now consider the following recursive definition def_{nonhalt}:

$$\begin{aligned}\forall x : \mathsf{List}.\, \mathsf{nonhalt}(x) = \mathsf{ite}(x = \mathit{Nil}, \mathit{False},\\ \mathsf{ite}(\mathsf{halt}(x), \mathit{False}\\ \mathsf{nextconfig}(x, \mathit{tail}(x)) \wedge \mathsf{nonhalt}(\mathit{tail}(x))\end{aligned}$$

Note that this function recurses on $\mathit{tail}(x)$, which is a strict syntactic subterm of x. Thus its definition meets the requirement of the FLUID fragment.

Consider the following property to prove valid under the above definition:

$$\varphi \equiv \forall x.(\mathsf{init}(x) \rightarrow \neg\ \mathsf{nonhalt}(x))$$

We claim $def_{\mathsf{nonhalt}} \rightarrow \varphi$ is valid in the combined theory if and only if the two-counter machine *halts*.

If the formula is not valid, then there is a model and an ADT element x such that $\mathsf{init}(x)$ and $\mathsf{nonhalt}(x)$ hold. Then there are two cases: x corresponds to a finite list (reaching Nil in finitely many destructions) or it is a nonstandard element corresponding to an infinite list. The former case is impossible, as no finite list can have nonhalt to be true on it. In the second case, the recursive definition of nonhalt ensures that the list pointed to by x encodes an execution of the two-counter machine, and hence the machine does not halt.

Conversely, assume the machine does not halt. It turns out that we can build a nonstandard model where x points to a nonstandard ADT element encoding an infinite list that corresponds to the non-halting execution of the machine. Formally, we use the compactness theorem instead of constructing this model explicitly. Note that for any $k \in \mathbb{N}$, there is a *standard* ADT element that encodes a finite list corresponding to a partial execution of the two-counter machine for k steps. Hence any *unfolding* of the definition of $\mathsf{nonhalt}$ on x, $tail(x)$, etc. up to k destructions is satisfiable. By the compactness theorem, the unfolding for ω number of steps is also satisfiable. One can now see that $\mathsf{nonhalt}(x)$ will hold in this model. QED.

Incompleteness with Terminating Definitions It is natural to ask whether UQFR is complete for all terminating functions, not just provably acyclic ones. We show that this is not the case.

Theorem 2.6. There exists a signature $(\mathcal{S}, \mathcal{F}, \mathcal{D}, \mathcal{T}_{comb})$, a set DEF of well-defined definitions for $\mathcal{D}$ that are not provably acyclic, and a universally quantified formula φ such that $\mathcal{T}_{comb} \models (DEF, \varphi)$ but UQFR does not terminate.

Note that the above result implies that generalizing definitions to arbitrary universally quantified formulas also leads to incompleteness.

Proof. We construct an instance of the validity problem without provably acyclic definitions that is unprovable for UQFR. We use the ADT of lists over integers as our foreground sort:

```
data List = Nil | Cons (head : Int) (tail : List)
```

as well as the following definitions:

$$def_{std} \equiv \forall x : \mathsf{List}.\ std(x) = \mathsf{ite}(x = Nil, True, std(tail(x)))$$

$$def_{R} \equiv \forall x : \mathsf{List}.\ R(x) = \mathsf{ite}(std(x), True, \mathsf{ite}(head(x) = 0, R(x), \neg R(x)))$$

Both functions are well-defined definitions as they terminate on the standard model. The termination of std is apparent: it simply destructs the input term recursively until Nil and then returns $True$. R is also terminating on the standard model since $std(x)$ is always true on standard elements, therefore the *else* branch of the outer ite is never taken.

std is also provably acyclic since the arguments to the recursive call are smaller according to the subterm ordering. However, R is not provably acyclic. We demonstrate this indirectly by proving the incompleteness of UQFR and defer the discussion of why it is not provably acyclic.

Consider the theorem $\varphi \equiv \forall y : \mathsf{List}.\ R(y)$. We claim: (1) $\mathcal{T}_{comb} \models DEF \rightarrow \varphi$, and (2) UQFR does not terminate on (DEF, φ). We do not prove the latter here as it can be deduced easily by following Algorithm 2.1 and only show the former.

Suppose the claim is not true. Note that the antecedent is not vacuous since it is possible to satisfy the definitions on the standard model. Therefore, for the claim to be false there must exist a model where the definitions are satisfied, but there is a y such that $R(y)$ does not hold. From the definition of R, we know that this is only possible when $\neg std(x)$ and $head(x) = 0$. Any other path in the definition either leads to $R(x)$ being true or the impossibility $R(x) = \neg R(x)$. Now, consider the element $Cons(1, y)$. From the definition of std, we have $std(Cons(1, y)) = std(y) = False$. Then, following the definition of R yields $R(Cons(1, y)) = \neg R(Cons(1, y))$, which is impossible. Therefore, it must be the case that there is no model where the definitions are satisfied but φ does not hold. In other words, $\mathcal{T}_{comb} \models (DEF, \varphi)$.

UQFR never terminates on the algorithm because unfolding the definitions only ever produce terms that are destructions of y, whereas we proved the validity above by instantiating the definitions on a superterm of y. This shows that UQFR is incomplete for this instance. QED.

Since we prove completeness for provably acyclic definitions, the above shows that R is not provably acyclic. More specifically, this is because in order to prove that the absurd recursive call $\neg R(x)$ is unreachable, we must essentially prove that $std(x)$ always holds. However, this not true in the combined theory, and one would typically use induction to establish this.

Consequently, in models where $std(y)$ does not hold for some y, R is in fact unrealizable as the element $Cons(1, y)$ cannot be given a valuation that is consistent with the definition of R. This does not happen with provably acyclic definitions because such functions can always be given a valuation consistent with their definitions on any model (see Theorem 2.1).

The completeness of UQFR arises from the fact that unfolding the (provably acyclic) definition of some R ($R \in \mathcal{D}$) on x amounts to a simulation of the "computation" of $R(x)$ in any model. Without provable acyclicity, we have shown that it is possible to construct well-defined definitions that are unrealizable in some models, rendering UQFR incomplete.

Chapter 3: Model-Guided Synthesis of Inductive Lemmas for FO+*lfp*

In the previous chapter we formally identified creativity gaps in automated verification techniques for recursive programs over algebraic datatypes. Our technical results studied completeness of unfolding for first-order logics with recursive definitions over algebraic datatypes, and as a result characterized the role of user-provided inductive lemmas as bridging the gap between models of combined theories and the standard model. Earlier related work [27] studied the completeness of a UQFR-like heuristic for first-order logics with recursive definitions over uninterpreted sorts. These logics are typically used to model verification of heap-manipulating programs. This work showed similarly that inductive lemmas bridge the gap between the combined theories and a class of standard models (defined by least fixpoints over the uninterpreted sorts). In this chapter, we undertake a novel approach for synthesizing inductive lemmas to prove validity in this logic. The idea is to utilize several kinds of finite first-order models as counterexamples that capture the non-provability and invalidity of formulas to guide the search for inductive hypotheses. We implement our procedures and evaluate them extensively over theorems involving heap data structures that require inductive proofs and demonstrate the effectiveness of our methodology.

3.1 INTRODUCTION

One of the key revolutions that has spurred program verification is *automated reasoning of logics*. Particularly, in deductive verification, engineers write inductive invariants that punctuate recursive loops and contracts for methods and then use logical analysis to reason with *verification conditions* that correspond to correctness of small, loop-free snippets. In this realm, automatic reasoning in combinations of quantifier-free theories using SMT solvers has been particularly useful; in turn, these tools are based on the logics having a *decidable* validity (and satisfiability) problem [12, 28].

However, reasoning even with loop-free snippets of programs is challenging when the code manipulates *linked data structures embedded in pointer-based heaps*. Data structures are finite but unbounded structures that are often characterized using recursive definitions whose semantics are defined using both quantifiers and least fixpoints.

First-order logic with least fixpoint definitions (FO+*lfp*) which accesses various background sorts or theories (e.g., integers and sets) is a powerful extension of FOL that can define data

The material in this chapter is reproduced from the publication cited as [15] co-authored by the author of this thesis, with minor changes.

structures and express their properties. For example, fairly expressive dialects of separation logic have been translated to FO+*lfp* in order to aid automated reasoning [27, 48, 49, 50]. We focus here on automated reasoning for first-order logics with least fixpoint definitions or recursive definitions that utilize SMT solvers for quantifier-free reasoning.

The novel automation of FO+*lfp* reasoning that we propose is a counterexample-guided synthesis of inductive lemmas utilizing *complete* procedures for pure first-order (FO) reasoning. Our framework requires the FO reasoning procedure to be able to compute counterexample models. The technique we present can be parameterized over any FO reasoning engine able to provide counterexamples for provability. In our work we use a particular technique called *natural proofs* that are based on systematic quantifier instantiation [27] and that is able to provide such counterexamples.

The Anatomy of Proofs for FO+*lfp*: Proofs by Induction Unlike FOL, FO+*lfp* does not admit complete procedures, i.e., sound proof systems for FO+*lfp* cannot admit proofs for every theorem. Quick proof: given a Peano "number line", true addition and multiplication are definable using *lfp*. Hence by Gödel's incompleteness theorem [42], even quantifier-free FO+*lfp* has an undecidable validity (and satisfiability) problem.

Humans usually prove properties involving recursive definitions (or least fixpoints) using *induction*. We consider logics with recursive definitions, where each recursive definition is of the form $\forall \overline{x}.\ R(\overline{x}) :=_{lfp} \rho(\overline{x})$. Theorems are expressed using first-order logic over a signature that includes these recursive definitions. An inductive proof of a theorem typically involves sub-proofs, each of which identify a fairly strong property (the induction hypothesis) and its proof (the *induction step*). We use a more general notion of induction proofs based on pre-fixpoints, not requiring a concept of size or measure based on natural numbers upon which to induct. We defer this notion until later and instead encourage the reader to simply think of an inductive hypothesis as an *inductive lemma* and the induction step of the lemma as the *pre-fixpoint (PFP) of the lemma*.

In this chapter we propose to build automated reasoning for FO+*lfp* with background theories using a combination of (a) complete procedures for FO reasoning to prove theorems and PFPs of lemmas, and (b) counterexample-guided expression synthesis for synthesizing lemmas (i.e., induction hypotheses) that aid in proving a theorem.

We observe that proofs of the induction step (PFP) of the formula can be seen as reasoning using *pure first-order logic reasoning without induction*. More precisely, we can think of a proof of a theorem in FO+*lfp* as split into sub-proofs mediated by an *induction principle* but otherwise consisting of pure FO reasoning. The induction principle says that proving the PFP (induction step) of any lemma proves the lemma.

We can thus view the structure of an induction proof of a theorem α as identifying a finite set $\mathcal{L} = \{L_1, \ldots, L_n\}$ of lemmas such that:

- For each $i \in \{1, \ldots, n\}$, there is a purely FO proof of $PFP(L_i)$ using the earlier lemmas $L_1, \ldots, L_{i-1}$ as assumptions, and
- There is a purely FO proof of α with the lemmas from $\mathcal{L}$ as assumptions.

Notice that proofs of the above form lack any explicit induction proof and the purely FO proofs consider each relation R is interpreted as a fixpoint definition (not least fixpoint) of the form $\forall \overline{x}.\ R(\overline{x}) \iff \rho(\overline{x})$ rather than $\forall \overline{x}.\ R(\overline{x}) :=_{lfp} \rho(\overline{x})$. The fact that proving $PFP(L_i)$ suffices as a proof of L_i is implicit and appeals to the least fixpoint semantics of recursive definitions only to argue that the above constitutes a proof of the theorem.

This view of an inductive proof of an FO+*lfp* formula as pure FO proofs mediated by induction principles suggests a "synthesis + reasoning" methodology: (a) synthesize lemmas that are *likely* to be true and inductively provable, and (b) prove theorems and lemmas using pure FO reasoning. This idea is itself not new. For example, the induction axiom schema in Peano arithmetic is:

$$\forall \overline{y}.\ (\varphi(0, \overline{y}) \wedge (\forall x.\ \varphi(x, \overline{y}) \Rightarrow \varphi(S(x), \overline{y}))) \Rightarrow \forall x.\ \varphi(x, \overline{y}) \tag{3.1}$$

for *any* formula φ. A proof using this axiom can hence be seen as divining formulas φ and proving lemmas of form $\forall x.\ \varphi(x, \overline{y})$ by using purely first-order logic over the non-inductive axioms to prove $\forall \overline{y}.\ (\varphi(0, \overline{y}) \wedge (\forall x.\ \varphi(x, \overline{y}) \Rightarrow \varphi(S(x), \overline{y})))$.

The idea of finding proofs by induction by synthesizing inductive hypotheses and proving them using simpler non-inductive reasoning is also not new. For example, in program verification, inductive hypotheses are written as loop invariants or method contracts that capture invariants of program states or effects of calling procedures. Synthesizing such invariants and contracts has been explored using a combination of inductive synthesis and reasoning (see work on the ICE framework [51], for example, that explicitly takes this approach, and also the related work section). The novelty of our work lies in realizing this technique for proving theorems in FO+lfp using finite models that witness invalidity and non-provability for counterexample-guided synthesis.

Synthesizing Inductive Lemmas The primary technical contributions of this chapter lie in techniques for synthesizing lemmas that (a) can be proved inductively, with their own statement as the induction hypothesis, and (b) aid the proof of a target theorem. We embrace the paradigm of *counterexample-guided synthesis* that has met impressive success in automating verification and synthesis (e.g., in finding predicates for abstraction [5, 52] or

in program synthesis through the CEGIS paradigm [53, 54, 55]). The salient feature of our technique is the use of *finite FO models* as counterexamples to guide the search for lemmas.

Suppose a theorem α in FO+*lfp* is desired to be proved valid. Our technique for automated quantified FO reasoning (without least fixpoints), called *natural proofs*, uses systematic quantifier instantiation followed by SMT-based validation of the resulting quantifier-free formula [27, 50, 56]. Let $SQI(k)$ be the method that systematically instantiates terms of depth k for quantified variables then checks satisfiability of the resulting quantifier-free formula (the latter is a decidable problem). As a simple consequence of Herbrand's theorem and compactness, we know that this method is complete in the sense that if β is a valid formula in FOL, then there is some k for which $SQI(k)$ will prove the validity of β.

At any point of the lemma synthesis procedure, we would have synthesized a set of potentially useful lemmas already proved valid and then seek a new lemma to help prove α.

We utilize *three* kinds of counterexample models to guide the search for useful and provable lemmas. In our iterative framework for synthesizing useful and provable lemmas, a prover and a synthesizer interact: the synthesizer proposes lemmas, and the prover provides constraints for synthesizing new lemmas. When the synthesizer proposes a lemma, the lemma can be (a) valid and provable using $SQI(k)$ reasoning using existing lemmas, (b) invalid but easily shown to be so using a small model, or (c) valid or invalid, but in either case not provable using $SQI(k)$ and existing lemmas. Note that (a) and (c) are already exhaustive cases, and (b) overlaps with (c).

These correspond to the three kinds of counterexamples, which we now name. *Type−1* models guide the search toward lemmas that help prove the theorem α and are obtained from the failure to prove α using FO reasoning via $SQI(k)$. *Type−2* models are small counterexamples to validity of proposed lemmas and are obtained by searching for bounded models using SMT solvers. *Type−3* models show non-provability of lemmas and are obtained from failure to prove the PFP of lemmas using FO reasoning via $SQI(k)$. We narrow and guide the search space for lemma synthesis using these three kinds of counterexamples.

The main contribution of this chapter is FOSSIL, a novel algorithmic framework for synthesizing lemmas that uses such counterexamples and proves both lemmas and target theorems using FO reasoning. In each round, the algorithm begins with a target theorem α and tries proving it using the lemmas synthesized and proved valid so far. If the proof of α fails, this failure precipitates a *Type−1* counterexample which will be used to guide the search towards lemmas that do help prove the theorem α. The lemma synthesis phase follows, generating a lemma that satisfies the *Type−1* counterexample and then attempting to prove the validity of its PFP. If the proof of the PFP fails, this failure yields either a *Type−2* counterexample (which is a bounded model) if possible or otherwise a *Type−3*

counterexample to show non-provability of the PFP. We continue to seek new lemmas guided by the counterexamples until a valid lemma is found, at which point we add the new lemma to our set of valid lemmas. We recurse, trying to prove the target theorem α. Off-the-shelf synthesis tools do not scale when employed in our framework; however, our algorithm works efficiently via constraint solving with SMT solvers, carefully representing counterexamples as *ground formulas* and formulating synthesis constraints as ground constraints.

Background Theories and Relative Completeness The techniques for inductive reasoning that we develop in this chapter are more involved than as described above. First, many applications, such as program verification, require handling of domains that are constrained to satisfy certain theories, such as arithmetic and sets (sets allow the expression of collections such as "the set of keys stored in a list" in heap-based verification and "the set of heap locations that constitute a list" in heaplets for frame reasoning). Consequently, our framework maintains a *foreground sort* modeling the heap with pointers as well as multiple background sorts, with the background sorts constrained by theories and that admit Nelson-Oppen style decision procedures for *quantifier-free* reasoning. In such settings, the work in [27] proved that for formulas that quantify only over the foreground sort (i.e., only involving quantification over locations of the heap), systematic quantifier instantiation is still complete. Moreover, satisfiability of quantifier-free formulas after instantiation are supported by SMT solvers, which can also return the three kinds of counterexamples we seek.

Second, we carefully build lemma search to admit relative completeness. We show that if there is a proof of a theorem involving finitely many independently provable lemmas (in the grammar of lemmas provided by the user), then our procedure is guaranteed to eventually find one. More precisely, there are two *infinities* to explore—one is the search for lemmas and the other is the instantiation depth k chosen for finding proofs. As long as our procedure fairly dovetails between these two infinities, it is guaranteed to find a proof.

Evaluation We implement and evaluate our procedure for a logic that combines an uninterpreted foreground sort with background sorts, where background sorts have quantifier-free fragments that are decidable using SMT solvers. Our tool can employ generic syntax-guided synthesis (SyGuS) engines as well as a custom synthesis tool we built; both of these can synthesize lemmas using FO countermodels that are encoded using logical constraints.

We perform an extensive evaluation on two suites of benchmarks: one of 50 theorems on data structure verification and another of 673 synthetically generated theorems. Our experiments give evidence that the first-order counterexample-based techniques we develop here are effective in synthesizing inductive lemmas and proving theorems. Apart from

evaluating the efficiency of our tool, we evaluate several design decisions and optimizations in our tool. In particular, we study the efficacy of using various kinds of counterexamples and compare our custom synthesis engine with off-the-shelf state-of-the-art synthesis engines.

Contributions The main contributions of this chapter are: (1) a counterexample-guided synthesis framework, FOSSIL, for synthesizing inductive lemmas for proving validity in FO+*lfp* with relative completeness guarantees, (2) the formulation of three kinds of counterexamples that guide synthesis towards lemmas that are relevant to the theorem, lemmas that hold at least on small models, and are provable using induction, (3) efficient synthesis algorithms using specifications formulated as ground formulas, and (4) an implementation and evaluation of FOSSIL on two benchmark suites of theorems over heap data structures[1].

Outline This chapter is structured as follows. In Section 3.2, we define the logic used in this paper (i.e., FO+*lfp*) and the notion of counterexample models, and formalize the problem of sequential lemma synthesis that we study. In Section 3.3 we present our main contribution: the FOSSIL algorithm for solving the sequential lemma synthesis problem, and furnish a full run of the algorithm on a running example. Section 3.3 also formalizes the components for checking FO+*lfp* validity, counterexample generation, and model-guided synthesis of lemmas used by the core FOSSIL algorithm. Section 3.4 describes how to implement these components. In Section 3.5 we prove that FOSSIL is relatively complete for the problem of synthesizing independent lemmas and indicate directions for building complete algorithms for sequential lemma synthesis. Section 3.6 details the implementation and evaluation of FOSSIL on two benchmark suites of FO+*lfp* theorems that we create, including several ablation studies that show the efficacy of various choices.

3.2 PRELIMINARIES AND PROBLEM DEFINITION

In this section, we define the first-order logic framework we work with (first-order logic with recursive definitions that have *lfp* semantics) and give the problem definition for solving theorems in FO+*lfp* using synthesis of inductive lemmas and first-order proofs.

3.2.1 First-Order Logic over Theory-Constrained Background Sorts

The first-order logics (with and without recursive definitions) that we work with are over a multisorted universe that has a single distinguished *foreground* sort and multiple *background*

[1]Our benchmarks and tool can be found at: https://github.com/muraliadithya/FOSSIL

sorts. The universes of all these sorts are pairwise disjoint. The foreground sort and the functions and relations that refer to it (as part of the domain or codomain) are entirely uninterpreted (no axioms that constrain them). Background sorts and functions and relations involving only background sorts are constrained by certain theories.

Formally, we work with a signature of the form $\Sigma = (S; C; F; \mathcal{R})$, where S is a finite non-empty set of sorts. C is a set of constant symbols, where each $c \in C$ has some sort $\sigma \in S$. F is a set of function symbols, where each function $f \in F$ has a type of the form $\sigma_1 \times \ldots \times \sigma_m \to \sigma$ for some m, with $\sigma_i, \sigma \in S$. $\mathcal{R}$ is a set of relation symbols, where each relation $R \in \mathcal{R}$ has a type of the form $\sigma_1 \times \ldots \times \sigma_m$.

We assume a designated foreground sort, denoted by σ_{f}. All other sorts in S are called background sorts, and for each such background sort σ we allow the constant symbols of type σ, function symbols that have type $\sigma^n \to \sigma$ for some n, and relation symbols that have type σ^m for some m to be constrained using an arbitrary theory T_σ. All other functions and relations that involve either the foreground sort or multiple background sorts are assumed to be uninterpreted (not constrained by any theory). We consider standard first-order logic (FO) over these multisorted signatures, with standard syntax and semantics, under the combined theories [42].

Counterexamples We require that validity of *quantifier-free* logic under the combined theories is *decidable*. Furthermore, when a quantifier-free formula is not valid, we require this decision procedure to provide models that show satisfiability of the negation of the formula. The truth value of the quantifier-free formula only depends on a finite portion of the model (corresponding to the terms used in the formula, since the formula is quantifier-free). This finite portion can be described by a conjunction of atomic ground formulae. We require models to be given indirectly by such *conjunctive ground formulae*. Formally, given a quantifier-free formula φ that is satisfiable, we require that the solver return a conjunctive ground formula *gf* such that (a) *gf* is satisfiable and (b) $gf \Rightarrow \varphi$ is *valid*. If φ contains variables, then these are interpreted as or replaced by Skolem constants that are part of the signature of *gf*. Intuitively, *gf* indicates the existence of one or more models such that φ is satisfied on all of them. The formula *gf* encodes enough information about these models to ensure that φ is satisfied in them. The following example illustrates these ideas.

Example 3.1 (Counterexample models as conjunctive ground formulas)**.** Consider the formula $(f(x) = y) \Rightarrow y > 3$ where $f : \sigma_0 \to \mathit{Int}$ is an uninterpreted function, x is of the sort σ_0, and y is of sort Int. This formula is invalid, and we can witness the satisfiability of its negation $(f(x) = y) \wedge \neg(y > 3)$ using a model $\mathcal{M}$ where x is interpreted to an element u and

$f(u)$ is interpreted to 2. $\mathcal{M}$ can be captured using the formula $gf : f(x) = 2$. Indeed, one can see that $\big(f(x) = 2\big) \Rightarrow \big((f(x) = y) \wedge \neg(y > 3)\big)$ is a valid formula. It is also imminent that gf is satisfiable since $\mathcal{M}$ realizes it.

In our tools, we work with certain Nelson-Oppen combinable decidable theories [11, 28, 29, 30] (in particular linear arithmetic over integers, sets of integers). These are supported by SMT solvers that guarantee both decidability of quantifier-free formulae as well as model generation as above.

3.2.2 First-Order Logic with Recursive Definitions (FO+*lfp*)

Our target theorems are in a dialect of first-order logic over a multisorted universe (universes similar to the one above) but with recursive definitions that have least fixpoint semantics.

We identify a subset $\mathcal{R}^{rec}$ of the relational symbols $\mathcal{R}$ and endow them with definitions; these relations are not directly interpreted by models, rather they are defined uniquely by their definitions. In our work we assume that these recursive definitions only relate elements of the foreground sort. The set of recursive definitions $\mathcal{D}$ for the symbols $\mathcal{R}^{rec}$ are of the form

$$R(\overline{x}) :=_{lfp} \rho_R(\overline{x}) \tag{3.2}$$

where $R \in \mathcal{R}^{rec}$, $\overline{x}$ are variables over the foreground sort, and $\rho_R(\overline{x})$ is a quantifier-free first-order logic formula. Note that a definition ρ_R can utilize all the sorts and functions/relations in the model. We also assume that there is only one definition for each $R \in \mathcal{R}^{rec}$.

To ensure the well-definedness of definitions, we assume that the symbols in $\mathcal{R}^{rec}$ are ordered in layers, and that each $R' \in \mathcal{R}^{rec}$ that occurs in the definition of R is either in a smaller layer, or it is in the same layer and only occurs positively (under an even number of negations) in the definition of R (similar to stratified Datalog [57]). The semantics of recursively defined relations is given by the least fixpoint (*lfp*) that satisfies the relational equations (the condition that each recursive definition only refers positively to recursively defined relations in the same layer ensures that the least fixpoint exists [58])[2].

Our theoretical treatment assumes that there is only one layer for simplicity. Therefore, each recursive definition only mentions other recursively defined relations positively. However, the results also hold for several layers of recursive definitions, and indeed our experiments utilize them.

[2]Our definition of FO+*lfp* is similar to the one used in Finite Model Theory: see Libkin [59], Chapter 10. Notably, our notion of recursive definitions is more restrictive than general FO+*lfp* because recursive definitions should only be universally quantified and only over the foreground sort. This technical condition enables us to build effective complete FO validity procedures: see Section 3.2.4.

Example 3.2 (Linked Lists)**.** Let n be a unary function symbol modeling a pointer of type $\sigma_0 \to \sigma_0$, i.e., from the foreground sort to the foreground sort. Let *nil* be a constant of sort σ_0, and *list* be a unary relation with the recursive definition

$$list(x) :=_{lfp} \mathsf{ite}(x = Nil, true, list(n(x)))$$

Then, in any model $\mathcal{M}$ where *list* is interpreted using its *lfp* definition, *list* holds precisely for those elements that are the head of a *finite* linked list with n as the next pointer. FOL without *lfp* cannot describe such linked lists [59]. Note that unlike Algebraic Datatypes (ADTs), if $list(x) \wedge list(y) \wedge x \neq y$ holds in a model, the lists pointed to by x and y are not necessarily disjoint and could "merge" in the model. We can also model disjointedness using heaplets, as we show in the following example.

Example 3.3 (Trees and Heaplets)**.** Consider the following recursive definition for a predicate $tree(x)$ which expresses that x is the root of a binary tree on pointers l (left) and r (right):

$$\begin{aligned} tree(x) :=_{lfp} ite(x{=}nil,\ true, tree(l(x)) \wedge tree(r(x)) \\ \wedge\ htree(left(x)) \cap htree(right(x)){=}\emptyset \\ \wedge\ Singleton(x) \cap (htree(l(x)) \cup htree(r(x))){=}\emptyset) \\ htree(x) :=_{lfp} ite(x = nil,\ \emptyset, Singleton(x) \cup htree(l(x)) \cup htree(r(x))) \end{aligned}$$

Observe again that since our data structures are unlike ADTs, pointers l and r may possibly point to the same element ("merge") in arbitrary heaps/models. Therefore, to define trees we define a recursive definition for the partial function expressing the *heaplet* of a tree $htree : \sigma_0 \to \sigma_{sl}$ where σ_{sl} is a background theory of sets of locations with which we demand that the left and right subtrees are disjoint. This is similar to constraints used in Separation Logic to express trees [60].

We now state the usual notion of validity/entailment in FO+*lfp*:

Definition 3.1 (FO+*lfp* Entailment)**.** For a sentence α and a set Γ of formulas we write $\Gamma \cup \mathcal{D} \models_{\mathsf{LFP}} \alpha$ if α is true in all models of Γ using the lfp semantics for relations with definitions given in $\mathcal{D}$.

We conclude this section with some remarks.

First-Order Abstractions of Recursive Definitions Given an FO+*lfp* formula, we can sometimes prove it valid using pure FOL. We can do this by interpreting recursive definitions in $\mathcal{D}$ to be *fixpoint definitions* (as opposed to *lfp*). More precisely, we constrain the relations using FOL as $\forall \overline{x}.\ R(\overline{x}) \leftrightarrow \rho_R(\overline{x})$. If α is valid under the fixpoint interpretation of recursive

relations, then it is of course valid using least fixpoint interpretation as well, but the converse does not hold. Interpreting recursive definitions as fixpoint definitions rather than least fixpoint definitions is hence a form of sound abstraction. We write $\Phi \cup \mathcal{D}^{fp} \models_{\mathsf{FO}} \alpha$ to denote that α is valid using the FO fixpoint abstractions $\mathcal{D}^{fp}$ of $\mathcal{D}$.

Partial Functions The reader may have observed in Example 3.3 that we presented a recursively defined *function htree*. Although we don't allow them in the theoretical treatment, our tools support recursively defined partial functions from the foreground sort to both foreground and background sorts (for modeling heaplets of structures, lengths of lists, heights of trees, etc.). However, partial functions can be modeled using two predicates: one recursively defined predicate that captures the domain of the partial function and another predicate defined using only FOL that captures the map of the function.

FO+*lfp* Fragment In this work we only handle the validity of formulas whose quantification is purely over the foreground sort. This fragment is well suited for the domain of heap verification that we study. We can model the heap as the foreground sort and express recursively defined functions and properties that only quantify over the heap. However, it is not as powerful as full FO+*lfp*. For example, the logic cannot talk about array properties (where the array is modeled as a map $f : Int \to V$ from indices to values in a domain V) that quantify over integers, which is a background sort. We also cannot express theorems like "For every positive integer n, there is a linked list of length n" as this requires universal quantification over the background sort. These restrictions are important as they allow us to leverage practical complete algorithms [27] for FOL validity for this restricted fragment in implementing the FOSSIL framework (see Section 3.2.4).

3.2.3 The Inductive Lemma Synthesis Problem for Proving FO+*lfp* Formulas

In this chapter we develop algorithms that prove an FO+*lfp* formula α valid given a finite set $\mathcal{A}$ of axioms and a set $\mathcal{D}$ of recursive definitions with *lfp* semantics. We want to show that $\mathcal{A} \cup \mathcal{D} \models_{\mathsf{LFP}} \alpha$ mainly using first-order reasoning. Clearly, if $\mathcal{A} \cup \mathcal{D}^{fp} \models_{\mathsf{FO}} \alpha$, then $\mathcal{A} \cup \mathcal{D} \models_{\mathsf{LFP}} \alpha$ as argued above.

We use the following running example to illustrate ideas developed in the sequel:

Example 3.4 (Running Example)**.** Consider the recursively defined relation $lseg(x, y)$ defining linked list *segments* between locations x and y on the pointer n:

$$lseg(x, y) :=_{lfp} \mathsf{ite}(x = y, true, lseg(n(x), y))$$

Now, consider the following Hoare Triple:

{pre:$lseg$(x,y1)} if (y1 = nil) then y2 := y1 else y2 := y1.n {post:$lseg$(x,y2)}

The above triple generates the following Verification Condition (VC) α_*:

$$lseg(x, y_1) \Rightarrow \Big(\mathsf{ite}(y_1 = Nil, y_2 = y_1, y_2 = n(y_1)) \Rightarrow lseg(x, y_2)\Big)$$

We denote by $\mathcal{D}_*$ the singleton set containing the definition of $lseg$. We will use the problem of checking $\mathcal{D}_* \models_{\mathsf{LFP}} \alpha_*$ as a running example in this chapter. Note that α_* is actually valid in FO+lfp but it is not FO-valid, i.e., $\mathcal{D}_* \models_{\mathsf{LFP}} \alpha_*$ holds but $\mathcal{D}_*^{fp} \models_{\mathsf{FO}} \alpha_*$ does not. This makes the problem a good candidate for lemma synthesis. We describe a run of our algorithm on this example in Section 3.3.4.

The overall idea in our approach is to use intermediate inductive lemmas to find an FO proof of the goal. We handle a particular fragment of FO+lfp in our work. First, we require the goal α to have quantification only over the foreground sort. Second, we only consider lemmas of the form $L = \forall \overline{x}.\ R(\overline{x}) \Rightarrow \psi(\overline{x})$ for variables $\overline{x}$ over the foreground sort, a quantifier-free formula ψ, and a recursively defined relation $R \in \mathcal{R}^{rec}$. Finally, we prove lemmas valid using a specific form of induction called the pre-fixpoint (PFP) formula. Given a lemma L of the form above, the PFP of L expresses that $R \wedge \psi$ is a pre-fixpoint of the definition of R:

$$PFP(L) := \forall \overline{x}. \rho_R(\overline{x}, R \wedge \psi) \Rightarrow \psi(\overline{x})$$

where $\rho_R(\overline{x}, R \wedge \psi)$ is the formula obtained from $\rho_R(\overline{x})$ by replacing every occurrence of $R(t_1, \ldots, t_k)$ for terms $t_1, \ldots, t_k$ in ρ_R by $\psi(t_1, \ldots, t_k) \wedge R(t_1, \ldots, t_k)$. It turns out that if $PFP(L)$ is FO-valid, then L is a valid FO+lfp formula, as the following theorem states:

Theorem 3.1. [27] If $\mathcal{A} \cup \mathcal{D}^{fp} \models_{\mathsf{FO}} PFP(L)$, then $\mathcal{A} \cup \mathcal{D} \models_{\mathsf{LFP}} L$.

We use the above formalism to define the notion of an *inductive lemma*, as well as the notion of a sequence of lemmas that prove a theorem using FO reasoning.

Definition 3.2 (Inductive Lemmas)**.** A lemma L is inductive for $\mathcal{A} \cup \mathcal{D}^{fp}$ if $\mathcal{A} \cup \mathcal{D}^{fp} \models_{\mathsf{FO}} PFP(L)$. If $\mathcal{A}$ and $\mathcal{D}$ are clear from the context, we just say that L is inductive.

Example 3.5 (Running Example: Inductive Lemma)**.** Consider in the setting of Example 3.4 the following lemma L_*:

$$\forall x, y_1, y_2.\ lseg(x, y_1) \Rightarrow \Big(lseg(y_1, y_2) \Rightarrow lseg(x, y_2)\Big) \qquad (L_*)$$

which expresses that if we have a list segment pointed to by x until y_1, as well as one pointed to by y_1 until y_2, then x points to a list segment until y_2. It turns out that L_* is inductive i.e.,

$\mathcal{D}^{fp}_* \models_{\mathsf{FO}} PFP(L_*)$. In other words, the PFP of the lemma is provable in pure FOL, without induction, and with FO abstractions of the definitions (fixpoint instead of least fixpoint).

The crucial part of the proof is the following subformula of $PFP(L_*)$:

$$\forall x, y_1, y_2.\ \big(lseg(y_1, y_2) \Rightarrow lseg(n(x), y_2)\big) \Rightarrow \big(lseg(y_1, y_2) \Rightarrow lseg(x, y_2)\big)$$

which is valid given $\mathcal{D}^{fp}_*$ since, according to the definition of $lseg$, if $lseg(n(x), y_2)$ holds then $lseg(x, y_2)$ also holds (in the non-degenerate case).

We now define the notion of proving a theorem using lemmas as well as the synthesis problem that it poses which we tackle in this chapter.

Definition 3.3 (Sequential Lemmas that Prove a Theorem)**.** A sequence $(L_1, \ldots, L_n)$ of lemmas provides an inductive proof of α if $\mathcal{A} \cup \mathcal{D}^{fp} \cup \{L_1, \ldots, L_n\} \models_{\mathsf{FO}} \alpha$ and for each $1 \leq i \leq n$, L_i is inductive for $\mathcal{A} \cup \mathcal{D}^{fp} \cup \{L_1, \ldots, L_{i-1}\}$ (i.e., $\mathcal{A} \cup \mathcal{D}^{fp} \cup \{L_1, \ldots, L_{i-1}\} \models_{\mathsf{FO}} PFP(L_i)$).

Definition 3.4 (**Sequential Lemma Synthesis Problem**)**.** Given a grammar G for expressing lemmas and a theorem α, find a sequence of lemmas admitted by G that provides an inductive proof of α (as in Definition 3.3).

Independently Proven Lemmas We can also define a simpler synthesis problem corresponding to a weaker class of inductive proofs. Specifically, we can require a set of lemmas that are *independently* proven inductive and help prove a theorem:

Definition 3.5 (Independent Lemmas that Prove a Theorem)**.** A set $\{L_1, \ldots, L_n\}$ of lemmas provides an inductive proof of α if $\mathcal{A} \cup \mathcal{D}^{fp} \cup \{L_1, \ldots, L_n\} \models_{\mathsf{FO}} \alpha$ and for each $1 \leq i \leq n$, $\mathcal{A} \cup \mathcal{D}^{fp} \models_{\mathsf{FO}} PFP(L_i)$.

The difference between the two classes of proofs is that the inductiveness of lemmas in a sequential proof can depend on previous lemmas. As one might expect, the notion of proof using independent lemmas is strictly weaker than the one that uses a sequence of lemmas. We conclude this section with the running example.

Example 3.6 (Running Example: Lemma Proving a Theorem)**.** Consider L_* and α_* introduced earlier in the running example. Now, observe that $\mathcal{D}^{fp}_* \cup \{L_*\} \models_{\mathsf{FO}} \alpha_*$. This is because the crucial part of the validity of α_* is the following formula:

$$lseg(x, y_1) \Rightarrow \Big((y_1 \neq Nil \wedge y_2 = n(y_1)) \Rightarrow lseg(x, y_2)\Big)$$

which captures the 'else' case of the ite subformula of α_* (see Example 3.4). We can see that L_* entails the above formula in FO since (informally) $y_2 = n(y_1)$ is a special case of $lseg(y_1, y_2)$. Combined with the fact that L_* is inductive (Example 3.5), we have that L_*

proves α_* in the sense of Definition 3.3. We illustrate a run of our synthesis algorithm that proves α_* by synthesizing L_* in Section 3.3.4.

In Section 3.3 we present our core algorithm FOSSIL for solving the *sequential lemma synthesis problem*. This algorithm, apart from being sound in producing sequential lemmas that prove the theorem, is accompanied by a relative completeness result: it is guaranteed to find a proof as long as there is a set of *independent* lemmas that prove the theorem.

3.2.4 Background: First-Order Validity using Systematic Quantifier Instantiation

In this section we describe the Systematic Quantifier Instantiation (SQI) mechanism for FO validity (without recursive definitions/lfp) that we use, developed in earlier work [27].

Let φ be an FO formula. To check the validity of φ, we negate and Skolemize it — introducing both Skolem constants and Skolem functions — and obtain a purely universally quantified formula ψ such that φ is valid if and only if ψ is unsatisfiable. Let ψ be of the form $\forall \overline{x}.\, \eta(\overline{x})$ where $\eta(\overline{x})$ quantifier-free. For a set of ground terms T, we denote by $\psi[T]$ the set of all quantifier-free formulas that are obtained by instantiating the variables $\overline{x}$ in ψ by terms in T, i.e.,

$$\psi[T] := \{\eta(\overline{t}) \mid \overline{t} \textit{ is a tuple of terms in } T \textit{ of arity } \quad \mid \overline{x} \mid\} \tag{3.3}$$

It follows that if $\psi[T]$ is unsatisfiable then ψ is unsatisfiable and therefore α is valid. Since we assume in our setting that satisfiability/validity of quantifier-free formulas is decidable (see Section 3.2.1), checking whether $\psi[T]$ is unsatisfiable is decidable.

Systematic Quantifier Instantiation The above suggests a complete semi-decision procedure for validity based on systematic quantifier instantiation (SQI). Let $\psi \equiv \forall \overline{x}.\, \eta(\overline{x})$ be the formula that we want to check for unsatisfiability where $\overline{x}$ are variables of the foreground sort and η is quantifier-free. For any $k \in \mathbb{N}$, let T_k denote the set of all ground terms whose type is the foreground sort and are of *depth* at most k (we assume that the signature contains at least one constant symbol for the foreground sort). Then, starting with $k = 0$, we check whether $\psi[T_k]$ is unsatisfiable. If it is then we halt and report that φ is valid; otherwise, we increment k and repeat. This motivates the following definition:

Definition 3.6 (Provability at depth k using SQI)**.** A formula φ is provable at depth k using SQI if the negated and Skolemized formula ψ is such that $\psi[T_k]$ is unsatisfiable.

The above is a sound procedure, i.e., if φ is provable at depth k using SQI (for some k) then it is clearly valid. It is also a complete procedure for validity in pure first-order logic without

any theories (i.e., just uninterpreted functions). This follows from Herbrand's theorem and the compactness theorem. It turns out that this continues to be a complete procedure in the multisorted setting for the kind of FOL formulas that we work with. i.e., those that *quantify only over the foreground sort*. We formally state below this result from the work in [27]:

Theorem 3.2 (From [27])**.** Let φ be a formula with quantification only over the foreground sort. Then φ is valid if and only if there exists $k \in \mathbb{N}$ such that φ is provable at depth k using SQI.

We implement and use SQI for proving validity of first-order logic formulae in this work.

3.3 THE FOSSIL ALGORITHM FOR SEQUENTIAL LEMMA SYNTHESIS

In this section, we present the main contribution of this chapter: FOSSIL (First-Order Solver with Synthesis of Inductive Lemmas), our algorithm for solving the Sequential Lemma Synthesis problem formulated in Definition 3.4. Figure 3.1 shows the components of our framework which we describe in Section 3.3.1. FOSSIL is a counterexample-based lemma synthesis algorithm that orchestrates interactions between these external components through three kinds of counterexamples. We formally define these counterexamples in Section 3.3.2. We then present the FOSSIL algorithm in Section 3.3.3. Finally, we illustrate a run of FOSSIL on our running example in Section 3.3.4. (the algorithm is guaranteed to find a proof if there is a set of independent lemmas that prove the goal).

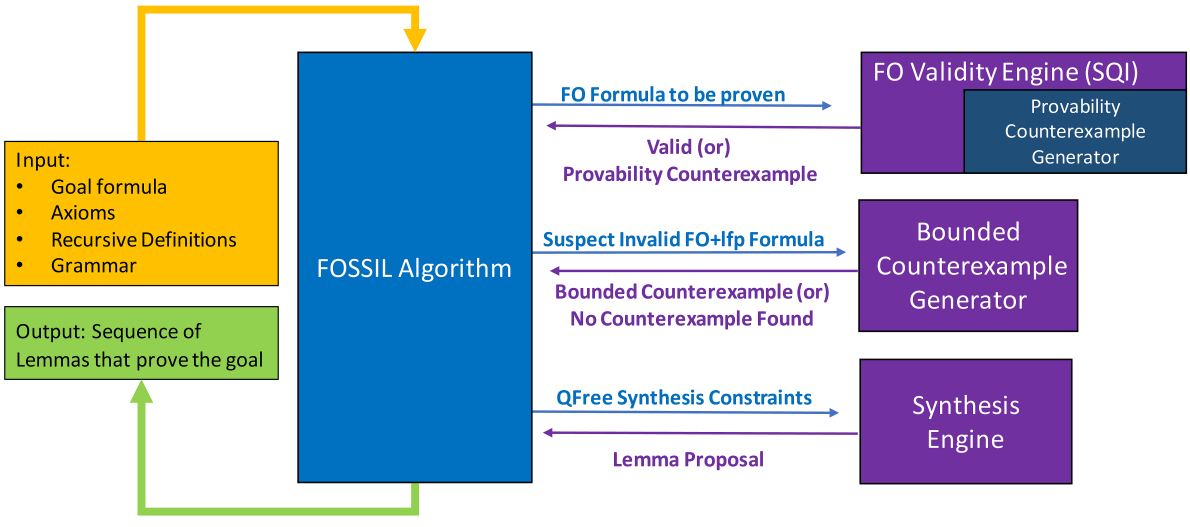


Figure 3.1: Components of FOSSIL

3.3.1 Components of FOSSIL

In this section, we discuss the external components used by the core FOSSIL algorithm. We only describe *what* these components are, deferring implementation details to Section 3.6. Let us fix a set $\mathcal{A}$ of axioms and a set $\mathcal{D}$ of recursive definitions throughout the following presentation. We also fix a *goal* formula α and a grammar $\mathcal{G}$ for lemmas. We assume that $\mathcal{A}$ consists of universally quantified sentences, and that α is also a universally quantified sentence (using Skolemization if necessary).

FOSSIL finds a proof of α by synthesizing a sequence of lemmas $\mathcal{L} = (L_1, L_2, \ldots, L_n)$ belonging to $Lang(\mathcal{G})$ such that $\mathcal{L}$ is a sequence of lemmas proving α according to Definition 3.3. The high-level external components of FOSSIL are shown as purple boxes/arrows in Figure 3.1. We describe their abstract interface below in terms of formulae and counterexamples. We encourage the reader to think of counterexamples as *finite FO models* for now, pending their formalization in Section 3.3.2. The components of FOSSIL are:

1. **First-Order Validity Engine SQI**(φ, k): This is an FO validity checking algorithm based on Systematic Quantifier Instantiation (see Section 3.2.4). It takes as input a formula φ and a natural number k, and outputs whether φ *valid* or *unprovable* at depth k using SQI.

2. **Provability Counterexample Generator** $\mathsf{Counterexample}(\varphi, k)$: This is a counterexample generation module that is part of the FO validity engine. When a formula is found to be unprovable (using term instantiation with terms of depth k) it returns a *finite counterexample model*. This model is one in which $(\neg\varphi)[T_k]$ holds. $(\neg\varphi)[T_k]$ is the negation of the formula instantiated by terms up to depth k. Intuitively, the counterexample witnesses the *non-provability* of φ using term instantiation with depth k terms. Note that finite counterexample models (i.e., where the foreground universe is finite) always exist because $(\neg\varphi)[T_k]$ is a quantifier-free formula. This module is used to generate the *Type−1* and *Type−3* counterexamples (the inputs being the goal or a proposed lemma respectively). The types of counterexamples are explained below in Section 3.3.2.

3. **Bounded Counterexample Generator** $\mathsf{BoundedCex}(\varphi, size)$: Given an FO+*lfp* formula φ and a parameter *size* this module returns a finite model with at most *size* elements in the foreground sort that shows that the formula is *not valid*, if possible. It may also return that such a model could not be found (because one may not exist at that size). These models interpret recursively defined predicates using the true *lfp* semantics and will be used as *Type−2* counterexamples.

4. **Synthesis Engine** $\mathsf{Synthesize}(C, \mathcal{G})$: This module synthesizes candidate lemmas. It takes as input a set of counterexample models expressed as quantifier-free constraints C and

a grammar $\mathcal{G}$, and generates an expression in $Lang(\mathcal{G})$, if one exists, that avoids all counterexamples.

3.3.2 Counterexamples

FOSSIL is a *counterexample-guided* algorithm that uses the verification and synthesis components in rounds of lemma proposals. In this section, we define the notion of the various counterexamples that we use.

While counterexamples can be intuitively thought of as finite models for the foreground universe, we will formally treat them as conjunctive ground formulae as described in Section 3.2.1. For example, consider the model depicting a one-element linked list on the pointer n. The foreground universe has two elements, say v_1 and v_2, such that $n(v_1) = v_2$ and nil is interpreted to be v_2. Then the ground formula $gf \equiv v_1 \neq v_2 \wedge v_2 = Nil \wedge n(v_1) = v_2$ with new constant symbols v_1 and v_2 defines a class of models that contains the intended model. In general, a ground formula captures a *class* of models where a finite portion of the model is constrained by the formula.

In our algorithm we evaluate formulas over tuples of elements on models represented by a ground formula gf. We use the notation $gf(\overline{c})$ to indicate that the model contains interpretations for the constants in $\overline{c}$, and we use this tuple to instantiate the variables of formulas that we evaluate over the model. For example, see lines 13 and 14 of the FOSSIL algorithm (Algorithm 3.1). Similarly, we refer to a set of elements interpreted by a model by C and use it to evaluate a formula on all tuples over C, as in line 12.

The FOSSIL uses three kinds of counterexamples. Let us fix $ctx \equiv \bigwedge(\mathcal{A} \cup \mathcal{D}^{fp} \cup \mathcal{L})$ to be the *context* formula containing the axioms, recursive definition abstractions, and the valid lemmas $\mathcal{L}$ discovered so far.

Type-1 Counterexamples *Type-1* counterexamples guide the synthesis toward lemmas that help prove the goal. Given a term depth k, *Type-1* counterexamples witness non-provability of the goal α using term instantiation with depth k terms. In other words, this is a counterexample to the non-provability of $ctx \Rightarrow \alpha$ using the instantiation.

Formally, a *Type-1* counterexample at depth k is a satisfiable ground formula gf_1 such that $\models_{\mathsf{FO}} gf_1 \Rightarrow (ctx \wedge \neg\alpha[\overline{c}])[T_k]$, where $\overline{c}$ is a tuple of Skolem constants resulting from the Skolemization of the existential quantifiers in the negation of α. Such a model witnesses that α cannot be proven from ctx by instantiation with terms of depth k. We use *Type-1* counterexamples in FOSSIL in lemma synthesis by accessing tuples of elements that correspond to terms in T_k (line 12 in Algorithm 3.1). We name these elements and represent them as a

set C, denoting the counterexample by $gf_1(C)$.

Generating Type−1 counterexamples: Recall from Section 3.2.1 that in our setting, for any satisfiable quantifier-free formula φ, we can obtain a satisfying model as a conjunctive ground formula. When failing to prove $ctx \Rightarrow \alpha$ using depth k term instantiation we obtain a satisfiable conjunctive ground formula from the satisfiability of $(ctx \wedge \neg\alpha[\overline{c}])[T_k]$. This is the *Type−1* counterexample.

Type−2 Counterexamples These counterexamples correspond to finite models (i.e., those in which the foreground sort is finite) that falsify a candidate lemma L in FO+*lfp*. Such a model $\mathcal{M}$ satisfies $\mathcal{M} \models_{\mathsf{LFP}} ctx \wedge \neg L$. When a lemma L is proposed, creating a small model in which L is false can easily show its invalidity.

Formally, a *Type−2* counterexample for a lemma of the form $\forall \overline{x} R(\overline{x}) \rightarrow \psi(\overline{x})$ is represented as a ground formula $gf_2(\overline{c})$ with constants $\overline{c}$ of the foreground sort such that $\mid \overline{c} \mid = \mid \overline{x} \mid$. The formula can include constraints involving relations in $\mathcal{R}^{rec}$. gf_2 interprets the recursively defined predicates with *lfp* semantics. We require that there exists an FO model $\mathcal{M}$ whose interpretation for predicates in $\mathcal{R}^{rec}$ matches their recursive definitions $\mathcal{D}$ (we describe how we implement this requirement in Section 3.4.2). Finally, we require $\models_{\mathsf{FO}} gf_2(\overline{c}) \Rightarrow R(\overline{c}) \wedge \neg\psi(\overline{c})$.

For example, consider the finite model consisting of two locations, say e_1 and e_2, where e_1 is the head of a one-element linked list and e_2 points to itself on the n pointer. This model is captured by the formula $e_1 \neq nil \wedge e_2 \neq nil \wedge next(e_1) = nil \wedge next(e_2) = e_2 \wedge list(nil) \wedge list(e_1) \wedge \neg list(e_2)$. Note that the correct valuation of *list* on this universe is given by the formula.

Generating Type−2 counterexamples: We fix a bound $size \in \mathbb{N}$ and use an SMT solver to identify a model with at most *size* elements in the foreground sort that falsifies the lemma, if one exists. We provide further details in Section 3.4.2 and Section 3.6.

Type−3 Counterexamples *Type−3* counterexamples guide the search towards lemmas that are inductively provable using their PFP. When the PFP of a proposed lemma is found to be unprovable (using depth k term instantiation), we obtain a counterexample that witnesses the non-inductiveness of L (with respect to the lemmas discovered so far). Note that we do not actually know whether the lemma is valid/invalid or provable/unprovable as it may require discovering other lemmas or a bigger instantiation depth. This is similar to a *Type−1* counterexample, where instead of the target theorem we generate counterexamples to the PFP of a candidate lemma.

Formally, a *Type−3* counterexample for a lemma $\forall \overline{x} R(\overline{x}) \rightarrow \psi(\overline{x})$ is a ground formula $gf_3(\overline{c})$ with $\mid \overline{c} \mid = \mid \overline{x} \mid$ such that $\models_{\mathsf{FO}} gf_3(\overline{c}) \Rightarrow (ctx \wedge \neg PFP(L)[\overline{c}])\,[T_k]$ holds and gf_3 is satisfiable.

The constants $\overline{c}$ are Skolem constants obtained from Skolemizing the existential formula $\neg PFP(L)$.

Generating Type−3 counterexamples: Similar to *Type−1* counterexamples, the generation of *Type−3* counterexamples is done using the quantifier-free formula obtained from the proof failure of $ctx \Rightarrow PFP(L)$ using depth k term instantiation.

3.3.3 The FOSSIL Algorithm

Algorithm 3.1 shows the pseudocode of FOSSIL using the external components SQI, Counterexample, BoundedCex, and Synthesize described in Section 3.3.1. The input is a set of axioms $\mathcal{A}$, a set of recursively defined predicates $\mathcal{D}$, a grammar $\mathcal{G}$ whose language potentially contains the lemmas of interest, and the goal α. The algorithm is parameterized over a depth k for term instantiation and a bound h on the height of the expressions to synthesize from $\mathcal{G}$.

The algorithm has an *outer loop* for proving the goal correct on line 5 and an *inner loop* for discovering valid lemmas on line 8. At a general point in the execution on line 5, we try to prove the formula Φ_α (which says that the valid lemmas found imply the goal) using SQI with terms of depth k. If it is valid, we halt and return the sequence of lemmas found.

If Φ_α is unprovable, we obtain a *Type−1* counterexample with a foreground universe C on line 7 and enter the inner loop to discover valid lemmas that will help the proof.

At a general point in the inner loop execution on line 8, we have a *Type−1* counterexample, along with a set of *Type−2* counterexamples and a set of *Type−3* counterexamples. We call the Synthesize module to find a lemma in $\mathcal{G}$ of the form $L(\overline{x}) = \forall \overline{x}.R(\overline{x}) \rightarrow \psi(\overline{x})$ and height bounded by h such that: (a) the lemma is false on the *Type−1* model, i.e., false on some tuple of elements from C (line 12— note that $L[C]$ denotes the set of all instantiations of L by elements from C, and $\bigwedge L[C]$ their conjunction); (b) the lemma holds on every *Type−2* counterexample at the tuple $\overline{c}$ witnessing the invalidity of a previously proposed lemma for R, i.e., with R appearing in the antecedent (line 13); and (c) the *PFP* of the lemma holds on every *Type−3* counterexample at the tuple $\overline{c}$ witnessing the non-inductiveness of a previously proposed lemma for R (line 14).

If no such lemma is found, we halt and restart the algorithm with higher values for k and h. If a lemma L is found, we try to prove Φ_L valid on line 18 using terms of depth k, which says that $PFP(L)$ holds (i.e., L is inductive) given the other valid lemmas discovered. If it is valid, we add L to our assumptions and the current sequence of lemmas, stop the inner loop, and retry the proof of the theorem on line 5. We also discard *Type−3* counterexamples since previously non-provable lemmas may now be provable.

If Φ_L is unprovable, we try to obtain a bounded *Type−2* counterexample $gf_2(\overline{c})$ on line 25

Input: axioms $\mathcal{A}$, recursive definitions $\mathcal{D}$, grammar $\mathcal{G}$, goal α, SQI depth parameter k, lemma height h
Output: Sequence of valid lemmas $\mathcal{L} \in L(\mathcal{G})$ (of height at most h) that prove α valid in FO+*lfp* using $SQI(k)$
Imports: SQI, Counterexample, BoundedCex, Synthesize

1: **procedure FOSSIL**$[\mathcal{A}, \mathcal{D}, \mathcal{G}, \alpha; k, h]$
2: Compute $\mathcal{G}_h \subseteq \mathcal{G}$ such that $Lang(\mathcal{G}_h)$ only contains formulas of height at most h
3: $\mathcal{L} := ()$, $Type\text{-}2 := \emptyset$, and $Type\text{-}3 := \emptyset$ for each $R \in \mathcal{D}$
4: $\Phi_\alpha := \left(\bigwedge \mathcal{A} \cup \mathcal{D}^{fp}\right) \Rightarrow \alpha$
5: **while** $\mathsf{SQI}(\Phi_\alpha, k) \neq VALID$ **do**
6: $gf_1(C) := \mathsf{Counterexample}(\Phi_\alpha, k)$
7: $Type\text{-}1 := gf_1(C)$
8: **while** True **do**
9: $L := \mathsf{Synthesize}(S, \mathcal{G}_h)$
10: *such that*
11: L is of the form $\forall \overline{x}.R(\overline{x}) \rightarrow \psi(\overline{x})$ and satisfies the following constraints:
12: $\models_{\mathsf{FO}} gf_1(C) \Rightarrow \neg(\bigwedge L[C])$, where $gf_1(C)$ is the current $Type\text{-}1$ model
13: $\models_{\mathsf{FO}} gf_2(\overline{c}) \Rightarrow L(\overline{c})$ for all $(gf_2(\overline{c}), R) \in Type\text{-}2$
14: $\models_{\mathsf{FO}} gf_3(\overline{c}) \Rightarrow PFP(L)(\overline{c})$ for all $(gf_3(\overline{c}), R) \in Type\text{-}3$
15:
16: If no lemma found, call FOSSIL$(\mathcal{A}, \mathcal{D}, \mathcal{G}, \alpha; k{+}1, h{+}1)$
17: $\Phi_L := \left(\bigwedge \mathcal{A} \cup \mathcal{D}^{fp} \cup \mathcal{L}\right) \Rightarrow PFP(L)$
18: **if** $\mathsf{SQI}(\Phi_L, k) = VALID$ **then**
19: // Valid Lemma
20: $\mathcal{L} := \mathcal{L} \circ L$ // sequence extension
21: $\Phi_\alpha := \left(\bigwedge \mathcal{A} \cup \mathcal{D}^{fp} \cup \mathcal{L}\right) \Rightarrow \alpha$
22: $Type\text{-}3 := \emptyset$
23: Continue Loop on Line 5
24: **else** // Unprovable Lemma
25: $gf_2(\overline{c}) := \mathsf{BoundedCex}(L, size)$
26: **if** $gf_2(\overline{c})$ found **then** // Invalid Lemma
27: $Type\text{-}2 := Type\text{-}2 \cup \{(gf_2(\overline{c}), R)\}$
28: **else** // Lemma is neither provable nor refutable
29: $gf_3(\overline{c}) := \mathsf{Counterexample}(\Phi_L, k)$
30: $Type\text{-}3 := Type\text{-}3 \cup \{(gf_3(\overline{c}), R)\}$
31: Continue loop on Line 8

Algorithm 3.1: Algorithm for Unfolding Definitions followed by Quantifier-Free Reasoning

such that L does not hold on $\overline{c}$. If we cannot obtain a $Type\text{-}2$ counterexample, then we obtain a $Type\text{-}3$ counterexample $gf_3(\overline{c})$ such that $PFP(L)$ does not hold on $\overline{c}$. We add these counterexamples to their respective sets and continue searching for valid lemmas on line 8.

3.3.4 Running Example: List Segments

In this section, we present a full execution of our algorithm on the running example introduced in Example 3.4. Let us recall the Verification Condition (VC) α_* introduced earlier:

$$lseg(x, y_1) \Rightarrow \Big(\mathsf{ite}(y_1 = Nil, y_2 = y_1, y_2 = n(y_1)) \Rightarrow lseg(x, y_2)\Big) \tag{3.4}$$

We illustrate a run of our algorithm that proves α_* First, it turns out that α_* is not FO-valid and therefore not provable using SQI. It is also not provable by induction using the formula itself as the induction hypothesis, i.e., $\mathcal{D}_*^{fp} \models_{\mathsf{FO}} PFP(\alpha_*)$ does not hold.

Type−1 Counterexample We feed our goal α_* to the SQI module with $k = 1$ from which we obtain a *Type−1* counterexample $\mathcal{M}_1$ (line 7 in Algorithm 3.1):

$$\begin{array}{c} u_1 \mapsto u_2 \mapsto u_3 \mapsto u_3 \\ u_4 \mapsto u_3,\ u_5 \mapsto u_5 \end{array} \quad \textit{and} \quad \begin{array}{c} x = u_1,\ y_1 = u_4,\ y_2 = u_3,\ Nil = u_5 \\ lseg(u_1, u_3) = \mathit{false},\ \text{and } lseg \text{ is } \mathit{true} \text{ otherwise} \end{array} \tag{3.5}$$

where we use $u \mapsto v$ to represent $n(u) = v$ and u_i are elements of the model returned by the solver (one can think of them as new constants). We make some observations here about the interpretation of *lseg* in $\mathcal{M}_1$. The interpretation is not consistent with *lfp* semantics as $lseg(u_1, u_4) = \mathit{true}$ but u_1 never reaches u_4 following the n pointer. In fact, the interpretation is not even consistent with the fixpoint semantics $\mathcal{D}_*^{fp}$ as the definition does not hold for $lseg(u_1, u_3)$. This is because SQI at $k = 1$ only enforces the fixpoint interpretation for *lseg* if the two locations are one step away. Therefore, $\mathcal{M}_1$ merely witnesses the non-provability of α_* using SQI with $k = 1$[3].

Type−2 Counterexample We now search for a lemma using the Synthesize module (line 9), which could propose the lemma $L_1 \equiv \forall x, y.\ lseg(x, Nil) \Rightarrow lseg(y, x)$. L_1 is not true on $\mathcal{M}_1$ and eliminates it as expected, but it is not valid (and is hence found not provable on line 18). We now give it to the BoundedCex module (line 25) which returns the *Type−2* counterexample $\mathcal{M}_2$:

$$v_1 \mapsto v_2 \mapsto v_2 \quad \textit{and} \quad \begin{array}{c} x = v_1,\ y = v_2,\ Nil = v_2 \\ lseg(v_2, v_1) = \mathit{false},\ \text{and } lseg \text{ is } \mathit{true} \text{ otherwise} \end{array} \tag{3.6}$$

$\mathcal{M}_2$ is a model of a one-element linked list where the interpretation of *lseg* is consistent

[3]The reader may wonder whether using SQI at $k = 2$ proves α_*. However, this is also not true as one can construct a model similar to $\mathcal{M}_1$ where u_3 is three steps away from u_1 instead of two. In fact, there exists such a counterexample for any k.

with the *lfp* semantics. We add $\mathcal{M}_2$ to the set of *Type−2* models (line 27) ensuring that future lemmas at least hold true on this simple model and continue our search.

Type−3 Counterexample At some point in the search we obtain the lemma $L_2 \equiv \forall x, y.\ lseg(x, y) \Rightarrow (lseg(y, Nil) \Leftrightarrow lseg(x, Nil))$. L_2 is valid but, as it turns out, $PFP(L_2)$ is not FO-valid (under $\mathcal{D}_*^{fp}$) and therefore L_2 is not provable. The failure of the check on line 18 leads to the generation of a *Type−3* counterexample[4] (line 29) $\mathcal{M}_3$ which is similar in spirit to $\mathcal{M}_1$ as it witnesses the non-provability of $PFP(L_2)$ by SQI. We do not present the model here in the interest of brevity. We add $\mathcal{M}_3$ to our set of countermodels (line 30) to ensure that L_2 is not re-proposed (until we get another valid proposal) and continue lemma search.

Denouement After many such rounds of lemma proposal and counterexample generation, the synthesizer proposes the lemma $L_* \equiv \forall x, y_1, y_2.\ lseg(x, y_1) \Rightarrow (lseg(y_1, y_2) \Rightarrow lseg(x, y_2))$ introduced in our running example (Example 3.5 in Section 3.2.3). We know from Examples 3.5 and 3.6 that L_* is inductive and proves α_*, and in fact it is provable with SQI at $k = 1$. Therefore, the checks on line 18 and subsequently on line 5 both succeed, whereupon FOSSIL terminates and reports that α_* is valid along with the lemma L_* used to prove it.

3.4 SYNTHESIS AND COUNTEREXAMPLE GENERATION ENGINES

In this section, we provide details of the individual modules from Figure 3.1. We refer the reader to Section 3.2.4 for the SQI module and only describe the synthesis and counterexample generation modules below.

3.4.1 Synthesis Engine

The module Synthesize takes a finite grammar for expressing lemmas along with a set of *ground* constraints $\psi(exp)$ over an expression variable *exp*. A finite grammar is one that generates a finite language. It produces a formula φ in the grammar such that ψ is valid when *exp* is replaced with φ.

This problem formulation is similar to SyGuS [53, 61] in that we have a grammar and constraints on the synthesized expression. However, SyGuS specifications are of the form $\forall \overline{x}.\ \psi(exp, \overline{x})$ and can therefore be more complex. In contrast, our constraints have no

[4] Observe here that a *Type−3* counterexample can always be generated for an unprovable lemma, regardless of whether the lemma is truly invalid or not.

variables or quantification and are *grounded*. We can of course use SyGuS solvers as synthesis engines, and indeed we do so in a version of our implementation of FOSSIL (see Section 3.6.1).

We now describe our custom synthesis engine tailored for ground constraints. First, since our lemmas are all purely universally quantified over the foreground sort we make the quantifiers implicit and only synthesize quantifier-free expressions. Second, we reduce the synthesis to a quantifier-free query over a combination of theories that can be effectively handled by modern SMT solvers [11, 29]. Since derivations from the grammar are of finite height, it is easy to see that we can encode any expression in the language using a finite set of boolean variables representing choices of production rules for each nonterminal in a derivation. Encodings like these are typical in constraint-based synthesis. Combined with the fact that the constraints are grounded, synthesis reduces to a quantifier-free SMT query that asks for an assignment to the boolean variables representing a candidate lemma that satisfies the constraints.

Grounded Constraints and Using Boolean Constraint Solvers One important optimization that we did in the synthesis engine is to solve it using (essentially) Boolean constraints. Counterexamples in our setting are finite models that can be captured using *grounded formulas* as described in the previous section. Given a grammar, we first bound the depth of the grammar (this bound is incremented in an outer loop) and we model the choices of which production rules are applied using a set of Boolean variables $\overline{b}$. Consequently, each valuation of $\overline{b}$ stands for a formula $\psi[\overline{b}]$. For conforming to a counterexample ce, we need to write a formula $Eval_{ce}(\overline{b})$ that checks whether the formula $\psi[\overline{b}]$, the formula encoded by $\overline{b}$, holds on the model ce for a particular instantiation of the free variables in ψ. (The actual lemma universally quantifies over variables and asserts ψ.)

The straightforward encoding of this problem will essentially evaluate the parse tree of the formula, examining the appropriate Boolean variables in $\overline{b}$ to interpret subformulas or subterms at each node of the parse tree, introducing variables of appropriate sort for subterms. This introduction of variables causes the problem to be an SMT query. However, if we restrict to grammars where all nonterminals generate only formulas (no terms), then it turns out that we can encode the problem without additional variables.

Grammars can be made to have nonterminals generate only formulas by enumerating terms in the derivation rules of atomic formulas. Furthermore, evaluation of atomic formulas over models can be effected using just ground formulae, for a particular instantiation of the free variables over a model, which can be modeled using Skolem constants.

The above yields constraints over $\overline{b}$ that are grounded, which is essentially Boolean satisfiability. We implement the optimization and find it extremely effective on our benchmarks.

We implement the above technique in a custom synthesis engine (see Section 3.6.1) and evaluate its efficacy in Section 3.6.

3.4.2 Counterexample Generators

FOSSIL uses three kinds of finite counterexample models to guide lemma synthesis. The *Type−1* model witnesses non-provability of the goal given the current set of synthesized lemmas and makes the synthesis goal-directed. *Type−2* models witness the invalidity of lemmas proposed and guide synthesis towards producing valid lemmas. Finally, the *Type−3* models witness non-inductiveness of lemmas proposed and guide synthesis towards producing provable lemmas.

The *Type−1* and *Type−3* counterexamples are generated using the Counterexample module as shown on lines 7 and 29 in Algorithm 3.1. These are obtained as a by-product of using the SQI module for verification since it reduces the validity of a quantified formula φ to the satisfiability of a *quantifier-free formula* ψ (see Section 3.2.4).

The generation of *Type−2* models is more involved. We realize the BoundedCex module which generates them using an SMT solver. Given a bound *size* on the size of the model, we construct a formula that represents the existence of *size*-many elements $u_1, u_2 \ldots, u_{size}$ such that the valuation of functions (including recursively defined predicates) satisfies the axioms and falsifies the given lemma. The key aspect of our construction is the notion of the *rank* of $(R, \overline{u})$ for every $R \in \mathcal{R}^{rec}$ and argument $\overline{u}$ in the domain of R. The rank of $(R, \overline{u})$ is an integer in the range $[-1, \infty)$ which we constrain to ensure that the valuation of recursively defined predicates on a *Type−2* model is consistent with their definitions interpreted using *lfp* semantics.

Let us consider the simple case where we only have one recursively defined predicate R which is unary and has the definition $R(x) :=_{lfp} \rho(x, R)$. Since there is only one recursively defined predicate, we drop R from the notation for simplicity and simply refer to the rank of u instead of the rank of (R, u). Assume that the definition $\rho(x, R)$ refers to R over a particular set of terms—say $R(t_1(x)), R(t_2(x)), \ldots R(t_m(x))$. The rank of u is an integer variable $Rank_u$ whose value is in the range in the range $[-1, \infty)$. We then enforce the following constraints: (a) R holds on u iff the rank of u is not -1, (b) if the base case of the definition holds then the rank is 0, i.e., iff $\rho(u, \bot)$ holds then the rank of u is 0, (c) if the rank of u is positive, then the witnessing atomic formulae $R(t_i(u))$ that make $\rho(u, R)$ true are such that each t_i gets a smaller non-negative rank than the rank of u, and (d) if the rank of u is -1, then in any set of witnessing atomic formulae $R(t_i(u))$ we pick such that their truth would make $\rho(u, R)$ true, there is at least one t_i whose rank is -1.

Intuitively, the rank of $(R, \overline{u})$ mimics the iteration order of the usual iterative least fixpoint computation of R at which the tuple $\overline{u}$ is "added" to R. It is easy to see that if we assign ranks this way, i.e., assigning the rank of u to be the iteration number at which it is added to R (and -1 if it is never added), then the ranks will satisfy the above constraints. Furthermore, if an assignment of ranks satisfying the constraints exists, then we are assured that R evaluates to the true least fixpoint. Finally, since we only want a bounded model the above constraints can be expressed as a quantifier-free SMT query. We use this technique to produce true counterexamples to lemmas.

Computing Least-Fixpoints versus Using Under-Approximations The reader may wonder whether it is possible to use under-approximations of the least-fixpoint instead of computing the precise *lfp* valuations for *Type−2* counterexamples. After all, if a predicate R holds in an under-approximation, then it certainly holds in the least-fixpoint semantics. However, under-approximations will not work because of the presence of negation in two ways. First, our lemmas and theorems can mention recursively defined functions/predicates in negated form. In this case, computing an under-approximation of the lfp will not be correct. For example, consider a lemma $\forall x.R(x) \Rightarrow S(x)$ for recursively defined predicates R and S. Negating this lemma would require a model of $R(x) \wedge \neg S(x)$. An under-approximate computation of S will not work in this case as we may obtain models that do not satisfy this negated formula. Second, negations are also needed in recursive definitions. Our general theoretical treatment allows negations in layers. Such definitions do occur in our experiments. For example, the definition of a binary tree (see Example 3.3) recursively requires the root not to be present in the heaplets of subtrees rooted at the left and right children of the root. This involves negation of the heaplet function *htree* which is recursively defined.

3.5 SOUNDNESS AND RELATIVE COMPLETENESS

The soundness of FOSSIL is clear from the problem description and the termination conditions in Algorithm 3.1: the branch on line 19 is only taken when a lemma is proved valid, and the loop condition on line 5 establishes that if FOSSIL terminates, it does so with a sequence of lemmas that prove α. We can now ask whether the algorithm will always find a sequence of lemmas in $\mathcal{G}$ that prove α if one exists. It turns out that FOSSIL is not complete for the problem of sequential lemma synthesis. However, FOSSIL is complete with respect to *independent lemmas* (see Definition 3.5). That is, if there is a set of independent lemmas that prove α, then it is guaranteed that FOSSIL will find a sequential proof of α.

Theorem 3.3 (Relative completeness of FOSSIL with respect to independent lemmas)**.** If α is provable from $\mathcal{A}$ and $\mathcal{D}$ by a finite set of independent inductive lemmas in $\mathcal{G}$ in the sense of Definition 3.5, then there is an instantiation depth k and a grammar height h such that FOSSIL terminates and returns a sequence $\mathcal{L}$ of lemmas that proves α.

Proof. Assume that there exists some set of independent lemmas $\{L_1, L_2, \ldots, L_n\}$ that proves α. Let us fix k and h to be such that every L_i as well as the goal (given the lemmas) is provable with a depth k instantiation, and the maximum height of any of the productions in $\mathcal{G}$ that yield a lemma L_i is h. We claim that FOSSIL with parameters k and h will terminate having found a sequence of lemmas that prove α.

We induct on the number n of lemmas in the set. Since the algorithm is sound, if it terminates there is clearly a sequence of lemmas that proves α. We establish that either the algorithm will terminate with a proof of the goal, or at least one $L_i, 1 \leq i \leq n$ will be eventually (at some finite time) chosen by the synthesis module, i.e., it cannot be that the algorithm restarts FOSSIL with new parameters in line 16 or runs forever without choosing one of the lemmas L_i. If some L_i is chosen by the synthesis module, since we know by our choice of k that L_i is provable with depth k instantiation, it will be added to Φ_α (see line 20) before all the variables are reset, which reduces the problem to discovering at most $n-1$ independent lemmas whereupon we will appeal to the inductive hypothesis.

It is clear from the definition of $\mathcal{G}_h$ that $Lang(\mathcal{G}_h)$ is finite for any h. Observe from the description of the algorithm in Section 3.3.3 that in each round the candidate proposal L will either: (i) be prevented from being proposed again in the inner loop (line 8) by the addition of a *Type−3* model, or (ii) be prevented from being proposed again permanently during the execution of FOSSIL (with parameters k and h) because it was proved valid and added to Φ_α or it was proved invalid using a *Type−2* model. This eliminates the possibility that the algorithm keeps on proposing lemmas that are not provable. It either finds a provably valid lemma, or it has no further candidate lemmas to propose, and thus would restart the algorithm with new parameters in line 16.

If it finds a valid lemma, the search space for the next round of synthesis is smaller (because the discovered valid lemma will not be proposed). This can happen only finitely often.

This leaves us with the possibility that the algorithm reaches line 16 without finding a new candidate lemma. In particular, this means that none of the L_i satisfies the constraints in line 8. We show that this cannot be the case, i.e., that at least one $L_i, 1 \leq i \leq n$ satisfies the constraints (and is therefore a viable proposal for the synthesis module).

It is easy to see that each $L_i, 1 \leq i \leq n$ satisfies constraints 13 and 14 since the former constraint is satisfied by any lemma valid in the FO-lfp theory defined by $\mathcal{A}$ and $\mathcal{D}$, and

the latter is satisfied by any lemma that is provable by induction at depth k. Both of these conditions are true of every L_i. This leaves us with constraint 12. Assume for the sake of contradiction that no lemma satisfies the constraint, i.e., there is a model M (namely the current *Type−1* model) such that $M \models (\mathcal{A} \cup \mathcal{D} \cup \{\neg\alpha\} \cup \{L_i\})[T_k]$ for any $L_i, 1 \leq i \leq n$. This yields that $M \models (\mathcal{A} \cup \mathcal{D} \cup \{\neg\alpha\} \cup \{L_i \mid 1 \leq i \leq n\})[T_k]$, which contradicts our initial assumption that $\{L_1, \ldots, L_n\}$ collectively prove α at depth k, i.e., $(\mathcal{A} \cup \mathcal{D} \cup \{\neg\alpha\} \cup \{L_i \mid 1 \leq i \leq n\})[T_k]$ is unsatisfiable. Therefore some L_i satisfies the constraint on line 12 and will eventually be proposed, which concludes our proof. QED.

Completeness for Sequential Lemma Synthesis We provide an example to show that FOSSIL is not complete for sequential lemma synthesis. They key obstacle is the *Type−1* model that makes the lemma synthesis goal-directed.

Example 3.7. Consider the case where α can be proved using a sequence (L_1, L_2) of two lemmas. Let L_1 be provable on its own, L_2 be provable assuming L_1, and α is provable assuming L_2. At line 7 in Algorithm 3.1, L_2 would be false on *Type−1* since it helps prove α. But there is nothing that prevents $L_1[T_k]$ from being true on *Type−1*, so let us suppose that it is true. If that is the case, then L_2 might be selected by the algorithm and then quickly dismissed since it cannot be proved valid without L_1. We would then add a counterexample for it on line 30 witnessing that L_2 has no inductive proof. However, the *Type−1* model has not changed (we only recompute it when we find a valid lemma) and therefore L_1 will never be proposed as well. We cannot therefore guarantee that FOSSIL will find a proof of α.

There are several possibilities for extending FOSSIL to achieve completeness for sequential lemma synthesis. We discuss some potential directions in the Technical Report for the publication associated with this chapter [62].

3.6 IMPLEMENTATION AND EVALUATION

In this section, we describe our implementation and evaluation of FOSSIL. We also compare with lemma synthesis tools over ADTs and Separation Logic. The version of the code and data artifacts used in our experiments is available via ACM DL [63].

3.6.1 Implementation

We implement FOSSIL in Python, building the components given in Figure 3.1 using Z3Py (an API for the SMT solver Z3 [11]) to handle the various SMT queries for verification

and generation of counterexamples. Our implementation covers the various external modules as well as the main FOSSIL algorithm.

The first component is an implementation of the SQI module (see Section 3.2.4). As far as we know, this is the first implementation of systematic quantifier instantiation [27, 50, 56] that realizes a complete FO validity engine for quantified formulae using SMT. The second component is an extension of the SQI engine that provides provability counterexamples (used for *Type−1* and *Type−3* models). The third component is the bounded counterexample generator which we implement using the technique described in Section 3.4.2.

The fourth component is an implementation of a custom synthesis engine (a SyGuS solver) that uses constraint solvers (SMT) to synthesize expressions from a grammar given ground constraints. We implement this based on the technique described in Section 3.4.1. As we show in our experiments, reductions to off-the-shelf synthesis engines did not work well. Our synthesis engine exploits the fact that constraints are grounded and carefully generates constraints so that synthesis can be done using SMT solvers. These optimizations were crucial to ensuring efficiency of the synthesis engine. The synthesis engine explores the space of terms and the space of formulae independently, prioritizing exploring the space of formulae. It only explores terms of depth 0 or 1 as we found this sufficient to solve all our benchmarks.

Finally, we implement the core FOSSIL algorithm utilizing the components above.

3.6.2 Research Questions

Our evaluation aims to answer the following Research Questions (RQs).

RQ1: How effective is FOSSIL in synthesizing inductive lemmas to prove theorems?
RQ2: How effective are countermodels in FOSSIL?
RQ3: How effective is our constraint-based synthesis approach in FOSSIL?

3.6.3 Benchmarks

We curate two classes of benchmarks. The first suite consists of 50 theorems that were distilled from the work on VCDryad [56] repository[5] which verifies heap manipulating programs. VCDryad converts DRYAD, a variant of separation logic, to FO+*lfp*. From about 450 VCs (Verification Conditions), we eliminated those that were provable using pure FO reasoning, those that were provable by induction (using the theorem itself as the induction hypothesis), or those that could be proved using frame reasoning [60]. The goal was to retain

[5]The repository can be found at https://madhu.cs.illinois.edu/vcdryad/examples/.

only those VCs that required lemma synthesis. From these, we distilled a set of theorems (removing trivial reformulations) and added them to our suite. We also formulate several theorems that capture static properties of data structures. Six more theorems were obtained by modeling partial correctness of scalar programs with loops. We capture the computation of the program as a linked list of configurations and use *lfp* to determine *reachable* states, demanding that unsafe states are not reached. Table 3.1 shows the list of theorems that we include in our suite. For example, 'bst-leftmost' requires proving that the leftmost node in a binary search tree has the smallest key in the entire tree. We also include theorems about linked lists, sorted linked lists, list segments, dags, binary search trees, maxheaps, etc. The benchmarks obtained from scalar programs are labeled by the prefix 'reachability'.

The second suite of benchmarks consists of 673 synthetic theorems that are automatically generated using fixed recipes. The data structure is a dag/tree with a *key* field and data fields $d_1, \ldots, d_n$, all of type integer. The theorem requires proving that a predicate P holds on the d_1 field of the root of the tree. The predicates chosen were inspired by induction exercises for undergraduate students in discrete math courses. The inductive lemma requires stating the properties of several data fields. The data structures also satisfy other properties based on structure as well as the *key* field (dag, tree, binary search tree, max heap, and trees with parent pointers). The suite was obtained from combinations of predicates, the number of data fields, and properties of data structures.

Lemma Grammars Since our lemmas are of the form $L \equiv \forall \overline{x}.R(\overline{x}) \rightarrow \psi(\overline{x})$ we make the universal quantifiers implicit, and the grammars only restrict the quantifier-free formula ψ. For the first suite, we systematically generate grammars based only on the syntax of the recursive definitions and the theorem. All variables $\overline{x}$ and all foreground constants from the theorem are added to the grammar. All constants in the definitions and the theorem are also added. We allow all terms over these variables and constants. For atomic formulae, we add all relations (including recursively defined relations) over the foreground sort. If integers appear in the theorem, we add equality and inequality for integer terms. If sets appear in the theorem, we add membership and other set operators. The only Boolean connective allowed is implication. We stratify the grammars by the complexity of formulae (primarily split according to the inclusion or exclusion of set operations) to allow for efficient exploration.

For the second suite, we design the grammar automatically. We add the variables and constants of the foreground sort from the theorem. We add 0 and the integer terms built from *key* and the other data fields. The atomic formulae included are the data structure relation, equalities/disequalities between foreground sort terms, and the fixed predicate P per benchmark. Finally, we include implication and conjunction as Boolean operators.

Table 3.1: Experiment results of the FOSSIL tool. #P is the number of lemmas synthesized; #V is the number of valid lemmas synthesized; T (s) is the runtime in seconds.

Theorem	#P	#V	T (s)
dlist-list	1	1	1
slist-list	2	1	1
sdlist-dlist	2	1	2
sdlist-dlist-slist	4	2	3
listlen-list	1	1	0
even-list	3	1	1
odd-list	5	2	3
list-even-or-odd	11	4	124
lseg-list	7	1	5
lseg-next	6	1	6
lseg-next-dyn	1	1	1
lseg-trans	5	1	5
lseg-trans2	7	1	7
lseg-ext	12	1	12
lseg-nil-list	6	1	4
slseg-nil-slist	5	1	4
list-hlist-list	6	1	2
list-hlist-lseg	4	1	2
list-lseg-keys	7	1	4
list-lseg-keys2	7	1	4
rlist-list	2	1	2
rlist-black-height	21	7	125
rlist-red-height	20	7	124
cyclic-next	20	2	126
tree-dag	3	1	3

Theorem	#P	#V	T (s)
bst-tree	2	1	4
maxheap-dag	2	1	3
maxheap-tree	2	1	3
tree-p-tree	2	1	3
tree-p-reach	14	2	17
tree-p-reach-tree	12	3	18
tree-reach	9	2	25
tree-reach2	4	1	7
dag-reach	5	1	20
dag-reach2	6	1	4
reach-left-right	12	3	40
bst-left	10	1	57
bst-right	8	1	104
bst-leftmost	39	10	167
bst-left-right	27	6	104
bst-maximal	5	1	5
bst-minimal	7	1	7
maxheap-htree-key	29	3	155
maxheap-keys	9	2	140
reachability	4	1	4
reachability2	2	1	2
reachability3	3	1	3
reachability4	2	1	2
reachability5	4	1	4
reachability6	4	1	3

3.6.4 RQ1: Effectiveness of FOSSIL in Proving Theorems

We study the effectiveness of our tool in solving both benchmark suites.

Benchmark Suite #1 Table 3.1 gives the names of the 50 theorems in Suite #1, along with the total time taken by our tool to prove each theorem. We find that our tool solves all benchmarks within 5 minutes per benchmark, splitting time between the grammar strata. Guided by early empirical results, we put in an optimization of the general description of FOSSIL in our tools by incrementing h but not k when we exhaust the given grammar (line 16 in Algorithm 3.1). The table also reports the total number of lemmas synthesized and the number of lemmas among those that were proved valid.

FOSSIL is effective on these benchmarks. The average time per theorem was 30s (with a maximum of 167s). The total number of lemmas proposed varied from 1 (i.e., the first

proposed lemma was sufficient) to 39, with up to 10 valid lemmas discovered when solving some benchmarks. Most benchmarks were solved with formula depth $h = 3$ and term instantiation depth $k = 1$. For 14 benchmarks, the tool reached $h = 4$ and $k = 1$.

The tool finds interesting lemmas such as those characterizing properties of data structures, and relating different structures (like lists and list segments), relating different constraints on data structures, etc. We refer the reader to the Technical Report for the publication associated with this chapter [62] for a sample of valid lemmas discovered in proving each theorem. For example, for `bst-left-right`, the tool proposes 27 lemmas of which 6 were proved valid, including complex lemmas such as

$$bst(x) \Rightarrow (y \in hbst(x) \Rightarrow minr(x) \leq minr(y)) \tag{3.7}$$

Here, $bst(x)$ means x is the root of a binary search tree, and $minr(x)$ denotes the minimum key in the subtree rooted at x; both are recursively defined. The lemma states that for every node y in a bst, the minimum key in the subtree of y is less than or equal to the minimum key of the whole tree. While intuitively true for any bst, formal proof of this property requires induction. Note that the synthesis engine itself has no context about the datastructure and does not employ any inductive reasoning! It is simply an expression generator that uses the rich counterexamples we provide to *guess* a formula that satisfies the various constraints.

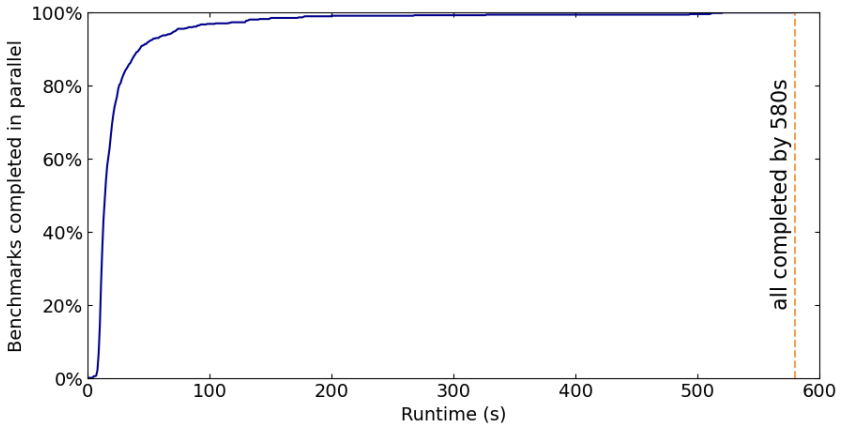
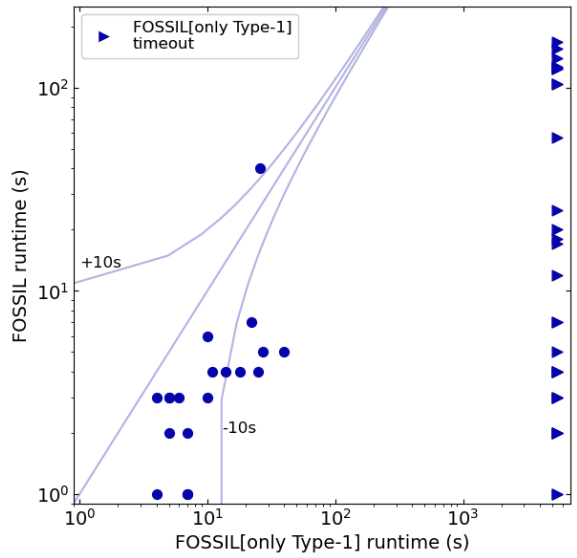
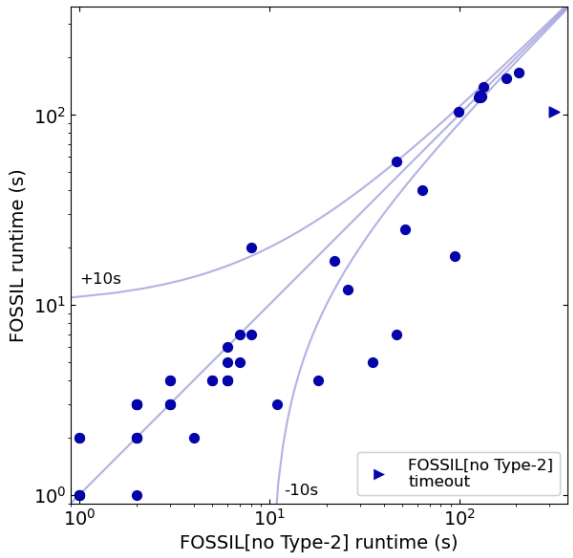


Figure 3.2: Cumulative sum graph of FOSSIL on the synthetic benchmark suite of 673 theorems.

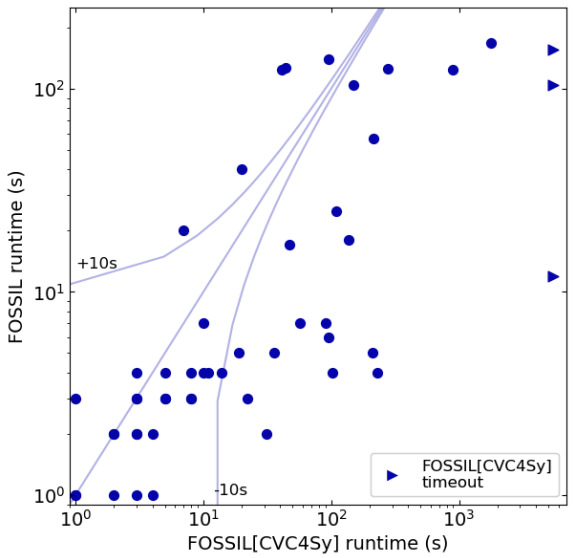
Benchmark Suite #2 Figure 3.2 contains a cumulative sum graph depicting the time taken by our tool on the synthetic benchmarks. Our tool performs well, proving all 673 theorems within the timeout of 10 minutes. 628 of the benchmarks, approximately 93%, were solved within one minute.



(a) Runtime comparison of FOSSIL vs. FOSSIL with no *Type−3* or *Type−2* counterexamples.



(b) Runtime comparison of FOSSIL vs. FOSSIL with no *Type−2* counterexamples.



(c) Runtime comparison of FOSSIL vs. FOSSIL using CVC4Sy.

Figure 3.3: Ablation studies of the FOSSIL tool. The timeout is 1 hour. In 3.3a, 3.3b, and 3.3c, the diagonal lines represent equal running time for both axes. Points on the super-diagonal curves signify FOSSIL is 10 seconds slower than its ablated counterpart, while points on the sub-diagonal curves signify FOSSIL is 10 seconds faster.

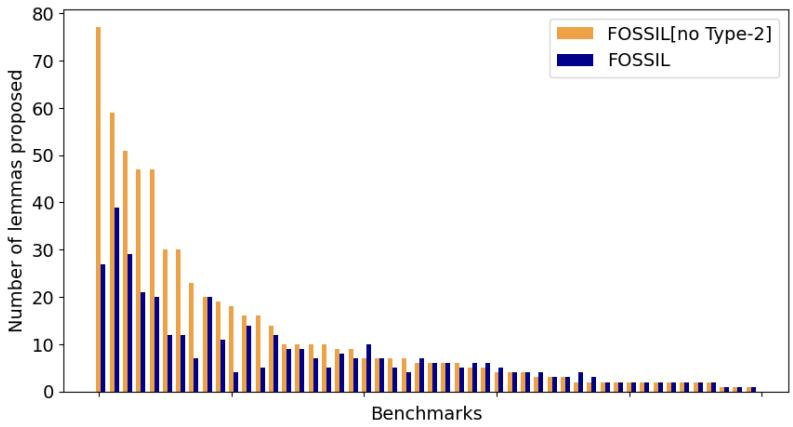


Figure 3.4: Comparison of lemma proposal counts by FOSSIL vs. FOSSIL without *Type−2* counterexamples.

3.6.5 RQ2: Comparison to Synthesis without Use of Counterexamples

We test the efficacy of counterexamples by removing each kind during synthesis. We do not ablate *Type−1* counterexamples since proposed lemmas would be unrelated to the theorem and a comparison is not meaningful. We perform ablation studies removing both *Type−2* and *Type−3* counterexamples or only removing *Type−2* counterexamples.

Efficacy of *Type−2* and *Type−3* counterexamples: It is not possible to directly run our synthesis engine without *Type−2* and *Type−3* counterexamples as the same invalid lemmas can be continuously re-proposed. We hence modify our algorithm to perform the ablation study. The algorithm differs from FOSSIL (Algorithm 3.1) in two ways. First, the Synthesize module can *skip* solutions, proceeding to others. Second, when a lemma is not provable (line 24 in Algorithm 3.1) we simply discard the lemma by asking the synthesis engine to skip to the next solution. We do this until a valid lemma is found, at which point we move to the outer loop (line 5) and attempt to prove the goal again. Of course, in this algorithm, we also do not maintain sets of *Type−2* or *Type−3* counterexamples and only use the *Type−1* counterexample in the synthesis query.

In our implementation, we integrate a version of FOSSIL with the state-of-the-art SyGuS solver in CVC4 (CVC4Sy), providing only *Type−1* counterexamples during synthesis. We used the efficient *streaming* mode of CVC4Sy that can skip solutions. This mode generates a stream of solutions to a synthesis query without repetition, and we simply skip along this stream when we reject candidate lemmas. CVC4Sy is well-optimized, performing symmetry and semantic reductions [64]. We used a timeout of 1 hour for the ablated algorithm.

Figure 3.3a compares the ablated tool against our tool (with all types of counterexamples) on Suite #1 benchmarks. Apart from a few outliers where the lemmas proposed are simple, FOSSIL with only *Type−1* counterexamples performs drastically worse than FOSSIL with all three counterexamples. 31/50 benchmarks timed out with the ablated algorithm. This shows the efficacy of *Type−2* and *Type−3* counterexamples in guiding search.

We also perform this experiment with the synthetic benchmarks (Suite #2). FOSSIL using only *Type−1* counterexamples surprisingly solves only *1 out of the 673 benchmarks* within 10 minutes. This again demonstrates the efficacy of *Type−2* and *Type−3* counterexamples.

Efficacy of *Type−2* counterexamples We evaluate the efficacy of *Type−2* countermodels in FOSSIL by building a version of FOSSIL that does not use *Type−2* counterexamples.

The ablated algorithm is similar to the one in Algorithm 3.1 except for the case where a lemma is not provable (line 24). If a lemma cannot be proven valid, we do not try to generate a *Type−2* counterexample (lines 25- 27) and skip directly to generating a *Type−3* counterexample (line 29). A *Type−3* counterexample can always be generated since it witnesses the non-provability of a lemma (see Section 3.3.2). It also ensures that such unprovable lemmas will not be re-proposed.

Figure 3.3b shows the running time comparison between the FOSSIL tool and the FOSSIL tool without *Type−2* counterexamples. The ablated tool does not solve one of the benchmarks and is slower in general for many benchmarks, especially those that require more than 10 seconds to solve. *Type−2* countermodels seem to have a higher impact in pruning the search space for more complex theorems. Figure 3.4 shows a comparison in the number of proposed lemmas for FOSSIL vs. FOSSIL without *Type−2* counterexample models. Fewer lemmas are proposed for most benchmarks in the FOSSIL tool, showing the efficacy of the guidance of *Type−2* counterexamples.

3.6.6 RQ3: Comparison with CVC4 SyGuS Solver

To evaluate the efficacy of our custom synthesis tool that learns from first-order models with grounded constraint solving, we compare our synthesis tool with CVC4Sy (in standard mode with *all* counterexamples), utilizing the synthesis engines in an identical fashion to the FOSSIL tool. We use a timeout of 1 hour for the ablated algorithm. Figure 3.3c shows the results of this evaluation and indicates that as theorems become more complex, FOSSIL with our custom constraint-based synthesis solver solidly outperforms FOSSIL with CVC4Sy as the synthesis solver. Thus, exploiting the form of synthesis in this domain that has ground constraints is useful.

3.6.7 Comparison with ADT/Separation Logic Tools

The idea of discovering inductive hypotheses to prove theorems is a problem that has been studied in many logical contexts. We are not aware of any tools that synthesize inductive lemmas for FO+*lfp*, especially ones that can handle foreground and background sorts as in our setting.

Comparing tools that work for different logics (FO+*lfp*, algebraic datatypes, separation logic) is inherently hard and poses several challenges: the logics being different, the hardness of translating theorems between them, translation bloat, translations that make theorems harder and required lemmas more complex, tools supporting only restricted fragments, and so on. These make fair comparisons hard.

In this section however, we attempt to compare our tool with tools for algebraic datatypes (ADTs) and separation logic on our benchmarks, making the best translation effort. Though our tool performs much better than these tools on our benchmarks, this should not be construed as evidence that the other tools are inferior in their native settings. Yet, as the comparison below will show, solving theorems in FO+*lfp* effectively by reducing them to tools for other logics does not seem possible. We also believe that incorporating our ideas into lemma synthesis tools for these logics natively is an interesting future direction.

Comparison with Tools for Algebraic Datatypes Theoretically, the logic FO+*lfp* and FO logic over algebraic datatypes are very different. In pure ADT logics, the universe is a *single* universe while FO+*lfp* admits a multitude of universes. Furthermore, our benchmarks are motivated by reasoning over pointer-based heaps that embed data structures, which are different from pure mathematical algebraic datatypes (heaps admit a spaghetti of pointers that embed overlapping data structures). Consequently, we find it impossible to encode our benchmarks in a pure ADT logic.

However, when a first-order logic over ADTs includes *uninterpreted functions* (or higher-order functions), we can find reasonable encodings. We can model locations using elements of some ADT (say 0 with *succ*) or even a background theory of integers if supported. We can model pointers using uninterpreted functions from locations to locations. Least fixpoint definitions can be modeled in several ways. We choose one that does not involve specific background sorts (such as true natural numbers) and instead uses the structure of ADTs.

We encode finite pointer-linked data structures such as linked lists and linked trees using ADTs such as lists and trees, respectively, that store *locations* constituting the linked data structure. Now, recursive definitions on ADTs can capture whether a list/tree of locations corresponds to a linked list/tree by checking, recursively, that the relevant pointers (*next*,

or *left*/*right*) relate the locations stored in the ADT correctly. Using a mild generalization of this technique, we can encode recursively defined data structures of all kinds used in our benchmarks (including list segments, cyclic lists, doubly linked lists, binary search trees, etc.) in a fairly natural way.

We encoded all 50 benchmarks from Suite #1 into CVC4+ig [65] and ADTInd [66], both of which use induction and lemma synthesis. CVC4+ig solved 1/50 benchmarks, and ADTInd solved 8/50 benchmarks within 15 minutes. This demonstrates that our tool performs significantly better on our benchmarks than reductions to these tools do.

Comparison with Separation Logic Tools We consider tools in the Separation Logic Competition (SL-COMP) [67] (note that these tools do not have grammars for lemmas).

There are many restrictions imposed by the various divisions and tools that make encoding our benchmarks challenging. None of the tools for the closest division **qf_shid_entl** support *conjunction of heap formulas* that we require to encode our benchmarks. Also, some of our benchmarks mention heaplets explicitly and thus are hard to encode.

We consider the solver SLS (Songbird+Lemma Synthesis) [68] that won the 2019 SL-COMP competition for the **qf_shid_entl** division. SLS has support for synthesizing inductive lemmas. As mentioned above, many of our examples cannot be translated faithfully into SLS. We were able to encode and prove valid 14 of the 50 examples from Suite #1. There were several examples that we could encode but which SLS was unable to prove (at least 8 such: `cyclic-next`, `list-even-or-odd`, and the 6 program reachability examples).

Chapter 4: Predictable Verification using Intrinsic Definitions

In previous chapters we investigated the limitations of frameworks for verifying complex specifications that use recursively defined functions and predicates. In particular, we tried to automate away the creative task of inductive lemma synthesis in Chapter 3. We return again to this setting of verifying heap manipulating programs over recursively defined heap datastructures. In this chapter we propose a novel mechanism of defining data structures using *intrinsic definitions* that avoids recursion and instead utilizes *monadic maps satisfying local conditions*. We show that intrinsic definitions are a powerful mechanism that can capture a variety of data structures naturally. We show that they also enable a predictable verification methodology that allows engineers to write ghost code to update monadic maps and perform verification using reduction to decidable logics. We evaluate our methodology using BOOGIE and prove a suite of data structure manipulating programs correct.

4.1 INTRODUCTION

In computer science in general, and program verification in particular, classes of finite structures (such as data structures) are commonly defined using *recursive definitions (aka inductive definitions)*. Proving that a set of structures is in such a class or proving that structures in the class have a property is naturally performed using *induction*, typically mirroring the recursive structure in its definition. For example, trees in pointer-based heaps can be defined using the following recursive definition in first-order logic (FOL) with least fixpoint semantics for definitions:

$$\begin{aligned} tree(x) ::=_{lfp}\ & x = Nil \vee \big(x \neq Nil \wedge tree(l(x)) \wedge tree(r(x)) \\ & \wedge x \notin htree(l(x)) \wedge x \notin htree(r(x)) \wedge htree(l(x)) \cap htree(r(x)) = \emptyset\big) \quad (4.1) \\ htree(x) ::=_{lfp}\ & ite\,(x = Nil,\ \emptyset,\ htree\,(l(x)) \cup htree\,(r(x)) \cup \{x\}) \end{aligned}$$

In the above, *htree* maps each location x in the heap to the set of all locations reachable from x using l and r pointers, and the definition of *tree* uses this to ensure that the left and right trees are disjoint from each other and the root. Definitions in separation logic are similar (with heaplets being implicitly defined, and disjointness expressed using the separating conjunction '$\star$' [60, 69, 70]).

When performing imperative program verification, we annotate programs with loop invari-

The material in this chapter is entirely reproduced from the publication cited as [18] co-authored by the author of this thesis, with minor changes.

ants and contracts for methods, and reduce verification to validation of Hoare triples of the form $\{\alpha\}s\{\beta\}$, where s is a straight-line program (potentially with calls to other methods encoded using their contracts). The validity of each Hoare triple is translated to a pure logical validity question, called the *verification condition* (VC). When α and β refer to data structure properties, the resulting VCs are typically proved using induction on the structure of the recursive definitions. Automation of program verification reduces to automating validity of the logic in which the VCs are expressed.

Logics that are powerful enough to express rich properties of data structures are invariably incomplete, not just undecidable, i.e., they do not admit any automated procedure that is complete (guaranteed to eventually prove any valid theorem, but need not terminate on invalid theorems). For instance, validity is incomplete for both first-order logic with least fixpoints and separation logic. Consequently, though verification frameworks like DAFNY [9] support rich specification languages, validation of verification conditions can fail even for valid Hoare triples. Automated verification engines hence support several heuristics resulting in sound but incomplete verification.

When proofs succeed in such systems, the verification engineer is happy that automation has taken the proof through. However, when proofs *fail*, as they often do, the verification engineer is stuck and perplexed. First, they would crosscheck to see whether their annotations are strong enough and that the Hoare triples are indeed valid. If they believe they are, they do not have clear guidelines to help the tool overcome its incompleteness. Engineers are instead required to know the *underlying proof mechanisms/heuristics* the verification system uses in order to figure out why the system is unable to succeed, and figure out how to help the system. For instance, for data structures with recursive definitions, the proof system may just unfold definitions a few times, and the engineer must be able to see why this heuristic will not be able to prove the theorem and formulate new inductively provable lemmas or quantification triggers that can help. Such *unpredictable* systems that require engineers to know their internal heuristics and proof mechanisms frustrate verification experience.

Predictable Verification In this chapter we seek an entirely new paradigm of *predictable* verification. We want a technique where:

(a) the verification engineer is asked to provide upfront a set of annotations that help prove programs correct, where these annotations are entirely *independent* of the verification mechanisms/tools, and

(b) the program verification problem, given these annotations, is guaranteed to be *decidable* (and preferably decidable using efficient engines such as SMT solvers).

The upfront agreement on the information that the verification engineer is required to provide makes their task crystal clear. The fact that the verification is decidable given these annotations ensures that the verification engine, given enough resources of time and space (of course) will eventually return proving the program correct or showing that the program or annotations are incorrect. There is no second-guessing by the engineer as the verification will never fail on valid theorems, and hence they need not worry about knowing how the verification engine works, or give further help. Note that the verification *without annotations* can (and typically will be) undecidable.

Intrinsic Definitions of Data Structures In this chapter we propose an entirely new way of defining data structures, called *intrinsic definitions*, that facilitates a predictable verification paradigm for proving their maintenance. Rather than defining data structures using recursion, like in equation (1) above (which naturally calls for inductive proofs and invariably entails incompleteness), we define data structures by augmenting each location with additional information using *ghost maps* and demanding that certain *local conditions* hold between each location and its neighbors.

Intrinsic definitions formally require a set of monadic maps (maps of arity one) that associate values to each location in a structure (we can think of these as ghost fields associated with each location/object). We demand that the monadic maps on local neighborhoods of every location satisfy certain logical conditions. The existence of maps that satisfy the local logical conditions ensures that the structure is a valid data structure.

For example, we can capture trees in pointer-based heaps in the following way. Let us introduce maps $\mathit{tree} : \mathit{Loc} \to \mathit{Bool}$, $\mathit{rank} : \mathit{Loc} \to \mathbb{Q}^+$ (non-negative rationals), and $p : \mathit{Loc} \to \mathit{Loc}$ (for "parent"), and demand the following local property:

$$\begin{aligned}\forall x :: \mathit{Loc}.(\mathit{tree}(x) \Rightarrow (\ & (l(x) \neq \mathit{Nil} \Rightarrow (\mathit{tree}(l(x)) \wedge p(l(x)) = x \wedge \mathit{rank}(l(x)) < \mathit{rank}(x))) \\ & \wedge (r(x) \neq \mathit{Nil} \Rightarrow (\mathit{tree}(r(x)) \wedge p(r(x)) = x \wedge \mathit{rank}(r(x)) < \mathit{rank}(x))) \\ & \wedge ((l(x) \neq \mathit{Nil} \wedge r(x) \neq \mathit{Nil}) \Rightarrow l(x) \neq r(x)) \\ & \wedge (p(x) \neq \mathit{Nil} \Rightarrow (r(p(x)) = x \vee l(p(x)) = x))) \ \ ) \end{aligned} \tag{4.2}$$

The above demands that ranks become smaller as one descends the tree, that a node is the parent of its children, and that a node is either the left or right child of its parent.

Given a *finite* heap, it is easy to see that if there exist maps *tree*, *rank* and p that satisfy the above property, and if $\mathit{tree}(l)$ is true for a location l, then l must point to a tree (strictly decreasing ranks ensure that there are no cycles and existence of a unique parent ensures that there are no "merges"). Furthermore, in any heap, if T is the subset of locations that

are roots of trees, then there are maps that satisfy the above property and have precisely $tree(l)$ to be true for locations in T.

Note that the above intrinsic definition *does not use recursion* or least fixpoint semantics. It simply requires maps such that each location satisfies the local neighborhood condition.

Fix-What-You-Break Program Verification Methodology Intrinsic definitions are particularly attractive for proving *maintenance* of structures when structures undergo mutation. When a program mutates a heap H to a heap H', we start with monadic maps that satisfy local conditions in the pre-state. As the heap H is modified, we ask the verification engineer to also *repair* the monadic maps, using ghost map updates, so that the local conditions on all locations are met in the heap in the post-state H'.

For instance, consider a program that walks down a tree from its root to a node x and introduces a newly allocated node n between x and x's right child r. Then we would assume in the precondition that the monadic maps *tree*, *rank*, and p exist satisfying the local condition (2) above. After the mutation, we would simply update these maps so that $tree(n)$ is true, $p(r) = n$, $p(n) = x$, and $rank(n)$ is, say, $(rank(x) + rank(r))/2$.

The annotations required of the user, therefore, are ghost map updates to locations such that the local conditions are valid for each location. We will guarantee that checking whether the local conditions holds for each location, after the repairs, is expressible in decidable logics.

We propose a modular verification approach for verifying data structure maintenance that asks the programmer to fix what they break. Given a program to verify, we instead verify an *augmented program* that keeps track of a ghost set of *broken locations* Br. Broken locations are those that (potentially) do not satisfy the local condition. When the program destructively modifies the fields at a location, it (and some of its neighbors, accessible using pointers from the object) may not satisfy the local condition anymore. These get added to the broken set. The verification engineer must repair the monadic maps on these broken locations and ensure (through an assertion) that the local condition holds on them before removing them from the broken set Br. However, while repairing monadic maps on a location, the local condition on *its neighboring* locations may fail and get added to the broken set.

We develop a *fix-what-you-break (FWYB)* program verification paradigm, giving formal rules of how to augment programs with broken sets, how users can modify monadic maps, and fixed recipes of how broken sets are maintained in any program. In order to verify that a method m maintains a data structure, we need to prove that if m starts with the broken set being empty, it returns with the empty broken set. We prove this methodology sound, i.e., if the program augmented with broken sets and ghost updates is correct, then the original program maintains the data structure properties mentioned in its contracts.

Decidable Verification of Annotated Programs The general idea of using local conditions to capture global properties has been explored in the literature to reduce the complexity of proofs (e.g., iterated separation in separation logic [60]; see Chapter 6). Intrinsic definitions of data structures and the fix-what-you-break program verification methodology are more specifically designed to ensure the key property of *decidable verification of annotated programs* by avoiding both recursion/least-fixpoint definitions and avoiding even quantified reasoning.

The verification conditions for Hoare triples involving basic blocks of our annotated programs have the following structure. First, the precondition can be captured using *uninterpreted monadic functions* that are *implicitly* assumed to satisfy the local condition on each location that is not in the broken set Br (avoiding universal quantification). The monadic map updates (repairs) that the verification engineer makes can be captured using map updates. The postcondition of the ghost-code augmented program can, in addition to properties of variables, assert properties of the broken set Br using logics over sets. Finally, we show that capturing the modified heap after function calls can be captured using *parameterized map update* theories, that are decidable [71]. Consequently, the entire verification condition is captured in quantifier-free logics involving maps, parametric map updates, and sets over combined theories. These verification conditions are hence decidable and efficiently handled by modern SMT solvers[1].

Intrinsic Definitions for Representative Data Structures and Verification in Boogie Intrinsic definitions of data structures is a novel paradigm and capturing data structures requires thinking anew in order to formulate monadic maps and local conditions that characterize them.

We give intrinsic definitions for several classic data structures such as linked lists, sorted lists, circular lists, trees, binary search trees, AVL trees, and red-black trees. These require novel definitions of monadic maps and local conditions. We also show how standard methods on these data structures (insertions, deletions, concatenations, rotations, balancing, etc.) can be verified using the fix-what-you-break strategy and standard loop invariant/contract annotations. We also consider *overlaid data structures* consisting of multiple data structures overlapping and sharing locations. In particular, we model the core of an overlaid data structure that is used in an I/O scheduler in Linux that has a linked list (modeling a FIFO queue) overlaid on a binary search tree (for efficient search over a key field). Intrinsic definitions beautifully capture such structures by compositionally combining the instrinsic

[1]Assuming of course that the underlying quantifier-free theories are decidable; for example, integer multiplication in the program or in local conditions would make verification undecidable, of course.

definitions for each structure and a local condition linking them together. We show methods to modify this structure are provable using fix-what-you-break verification.

We model the above data structures and the annotated methods in the low-level programming language Boogie. Boogie is an intermediate programming language with verification support that several high-level programming languages compile to for verification (e.g., C [72, 73], Dafny [9], Civl [74], Move [75]). These annotated programs do not use quantifiers or recursive definitions, and Boogie is able to verify them automatically using decidable verification in negligible time, without further user-help.

Contributions This chapter makes the following contributions:

- A new paradigm of *predictable verification* that asks upfront for programmatic annotations and ensures annotated program verification is decidable, without reliance on users to give heuristics and tactics.
- A novel notion of intrinsic definitions of data structures based on ghost monadic maps and local conditions.
- A predictable verification methodology for programs that manipulate data structures with intrinsic definitions following a fix-what-you-break (FWYB) methodology.
- Intrinsic definitions for several classic data structures, and fix-what-you-break annotations for programs that manipulate such structures, with realization of these programs and their verification using Boogie.

Outline This chapter is structured as follows. In Section 4.2 we define the notion of heap datastructure we work with and develop our first main contribution, namely the notion of intrinsic definitions for datastructures. In Section 4.3 we briefly define a small while programming language and formalize the notion of correctness that we work with. Importantly, we also formalize the notion of ghost code (Section 4.3.2) that we use our study. In Section 4.4 we begin the development of our second main contribution, the Fix-What-You-Break methodology. We show that FWYB ultimately reduces to an intuitive programming discipline that we dub *well-behaved programming*, and prove the soundness of our construction in Section 4.5. In Section 4.6 we illustrate a full example worked out in the FWYB methodology and use it to develop the concepts of a language paradigm where programs are well-behaved by design. We then apply this paradigm to several illustrative case studies that highlight various aspects of programming in the FWYB methodology in Section 4.7. Finally, we describe our implementation of the language paradigm as well

as FWYB in BOOGIE and discuss the result of evaluating our technique on a suite of benchmarks.

4.2 INTRINSIC DEFINITIONS OF DATA STRUCTURES

In this section we present intrinsic definitions for data structures. We first define the notion of a data structure in a pointer-based heap.

4.2.1 Data Structures

In this work we consider data structures defined using a *class* C of objects. The class C can coexist with other classes, heaps, and data structures, potentially modeled and reasoned with using other mechanisms. For technical exposition and simplicity, we restrict the technical definitions to a single class of data structures over a class C.

A class C has a signature $(\mathcal{S}, \mathcal{F})$ consisting of a finite set of sorts $\mathcal{S} = \{\sigma_0, \sigma_1 \ldots, \sigma_n\}$ and a finite set of fields $\mathcal{F} = \{f_1, f_2 \ldots, f_m\}$. We assume without loss of generality that the sort σ_0 represents the sort of objects of the class C, and we denote this sort by C itself. We use C to model objects in the heap. The other "background" sorts, e.g., integers, are used to model the values of the objects' fields. Each field $f_i : C \to \sigma$ is a unary function symbol and is used to model pointer and data fields of heap locations/objects. We model *Nil* as a non-object value and denote the sort $C \uplus \{Nil\}$ consisting of objects as well as the *Nil* value by $C?$.

A C-heap H is a *finite* first-order model of the signature of C. More formally, it is a pair (O, I) where O is a finite set of *objects* interpreting the foreground sort C and I is an interpretation of every field in $\mathcal{F}$ for every object in O.

Example 4.1 (C-Heap)**.** Let C be the class consisting of a pointer field $next : C \to C?$ and a data field $key : C \to Int$. The figure on the right represents a C-heap consisting of objects $O = \{o_1, o_2\}$ and the illustrated interpretation I for *next* and *key*.

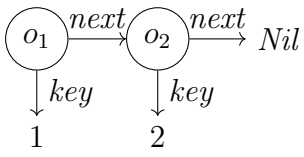


Figure 4.1: A C-Heap.

We now define a data structure. We fix a class C.

Definition 4.1 (Data Structure)**.** A data structure D of arity k is a set of triples of the form $(O, I, \overline{o})$ such that (O, I) is a C-heap and $\overline{o}$ is a k-tuple of objects from O.

Informally, a data structure is a particular subset of C-heaps along with a distinguished tuple of locations $\overline{o}$ in the heap that serve as the "entry points" into the data structure, such as the root of a tree or the ends of a linked list segment.

Example 4.2 (Sorted Linked List)**.** Let C be the class defined in Example 4.1. The data structure of sorted linked lists is the set of all (O, I, o_1) such that O contains objects $o_1, o_2 \ldots o_n$ with the interpretation $next(o_i) = o_{i+1}$ and $key(o_i) \leq key(o_{i+1})$ for every $1 \leq i < n$, and $next(o_n) = Nil$. For example, let (O, I) be the C-heap described in Example 4.1. The triple (O, I, o_1) is an example of a sorted linked list. Here o_1 is the head of the sorted linked list.

4.2.2 Intrinsic Definitions of Data Structures

We now propose a characterization of data structures using *intrinsic definitions*. Intrinsic definitions consist of a set of *monadic maps* that associate (ghost) values to each object and a set of *local* conditions that constrain the monadic maps on each location and its neighbors. A C-heap is considered to be a valid data structure if *there exists* a set of monadic maps for the heap that satisfy the local conditions.

Annotations using intrinsic definitions enable local and decidable reasoning for correctness of programs manipulating data structures using the Fix-What-You-Break (FWYB) methodology, which is described later in Section 4.4. We develop the core idea of intrinsic definitions below.

Ghost Monadic Maps We denote by $C_{\mathcal{G}} = (\mathcal{S}, \mathcal{F} \cup \mathcal{G})$ an extension of C with a finite set of monadic (i.e., unary) function symbols $\mathcal{G}$. We can think of these as *ghost* fields of objects.

The key idea behind intrinsic definitions is to extend a C-heap with a set of ghost monadic maps and formulate local conditions using the maps that characterize the heaps belonging to the data structure. The existence of such ghost maps satisfying the local conditions is then the intrinsic definition. Definitions are parameterized by a multi-sorted first-order logic $\mathcal{L}$ in which local conditions are stated. The logic has the sorts $\mathcal{S}$ and contains the function symbols in $\mathcal{F} \cup \mathcal{G}$, as well as interpreted functions over background sorts (such as $+$ and $<$ on integers, and $\subseteq$ on sets).

Definition 4.2 (Intrinsic Definition)**.** Let $C = (\mathcal{S}, \mathcal{F})$ be a class. An intrinsic definition $IDS(\overline{y})$ over the class C is a tuple $(\mathcal{G}, \mathcal{L}, LC, \varphi(\overline{y}))$ where:

1. $\mathcal{G}$ is a finite set of *monadic map names and function signatures* disjoint from $\mathcal{F}$,
2. $\mathcal{L}$ is a first-order logic over the sorts $\mathcal{S}$ containing the interpreted functions of the background sorts as well as the function symbols in $\mathcal{F} \cup \mathcal{G}$,
3. A *local condition* formula LC of the form $\forall x : Loc.\, \rho(x)$ such that ρ is a quantifier-free $\mathcal{L}$-formula, and
4. A *correlation formula* $\varphi(\overline{y})$ that is a quantifier-free $\mathcal{L}$-formula over free variables $\overline{y} \in Loc$.

We denote an intrinsic definition by $(\mathcal{G}, LC, \varphi(\overline{y}))$ when the logic $\mathcal{L}$ is clear from context. In our work $\mathcal{L}$ is typically a decidable combination of quantifier-free theories [29, 30, 76], containing theories of integers, sets, arrays [71], etc., supported effectively in practice by SMT solvers [11, 12].

Definition 4.3 (Data Structures defined by Intrinsic Definitions)**.** Let $C = (\mathcal{S}, \mathcal{F})$ be a class and $IDS(\overline{y}) = (\mathcal{G}, LC, \varphi(\overline{y}))$ be an intrinsic definition over C consisting of monadic maps $\mathcal{G}$, local condition LC and correlation formula φ. The data structure defined by IDS is precisely the set of all $(O, I, \overline{o})$ where *there exists* an interpretation J that extends I with interpretations for the symbols in $\mathcal{G}$ such that $O, J \models LC$ and $O, J[\overline{y} \mapsto \overline{o}] \models \varphi(\overline{y})$, where $[\overline{y} \mapsto \overline{o}]$ denotes that the free variables $\overline{y}$ are interpreted as $\overline{o}$.

Informally, given a data structure DS consisting of triples $(O, I, \overline{o})$, an intrinsic definition demands that there exist monadic maps $\mathcal{G}$ such that the C-heaps (O, I) in the data structure can be extended with values for maps in $\mathcal{G}$ satisfying the local conditions LC, and the entrypoints $\overline{o}$ are characterized in the extension by the quantifier-free formula φ.

Example 4.3 (Sorted Linked List)**.** Recall the data structure of sorted linked lists defined in Example 4.2. We capture sorted linked lists by an intrinsic definition $SortedLL(y)$ using monadic maps $sortedll : C \to Bool$ and $rank : C \to \mathbb{Q}^+$ such that:

$$\begin{aligned} LC \equiv \forall x.\ \Big(& (sortedll(x) \ \wedge \ next(x) \neq Nil) \Rightarrow \\ & (sortedll(next(x)) \ \wedge \ rank(next(x)) < rank(x) \ \wedge \ key(x) \leq key(next(x))) \Big) \\ \varphi(y) \equiv \ & sortedll(y) \end{aligned}$$

In the above definition the *rank* field decreases wherever *sortedll* holds as we take the *next* pointer, and hence assures that there are no cycles. Observe that without the constraint on *rank*, the triple $(\{o_1, o_2\}, I, o_1)$ where $I = \{next(o_1) = o_2, next(o_2) = o_1, key(o_1) = key(o_2) = 0\}$ denoting a two-element circular list would satisfy the definition, which is undesirable.

Note that the above allows for a heap to contain both sorted lists as well as unsorted lists. We are guaranteed by the local condition that the set of all objects where *sortedll* is true will be the heads of sorted lists.

We can also replace the domain of ranks in the above definition using any strict partial order, say integers or reals (with the usual $<$ order on them), and the definition will continue to define sorted lists. Well-foundedness of the order is not important as heaps are *finite* in our work (see definition of C-heaps in Section 4.2.1)

$$
\begin{aligned}
P \coloneqq\ & x := \mathit{Nil} \mid x := y \mid v := be \mid y := x.f \mid v := x.d \\
& \mid x.f := y \mid x.d := v \mid x := \mathsf{new}\ C() \mid \overline{r} := \mathit{Function}(\overline{t}) \\
& \mid \mathsf{skip} \mid \mathsf{assume}\ \mathit{cond} \mid \mathsf{return} \\
& \mid P\,;\, P \mid \mathsf{if}\ \mathit{cond}\ \mathsf{then}\ P\ \mathsf{else}\ P \mid \mathsf{while\ cond\ do}\ P \\
\mathit{cond} \coloneqq\ & x = y \mid x \neq y \mid be \quad \text{(Condition Expressions)}
\end{aligned}
$$

Figure 4.2: Grammar of while programs with recursion. x, y are variables denoting objects of class C? (i.e., C objects or *Nil*), v, w are a background sort(s) variables, r, t denote variables of any sort, f is a pointer field, d is a data field, and be is a expression of the background sort(s).

4.3 PRELIMINARIES: PROGRAMS, CORRECTNESS, AND GHOST CODE

We describe here a few concepts that are relevant to our development of the FWYB verification technique in Section 4.4. We recommend that the reader peruse the concepts somewhat quickly following the prose (pausing perhaps for Examples) and return to the formal definitions and statements when they are mentioned in the following Sections.

We begin by describing a while programming language and defining the verification problem we study. We fix a class $C = (\mathcal{S}, \mathcal{F})$ throughout this section.

4.3.1 Programs, Contracts, and Correctness

Programs Figure 4.2 shows the programming language. Note that we have variables and expressions over non-object sorts. Functions can return multiple outputs. We assume that method signatures contain designated output variables and therefore the return statement does not mention values.

Our language is safe (i.e., allocated locations cannot point to un-allocated locations) and garbage-collected. Formally we consider configurations θ consisting of a store (map from variables to values) and a heap along with an error state $\perp$ to model error on a null dereference. We denote that a formula α is satisfied on a configuration θ by writing $\theta \models \alpha$. We elide the definition of a formal operational semantics for now as it is the usual one for pointer-based programs over heaps. We provide the formal presentation in an addendum at the end of the chapter.

Intrinsic Hoare Triples In this chapter we study verifying the *maintenance* of data structure properties. Fix an intrinsic definition $(\mathcal{G}, LC, \varphi(\overline{y}))$ where $\mathcal{G} = \{g_1, g_2 \ldots, g_k\}$. Let

$\overline{z}$ be the input/output variables for a program that we want to verify. We consider pre and post conditions of the form

$$\exists\, g_1, g_2 \ldots, g_k.\, (LC \wedge \varphi(\overline{w}) \wedge \psi(\overline{z})) \tag{4.3}$$

where each g_i is a ghost monadic map (unary function over locations), ψ is a quantifier-free formula over $\overline{z}$ that can use the ghost monadic maps g_i, and $\overline{w}$ is a tuple of variables from $\overline{z}$ whose arity is equal to $\overline{y}$. Note that the above has a second-order existential quantification ($\exists$) over function symbols $g_1, \ldots, g_k$, and LC has first-order universal quantification over a single location variable. Read in plain English, "$\overline{w}$ points to a data structure (defined by the LC and φ) such that the (quantifier-free) property $\psi(\overline{z})$ holds".
We study the validity of the following Hoare Triples:

$$\langle\, \alpha(\overline{x}) \,\rangle \;\; \mathrm{P}(\overline{x},\, ret\colon \overline{r}) \;\; \langle\, \beta(\overline{x}, \overline{r}) \,\rangle \tag{4.4}$$

where α and β are pre and post conditions of the above form, P is a program, and $\overline{x}, \overline{r}$ are input and output variables for P respectively.

Example 4.4 (Running Example: Insertion into a Sorted List)**.** Let $SortedLL(y) = (\mathcal{G}, LC, sorted(y))$ as in Example 4.3 where $\mathcal{G} = \{sortedll, rank\}$. The following Hoare triple says that insertion into a sorted list returns a sorted list:

$$\begin{gathered}\langle\, \exists\, sortedll, rank.\, LC \wedge sortedll(x) \,\rangle \\ sorted\text{–}insert(x, k,\, ret\colon x) \\ \langle\, \exists\, sortedll, rank.\, LC \wedge sortedll(x) \,\rangle\end{gathered}$$

where x, r have type C, k has type Int and *sorted–insert* is the usual recursive method.

We now define the validity of Hoare Triples.

Definition 4.4 (Validity of Intrinsic Hoare Triples)**.** An intrinsic triple $\langle\, \alpha \,\rangle\, P\, \langle\, \beta \,\rangle$ is *valid* if for every configuration θ such that $\theta \models \alpha$, transitioning according to P starting from θ does not encounter the error state $\perp$, and furthermore, if θ transitions to θ' under P, then $\theta' \models \beta$.

4.3.2 Ghost Code

In this chapter we consider the augmentation of procedures with *ghost* or non-executed code. Ghost code involves the manipulation of a set of distinct *ghost variables* and *ghost fields*, distinguished from regular or 'user' variables and fields. In program verification, ghost code provides a programmatic way of constructing values/functions that witness a property.

Intuitively, ghost variables/fields cannot influence the computation of non-ghost variables/-fields. Therefore, ghost variables and maps can be assigned values from user variables and maps, but the reverse is not allowed. Similarly, when conditional statements or loops use ghost variables in the condition, the body of the statement must also consist entirely of ghost code. Simply, ghost code cannot control the flow of the user program. These conditions can be checked statically. Finally, we also require that ghost loops and functions always terminate since nonterminating ghost code can change the meaning of the original program. Our definition is agnostic to the technique used to establish termination, however, we use ranking functions to establish termination in our implementation in Dafny.

Fix a set of user variables Var_U and ghost variables Var_G. Also recall user fields $\mathcal{F}$ and ghost fields/maps $\mathcal{G}$ introduced in Section 4.2.2. We formalize ghost code using a grammar in Figure 4.3 which defines a ghost-code augmented language extending the original programming language in Figure 4.2. The grammar for pure ghost code is similar to the grammar for the original language P except that we do not have allocation or assume statements, and loops/functions must always terminate. See prior literature for a more detailed formal treatment of ghost code [77, 78, 79, 80].

Projection that Eliminates Ghost Code We now define the notion of 'projecting out' ghost code, which takes a program that contains ghost code and yields a pure user program by eliminating all ghost code. Intuitively, the fact that ghost code does not affect the execution of the underlying user program makes the projection operation sensible.

Fix a main method M with body Q_0. Let $N_i, 1 \leq i \leq k$ be a set of auxiliary methods with bodies Q_i that Q_0 can call. Note that the bodies Q_0 and Q_i contain ghost code. Let us denote a program containing these methods by $[(M : Q_0); (N_1 : Q_1) \ldots (N_k : Q_k)]$.

Definition 4.5 (Projection of Ghost-Augmented Code to User Code)**.** The projection of the ghost-augmented program $[(M : Q_0); (N_1 : Q_1) \ldots (N_k : Q_k)]$ is the user program $[(\hat{M} : \hat{Q_0}); (\hat{N_1} : \hat{Q_1}) \ldots (\hat{N_k} : \hat{Q_k})]$ such that:

1. The input (resp. output) signature of $\hat{M}$ is that of M with the ghost input (resp. output) parameters removed. Formally, given a sequence of parameters $\overline{x}$ with some elements in the sequence marked as ghost, we can define the projection as the sequence formed by the non-continguous subsequence of parameters in $\overline{x}$ consisting of non-ghost parameters.

2. $\hat{Q_0}$ is derived from Q_0 by: (a) eliminating all ghost code, i.e., replacing yields of the nonterminal GP in Figure 4.3 with skip, and (b) replacing each non-ghost function

$$
\begin{array}{rl}
P \coloneqq & x := \mathit{Expr}[\mathit{Var}_U, \mathcal{F}] \;\mid\; y := x.f \;\mid\; x.f := y \;\mid\; z := \mathsf{new}\ C() \\
& \mid\; \overline{r} := \mathit{Func}(\overline{t}) \qquad \overline{r}, \overline{t} \text{ are variables in } \mathit{Var}_U \cup \mathit{Var}_G \\
& \text{(Functions can have ghost input/output parameters)} \\
& \mid\; GP \\
& \text{(GP are ``pure'' ghost programs)} \\
& \mid\; \mathsf{skip} \;\mid\; \mathsf{assume}\ \mathit{cond} \;\mid\; \mathsf{return} \\
& \mid\; P\,;\, P \;\mid\; \mathsf{if}\ \mathit{cond}\ \mathsf{then}\ P\ \mathsf{else}\ P \;\mid\; \mathsf{while\ cond\ do}\ P \\
\mathit{cond} \coloneqq & \mathit{BoolExpr}[\mathit{Var}_U, \mathcal{F}] \\
GP \coloneqq & a := \mathit{Expr}[\mathit{Var}_U \cup \mathit{Var}_G, \mathcal{F} \cup \mathcal{G}] \;\mid\; b := x.g \;\mid\; b := x.f \\
& \text{(Ghost variables can read from both user and ghost variables/maps)} \\
& \mid\; x.g := b \;\mid\; x.g := y \\
& \text{(Ghost maps can only be assigned values from ghost variables)} \\
& \mid\; \overline{s} := \mathit{GhostFun}(\overline{v})\ \ \overline{s}, \overline{v} \text{ variables in } \mathit{Var}_G,\ \mathit{GhostFun} \text{ is } \textbf{always terminating} \\
& \mid\; \mathsf{skip} \;\mid\; GP\,;\, GP \;\mid\; \mathsf{if}\ \mathit{Gcond}\ \mathsf{then}\ GP\ \mathsf{else}\ GP \\
& \mid\; \mathsf{while\ Gcond\ do}\ GP \quad \text{loop is } \textbf{always terminating} \\
\mathit{Gcond} \coloneqq & \mathit{BoolExpr}[\mathit{Var}_U \cup \mathit{Var}_G, \mathcal{F} \cup \mathcal{G}]
\end{array}
$$

Figure 4.3: Grammar of programs with ghost code. x, y, z are user variables Var_U, a, b are ghost variables Var_G, $f \in \mathcal{F}$ is a user field, and $g \in \mathcal{G}$ is a ghost map. Notation $\mathit{Expr}[\mathit{Vars}, \mathit{Maps}]$ denotes expressions over the vocabulary given by variables Vars and maps Maps, similarly $\mathit{BoolExpr}[\mathit{Vars}, \mathit{Maps}]$ denotes boolean expressions. Termination for ghost loops and functions can be established in any way.

call statement of the form $\overline{r} := N_j(\overline{t})$ with the statement $\overline{s} := \hat{N}_j(\overline{u})$, where $\overline{u}, \overline{s}$ are obtained from $\overline{t}, \overline{r}$ by projecting out the elements corresponding to the ghost parameters in the signature of N_j. Each $\hat{Q}_i$ is derived from the corresponding Q_i by a similar transformation. Formally, we use the following recursive transformation:

$$
\begin{array}{rl}
\mathit{Projection}(GP) = & \mathsf{skip} \\
\mathit{Projection}(\overline{r} := \mathit{Func}(\overline{t})) = & \overline{s} := \hat{\mathit{Func}}(\overline{u}) \\
& \overline{u}, \overline{s} \text{ are obtained from } \overline{t}, \overline{r} \text{ by projecting out} \\
& \text{elements corresponding to ghost parameters} \\
\mathit{Projection}(\mathsf{stmt}) = & \mathsf{stmt} \quad \text{for all other statements} \\
\mathit{Projection}(P_1;\, P_2) = & \mathit{Projection}(P_1)\,;\, \mathit{Projection}(P_2) \\
\mathit{Projection}(\mathsf{if}\ \mathit{cond}\ \mathsf{then}\ P_1\ \mathsf{else}\ P_2) = & \mathsf{if}\ \mathit{cond}\ \mathsf{then}\ \mathit{Projection}(P_1)\ \mathsf{else}\ \mathit{Projection}(P_2) \\
\mathit{Projection}(\mathsf{while\ cond\ do}\ P) = & \mathsf{while\ cond\ do}\ \mathit{Projection}(P)
\end{array}
$$

Properties of Projection We now show some results about projection.

Projection for Configurations Projection deals with two kinds of triples, one whose validity is stated with respect to configurations that interpret ghost variables and maps, and one over configurations that only interpret user variables and fields. Given a configuration C that interprets ghost variables/maps, we denote by $\hat{C}$ the projection of that configuration to user variables that simply eliminates all ghost interpretations. Conversely, given a configuration c we say that C extends c with an interpretation for ghost variables/maps if $\hat{C} = c$. We define $\hat{bot} = \bot$.

We assume that there is only one procedure M in the program for simplicity of presentation. Recall that M can contain ghost code and $\hat{M}$ is the projection of M that eliminates the ghost code (with appropriately modified input/output parameters).

Lemma 4.1. Let C_1 be a configuration that interprets ghost variables/maps. If M is a "pure ghost" program, (i.e., a yield of GP in the grammar in Figure 4.3), then M always terminates starting from C_1.

The above lemma says that pure ghost programs always terminate. It follows directly from the definition of ghost code which requires pure ghost loops and functions to be terminating. QED.

Lemma 4.2. Let c be a configuration that does not interpret ghost variables/maps. If $\hat{M}$ (projected code that does not contain ghost code) terminates starting from c, then M (which contains additional ghost code) must terminate starting from any configuration C that extends c.

The above lemma says that the termination of the original user program is preserved by any augmentation with ghost code. In a certain sense, it 'lifts' Lemma 4.1 to programs that contain both user and ghost code.

Proof. The lemma follows from structural induction on the definition of projection, i.e., on the structure of the grammar for the nonterminal P in Figure 4.3. The argument for basic statements is trivial. For pure ghost programs the result follows from Lemma 4.1. The argument for all compositions (sequential, conditional, loop) and function calls follows from the induction hypothesis. QED.

Lemma 4.3. Let C_1 be a configuration that interprets ghost variables/maps. If M is a "pure ghost" program and M starting from C_1 reaches some C_2 and $C_2 \neq \bot$, then $\hat{C_1} = \hat{C_2}$.

The above lemma says that ghost code does not affect the values of user (non-ghost) variables and maps. It follows trivially by structural induction on the *GP* grammar, using the definition of operational semantics (Figure 4.17). The key observation is that *GP* syntactically disallows non-ghost variables/maps to read from ghost variables/maps. QED.

We can similarly 'lift' the above lemma to programs that contain both user code and ghost code.

Lemma 4.4. Let c be a configuration that does not interpret ghost variables/maps. If $\hat{M}$ starting from c_1 reaches some c_2, then M starting from any configuration C_1 that extends c_1 must either reach $\bot$ or some C_2 that extends c_2.

The above lemma says that augmentation with ghost code does not affect how the original program executes.

Proof. As with Lemma 4.2, we proceed by structural induction on the grammar for P in Figure 4.3. The argument for basic non-ghost statements follows trivially from the definition of operational semantics. They key observation is that non-ghost statements do not affect the values of ghost variables/maps (ensured by the syntactic restrictions). For pure ghost programs the result follows from Lemma 4.3. The argument for all compositions (sequential, conditional, loop) and function calls follows from the induction hypothesis. QED.

4.4 FIX WHAT YOU BREAK (FWYB) VERIFICATION METHODOLOGY

In this section we present the second main contribution of this chapter: the Fix-What-You-Break (FWYB) methodology. As with the previous section, we fix a class $C = (\mathcal{S}, \mathcal{F})$ throughout the presentation.

Overview of FWYB We develop the Fix-What-You-Break (FWYB) methodology in three stages, in the following subsections. We give here an overview of the methodology and the stages.

We start with triples of the form $\langle\, \exists\, g_1, g_2 \ldots, g_k.\, (LC \wedge \varphi \wedge \alpha) \,\rangle\ P\ \langle\, \exists\, g_1, g_2 \ldots, g_k.\, (LC \wedge \varphi \wedge \beta) \,\rangle$. In Stage 1 (Section 4.4.1) we remove the second-order quantification. We do this by requiring the verification engineer to explicitly construct the g_i maps in the post state from the maps in the pre state using *ghost code*. We then obtain triples of the form $\langle\, LC \wedge \varphi \wedge \alpha \,\rangle\ P_{\mathcal{G}}\ \langle\, LC \wedge \varphi \wedge \beta \,\rangle$ where $P_{\mathcal{G}}$ is an augmentation of P with ghost code that updates the $\mathcal{G}$ maps.

Note that the LC in the contract universally quantifies over objects. In Stages 2 (Section 4.4.2) and 3 (Section 4.4.3) we remove the quantification by explicitly tracking the objects where the local conditions do not hold and treating them as implicitly true on all other objects. We call this set Br the *broken set*. Intuitively, the broken set grows when the program mutates pointers or makes other changes to the heap, and shrinks when the verification engineer repairs the $\mathcal{G}$ maps using ghost code to satisfy the LC on the broken objects. The specifications assume an empty broken set at the beginning of the program and the engineer must ensure that it is empty again at the end of the program. However, they do not have to track the objects manually. We develop in Stage 3 (Section 4.4.3) a discipline for writing only *well-behaved* manipulations of the broken set. This reduces the problem to triples of the form $\langle\, \varphi \wedge \alpha \,\rangle\ P_{\mathcal{G},Br}\ \langle\, \varphi \wedge \beta \,\rangle$, where $P_{\mathcal{G},Br}$ contains ghost code for updating both $\mathcal{G}$ and Br. Note that these specifications are quantifier-free, and checking them can be effectively automated using SMT solvers [11, 12].

4.4.1 Stage 1: Removing Existential Quantification over Monadic Maps using Ghost Code

Consider an intrinsic Hoare Triple

$$\langle\, \exists\, g_1, g_2 \ldots, g_k.\, (LC \wedge \varphi \wedge \alpha) \,\rangle\ P\ \langle\, \exists\, g_1, g_2 \ldots, g_k.\, (LC \wedge \varphi \wedge \beta) \,\rangle \tag{4.5}$$

Read as a proof obligation, the precondition says that *there exist* maps $\{g_i\}$ satisfying some properties, and the postcondition says that we must *show the existence* of maps $\{g_i\}$ satisfying the post state properties.

We remove existential quantification by re-formulating the problem: assuming that we are *given* the maps $\{g_i\}$ as part of the pre state such that they satisfy $LC \wedge \varphi \wedge \alpha$, we ask the verification engineer to *compute* the $\{g_i\}$ maps in the post state satisfying $LC \wedge \varphi \wedge \beta$. The engineer computes the post state maps by taking the given pre state maps and 'repairing' them on an object whenever the program breaks local conditions on that object. The repairs are done using ghost code, which is a common technique in verification literature [77, 78, 79, 80].

Formally, fix an intrinsic data structure $(\mathcal{G}, LC, \varphi)$. We extend the class signature $C = (\mathcal{S}, \mathcal{F})$ to $C_{\mathcal{G}} = (\mathcal{S}, \mathcal{F} \cup \mathcal{G})$ and treat the symbols in $\mathcal{G}$ as *ghost fields* of objects of class C in programs. Performing the above transformation reduces the problem to proving triples of the form

$$\langle\, LC \wedge \varphi \wedge \alpha \,\rangle\ P_{\mathcal{G}}\ \langle\, LC \wedge \varphi \wedge \beta \,\rangle \tag{4.6}$$

where there is no existential quantification over $\mathcal{G}$ and $P_{\mathcal{G}}$ is an augmentation of P with ghost code that updates the $\mathcal{G}$ maps. The following proposition captures the correctness of this reduction:

Proposition 4.1. Let ψ_{pre} and ψ_{post} be quantifier-free formulae over $\mathcal{F} \cup \mathcal{G}$. If the $\exists$-free triple

$\langle\, LC \wedge \psi_{pre} \,\rangle\ P_{\mathcal{G}}\ \langle\, LC \wedge \psi_{post} \,\rangle$ is valid then $\langle\, \exists g_1, g_2 \ldots, g_k.\, LC \wedge \psi_{pre} \,\rangle\ P\ \langle\, \exists g_1, g_2 \ldots, g_k.\, LC \wedge \psi_{post} \,\rangle$ is valid [2], where P is the projection of $P_{\mathcal{G}}$ obtained by eliminating ghost code.

We wish to focus the presentation in this section on the development of the methodology itself as well as the statements of the soundness results. We hence defer a detailed proof to Section 4.5 and only furnish a gist here.

Proof Gist. The first Hoare triple shows that if we are given *any* maps g_i (implicitly encoded as values of ghost fields) that satisfy LC in the pre-state, then the program with ghost code computes a modified version of these maps such that the LC is holds in the post-state. Surely then, if there was a set of maps g_i that satisfied LC in the pre-state, there will exists a set of maps g_i' that satisfy LC in the post-state. QED.

We note a point of subtlety about the reduction in this stage here: the simplified triple eliminates existential quantification over $\mathcal{G}$ by claiming something stronger than the original specification, namely that for *any* maps $\{g_i\}$ such that ψ_{pre} is satisfied in the pre state, there is a *computation* that yields corresponding maps in the post state such that ψ_{post} holds. The onus of coming up with such a computation is placed on the verification engineer.

4.4.2 Stage 2: Relaxing Universal Quantification using Broken Sets

We turn to verifying programs whose pre and post conditions are of the form $LC \wedge \gamma$, where $LC \equiv \forall z.\, \rho(z)$ is the local condition. Consider a program P that maintains the data structure. The local conditions are satisfied everywhere in both the pre and post state of P. However, they need not hold everywhere in the intermediate states. In particular, P may call a method N which may neither receive nor return a proper data structure. To reason about P modularly we must be able to express contracts for methods like N. To do this we must be able to talk about program states where only some objects may satisfy the local conditions.

Broken Sets We introduce in programs a ghost set variable Br that represents the set of (potentially) broken objects. Intuitively, at any point in the program the local conditions must always be satisfied on every object that is *not* in the broken set. Formally, for a program P we extend the signature of P with Br as an additional input and an additional output. We also write pre and post conditions of the form $(\forall z \notin Br.\, \rho(z)) \wedge \gamma$ to denote that local conditions are satisfied everywhere outside the broken set, where γ can now use Br. In particular, given the Hoare triple

[2] Here the notion of validity for both triples is given by Definition 4.4, where configurations are interpreted appropriately with or without the ghost fields.

$$\langle\, (\forall z.\, \rho(z)) \wedge \alpha \,\rangle\ P_{\mathcal{G}}(\overline{x},\ ret\colon\ \overline{y})\ \langle\, (\forall z.\, \rho(z)) \wedge \beta \,\rangle \tag{4.7}$$

from Stage 1, we now reduce the problem to proving the following triple valid:

$$\langle\, (\forall z \notin Br.\, \rho(z)) \wedge \alpha \wedge Br = \emptyset \,\rangle\ P_{\mathcal{G},Br}(\overline{x}, Br,\ ret\colon\ \overline{y}, Br)\ \langle\, (\forall z \notin Br.\, \rho(z)) \wedge \beta \wedge Br = \emptyset \,\rangle \tag{4.8}$$

where Br is a ghost input variable of the type of set of objects and $P_{\mathcal{G},Br}$ is an augmentation of P with ghost code that computes the $\mathcal{G}$ maps as well as the Br set satisfying the postcondition.

P may also call other methods N with bodies Q. We similarly extend the input and output signatures of the called methods and use the broken set to write appropriate contracts for the methods, introducing triples of the form $\langle\, (\forall z \notin Br.\, \rho(z)) \wedge \alpha_N \,\rangle\ Q_{Br}(\overline{s}, Br,\ ret : \overline{r}, Br)\ \langle\, (\forall z \notin Br.\, \rho(z)) \wedge \beta_N \,\rangle$. Again, $Q_{\mathcal{G},Br}$ is an augmentation of Q with ghost code that updates $\mathcal{G}$ and Br.

For the main method that preserves the data structure property, the broken set is empty at the beginning and end of the program. However, called methods or loop invariants can talk about states with nonempty broken sets. We require the verification engineer to write ghost code that maintains the broken set accurately. The soundness of this reduction is captured by the following Proposition:

Proposition 4.2. Let α and β be quantifier-free formulae over $\mathcal{F} \cup \mathcal{G}$ (they cannot mention Br). If $\langle\, (\forall z \notin Br.\, \rho(z)) \wedge \alpha \wedge Br = \emptyset \,\rangle\ P_{\mathcal{G},Br}(\overline{x}, Br,\ ret\colon\ \overline{y}, Br)\ \langle\, (\forall z \notin Br.\, \rho(z)) \wedge \beta \wedge Br = \emptyset \,\rangle$ is valid then $\langle\, (\forall z.\, \rho(z)) \wedge \alpha \,\rangle\ P_{\mathcal{G}}(\overline{x},\ ret\colon\ \overline{y})\ \langle\, (\forall z.\, \rho(z)) \wedge \beta \,\rangle$ is valid, where $P_{\mathcal{G}}$ is the projection of $P_{\mathcal{G},Br}$ obtained by eliminating the statements that manipulate Br.

The proof of this proposition is similar to the proof of Proposition 4.1, except that projections only eliminate Br. We provide a detailed proof in Section 4.5.

4.4.3 Stage 3: Eliminating the Universal Quantifier for Well-Behaved Programs

We consider triples of the form

$$\langle\, (\forall z \notin Br.\, \rho(z)) \wedge \alpha \,\rangle\ P_{\mathcal{G},Br}(\overline{x}, Br,\ ret\colon\ \overline{y}, Br)\ \langle\, (\forall z \notin Br.\, \rho(z)) \wedge \beta \,\rangle \tag{4.9}$$

where $P_{\mathcal{G},Br}$ is a program augmented with ghost updates to the $\mathcal{G}$-fields as well as the Br set, and α, β are quantifier-free formulae that can also mention the fields in $\mathcal{G}$ and the Br set. In this stage we would like to eliminate the quantified conjunct entirely and instead ask the engineer to prove the validity of the triple

$$\{\alpha\}\ P_{\mathcal{G},Br}(\overline{x}, Br,\ ret\colon\ \overline{y}, Br)\ \{\beta\} \tag{4.10}$$

However, the above two triples are not, in general, equivalent (as broken sets can be manipulated wildly). In this section we define a syntactic class of *well-behaved* programs that force the verification engineer to maintain broken sets correctly, and for such programs the above triple are indeed equivalent. For example, for a field mutation, well-behaved programs require the engineer to determine the set of *impacted objects* where local conditions may be broken by the mutation. The well-behavedness paradigm then mandates that the engineer add the set of impacted objects to the broken set immediately following the mutation statement. Similarly, well-behaved programs do not allow the engineer to remove an object from the broken set unless they show that the local conditions hold on that object. The imposition of this discipline ensures that programmers carefully preserve the meaning of the broken set (i.e., objects outside the broken set must satisfy local conditions). This allows for the quantified conjunct in the triple obtained from Stage 2 to be dropped since it always holds for a well-behaved program. Let us look at such a program:

Example 4.5 (Well-Behaved Sorted List Insertion)**.** We use the running example (Example 4.4) of insertion into a sorted list. We consider a snippet where the key k to be inserted lies between the keys of x and $next(x)$ (which we assume is not Nil). We ignore the conditionals that determine $next(x) \neq Nil$ and $key(x) \leq k \leq key(next(x))$ for brevity.

We first relax the universal quantification as described in Stage 2 (Section 4.4.2) and rewrite the pre and post conditions to $(\forall\, z \notin Br.\, LC(z)) \wedge sortedll(x) \wedge Br = \emptyset$. Making the first conjunct implicit, we write the following program that manipulates the broken set in a well-behaved manner. We show the value of Br through the program in comments:

```
pre: sortedll(x) ∧ Br = ∅                z.sortedll := True;
post: sortedll(x) ∧ Br = ∅               Br := Br ∪ {z}; // {z}
 assert x ∉ Br;                          x.next := z;
 assume LC(x);                           Br := Br ∪ {x}; // {x,z}
 y := x.next;     // {}                  z.rank := (x.rank + y.rank)/2;
 z := new C();                           Br := Br ∪ {z}; // {x,z}
 Br := Br ∪ {z}; // {z}                  // x and z satisfy LC
 z.key := k;                             assert LC(z);
 Br := Br ∪ {z}; // {z}                  Br := Br \ {z}; // {x}
 z.next := y;                            assert LC(x);
 Br := Br ∪ {z}; // {z}                  Br := Br \ {x}; // {}
```

We depict the statements enforced by the well-behavedness paradigm in pink and the ghost updates written by the verification engineer in blue. Observe that the paradigm adds the impacted objects to the broken set after each mutation and allocation. Determining the impact set of a mutation is nontrivial; we show how to construct them in Section 4.6. Note also that to remove x from the broken set we must show $LC(x)$ holds (assert followed by

removal from Br). Finally, we see at the beginning of the snippet that if we show $x \notin Br$ then we can infer that $LC(x)$ holds. This follows from the meaning of the broken set.

Putting it All Together. The above program corresponds to the program $P_{\mathcal{G},Br}$ obtained from the Stage 3 reduction, consisting of ghost updates to the $\mathcal{G}$ maps and Br. Since it is well-behaved and satisfies the contract $\langle\, sortedll(x) \wedge Br = \emptyset \,\rangle\ P_{\mathcal{G},Br}\ \langle\, sortedll(x) \wedge Br = \emptyset \,\rangle$ we can conclude that it satisfies the contract $\langle\, (\forall z \notin Br.\, \rho(z)) \wedge sortedll(x) \wedge Br = \emptyset \,\rangle\ P_{\mathcal{G},Br}\ \langle\, (\forall z \notin Br.\, \rho(z)) \wedge sortedll(x) \wedge Br = \emptyset \,\rangle$. Using Propositions 4.1 and 4.2 we can project out all augmented code and conclude that the triple given in Example 4.4 with the user's original program and intrinsic specifications is valid! In this way, using FWYB we can verify programs with respect to intrinsic specifications by verifying augmented programs with respect to quantifier-free specifications. The latter can be discharged efficiently in practice using SMT solvers [11, 12] (see Section 4.4.4).

We dedicate the rest of this section to developing the general theory of well-behaved programs.

Rules for Constructing Well-Behaved Programs We define the class of well-behaved programs using a set of rules. We first introduce some notation.

We distinguish the triples over the augmented programs and quantifier-free annotations by $\{\psi_{pre}\}\, P\, \{\psi_{post}\}$, with $\{\}$ brackets rather than $\langle\,\rangle$. $\vdash \{\psi_{pre}\}\, P\, \{\psi_{post}\}$ denotes that a triple is provable. Our theory is agnostic to the underlying mechanism for proving triples correct (we use the off-the-shelf verification tool Boogie in our evaluation). However, we assume that the mechanism is sound with respect to the operational semantics. We denote that a snippet P is well-behaved by $\vdash_{\text{WB}} P$. We also denote that LC holds on a single object x by $LC(x)$.

Figure 4.4 shows the rules for writing well-behaved programs. We only explain the interesting cases here.

Mutation. Since mutations can break local conditions, we must grow the broken set. Let A be a finite set of object-type terms over x such that for any $z \notin A$, if $LC(z)$ held before the mutation, then it continues to hold after the mutation. We refer to such a set A as an *impact set* for the mutation, and we update Br after a mutation with its impact set. The impact set may not always be expressible as a finite set of terms, but this is indeed the case for all the intrinsically defined data structures we study in this chapter. We show how to construct impact sets in Section 4.6.

Allocation. Allocation does not modify the heap on any existing object. Therefore, we simply update the broken set by adding the newly created object x (see Example 4.5).

Assert LC and Remove. This rule enables shrinking the broken set once the engineer fixes the local conditions on a broken location. The snippet `assert LC(x); Br := Br\{x}` in

Example 4.5 uses this rule. Informally, the verification engineer is required to show that $LC(x)$ holds before removing x from Br.

Infer LC Outside Br. Recall that for well-behaved programs we know implicitly that $\forall x \notin Br.\, \rho(x)$ holds. This rule allows us to instantiate this implicit fact on objects that we can show lie outside the broken set. The snippet `assert x∉Br; assume LC(x)` in Example 4.5 uses this rule.

We show that the above rules are sound for the reduction in Stage 3:

Proposition 4.3. Let $[(M : P); (N_1 : Q_1) \ldots, (N_k : Q_k)]$ be a program (which can use $\mathcal{G}$ and Br) such that $\vdash_{\text{WB}} P$ and $\vdash_{\text{WB}} Q_i, 1 \leq i \leq k$. Let α and β be quantifier-free formulae over $\mathcal{F} \cup \mathcal{G}$ which can use Br. If $\{\alpha\}\ P(\overline{x}, Br,\ ret :\ \overline{y}, Br)\ \{\beta\}$ is valid, then $\langle\, (\forall z \notin Br.\, \rho(z)) \wedge \alpha \,\rangle\ P(\overline{x}, Br,\ ret\colon\ \overline{y}, Br)\ \langle\, (\forall z \notin Br.\, \rho(z)) \wedge \beta \,\rangle$ is valid.

We prove the above proposition by structural induction on the rules in Figure 4.4. We provide the proof in Section 4.5.

In the above presentation we use only one broken set for simplicity of exposition. Our general framework allows for finer-grained broken sets that can track breaks over a partition on the local conditions. For example, in Section 4.7.5 we verify deletion in an overlaid data structure consisting of a linked list and a binary search tree using two broken sets: one each for the local conditions of the two component data structures.

4.4.4 Generating Quantifier-Free Verification Conditions

We state at several points in this chapter that verifying augmented programs with quantifier-free specifications reduces to validity over combinations of quantifier-free theories. However, this is not obvious. Unlike scalar programs, quantifier-free contracts do not guarantee quantifier-free verification conditions (VCs) for heap programs. In particular, commands such as allocation and function calls pose challenges. However, we show that in our case it is indeed possible to obtain quantifier-free VCs. At a high level, our solution transforms the given heap program into a scalar program that explicitly encodes changes to the heap. Specifically, we show an encoding for the field mutation, allocation, and function call statements.

Modeling Field Mutation As described earlier, we model the monadic maps and fields as updatable maps [71]. Formally, we introduce a map M_f (also called an *array* in SMT solvers like Z3 [11]) for every field/monadic map f. We then encode the commands for field lookup and mutation as map operations, e.g., the mutation $x.f := y$ is encoded as $M_f[x] := y$.

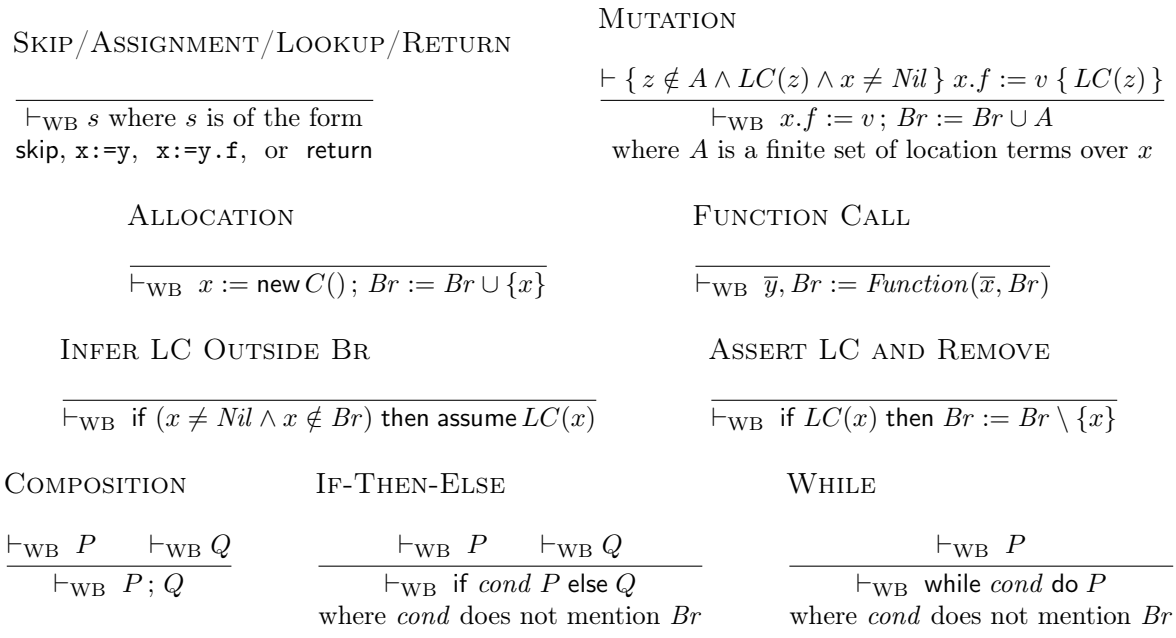


Figure 4.4: Rules for constructing well-behaved programs. Local condition formula instantiated at x is denoted by $LC(x)$. The statement (if $cond$ then S) is sugar for (if $cond$ then S else skip).

Modeling Allocation We model programs in a safe garbage-collected programming language. We introduce a ghost global variable *Alloc* to model the allocated set of objects. We then add several assume statements throughout the program. Specifically, we assume for every program parameter of type Object, the parameter itself as well as the values of the monadic maps of type Object/Set-of-Objects on the parameter are all contained in *Alloc*. In Example 4.5, we add the assumptions $x \in Alloc$ and $next(x) \neq Nil \Rightarrow next(x) \in Alloc$. If we had a monadic map *hslist* corresponding to the heaplet of the sorted list, we would also add the assumption $hslist(x) \subseteq Alloc$. Similarly, whenever an object is dereferenced on a field of type Object/Set-of-Objects in the program, we add an assumption that the resulting value is contained in *Alloc*. Note that these are quantifier-free assumptions. They can be added soundly since they are valid under the semantics of the underlying language.

We then model allocation by introducing a new object to *Alloc* and ensure that the default values of the various fields on the newly allocated object belong to *Alloc*. These constraints can be expressed using a quantifier-free formula over maps.

Modeling Heap Change Across Function Calls The main challenge in modeling function calls is to ensure the ability to do frame reasoning. To do this, we extend the

programming language with a *modified set* annotation for methods. We require the modified set to be a term of type Set-of-Objects that is constructed using object variables in the current scope and monadic maps over them. In the case of our running example (Example 4.4), we would add a monadic map *hslist* of type Set-of-Objects corresponding to the heaplet of the sorted list and annotate the program with $hslist(x)$ as the modified set. Figure 4.6 shows the full version of sorted list insertion with the modified set annotation.

Given a modified set Mod, we model changes to the heap across a function call by introducing new maps corresponding to the various fields (including monadic maps) after the call. We then add assumptions that the values of the new maps are equal to the values of the maps before the call on all locations that do not belong to the modified set Mod. Although this constrains the maps on unboundedly many objects, it can be written without quantifiers by using pointwise operators on maps [71]. Formally, for a field f modeled as a map M_f, we introduce a new map M'_f and update M_f as:

$$M_f[x] := \mathsf{ite}(x \in \mathsf{Mod},\ M'_f[x],\ M_f[x]) \tag{4.11}$$

The above update can be expressed using pointwise operators as $M_f \coloneqq \mathsf{ite}(\mathsf{Mod}, M'_f, M_f)$, where the ite operator is applied pointwise over the maps Mod, M_f, and M'_f. The value of the field f on an object x after the call will then be equal to $x.f$ before the call if x was not modified, and a *havoc*-ed value given by M'_f otherwise. Pointwise operators are supported by the generalized array theory [71] whose quantifier-free fragment is decidable.

Program verifiers like Boogie [81] offer VC generation frameworks that are amenable to the modeling described in this section. Indeed, our implementation of the IDS/FWYB methodology described in Section 4.8.1 uses Boogie.

4.5 SOUNDNESS OF FWYB

In this section we detail the proofs of soundness for the various stages of FWYB described in Section 4.4 and prove our central theoretical result on the soundness of FWYB. We first recall some lemmas regarding projection for ghost code (Section 4.3.2).

Lemma 4.2. Let c be a configuration that does not interpret ghost variables/maps. If $\hat{M}$ (projected code that does not contain ghost code) terminates starting from c, then M (which contains additional ghost code) must terminate starting from any configuration C that extends c.

The lemma says that the termination of the original user program is preserved by any augmentation with ghost code.

Lemma 4.4. Let c be a configuration that does not interpret ghost variables/maps. If $\hat{M}$ starting from c_1 reaches some c_2, then M starting from any configuration C_1 that extends c_1 must either reach $\perp$ or some C_2 that extends c_2.

This lemma says that augmentation with ghost code does not affect how the original program executes.

PROOF OF PROPOSITION 4.1

We can state the proposition simply as follows: if $\langle\, LC \wedge \psi_{pre} \,\rangle\ M\ \langle\, LC \wedge \psi_{post} \,\rangle$ is valid, then $\langle\, \exists g_1, g_2 \ldots, g_k.\, LC \wedge \psi_{pre} \,\rangle\ \hat{M}\ \langle\, \exists g_1, g_2 \ldots, g_k.\, LC \wedge \psi_{post} \,\rangle$ is valid.

Fix configurations (without ghost state) c_1, c_2 such that c_1 satisfies $\exists g_1, g_2 \ldots, g_k.\, LC \wedge \psi_{pre}$ and $\hat{M}$ starting from c_1 reaches c_2. To show that the given Hoare triple for $\hat{M}$ is valid, we must establish that c_2 is not $\perp$, and further that c_2 satisfies $\exists g_1, g_2 \ldots, g_k.\, LC \wedge \psi_{post}$.

Since $c_1 \models \exists g_1, g_2 \ldots, g_k.\, LC \wedge \psi_{pre}$, there exists a configuration (taken as a model) extending c_1, say C_1, such that $C_1 \models LC \wedge \psi_{pre}$ (we assume a reasonable semantics for second-order logic). First, using Lemma 4.2 we have that M starting from C_1 must terminate. Further, since the triple $\langle\, LC \wedge \psi_{pre} \,\rangle\ M\ \langle\, LC \wedge \psi_{post} \,\rangle$ is valid, it must be the case that M starting from C_1 reaches some C_2 such that $C_2 \neq \perp$ and $C_2 \models LC \wedge \psi_{post}$.

We now use Lemma 4.4 to conclude that $\hat{C_2} = c_2$. Since $C_2 \neq \perp$, we have that $c_2 \neq \perp$. Further, since $C_2 \models LC \wedge \psi_{post}$, we have from the semantics of the logic that $C_2 \models \exists g_1, g_2 \ldots, g_k.\, LC \wedge \psi_{post}$.

Observe that the formula $\exists g_1, g_2 \ldots, g_k.\, LC \wedge \psi_{post}$ is stated over the common vocabulary of C_2 and c_2, where the interpretations of the two configurations agree. Therefore, we can conclude that $c_2 \models \exists g_1, g_2 \ldots, g_k.\, LC \wedge \psi_{post}$. This concludes the proof. QED.

PROOF OF PROPOSITION 4.2

Recall that Proposition 4.2 is about the introduction/elimination of the broken set variable Br. It similar to Proposition 4.1, except that we consider *only* Br as ghost and no other variables/maps. Formally, if $\langle\, (\forall z \notin Br.\, \rho(z)) \wedge \alpha \wedge Br = \emptyset \,\rangle\ P_{\mathcal{G},Br}(\overline{x}, Br,\ ret\colon\ \overline{y}, Br)\ \langle\, (\forall z \notin Br.\, \rho(z)) \wedge \beta \wedge Br = \emptyset \,\rangle$ is valid then we must show that $\langle\, (\forall z.\, \rho(z)) \wedge \alpha \,\rangle\ P_{\mathcal{G}}(\overline{x},\ ret : \overline{y})\ \langle\, (\forall z.\, \rho(z)) \wedge \beta \,\rangle$ is valid.

The proof of this proposition follows the pattern of the proof of Proposition 4.1 above, with the appropriate modification to the definition of what is considered ghost. Repeating the arguments in the above proof appropriately, we obtain that if

$$\langle\, (\forall z \notin Br.\, \rho(z)) \wedge \alpha \wedge Br = \emptyset \,\rangle\ P_{\mathcal{G},Br}(\overline{x}, Br,\ ret\colon \overline{y}, Br)\ \langle\, (\forall z \notin Br.\, \rho(z)) \wedge \beta \wedge Br = \emptyset \,\rangle$$

is valid, then

$$\langle\, \exists Br.\, \big((\forall z \notin Br.\, \rho(z)) \wedge \alpha \wedge Br = \emptyset\big) \,\rangle\ P_{\mathcal{G}}(\overline{x},\ ret\colon \overline{y})\ \langle\, \exists Br.\, \big((\forall z \notin Br.\, \rho(z)) \wedge \beta \wedge Br = \emptyset\big) \,\rangle$$

is valid. This triple can be simplified to $\langle\, (\forall z.\, \rho(z)) \wedge \alpha \,\rangle\ P_{\mathcal{G}}(\overline{x},\ ret\colon\ \overline{y})\ \langle\, (\forall z.\, \rho(z)) \wedge \beta \,\rangle$, which concludes the proof.

PROOF OF PROPOSITION 4.3

Given a well-behaved program P such that $\{\alpha\}\ P\ \{\beta\}$ is valid, we must show that $\langle\, (\forall z \notin Br.\, \rho(z)) \wedge \alpha \,\rangle\ P\ \langle\, (\forall z \notin Br.\, \rho(z)) \wedge \beta \,\rangle$ is valid.

The proof proceeds by an induction on the nesting depth of method calls in a trace of the program P. We elide this level of induction here because it is routine. Importantly, given a particular execution of the program P, we must show that the claim holds, assuming it holds for all method calls occurring in the execution. We show this by structural induction on the proof of well-behavedness of P (i.e., on the rules in Figure 4.4).

There are several base cases.

Skip/Assignment/Lookup/Return There is nothing to show for skip, assignment, lookup, or return statements. These do not change the heap at all and the rule does not update Br either, therefore if $\langle\, \alpha \,\rangle$ stmt $\langle\, \beta \,\rangle$ is valid then certainly $\langle\, (\forall z \notin Br.\, \rho) \wedge \alpha \,\rangle$ stmt $\langle\, (\forall z \notin Br.\, \rho) \wedge \beta \,\rangle$ is valid.

Mutation The claim is true for the mutation rule since by the premise of the rule we update the broken set with the impact set consisting of all potential objects where local conditions may not hold.

Function Call Here we simply appeal to the induction hypothesis.

Allocation We refer to our operational semantics, which ensures that no object points to a freshly allocated object. Therefore, the allocation of an object could have only broken the local conditions on itself at most.

Infer LC Outside Br There is nothing to prove as Br is not altered.

Assert LC and Remove The claim holds for this rule by construction. If LC holds everywhere outside Br , and we know that $LC(x)$ holds, then we can conclude that LC holds everywhere outside $Br \setminus \{x\}$.

It only remains to show that the claim holds for larger well-behaved programs obtained by composing smaller well-behaved programs using sequencing, branching, or looping constructs. The proof here is trivial as the argument for sequencing is trivial (we can think of a loop as

unboundedly many sequenced compositions of the smaller well-behaved program): we can *always* compose two well-behaved programs to obtain a well-behaved program. QED.

MAIN RESULT: SOUNDNESS OF FWYB

Theorem 4.1 (FWYB Soundness)**.** Let $(\mathcal{G}, LC, \varphi)$ be an intrinsic definition with $\mathcal{G} = \{g_1, g_2 \ldots, g_l\}$. Let $[(M : P); (N_1 : Q_1) \ldots, (N_k : Q_k)]$ be an augmented program constructed using the FWYB methodology such that $\vdash_{\text{WB}} P$ and $\vdash_{\text{WB}} Q_i$, $1 \leq i \leq k$, i.e., the programs P and Q_i are well-behaved (according to the rules in Figure 4.4). Let φ, ψ_{pre}, and ψ_{post} be quantifier-free formulae that do not mention Br (but can mention the maps in $\mathcal{G}$). Finally, let $[(\hat{M} : \hat{P}); (\hat{N}_1 : \hat{Q}_1) \ldots, (\hat{N}_k : \hat{Q}_k)]$ be the projected user-level program according to Definition 4.5. Then, if the triple:

$$\{\varphi \wedge \psi_{pre} \wedge Br = \emptyset\}\ P\ \{\varphi \wedge \psi_{post} \wedge Br = \emptyset\}$$

is valid, then the triple

$$\langle\, \exists\, g_1, g_2 \ldots, g_l.\, (LC \wedge \varphi \wedge \psi_{pre})\, \rangle\ \hat{P}\ \langle\, \exists\, g_1, g_2 \ldots, g_l.\, (LC \wedge \varphi \wedge \psi_{post})\, \rangle$$

is valid (according to Definition 4.4).

Informally, the soundness theorem says that given a user-written program, if we (a) augment it with updates to ghost fields and the broken set only using the discipline for well-behaved programs, and (b) show that if the broken set is empty at the beginning of the program it will be empty at the end, then the original user-written program satisfies the intrinsic specifications on preservation of the data structure.

The proof of the theorem trivially follows from the soundness of the three stages. Let us write P as $P_{\mathcal{G},Br}$ to emphasize that the program contains ghost code that manipulates both the $\mathcal{G}$ maps and Br. We begin with the fact that $\{\varphi \wedge \psi_{pre} \wedge Br = \emptyset\}\ P_{\mathcal{G},Br}\ \{\varphi \wedge \psi_{post} \wedge Br = \emptyset\}$ is valid. Since P and its auxiliary functions are well-behaved we have from Proposition 4.3 that $\langle\, (\forall z \notin Br.\, \rho(z)) \wedge \varphi \wedge \psi_{pre}\, \rangle\ P_{\mathcal{G},Br}\ \langle\, (\forall z \notin Br.\, \rho(z)) \wedge \varphi \wedge \psi_{post}\, \rangle$ is valid.

Next, we use Proposition 4.2 to conclude that $\langle\, (\forall z.\, \rho(z)) \wedge \varphi \wedge \psi_{pre}\, \rangle\ P_{\mathcal{G}}\ \langle\, (\forall z.\, \rho(z)) \wedge \varphi \wedge \psi_{post}\, \rangle$ is valid, where $P_{\mathcal{G}}$ is the projection of $P_{\mathcal{G},Br}$ obtained by eliminating the statements that manipulate Br. Finally, we use Proposition 4.1, along with the fact that $\forall z.\, \rho(z)$ is LC and $\hat{P}_{\mathcal{G}}$ is the same as $\hat{P}$ to conclude that $\langle\, \exists\, g_1, g_2 \ldots, g_l.\, (LC \wedge \varphi \wedge \psi_{pre})\, \rangle\ \hat{P}\ \langle\, \exists\, g_1, g_2 \ldots, g_l.\, (LC \wedge \varphi \wedge \psi_{post})\, \rangle$ is valid[3]. QED.

[3]The presentation of FWYB augments the original program P with manipulations to $\mathcal{G}$ and Br in separate

4.6 PROGRAMMING IN THE FWYB METHODOLOGY

In earlier sections we develop the FWYB methodology using a top-down approach, based on a sequence of transformations that reduce the complexity of the verification problem. In this section we adopt a bottom-up approach and present the verification of insertion into a sorted list implemented in the FWYB methodology in its entirety. Our running example in Section 4.4 illustrates the key technical ideas involved in verifying the program. Here we present an end-to-end picture that showcases the verification experience in practice. To this end we develop new ideas for the design of a language paradigm based on FWYB for writing well-behaved programs. We begin with the definition of the data structure.

4.6.1 Data Structure Definition

We revise the definition of a sorted list presented earlier in Example 4.3 with a different set of monadic maps. We have the following monadic maps $\mathcal{G}$— $prev : C \to C?$, $length : C \to \mathbb{N}$, $elems : C \to Set(Int)$, $hslist : C \to Set(C)$ that model the *previous* pointer (inverse of next), length of the sorted list, the set of keys stored in it, and its heaplet (set of locations that form the sorted list) respectively. We use the length, keys, and heaplet maps to state full functional specifications of methods. The local conditions are:

$$
\begin{aligned}
\forall x.\, next(x) \neq Nil \Rightarrow (\ & key(x) \leq key(next(x)) \ \wedge\ prev(next(x)) = x \\
& \wedge\ length(x) = 1 + length(next(x)) \\
& \wedge\ elems(x) = \{key(x)\} \cup elems(next(x)) \\
& \wedge\ hslist(x) = \{x\} \uplus hslist(next(x))\ ) \qquad (\uplus\text{: disjoint union}) \\
\wedge\ prev(x) \neq Nil \Rightarrow\ & next(prev(x)) = x \\
\wedge\ next(x) = Nil \Rightarrow (\ & length(x) = 1 \wedge elems(x) = \{key(x)\} \wedge hslist(x) = \{x\}\ ) \qquad (4.12)
\end{aligned}
$$

The above definition is slightly different from the one given in Example 4.3. The *length* map replaces the *rank* map, requiring additionally that lengths of adjacent nodes differ by 1.

The *prev* map is a gadget we find useful in many intrinsic definitions. The constraints on *prev* ensure that the C-heaps satisfying the definition only contain non-merging lists. To see why this is the case, consider for the sake of contradiction distinct objects o_1, o_2, o_3 such that $next(o_1) = next(o_2) = o_3$. Then, we can see from the local conditions that we

stages. This is done for clarity of exposition. This may not be possible in general since we may write ghost code with expressions that use both the $\mathcal{G}$ maps and Br. However, we can combine the proofs of Propositions 4.1 and 4.2 to show the soundness of projecting out all ghost code in a single stage, and Theorem 4.1 continues to hold in the general case.

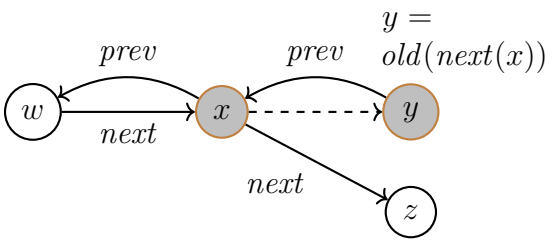


Figure 4.5: Reasoning about the set of objects broken by `x.next := z`. The dashed arrow represents the old *next* pointer before the mutation. The grey nodes denote objects where local conditions can be broken by the mutation. We see that only x and y may violate *next* and *prev* being inverses.

Table 4.1: Table of impact sets corresponding to field mutations for sorted lists. $old(t)$ refers to the value of the term t before the mutation. Terms only belong to the sets if not equal to *Nil*.

Mutated Field f	Impacted Objects A_f
$x.next$	$\{x, old(next(x))\}$
$x.key$	$\{x, prev(x)\}$
$x.prev$	$\{x, old(prev(x))\}$
$x.hslist$	$\{x, prev(x)\}$
$x.length$	$\{x, prev(x)\}$
$x.elems$	$\{x, prev(x)\}$

must simultaneously have $prev(o_3) = o_1$ and $prev(o_3) = o_2$, which is impossible. Finally, the *hslist* and *elems* maps represent the heaplet and the set of keys stored in the sorted list (respectively).

The heads of all sorted lists in the C-heap is then defined by the following correlation formula: $\varphi(y) \equiv prev(y) = Nil$

4.6.2 Constructing Provably Correct Impact Sets for Mutations

Impact Sets for Sorted List We now instantiate the rules developed in Section 4.4.3 for sorted lists. Recall that well-behaved programs must update the broken set with the impact set of a mutation. Table 4.1 captures the impact set for each field mutation. Note that the terms denoting the impacted objects belong to A_f only if they do not evaluate to *Nil*.

Let us consider the correctness of Table 4.1, focusing on the mutation of *next* as an example. Figure 4.5 illustrates the heap after the mutation `x.next := z`. We make the following key observation: the local constraints $LC(v)$ for an object v refer only to the properties of objects v, $next(v)$, and $prev(v)$ (see definition above), i.e., objects that are at most "one step" away on the heap. Therefore, the only objects that can be broken by the mutation `x.next := z` are those that are one step away from x either via an incoming or an outgoing edge via pointers *next* and *prev*. This is a general property of intrinsic definitions: *mutations cannot immediately affect objects that are far away on the heap.* [4]

In our case, we claim that the impact set contains at most x and $old(next(x))$. Here's

[4]Note that a mutation can necessitate changes to monadic maps for an unbounded number of nodes *eventually*; however, these are not necessary immediately. As we fix monadic maps on a broken object, its neighbors could get broken and need to be fixed, leading to their neighbors breaking, etc. This can lead to a ripple effect that would eventually require an unbounded number of locations to be fixed.

a proof (see Fig 4.5): Consider z such that $z \neq old(next(x))$ (as there is no real mutation otherwise). If z was not broken before the mutation, then it cannot be the case that $prev(z) = x$. Looking at the local conditions, it is clear that such a z will remain unbroken after the mutation. Now consider a w not broken before the mutation such that $next(w) = x$. Then it follows from the local conditions that there can only be one such (unbroken) w, and further $w \neq x$. w's fields are not mutated, and by examining LC, it is easy to see that w will not get broken (as $LC(v)$ does not refer to $next(next(v))$). The argument is the same for w such that $prev(x) = w$. Finally, consider a y not broken before the mutation such that $prev(y) = x$. We can then see from the local conditions that $y = old(next(x))$, which is already in the impact set.

The above argument is subtle, but we can automatically check whether impact sets declared by a verification engineer are correct. The MUTATION rule in Figure 4.4 characterizes the impact set A_{next} for mutation of the field $next$ as follows:

$$\vdash \{u \neq x \wedge u \neq next(x) \wedge LC(u) \wedge x \neq Nil\}\ x.next := z\ \{LC(u)\} \tag{4.13}$$

The above says that any location u that is not in the impact set which satisfied the local conditions before the mutation must continue to satisfy them after the mutation. Note that the validity of the above triple is decidable. In our realization of the FWYB methodology we prove our impact sets correct by encoding the triple in BOOGIE (see Section 4.8.3).

General Construction We now present our formulation for the general case. Fix a class with maps $\mathcal{F} \cup \mathcal{G} = \{f_1, f_2, \ldots f_n\}$ (includes both original and ghost fields) and an intrinsic definition $(\mathcal{G}, LC, \varphi)$ over which we prove correctness of programs. Without loss of generality, let $f_1, \ldots f_k$ for some $k \leq n$ alone correspond to pointer fields (where the range is an object); the others we assume are data fields that range over background sorts. In the sequel we assume for simplicity that $LC(x)$ only relates the fields of x with those of $f_i(x)$ for $1 \leq i \leq k$, i.e., the local conditions only constrain the fields of x with those of its neighboring objects that are "one pointer hop" away from x.

Consider a mutation $x.f \coloneqq y$ for some f in f_1 through f_n and an arbitrary y. It is clear that the only set of objects whose local condition can be impacted by this mutation are those that are one pointer hop away via an incoming or outgoing edge in the heap (seen as a directed graph with labeled edges corresponding to pointers), apart from x itself. In general there can be unboundedly many such objects, but in our work we only handle impact sets that can be expressed as a finite set of terms over x (see Section 4.4.3 under 'Rules for Constructing Well-Behaved Programs'). Note here that the impact set can be larger than the set of impacted objects as we only require that objects not belonging to the impact set

retain that LC holds on them under mutation. However, we attempt here to construct of impact sets that are as small as possible.

Following the above discussion, let us assume that the impact set consists of terms from the following set:

$$\mathit{ImpactableObjects} = \{x, f_1(x), \ldots f_k(x)\} \cup \{\mathit{old}(f(x)) \mid f \text{ is a pointer field}\} \tag{4.14}$$

The reader may be inclined to suggest here that when f is a pointer field, y (the new value of $f(x)$) may also belong to the minimal impact set. However, this is not possible in general since y is arbitrary, and in particular y can be an object in the heap that is "far away" from x, i.e., not one pointer hop away (either incoming or outgoing). The same argument applies to terms over y. Therefore, if the (minimal) impact set is at all expressible as a set of terms over the vocabulary of the mutation statement it must be a subset of the terms in the set *ImpactableObjects* defined above.

Let this subset of terms be A. We then generate the following triple to check that A is in fact an impact set:

$$\vdash \{(\textstyle\bigwedge_{t \in A} u \neq t) \wedge LC(u) \wedge x \neq \mathit{Nil}\}\ x.f := y\ \{LC(u)\} \tag{4.15}$$

The triple says that any location u that is not A which satisfied LC before the mutation must continue to satisfy it after the mutation. As discussed in the main text, this validity of this triple can be check effectively by decision procedures over quantifier-free combinations of theories that are supported by SMT solvers [11, 12].

Finally, we can compute a provably correct and minimal impact set automatically, if one exists, by considering subsets of *ImpactableObjects* in turn and checking the corresponding triple as described above. However, in our experiments we compute impact sets manually and check their correctness automatically.

4.6.3 Language Macros that Ensure Well-Behaved Programs

In Section 4.4.3 we characterized well-behaved programs as a set of syntactic rules (Figure 4.4). We can realize these restrictions using macros:

1. `Mut(x,f,v,Br)` for each $f \in \mathcal{F} \cup \mathcal{G}$, which represents the sequence of statements `x.f := v; Br := Br U` $\mathtt{A_f(x)}$. Here $\mathtt{A_f(x)}$ is the impact set corresponding to the mutation on f on x as given by the table above. This macro is used instead of `x.f := v` and automatically ensures that the impact set is added to the broken set.

2. `NewObj(x,Br)`, which represents the statements `x := new C(); Br := Br U { x }`. This macro is used instead of `x := new C()` and ensures that any newly allocated object is automatically added to the broken set.

3. `AssertLCAndRemove(x,Br)`, which represents the statements `assert LC(x); Br := Br \ { x }`. This macro is allowed anytime the engineer wants to assert that x satisfies the local condition, and then remove it from the broken set.[5]

4. `InferLCOutsideBr(x, Br)`, which represents the statements `assert (x` $\neq$ `nil` $\wedge$ `x` $\notin$ `Br); assume LC(x)`. This allows the engineer at any time to assert that x is not in the broken set and assume it satisfies the local condition.

The above macros correspond to the rules Mutation, Allocation, Assert LC and Remove, and Infer LC Outside Br respectively. Restricting to the syntactic fragment that contains the above macros and disallows mutation and allocation otherwise enforces the *programming discipline* that ensures well-behaved programs.

4.6.4 Verifying Insertion into a Sorted List

We provide the specifications and the code augmented with ghost annotations in Figure 4.6.

Specifications The precondition states that the broken set is empty at the beginning of the program. The postcondition states that the returned object r satisfies the local conditions and satisfies the correlation formula for a sorted list (i.e., $prev(r) = Nil$). However, the broken set is only empty if the input object x was the head of a sorted list, and it is $\{prev(x)\}$ otherwise. The other conjuncts express functional specifications for insertion in terms of the length, heaplet, and set of keys. We also add a 'modifies' clause which enables program verifiers for heap manipulating programs to utilize frame reasoning across function calls.

Summary The proof works at a high-level as follows: we recurse down the list, reaching the appropriate object x before which the new key must be inserted. This is the first branch in Figure 4.6, and we show the broken set at each point in the comments to the right. We create the new object z with the appropriate key and point $z.next$ to x. We then fix the local conditions on x and z. However, these fixes break the LC on $old(prev(x))$. We maintain this property up the recursion, at each point fixing LC on x and breaking it on $old(prev(x))$ in

[5]We extend our basic programming language defined in Figure 4.2 with an assert statement and give it the usual semantics (program reaches an error state if the assertion is not satisfied, but is equivalent to skip otherwise).

```
pre: Br = ∅
post: LC(r) ∧ prev(r) = Nil
    ∧ Br = ite(old(prev(x)) = Nil, ∅, {old(prev(x))})
    ∧ length(r) = old(length(x)) + 1
    ∧ elems(r) = old(elems(x)) ∪ {k}
    ∧ old(hslist(x)) ⊂ hslist(r)
modifies: hslist(x)
sorted_list_insert(x: C, k: Int, Br: Set(C))
returns r: C, Br: Set(C)
{
  InferLCOutsideBr(x, Br);
  if (x.key ≥ k) then { // k inserted before x
    NewObj(z, Br);          // {z}
    Mut(z, key, k, Br);     // {z} since z.prev = nil
    Mut(z, next, x, Br);    // {z} since z.next = nil
    Mut(z, hslist, {z} ∪ x.hslist, Br); // {z}
    Mut(z, length, 1 + x.length, Br);   // {z}
    Mut(z, keys,  {k} ∪ x.keys, Br);    // {z}
    Mut(x, prev, z, Br);    // {z, x, old(prev(x))}
    AssertLCAndRemove(z, Br);  // {x, old(prev(x))}
    AssertLCAndRemove(x, Br); // {old(prev(x))}
    r := z;
  }
  else {
  if (x.next = nil) then { // one-element list
    NewObj(z, Br);
    Mut(z, key, k, Br);
    Mut(z, next, nil, Br);
    Mut(z, hslist, {z}, Br);
    Mut(z, length, 1, Br);
    Mut(z, keys, {k}, Br);
    Mut(x, next, z, Br);
    Mut(z, prev, x, Br);
    AssertLCAndRemove(z, Br);
    Mut(x, prev, nil, Br);
    Mut(x, hslist, {x} ∪ {z}, Br);
    Mut(x, length, 2, Br);
    Mut(x, keys, {x.key} ∪ {k}, Br);
    AssertLCAndRemove(x, Br);
    r := x;
  }
  else { // recursive case
    y := x.next;
    InferLCOutsideBr(y, Br);
    tmp, Br := sorted_list_insert(y, k, Br);   // {x}
    InferLCOutsideBr(y, Br);
    if (y.prev = x) then {
      Mut(y, prev, nil, Br);    // {y, x}
    }
    Mut(x, next, tmp, Br);      // {y, x}
    AssertLCAndRemove(y, Br);   // {x}
    Mut(tmp, prev, x, Br);      // {tmp, x}
    AssertLCAndRemove(tmp, Br); // {x}
    Mut(x, hslist, {x} ∪ tmp.hslist, Br); // {x, prev(x)}
    Mut(x, length, 1 + tmp.length, Br);   // {x, prev(x)}
    Mut(x, keys, {x.key} ∪ tmp.keys, Br); // {x, prev(x)}
    Mut(x, prev, nil, Br);           // {x, old(prev(x))}
    AssertLCAndRemove(x, Br);   // {old(prev(x))}
    r := x;
  }}
}
```

Figure 4.6: Code for insertion into a sorted list written in the syntactic fragment for well-behaved programs(Section 4.6). Black lines denote code written by the user, and blue lines denote lines written by the verification engineer. The comments on the right show the state of the broken set *Br* after the statement on the corresponding line.

the process. This is shown in the last branch in the code. We eventually reach the head of the sorted list, whose *prev* in the pre state is *Nil*, and at that point the fixes do not break anything else, i.e., the broken set is empty (as desired).

The verification engineer adds ghost code to perform these fixes as shown in blue in Figure 4.6. We can also see that there are essentially as many lines of ghost code as there are lines of user code; we compare these values across our benchmark suite (see Table 4.3) and find that this is typical for many methods. However, the verification conditions for the (augmented) program are *decidable* because they can be stated using quantifier-free formulas over decidable combinations of theories including maps, map updates, and sets.

4.7 ILLUSTRATIVE DATA STRUCTURES AND VERIFICATION

Intrinsic definitions and the fix-what-you-break verification methodology are new concepts that require thinking afresh about data structures and annotating methods that operate over

them. In this section, we present several classical data structures and methods over them, and illustrate how the verification engineer can write intrinsic definitions (which maps to choose, and what the local conditions ensure) and how they can fix broken sets to prove programs correct. We begin with case studies that embody simpler ideas and gradually move towards more interesting case studies involving higher level concepts towards the end of the section. We recommend following the earlier examples as thoroughly as possible to get an understanding of how IDS and FWYB work together.

4.7.1 Right-Rotation of a BST

In this section we illustrate the IDS and FWYB methodology for trees via the verification of right rotation on a binary search tree. Such an operation is a common tree operation, and rotations are used widely in maintaining balanced search trees, such as AVL and Red-Black Trees, on which several of our benchmarks operate.

We augment the definition of binary trees discussed in Section 4.1 to include the $min : BST \to Real$ and $max : BST \to Real$ maps, which capture the minimum and maximum keys stored in the tree rooted at a node, to help enforce binary search tree properties locally. The local condition and the impact sets are shown in Figure 4.7 below.

$$
\begin{aligned}
LC \equiv & \forall x.\, min(x) \leq key(x) \leq max(x) \\
& \wedge\ (p(x) \neq Nil \Rightarrow\ l(p(x)) = x \vee r(p(x)) = x) \\
& \wedge\ (l(x) = nil \Rightarrow\ min(x) = key(x)) \\
& \wedge\ (l(x) \neq Nil \Rightarrow\ p(l(x)) = x \wedge rank(l(x)) < rank(x) \\
& \qquad\qquad \wedge\ max(l(x)) < key(x)\ \wedge\ min(x) = min(l(x))) \\
& \wedge\ (r(x) = nil \Rightarrow\ max(x) = key(x)) \\
& \wedge\ (r(x) \neq Nil \Rightarrow\ p(r(x)) = x \wedge rank(r(x)) < rank(x) \\
& \qquad\qquad \wedge\ min(r(x)) > key(x)\ \wedge\ max(x) = max(r(x)))
\end{aligned}
$$

Mutated Field f	Impacted Objects A_f
l	$\{x, old(l(x))\}$
r	$\{x, old(r(x))\}$
p	$\{x, old(p(x))\}$
key	$\{x\}$
min	$\{x, p(x)\}$
max	$\{x, p(x)\}$
$rank$	$\{x, p(x)\}$

Figure 4.7: Local Conditions and Impact Sets for BST

We first describe the gist of how the data structure is repaired and provide the fully annotated program below. Recall that in a BST right rotation, that there are two nodes x and y such that y is x's left child. After the rotation is performed, y becomes the new root of the subtree, while x becomes y's right child. Several routine updates of the monadic map p (parent) will have to be made. The most interesting update is that of the $rank : BST \rightarrow Real$ map. Since y is now the root of the affected subtree, its rank must be greater than all its children. One way of doing this is to increase y's rank to something greater than x's rank. This works if y has no parent, but not in general. To solve this issue, we use the density of the Reals to set the rank of y to $(rank(x) + rank(p(y)))/2$. Note that there are a fixed number of ghost map updates, as the various monadic maps for distant ancestors and descendants of x, y do not change (the min/max of subtrees of such nodes do not change).

We present the fully annotated program below, with comments displaying the state of the broken set Br at the corresponding point in the program.

```
pre: Br = ∅ ∧ l(x) ≠ Nil ∧ p(x) = xp
post: Br = ∅ ∧ p(ret) = xp
      ∧ l(ret) = old(l(l(x))) ∧ ret = old(l(x)) ∧ r(ret) = x
      ∧ l(r(ret)) = old(r(l(x))) ∧ r(r(ret)) = old(r(x))
bst_right_rotate(x: BST, xp: BST?, Br: Set(BST))
returns ret: BST, Br: Set(BST)
{
  LCOutsideBr(x, Br);
  if (xp ≠ nil) then {
       LCOutsideBr(xp, Br);
  }
  if (x.l ≠ nil) then {
       LCOutsideBr(x.l, Br);
  }
  if (x.l ≠ nil ∧ x.l.r ≠ nil) then {
       LCOutsideBr(x.l.r, Br);
  }
  var y := x.l;                  // {}
  Mut(x, l, y.r, Br);            // {x, y}
  if (xp ≠ nil) then {
       if (x = xp.l) then {
           Mut(xp, l, y, Br);    // {xp, x, y}
       }
       else {
           Mut(xp, r, y, Br);    // {xp, x, y}
```

Figure 4.8: Right Rotation of a Binary Search Tree

Figure 4.8 cont.

```
        }
    }
    Mut(y, r, x, Br);              // {xp, x, y, x.l} (Note: x.l == old(y.r))
    // (1): Repairing x.l
    if (x.l ≠ nil) then {
        Mut(x.l, p, x, Br);        // {xp, x, y, x.l}
    }
    // (2): Repairing x
    Mut(x, p, y, Br);              // {xp, x, y, x.l}
    Mut(x, min, if x = nil then x.k else x.l.min, Br);    // {xp, x, y, x.l}
    // (3): Repairing y
    Mut(y, p, xp, Br);             // {xp, x, y, x.l}
    Mut(y, max, x.max, Br);        // {xp, x, y, x.l}
    Mut(y, rank,
          if xp = nil then x.rank+1 else (xp.rank+x.rank)/2,
          Br);                     // {xp, x, y, x.l}
    AssertLCAndRemove(x.l, Br);    // {xp, x, y}
    AssertLCAndRemove(x, Br);      // {xp, y}
    AssertLCAndRemove(y, Br);      // {xp}
    AssertLCAndRemove(xp, Br);     // {}
    ret := y // return y
}
```

4.7.2 Reversing a Sorted List

We return to lists for another case study: reversing a sorted list. The purpose of this example is to demonstrate how the fix-what-you-break philosophy works with iteration/loops. We augment the definition of sorted linked lists from Case Study 4.6 to make sortedness optional and determined by predicates that capture sortedness in non-descending order, with $sorted : C \rightarrow Bool$, and sortedness with non-ascending order, with $rev\_sorted : C \rightarrow Bool$. We present the local conditions and impact sets in Figure 4.9, with the relevant additions for these monadic maps are marked in cyan.

We also present the method in full below. However, the gist of the method is that we pop nodes off of the front of a temporary list *cur* and push them to the front of a new reversed list *ret* repeatedly using a loop. A technique we use to verify loops using FWYB is to maintain that the broken set contains no nodes or only a finite number of nodes for which we specify how they are broken. In the case of this method, Br remains empty, as the loop maintains

$$
\begin{aligned}
LC \equiv \forall x.\, prev(x) \neq Nil \Rightarrow next(prev(x)) = x \\
\wedge\ next(x) \neq Nil \Rightarrow prev(next(x)) = x \\
\wedge\ length(x) = length(next(x)) + 1 \\
\wedge\ elems(x) = elems(next(x)) \cup \{key(x)\} \\
\wedge\ hslist(x) = hslist(next(x)) \uplus \{x\} \\
\wedge\ sorted(x) \Rightarrow key(x) \leq key(next(x)) \\
\wedge sorted(x) = sorted(next(x)) \\
\wedge\ rev\_sorted(x) \Rightarrow key(x) \geq key(next(x)) \\
\wedge rev\_sorted(x) = rev\_sorted(next(x)) \\
\wedge\ (next(x) = Nil \Rightarrow length(x) = 1 \wedge elems(x) = \{x\} \wedge hslist(x) = \{x\})
\end{aligned}
$$

Mutated Field f	Impacted Objects A_f
$next$	$\{x, old(next(x))\}$
key	$\{x, prev(x)\}$
$prev$	$\{x, old(prev(x))\}$
$length$	$\{x, prev(x)\}$
$elems$	$\{x, prev(x)\}$
$hslist$	$\{x, prev(x)\}$
$sorted$	$\{x, prev(x)\}$
rev_sorted	$\{x, prev(x)\}$

Figure 4.9: Local Conditions and Impact Sets for Sorted List

cur and *ret* as two valid lists, not modifying any other nodes. When popping x from *cur* and adding it to *ret*, in addition to repairing the new *cur* by setting its parent pointer to *Nil*, we also need to update fields such as *length* and *elems* on x, so it satisfies the relevant local conditions as the new head of the *ret* list. We present the full code below.

```
pre: Br = ∅ ∧ φ(x) ∧ sorted(x)
post: Br' = ∅ ∧ φ(ret) ∧ rev_sorted(ret) ∧ elems(ret) = old(elems(x)) ∧ hslist(ret) = old(hslist(x))
sorted_list_reverse(x: C, Br: Set(C))
returns ret: C, Br: Set(C)
{
  LCOutsideBr(x, Br);
  var cur := x;
  ret := null;
  while (cur ≠ nil)
     invariant cur ≠ Nil ⇒ LC(cur) ∧ sorted(cur) ∧ φ(cur)
```

Figure 4.10: Reversing a Sorted List

Figure 4.10 cont.

```
      invariant ret ≠ Nil ⇒ LC(ret) ∧ rev_sorted(ret) ∧ φ(ret)
      invariant cur ≠ Nil ∧ ret ≠ Nil ⇒ key(ret) ≤ key(cur)
      invariant old(elems(x)) = ite(cur = Nil, ∅, elems(cur)) ∪ ite(ret =
Nil, ∅, elems(ret))
      invariant old(hslist(x)) = ite(cur = Nil, ∅, hslist(cur)) ∪ ite(ret =
Nil, ∅, hslist(ret))
      invariant Br = ∅
      decreases ite(cur ≠ Nil, 0, length(cur))
  {
     var tmp := cur.next;                    // {}
     if (tmp ≠ nil) then {
       LCOutsideBr(tmp, Br);                 // {}
       Mut(tmp, p, nil, Br);                 // {cur, tmp}
     }
     Mut(cur, next, ret, Br);                // {cur, tmp}
     if (ret ≠ nil) then {
       Mut(ret, p, cur, Br);                 // {cur, tmp, ret}
     }
     Mut(cur, keys,
              {cur.k} ∪ (if cur.next=nil then {} else cur.next.keys),
              Br);    // {cur, tmp, ret}
     Mut(cur, hslist,
               {cur} ∪ (if cur.next=nil then {} else cur.next.hslist),
               Br);    // {cur, tmp, ret}
     if (cur.next ≠ nil ∧ (cur.key > cur.next.key ∨ ¬cur.next.sorted)) {
       Mut(cur, sorted, false, Br);          // {cur, tmp, ret}
     }
     Mut(cur, rev_sorted, true, Br);         // {cur, tmp, ret}
     AssertLCAndRemove(cur, Br);             // {tmp, ret}
     AssertLCAndRemove(ret, Br);             // {tmp}
     AssertLCAndRemove(tmp, Br);             // {}
     ret := cur;
     cur := tmp;
  }
  // The current value of ret is returned
}
```

4.7.3 Circular Lists

Our next example concerns circular lists. This example illustrates a very useful gadget we can employ in FWYB where we assert that we can reach a special node known as a *scaffolding* node. In addition to asserting properties on the arguments given to a method, one

can also assert properties on this scaffolding node. We employ these nodes in a very special way to make verification easier: namely, the designated scaffolding node remains unchanged in the heap (i.e., we never "reassign" a different node to be the scaffolding node) and it is also never deleted.

To define circular lists we start with a data structure containing a pointer $next : C \to C$ and a monadic map $prev : C \to C$. We define the scaffolding node for a list to be its last element. We refer to it using a new monadic map $last : C \to C$, the idea being that $last(x)$ for any location x points to the last item in the list. To ensure a circular list, the scaffolding node s must in turn point to another node, say h, such that $h.last$ points to s. As usual, we also define monadic maps $length : C \to Nat$ and $rev\_length : C \to Nat$ to denote the distance to the $last$ node by following $prev$ or $next$ pointers. The partial local conditions for x are as below:

$$\begin{aligned}(x = last(x) \Rightarrow last(next(x)) = x \wedge length(x) = 0 \wedge rev\_length(x) = 0)\\ \wedge\ (x \neq last(x) \Rightarrow last(next(x)) = last(x) \wedge length(x) = length(next(x)) + 1\\ \wedge rev\_length(x) = rev\_length(prev(x)) + 1)\end{aligned} \tag{4.16}$$

We consider the insertion of a node at the back of a circular list. We are given a node x such that $next(x) = last(x)$ (at the end of a cycle). We insert a newly allocated node after x, making local repairs there. Then, in a ghost loop similar to the one in Case Study 4.7.2, we make appropriate updates to the *length* and *elems* maps, which are not fully described here, following the $prev$ map until we reach $last(x)$.

For The Interested Reader We provide here for the interested reader some additional details regarding this case study. We first provide the full local conditions in Figure 4.11:

It turns out that in this benchmark (as in a few others), the local conditions are *partially* preserved on the "frontier" node (i.e., the node corresponding to the primary loop variable, here *cur*, at a given point in the loop). One of the loop invariants is the assertion of this partial satisfaction. To express this, we refer to two variants of the local condition, defined as follows. The first variant is $LC_{MinusNode}(x, n)$ for node variables x and n, which is formed from LC by replacing the clauses (C1), (C3), and (C4) in Figure 4.11 with the three clauses (C1'), (C3'), and (C4') in Figure 4.12. The second variant $LC_{Last}(x, n)$ is formed from LC by replacing the clause (C2) in Figure 4.11 with $hslist(x) = \{x, n\} \cup hslist(next(x))$. In our actual implementation we define these variants by simply copying the code of the LC and make the appropriate changes. We leave the systematic development of a language that deals with the definition of such variants to future work.

$$
\begin{aligned}
&\forall x.\, next(x) \neq Nil \ \wedge \ prev(x) \neq Nil \\
&\ \wedge \ next(prev(x)) \ = \ x \ \wedge \ prev(next(x)) \ = \ x \\
&\wedge \ last(x) = x \Rightarrow length(x) = 0 \wedge rev\_length(x) = 0 \\
&\qquad\qquad \wedge \ last(x) = last(next(x)) \\
&\qquad\qquad \wedge \ next(x) = x \Rightarrow elems(x) = \emptyset \wedge hslist(x) = \{x\} \\
&\qquad\qquad \wedge \ next(x) \neq x \Rightarrow elems(x) = elems(next(x)) && \text{(C1)} \\
&\qquad\qquad\qquad\qquad \wedge \ hslist(x) = \{x\} \cup hslist(next(x)) && \text{(C2)} \\
&\wedge \ last(x) \neq x \Rightarrow length(x) = length(next(x)) + 1 \\
&\qquad\qquad \wedge \ rev\_length(x) = rev\_length(prev(x)) + 1 \\
&\qquad\qquad \wedge \ next(x) = last(x) \Rightarrow elems(x) = \{key(x)\} \wedge hslist(x) = \{x\} \\
&\qquad\qquad \wedge \ next(x) \neq last(x) \Rightarrow elems(x) = \{key(x)\} \cup elems(next(x)) && \text{(C3)} \\
&\qquad\qquad\qquad\qquad \wedge \ hslist(x) = \{x\} \cup hslist(next(x)) && \text{(C4)} \\
&\qquad\qquad\qquad\qquad \wedge \ x \notin hslist(next(x)) \\
&\qquad\qquad \wedge \ last(x) = last(next(x)) \\
&\qquad\qquad \wedge \ last(last(x)) = last(x) \\
&\qquad\qquad \wedge \ x \in hslist(last(x)) \\
&\qquad\qquad \wedge \ prev(x) \in hslist(last(x)) \\
&\qquad\qquad \wedge \ next(x) \in hslist(last(x))
\end{aligned}
$$

Figure 4.11: Full Local Conditions for circular lists

$$
\begin{aligned}
&(elems(x) = elems(next(x)) \setminus \{key(n)\} \vee elems(x) = elems(next(x))) && \text{(C1')} \\
&(elems(x) = (key(x) \cup elems(next(x))) \setminus \{key(n)\} && \text{(C3')} \\
&\quad \vee elems(x) = (key(x) \cup elems(next(x)))) \\
&(hslist(x) = (x \cup hslist(next(x))) \setminus \{n\}) && \text{(C4')}
\end{aligned}
$$

Figure 4.12: Alterations to Figure 4.11 to form $LC_{MinusNode}$

Our development of rules and macros for well-behaved programming in Sections 4.4.3 and 4.6 presented a simplified version that reflected the common cases we encountered in our benchmarks. Adding scaffolding nodes adds complexity to this picture in that mutations are not always allowed. When we introduce scaffolding nodes, we additionally require that a precondition ϕ holds before we mutate particular fields of nodes. This helps ensure that the set of impacted objects can be described by a finite set of terms. The fields, preconditions, and impact sets for every node can be seen in Table 4.2 below.

Note that the table precludes the *hslist* field from being updated for the scaffolding node.

Table 4.2: Full Impact Sets for circular lists

Mutated Field f	Mutation Precond. ϕ	Impacted Objects A_f
$next$	$\top$	$\{x, old(next(x))\}$
key	$\top$	$\{x, prev(x)\}$
$prev$	$\top$	$\{x, old(prev(x))\}$
$last$	$last(x) \neq x \vee (last(x) = x \wedge hslist(x) = \{x\})$	$\{x, prev(x)\}$
$length$	$\top$	$\{x, prev(x)\}$
rev_length	$\top$	$\{x, next(x)\}$
$elems$	$\top$	$\{x, prev(x)\}$
$hslist$	$last(x) \neq x \vee (last(x) = x \wedge hslist(x) = \{x\})$	$\{x, prev(x)\}$

This is obviously restrictive, since it would change when we insert a node into the list. To address this, we define a separate manipulation macro `AddToLastHsList(x, n, Br)`, which, if `x` is a scaffolding node (or $last(x) = x$), adds the node `n` to the set $hslist(x)$. The precondition for invoking this macro is that $last(x) = x$, and the only object impacted by the macro is $\{x\}$. Once again, there are several nuances that arise when verifying a diverse set of intrinsic datastructures that do not fit neatly into the syntax of well-behaved programming presented in this chapter. We leave the development of a more comprehensive language paradigm to the future. We end our discussion with the full code.

```
pre: Br = ∅ ∧ next(x) = last(x)
post: Br = ∅ ∧ next(ret) = last(ret) ∧ last(ret) = old(last(x))
      ∧ elems(last(ret)) = old(elems(last(x))) ∪ {k}
      ∧ fresh(hslist(last(ret)) \ old(hslist(last(x))))
circular_list_insert_back(x: C, k: Int Br: Set(C))
returns ret: C, Br: Set(C)
{
  LCOutsideBr(x, Br);
  LCOutsideBr(x.next, Br);
  LCOutsideBr(x.prev, Br);

  var last: C = x.next;
  var node: C;
  NewObj(node, Br);
  Mut(node, key, k, Br);
  Mut(node, next, x.next, Br);
  Mut(x, next, node, Br);
```

Figure 4.13: Insertion into a Circular List

Figure 4.13 cont.

```
  AddToLastHsList(last, node, Br);
  Mut(last, prev, node, Br);
  Mut(node, prev, x, Br);
  Mut(node, length, 1, Br);
  Mut(node, rev_length, 1 + node.prev.rev_length, Br);
  Mut(node, keys, {k}, Br);
  Mut(node, hslist, {node}, Br);
  Mut(node, last, node.prev.last, Br);
  AssertLCAndRemove(node, Br);

  ghost var cur: C = x;
  label PreLoop:
  while (cur ≠ last)
     invariant cur ≠ last ⇒ Br = {cur, last}
                          ∧ LC_MinusNode(cur, node)
                          ∧ last(cur) = last
                          ∧ LC_Last(last, node)
     invariant cur = last ⇒ LC_MinusNode(cur, node)
     invariant node ∈ hslist(next(cur))
     invariant k ∈ elems(next(cur))
     invariant Unchanged@PreLoop(node)
     invariant Unchanged@PreLoop(last)
     invariant Br ⊆ {cur, last}
     decreases rev_length(cur)
  {
     if (cur.prev ≠ last) {
       LCOutsideBr(cur.prev, Br);
     }
     Mut(cur, length, cur.next.length + 1, Br);
     Mut(cur, hslist, cur.next.hslist + {node});
     Mut(cur, keys, cur.next.keys + {node.k});
     AssertLCAndRemove(cur, Br);
     cur := cur.prev;
  }

  LCOutsideBr(node, Br);
  Mut(cur, keys, cur.next.keys, Br);
  AssertLCAndRemove(cur, Br);
  AssertLCAndRemove(node, Br);
  ret := node;
}
```

4.7.4 Merging Sorted Lists

We demonstrate the ability of intrinsic definitions to handle multiple data structures at once, using the example of in-place merging of two sorted lists. The method merges the two lists by reusing the two lists' elements, which is a natural pattern for imperative code. Once again, we extend the definition of sorted lists from Case Study 4.6. We add the predicates $list1 : C \to Bool$, $list2 : C \to Bool$, and $list3 : C \to Bool$, to indicate disjoint classes of lists. The relevant differences in the local conditions and impact sets is shown in cyan Figures 4.15 and 4.14 below. Note that disjointness of the three lists is ensured by insisting that every object has at most one of the three list predicates hold.

We now describe the high level ideas involved in the proof. The recursive merge method compares the keys at the heads of the first and second sorted lists, and adds the appropriate node to the front of the third list. It turns out that we can easily update the ghost maps for this node (making it belong to the third list, and updating its parent pointer and key set) as well as updating the parent pointer of the head of the list where the node is removed from. When one of the lists is empty, we append the third list to the non-empty list using a single pointer mutation and then, using a ghost loop, we update the nodes in the appended list to make $list3$ true (this needs a loop invariant involving the broken set). The full code along with the ghost updates is available at: https://dl.acm.org/do/10.5281/zenodo.10963124

$$
\begin{aligned}
LC \equiv \forall x.&(list1(x) \lor list2(x) \lor list3(x)) \\
&\land \neg(list1(x) \land list2(x)) \land \neg(list2(x) \land list3(x)) \land \neg(list1(x) \land list3(x)) \\
&\land (list1(x) \Rightarrow (next(x) \neq Nil \Rightarrow list1(next(x)))) \\
&\land (list2(x) \Rightarrow (next(x) \neq Nil \Rightarrow list2(next(x)))) \\
&\land (list3(x) \Rightarrow (next(x) \neq Nil \Rightarrow list3(next(x)))) \\
&\land\ (prev(x) \neq Nil \Rightarrow next(prev(x)) = x) \\
&\land\ (next(x) \neq Nil \Rightarrow prev(next(x)) = x \\
&\qquad\qquad \land\ length(x) = length(next(x)) + 1 \\
&\qquad\qquad \land\ elems(x) = elems(next(x)) \cup \{key(x)\} \\
&\qquad\qquad \land\ hslist(x) = hslist(next(x)) \uplus \{x\} \quad \text{(disjoint union)} \\
&\qquad\qquad \land\ key(x) \leq key(next(x))) \\
&\land\ (next(x) = Nil \Rightarrow length(x) = 1 \land elems(x) = \{key(x)\} \land hslist(x) = \{x\})
\end{aligned}
$$

Figure 4.14: Local Conditions for disjoint sorted lists

Mutated Field f	Impacted Objects A_f
$next$	$\{x, old(next(x))\}$
key	$\{x, prev(x)\}$
$prev$	$\{x, old(prev(x))\}$
$length$	$\{x, prev(x)\}$
$elems$	$\{x, prev(x)\}$
$hslist$	$\{x, prev(x)\}$
$list1$	$\{x, prev(x)\}$
$list2$	$\{x, prev(x)\}$
$list3$	$\{x, prev(x)\}$

Figure 4.15: Impact Sets for disjoint sorted lists

4.7.5 Overlaid Data Structure of List and BST

One of the settings where intrinsic definitions shine is in defining and manipulating an *overlaid data structure* that overlays a linked list and a BST. The list and tree share the same locations, and the *next* pointer threads them into a linked list while the *left, right* pointers on them defines a BST. Such structures are often used in systems code (such as the Linux kernel) to save space [82]. For example, I/O schedulers use an overlaid structure as above, where the list/queue stores requests in FIFO order while the bst enables faster searching according requests with respect to a key. While there has been work in verification of memory safety of such structures [82], we study the preservation of such data structures.

The intrinsic definition of such an overlaid data structure is pleasantly *compositional*. We simply take the union of the monadic maps and conjoin the local conditions corresponding to linked lists and BSTs!. Of course, we must ensure that the two structures contain the same set of locations. We introduce a monadic map bst_root that maps a node to the root of the BST it belongs to, and a map $list\_head$ that maps a node to the head of the list it belongs to. We then demand that all locations in a list have the same bst_root and all locations in a tree have the same $list\_head$, using local conditions. We also use monadic maps corresponding to bst-heaplets and list-heaplets (i.e., locations that belong to the tree under the node or the list from that node, respectively). We define a correlation predicate $Valid$ that relates the head h of the list and root r of the tree by demanding that the bst-root of h is r and the list-head of r is h, and furthermore, the list-heaplet of h and tree-heaplet of r are equal. Formally:

$$Valid \equiv\ bst\_root(h) = r \wedge list\_root(r) = h \wedge\ list\_heaplet(h) = bst\_heaplet(r) \tag{4.17}$$

We prove certain methods manipulating this overlaid structure correct (such as deleting the first element of the list and removing it both from the list as well as the BST). Except those fields whose mutation breaks local conditions for both structures, the ghost annotations are mostly compositional— we repair maps for the BST component in the same way we fix them for stand-alone BSTs and fix them for the list component in the same way we would for stand-alone lists. In fact, we maintain two broken sets, one for BST and one for list, as updating a pointer for BST often does not break the local property for lists, and vice versa.

Limitations In modeling the data structures above, we crucially used the fact that for any location, there is at most one location (or a bounded number of locations) that has a field pointing to this location. We used this fact to define an inverse pointer (*prev* or *parent*/*p*), which allows us to capture the impact set when a location's fields are mutated. Consequently, we do not know how to model structures where locations can have unbounded indegree. We could model these inverse pointers using a sequence/array of pointers, but verification may get more challenging. Data structures with unbounded outdegree can however be modeled using just a linked-list of pointers and hence seen as a structure with bounded outdegree.

4.8 IMPLEMENTATION AND EVALUATION

4.8.1 Implementation Strategy of IDS and FWYB in Boogie

We implement intrinsically defined data structures and FWYB verification in the program verifier Boogie [83]. Boogie is an intermediate verification language which supports systematic generation of verification conditions that are checked using SMT solvers. Our benchmarks are available at: https://dl.acm.org/do/10.5281/zenodo.10963124

We choose Boogie for our realization of the FWYB framework as it is a low-level verification condition generator, which allows us to reasonably expect that scalar programs with quantifier-free specifications, annotations, and invariants, and given our careful modeling of the heap and its modification across function calls (Section 4.4.4), reduces to quantifier-free verification conditions that fall into decidable logics. We further cross-check that our encodings indeed generate decidable queries by checking the generated SMT files. Furthermore, a plethora of higher-level languages compile to Boogie (e.g., VCC and Havoc for C [72, 73], Dafny [9] with compilation to .NET, Civl for concurrent programs [74], Move for smart contracts [75], etc.). Implementing a technique in Boogie hence shows a pathway for implementing IDS and FWYB for higher-level languages as well.

Modeling Fix-What-You-Break Verification in Boogie We model heaps in BOOGIE by having a sort *Loc* of locations and modeling pointers as maps from *Loc* to sorts. We implement monadic maps also as maps from locations to field values. We implement our benchmarks using the *macros* for well-behaved programming defined in Section 4.6. We implement allocation with an *Alloc* set and heap change across function calls using parameterized map updates as described in Section 4.4.4.

We ensure that the VCs generated by BOOGIE fall into decidable fragments, and there are several components that ensure this. First, note that all specifications (contracts and invariants) are quantifier-free. Second, pure functions (used to implement local conditions) are typically encoded using quantification, but we ensure BOOGIE inlines them to avoid quantification. Third, heap updates that are the effect of procedures and set operations for set-valued monadic maps are modeled using parameterized map updates [71], which BOOGIE supports natively. Finally, we cross-check that the generated SMT query is quantifier-free and decidable by checking the absence of statements that introduce quantified reasoning, including `exists`, `forall`, and `lambda`.

4.8.2 Benchmarks

We evaluate our technique on a variety of data structures and methods that manipulate them. Our benchmark suite consists of data structure manipulation methods for a variety of different list and tree data structures, including sorted lists, circular lists, binary search trees, and balanced binary search trees such as Red-Black trees and AVL trees. Methods include core functionality such as search, insertion and deletion. The suite includes an *overlaid* data structure that overlays a binary search tree and a linked list, implementing methods needed by a simplified version of the Linux deadline IO scheduler [82]. The contracts for these functions are complete functional specifications that not only ask for maintenance of the data structure, but correctness properties involving the returned values, the keys stored in the container, and the heaplet of the data structure.

4.8.3 Evaluation

We first evaluate the following two research questions:

RQ1: Can the data structures be expressed using IDS, and can the FWYB methodology for methods on these structures be expressed in BOOGIE?

RQ2: Is BOOGIE with decidable verification condition generation dispatched to SMT solvers effective in verifying these methods?

Table 4.3: Implementation and verification of benchmarks in BOOGIE. The columns give data structure, size of local conditions for capturing the datatructure as number of conjuncts, method, lines of executable code in the method, lines of specification (pre/post), lines of ghost code annotations (invariants/monadic map updates), and verification time in seconds.

Data Structure	*LC* Size	Method	LOC+Spec +Ann	T(s)	Method	LOC+Spec +Ann	T(s)
Linked List	8	Append	4+11+10	2.0	Insert-Back	6+13+12	2.0
		Copy-All	7+8+9	2.0	Insert-Front	3+13+7	2.0
		Delete-All	10+9+16	2.0	Insert	9+13+23	2.0
		Find	4+4+2	1.9	Reverse	6+8+18	2.1
Sorted List	14	Delete-All	10+9+16	2.1	Merge	11+9+20	2.1
		Find	4+4+2	1.9	Reverse	5+14+22	2.1
		Insert	9+16+27	2.1			
Sorted List-Alt	20	Concatenate	6+10+13	2.2	Find-Last	5+10+9	2.0
Circular List	27	Insert-Front	4+12+41	2.3	Delete-Front	3+12+39	2.4
		Insert-Back	5+14+45	2.4	Delete-Back	3+13+55	2.4
BST	35	Find	4+3+5	2.0	Delete	10+13+30	2.8
		Insert	9+12+37	2.7	Remove-Root	17+15+47	3.8
Treap	37	Find	4+3+5	2.0	Delete	10+13+30	3.1
		Insert	19+12+74	10.2	Remove-Root	24+15+74	5.4
AVL Tree	45	Insert	12+12+36	5.1	Find-Min	5+5+8	2.1
		Delete	43+13+62	5.3	Balance	40+17+95	5.0
RB Tree	48	Insert	76+12+203	74.1	Del-L-Fixup	33+20+93	8.9
		Delete	56+13+76	5.8	Del-R-Fixup	33+20+93	7.4
		Find-Min	5+5+8	2.1			
BST-Scaffolding	59	Delete-Inside	1+24+51	4.8	Remove-Root	44+31+61	10.2
Overlaid Queue (SLL+BST)	72	Move-Request	4+10+8	2.9	Delete-Inside	1+29+55	4.9
		List-Remove-First	5+13+10	2.7	Remove-Root	44+36+65	15.0

As we have articulated earlier, intrinsic definitions and monadic map updates require a new way of thinking about programs and repairs. We implement the specifications using monadic maps and local conditions, and the benchmarks using the well-behavedness macros and ghost updates. We were able to express all data structures and FWYB annotations for the methods on these structures for our benchmarks in BOOGIE (RQ1). Importantly, we were able to write quantifier-free modular contracts for the auxiliary methods and loop invariants using the monadic maps and strengthening the contracts using quantifier-free assertions on broken sets (which may not be empty for auxiliary methods). We do not prove termination for these methods except for ghost loops and ghost recursive procedures (termination for the latter is required for soundness).

Our annotation measures and verification results are detailed in the table in Table 4.3, for 42 methods across 10 data structure definitions. These measurements were taken from a machine with an Intel™ Core i5-4460 processor at 3.20 GHz. We found the verification

performance excellent overall (RQ2): all the methods verify in under 2 minutes, and all but four verify in under 10 seconds. We used the option that sets the maximum number of VC splits to 8 in Boogie. The times reported for each method are the sum of times taken for the following steps: verifying that the impact sets are correct (<3s for all data structures), generating verification conditions with Boogie, injecting parametric update implementations, and solving the SMT queries.

Notice that the lines of ghost code written is nontrivial, but these are typically simple, involving programmatically repairing monadic maps and manipulating broken sets. In fact, a large fraction ($\sim$ 60%) of ghost updates in our benchmarks were *definitional updates* that simply update a field according to its definition in the local condition. An example is updating $x.length$ to $x.next.length + 1$ for lists. We believe that the annotation burden can be significantly lowered in future work by automating such updates. More importantly, note that none of the programs required further annotations like instantiations, triggers, inductive lemmas, etc. in order to prove them correct.

RQ3: What is the performance impact of generating decidable VCs?

To study this, we implemented the entire benchmark suite described in Table 4.3 in Dafny, a higher-level programming language that uses Boogie to perform its verification. We defined the datastructures and the repaired the ghost maps identically to the Boogie version.

Even though our annotations are all quantifier-free, Dafny generates Boogie code where several aspects of the language, in particular allocation and heap change across function calls, are modeled using *quantifiers*, resulting in quantified queries to SMT solvers. The scatter plot on the right shows the performance of Boogie and Dafny on the benchmarks. The plot clearly strongly suggests that even though Dafny is able to prove the FWYB-annotated programs correct, using decidable verification conditions results in much better performance. We hence believe that implementing program verifiers (such as Dafny) that exploit the fact that FWYB annotations can be compiled to annotations in Boogie that result in decidable VCs is a promising future direction to achieve faster high-level IDS+FWYB frameworks.

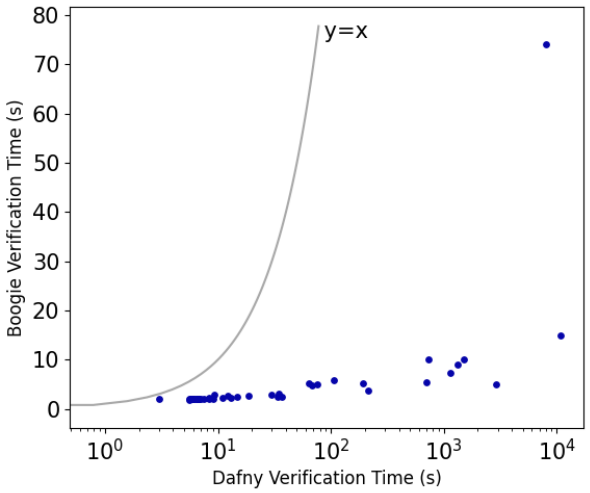


Figure 4.16: Performance comparison between using Boogie and Dafny

ADDENDUM: OPERATIONAL SEMANTICS

We give the formal operational semantics for programs in our language (Figure 4.2) in Figure 4.17 below and briefly describe some important design decisions.

$$\bot \xrightarrow{*} \bot$$

$$(s,O,I) \xrightarrow{\mathsf{skip}} (s,O,I)$$

$$(s,O,I) \xrightarrow{x := Nil} (s[x \mapsto Nil],O,I)$$

$$(s,O,I) \xrightarrow{x := y} (s[x \mapsto s(y)],O,I)$$

$$(s,O,I) \xrightarrow{v := be} (s[v \mapsto e],O,I) \qquad \text{where } be \text{ interprets to } e \text{ according to } s \text{ and } I$$

$$(s,O,I) \xrightarrow{y := x.f} (s[y \mapsto I(f,s(x))],O,I) \quad \text{if } (f,s(x)) \in dom(I) \qquad (\text{similarly for } v := x.d)$$

$$(s,O,I) \xrightarrow{y := x.f} \bot \quad \text{if } (f,s(x)) \notin dom(I) \qquad (\text{similarly for } v := x.d)$$

$$(s,O,I) \xrightarrow{x.f := y} (s,O,I[(f,s(x)) \mapsto s(y)]) \quad \text{if } (f,s(x)) \in dom(I) \quad (\text{similarly for } x.d := v)$$

$$(s,O,I) \xrightarrow{x.f := y} \bot \quad \text{if } (f,s(x)) \notin dom(I) \qquad (\text{similarly for } x.d := v)$$

$$(s,O,I) \xrightarrow{x := \mathsf{new}\ C()} (s[x \mapsto o],O \cup \{o\},I[(f,o) \mapsto default_f])$$
for some $o \in \mathbb{N}$ such that $o \notin O$

$$(s,O,I) \xrightarrow{\overline{r} := Function(\overline{t})} (s[\overline{r} \mapsto s'(\overline{n})],O',I') \quad \text{if } (\emptyset[\overline{m} \mapsto s(\overline{t})],O,I) \xrightarrow{Q(\overline{m},\, ret:\, \overline{n})} (s',O',I')$$
where $Q(\overline{m},\ ret : \overline{n})$ is the code of the method *Function*,
with $\overline{m}$ and $\overline{n}$ being the formal input and output parameters for Q

$$(s,O,I) \xrightarrow{\mathsf{assume}\ cond} (s,O,I) \quad \text{if } cond \text{ interprets to } True \text{ according to } s \text{ and } I$$

$$(s,O,I) \xrightarrow{P_1\,;\,P_2} (s'',O'',I'') \quad \text{if } (s,O,I) \xrightarrow{P_1} (s',O',I')$$
and $(s',O',I') \xrightarrow{P_2} (s'',O'',I'')$ for some (s',O',I')

$$(s,O,I) \xrightarrow{\mathsf{if}\ cond\ \mathsf{then}\ P_1\ \mathsf{else}\ P_2} (s',O',I') \quad \text{if } (s,O,I) \xrightarrow{\mathsf{assume}\ cond\,;\,P_1} (s',O',I')$$

$$(s,O,I) \xrightarrow{\mathsf{if}\ cond\ \mathsf{then}\ P_1\ \mathsf{else}\ P_2} (s',O',I') \quad \text{if } (s,O,I) \xrightarrow{\mathsf{assume}\ \neg cond\,;\,P_2} (s',O',I')$$

$$(s,O,I) \xrightarrow{\mathsf{while\ cond\ do}\ P} (s',O',I') \quad \text{if } (s,O,I) \xrightarrow{\mathsf{assume}\ cond\,;\,P\,;\,\mathsf{while\ cond\ do}\ P} (s',O',I')$$

$$(s,O,I) \xrightarrow{\mathsf{while\ cond\ do}\ P} (s,O,I) \quad \text{if } (s,O,I) \xrightarrow{\mathsf{assume}\ \neg cond} (s,O,I)$$

Figure 4.17: Operational Semantics

Configurations are of the form (s,O,I) where $O \subset_{finite} \mathbb{N}$ represents the set of allocated objects, s represents the store and interprets program variables, and I represents the heap

and interprets mutable fields in $\mathcal{F}$—including ghost fields $\mathcal{G}$ when they are used— on O (interpretations are total). Although formally s and I are a family of functions indexed by the sorts of the variables (resp. signatures of the maps), we abuse notation and use $s(x)$ to denote the interpretation of a variable x, and similarly $I(f, o)$ to denote the value of the field f on the object o in the configuration. We add a sink state $\perp$ to model error.

Our language is safe, (i.e., allocated locations cannot point to un-allocated locations) and garbage-collected. The operational semantics is the usual one for such programs. Figure 4.17 presents a simplified operational semantics without considering return statements. The full semantics adds a marker to signify completion of a procedure. Procedures can only end after return statements (we syntactically disallow statements after a return) or at the end of a program.

The rules for assignments, skip, sequencing, conditionals, and loops are trivial. Dereferencing a variable that does not point to an object (i.e., is Nil) leads to the error state $\perp$. Allocation ensures memory safety by assigning the value of a field f on a newly allocated object to a constant $default_f$. For pointer fields this value is Nil. Finally, we define the operational semantics for function calls using summaries.

Chapter 5: Synthesizing Axiomatizations using Logic Learning

In this final work of the thesis we turn our sights to broader avenues for the automation of creativity gaps through learning. Axioms and inference rules form the foundation of deductive systems and are crucial in the study of reasoning with logics over structures. Historically, axiomatizations have been discovered manually with much expertise and effort. In this chapter we show the feasibility of using learning-based synthesis techniques similar to the ones developed earlier in Chapter 3 to discover axiomatizations for different classes of structures, and in some contexts, automatically prove their completeness. For evaluation, we apply our technique to find axioms for (1) classes of frames in modal logic characterized in first-order logic and (2) the class of language models with regular operations.

5.1 INTRODUCTION

Several applications in programming languages, formal verification, and associated fields benefit from deductive reasoning in logics. Depending on the application domain, we need to reason with particular logics over particular classes of structures[1]. The logics are often specialized, and the classes of structures are those relevant to the problem domain, which may be known intuitively or defined precisely. Examples abound including modal logics to reason about transition systems (or Kripke structures) [84], temporal logics to reason about sequential or branching behavior of systems [85, 86], logics to reason with algebraic data types in functional programming languages [87], logics to reason with Kleene algebras that can model packet movement in networks [88], and separation logic to reason with pointer-based data structures in imperative programs [89].

The foundations of deductive reasoning for a logic $\mathcal{L}$ over a particular class of structures $\mathcal{C}$ lies in the axiomatic method, which utilizes general deductive mechanisms for deriving logical truths in $\mathcal{L}$ applied to a set of axioms $A_{\mathcal{C}}$ that captures fundamental properties of $\mathcal{C}$. Whether reasoning over one particular intended structure (e.g., arithmetic or an algebraic data type) or an intended class of structures (e.g., lists or other data structures in a heap, mathematical groups, or Kleene algebras), the axiomatic method involves finding a basic set of properties that logically characterize the structures. Other properties of the structures are then *logically entailed* by the basic axioms, and entailment can be mechanized using a variety of reasoning methods for the logic, including proof systems and algorithmic procedures.

The material in this chapter is entirely reproduced from the publication in cited as [16] co-authored by the author of this thesis, with minor changes.

[1]Also known as *models*. We use *structure* and *model* interchangeably.

In special circumstances, *all* properties common to structures in $\mathcal{C}$ are logically entailed by a finite or recursive set of axioms, in which case the axiom system is said to be *complete*. A complete set of axioms coupled with a powerful enough deductive system then yields a complete proof system for the class of structures $\mathcal{C}$.

In recent years there has been tremendous progress in two fields that are relevant to our quest— (1) automated validity checking for different logics and (2) program synthesis. Significant strides have been made in identifying (semi-)decidable logical theories and in building automatic and efficient procedures for validity [11, 90]. There has also been significant progress in program and expression synthesis, where the goal is to synthesize a program or logical expression that satisfies a given set of constraints (e.g., see [91]).

Formulating axiomatizations is a difficult task typically done by humans, and most typically by researchers with considerable prowess in logic. Armed with current automated reasoning and program synthesis techniques, we ask the following (perhaps audacious) question:

Can computers help us find axiomatizations?

Intuitively, both logical reasoning and expression synthesis are useful for this problem—axioms are logical expressions that we want to synthesize, and reasoning is needed to prove that an axiom is valid over a class of structures, as well as for other tasks, e.g., checking if an axiom is implied by another set of axioms.

In this chapter we formulate the problem of axiomatizing a class of structures in a logic. This is a *model-theoretic* formulation of axiomatization that is independent of proof systems. Given a logic $\mathcal{L}$ and a subclass $\mathcal{C}$ of structures within a larger class $\mathcal{S}$, an axiomatization we aim to find is a finite set A of sentences in $\mathcal{L}$ that (a) hold on all structures in $\mathcal{C}$, (b) are *nontrivial* in that they do not hold over all structures in $\mathcal{S}$, and (c) are mutually independent in terms of semantic entailment with respect to $\mathcal{S}$. Such an axiomatization A is said to be *complete* if it semantically entails all sentences expressible in $\mathcal{L}$ that hold over $\mathcal{C}$. That is, for any sentence $\varphi \in \mathcal{L}$ that is true for every structure in $\mathcal{C}$, we have that every structure in $\mathcal{S}$ satisfying the axioms A also satisfies φ, i.e. $A \models \varphi$.

5.1.1 Learning-based Axiom Synthesis (LAS) Framework

The main contribution of this chapter is a framework and core algorithm for solving the axiom synthesis problem, consisting of logical reasoning and expression synthesis components. We propose the *Learning-based Axiom Synthesis (LAS)* framework to facilitate synthesis of sound/complete axiomatizations. For a particular logic $\mathcal{L}$ and a subclass $\mathcal{C}$ of a class of structures $\mathcal{S}$, this framework calls for implementing and combining the following components:

- VC: a procedure that checks the validity of formulae in $\mathcal{L}$ over the class $\mathcal{C}$
- Cex: a procedure that generates a *counterexample* to rule out a formula that is *invalid* over $\mathcal{C}$
- VS: a procedure that checks entailment of formulae in $\mathcal{L}$ over the class $\mathcal{S}$
- Learner: a procedure that proposes axioms in $\mathcal{L}$ using counterexamples reported by Cex

Given $\mathcal{C}$ and the logic $\mathcal{L}$, whose semantics over $\mathcal{S}$ is known, the LAS framework can be instantiated to axiomatize $\mathcal{C}$ by implementing the procedures VC, Cex, VS, and Learner. The framework specifies a core algorithm that combines these procedures to find a sound axiomatization for $\mathcal{C}$.

Note that, because individual structures may have infinite domains and the class $\mathcal{C}$ may be infinite as well, it does not make sense to think of $\mathcal{C}$ as an input. Instead, the framework requires reasoning and counterexample generation procedures to be implemented for $\mathcal{C}$ and $\mathcal{S}$. The notion of counterexamples is rather nuanced. If structures in $\mathcal{C}$ are finite (even though the set $\mathcal{C}$ is not), we may expect Cex to directly provide structures as counterexamples for the soundness of proposed axioms. When structures are infinite, however, Cex may generate what we call *pseudo-models*, which are finite objects that rule out unsound formulae and indicate the existence of a counterexample structure (which may be infinite). The notion of pseudo-models depends on the specific setting, and we elaborate on this later.

Each procedure required by the framework serves a specific purpose for axiom synthesis. First, when Learner proposes a candidate axiom in $\mathcal{L}$, we can determine whether it is *sound*, i.e. valid over $\mathcal{C}$, by using the procedure VC. Second, having synthesized a set of sound axioms A, we can check for redundancies, i.e. whether any axioms in A are logically entailed by the others, by using the procedure VS. Third, Learner is a generic expression synthesis procedure that is aware of the universal class of structures $\mathcal{S}$ and the semantics of the logic $\mathcal{L}$ over $\mathcal{S}$, but it is agnostic to the class $\mathcal{C}$. To efficiently search for axioms, it hence needs counterexamples from $\mathcal{C}$. Finally, each time Learner proposes a candidate that is unsound, the procedure Cex generates a pseudo-model that witnesses that the axiom is false, and Learner uses it to prune the search space for new axioms.

Since we would like to deal with expressive logics and classes of structures, we cannot avoid that the procedures may be incomplete and non-terminating in various ways. In particular, in many cases the validity procedures VC and VS will be non-terminating, though of course they must be sound: if they terminate with an answer then that answer is correct.

The LAS framework can be seen as a kind of counterexample-guided inductive synthesis (CEGIS) framework for synthesizing axiom systems [55]. In typical synthesis problems the

specification for synthesized expressions is formalized as a logical constraint. However, in axiom synthesis, axioms are a sound and independent set of statements for a class $\mathcal{C}$, and it is not possible to capture this requirement as a logical constraint (e.g., as in SyGuS [91]). However, by implementing the *teacher* using the components VC, Cex, and VS, LAS facilitates a CEGIS algorithm using a learner that inductively learns expressions from counterexamples. The LAS framework and the notion of pseudo-models to learn classes with infinite models are contributions of our work.

5.1.2 Complete Axiomatizations

The framework suggested above does not address the problem of finding *complete* axiomatizations. First, we need to define precisely what we mean by completeness, and there are multiple natural notions here. We could say that a set of sound axioms A is complete if the subclass of $\mathcal{S}$ that satisfies A is precisely $\mathcal{C}$. This is equivalent to requiring that every structure in $\mathcal{S}$ that is not in $\mathcal{C}$ violates at least one axiom in A.

However, there is another natural and common notion: A is complete if every property φ (expressible in $\mathcal{L}$) that holds over $\mathcal{C}$ is semantically entailed by A. Note the difference: this does not demand that $\mathcal{C}$ is captured precisely, but rather that it is captured *up to properties expressible in* $\mathcal{L}$. In other words, if we take all structures in $\mathcal{S}$ that satisfy the axioms A, we should get the class $\mathcal{C}' \supseteq \mathcal{C}$ of all structures that satisfy every property that holds over $\mathcal{C}$. If $\mathcal{C}' = \mathcal{C}$ then the above two notions coincide. (For readers familiar with the notion of *elementary class* [87]: this is the same notion except that instead of first-order properties it involves properties expressible in $\mathcal{L}$.)

The LAS framework involves an optional completeness checker CC. Given axioms A, the procedure CC checks whether there is a structure M that satisfies A but does not satisfy all properties common to $\mathcal{C}$. Completeness checks are extremely hard to automate, in part because $\mathcal{C}'$ may be a complex class that is hard to understand. However, when $\mathcal{C}' = \mathcal{C}$, then CC need only check whether there is a structure in $\mathcal{S} \setminus \mathcal{C}$ that satisfies the axioms, which is feasible in some cases.

5.1.3 Instantiations of LAS

The second contribution of this chapter is an instantiation of LAS in two different settings:

- **Modal Logics** [92, 93]. The subfield of modal logic called correspondence theory characterizes classes of Kripke structures (transition systems with propositional valuations

for each state) using axioms in modal logic. We instantiate LAS in this setting to synthesize axioms for 17 different classes of structures from the literature.

- **Languages with Kleene star** [94, 95, 96]. We instantiate LAS to synthesize equational axioms for a class of structures consisting of arbitrary word languages over finite alphabets with the operations of concatenation, union, and Kleene star.

While the framework remains the same, both of the settings above have nuances. Modal logic axiomatizations for classes of Kripke structures is a natural example for our framework, especially since there are a large number of classes to study systematically. It presents unique challenges, however— axioms in modal logic have semantics that involves universal quantification over the propositional valuations, which requires us to handle *second-order* reasoning. Completeness checking is also highly nontrivial, but we are able to synthesize complete axiomatizations for all 17 classes (and with 14 of them automatically proven complete). Axiomatizing word languages with operations poses a different set of challenges— though our instantiation of LAS manages to effectively discover and reason with equational axioms, it is known that any complete finite axiomatization must go beyond equations [97], e.g., to conditional equations, which substantially increases the complexity of reasoning procedures. In fact, the equational axioms that our tool finds are stronger than the purely equational axioms in standard axiomatizations for Kleene algebras.

Despite the nuances and complexity described above, the tool we develop for these instantiations effectively discovers sound (and in some cases complete) axiomatizations in reasonable time, and furthermore, the axioms resemble those found by human researchers, after accounting for simple, semantically-equivalent rearrangements. We believe that the relative success of our framework in two different settings shows its promise for automating axiomatizations. As logical reasoning and expression synthesis technologies improve, the framework, being parameterized over these, will also become more effective.

In summary, the contributions of this chapter are: (a) a model-theoretic formulation of the axiom synthesis problem, (b) the Learning-based Axiom Synthesis framework that facilitates a CEGIS-style algorithm for automating axiom synthesis, (c) the notion of pseudo-models which can be used as counterexamples for axiom synthesis, and (d) instantiations of the LAS framework in two domains that argue its efficacy, modal logic (17 classes of structures) and equational axioms for Kleene algebras.

Outline This chapter is structured as follows. In Section 5.2, we show how LAS works for an illustrative example, namely, axiomatizing *equivalence relations* from structures that encode partitions. Section 5.3 presents our model-theoretic formulation of the axiom synthesis

problem, with discussions on sound and complete axiomatizations. We present the LAS framework in Section 5.4 and define the components needed for reasoning, counterexample generation, and synthesis. In Sections 5.5 and 5.6, we present instantiations of the LAS framework for the settings of modal logic (correspondence theory) and languages with regular operations. The nuances in each setting, realizations of components, implementation details, evaluation, and the axiomatizations found by our tool are reported in the corresponding sections.

5.2 EXAMPLE: AXIOMATIZING EQUIVALENCE RELATIONS

We begin by illustrating LAS in a very simple setting. We want to synthesize the axioms of the relations that characterize equi-membership in partitions of sets (which we familiarly call equivalence relations, with axioms for reflexivity, symmetry, and transitivity). This example covers many aspects of our framework (except completeness). It illustrates how the axiomatization problem is defined, how to build the components for reasoning, and also the axiomatization produced by our tool.

The *equi-membership* relation for a partition of a set is a binary relation that relates two elements precisely when they occupy the same partition cell. Let us model a partition P of a set X by a function $f : X \to X$, where for each element $c \in f(X) \subseteq X$ we have one cell $P_c \coloneqq \{x \in X \,:\, f(x) = c\}$. In other words, two elements belong to the same set in a partition if and only if f maps them to the same elements. Intuitively, our goal is to find a (preferably small) set of axioms expressed as first-order logic sentences that captures the theory of relations R defined using such partitions. More precisely, we want an axiomatization that uses sentences referring only to R (and not to "f" or "$=$") which are true in *every* structure that satisfies:

$$\psi \coloneqq \forall x. \forall y. \, (R(x, y) \leftrightarrow f(x) = f(y)) \tag{5.1}$$

The target class $\mathcal{C}$ hence consists of structures that satisfy formula 5.1, each being a set of elements together with interpretations for f, $=$, and R, with $=$ possessing the usual meaning. Each structure in $\mathcal{C}$ therefore interprets R as the equi-membership relation defined by the partition that puts two elements x and y in the same cell if $f(x) = f(y)$.

Suppose we have an axiom synthesizer that proposes candidate axioms. Leaving aside how to obtain the synthesizer, we must at least be able to determine whether a candidate axiom is truly an axiom, i.e., whether it is true in each structure in $\mathcal{C}$. In this example, the target class is characterized in first-order logic by ψ above. Checking that a candidate formula φ

is in fact a true axiom requires checking that every structure $M \in \mathcal{C}$ makes φ true, which corresponds to the validity of

$$\psi \to \varphi \tag{5.2}$$

Validity in the class $\mathcal{C}$, which is equivalent to the first-order validity of 5.2 for this example, corresponds to what we call *soundness* and is checked in our framework by a component we call VC. We refer to formulae that are true for all structures in $\mathcal{C}$ as *sound axioms*. As an example, suppose the synthesizer proposes the candidate

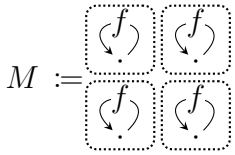
$$\varphi \coloneqq \forall x. \forall y.\, R(x, y) \tag{5.3}$$

This is not a sound axiom, because any partition with at least two cells satisfies ψ (5.1) but not φ, and therefore does not satisfy 5.2. In this setting, we can implement VC using any semi-decision procedure for first-order logic validity. Note, however, that we cannot hope to mechanically prove that a candidate axiom is *not* sound. We must be content with admitting axioms that we can prove sound and discarding those that we cannot prove.

Can we do better than merely filtering candidate axioms for soundness? A key idea in synthesis is the use of counterexamples to guide search. If we can find *counterexamples that witness unsoundness*, then we can use them to rule out many other unsound candidates. For any unsound φ like the one above, there must be a structure $M \in \mathcal{C}$ such that $M \not\models \varphi$ (though it may be difficult or impossible to automatically find one, and it may not be finitely representable). If we can indeed effectively find counterexamples, then we can maintain a set of counterexample structures in a counterexample-guided synthesis loop. In each iteration, we can query the synthesizer for candidate axioms that are true in all counterexample structures found up to that point. Candidate axioms are then subjected to a soundness check, and if proven, added to a growing set of axioms. Continuing with our example above, suppose we begin with an empty set of counterexample structures and the first candidate axiom is $\varphi \coloneqq \forall x. \forall y.\, R(x, y)$. After failing to prove it sound, we may generate a counterexample structure M with, say, four solitary elements in their own partition cells, as shown in Figure 5.1. In the following iterations, all candidate axioms proposed by the synthesizer are required to be true in M. Notice that this is a favorable counterexample because it rules out several unsound candidates beyond φ. For instance, the unsound axiom

$$\forall x. \forall y. \forall z.\, (R(x, y) \lor R(y, z) \lor R(x, z)) \tag{5.4}$$

is eliminated by M, but would not have been eliminated by a partition of two elements in their own cells.



Rules out: $\forall xy.\, R(x,y)$, $\forall xyz.\, (R(x,y) \vee R(y,z) \vee R(x,z))$, $\exists x.\forall y.\, R(x,y)$, ...

Figure 5.1: A counterexample structure M that encodes a partition with 4 singleton cells. It witnesses unsoundness of the candidate axiom $\forall x.\forall y.\, R(x,y)$ and many others.

Building a counterexample synthesizer, which we call Cex, is a hard problem. In general, there is no procedure that delivers a counterexample even when one exists. However, since counterexamples guide search but are not crucial for progress, we resort to heuristics to find them. For example, we can query a constraint solver to find structures of small, bounded size (or finitary witnesses for infinite ones, which we call *pseudo-models*) that witness the unsoundness of candidate axioms.

Beyond soundness and efficiency, we also want axioms to be *independent* of each other. In other words, no single axiom should be logically implied by the other axioms. For example, suppose we have discovered the axiom for transitivity,

$$\forall x.\forall y.\forall z.(R(x,y) \wedge R(y,z) \rightarrow R(x,z)) \tag{5.5}$$

and the synthesizer proposes a similar axiom

$$\forall w.\forall x.\forall y.\forall z.(R(w,x) \wedge R(x,y) \wedge R(y,z) \rightarrow R(w,z)) \tag{5.6}$$

Both axioms are sound, but we would like to discard the latter because it is logically implied by the former. The problem of checking independence, i.e., whether a set of axioms $\{\psi_1, \ldots, \psi_n\}$ implies a candidate axiom φ, can be solved in this setting by checking the validity of

$$\bigwedge_i \psi_i \rightarrow \varphi \tag{5.7}$$

We can build an independence checker using VS, which is a validity procedure for arbitrary structures and signatures, and in this case, it can be realized using a semi-decision procedure for first-order validity. If we are able to prove 5.7 valid, then we discard φ. Otherwise, we add it to the growing set of axioms.

Despite our (unavoidable) reliance on semi-decision procedures and heuristics, our tool is able to find axiomatizations effectively. Given the description of the target class ψ (5.1), our tool finds the following axioms for reflexivity, symmetry, and transitivity (see next page):

$$\forall x.\, R(x,x)$$

$$\forall x.\forall y.\, (R(x,y) \leftrightarrow R(y,x))$$

$$\forall x.\forall y.\forall z.\, (R(x,y) \rightarrow (R(x,z) \rightarrow R(y,z))) \tag{5.8}$$

The reader may be wondering: at what point does the synthesis loop stop? When is an axiomatization *complete*, or good enough? What does *good enough* mean? These are interesting and tricky questions (as discussed in Section 5.1), and answers depend heavily on the specific problem, which makes automation very hard.

In the case of relations defined by a partition, proving completeness of a set of axioms A turns out to be subtle. Intuitively, we want to know whether for any structure that satisfies A there is a function f such that formula 5.1 holds (a second-order quantification over functions). The difficulty with proving this, intuitively, is that one needs to select a representative from each equivalence class of R. For example, if $R(a,b)$ holds, then it should be the case that $f(a) = f(b)$, but there are many choices for this element. Proving the existence of a suitable function f is difficult to automate (in general it seems to require the axiom of choice), and we did not implement a completeness checker for this example.

5.3 THE AXIOM SYNTHESIS PROBLEM

In this section we define the problem of study. We first introduce some useful concepts.

5.3.1 Preliminaries

We formulate the problem of axiom synthesis as parameterized by an *abstract logic* $\mathcal{L}$:

Definition 5.1 (Abstract Logic)**.** An *abstract logic* $\mathcal{L}$ is a tuple $(\mathcal{F}, \mathcal{S}, \models)$ where

- $\mathcal{F}$ is a set of *formulae*.
- $\mathcal{S}$ is a class of *models*.
- $\models \subseteq \mathcal{S} \times \mathcal{F}$ is the binary *satisfaction relation*.

Satisfaction and Entailment We say M *satisfies* φ if $M \models \varphi$ holds. Equivalently, we say φ *holds in* M or M *is a model of* φ or φ *is true in* M. Let $M \in \mathcal{S}$ be a model, $\mathcal{C} \subseteq \mathcal{S}$ be a subclass of models, $T \subseteq \mathcal{F}$ be a subset of formulae, and $\varphi \in \mathcal{F}$ be a formula. We say M satisfies T, written $M \models T$, to mean $M \models \varphi$ for every $\varphi \in T$. We lift this to a class of

models $\mathcal{C}$ and write $\mathcal{C} \models \varphi$ (or $\mathcal{C} \models T$) if every model $M \in \mathcal{C}$ satisfies φ (resp. satisfies T). We also use $\models$ to denote logical entailment. We say φ is entailed by T, written $T \models \varphi$, to mean that every model of T satisfies φ.

Theories and Models A *theory* is a set of formulae $T \subseteq \mathcal{F}$ that is entailment closed, i.e., for every $\varphi \in \mathcal{F}$, if $T \models \varphi$, then $\varphi \in T$. The theory of a class of models $\mathcal{C}$, denoted by $Th(\mathcal{C})$, is the set $\{\varphi \in \mathcal{F} \mid \mathcal{C} \models \varphi\}$ of formulae that hold in all models in $\mathcal{C}$. The dual of this view is the class $\{M \in \mathcal{S} \mid M \models T\}$ consisting of all models that satisfy a set of formulae T, which we denote by $\mathsf{Mod}(T)$. Observe that the larger the theory, the smaller its class of models, i.e., if $T \subseteq T'$ then $\mathsf{Mod}(T') \subseteq \mathsf{Mod}(T)$. In fact, taking the two partially-ordered sets $(2^{\mathcal{F}}, \subseteq)$ and $(2^{\mathcal{S}}, \supseteq)$, the monotonic functions $\mathsf{Mod} : 2^{\mathcal{F}} \to 2^{\mathcal{S}}$ and $Th : 2^{\mathcal{S}} \to 2^{\mathcal{F}}$ form a monotone Galois connection [98, 99], since for any set of formulae T and class $\mathcal{C}$ we have $T \subseteq Th(\mathcal{C}) \Leftrightarrow \mathsf{Mod}(T) \supseteq \mathcal{C}$.

$\mathcal{L}$-Elementary Class One consequence of the above Galois connection is that for any $\mathcal{C} \subseteq \mathcal{S}$ we have $\mathsf{Mod}(Th(\mathcal{C})) \supseteq \mathcal{C}$. We distinguish a special case, namely, when $\mathsf{Mod}(Th(\mathcal{C})) = \mathcal{C}$. We refer to such a class $\mathcal{C}$ as $\mathcal{L}$-elementary. One can see from the above definitions that a class is $\mathcal{L}$-elementary if and only if it can be defined as $\mathsf{Mod}(T)$ for some theory T. This is inspired by the concept of an elementary class from model theory [100], and it informs our definition of completeness in Section 5.3.2.

5.3.2 A Model-Theoretic Formulation of Axiom Synthesis

The model-theoretic axiomatization problem is parameterized by an abstract logic $\mathcal{L} = (\mathcal{F}, \mathcal{S}, \models)$. The objective is to axiomatize a *target class* of models $\mathcal{C} \subseteq \mathcal{S}$ using formulae in $\mathcal{F}$. We use the word *axiom* for any formula in $\mathcal{F}$ that is true in all models in $\mathcal{C}$, equivalently, $\varphi \in Th(\mathcal{C})$[2].

We typically want a mutually-independent set of axioms, i.e., one where no single axiom follows from the others. A set of formulae $T \subseteq \mathcal{F}$ is *mutually independent* if $T \setminus \{\varphi\} \not\models \varphi$ for every $\varphi \in T$.

Definition 5.2 (Finite Axiomatization)**.** Given an abstract logic $\mathcal{L} = (\mathcal{F}, \mathcal{S}, \models)$ and target class $\mathcal{C} \subseteq \mathcal{S}$, a finite set $A \subseteq \mathcal{F}$ is said to be an axiomatization of $\mathcal{C}$ if $\mathcal{C} \models A$. Additionally, an axiomatization is said to be *non-vacuous* if A contains no tautologies over $\mathcal{S}$, and it is *mutually independent* if A is also mutually independent.

[2]In practice, we focus on discovering *simple* (e.g., short) axioms.

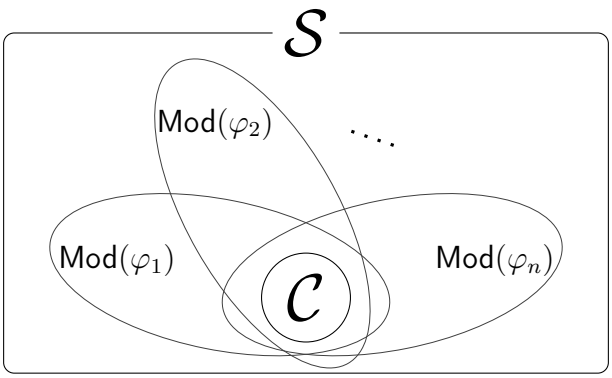


Figure 5.2: Model-theoretic axiomatization.

We develop algorithms for synthesizing sound, non-vacuous, and mutually-independent finite axiomatizations in this chapter[3].

In the space of models, axiomatization can be viewed as over-approximating $\mathcal{C}$ as a subclass of $\mathcal{S}$ using independent axioms A. Consider Figure 5.2. We see that an axiom φ corresponds to a class of models $\mathsf{Mod}(\varphi) \supseteq \mathcal{C}$ such that φ is satisfied everywhere in that class. As the set A grows, the space of models shrinks to

$$\mathsf{Mod}(A) = \bigcap_{\varphi \in A} \mathsf{Mod}(\varphi) \tag{5.9}$$

as fewer and fewer models satisfy every axiom. The dual perspective gives the *theory of the axioms* $Th(A) = \{\varphi \in \mathcal{F} \mid A \models \varphi\}$, which grows with A. For any axiomatization A, the theory of A is contained in the theory of $\mathcal{C}$, i.e., $Th(A) \subseteq Th(\mathcal{C})$.

Axiomatizations are also equipped with a natural partial order on their theories.

Definition 5.3 (Precision Order on Axiomatizations)**.** Let A and A' be axiomatizations of $\mathcal{C}$ according to Definition 5.2. We say A is *more precise* than A' if $Th(A) \supset Th(A')$.

We focus on finding axiomatizations that are as precise as possible, with the ideal scenario being a *complete axiomatization*.

As discussed in Section 5.1, although it is tempting to think of a complete axiomatization as an axiomatization A such that $\mathsf{Mod}(A) = \mathcal{C}$, it may happen that $\mathcal{C} \subset \mathcal{C}^* := \mathsf{Mod}(Th(\mathcal{C}))$, i.e., $\mathcal{C}$ is not $\mathcal{L}$-elementary, and hence the models of any axiomatization of $\mathcal{C}$ will always be a strict superset of $\mathcal{C}$. We hence use the following notion of completeness:

[3] Note that mutual independence entails non-vacuity since tautologies are entailed by the empty set of axioms. However, we state non-vacuity explicitly as it is a basic property of axiomatizations.

Definition 5.4 (Complete Finite Axiomatization)**.** A finite axiomatization A is said to be a complete axiomatization for $\mathcal{C}$ if $Th(A) = Th(\mathcal{C})$.

This definition allows for completeness with respect to the expressive power of $\mathcal{L}$. Since $Th(A) \subseteq Th(\mathcal{C})$ for any axiomatization A, a complete axiomatization is the most precise, i.e., has the largest possible theory among all axiomatizations of $\mathcal{C}$.

Discussion on axioms for proof systems While our formulation is model-theoretic, it is of course useful in the context of proof systems as well. A proof system is a set of facts along with rules that allow one to infer judgments of the form $\Gamma \vdash \varphi$, which mean that φ is derivable from Γ using the rules. Let us assume a proof system that is sound and *strongly complete* over all models $\mathcal{S}$, or, formally, $\Gamma \vdash \varphi \Leftrightarrow \Gamma \models \varphi$. An axiomatization is then sound for the proof system as well: if A is an axiomatization of $\mathcal{C}$ per Definition 5.2 and $A \vdash \varphi$, then $\varphi \in Th(\mathcal{C})$. Observe that the completeness criterion in this case also reduces to our definition using the completeness of the proof system. Let A be an axiomatization that is complete for $\mathcal{C}$ per Definition 5.4, i.e., $A \models \varphi \Leftrightarrow \varphi \in Th(\mathcal{C})$. Using the soundness and strong completeness of the proof system, we get that $A \vdash \varphi \Leftrightarrow \varphi \in Th(\mathcal{C})$. Therefore A is also complete for the theory of $\mathcal{C}$ with respect to the proof system. In our presentation we do not explicitly consider axioms for proof systems; we concentrate only on the model-theoretic formulation.

5.4 LEARNING-BASED AXIOM SYNTHESIS FRAMEWORK

In this section, we describe the Learning-based Axiom Synthesis (LAS) framework to synthesize precise or complete axiomatizations. We argue that effective axiom synthesis algorithms can be built by implementing the LAS framework for domains of interest. We describe the atomic components of the framework in Section 5.4.1, the high-level algorithm which uses the components in Section 5.4.2, and in Section 5.4.3 we formulate a constraint-based synthesizer used for our instantiations of LAS (Sections 5.5 and 5.6).

5.4.1 Components of the LAS Framework

We now develop the set of independent functional components that underlie LAS. Instantiating LAS for a given domain requires the implementation of these components for that domain. The components are parameterized by an abstract logic $\mathcal{L} = (\mathcal{F}, \mathcal{S}, \models)$ and a target class $\mathcal{C} \subseteq \mathcal{S}$, which we fix throughout this section. We describe the components below.

- **Soundness checker**: A procedure dubbed VC (*validity over* $\underline{\mathcal{C}}$), which takes as input a formula $\varphi \in \mathcal{F}$ and determines whether φ is an axiom for $\mathcal{C}$, i.e. whether $\mathcal{C} \models \varphi$

Although soundness can be determined using an axiomatization in $\mathcal{L}$, note that having an axiomatization in $\mathcal{L}$ is certainly not a requirement for building VC. We explore two approaches. For the domain of modal logic, we build VC using an axiomatization in first-order logic, the axioms of which do not indicate the modal axioms in any reasonable sense. For the domain of languages with Kleene star, we build VC using a reference implementation of operations on regular languages.

- **Independence checker**: A procedure dubbed VS (*validity over* $\underline{\mathcal{S}}$), which takes a set of axioms $A \subseteq \mathcal{F}$ and an axiom $\varphi \in \mathcal{F}$ as inputs and determines whether φ is independent of A, i.e. whether $A \not\models \varphi$. Note that the entailment is over all models in $\mathcal{S}$.

- **Counterexample generator**: A procedure dubbed Cex, which takes as input a formula $\varphi \in \mathcal{F}$ that is not an axiom of $\mathcal{C}$ and produces a *pseudo-model* pm as a counterexample to φ.

A natural notion of a counterexample for φ is a model $M \in \mathcal{C}$ such that $M \not\models \varphi$. In many domains, however, it may be the case that most (or even all) models in $\mathcal{C}$ are infinite (see Section 5.6). Therefore, we require for such domains the definition of finitely-representable objects called *pseudo-models*, along with a *witness map* $\mathcal{W}$ that associates to every pseudo-model a set of formulae that it rules out. A pseudo-model pm is said to be a *counterexample to* φ if and only if $\varphi \in \mathcal{W}(pm)$. Pseudo-models *witness the existence of models* in $\mathcal{C}$ that rule out unsound candidate axioms, and often we can precisely describe these models with a mapping $\mathcal{M}$ that maps a pseudo-model pm to the set $\mathcal{M}(pm) \subseteq \mathcal{C}$ of models indicated by pm. When we have the mapping $\mathcal{M}$, one natural definition for $\mathcal{W}(pm)$ is the one that rules out any candidate that is false in all models from $\mathcal{M}(pm)$, i.e. $\mathcal{W}(pm) = \{\varphi \in \mathcal{F} \mid \forall M \in \mathcal{M}(pm),\ M \not\models \varphi\}$. These notions are reminiscent of those from abstract formulations of learning which associate with any sample a *set* of concepts that the sample is consistent with (e.g. [101]).

As an example, suppose we are working with a class of models that have as their domain the set of integers. Consider an unsound candidate axiom $\varphi \coloneqq \forall x.\, f(x) = x$, where f is uninterpreted. One possible pseudo-model pm might specify that $f(1) = 2$ while leaving f unspecified everywhere else. We can take $\mathcal{M}(pm)$ to be the set of all models that have $f(1) = 2$, with $\mathcal{W}$ stipulating that $\varphi \in \mathcal{W}(pm)$ because φ is already false with the partially defined f (and extending f cannot help). Thus pm is a finitely-presented counterexample to

φ, but it is not a model, since it does not specify the value of f on all integers. The notion of pseudo-models will vary by domain, and we present the specifics where relevant.

- **Completeness checker**: A procedure dubbed CC, which takes as input a set of axioms A and determines whether A is a complete axiomatization in the sense of Definition 5.4.

As discussed earlier, building CC is difficult, especially when we have $\mathcal{C} \subset \mathcal{C}^* \coloneqq \mathsf{Mod}(\mathit{Th}(\mathcal{C}))$. Therefore, we view the completeness checker as an optional component in the LAS framework, and typically we would only consider implementing it when $\mathcal{C}^* = \mathcal{C}$, where we can make use of knowledge about the target class $\mathcal{C}$. In general, we aim to find axiomatizations that are as precise as possible.

- **Formula Synthesizer**: A procedure dubbed $\mathsf{Learner}_{\mathcal{W}}$, which takes as input a set PM of pseudo-models and a set Avoid of formulae and synthesizes a formula $\varphi \in \mathcal{F}$ such that $\varphi \notin \mathcal{W}(pm)$ for every $pm \in PM$ and $\varphi \notin \mathit{Avoid}$. Note that the procedure is parameterized by the domain-specific map $\mathcal{W}$. We describe a formulation for $\mathsf{Learner}_{\mathcal{W}}$ using SMT solvers in Section 5.4.3.

We demonstrate in the following sections that, by building the various components described above, it is possible to realize effective axiom synthesis for different domains using a single framework. Building effective components is crucial to the success of axiom synthesis in LAS; we make contributions in this respect by implementing variants of the above components for two domains with complex requirements.

5.4.2 The Core LAS Algorithm

Here we describe the core algorithm for the LAS framework, which utilizes the components described in Section 5.4.1. We make two simplifications for presentation: (1) we assume all components implement decision procedures and (2) we exclude completeness checking, given that it is optional and there are many possible variants for incorporating it within the algorithm. After describing the core algorithm we discuss how it works in general without the simplifications.

The core of the LAS framework is presented in Algorithm 5.1. It is parameterized by a logic $\mathcal{L}$ and target class $\mathcal{C}$, for which we must implement the soundness checker VC, independence checker VS, counterexample generator Cex, and the formula synthesizer $\mathsf{Learner}_{\mathcal{W}}$.

The algorithm maintains a set A of discovered axioms, a set PM of pseudo-models, and a set Avoid of formulae, all initially empty. It synthesizes axioms in a loop until a timeout

parameters: Logic $\mathcal{L} = (\mathcal{S}, \mathcal{F}, \models)$, target class $\mathcal{C} \subseteq \mathcal{S}$, and *timeout*
imports: VC, VS, Cex, Learner$_{\mathcal{W}}$ over $\mathcal{L}, \mathcal{C}$
output: Axioms $A \subseteq \mathcal{F}$ for $\mathcal{C}$

```
1:  procedure LAS:
2:      A, PM, Avoid ← ∅
3:      repeat
4:          φ ← Learner_W(PM, Avoid)    ▷ Get a proposal not ruled out by counterexamples
5:          sound ← VC(φ)
6:          if sound then
7:              independent ← VS(A, φ)
8:              if independent then
9:                  A ← A ∪ {φ}
10:             else// Rule out φ and continue
11:                 Avoid ← Avoid ∪ {φ}
12:                 continue
13:         else// Get counterexample and continue
14:             pm ← Cex(φ)
15:             PM ← PM ∪ {pm}
16:     until timeout
17:     return A
```

Algorithm 5.1: Core LAS Algorithm.

is reached (lines 3-16), at which point it returns A. In a given iteration of the loop, with discovered axioms $A = \{\psi_1, \psi_2, \ldots, \psi_n\}$, it ensures ψ_i is independent of $\{\psi_j \mid 1 \leq j < i\}$, assuming the ψ_i are enumerated in order of discovery. Additionally, for every $\psi \in A$, the algorithm ensures that $\psi \notin \mathcal{W}(pm)$ for every $pm \in PM$.

At the head of the loop, the algorithm queries Learner for a new candidate φ which is neither ruled out by the current counterexample pseudo-models nor a member of the set *Avoid* of formulae (line 4). The algorithm checks whether φ is sound using VC (line 5). If sound, it checks for independence from A using VS (line 7). If φ is independent, the algorithm adds it to A and continues to the next iteration of the loop to find more axioms (line 9). If not independent, i.e., φ is entailed by A, the algorithm discards φ (adding it to the *Avoid* set to prevent it from being proposed in the future) and goes back to the head of the loop to get another candidate (lines 11-12). If the soundness check fails, the algorithm queries Cex for a pseudo-model pm such that $\varphi \in \mathcal{W}(pm)$ and adds it to PM before returning to the head of the loop (lines 14-15).

We now discuss some technical details about how the algorithm works in the general case without the simplifications.

Nonterminating Procedures. In general, the components described in Section 5.4.1 may only be realizable as nonterminating procedures. For example, in Section 5.5 we instantiate the framework for modal logics, where VS is implemented with a procedure for first-order logic validity, which can only be a *semi*-decision procedure that halts and returns true on valid formulae but may not terminate on invalid formulae. Therefore, we modify each component to take an additional input $\mathit{fuel} \in \mathbb{N}$ that models a resource bound and ensures termination. For example, we modify VC in this way so that for every $\mathit{fuel} \in \mathbb{N}$ and $\varphi \in \mathcal{F}$ we have that $\mathsf{VC}(\varphi, \mathit{fuel})$ always terminates, saying either that φ is an axiom or is not an axiom, or else *Unknown*. This allows us to stage the various procedures as terminating sub-procedures which we can call iteratively with increasing *fuel*.

With this modification, we must also extend the algorithm to make decisions when a component returns *Unknown*. If Cex returns *Unknown* (line 14), we can discard φ after adding it to the *Avoid* set, which rules it out from future proposals, and continue searching for more axioms without adding a counterexample. If Learner returns *Unknown*, we can increase its *fuel* or exit and return A. If VC returns *Unknown* on a given axiom φ, we could add φ to the list of axioms and emit a warning that φ may not be sound. We could also discard φ (risking that we do not find a useful, sound axiom) or increase *fuel* and rerun VC on φ. Similarly, if we cannot prove independence from A for a given sound axiom using VS, we must choose between running again with more *fuel*, keeping the potentially redundant axiom, or discarding it. Which choices are effective will vary by setting, and we leave them up to the implementation.

Mutual Independence Observe that the axioms A returned by the algorithm are not necessarily mutually independent. The algorithm only ensures that each axiom is independent from those discovered before it. This is a practical choice, since finding a minimal mutually-independent subset of A that covers A could require an exponential number of calls to VS. In many cases it may be preferable to have a larger set of simple axioms rather than a smaller set of complex axioms. However, if we assume that the Learner component outputs candidates in order of increasing complexity, it may happen that a simple axiom discovered early may be entailed by a combination of more complex axioms discovered later. One way to address this is to run the algorithm to obtain a set of axioms $A = \{\psi_1, \psi_2, \ldots, \psi_n\}$, and then use VS to remove ψ_1 from A if it is entailed by $A \setminus \{\psi_1\}$. We can repeat this process with ψ_2 and $A \setminus \{\psi_1\}$, and continue until each remaining axiom is not entailed by the others, using only a linear number of calls to VS.

Completeness Check If it is possible to check completeness, then the algorithm can use completeness as a termination condition. However, it may not make practical sense to check completeness each time a new axiom is added to A as it could be expensive. Other choices include checking completeness at the end of synthesis or checking at regular intervals. Again, these are choices that depend on the domain and are left up to the implementation.

5.4.3 Realizing the Learner using an SMT Solver

We implement a generic learner parameterized by a logic (syntax and semantics), which synthesizes expressions that are not ruled out by any of a given set of pseudo-models. We use established constraint solving techniques to synthesize formulae of bounded height (we assume that the logic has a bounded number of constants). Note that the constraints encoding whether a formula is ruled out by a pseudo-model are determined by the domain-specific map $\mathcal{W}$. We thus require that, for any pseudo-model pm, the constraints encoding $\varphi \notin \mathcal{W}(pm)$ are expressible as a quantifier-free first-order formula[4]. Consequently, the synthesis of bounded-depth logical expressions reduces to quantifier-free satisfiability and can be accomplished (often) with an SMT solver. More precisely, given a bound b on the depth of expressions, we can use a set of Boolean variables V_b to encode choices for how the nodes in the parse tree of the expression are to be filled. For any fixed pseudo-model pm, we then write a formula $Allowed_{pm}(V_b)$ that constrains the expression (represented by an assignment to V_b) so that it is *not* ruled out by pm. We then check the satisfiability of the formula:

$$\bigwedge_{pm \in PM} Allowed_{pm}(V_b) \tag{5.10}$$

conjoined with a constraint that rules out all the formulae in the set $Avoid$. If satisfiable, we can extract the expression from the assignment to V_b in the satisfying model. We implement this learner for the required logic in each setting.

5.5 AXIOMATIZING CLASSES OF FRAMES IN MODAL LOGIC

In this section, we instantiate the LAS framework for the setting of modal logic and the problem of axiomatizing first-order definable classes of frames, which are first-order structures over a single binary "accessibility" relation R. Section 5.5.1 provides background, Section 5.5.2

[4]Evaluating operators like universal quantification on a pseudo-model with a finite domain can be expressed as a quantifier-free formula using a conjunction over the elements of the domain of the pseudo-model.

$M, w \models \top$ always
$M, w \models p$ iff $p \in V(w)$
$M, w \models \neg\varphi$ iff $M, w \not\models \varphi$
$M, w \models \varphi \vee \varphi'$ iff $M, w \models \varphi$ or $M, w \models \varphi'$
$M, w \models \varphi \wedge \varphi'$ iff $M, w \models \varphi$ and $M, w \models \varphi'$
$M, w \models \Box\varphi$ iff $M, w' \models \varphi$ for every $(w, w') \in R$
$M, w \models \Diamond\varphi$ iff $M, w' \models \varphi$ for some $(w, w') \in R$

Figure 5.3: Semantics of Modal Logic

explains the components of the framework and discusses nuances for this particular setting, and Section 5.5.3 describes details of the implementation and the axiomatizations we find.

5.5.1 Modal Logic and Correspondence Theory

We now briefly review the syntax and semantics of propositional modal logic over Kripke structures. We are interested in axiomatizing properties of the accessibility relation for Kripke structures using modal logic formulae. In particular, we aim to find axiomatizations of first-order definable properties that are classically studied in correspondence theory. We review some background from correspondence theory following the syntax and semantics of propositional modal logic.

Syntax and Semantics of Modal Logic We consider propositional modal logic over Kripke structures (henceforth called models) of the form $M = (W, R, V)$, consisting of a set of *worlds* W an *accessibility relation* $R \subseteq W \times W$, and a *valuation* $V : W \to \mathcal{P}(Prop)$ that maps worlds to a subset of propositions from a finite set $Prop$. We refer to a pair $F = (W, R)$ as a *frame*, and in the context of a specific model $M = (W, R, V)$ we refer to F as the frame of M. Frames are the basic object we aim to axiomatize.

Formulae in the logic are given by the following grammar:

$$\varphi \ ::= \ \top \mid p \in Prop \mid \neg\varphi \mid \varphi \vee \varphi' \mid \varphi \wedge \varphi' \mid \Box\varphi \mid \Diamond\varphi \tag{5.11}$$

The formulae $\Diamond\varphi$ and $\Box\varphi$ can be read in the usual way as "it is possible that φ" and "it is necessary that φ", respectively. The semantics of modal logic in Figure 5.3 defines when a formula is true at a given world $w \in W$ in a model $M = (W, R, V)$. These are standard syntax and semantics of propositional modal logic [92].

Correspondence Theory in Modal Logic Modal Correspondence Theory [93] studies connections between modal logic formulae and classical first-order properties of frames. As an example, the modal formula $\Box p \to p$ corresponds to the *reflexive frames*, i.e., those for which $\forall x.R(x,x)$ holds when we treat a frame $F = (W, R)$ as a first-order structure. The precise correspondence is the following:

Theorem 5.1 ([93])**.** For any frame $F = (W, R)$, the accessibility relation R is reflexive if and only if the modal formula $\Box p \to p$ is true in $M = (W, R, V)$ for all valuations V and worlds w.

Many correspondences of this kind are known to exist between modal formulae and the accessibility relation, and they rely on the notion of *frame validity*, defined as follows:

Definition 5.5 (Frame Validity)**.** A modal formula is valid in a frame $F = (W, R)$, written $F \models_f \varphi$, if $(W, R, V), w \models \varphi$ for every world $w \in W$ and every valuation $V : W \to \mathcal{P}(Prop)$.

For intuition, let us work through the proof of Theorem 5.1.

Proof of Theorem 5.1. Soundness ($\Rightarrow$). Suppose $F = (W, R)$ is a reflexive frame, i.e., $\forall x.R(x,x)$ is true, and let V, w be an arbitrary valuation and world, respectively. Suppose $(W, R, V), w \models \Box p$, i.e., every world accessible from w has p true under V (if not, the implication is already true). We have that $(W, R, V), w \models p$ holds because $R(w, w)$ holds. Note that this direction of the proof corresponds to what VC checks in our framework.

Completeness ($\Leftarrow$). We prove the contrapositive. Intuitively, we pick a w for which $R(w, w)$ is false, and from the form of the axiom we pick a valuation V that makes the axiom false at w. (We explain how we automate some of this intuition in Section 5.5.2.) Formally, suppose F is not reflexive. That means there is some $w \in W$ where $R(w, w)$ does not hold. Let V be a valuation that makes p false at w and true at all w' for which $R(w, w')$ holds. Then $(W, R, V), w \models \Box p$, but $(W, R, V), w \not\models p$. QED.

Our goal now is to instantiate the LAS framework to find modal logic formulae that characterize various classes of frames, as described above. Note that the axioms we aim to find in this setting can be interpreted as axiom schemas. For instance, the formula $\Box p \to p$ can soundly be interpreted as a schema $\Box\alpha \to \alpha$, where α is a placeholder for a modal formula, i.e., all instances of this schema are valid in the class of reflexive frames, and the same can be said of the other classes we axiomatize. We refer the reader to [93] for more about correspondence theory.

5.5.2 Instantiating the LAS Framework for Modal Logic

We now instantiate the framework from Section 5.4 for the problem of synthesizing modal logic axioms that characterize a target class of frames. We aim to axiomatize classes of frames that are definable in first-order logic, i.e., classes defined by a first-order logic sentence ψ over the relation symbols $R, =$, e.g., reflexive frames defined by $\psi \coloneqq \forall x.R(x,x)$. Because modal logic formulae can be translated to first-order logic, the components make use of existing validity procedures for first-order logic. We discuss this translation next.

Definition 5.6 (Translation of Modal Logic to First-Order Logic)**.** Given a modal logic formula φ, we can translate φ into a first-order logic formula $\psi(x)$ such that $(W, R, V), w \models \varphi$ if and only if $M' \models \psi(w)$, where M' extends the frame (W, R) (as a first-order structure) with interpretations for unary predicates P, one for each proposition $p \in \mathit{Prop}$. Each predicate P holds for the worlds w for which $p \in V(w)$. The translated formula $\psi(x) \coloneqq \mathsf{ml2fo}_x(\varphi)$ is defined inductively in the structure of φ as shown below, with x, y, etc. drawn from an infinite supply of fresh variables:

- $\mathsf{ml2fo}_x(\top) = \top$
- $\mathsf{ml2fo}_x(p) = P(x)$
- $\mathsf{ml2fo}_x(\neg\varphi) = \neg\mathsf{ml2fo}_x(\varphi)$
- $\mathsf{ml2fo}_x(\varphi \vee \varphi') = \mathsf{ml2fo}_x(\varphi) \vee \mathsf{ml2fo}_x(\varphi')$
- $\mathsf{ml2fo}_x(\varphi \wedge \varphi') = \mathsf{ml2fo}_x(\varphi) \wedge \mathsf{ml2fo}_x(\varphi')$
- $\mathsf{ml2fo}_x(\Box\varphi) = \forall y.\,(R(x,y) \rightarrow \mathsf{ml2fo}_y(\varphi))$
- $\mathsf{ml2fo}_x(\Diamond\varphi) = \exists y.\,(R(x,y) \wedge \mathsf{ml2fo}_y(\varphi))$

Recall that the LAS framework involves the procedures VC, VS, and Cex, whose respective purposes are to check soundness, check independence, and generate counterexamples for candidate axioms. In this setting, we aim to find axioms for a subclass of frames, and thus $\mathcal{S}$ is the class of all frames and $\mathcal{C}$ is a subclass of $\mathcal{S}$, e.g., reflexive frames. The logic $\mathcal{L}$ is modal logic with *frame validity* for the entailment relation (Definition 5.5). In this setting, we also implement a completeness checker CC. We explain each component next and discuss some details for the $\mathsf{Learner}$ in Section 5.5.3.

Instantiating VC. We reduce the soundness condition for modal logic axioms over a class of first-order definable frames $\mathcal{C}$ to the validity problem in first-order logic (more

precisely, the relational theory of equality and uninterpreted relations) by using the first-order characterization for $\mathcal{C}$. Recall the proof of soundness for Theorem 5.1. Suppose we are axiomatizing the class of reflexive frames, which are characterized by the first-order sentence $\forall x.R(x,x)$. To check the proposed axiom $\Box p \to p$ is sound (true in all frames from $\mathcal{C}$), we can check the validity of the first-order sentence $\forall x.R(x,x) \to (\forall x.\mathsf{ml2fo}_x(\Box p \to p))$. More generally, for a class of frames defined by a first-order sentence ψ, we check the soundness of a candidate modal axiom φ by checking the validity of

$$\mathsf{sound}(\psi, \varphi) \coloneqq \psi \to \forall x.\mathsf{ml2fo}_x(\varphi) \tag{5.12}$$

Observe that the ml2fo translation turns propositions into uninterpreted unary relations, and thus first-order validity of the translated formula requires the formula to be true for all interpretations of the unary relations. This corresponds to quantifying over all valuations. We can thus implement VC using any semi-decision procedure for validity in first-order logic (see Section 5.5.3 for details).

Instantiating Cex. As discussed above, we reduce soundness to the relational theory of equality and uninterpreted relations, which is recursively enumerable but undecidable. Since satisfiability is not recursively enumerable, it is straightforward to show there can be no complete procedure to find counterexample models for an unsound candidate φ, i.e., a model of $\neg\mathsf{sound}(\psi, \varphi)$. However, our intuition is that finite counterexample models are enough to efficiently synthesize many modal axiomatizations. Given a candidate φ that is not sound, the component Cex produces a frame $F = (W, R) \in \mathcal{C}$ of a small, bounded size such that $F \not\models_f \varphi$. Note that in this setting we can take pseudo-models to simply be finite frames F, with $\mathcal{M}(F) = \{F\}$ and $\varphi \in \mathcal{W}(F) \Leftrightarrow F \not\models_f \varphi$.

Instantiating VS. Given a set of modal axioms $A = \{\varphi_1, \ldots, \varphi_n\}$ and a candidate axiom φ, we want to check that φ is independent of A. That is, we want to check that there is a frame $F = (W, R)$ such that $F \models_f \varphi_i$ for each i but $F \not\models_f \varphi$. Checking the existence of such a frame (i.e., independence) does not reduce to first-order validity, but is captured by the second-order formula

$$\exists(W, R). \bigwedge_i \forall P.\forall x.\mathsf{ml2fo}_x(\varphi_i) \wedge (\exists P'.\exists x.\neg\mathsf{ml2fo}_x(\varphi)) \tag{5.13}$$

where P, P' are sequences of second-order variables corresponding to the unary predicates produced by the ml2fo translations.

We approximate independence by checking whether there is such a frame $F = (W, R)$ of some small, bounded size. As for finding counterexamples to soundness, our intuition is that

small witnesses will exist to show the independence of modal axioms. We note that this heuristic could cause independent axioms to be discarded if we insist on finding independence proofs and there are only large frames witnessing independence. In principle, we can mitigate that risk by increasing the size bound, but this proved unnecessary in experiments.

Instantiating CC. We now describe a heuristic procedure for completeness. Let us assume a first-order characterization ψ for a class of frames and a set of sound modal logic axioms A. Let us also assume, without loss of generality, that the propositions in each axiom are disjoint, and let φ be the conjunction of the modal axioms in A.

The formula φ is complete for the class of frames described by ψ if for all frames F, whenever $F \models_f \varphi$ (frame validity; see Definition 5.5), then $F \models \psi$ (first-order validity). In other words, we need to check the validity (over all frames) of the second-order sentence

$$(\forall P. \forall w. \mathsf{ml2fo}_w(\varphi)) \to \psi \tag{5.14}$$

where once again P is a sequence of second-order variables introduced by $\mathsf{ml2fo}_w(\varphi)$.

Following a common pattern for completeness proofs in correspondence theory, we can try to prove the contrapositive of 5.14: assume the first-order characterization ψ does not hold on some frame and then try to find a specific valuation and world that violate φ.

Using this intuition from manual proofs, we reduce the problem to a stronger version where we try to find a *finite* set of worlds $\{w_1, \ldots, w_n\}$ and a valuation on them that violates φ. For worlds outside $\{w_1, \ldots, w_n\}$, we simply assume that the valuation uniformly assigns the same default value v_{def}. For instance, suppose we only have one atomic proposition p. Then, instead of the second-order sentence 5.14, we can check the validity of the following first-order sentence:

$$(\forall w_1, \ldots, w_n. \forall v_1, \ldots, v_n, v_{\text{def}}. \forall w. \mathsf{ml2fo}'_w(\varphi)) \to \psi \tag{5.15}$$

where $v_1, \ldots, v_n, v_{\text{def}}$ range over Booleans (modeling the valuation of P on $w_1, \ldots, w_n$ and the default value), and where $\mathsf{ml2fo}'_w(\varphi)$ is $\mathsf{ml2fo}_w(\varphi)$ where each predicate occurrence $P(w)$ is defined as:

$$P(w) \coloneqq ite(w = w_1, v_1, ite(w = w_2, v_2, \ldots ite(w = w_n, v_n, v_{\text{def}}) \ldots)) \tag{5.16}$$

Notice that the validity of Formula 5.15 implies the validity of Formula 5.14. Furthermore, Formula 5.15 is in first-order logic (as valuations have been replaced by a finite set of Boolean variables), and we can use automatic procedures for first-order logic to solve validity. This approximate checking for completeness works well in practice (in fact, it works for 14/17 of the modal axiomatizations we explore in Section 5.5.3).

5.5.3 Implementation and Evaluation

We implemented the procedures VC, VS, Cex, and CC as described in previous sections, reducing the problems to SMT queries in Z3 [11]. We built an axiom synthesizer (Learner) for modal logic and combined the components as described in Algorithm 5.1. Our artifact can be found at: https://zenodo.org/records/7072506.

Table 5.1: First-order descriptions used in Table 5.2.

FO Description	**Definition**
Reflexive	$\forall x.R(x,x)$
Transitive	$\forall x,y,z.(R(x,y) \land R(y,z)) \rightarrow R(x,z)$
Symmetric	$\forall x,y.R(x,y) \rightarrow R(y,x)$
Euclidean	$\forall x,y,z.(R(x,y) \land R(x,z)) \rightarrow (R(y,z) \land R(z,y))$
Functional	$\forall x,y,z.(R(x,y) \land R(x,z)) \rightarrow y = z$
Shift Reflexive	$\forall x,y.R(x,y) \rightarrow R(y,y)$
Dense	$\forall x,y.R(x,y) \rightarrow \exists z.R(x,z) \land R(z,y)$
Serial	$\forall x.\exists y.R(x,y)$
Convergent	$\forall x,y,z.(R(x,y) \land R(x,z)) \rightarrow \exists w.R(y,w) \land R(z,w)$

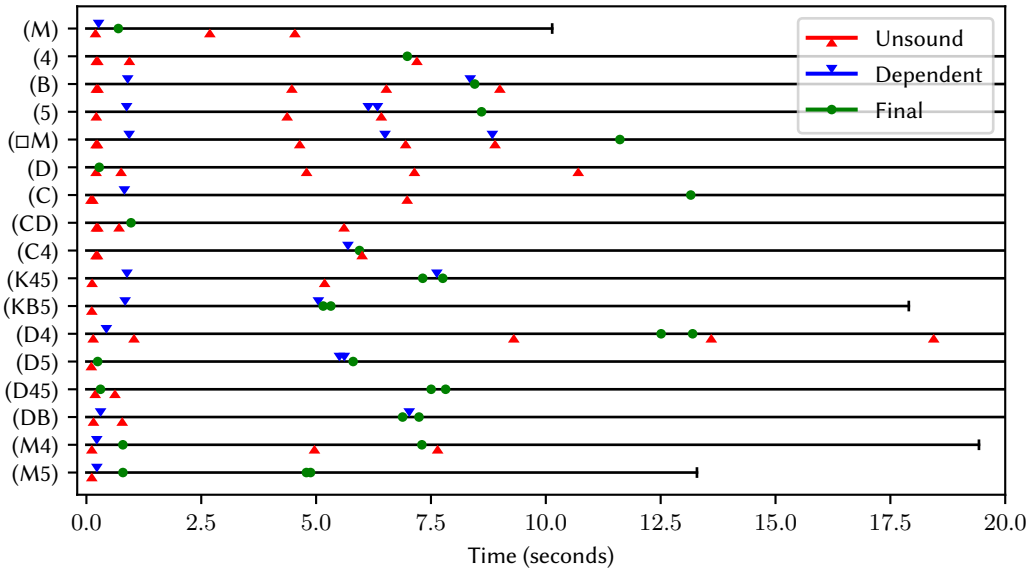


Figure 5.4: Distribution of candidate axioms proposed for each modal logic. The x-axis shows the time at which each axiom was proposed, and each horizontal line shows the duration of synthesis (not including the completeness check) cut off at 20 seconds (see Table 5.2 for the full duration). "Unsound" refers to axioms that failed the soundness checker VC; "Dependent" refers to axioms that were proven sound but were entailed by later proposals and pruned in a post-processing pass; "Final" indicates the axioms that our tool outputs given in Table 5.2.

Implementation Details Let ψ be a first-order description for a class of frames for which we want to synthesize modal axioms, let φ be a candidate modal axiom, and let A be a set of sound modal axioms that have already been synthesized.

For soundness, VC generates an SMT query checking whether $\psi \wedge \neg\forall P.\forall w.\mathsf{ml2fo}_w(\varphi)$ is satisfiable, where P stands for a sequence of unary relation symbols corresponding to propositions in φ. This is equivalent to $\psi \wedge \exists P.\exists w.\neg\mathsf{ml2fo}_w(\varphi)$. As discussed in Section 5.5.2,

Table 5.2: Synthesis results for modal logics. For each logic, we synthesized modal axioms from the first-order description of the logic (see Table 5.1 for the formulae used). The Reference Axioms column shows canonical axioms studied in the literature; the LN column shows the time taken by the learner $\mathsf{Learner}_{\mathcal{W}}$; the CX column shows the time taken by the counterexample generator Cex; the SN column shows time taken by the soundness checker VC; the CM column shows the time taken by the completeness check. Times are in seconds.

Logic	**FO Desc.** (See Table 5.1)	**Synthesized Axioms**	**Reference Axioms**	**LN**	**CX**	**SN**	**CM**
(M)	Reflexive	$\neg\alpha \vee \Diamond\alpha$	$\Box\alpha \to \alpha$	9.9	0.00	0.06	0.03
(4)	Transitive	$\Box\alpha \to \Box\Box\alpha$	$\Box\alpha \to \Box\Box\alpha$	22.7	0.01	0.03	1.01
(B)	Symmetric	$\Diamond\Box\alpha \to (\Diamond\alpha \to \alpha)$	$\alpha \to \Box\Diamond\alpha$	25.0	0.01	0.09	0.05
(5)	Euclidean	$\Diamond\Box\alpha \to \Box\alpha$	$\Diamond\alpha \to \Box\Diamond\alpha$	29.7	0.01	0.13	-
($\Box$M)	Shift Reflexive	$\Box(\alpha \to \Diamond\alpha)$	$\Box(\Box\alpha \to \alpha)$	27.7	0.01	0.14	0.04
(D)	Serial	$\Diamond\top$	$\Box\alpha \to \Diamond\alpha$	26.9	0.00	0.03	0.04
(C)	Convergent	$\Diamond\Box\alpha \to \Box\Diamond\alpha$	$\Diamond\Box\alpha \to \Box\Diamond\alpha$	23.4	0.05	0.07	-
(CD)	Functional	$\Diamond\alpha \to \Box\alpha$	$\Diamond\alpha \to \Box\alpha$	46.8	0.01	0.04	0.06
(C4)	Dense	$\Diamond\alpha \to \Diamond\Diamond\alpha$	$\Box\Box\alpha \to \Box\alpha$	24.2	0.01	0.13	0.05
(K45)	(4)+(5)	$\Diamond\Diamond\alpha \to \Diamond\alpha$ $\neg\Diamond\alpha \vee \Box\Diamond\alpha$	$\Box\alpha \to \Box\Box\alpha$ $\Diamond\alpha \to \Box\Diamond\alpha$	27.0	0.01	0.15	0.82
(KB5)	(B)+(5)	$\neg\alpha \vee \Box\Diamond\alpha$ $\Box\Diamond\alpha \vee \Box\neg\alpha$	$\alpha \to \Box\Diamond\alpha$ $\Diamond\alpha \to \Box\Diamond\alpha$	17.7	0.00	0.14	43.06
(D4)	(D)+(4)	$\Box\Diamond\alpha \to \Diamond\alpha$ $\Diamond\Diamond\alpha \to \Diamond\alpha$	$\Box\alpha \to \Diamond\alpha$ $\Box\alpha \to \Box\Box\alpha$	55.2	0.01	0.33	0.17
(D5)	(D)+(5)	$\Diamond\top$ $\Box\neg\alpha \vee \Box\Diamond\alpha$	$\Box\alpha \to \Diamond\alpha$ $\Diamond\alpha \to \Box\Diamond\alpha$	30.8	0.01	0.19	11.66
(D45)	(D)+(4)+(5)	$\Diamond\top$ $\Box\Diamond\alpha \vee \Box\neg\alpha$ $\Diamond\Diamond\alpha \to (\Box\alpha \vee \Diamond\alpha)$	$\Box\alpha \to \Diamond\alpha$ $\Box\alpha \to \Box\Box\alpha$ $\Diamond\alpha \to \Box\Diamond\alpha$	43.5	0.01	0.15	6.41
(DB)	(D)+(B)	$\alpha \to \Diamond\Diamond\alpha$ $\neg\alpha \vee \Box\Diamond\alpha$	$\Box\alpha \to \Diamond\alpha$ $\alpha \to \Box\Diamond\alpha$	36.2	0.01	0.18	0.05
(M4)	(M)+(4)	$\Diamond\alpha \vee \neg\alpha$ $\Diamond\Diamond\alpha \to \Diamond\alpha$	$\Box\alpha \to \alpha$ $\Box\alpha \to \Box\Box\alpha$	19.3	0.00	0.10	0.74
(M5)	(M)+(5)	$\Diamond\alpha \vee \neg\alpha$ $\Box\Box\alpha \vee \Diamond\neg\alpha$ $\Diamond\Box\alpha \to \alpha$	$\Box\alpha \to \alpha$ $\Diamond\alpha \to \Box\Diamond\alpha$	13.1	0.00	0.14	-

the translated formula $\mathsf{ml2fo}_w(\varphi)$ replaces propositions with uninterpreted unary relations, and thus the quantification of P can be removed, giving the formula $\psi \wedge \exists w.\neg\mathsf{ml2fo}_w(\varphi)$ in the first-order theory of equality and uninterpreted relations. If this formula is not satisfiable, then φ is sound; otherwise, we proceed to generate a counterexample as in Algorithm 5.1.

For independence, VS queries the SMT solver to find a bounded frame witnessing $A \not\models \varphi$. If such a bounded frame exists, then the axiom is *independent*, and we discard the proposal otherwise.

We optimize the implementation for this domain by merging the SMT query for checking independence and the SMT query for synthesis (i.e. Learner). This is sound since we proceed with a candidate axiom only if both queries are satisfiable. We found that not combining these queries resulted in the synthesis of a large number of candidates that were immediately ruled out by the independence checker. This essentially makes the algorithm enumerate all semantically equivalent variants of a formula and it resulted in performance similar to naive enumeration (which we show is less efficient in our evaluation below). The use of a single query also results in only a small number of non-independent axioms being proposed, as we show in our evaluation (see Figure 5.4).

For counterexamples, Cex also queries the SMT solver to find a bounded frame satisfying ψ but not φ. In our evaluation, we use a size bound of 4. Note that since the counterexample frames are bounded, VS and Cex are guaranteed to terminate. However, VC tries to decide a first-order sentence using a semi-decision procedure so it may not terminate. The *fuel* parameter determines the timeout (Section 5.4.2) in this case.

Evaluation Results We attempted 17 historically-studied classes of modal logic frames, and we were able to synthesize complete axiomatizations for all of them. Our completeness procedure in Section 5.5.2 verifies automatically that the axiomatization is complete with respect to the FO description for 14 classes. We manually verified the completeness for the other 3 classes.

We used a laptop with a 4-core (8-thread) Intel CPU i7-8550U and 16 GB of memory. Table 5.2 shows the results of our evaluation with the synthesized axioms, reference axioms studied in the literature, and the breakdown of time spent in each component. Figure 5.4 shows the distribution of candidate axioms proposed for each modal axiomatization, including unsound proposals.

In our evaluation, we considered grammars that allowed all possible modal logic formulae with one proposition, and the height of formulae was increased incrementally until 3. Counterexample frames were bound to at most 4 worlds.

In all of the cases, the synthesizer effectively produces the complete axioms in less than a

minute. We believe that the main reason for this effectiveness is that we have an ideal type of counterexample in modal logic: extremely small frames are already able to characterize a class with a first-order description.

The axiomatizations synthesized by the tool, modulo small syntactic equivalences, correspond closely to the reference axiomatizations we see in the literature (see Table 5.2). For the class of serial frames, the tool found the axiom $\Diamond\top$ while the reference axiom is $\Box\alpha \to \Diamond\alpha$. We verified that the tool's simpler axiom is indeed sound and complete. There may be aesthetic reasons why humans avoid axioms with constants such as $\top$, and there are several other frames where our tool generates axioms with constants.

For classes that are characterized by conjunctions of multiple first-order properties, note that our tool does not know this fact, and often synthesizes a different set of axioms than the union of the axioms for each property, such as (M5).

The completeness procedure failed in some cases. Recall that for completeness, we need to check the validity of the second-order condition 5.14, which we handled using an incomplete reduction to condition 5.15. But this reduction is not enough for some of the cases we evaluated. For example, to prove completeness for the convergence axiom $\varphi = \Diamond\Box\alpha \to \Box\Diamond\alpha$ (corresponding to the first-order description $\psi = (R(x,y) \wedge R(x,z)) \to \exists w.R(y,w) \wedge R(z,w)$), one has to show that for any non-convergent frame (satisfying $\neg\psi$), we can find a valuation V such that $\neg\varphi$ holds for some world w. To do this, we have to first find a witness to $\neg\psi$, i.e., a world w and two worlds v_1 and v_2 with $R(w,v_1)$ and $R(w,v_2)$, such that v_1 and v_2 have disjoint successors in R. Then we can pick an atomic proposition α, and assign α to all successors of v_1 (which may be infinite), and $\neg\alpha$ to all successors of v_2. Then φ is not true at w because $\Diamond\Box\alpha$ is true, but $\Box\Diamond\alpha$ is false. This proof requires us to find a valuation that may vary on an infinite set of worlds (all the successors of v_1 and v_2) as opposed to the finite-varying valuation in the reduction to condition 5.15.

Comparison with Brute-force Enumeration We evaluated a synthesis procedure that uses brute-force enumeration without counterexamples for synthesis rather than constraint solving with counterexamples. In terms of our core algorithm (see Algorithm 5.1), this corresponds to implementing VC (needed for soundness) and VS (needed for independence checking), but skipping Cex and having the Learner component simply enumerate formulae.

In this evaluation, Learner enumerates all modal logic formulae up to height 3 (which is the maximum height used in our previous evaluation). For each enumerated candidate, we perform the soundness check using VC and the independence check using VS as before.

We ran the enumerative procedure for all 17 modal axiomatizations (the enumeration is naive and does not rule out any kind of symmetries), on a server with 32 Intel Xeon 8124M

CPUs and 72 GB of memory. We used a timeout of 3 hours for each modal axiomatization setting.

In all 17 settings, brute force enumeration was unable to exhaustively search the space for axiomatizations, while our technique was able to exhaustively search the space within a minute. However, one can argue that exhausting the search space is unnecessary if the synthesizer can produce a provably complete axiomatization earlier. This would require running the completeness checker each time an axiom is added. Indeed, enumeration does produce provably complete axiomatizations for 13 cases early on, and the time to reach such a complete axiomatization was less than a minute in 5 cases and about 8-11 minutes in the other 8 cases. Note that this does not account for the additional time spent checking completeness with each discovered axiom. However, for the 4 remaining settings where completeness cannot be checked automatically, an enumerative tool that implements completeness checks with each discovered axiom would have to run until the timeout of 3 hours. More importantly, the enumerative tool's final axiomatizations (terminated at 3 hours) were not complete for two settings ((C) and ($\Box$M)), whereas our tool found complete axiomatizations for all settings within a minute.

This evaluation suggests that although the synthesized axioms in each setting are short, brute force enumeration is not promising and is unlikely to scale to more complex settings, while using counterexamples to guide search with constraint-solving is significantly faster. Moreover, early termination is not possible when completeness checks fail, and exhaustive enumeration takes prohibitively long in these cases.

5.6 AXIOMATIZING LANGUAGES WITH KLEENE STAR

In this section, we instantiate the LAS framework to find axioms for *languages with Kleene star*. Each model in this case consists of a set of languages over a finite alphabet that is closed under the usual operations from formal language theory. Section 5.6.1 reviews some background from language theory, Section 5.6.2 discusses nuances for this setting, and Section 5.6.3 describes our implementation and results.

5.6.1 Language Models

Languages of finite words and the operations of concatenation, union, and Kleene closure are fundamental concepts in computer science. Much work went into the discovery of axioms for reasoning with these concepts (e.g. [94, 95, 96]), and we are interested in discovering such axioms *de novo*. In particular, we want to axiomatize a class of algebraic structures over

the signature $\tau = (\cdot, \mathit{plus}, *, 1, \mathit{zero})$, which we refer to as the class of *language models*. The symbols $\cdot$ and *plus* are binary function symbols corresponding to concatenation and union, $*$ is a unary function symbol corresponding to Kleene closure, and 1 and *zero* are constants corresponding to the singleton language containing the empty word and the empty language, respectively. The domain of a language model over an alphabet Σ consists of languages of words over Σ, i.e., $D \subseteq \mathcal{P}(\Sigma^*)$. We review the standard language theory interpretations for τ-symbols below.

Let x, y be languages over Σ. The operation $\mathit{plus} : D \times D \to D$ is interpreted as union:

$$x \mathit{\ plus\ } y \coloneqq x \cup y \tag{5.17}$$

The operation $\cdot : D \times D \to D$ is interpreted as the concatenation of languages:

$$xy = x \cdot y \coloneqq \{w_1 w_2 \mid w_1 \in x, w_2 \in y\} \tag{5.18}$$

where $w_1 w_2$ denotes the concatenation of words defined in the usual way. The n-fold concatenation of a language x with itself is given by:

$$x^0 \coloneqq \{\epsilon\} \qquad x^{n+1} \coloneqq x^n x \tag{5.19}$$

where ϵ is the empty word. The constants *zero* and 1 denote the empty language $\emptyset$ and the singleton language $\{\epsilon\}$, respectively. Finally, the operation $* : D \to D$ forms the closure of a language under concatenation with itself:

$$x^* = *(x) \coloneqq \bigcup_{i \in \mathbb{N}} x^i \tag{5.20}$$

For a fixed alphabet Σ, we refer to these models as Σ-*language models*. We write t_i^M for the language denoted by t in a model M under variable assignment i. For example, if M contains the languages $a^*, \{\epsilon\}$, and $\emptyset$, then for an assignment i with $i(x) = a^*$ and $i(y) = \{\epsilon\}$ we have $(xy)_i^M = a^*$.

Note that language models need not consist of regular languages, and many equations hold in some models and not others. For instance, we may have a model consisting of the context-free language $L = \{a^n b^n : n \in \mathbb{N}\}$, as well as the languages formed by closing the domain under concatenation, union, and Kleene closure (and also adding the languages for 0 and 1). In this model, it happens that the equation $xy = yx$ is true. But of course, this is not true for all language models: for example, we can take x to be the language $\{a\}$ and y to be the language $\{b\}$ in any model that contains those languages.

5.6.2 Instantiating the LAS Framework for Language Models

We now describe how the main components of the framework are instantiated to find equational first-order logic axioms for language models. Following the discussion above, we can identify the class $\mathcal{S}$ to be all models over the signature $\tau = (\cdot, \mathit{plus}, *, 1, \mathit{zero})$, with $\mathcal{C}$ consisting of all language models over finite alphabets, as defined in setting we make use of only basic knowledge about the target class, which we discuss next.

It so happens that the equational theory (the set of all true universally-quantified equations) of the class of Σ-language models coincides with a distinguished language model called Reg_Σ, whose domain consists of all regular languages over Σ. The problem of axiomatizing the equational theory of Reg_Σ has a long history. It was first posed by Kleene [102], with several contributions toward axiomatization (e.g., [95, 96, 97]), and culminated in Kozen's finite axiomatization consisting of conditional and unconditional equations that were proven complete for the equational theory [94]. Though we do not aim *a priori* to rediscover precisely that axiomatization, it informs our choice to focus on finding *equational* axioms. Thus the set of formulae $\mathcal{F}$ consists of universally-quantified equations in first-order logic over the signature τ. We invite the reader in the remainder of this section to start fresh and naively explore what is necessary to find axioms in this setting, and in particular, how to build VC and Cex.

Instantiating VC. Our goal is to build a procedure that checks whether a candidate equational axiom is true in the class $\mathcal{C}$ of language models. Suppose in particular that we want to prove that a (universally-quantified) equation in n variables is true for all language models (over arbitrary finite alphabets Σ). Assume we have a formula $\forall x.t(x) = t'(x)$, for some τ-terms t, t' and a sequence of variables x, and we want to prove that for any language model M, we have $t = t'$ for any assignment of variables to languages from the domain of M. Observe first that it is *necessary* that $t = t'$ holds when each variable x_i is assigned the singleton language $\{x_i\}$, where we treat each variable as a distinct alphabet symbol. If not, then the equation does not hold in any language model containing these n singleton languages $\{x_i\}$. Call this singleton assignment s.

It turns out this condition is also *sufficient* for an equation to be true in all language models (observed by Gischer [103], see also a detailed proof in [104]). The argument runs as follows. Suppose for contradiction that the equation is true under the singleton assignment s in some suitable model N, but the languages denoted by t and t' are different for another assignment i in a (possibly different) model M. Then, without loss of generality, we can assume there is a word $w \in t_i^M$ and $w \notin t_i'^M$. Since the equation holds under s in N, the terms t, t' must generate the same words over x when treated as regular expressions over the

alphabet x. It follows by a straightforward induction on t that membership of a word in the language t_i^M is witnessed by such a word over x (say w is witnessed by w_x). That is, the language t_i^M has the form:

$$\bigcup_{w_x \in t_s^N} i(w_x) \tag{5.21}$$

Membership of the word w in t_i^M is witnessed by $w \in i(w_x)$ for some word $w_x \in t_s^N$, but this is a contradiction because $w_x \in t_s'^N$ and thus $w \in i(w_x) \subseteq t_i'^M$. Thus, for equational axioms, the VC component can simply check that t and t' are equivalent as regular expressions over x.

The preceding observation reduces the soundness of equational axioms to the equivalence of regular expressions. We note that Kozen's complete axiomatization [94] involves two conditional equations. Building VC for conditional equations is more difficult, as it would likely require proofs by induction. We leave such automation to future work, but note that it does not fall outside the scope of the framework.

Instantiating Cex. It follows from above that the counterexample generator for false equations over language models can always provide as a counterexample a finite prefix of a canonical language model. For example, for the false equation $xy = x$, any language model over $\Sigma = \{a, b\}$ that has the languages $\{a\}, \{b\}$, and $\{ab\}$ witnesses that the equation is false. Of course, such a model must also contain the languages $\{aa\} = \{a\} \cdot \{a\}$, $\{bb\} = \{b\} \cdot \{b\}$, and many others (it must be closed under the operations). Intuitively, the counterexample models, though infinite, can be witnessed finitely. As discussed in Section 5.4, we formalize this with the concept of *pseudo-models*, and in this context *language pseudo-models*.

Definition 5.7 (Language pseudo-model)**.** A *language pseudo-model* over an alphabet Σ is a model over $\tau = (\cdot, +, *, 1, 0)$. Its domain D is finite and consists of Σ-languages, and each operation is a partial function on the domain. For every $x \in D^i$, each operation f of arity i is either undefined or else $f(x)$ is the language given by the standard interpretation from language theory.

The counterexample generator Cex produces language pseudo-models as counterexamples, and the Learner can propose any equation that is not false in the pseudo-models it has seen so far, with satisfaction defined as follows.

Definition 5.8 (Satisfaction in a language pseudo-model)**.** An equation $t = t'$ is *false* in a language pseudo-model M just when t_i^M and $t_i'^M$ are both defined and $t_i^M \neq t_i'^M$ for some variable assignment i. Otherwise, $t = t'$ is *true* in M (or *satisfied* by M), written as $M \models_p t = t'$.

For every language pseudo-model M and equation φ, we have:

$$\mathcal{M}(M) = \mathit{extensions}(M) \tag{5.22}$$

$$\text{and } \varphi \in \mathcal{W}(M) \Leftrightarrow M \not\models_p \varphi \Leftrightarrow \forall M' \in \mathcal{M}(M),\ M' \not\models \varphi \tag{5.23}$$

where $\mathit{extensions}(M)$ denotes all language models that contain the domain of M and that agree with the operations of M wherever they are defined. As an example, if the Learner proposes the false equation $xy = x$, then Cex may produce a pseudo-model of size 3 with domain $D = \{\{a\}, \{b\}, \{ab\}\}$ and an interpretation of concatenation such that $\{a\}\cdot\{b\} = \{ab\}$ and all other operations on all other elements are undefined. Such a pseudo-model is enough to show the equation $xy = x$ is false using an assignment that maps x to $\{a\}$ and y to $\{b\}$. Note that this pseudo-model does not rule out, for example, the false equation $xx = x$, because xx is undefined for every x.

Instantiating VS. Unlike for modal logic, validity in the class $\mathcal{S}$ can be stated directly in first-order logic, and thus VS is instantiated as a semi-decision procedure for first-order validity. Given axioms $\varphi_1, \ldots, \varphi_n$, checking that a candidate axiom φ is independent from the axioms φ_i amounts to checking that $\psi \coloneqq \bigwedge_i \varphi_i \to \varphi$ is not valid, or equivalently, that $\neg\psi$ is satisfiable. Since satisfiability is hard to tackle directly, we instead choose to take *failure to prove dependence* as a proxy for *independence*. We use VS to attempt a proof of ψ. If the proof fails we add the axiom φ to the growing set of axioms, and otherwise we discard it.

Completeness. It is known that the equational theory of language models has no complete *finite* axiomatization in terms of only equations [97]. And as mentioned, Kozen's complete axiomatization [94] involves two conditional equations. The proof of completeness relies on the uniqueness of minimal deterministic finite automata for regular languages and involves algebraically encoding the determinization and minimization constructions for such automata. Automating the discovery of such a proof is very difficult and beyond the scope of this chapter.

5.6.3 Implementation and Evaluation

We implemented this instantiation of the framework following the general algorithm (Algorithm 5.1). Using this algorithm, we obtain a synthesizer for sound equations over language models with the operations of concatenation, union, and Kleene star. Our artifact can be found at: https://zenodo.org/records/7072506.

Implementation Details The implementations of VC, VS, and Cex are based on the SMT solver Z3 [11]. We discuss their implementation details in this section.

Suppose we have already synthesized a set A of (universally-quantified) equations, and suppose that $t = t'$ is a candidate equation generated by Learner. VC employs the decision procedure described in Section 5.6.2 to check the validity of $t = t'$ by checking the equivalence of two regular expressions representing t and t'. We check the equivalence of regular expressions by encoding an SMT query over the theory of strings and using Z3. If $t = t'$ is not valid, Cex generates a large enough pseudo-model as described in Definition 5.7. We pre-compute a finite portion of the canonical model of regular languages with Kleene star corresponding to small regular expressions. When an axiom cannot be proven valid we look up this pseudo-model for an instantiation that witnesses the non-validity of the candidate.

For VS, we use a procedure called natural proofs [27] to check the entailment $A \models t = t'$. Natural proofs are semi-decision procedures for checking the validity of first-order formulae using systematic quantifier instantiation. VS instantiates A with ground terms up to a certain height and generates an SMT query whose unsatisfiability would imply $A \models t = t'$. This query is a quantifier-free formula over the theory of uninterpreted functions, and hence the SMT solver is guaranteed to terminate. If the query is satisfiable, it may still be the case that $A \models t = t'$, but we treat it as if $t = t'$ is independent from A to avoid missing axioms.

We optimized our algorithm for this domain in our implementation by merging the SMT query by VS and the SMT query by Learner (line 4 of Algorithm 5.1). This optimized version is equivalent to the original algorithm since the core algorithm only proceeds with a candidate equation if both queries are satisfiable.

Evaluation Results Recall that our signature is $\tau = (\cdot, \mathit{plus}, *, 1, \mathit{zero})$. We ran three passes of the algorithm to synthesize equations with increasingly larger term grammars:

1. τ-terms of height 1 with 2 free variables,
2. τ-terms of height 2 with 2 free variables, and
3. τ'-terms of height 2 with 3 free variables, where τ' is τ without the constants 0 and 1.

The set of axioms found in each pass is first pruned for redundant axioms that may be entailed by others in the set. Recall that our independence check using VS only ensures that axioms that are proposed later are not entailed by those that were proposed earlier; the converse may not be true. This additional pruning is done using the first-order theorem prover Vampire [90]. We treat the symbols in the signature as uninterpreted functions and ask whether any axioms in the set are entailed by the others. The final set of axioms after

Table 5.3: Synthesis statistics for language models.

	# of Axioms		Time (seconds)			
Pass	**New**	**Pruned**	**Synthesis**	**Pruning**	Cex	**Total**
1	12	3	0.6	0.6	0.4	1.6
2	25	14	136.4	88.4	227.4	452.2
3	12	17	1821.5	70.0	760.2	2651.7

pruning in each pass is used as an initial set of axioms in subsequent passes. Note that in this evaluation we run the core algorithm (Algorithm 5.1) multiple times, i.e., once for each pass.

Table 5.3 presents some statistics about our evaluation, including the number of axioms synthesized in each pass, the total time taken in each pass, and a breakdown of the time spent per component. The evaluation was performed on a machine with a 4-core (8-thread) Intel CPU i7-8550U and 16 GB of memory. Our tool synthesizes the following axioms (post pruning after all passes):

1. $0 = 0b$
2. $0 = b0$
3. $0^* = 1$
4. $0^* + b^* = (1 + b)^*$
5. $00 + (a + b) = (b + a) + a$
6. $(b^*)^* = b^*(b + 1)$
7. $b^* + (1 + b) = b^*$
8. $(a + a)a^* = a^*a$
9. $(1a)(1 + b) = ab + a$
10. $a + a = a$
11. $(b + b)^* = b^*b^*$
12. $b + (c + a) = (c + a) + (a + b)$
13. $(ac)b = a(cb)$
14. $(a + a)(b + c) = ac + ab$
15. $cb + ab = (c + a)b$

The axioms for this class of models are expected to be similar to those of Kleene algebras from the literature, and we compared them to Kozen's axioms [94] (see also [95]). The axioms synthesized by our tool are quite different from these reference axioms. As noted in Section 5.6.2, however, Kozen used *two* kinds of axioms: equations and conditional equations (axioms formulated as inequalities can be reformulated as equations). Handling soundness for conditional equations is more complex, and we haev not tackled this in our work.

Kozen's axioms are *complete* for the equational theory of regular languages under concatenation, union, and Kleene star (all valid equations are semantically entailed by Kozen's axioms). Since our axioms are valid on the same class, Kozen's axioms imply our axioms by completeness.

On the other direction, it turns out that if we consider only the unconditional equational axioms in [94, Section 2, Axioms (3) - (15)], then our axioms are stronger. That is, our axioms imply all of the unconditional equational axioms in [94], but the converse is not true.

We used Vampire [90] to automatically verify that our axioms (3), (4), (6), (8), and (11) are not implied by the unconditional equational axioms in [94], and Vampire was able to produce finite counterexample models. This result shows that our technique has the potential to discover new and useful axioms. The equational axioms discovered in our work may already have applications; there are several rewrite engines and solvers that use equational axioms to reason with regular expressions where this expanded set of axioms could be useful.

Comparison with Brute-force Enumeration Similar to the experiments for modal logics, we tried to use a brute-force enumerative Learner, instead of a constraint-solving-based one that learns from counterexamples. The enumerative version took about 25 hours to exhaust the axiom search space, while our tool took only 50 minutes. The enumeration scanned through ∼10 million equations with height-2 terms and 2 free variables (corresponding to pass 2), and ∼1.3 million equations with height-2 terms and 3 free variables (corresponding to pass 3). Note that there are more equations in pass 2 than in pass 3 because pass 3 does not have constant terms 0 and 1 in the signature. Note also that since no finite set of equational axioms can be complete, early termination on completeness is impossible in this setting. We again conclude that enumerative approaches are unlikely to scale for axiom synthesis.

Chapter 6: Related Work and Discussion

In this chapter we discuss prior work in several key areas relevant to this thesis. Each of these areas have gigantic footprints in literature that exceed the scope of our purpose here, so this chapter merely serves to set the context for the design decisions we made in our work.

6.1 HEAP VERIFICATION: LOGICS AND REASONING

We begin with an overview of logics and reasoning approaches for verifying heap manipulating programs, which has been one of the primary domains of study in our work for pursuing the vision of eliminating expert creative help in automated reasoning.

There have been mainly two paradigms to automated verification of programs annotated with rich contracts written in logic. The first is to restrict the specification logic so that verification conditions fall into a decidable logic. The second allows validity of verification conditions to fall into an undecidable or even an incomplete logic (where validity is not even recursively enumerable), but nonetheless support effective strategies using various heuristics and sometimes external help from the programmer for verification.

Decidable Logics There is a rich body of research on decidable logics for heap verification: first-order logics with reachability [105], the logic LISBQ in the HAVOC tool [106], several decidable fragments of separation logic known [107, 108, 109, 110, 111, 112, 113, 114, 115] as well as fragments that admit a decidable entailment problem [116]. The work based on EPR (Effectively Propositional Reasoning) for specifying heap properties [117, 118, 119] provides decidability, as does some of the work that translates separation logic specifications into classical logic [120]. Decidable logics based on interpreting bounded treewidth data structures on trees have also been studied, for separation logics as well as other logics [121, 122, 123]. In general, these logics are heavily restricted in order to obtain decidability.

Undecidable and Incomplete Logics There is extensive work on highly expressive heap logics. The key problem that all of them attempt to address is the so-called *frame problem*: if an operation does not modify an unbounded region of the heap, how can we effectively conclude that properties that only "involve" that region of the heap are preserved under the operation? This is an incredibly useful reasoning pattern for checking correctness of heap manipulating programs, and logics that seek to facilitate effective frame reasoning make different choices to define what it means for a property to only involve a certain heap

region and to logically express what it means for a program's operation to not modify a heap region. The most prominent among these (at the time of writing this thesis) is Separation Logic [60, 69, 70, 124] which defines a heap semantics for formulas in the logic, wherein sub-heaps (i.e., heaplets) are associated with formulas whose truth value only requires looking up the relations represented in that particular heaplet. The logic also provides a separating conjunction operator $*$ which expresses that two operand formulae that it conjoins are associated with disjoint heaplets. Then, if a program only modifies one of the heaplets, we can conclude that the formula corresponding to the other heaplet continues to hold.

Separation Logic can be described as having *implicit* heaplets, since one does not represent memory regions as logical objects directly, rather formulae are used to refer to the properties witnessed by various memory regions. Other logics make different choices. For example, Dynamic Frames [125, 126] and related approaches like Region Logic [127, 128, 129] allow users to explicitly specify heaplets in the logic in the form of variables that represent sets of heap locations or variables that model heaplets themselves as maps (corresponding to each pointer) between heap locations. This allows verification engineers finer grained expressive power to talk about heaplets.

The work on Implicit Dynamic Frames [130, 131, 132, 133] defines a variant of Separation Logic that combines the ideas of implicit heaplets in separation logic and explicitly accessible heaplets in dynamic frames. The work on Natural Proofs [49, 50, 56] takes a similar approach, but is a variant of First-Order Logic with Recursive Definitions (FORD). The implicit heaplets are defined only for recursively defined functions and are given manually by the specification/verification engineer. The heaplets themselves are recursively defined: a recursively defined function H_R corresponding to each recursively defined function R. The work on Frame Logic [19, 20] goes further, defining an extension of FORD with a *Support* operator $Sp(\alpha)$ to represent the heaplet of any FORD formula α. The Sp operator has a fixed semantics, and Frame Logic therefore also has implicit heaplets. The work in [134] is very close in spirit to Frame Logic, but only defines support for a restricted subset of formulas. The uniquely defined heaplets in the above logics is interesting since determined heaplets are potentially more amenable to automated reasoning [50, 56, 135]. A technical point to note here is that the Separation Logic and its variants are typically defined over finite partial heaps, whereas Implicit Dynamic Frames and Frame Logic are defined over total heaps. In particular, the semantics of Frame Logic is defined over both finite and infinite heaps. The formal semantics of Implicit Dynamic Frames is less clear, although there have been attempts to formalize it and in fact connect it to Separation Logic [136].

The work on parametric shape analysis [137] takes an entirely different approach: it uses a variant of FORD as the logic for expressing properties, but it does not model the heap

precisely, instead defining a three-valued semantics [138] for the logic and representing the heap as a finitary structure that corresponds to (induces) a particular three-valued semantics for formulas. This work has also been extended to handle inductively defined predicates [139].

While the above logics deal with expressing rich properties (typically data structures and various measures over them like length, height, etc) using recursion, there are several others that instead use a form of quantification. Our work on intrinsic definitions of datastructures [18] covered in Chapter 4 is of this kind. Other works like Implicit Dynamic Frames [130, 132] provide a programmatic construct whose semantics is that of an implicit quantifier over the operand formula. Some separation logics provide an iterated separating conjunction [60, 140, 141] and some variants of this kind of quantification have also been explored [142, 143].

Frame reasoning as we have described it above is itself not the only way to conceive the key mathematical burden in reasoning with unbounded heap changes, and in fact there is work on designing logics around variants of frame reasoning [144, 145, 146].

Reasoning Specifying rich properties of heap structures typically leads to undecidability or incompleteness for checking validity: the general entailment problem for separation logic with inductive predicates is undecidable [147], validity of first-order logic with recursive definitions is not even recursively enumerable (i.e., is incomplete), and the magic wand in separation logic has the expressive power of second-order logic [148].

There are two major approaches. The first one is to use a specialized proof system for the logic and saturate the entailment to check with respect to the proof rules. Early work on program verification with Separation Logic annotations used symbolic execution to transform verification problems into entailment [110] and provided a proof system to decide the fragment of entailment problems generated by the symbolic execution. There are several other works that use different techniques for proof search [149, 150]. A very different approach in this realm is the idea of cyclic proofs [151, 152, 153, 154].

The second approach is much more popular, and involves translating heap logics to other logics that have robust automation— typically First-Order Logic, via first-order theorem provers [90], SMT solvers [11, 12], and other FOL-based intermediate verification language frameworks like Why3 [155, 156] and Boogie [81, 157]. Tools such as Viper [158, 159, 160] provide a general intermediate language for permission logics and handle automation using a variety of back-ends, including SMT solvers. The core logic of Viper is essentially that of implicit dynamic frames, and has been used to implement automation for implicit dynamic frames by works such Chalice [133]. There is also work on translating VeriFast [161] predicates into Implicit Dynamic Frames [162]. Other works translate proof obligations into first-order

logic (either by translating entailments from the base logic or generating verification conditions in FOL) and use reasoning mechanisms for FOL [19, 20, 27, 41, 49, 50, 56, 120, 139, 157, 163, 164]. Works that use proof system/ search based automation also typically delegate reasoning about data values to SMT solvers [165].

This section would be incomplete without mentioning the importance of SMT solvers themselves, which are the most reliable form of automation that we have today. The software artifacts produced in this thesis rely heavily on the automation provided by SMT solvers, and our analysis of creativity gaps in Chapter 2 formally modeled SMT solvers as a subroutine of the heuristic whose limitations we wanted to study. SMT solvers [11, 12, 166] provide powerful automation for reasoning with quantifier-free fragments of decidable combinations of theories [28, 29, 30, 167]. The construction of decidable combinations of theories is its own line of research [168, 169, 170, 171, 172, 173, 174, 175, 176, 177, 178, 179].

6.2 REASONING ABOUT UNBOUNDED STRUCTURES: HEURISTICS AND CREATIVE HELP

The core technical problem in the domains of study undertaken in this thesis (Chapters 2- 4) is the specification and verification of programs over unbounded structures: heap datastructures and algebraic datatypes (e.g., unbounded ADT lists). We also studied axiomatizations for Kleene Algebras and Kripke Frames, which are unbounded structures. When talking about unbounded structures, logics typically resort to either using quantification or recursion. Both of these make the resulting logics hard to reason with. As mentioned earlier in this section, First-Order Logic with Recursive Definitions is not recursively enumerable, i.e., no sound proof mechanism can prove 100% of all valid theorems in the logic. Similarly, quantified combinations of FO theories are typically undecidable (but they are recursively enumerable, by Gödel's Strong Completeness Theorem on entailment with respect to a recursive set of premises). As a result, developers of tools for automating reasoning over such logics often implement heuristics that may not always succeed. We now discuss some of the techniques relevant to this thesis and the ways in which creative help is solicited from the user to improve the effectiveness of heuristics. We limit our discussion to first-order logics with and without recursion.

Reasoning with Quantification There are several ways to reason with quantified formulas, including quantifier elimination [28, 39, 57, 180, 181], superposition [90, 182], tableaux-based proofs [183, 184], and quantifier instantiation. However, the technique most commonly used along with SMT solving and the most relevant to this thesis is quantifier instantiation.

Many proofs, especially when theorems are stated at the level where users interact with them, can be seen as quantifier instantiation followed by some simple reasoning. In terms of automation, one must figure out relevant instantiations of the various quantified formulas in the query (typically given in the form of premises or axioms), and once this is done the rest of the proof is easily automated.

Techniques for quantifier instantiation use a mix of search heuristics along with modalities of soliciting the user for creative help to narrow the search space of likely instantiations. E-matching, introduced by the Simplify theorem prover [185], is one of the earliest approaches and forms the basis of most of the quantified reasoning techniques today. It has since been implemented inside several SMT solvers and expanded to handle many theories [186, 187]. Other techniques include model-based instantiation [188], conflict-based instantiation [189, 190], and enumerative instantiation [191]. Some quantifier elimination algorithms can also be seen as providing a set of sufficient instantiations [192], and quantifier elimination based techniques have been studied even for undecidable theories [193]. One interesting technique that solicits user help is based on *triggers*. The idea is that the user specifies a pattern corresponding to a quantified variable, and the solver only instantiates that variable with a ground term occurring in the set of ground clauses if it matches the pattern (this is typically implemented as a way to guide an E-matching procedure, so ground term selection is done modulo a current set of equalities). SMT solvers as well as tools such as Boogie [81, 194] and Dafny [9] provide mechanisms for specifying triggers [195]. Trigger patterns can get very complex, and there has in fact been work on modeling the language of trigger patterns itself as a programming language [196]. However, it is well-known that trigger-based instantiation can be unpredictable and flaky, and techniques for inferring a more stable instantiation scheme from a user's preliminary trigger input have been explored [197]. One recent work investigates instantiation schemes whose output produces formulas in more expressive decidable logics such as EPR rather than quantifier-free logics [198]. However, this work only considers pure FOL (i.e., do not support background theories) and the approach is known to be incomplete.

One particular thread of related work that we have not covered above is the idea of *ghost code* as a mechanism of soliciting user help. We used ghost code in a crucial way in Chapter 4 to reduce the complexity of the verification problems handled by automation. This is a familiar idea in deductive program verification [77, 78, 79, 80] and is supported by verification tools such as Boogie and Dafny. Ghost code is code that manipulates auxiliary variables to perform a parallel computation with the original code without affecting the original code (for example, conditionals on ghost code that update original code variables are prohibited, etc.). As opposed to the techniques discussed above which typically deal with universal quantification, ghost code is typically used to eliminate existential quantification. Intuitively, ghost code

computes witness values that are useful for verification of assertions/post-conditions. For example, a method may start with a concrete data structure that refines an abstract datatype witnessed by a refinement map, and the programmer can establish a new refinement map at the exit of the method by constructing the refinement map using ghost code. In our work we specifically use ghost code to reduce verification to decidable logics, which is relatively uncommon. For example, the work on region logic [127, 128, 129, 157] allows users to write ghost code to reason about explicit footprints, but generates quantified verification conditions that are undecidable in general.

Reasoning with Recursion In our work we studied first-order logics with recursive definitions (FORDs) where we model a designated foreground sort and many background sorts each with their individual theories (arithmetic, sets, arrays, etc.). FORD is essentially the classical first-order logic with least fixpoints (FO+lfp) studied in finite model theory and database theory [59, 199, 200, 201, 202]. The only difference is that we give names to recursive definitions that have least fixpoint semantics. We assume that the definitions are monotonic, which guarantees that the least fixpoint exists [58] (in practice we use a slightly more expressive fragment where we stratify recursively defined symbols and only require monotonicity on recursive calls within the same stratum).

There is vast literature on automated reasoning with recursive definitions. In this thesis we primarily use and study unfolding-based techniques which unfold the body of recursively defined function symbols a few times and treat the symbols themselves as uninterpreted in the resulting formula. These are also called "unfold-and-match" heuristics. When these heuristics fail, the creative help solicited from users is usually information that helps an inductive proof of the property at hand. Unfolding-based techniques were pioneered by the NQTHM prover developed by Boyer and Moore [24] and its successor ACL2 [203, 204] had support for recursive functions and had several induction heuristics to find inductive proofs. Different systems vary in their determination of terms to unfold [41, 49, 50, 56, 205, 206, 207, 208, 209]. In particular we have discussed the unfolding algorithms of Liquid Haskell [25, 32] and Leon [33, 46] extensively in Sections 2.6 and 2.7. As the completeness results we know for unfolding techniques [17, 27] are fairly recent, we do not expect that many tool developers are aware of the incompleteness that may stem from not unfolding enough (which is a different from help needed for induction proofs). Note that the unfolding recommended by complete instantiation schemes may be too large for certain problems in practice, and tool developers may choose to implement cheap but incomplete instantiation heuristics anyway. Still, some language frameworks like Dafny [210], Verifast [211], and Liquid Haskell allow for unfolding (or folding) recursive definitions based on user suggestions.

There are several other approaches which we have not considered in detail here. One ongoing area of research involves decidable logics for recursive data structures [212]. Naturally, the expressive power of these logics is restricted in order to obtain a decidable validity problem. The work on cyclic proofs [151, 152, 154] also uses certain heuristics for reasoning about recursive definitions, and in particular recent work argues that in certain cases inductive lemmas required for bridging the gaps in unfolding based techniques can be obviated by the use of cyclic proofs [153]. There is also a significant amount of work in the use of Constrained Horn Clauses (CHCs). CHCs have been used to model verification problems [213, 214], and enable automated reasoning in practice via efficient algorithms integrated into SMT solvers [215]. However, CHC solvers are not typically equipped to deal with verification conditions with recursively defined function and predicate symbols, although recent work has begun to address this issue [216, 217]. Note that while CHCs are a fragment of FOL and therefore admit a recursively enumerable validity problem, verification problems are typically modeled as *satisfiability* of CHCs (more specifically $\mathcal{L}$-satisfiability: solutions must be expressible as a formula in a given logic $\mathcal{L}$), which is not recursively enumerable. Consequently, although they are certainly incomplete, the limitations of CHC solving algorithms has not been formally studied (to our knowledge). The possible modalities for user help is also not clear for many of the approaches discussed above.

Although we have discussed reasoning with quantification and recursion separately, as we showed in Chapter 2 it turns out that several heuristics for reasoning with recursive definitions are in fact performing essentially first-order reasoning, and many of them can be seen as a strategy for quantifier instantiation in an appropriate first-order logic. In all these cases the heuristics employed are necessarily limited (and these limitations are also typically encountered in practical verification problems).

6.3 IDENTIFYING LIMITATIONS OF HEURISTICS

To the best of our knowledge, prior work on modeling heuristics is somewhat limited. The fact that heuristics often make nuanced, complex choices that are typically not well-documented may be a contributing factor. Formally defining the settings under which they succeed or fail appears even harder. Our work in Chapter 2 in fact encountered these issues: the FLUID fragment of logic that we develop in Chapter 2 took us a year to identify! Identifying the precise fragment of logic that both enjoyed the desired completeness properties and was expressive enough to cover all known practical examples turned out to be nontrivial[1].

[1]Anecdotes are not scientific evidence, of course, but here is another one: the author of this thesis learned recently that the completeness of the unfolding scheme that Liquid Haskell uses was conjectured some ten

Still, we discuss here the smattering of related work that we believe shares the spirit of formally characterizing various heuristics. Note that while the idea of formally identifying limitations of heuristics is itself not new, the approach of separating them from the creative help provided by the user and studying them in that setting appears relatively uncommon.

Natural Proofs The closest related work is the work on theoretical foundations for Natural Proofs [27]. Natural Proofs [50, 56] is a technique used for verifying heap manipulating programs with respect to specifications containing recursive definitions. The verification conditions generated by the framework have both recursive definitions (used to model data structures and measures over them) and quantification (used to express frame reasoning and user-provided lemmas). The heuristic works based on abstracting the least fixpoint recursive definitions into fixpoints (which can be expressed using quantification), and performing a systematic quantifier instantiation on the resulting quantified first-order formula. Crucially, the heuristic only handles the instantiation for the uninterpreted 'foreground' sort and delegates theory reasoning to an SMT solver. The work in [27] defines a generalization of the heuristic implemented in the earlier works [50, 56] and identifies a *safe* fragment of many-sorted first-order logic that contains the verification conditions generated by the natural proofs technique (after abstracting the least fixpoint definitions into fixpoint definitions). The authors then show that the generalized heuristic is a complete procedure for validity of formulas in the safe fragment, i.e., if the abstracted first-order formula is valid, then the heuristic will definitely prove it. The only gap in reasoning is then between FOL and FO+*lfp*, which is bridged by user-provided inductive lemmas. Similar to our work in Chapter 2, this work also shows the existence of rogue nonstandard first-order models when the generalized heuristic fails to prove a property that is valid in FO+*lfp*.

Another key similarity is that both works consider the completeness of *thrifty* instantiation techniques that are effective in practice. This is important for two reasons. First, observe that first-order logics over combined theories (defined, say, as a recursive set of axioms) already admit a complete systematic quantifier instantiation procedure by Herbrand's theorem: you simply instantiate every quantifier with every term (also called the Herbrand universe). However, the Herbrand universe is too large to effectively explore in general. Consider for instance a validity problem stated over a background theory of arrays with select and store operations [71]. Herbrand's theorem requires that we at least explore instantiating the axioms of the array theory with all possible finitely constructible arrays. It also does not provide any

years earlier, but the answers were not known until our results in [17]. The theoretical tools to investigate the problem were developed over several years and the idea of looking at combined theories was only developed a few years after the conjecture in related work, which ultimately paved the way for our results.

information about which instantiations are more likely to work, only that if the given formula is valid then there is certainly a finite set of terms that is sufficient for the proof. This is clearly impractical. The second reason is that developers of these heuristics do not want to engage with background theories directly! They merely want to relax the quantification directly present in the validity query (namely over the foreground sort) using instantiation and use off-the-shelf SMT solvers to efficiently reason with background theories. There is no middle ground here: the work in [27] in fact shows that thrifty instantiation is not complete for multi-sorted FOL in general (see Example 4.2). It only works for certain fragments that happen to be expressive enough to contain practical benchmarks, and theory aimed at explaining the effectiveness of practical heuristics and identifying their limitations must identify such expressive fragments for which the appropriate thrifty instantiation works. Our work shares these qualities with the work in [27] and is heavily inspired by the theoretical tools that it develops.

At its core, our work in Chapter 2 asks the same questions as the work in [27] but studies a different logic. However, there are significant technical differences between the logics considered in these two works as well as their results. First, the foreground sorts in our setting are ADT sorts and not uninterpreted[2]. Second, the safe fragment identified in [27] is very restrictive as it disallows uninterpreted functions to involve background sorts, which in our setting would mean programs cannot have input parameters of the background sort, like integers. Finally, the quantifier instantiation strategy studied in [27] is much more liberal than in our work (and what tools like Liquid Haskell and Leon do). For example, if $\bar{t}$ is a set of terms that occur in a theorem, the instantiation in [27] will always instantiate the definition of f on $\bar{t}$ it, while we will do so only when $f(\bar{t})$ occurs in the theorem. Consequently, the proof of our main theorem is quite complex and fundamentally different from the proof of completeness in [27]. The two results are ultimately incomparable. In particular, it is not clear whether there is a more general completeness result that subsumes both fragments or if it is possible in practice to translate benchmarks in one logic into the other such that the resulting formulas fall into the complete fragment. We will revisit the issue of translating between logics when we discuss approaches for lemma synthesis in the next section.

[2]We note here a common confusion about our work that stems from conflating the pure first-order theory of ADTs with the combined theory consisting of ADTs, uninterpreted functions, and other background sorts. The pure first-order theory of ADTs has a complete recursive axiomatization [35, 36] which yields decision procedures that have been efficiently implemented in SMT solvers [11, 12, 37, 38, 218, 219]. The pure theory does admit nonstandard models, but they are indistinguishable from the standard model using a FOL formula. On the other hand, the addition of uninterpreted functions destroys decidability and gives rise to rogue nonstandard models in the combined theory which make reasoning more complex.

Completeness for Practical Heuristics The work in [41] shows that LEON-like reasoning (and UQFR in our work) is actually a decision procedure for certain restrictive logics. More precisely, it exhibits a logic over restricted classes of user-defined abstractions of ADTs to collections/measures in a decidable sort using catamorphisms, and shows that unfolding function definitions just once followed by quantifier-free reasoning is a decision procedure. The classes of such abstractions (*infinitely surjective and sufficiently surjective abstractions*) however are extremely semantically restrictive compared to FLUID. In particular, as we show in Section 2.8, validity of FLUID is undecidable, which argues this difference.

The work in [32] shows that the PLE heuristic implemented in LH is complete if an 'equational proof' exists, but this result is much weaker than ours, and in fact PLE fails to prove simple theorems that UQFR can prove. For example, with a definition of the form $R(x) \equiv \mathsf{ite}(x > 0, 1, 2)$ PLE cannot show $\forall x.\, R(x) > 0$ since it looks for a 'match' for the case splits in the definition and would then only unfolds the body of R for the corresponding case, whereas UQFR unfolds the definition of R regardless of match and can therefore show the property.

There has also been work on evaluating the completeness/incompleteness of heuristics empirically using a large suite of benchmarks [14].

Logics that Admit Complete Thrifty Instantiation Turning to logics, works on resolution such as set-of-support resolution [220, 221] and Stickel's theory resolution [222] define fragments for which thrifty instantiation techniques (especially those that avoid instantiating theory axioms) are complete. These works do not delegate background reasoning to SMT solvers and it is unclear whether they can be lifted into a resolution-modulo-SMT setting. The work in [223], implemented in the iProverEq prover [224] develops a complete instantiation scheme for a fragment of pure first-order logic with equality. Unlike our work and the work in [27] where the instantiation scheme is syntactic (based on ground terms occurring in the unfoldings), the instantiation in [223] is guided by models arising from satisfiability checks over certain sets of ground terms. The work in [188] identifies a fragment of single sorted FOL over a combination of uninterpreted functions and relations constrained by theories. This *essentially uninterpreted* fragment is shown to admit a thrifty complete instantiation procedure. However, this fragment is quite restrictive and cannot support the VCs generated by tools in practice. In general, most related works in this area do not cater to fragments of many-sorted FOL with background theories. One exception is the MBQI instantiation technique introduced in the same work [188], which is shown to be complete for fragments where one can ensure (by meta-arguments outside the system) that only a finite number of instantiations will be generated before a real model is found [174].

6.4 BRIDGING CREATIVITY GAPS USING LOGIC LEARNING

There are two guiding principles that we have used in this thesis to explore the automation of creative tasks. The first is the formulation of objectives as *learning* problems. We seek to distinguish the problems we explore from the area of *synthesis* (also typically called program synthesis), which is much more frequently associated with the kinds of techniques we use. Although synthesis is a broad subject, much of it focuses on the production of programs or other symbolic expressions that conform to a *specification* or high-level description. In contrast, we seek to find logical formulas that satisfy certain *requirements*. As such, the problems we study have no formal specification, which is why we design frameworks that *elicit* the concepts we want using different kinds of examples. Consider for example the problem of inductive lemma synthesis that we study in Chapter 3 (Section 3.2.3). Written in plain English, the problem is: "Find a sequence of formulas such that each of them is provable (perhaps using the previous ones) by an automatic induction prover X and the formulas together prove the theorem at hand using some automatic theorem prover Y". Note that the manner of the induction proofs for each of the lemmas, the internals of the automatic provers, and the number of lemmas needed or their dependence on each other are all unspecified or unknown! This is in fact the problem that users are solving when they interact with an automatic prover like Dafny or Liquid Haskell. They must come up with an unknown number of lemmas and prove all that must be proven using only the language interface they are given. In our work we simplify this large pain point into a specific mathematical problem by studying a particular logic and develop a tool to automate lemma synthesis for a particular domain.

We note here that at the time of writing this thesis, the word *learning* overwhelmingly points in the direction of machine learning/artificial intelligence research based on neural networks. Although we do not use neural networks, there are many similarities. The key similarity is the need for *generalization*. Both problems are typically under-specified, and one must equip learners with inductive biases in order to find solutions. In our work we borrow the idea of using grammars as one kind of inductive bias from the field of program synthesis, and introduce a new manner of inductive bias through the different varieties of first-order (counter)example models. The key difference between contemporary ML/AI research and our learning problems is our second guiding principle, which is that we require the output of learning to be a logical formula. This is certainly not a canonical formulation, and one can certainly explore the generation of other kinds of objects that ease the burden of providing creative help, like (counter)example structures or even natural language utterances. However, creative insight encoded as a symbolic expression is an aesthetically pleasing formulation for

mechanization and is also a common modality for soliciting user help in automated reasoning. For example, one can directly use a learned/provided lemma in an automatic theorem prover.

In this final section of the chapter, we give a brief overview of literature related to these guiding principles in the context of our work.

Program Synthesis and Formula Synthesis The field of program synthesis is very relevant to our work, especially the Programming By Example (PBE) paradigm [225, 226]. Many different techniques [55, 64, 227, 228] proposed in the inductive synthesis[3] literature have proven useful for other kinds of synthesis/learning problems. Common formats and frameworks for synthesis like SyGuS [64, 91] and SemGuS [230, 231] have spurred advances in algorithmic techniques for synthesis and produced off-the-shelf tools from which our research benefits greatly.

The synthesis of logical formulas is a related problem that has been explored in prior literature, often in the context of loop invariant synthesis [232, 233, 234, 235, 236] (including approaches involving learning from examples [51, 237, 238]) or the synthesis of relational queries for databases [239, 240, 241]. However, the problem of learning quantified formulas from rich first-order example models has only seen theoretical and practical progress more recently. The works in [242, 243] investigate theoretical questions and propose algorithms for broad fragments of quantified FOL based on techniques such as constraint solving and tree automata emptiness [242, 243]. However, the algorithms do not appear to be effective in practice. The work in [243] tackles the problem of synthesizing formulas with unboundedly many quantifiers but over finitely many variables (possibly reusing variables) and proves decidability results using tree automata. However, they do not present any practically effective algorithms, and naive implementations of tree automata techniques suffer from state-space explosion. There has been some recent work in program synthesis that has shown promise for efficient algorithms based on refinements of tree automata [244, 245], but it is unclear whether these techniques transfer to synthesis of formulas. The work in [242] develops a synthesis technique based on SAT solving and uses it to synthesize global invariants for distributed protocols, but the complexity of the generated queries causes SAT solvers to struggle in practice. In contrast, works based on clever enumeration techniques perform better on the same benchmarks [246, 247].

The model-guided learning frameworks we develop are in spriti agnostic to the specific algorithmic techniques used for formula synthesis. However, we require a fairly expressive interface, and in general none of the aforementioned works appear to meet our requirements. First, we use grammars to specify our hypothesis space. Second, we utilize a variety of

[3] as opposed to *deductive* synthesis, *a la* [229]

example classes beyond the usual positive and negative examples. The contribution of each kind of example towards the learning objective is given to the learner in the form of a constraint over the structure of the formula to be synthesized. Finally, our models and constraints involve a combination of theories such as integers and sets. We except that advances in tools and techniques for formula synthesis will improve the efficacy of our overall approach.

Emerging work in neuro-symbolic learning defines continuous/differentiable relaxations of semantics for logics and uses optimization algorithms such as gradient descent to learn formulas from examples [248, 249]. While these approaches have the potential to provide stronger forms of inductive bias than previously developed for formula synthesis, the area is somewhat nascent and preliminary approaches seem to be brittle.

Synthesis and Learning for Automating Creative Tasks The general idea of using synthesis or even learning to automate burdensome manual tasks is certainly not new. Apart from the synthesis of inductive lemmas and loop invariants which we have mentioned earlier, there are several other forms of creative help that are being pursued for automation with great interest in contemporary work. Examples include quantifier instantiation [250, 251, 252], contracts/class interfaces [253, 254, 255, 256, 257], and termination arguments [258, 259, 260]. The synthesis of global state invariants for distributed protocols expressed in logic [242, 246, 247] is one with a lot of potential impact for building verified systems. The invariants are complex and use nested quantification, which makes it a hard problem for synthesis. However, the role of the expert in encoding the protocol in logic is unclear, and in particular it seems hard to determine whether invariants can even exist in the available vocabulary for a chosen logical abstraction. While this may preclude non-experts from using such a workflow, we suspect that the invariants may also be hard for experts to formulate, and synthesis can certainly aid in that setting. A potential complement to such a process may lie in works on *unrealizability* [261, 262].

Turning to frameworks, early work by Angluin developed the query-based active learning model [263]. In program synthesis and related problems, the oracles to be queried are often verification engines, test generators, or model checkers. The interesting aspect is instead the design of the learner itself and the kinds of queries it issues to these different oracles. In this realm, the CounterExample-Guided Inductive Synthesis (CEGIS) framework [55] in the program synthesis literature formulates an example-based learner which utilizes queries (usually candidate solutions) to obtain counterexamples from a verification oracle. CEGIS addresses synthesis problems that are stated in the $\exists\forall\ldots$ form (loosely read as follows: "exists a finitary encoding of the solution such that for all inputs/free variables the solution

meets the specification") by relaxing the universal quantification to satisfaction over a finite number of examples. Our model-guided synthesis framework is can be seen as a variant of this architecture. The key difference is that the problems we investigate (as we discussed at the beginning of this section) don't seem to have clear formal specifications at all. The oracle-guided inductive synthesis framework (OGIS) [264, 265] is a closer fit and does not require the problem to have a formal specification, but the framework does not typically allow for open-ended learning algorithms where the framework may run forever. Still, both CEGIS and OGIS are based on the principle that examples can encode information about the "manner" in which previous candidate solutions failed which can help rule out other structurally similar bad candidates. While this is now a well-understood perspective on program synthesis, the same cannot be said for hypothesis classes of formulas. In particular, structural similarity seems hard to gauge in formula space. Counterexample-guided abstraction refinement (CEGAR) [266, 267] is another form of counterexample-guided learning. Unlike CEGIS, CEGAR techniques produce an abstract model, upon which counterexamples are analyzed and the abstract model is accordingly refined.

The work in [268] develops an abstract learning framework for synthesis that subsumes many learning frameworks. It also investigate specific mathematical structure in the abstract frameworks to formulate properties such as progress or convergence for the learner. We believe that the model-guided synthesis framework can be seen as an instance of such abstract reasoning frameworks.

We now discuss literature related to the two problems we tackle in our work, namely inductive lemma synthesis and axiom synthesis.

Synthesis and Automation for Inductive Reasoning A first problem for inductive reasoning is to prove a given theorem by induction. Typical automation approaches for this this problem involve fixing one or more induction *templates* (say structural induction over the structure of an algebraic datatype, or the *PFP* formula we use in Chapter 3) and using search heuristics to fill the holes in the template. This is a fairly effective technique for several problems and has been explored in prior work [209, 269, 270, 271, 272, 273] including for various specialized fragments [274]. However, it may not be possible to prove a theorem by induction directly. A more general version of the problem involves finding one or more *lemmas* that are provable by induction (using some templates as before) such that the lemmas help prove the theorem at hand without inductive reasoning. In Section 3.2.3 of Chapter 3 we further distinguished between theorems that require a sequence of lemmas versus those that can be proven using lemmas that are independently provable by induction. Early systems

solicited these lemmas from users; we have discussed these in previous sections. The use of synthesis and learning techniques to find inductive lemmas is of course relatively recent.

In our work we look at inductive lemma synthesis in the context of verifying imperative programs. The work in [275] also studies verification of imperative programs and uses proof-theoretic techniques to discover subgoals during proofs that serve as inductive hypotheses to help the proof. This relies on chancing upon inductive lemmas during proof, and the paper does not provide any relative completeness results. In contrast, our technique is syntax-guided for arbitrary lemmas and is relatively complete. Other lemma synthesis approaches include that of the work in [276] which also uses SyGuS for lemma generation and operates over the domain of bitvector problems. SLS (Songbird+Lemma Synthesis) [152] is a tool for lemma synthesis over Separation Logic. SLS identifies candidate lemma templates by looking at the heap structure of a given entailment. It then conducts structural induction proofs to generate constraints on top of a lemma template, then solves the constraints to refine the template and discover inductive lemmas.

Lemma synthesis to aid verification of functional programs over Algebraic Datatypes has been better explored [24, 203, 209, 217, 273, 277, 278, 279, 280]. The work in [66] uses syntax-guided-synthesis similar to our approach but does not use counterexamples. The work in [217] formulates verification problems as CHCs and develops a CHC solving algorithm over algebraic datatypes and uninterpreted predicates to infer inductive facts similar to lemmas. Recent work studies lemma synthesis in an interactive theorem proving setting [281]. This is interesting since it is not always possible to find a lemma that can complete a proof. Importantly, the authors integrate their technique into Coq [10] as a tactic that users can readily employ. The work in [282] discovers lemmas for inductive equational proofs over algebraic datatypes using a method that it dubs lemma *discovery* as opposed to synthesis owing to the connection its approaches have with the problem of theory exploration (see below on works related to axiom discovery).

To the best of our knowledge, ours is the only work on inductive lemma synthesis for theorems arising from imperative program verification and written in FO+*lfp*. Although our technical setting is quite broad, we find that adapting synthesis techniques between logics is hard. At one level, translating between techniques for imperative and functional programs is challenging since the datastructures as well as the underlying theories are very different. For example, it is hard to think about the equivalent of a doubly linked list or a circular list using ADTs, and conversely it is hard to define algebraic trees in a heap setting. Even between different logics for verifying imperative programs, it turns out that translation can incur unnecessary bloat that can make lemma synthesis harder— or even easier, to the point of eliminating the need for a lemma at all! We refer the reader to Section 3.6.7 for a more

detailed comparison between our approach and state-of-the-art approaches developed for other logics.

We now turn our discussion to a very closely related problem and arguably the most popular problem in the realm of synthesis for inductive reasoning, namely loop invariant synthesis [51, 232, 233, 234, 235, 236, 237, 238]. Our work in Chapter 3 in fact appears to share many curious similarities with the work on ICE Learning [51] for loop invariant synthesis. There is some mathematical unity between the two problems in that inductive invariants are similar to inductive lemmas when programs are written as formulae in FO+*lfp*. The positive, negative, and implication counterexamples used in the ICE framework also possess an aesthetic resemblance to our *Type−2*, *Type−1*, and *Type−3* counterexamples respectively. However, to the best of our knowledge, this similarity falls apart on closer inspection. First, programs, especially stateful programs that modify heaps are hard to translate into pure logic. Note that one would have to capture the potentially unbounded modification to the pointers performed by a loop in the program in terms of a recursive definition.

The similarity between the counterexamples does not hold up either. Our *Type−1* counterexamples are not actually negative examples. They are witness *non-provability*, not invalidity. Interestingly, a different work on loop invariant synthesis [283] does indeed utilize such examples and also recognizes them as capturing non-provability information. The *Type−3* model = implication counterexample match does not hold up either. In the loop invariant setting programs evolve states, leading to counterexamples of the form (S_1, S_2) where S_1 and S_2 are program states and the example enforces that if S_1 is contained in the synthesized invariant then S_2 must be contained as well (hence the name implication counterexample). In contrast, in the pure FO+*lfp* theorem proving setting, there are no changes to models that call for having two separate models as in an implication counterexample. Instead, *Type−3* models are a single positive counterexample over which the *PFP* of a lemma must hold. The PFP formula is itself an implication where the lemma to be synthesized "appears" on both sides, giving the appearance of a counterexample capturing an implication. We leave the determination of the precise mathematical connections between the two worlds, if any, to future work.

Lastly, on the empirical front, we find in preliminary investigations that FOSSIL cannot handle invariant synthesis problems even when we can state them in FO+*lfp*. This is because we do not synthesize lemmas that quantify over background sorts such as integers. Our synthesis algorithm that uses essentially Boolean constraint solving exploits the fact that expressions synthesized do not have constants over the background sort. We manually identify

precise fragments of logic for expressing lemmas that enable the effective automation of induction proofs and are expressive enough to capture practical proofs. We then use the knowledge of the existence of this fragment to make our synthesis problem easier. The structure of loop invariants is quite different, and the structure of their hypothesis space is different as well.

Discovering Axioms and Inferring Laws from Data The axiomatization of logics and classes of structures has a rich history in the mathematical literature. But the term *axiomatization* is used to describe many different kinds of problems. To our knowledge, ours is the first work to study automated axiomatization of classes of structures from a model-theoretic perspective.

One class of problems rooted in the work of Leśniewski, Tarski, and Łukasiewicz [284, 285, 286] is to find *simple* axioms for algebraic structures (e.g., groups) for which axiomatizations are already known. The objective is to find axiomatizations that are shorter (as short as a single axiom) or that use a different set of operators (e.g., a division operator for groups). Work by William McCune and contemporaries [287, 288, 289, 290, 291, 292, 293, 294, 295] studies using computers to find simple axiomatizations for various algebraic structures.

The problem of automated theory discovery or theory exploration has also been studied [271, 282, 296, 297, 298, 299, 300, 301, 302, 303, 304, 305]. In theory exploration, one has a set of axioms A that defines a theory of interest, and the goal is to find formulae φ that belong to the theory of A, i.e., $A \models \varphi$, with a preference for finding interesting or complex theorems, e.g., discovering Sylow theorems given group axioms. The model-theoretic axiomatization problem can be thought of as a dual problem to theory exploration since it requires finding axioms given the theory, i.e., to find A such that $A \models \varphi$ for every φ in the theory of some class of structures $\mathcal{C}$.

Both axiom discovery and theory exploration are grounded in well defined mathematical realities: groups, modal frames, equations entailed by a theory, etc. A more general problem involves the discovery of *laws* or patterns that 'explain' observed data. This problem, also known sometimes as *symbolic regression*, overlaps much more heavily with contemporary artificial intelligence research, but the field is quite under-explored [306, 307, 308, 309]. This is an exciting avenue to extend the ideas developed in this thesis.

Chapter 7: Conclusion and Future Work

In this thesis we asked a rather presumptuous question: are we stuck in the field of automated verification? In what ways are we stuck, and how do we move forward? We then identified a very prevalent pain in using automated verification, namely the inexplicable need for expert creative help, and attempted to give this problem some structure. This led us to the idea of formally characterizing the role of creative help, and well as the idea of learning logical expressions from rich example structures as a potential means to bridge the creativity gaps in automation. We also explored how the design of the verification paradigm can in itself allow us draw boundaries differently between the role of automation and the role of user help, and thought about what an appropriate balance between the roles of these two parties might be.

Our quest is far from over, and there are a number of compelling questions to pursue for the future. We record a few thoughts on these below.

7.1 BETTER FRAMEWORKS FOR AUTOMATED VERIFICATION

Foundational Questions There are several foundational questions that remain open across the different reasoning frameworks and logics that we have encountered in this work. We first start from where we left our study in Chapter 2.

Complete Reasoning. We discussed the completeness result in the work on foundations of natural proofs [27] (henceforth called FNP) in Chapter 6, and noted that the our completeness result was essentially incomparable with theirs. To recall for the reader, the key differences are: (a) our result works over an interpreted foreground sort supporting a theory of ADTs, whereas the FNP result works over an uninterpreted foreground sort (used to model heaps), and (b) our result supports recursively defined functions that have arguments from background sorts, which falls outside the "safe" fragment defined in FNP, and (c) we do not know of complete thrifty instantiation schemes that can handle quantified statements other that definitions (lemmas, for example), whereas the FNP result does define a complete thrifty instantiation strategy for non-definitional quantified formulas (which is, of course, more liberal than simple unrolling of recursive definitions). The gap between unrolling-based reasoning and the intended semantics is really the gap between FO and FO+*lfp* in the FNP work, whereas in our result the gap is between FO over combined theories versus FO on the standard model. One open problem is to find a more general result that extends both results. More generally, what are the key principles that underlie the design of logics that admit complete thrifty

instantiation schemes? The design of programming languages and specification languages whose verification queries are guaranteed to fall into logics with complete thrifty reasoning strategies can lead to better practical automation.

There are also a number of extensions or variants of our completeness result that would be interesting to pursue:

1. Finding a complete thrifty instantiation scheme that can handle (user-written) universally quantified lemmas. This does not exist as far as we know; current tools like Liquid Haskell ask for users to provide the lemmas as well as the relevant instantiations for the proof. The user instantiation removes the quantification, which reduces the new query after writing a lemma once again into our fragment defined in Chapter 2, but this is not satisfactory.

2. Other interesting extensions include handling definitions that do not terminate on some inputs (i.e., *partial* functions), handling polymorphism, and handling higher-order functions (HOFs). The last of these is the most exciting. LH and Stainless in fact support defining HOFs but our result does not apply to such definitions. These tools reason with HOFs using defunctionalization [310], i.e., converting them to FOL definitions by modeling function symbols as constants in a new sort and introducing an uninterpreted function *apply*, where $apply(f, args)$ models the application of a (possibly) higher-order function f to arguments $args$. However, simply defunctionalizing higher-order definitions and applying our existing result would only yield a completeness theorem for the first-order fragment containing defunctionalized formulas. This is not enough, since the desired completeness theorem in this setting is a result for the appropriate higher-order logic. This effort is further complicated by the fact that Second-order logic is already incomplete (and can be as powerful as mathematical reasoning itself). Therefore, reasoning in the users' intended (a.k.a standard) model is incomplete. But this is also true of our existing result! Instead, our completeness pertains to the logic of *combined theories* which we show is the logic that the heuristics implicitly reason with. We believe that there does exist a completeness result for analogously defined higher-order logic whose reasoning matches with the heuristics but whose class of models would be larger than the standard model. Defining such a logic that supports higher-order functions with recursive definitions and yet admits completeness of a thrifty instantiation scheme seems challenging.

3. We believe that the theoretical tools discovered in our work can be used to invent new complete algorithms similar to UQFR for other logics. One example is logics that

support datatypes other than ADTs, e.g., abstract data types such as sets, maps, and queues.

4. Another possible transfer of our result involves relaxing the setting in the FNP work to have users supply instantiations for lemmas. It would then be useful to ask whether we can define a complete thrifty instantiation scheme that can support a practicable set of recursively defined functions with arguments from background sorts. Recall that the FNP setting deals with heaps where datastructures can be infinite (e.g., infinite linked lists), as opposed to ADTs where values are finite. Recursive definitions are hence given semantics using least fixpoints. Notions of 'termination' and 'provable acyclicity' are not easy to define in this setting.

Thrifty Instantiation. A separate direction involves investigating mechanisms to design thrifty instantiation schemes. The scheme that we study is a simple syntactic one: if you find a ground term $R(\bar{t})$, unroll the body of $\mathcal{R}$ on the arguments $\bar{t}$. However, one can look at schemes that are even thriftier. For example, when the PLE heuristic used by Liquid Haskell encounters a case split it requires that one of the case guards be proven to match (using a separate query to an SMT solver), and then only unrolls the body for the corresponding case rather than the whole definition. We illustrate an example in Chapter 6 which shows that PLE is not complete (i.e., can design an example where it fails but UQFR would not), but the broad principle of only unrolling a definition when it is *useful* to prove something is quite sensible. This is in fact similar in spirit to resolution, and we believe the use of resolution-like techniques to guide instantiation can result in extremely efficient algorithms.

An extremely relevant *twist* on the above problem is to develop theoretical tools to study instantiation heuristics obtained using machine learning algorithms. Such techniques are already being used to improve the efficiency of solvers in practical settings [250]. Although they may be extremely thrifty, they are unlikely to be complete in general. It would be interesting to explore fragments of logic with interesting statistical properties (e.g., "generated from an underlying distribution", whatever that may mean) where learning-based instantiation schemes are guaranteed to be optimal.

New Logics. We turn to discussing open foundational questions on other logics and frameworks encountered in this thesis. The most fertile ground here is of course the investigation of the Intrinsic Datastructures (IDS) paradigm developed in Chapter 4. Our evaluation of the expressive power of the logic was only empirical: we simply considered common heap datastructures and were able to encode all of them. There were many surprising gadgets that we discovered, such as the use of a monadic map with rational values to define trees

(see Section 4.1). Note that over finite heaps one can certainly express typical recursive definitions using IDS since we can always invent a monadic map to simulate a ranking function corresponding to the least fixpoint computation and demand that recursive calls decrease the ranks. However, we believe that IDS is a strictly more powerful logic, and it would be interesting to precisely characterize its expressive power.

Expanding the IDS/FWYB framework to relate to other paradigms is another useful direction. In particular, it would be interesting to see how intrinsic definitions with fix-what-you-break proof methodology can coexist and exchange information with traditional recursive definitions with induction-based proof methodology. For instance, programs with some classes/data structures being supported using intrinsic definitions while others are supported using recursive definitions (e.g., common datastructures which may be packaged as a verified library) will allow more flexibility for using IDS in practice. Another effort of this kind would be to adapt IDS for functional programs. Since functional data structures are not mutable, ghost fields will always meet local conditions. However, we may need to *(re-)establish* rather than *repair* local conditions, which may require ghost code, e.g., establishing that the ghost map *sorted* on a functional list x is true. Recent work in [311] for verifying functional programs shares some components of this vision.

We can also apply the idea of identifying limitations of paradigms to ghost code which we encountered in Chapter 4. Although writing ghost code is a well-established technique to bridge gaps, we do not know of any work on the limitations of the framework itself. For example, in our work we used *ghost loops* and *ghost methods* to construct monadic maps (see Section 4.7.2). However, these are hard to figure out for a user without a lot of expertise. It would therefore be interesting to look at, say, the power and limitation of ghost code when only straight-line ghost code is allowed (note that the original method can have loops, and one can write straight-line ghost code inside a loop body).

Building Tools with Better Automation Support One of the big questions we have not addressed in this work is how our various insights come together to inform the development of verification tools. We speculate some potential directions on this subject here.

Inductive Reasoning and Lemma Synthesis. One clear recommendation of this thesis is to make tools for inductive reasoning (and specifically inductive lemma synthesis) more readily available for integration into verification workflows. We boldly claim that the field is ready to begin an era of making available off-the-shelf solvers that can handle typical inductive reasoning queries in practice analogous to the role of SAT/SMT solvers today for decidable reasoning over quantifier-free combinations of theories. Tools like the Imandra prover [209]

already echo this sentiment.

There are several frontiers where contributions would push this agenda forward. The first is the integration of lemma synthesis into a user-facing verification engine used widely, such as Dafny [9]. This would allow us to focus on a particular language and investigate specific principles, heuristics, or tricks that can make lemma synthesis practical in the language. For example, the work in [312] describes a taxonomy of typical inductive lemmas that seem to be required for the specifications that they deal with. Consider also the unique advantage of working with a user-facing verifier: we can analyze the typical verification queries that the tool fails on due to failure of inductive reasoning and refine our techniques accordingly. For example, the automatic generation of grammars from a verification query is a crucial part of using grammar-based synthesis techniques. Although we use a fixed scheme in our work we find that it can make a substantial difference in the learner's ability to converge onto a useful set of lemmas. Since there are no clear principles for generating grammars automatically, understanding the typical use-cases for a particular tool may provide a way forward. We may also be able to solicit user feedback from a language interface, which can allow other kinds of learning techniques to also compete for providing effective lemma synthesis to a user.

Our work also represents significant progress in the verification of formulas in first-order logic with recursive definitions (FORD). Therefore, a second interesting direction to pursue would be to define a class of logics around FORD (similar to SAT/SMT) and develop off-the-shelf solvers for these logics. Such a standard will facilitate the use of solvers for this problem and yield a platform where solvers can compete and innovate to find induction proofs automatically.

There are also directions for impact that require less radical movements: for example, we show in Chapter 2 that the reason queries fail without lemmas is that there are rogue nonstandard models. Note that these are legitimate models that demonstrate invalidity in the underlying FO logic. However, we only use the possibly weaker non-provability examples (*Type−1* and *Type−3*) to guide the synthesis in Chapter 3. Integrating rogue nonstandard models into the set of counterexamples has the potential to make synthesis more efficient. However, extracting these models is hard; even expressing them is hard as they are typically infinite— see recent work in [313, 314] on handling infinite counterexamples.

Tools with Predictable Automation. In Chapter 4 we developed the FWYB methodology and the well-behaved language paradigm as an answer to the need for predictable verification. However, as we briefly pointed out during our case studies (Section 4.7.3), our first attempt at this language paradigm does not cover all the gadgets one might need while verifying even common heap datastructures. Another simple example is the idea of multiple broken

sets tracking the breakage of a mutually disjoint set of local conditions, which we use in our evaluation but do not provide a helpful language feature to leverage for users of our framework. Revisiting the well-behaved programming paradigm with a larger set of case studies and creating a more comprehensive language paradigm would be invaluable.

A related effort here would be to develop verification engines for higher-level languages (like Verifast for Java [161], Verus for Rust [315], and Dafny [9]) that that provide native support for intrinsic definitions and produce verification conditions in decidable theories that SMT solvers can handle efficiently (see RQ3 in Section 3.6 on the importance of generating decidable queries). In fact, we can also marry into this the idea of learning for bridging creativity gaps: as we mention in Section 3.6, many updates of monadic maps are straightforward using definitions, and tools that automate this can reduce annotation burden significantly. The use of program synthesis techniques may be able to drive down the annotation burden even further[1]. Integrating definitional updates of monadic maps into a high-level language with native support for IDS and exploring lightweight program synthesis techniques would make IDS an extremely attractive framework for developing verified software.

Finally, we can also look into building tools that implement complete procedures like the one we develop in Chapter 2 (which may run forever) to provide a different kind of reliable automation: one where the failure of the tool after spending a certain amount of time is not a guarantee that the problem is fundamentally unsolvable by the tool, but is typically the case in practice. More generally, completeness ensures that there are no *embarrassing* misses of proofs; see the tutorial cited as [316] for a more detailed presentation on the case for completeness as a theoretical standard for verification algorithms and tools.

Further Practical Domains In this work we have focused on the verification of heap-manipulating programs and the verification of functional programs over ADTs as our domains of study. However, these are admittedly well-trod paths in the literature and there are many complex programs and specifications that we can look at with the creative help (💡) + reliable automation (⚙) lens that we have espoused in this thesis.

Even within the broad realm of datastructures, we are intrigued with the ease with which intrinsic data structures capture complex data structures such as overlaid data structures in a simple, compositional manner (Section 4.7.5). Exploring intrinsic definitions for verifying concurrent and distributed programs that maintain data structures would be interesting.

[1]Note that this does not alleviate the burden of having to come up with the right monadic maps in the first place, but that may present its own opportunity for a learning problem: given a datastructure with a partial IDS definition (or even a recursive definition), identify a set of monadic maps and local conditions that that can prove a given program correct. These additional maps must not, of course, change the definition of the datastructure itself.

Beyond datastructures, intrinsic definitions open up an entirely new approach to defining properties of unbounded structures in a way that can simplify reasoning. Exploiting intrinsic definitions in other verification contexts like unbounded mathematical structures used in specifications (e.g., message queues in distributed programs), parameterized concurrent programs (where configurations are modeled as unbounded sequences of states), and programs that manipulate big data concurrently (like Apache Spark) are exciting future directions.

Another realm of verification problems beyond datastructures is the refinement of distributed protocols. Protocol designers and the engineers who implement them use several creative insights to argue the correctness of optimizations, changes, or refinements they make make to a base protocol. This problem is currently beyond the abilities of modern automated verification frameworks.

Finally, we briefly mention here the emerging area of using large language models (LLMs) trained using truly staggering amounts of data to write proofs [317, 318] (including proofs of programs [319, 320]). We can once again apply our lens in this realm and investigate the typical gaps in reasoning exhibited by LLMs[2] and use techniques similar to the ones developed in this work to bridge the gaps. A taxonomy of reasoning gaps for LLMs would be a seminal contribution in this space. Current literature seems to suggest that the synthesis of inductive lemmas is already such a gap [321], which provides a direct connection to the contributions of this thesis. A slightly less speculative direction is the use of LLMs as powerful search heuristics to guide synthesis algorithms, e.g., if an LLM can suggest "proof by induction on n" as the best hypothesis from a space of candidate hypotheses, an automated reasoning engine can then frame the appropriate formal induction step and check if it is valid.

Design Considerations for Verification Languages and Frameworks Our work on Predictable Verification looks at redesigning the verification experience of a user around the provision of creative help in a painless manner. This work can be seen as a point in an entire spectrum of verification frameworks. On one end of this spectrum lie frameworks that employ simple, lightweight types. The specifications are essentially fixed (one designs a type system to check a predetermined property or set of properties), but the user burden is relatively low (determine the types of variables/functions), and the level of automation is high (type checking is decidable). On the other end of the spectrum lie bare-bones interactive theorem proving frameworks. The specification language is very flexible and can express complex properties, but the user burden is high (user has to essentially write the entire proof), and automation is low (one only has a proof checker). Our IDS/FWYB framework strikes a

[2]This may be a moving goalpost of sorts given that one may be able to constantly update the abilities of intelligent agents as more data becomes available, but we speak here of capabilities in a broad sense.

different balance compared to these extremes, allowing for a moderate level of flexibility in specifications (namely, it has to be expressible as an intrinsic definition) that appears to be sufficient to express several datastructures, and the user burden is moderate as well (ghost code to update monadic maps), but the automation is a decision procedure and is very predictable (similar to lightweight types). There can of course be many other points on this spectrum: for example, the work in [22] identifies a class of programs that can be verified without providing any loop invariants. As one might expect, the expressiveness of the class is reduced in order to obtain such a result.

The idea of designing frameworks around the provision of independently described user inputs is itself not new. For example, the Rust programming language [322, 323] which uses the idea of resource ownership to ensure that users cannot write (while sticking to the safe fragment of the language) code with memory safety violations or data races. In theory, a user can carefully look at their code and assess whether it meets the independently described rules of the Rust borrow checker (this is not easy in practice, however, and users often report a significant learning curve). Rust thus offers a predictable verification framework for memory safety and data race avoidance. The Verus framework [315] augments the rust language with an expressive specification language and a verifier for checking functional correctness specifications. However, the handling of the higher-level specification features falls back into the unpredictable heuristics category. The contributions of the IDS framework are in a sense complementary to this setup. We leave the handling of memory safety, checking modifies clauses, and ensuring preconditions to an underlying language and focus instead on a higher-level specification language that offers predictable verification via ghost code described using requirements which are specified independently from the underlying verifier for the high-level specifications. In general, programmers may wish to check the correctness of their code against multiple specifications, and building language frameworks that offer flexibility of composing different analyses and separation between unrelated analyses would be a valuable pursuit.

There are several exciting directions to pursue starting from our work. One grand vision is to lift the language of the required creative input required to better matches a user's intuitive arguments for correctness. This idea of *bridging intuition with formal proofs* will of course require the study of reasoning patterns employed by users of various kinds (novices/amateurs/experts), and the exploration of new modalities for the provision of creative insight. Recent work of this kind [324] formalizes the idea of a proof of correctness that takes the form of arguments for a handful of distinct "scenarios". A user can formally provide these scenarios and the high level proof arguments associated with them. This is then shown to be sufficient to automatically verify a certain useful class of concurrent programs. This

vision also opens up the possibility of new algorithms to translate between high level creative insights and formal proofs. The work in [197] refines user provided triggers for quantifier instantiation into more stable ones, and can be seen through this lens of leveraging insights to ensure reliable automation of any kind. As mentioned above, the use of even restricted forms of natural language as a modality for providing insight is another interesting dimension in this space.

A dual problem to the above vision involves the failure of proofs. In this thesis we have largely focused on algorithms for proving correctness and their limitations. However, users often write formal proofs that are incorrect at first (even if their high level intuition is correct) and gradually refine their arguments. It is therefore critical for verification frameworks to support workflows for repairing proofs/creative insights as well. For example, the work in [325] develops a language of proof actions that modify a user written proof to resolve common failure modes in an interactive theorem proving setting.

In our work in Chapter 2, we show that rogue nonstandard models must exist when inductive lemmas are needed (since our procedure is FO-complete). This provides a potential direction to guide users (as well as automatic tools) towards required lemmas. Recall that in Section 2.6.4 we showed an intuitive rogue nonstandard model that can be eliminated by a useful lemma. Generalizing this example into a mechanism of feedback requires overcoming a couple of challenges. We must first determine whether it is possible for users to ingest feedback comprising of such models into a realization of *why* their current proof fails, and further use that realization to come up with a useful lemma. If it is indeed true that rogue nonstandard models can provide effective feedback, we can try to synthesize models with finitary descriptions similar to the ones showcased in Section 2.6 using program synthesis, template-based synthesis using DSLs, or even finite model finders [313, 314]. An alternative pathway towards this objective involves the use of non-provability examples developed in Chapter 3 which also contain the essence of proof failure. In particular, non-provability examples give valuations to inductive definitions that do not conform to least fixpoint semantics. A user could point to the discrepancies in this model that may be relevant to the proof, and in doing so provide the creative input necessary to synthesize a useful lemma.

We can also similarly investigate feedback mechanisms for the FWYB framework. For example, an interactive system that shows broken elements and perhaps suggests fixes to programmers as they annotate the code may reduce cognitive burden.

Algorithms for Logic Learning This thesis has clearly argued that logic learning is a fundamental computational problem that can be used to automate many creative tasks. However, as we discussed in Chapter 6, state-of-the art algorithms for logic learning suffer from many challenges. Fundamental innovations in logic learning algorithms can impel a multitude of applications.

A major challenge is the outsized impact of the quality of counterexample models generated by various reasoning oracles. In our work in Chapters 3 and 5, we sometimes found that such models did not rule out enough bad hypotheses. This problem can also be alternatively stated by saying that counterexamples must be used with stronger inductive biases/generalization metrics. However, this assumes that it is possible to use a few well-designed counterexamples with an appropriate inductive bias to arrive at the desired hypothesis from a hypothesis space (given, say, as a grammar). We discussed in Chapter 6 that the insight from the CEGIS program synthesis framework of counterexamples to a bad candidate ruling out other bad candidates was not well-established for synthesis in quantified logics. In general, few-shot symbolic learning is not well-explored for learning from FO models even in settings with a static set of example models, let alone feedback-based settings like ours or active learning settings. It would be useful to study this problem in its own right.

A different approach to the above challenge would be to use a large number weak counterexamples to provide sufficient generalization. However, almost all symbolic synthesis algorithms we know of cannot handle a large number of counterexamples (unless it turns out that a small subset of them might be sufficient). Innovating new algorithms for logic learning in the large would also propel the agenda forward.

The integration of different kinds of inductive bias is another challenge. For example, contemporary machine learning algorithms can mimic learning from experience and capture typical syntactic reasoning patterns, and the kinds of algorithms developed in our work crucially leverage semantic information through counterexamples. However, these class of algorithms typically do not borrow techniques from each other.

Finally, developing general frameworks for logic learning with standard formats and supporting solvers would be another worthy pursuit. The work in [326] identifies a class of logics that admit decidable synthesis given a finite number of examples. The learning algorithm is based on tree automata, whose relative merits and limitations we have discussed in Chapter 6. Extending the availability of tools implementing state-of-the-art techniques for logic learning (including ours) to synthesizing formulas (and inductive lemmas in particular) in other widely used logics such as logics over ADTs and Separation Logic is also interesting.

The techniques here can be quite general: for example, although we perform goal-driven lemma synthesis in Chapter 3, one can perhaps think of synthesizing inductive properties as a kind of *axiomatization* for the logic at hand consisting of various recursively defined functions which culls out the class of models with least-fixpoint interpretations from the larger class of models that conform to fixpoint interpretations. This would open up the possibility of using open-ended exploration techniques such as the one we used in Chapter 5.

Axiom Discovery The development of the Learning-Based Axiom Synthesis framework in Chapter 5 opens up a new direction for computers to generate axiomatizations that were hitherto found by humans. There are many potential applications. One kind of application is to alleviate the burden of finding axioms in situations where the problem is tedious and perhaps not interesting to a human. For example, the semantics of CPU instruction sets can be seen as a logic with intricate semantics that can change frequently (see [327] for work on synthesizing semantics of such instruction sets). The semantics of such instruction sets is complex and researchers may not be motivated to axiomatize such relationships, especially for multiple evolving CPUs. Axiomatizing ISAs can enable several downstream applications, e.g., the work in [328] uses equational axioms between instructions to rewrite programs into having constant-time implementations that avoid timing attacks. Another example of a tedious axiomatization task is the work in [329] where the authors manually axiomatized a theory of relational algebra with updates to prove equivalence of database programs. This is an interesting variant to consider because unlike the domains we explore in Chapter 5, we must likely use a reference implementation as the soundness checker and use a weaker logic (perhaps the theory of uninterpreted functions) to build the independence checker. Generating counterexamples or identifying the right grammar for synthesizing such axioms appears challenging. Yet another application is learning metamorphic properties in metamorphic testing, where program modules are tested against sequences or compositions of function calls when specifications are not easy to think of or express for individual methods (e.g. $f(g(x)$ must be equal to $g(f(x)))$. The axioms can then be used to test other implementations (see the work in [330]). Proving the soundness of an axiom may be intractable in this scenario since modular specifications for the methods may be hard! In this case, we may resort to testing the axioms instead, perhaps using a test generator as our standard of guarantee for correctness [255]. In general, exploring the possibility of implementing the soundness/independence checkers and the counterexample generator using oracles that are weaker than verification engines or considering settings with noise, probabilities, or other forms of stochasticity would be novel challenges that would expand the impact of the LAS framework.

Learning Concepts over Structured/Unstructured Data The above problems mainly explore finding axioms automatically to support downstream applications. A second class of applications pertains to problems where finding axioms is a goal of its own. A typical example might involve learning axioms from a few examples described by a scientist, or learning the underlying dynamics of a reactive system.

The problem of finding descriptors or summaries that characterizing data sets (structured or unstructured) is a unique application in this space. This is related to the problem of symbolic regression [306, 307, 308, 309] discussed in Chapter 6, but a key difference is that data descriptors or summaries typically involve multiple concepts each of which may not explain the data fully or (in a noisy setting) even hold true of all elements in the dataset. This problem also shares similarities with the problem of rule mining. Related work on this problem has been used to check conformance of entries in a database [331, 332] and characterize sets of images for downstream conceptual analysis [333, 334]. Characterizing processes that produce data, especially systems that allow an active learner to intervene and analyze the results is another interesting variant of this problem.

References

[1] V. Doshi, "Analysis | A security breach in India has left a billion people at risk of identity theft," *Washington Post*, Dec. 2021. [Online]. Available: https://www.washingtonpost.com/news/worldviews/wp/2018/01/04/a-security-breach-in-india-has-left-a-billion-people-at-risk-of-identity-theft/

[2] S. Matteson, "Report: Software failure caused $1.7 trillion in financial losses in 2017," Jan. 2018. [Online]. Available: https://www.techrepublic.com/article/report-software-failure-caused-1-7-trillion-in-financial-losses-in-2017/

[3] C. D. of Motor Vehicles, "Autonomous Vehicle Collision Reports." [Online]. Available: https://www.dmv.ca.gov/portal/vehicle-industry-services/autonomous-vehicles/autonomous-vehicle-collision-reports/

[4] B. Pietsch, "2 Killed in Driverless Tesla Car Crash, Officials Say," *The New York Times*, Apr. 2021. [Online]. Available: https://www.nytimes.com/2021/04/18/business/tesla-fatal-crash-texas.html

[5] T. Ball and S. K. Rajamani, "The slam project: Debugging system software via static analysis," p. 1–3, 2002. [Online]. Available: https://doi.org/10.1145/503272.503274

[6] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood, "Sel4: Formal verification of an os kernel," in *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, ser. SOSP '09. New York, NY, USA: Association for Computing Machinery, 2009. [Online]. Available: https://doi.org/10.1145/1629575.1629596 p. 207–220.

[7] C. Newcombe, T. Rath, F. Zhang, B. Munteanu, M. Brooker, and M. Deardeuff, "How amazon web services uses formal methods," *Communications of the ACM*, 2015. [Online]. Available: https://www.amazon.science/publications/how-amazon-web-services-uses-formal-methods

[8] N. Rungta, "A billion smt queries a day," in *CAV 2022*, 2022. [Online]. Available: https://www.amazon.science/publications/a-billion-smt-queries-a-day

[9] K. R. M. Leino, "Dafny: An automatic program verifier for functional correctness," in *Proceedings of the 16th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning*, ser. LPAR'10. Berlin, Heidelberg: Springer-Verlag, 2010, p. 348–370.

[10] The Coq development team, "The coq proof assistant reference manual," 2018, Version 8.8.2.

[11] L. de Moura and N. Bjørner, "Z3: An efficient smt solver," in *Proceedings of the 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, ser. TACAS'08. Berlin, Heidelberg: Springer-Verlag, 2008, pp. 337–340.

[12] C. Barrett, C. L. Conway, M. Deters, L. Hadarean, D. Jovanović, T. King, A. Reynolds, and C. Tinelli, "Cvc4," in *Computer Aided Verification*, G. Gopalakrishnan and S. Qadeer, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 171–177.

[13] T. Trinh, Y. Wu, Q. Le, H. He, and T. Luong, "Solving olympiad geometry without human demonstrations," *Nature*, 2024.

[14] M. Eilers, M. Schwerhoff, and P. Müller, "Verification algorithms for automated separation logic verifiers," in *Computer Aided Verification*, A. Gurfinkel and V. Ganesh, Eds. Cham: Springer Nature Switzerland, 2024, pp. 362–386.

[15] A. Murali, L. Peña, E. Blanchard, C. Löding, and P. Madhusudan, "Model-guided synthesis of inductive lemmas for fol with least fixpoints," *Proc. ACM Program. Lang.*, vol. 6, no. OOPSLA, 10 2022. [Online]. Available: https://doi.org/10.1145/3563354

[16] P. Krogmeier, Z. Lin, A. Murali, and P.Madhusudan, "Synthesizing axiomatizations using logic learning," *Proc. ACM Program. Lang.*, vol. 6, no. OOPSLA, 10 2022. [Online]. Available: https://doi.org/10.1145/3563348

[17] A. Murali, L. Peña, R. Jhala, and P. Madhusudan, "Complete first-order reasoning for properties of functional programs," *Proc. ACM Program. Lang.*, vol. 7, no. OOPSLA2, oct 2023. [Online]. Available: https://doi.org/10.1145/3622835

[18] A. Murali, C. Rivera, and P. Madhusudan, "Predictable verification using intrinsic definitions," *Proc. ACM Program. Lang.*, vol. 8, no. PLDI, jun 2024. [Online]. Available: https://doi.org/10.1145/3656450

[19] A. Murali, L. Peña, C. Löding, and P. Madhusudan, "A first-order logic with frames," *ACM Trans. Program. Lang. Syst.*, vol. 45, no. 2, 5 2023. [Online]. Available: https://doi.org/10.1145/3583057

[20] A. Murali, L. Peña, C. Löding, and P. Madhusudan, "A first-order logic with frames," in *Programming Languages and Systems*, P. Müller, Ed. Cham: Springer International Publishing, 2020, pp. 515–543.

[21] A. Murali, H. Balakrishnan, A. Councilman, and P. Madhusudan, "Automating program verification for frame logic," *Technical Report*, 2024.

[22] U. Mathur, A. Murali, P. Krogmeier, P. Madhusudan, and M. Viswanathan, "Deciding memory safety for single-pass heap-manipulating programs," *Proc. ACM Program. Lang.*, vol. 4, no. POPL, 2020. [Online]. Available: https://doi.org/10.1145/3371103

[23] P. Krogmeier, U. Mathur, A. Murali, P. Madhusudan, and M. Viswanathan, "Decidable synthesis of programs with uninterpreted functions," in *Computer Aided Verification*, S. K. Lahiri and C. Wang, Eds. Cham: Springer International Publishing, 2020, pp. 634–657.

[24] R. S. Boyer and J. S. Moore, *A Computational Logic Handbook*. USA: Academic Press Professional, Inc., 1988.

[25] P. M. Rondon, M. Kawaguci, and R. Jhala, "Liquid types," *SIGPLAN Not.*, vol. 43, no. 6, p. 159–169, jun 2008. [Online]. Available: https://doi.org/10.1145/1379022.1375602

[26] J. Hamza, N. Voirol, and V. Kunčak, "System fr: Formalized foundations for the stainless verifier," *Proc. ACM Program. Lang.*, vol. 3, no. OOPSLA, oct 2019. [Online]. Available: https://doi.org/10.1145/3360592

[27] C. Löding, P. Madhusudan, and L. Peña, "Foundations for natural proofs and quantifier instantiation," *PACMPL*, vol. 2, no. POPL, pp. 10:1–10:30, 2018.

[28] A. R. Bradley and Z. Manna, *The Calculus of Computation: Decision Procedures with Applications to Verification*. Berlin, Heidelberg: Springer-Verlag, 2007.

[29] C. G. Nelson, "Techniques for program verification," Ph.D. dissertation, Stanford University, Stanford, CA, USA, 1980, aAI8011683.

[30] G. Nelson and D. C. Oppen, "Simplification by cooperating decision procedures," *ACM Trans. Program. Lang. Syst.*, vol. 1, no. 2, p. 245–257, oct 1979. [Online]. Available: https://doi.org/10.1145/357073.357079

[31] Y. V. Matiyasevich, *Hilbert's Tenth Problem*. Cambridge, MA, USA: MIT Press, 1993.

[32] N. Vazou, A. Tondwalkar, V. Choudhury, R. G. Scott, R. R. Newton, P. Wadler, and R. Jhala, "Refinement reflection: complete verification with SMT," *Proc. ACM Program. Lang.*, vol. 2, no. POPL, pp. 53:1–53:31, 2018. [Online]. Available: https://doi.org/10.1145/3158141

[33] R. Blanc, V. Kuncak, E. Kneuss, and P. Suter, "An overview of the leon verification system: Verification by translation to recursive functions," in *Proceedings of the 4th Workshop on Scala*, ser. SCALA '13. New York, NY, USA: Association for Computing Machinery, 2013. [Online]. Available: https://doi.org/10.1145/2489837.2489838

[34] J. Barwise, *Handbook of Mathematical Logic*. Amsterdam: North-Holland Publishing Company, 1977.

[35] A. I. Mal'tsev, "Axiomatizable classes of locally free algebras of certain types," *Sibirsk. Mat. Zh.*, vol. 3, pp. 729–743, 1962.

[36] W. Hodges, *A Shorter Model Theory*. USA: Cambridge University Press, 1997.

[37] N. S. Bjorner, "Integrating decision procedures for temporal verification," Ph.D. dissertation, Stanford University, Stanford, CA, USA, 1999, aAI9924398.

[38] L. Kovács, S. Robillard, and A. Voronkov, "Coming to terms with quantified reasoning," in *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*, ser. POPL '17. New York, NY, USA: ACM, 2017, pp. 260–270.

[39] M. Presburger and D. Jabcquette, "On the completeness of a certain system of arithmetic of whole numbers in which addition occurs as the only operation," *History and Philosophy of Logic*, vol. 12, no. 2, pp. 225–233, 1991. [Online]. Available: https://doi.org/10.1080/014453409108837187

[40] T. A. Skolem, "Über die nicht-charakterisierbarkeit der zahlenreihe mittels endlich oder abzählbar unendlich vieler aussagen mit ausschliesslich zahlenvariablen," *Fundamenta Mathematicae*, vol. 23, pp. 150–161, 1934.

[41] P. Suter, M. Dotta, and V. Kunćak, "Decision procedures for algebraic data types with abstractions," in *Proceedings of the 37th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '10. New York, NY, USA: ACM, 2010, pp. 199–210.

[42] H. B. Enderton, *A mathematical introduction to logic*. New York: Academic Press, 1972.

[43] G. Winskel, *The Formal Semantics of Programming Languages: An Introduction*. Cambridge, MA, USA: MIT Press, 1993.

[44] C. Tinelli and C. G. Zarba, "Combining decision procedures for sorted theories," in *Logics in Artificial Intelligence*, J. J. Alferes and J. Leite, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 641–653.

[45] P. Wadler, "Propositions as types," *Commun. ACM*, vol. 58, no. 12, p. 75–84, nov 2015. [Online]. Available: https://doi.org/10.1145/2699407

[46] P. Suter, A. S. Köksal, and V. Kuncak, "Satisfiability modulo recursive programs," in *Static Analysis*, E. Yahav, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 298–315.

[47] J. E. Hopcroft, R. Motwani, and J. D. Ullman, *Introduction to Automata Theory, Languages, and Computation (3rd Edition)*. USA: Addison-Wesley Longman Publishing Co., Inc., 2006.

[48] C. Calcagno, P. Gardner, and M. Hague, "From separation logic to first-order logic," in *Foundations of Software Science and Computational Structures*, V. Sassone, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 395–409.

[49] P. Madhusudan, X. Qiu, and A. Ştefănescu, "Recursive proofs for inductive tree data-structures," in *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '12. New York, NY, USA: ACM, 2012, pp. 123–136.

[50] X. Qiu, P. Garg, A. Ştefănescu, and P. Madhusudan, "Natural proofs for structure, data, and separation," in *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '13. New York, NY, USA: ACM, 2013, pp. 231–242.

[51] P. Garg, C. Löding, P. Madhusudan, and D. Neider, "Ice: a robust framework for learning invariants," in *Computer Aided Verification*, A. Biere and R. Bloem, Eds. Cham: Springer International Publishing, 2014, pp. 69–87.

[52] K. S. Namjoshi and R. P. Kurshan, "Syntactic program transformations for automatic abstraction," in *Computer Aided Verification*, E. A. Emerson and A. P. Sistla, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2000, pp. 435–449.

[53] R. Alur, R. Bodík, E. Dallal, D. Fisman, P. Garg, G. Juniwal, H. Kress-Gazit, P. Madhusudan, M. M. K. Martin, M. Raghothaman, S. Saha, S. A. Seshia, R. Singh, A. Solar-Lezama, E. Torlak, and A. Udupa, *Syntax-Guided Synthesis*, 2015, p. 1–25. [Online]. Available: http://dx.doi.org/10.3233/978-1-61499-495-4-1

[54] A. Solar-Lezama, G. Arnold, L. Tancau, R. Bodík, V. A. Saraswat, and S. A. Seshia, "Sketching stencils," in *Proceedings of the ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation, San Diego, California, USA, June 10-13, 2007*, J. Ferrante and K. S. McKinley, Eds. ACM, 2007. [Online]. Available: https://doi.org/10.1145/1250734.1250754 pp. 167–178.

[55] A. Solar Lezama, "Program synthesis by sketching," Ph.D. dissertation, EECS Department, University of California, Berkeley, Dec 2008. [Online]. Available: http://www2.eecs.berkeley.edu/Pubs/TechRpts/2008/EECS-2008-177.html

[56] E. Pek, X. Qiu, and P. Madhusudan, "Natural proofs for data structure manipulation in C using separation logic," in *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '14. New York, NY, USA: ACM, 2014, pp. 440–451.

[57] E. Grädel, P. G. Kolaitis, L. Libkin, M. Marx, J. Spencer, M. Y. Vardi, Y. Venema, and S. Weinstein, *Finite Model Theory and Its Applications*, ser. Texts in Theoretical Computer Science. An EATCS Series. Springer, 2007. [Online]. Available: https://doi.org/10.1007/3-540-68804-8

[58] A. Tarski, "A lattice-theoretical fixpoint theorem and its applications." *Pacific Journal of Mathematics*, vol. 5, no. 2, pp. 285 – 309, 1955. [Online]. Available: https://projecteuclid.org/euclid.pjm/1103044538

[59] L. Libkin, *Elements Of Finite Model Theory (Texts in Theoretical Computer Science. An Eatcs Series)*. SpringerVerlag, 2004.

[60] J. C. Reynolds, "Separation logic: A logic for shared mutable data structures," in *Proceedings of the 17th Annual IEEE Symposium on Logic in Computer Science*, ser. LICS '02. USA: IEEE Computer Society, 2002, p. 55–74.

[61] R. Alur, R. Singh, D. Fisman, and A. Solar-Lezama, "Search-based program synthesis," *Commun. ACM*, vol. 61, no. 12, p. 84–93, Nov. 2018. [Online]. Available: https://doi.org/10.1145/3208071

[62] A. Murali, L. Peña, E. Blanchard, C. Löding, and P. Madhusudan, "Model-guided synthesis of inductive lemmas for fol with least fixpoints (technical report)," 2022. [Online]. Available: https://arxiv.org/abs/2009.10207

[63] A. Murali, L. Peña, E. Blanchard, C. Löding, and P. Madhusudan, "Artifact for oopsla 2022 article model-guided synthesis of inductive lemmas for fol with least fixpoints," 2022. [Online]. Available: https://doi.org/10.1145/3554331

[64] A. Reynolds, H. Barbosa, A. Nötzli, C. Barrett, and C. Tinelli, "cvc4sy: Smart and fast term enumeration for syntax-guided synthesis," in *Computer Aided Verification*, I. Dillig and S. Tasiran, Eds. Cham: Springer International Publishing, 2019, pp. 74–83.

[65] A. Reynolds and V. Kuncak, "Induction for smt solvers," in *Verification, Model Checking, and Abstract Interpretation*, D. D'Souza, A. Lal, and K. G. Larsen, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2015, pp. 80–98.

[66] W. Yang, G. Fedyukovich, and A. Gupta, "Lemma synthesis for automating induction over algebraic data types," in *Principles and Practice of Constraint Programming*, T. Schiex and S. de Givry, Eds. Cham: Springer International Publishing, 2019, pp. 600–617.

[67] M. Sighireanu, J. A. Navarro Pérez, A. Rybalchenko, N. Gorogiannis, R. Iosif, A. Reynolds, C. Serban, J. Katelaan, C. Matheja, T. Noll, F. Zuleger, W.-N. Chin, Q. L. Le, Q.-T. Ta, T.-C. Le, T.-T. Nguyen, S.-C. Khoo, M. Cyprian, A. Rogalewicz, T. Vojnar, C. Enea, O. Lengal, C. Gao, and Z. Wu, "Sl-comp: Competition of solvers for separation logic," in *Tools and Algorithms for the Construction and Analysis of Systems*, D. Beyer, M. Huisman, F. Kordon, and B. Steffen, Eds. Cham: Springer International Publishing, 2019, pp. 116–132.

[68] Q.-T. Ta, T. C. Le, S.-C. Khoo, and W.-N. Chin, "Automated lemma synthesis in symbolic-heap separation logic," *Proc. ACM Program. Lang.*, vol. 2, no. POPL, Dec 2017. [Online]. Available: https://doi.org/10.1145/3158097

[69] P. W. O'Hearn, "A primer on separation logic (and automatic program verification and analysis)," in *Software Safety and Security - Tools for Analysis and Verification*, ser. NATO Science for Peace and Security Series - D: Information and Communication Security, T. Nipkow, O. Grumberg, and B. Hauptmann, Eds. IOS Press, 2012, vol. 33, pp. 286–318. [Online]. Available: https://doi.org/10.3233/978-1-61499-028-4-286

[70] P. W. O'Hearn, J. C. Reynolds, and H. Yang, "Local reasoning about programs that alter data structures," in *Proceedings of the 15th International Workshop on Computer Science Logic*, ser. CSL '01. London, UK, UK: Springer-Verlag, 2001. [Online]. Available: http://dl.acm.org/citation.cfm?id=647851.737404 pp. 1–19.

[71] L. de Moura and N. Bjørner, "Generalized, efficient array decision procedures," in *2009 Formal Methods in Computer-Aided Design*. IEEE, 2009, pp. 45–52.

[72] E. Cohen, M. Dahlweid, M. Hillebrand, D. Leinenbach, M. Moskal, T. Santen, W. Schulte, and S. Tobies, "Vcc: A practical system for verifying concurrent c," in *Theorem Proving in Higher Order Logics*, S. Berghofer, T. Nipkow, C. Urban, and M. Wenzel, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 23–42.

[73] J. Condit, B. Hackett, S. K. Lahiri, and S. Qadeer, "Unifying type checking and property checking for low-level code," *SIGPLAN Not.*, vol. 44, no. 1, p. 302–314, jan 2009. [Online]. Available: https://doi.org/10.1145/1594834.1480921

[74] B. Kragl and S. Qadeer, "The civl verifier," in *2021 Formal Methods in Computer Aided Design (FMCAD)*, 2021, pp. 143–152.

[75] D. Dill, W. Grieskamp, J. Park, S. Qadeer, M. Xu, and E. Zhong, "Fast and reliable formal verification of smart contracts with the move prover," in *Tools and Algorithms for the Construction and Analysis of Systems*, D. Fisman and G. Rosu, Eds. Cham: Springer International Publishing, 2022, pp. 183–200.

[76] C. Tinelli and C. G. Zarba, "Combining decision procedures for sorted theories," in *Logics in Artificial Intelligence*, J. J. Alferes and J. Leite, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 641–653.

[77] C. B. Jones, "The role of auxiliary variables in the formal development of concurrent programs," in *Reflections on the Work of C.A.R. Hoare*, A. Roscoe, C. B. Jones, and K. R. Wood, Eds. London: Springer London, 2010, pp. 167–187. [Online]. Available: https://doi.org/10.1007/978-1-84882-912-1_8

[78] P. Lucas, "Two constructive realizations of the block concept and their equivalence, ibm lab," Vienna TR 25.085, Tech. Rep., 1968.

[79] J.-C. Filliâtre, L. Gondelman, and A. Paskevich, "The spirit of ghost code," *Formal Methods in System Design*, vol. 48, pp. 152–174, 2016.

[80] J. C. Reynolds, *The craft of programming*, ser. Prentice Hall International series in computer science. Prentice Hall, 1981.

[81] S. Qadeer, "Boogie pull request #669: Monomorphization of polymorphic maps and binders," 2023. [Online]. Available: https://github.com/boogie-org/boogie/pull/669

[82] O. Lee, H. Yang, and R. Petersen, "Program analysis for overlaid data structures," in *Computer Aided Verification*, G. Gopalakrishnan and S. Qadeer, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 592–608.

[83] M. Barnett, B.-Y. E. Chang, R. DeLine, B. Jacobs, and K. R. M. Leino, "Boogie: A modular reusable verifier for object-oriented programs," in *Formal Methods for Components and Objects*, F. S. de Boer, M. M. Bonsangue, S. Graf, and W.-P. de Roever, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 364–387.

[84] P. Blackburn, J. F. A. K. van Benthem, and F. Wolter, Eds., *Handbook of Modal Logic*, ser. Studies in logic and practical reasoning. Amsterdam, Netherlands: Elsevier, 2007, vol. 3. [Online]. Available: https://www.sciencedirect.com/bookseries/studies-in-logic-and-practical-reasoning/vol/3/suppl/C

[85] A. Pnueli, "The temporal logic of programs," in *Proceedings of the 18th Annual Symposium on Foundations of Computer Science*, ser. SFCS '77. USA: IEEE Computer Society, 1977. [Online]. Available: https://doi.org/10.1109/SFCS.1977.32 p. 46–57.

[86] E. M. Clarke and E. A. Emerson, "Design and synthesis of synchronization skeletons using branching time temporal logic," in *Logics of Programs*, D. Kozen, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 1982, pp. 52–71.

[87] W. Hodges, *A Shorter Model Theory*. USA: Cambridge University Press, 1997.

[88] C. J. Anderson, N. Foster, A. Guha, J.-B. Jeannin, D. Kozen, C. Schlesinger, and D. Walker, "Netkat: Semantic foundations for networks," in *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '14. New York, NY, USA: Association for Computing Machinery, 2014. [Online]. Available: https://doi.org/10.1145/2535838.2535862 p. 113–126.

[89] P. W. O'Hearn, J. C. Reynolds, and H. Yang, "Local reasoning about programs that alter data structures," in *Proceedings of the 15th International Workshop on Computer Science Logic*, ser. CSL '01. Berlin, Heidelberg: Springer-Verlag, 2001. [Online]. Available: https://dl.acm.org/doi/10.5555/647851.737404 p. 1–19.

[90] L. Kovács and A. Voronkov, "First-order theorem proving and vampire," in *Computer Aided Verification*, N. Sharygina and H. Veith, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 1–35.

[91] R. Alur, R. Bodik, G. Juniwal, M. M. K. Martin, M. Raghothaman, S. A. Seshia, R. Singh, A. Solar-Lezama, E. Torlak, and A. Udupa, "Syntax-guided synthesis," in *2013 Formal Methods in Computer-Aided Design*. Portland, OR, USA: IEEE, 2013, pp. 1–8.

[92] P. Blackburn, J. F. A. K. v. Benthem, and F. Wolter, *Handbook of Modal Logic, Volume 3 (Studies in Logic and Practical Reasoning)*. USA: Elsevier Science Inc., 2006.

[93] J. Van Benthem, *Correspondence Theory*. Dordrecht: Springer Netherlands, 1984, pp. 167–247. [Online]. Available: https://doi.org/10.1007/978-94-009-6259-0_4

[94] D. Kozen, "A completeness theorem for kleene algebras and the algebra of regular events," *Information and Computation*, vol. 110, no. 2, pp. 366–390, 1994. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0890540184710376

[95] J. H. Conway, *Regular algebra and finite machines*, ser. Chapman and Hall mathematics series. London, UK: Chapman and Hall, 1971.

[96] A. Salomaa, "Two complete axiom systems for the algebra of regular events," *J. ACM*, vol. 13, no. 1, p. 158–169, 1966. [Online]. Available: https://doi.org/10.1145/321312.321326

[97] V. Redko, "On defining relations for the algebra of regular events," *Ukrainian Mathematical Journal*, vol. 16, no. 2, pp. 120–126, 1964, in Russian.

[98] P. Smith, "The galois connection between syntax and semantics," Jun 2010. [Online]. Available: https://www.logicmatters.net/resources/pdfs/Galois.pdf

[99] F. W. Lawvere, "Adjointness in foundations," *Dialectica*, vol. 23, no. 3/4, pp. 281–296, 1969. [Online]. Available: http://www.jstor.org/stable/42969800

[100] W. Hodges, *Model Theory*, ser. Encyclopedia of Mathematics and its Applications. Cambridge, UK: Cambridge University Press, 1993.

[101] C. Löding, P. Madhusudan, and D. Neider, "Abstract learning frameworks for synthesis," in *Tools and Algorithms for the Construction and Analysis of Systems*, M. Chechik and J.-F. Raskin, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2016, pp. 167–185.

[102] S. C. Kleene, *Representation of Events in Nerve Nets and Finite Automata*. Princeton, NJ, USA: Princeton University Press, 1956, pp. 3–42.

[103] J. L. Gischer, "Partial orders and the axiomatic theory of shuffle (pomsets)," Ph.D. dissertation, Stanford University, Stanford, CA, USA, 1985, aAI8506191.

[104] J. E. Hopcroft, R. Motwani, and J. D. Ullman, *Introduction to Automata Theory, Languages, and Computation (3rd Edition)*. USA: Addison-Wesley Longman Publishing Co., Inc., 2006.

[105] T. Lev-Ami, N. Immerman, T. Reps, M. Sagiv, S. Srivastava, and G. Yorsh, "Simulating reachability using first-order logic with applications to verification of linked data structures," *Logical Methods in Computer Science*, vol. 5, 04 2009.

[106] S. Lahiri and S. Qadeer, "Back to the future: Revisiting precise program verification using smt solvers," *SIGPLAN Not.*, vol. 43, no. 1, p. 171–182, jan 2008. [Online]. Available: https://doi.org/10.1145/1328897.1328461

[107] J. Berdine, C. Calcagno, and P. W. O'Hearn, "A decidable fragment of separation logic," in *FSTTCS 2004: Foundations of Software Technology and Theoretical Computer Science*, K. Lodaya and M. Mahajan, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 97–109.

[108] J. Berdine, C. Calcagno, and P. W. O'Hearn, "Smallfoot: Modular automatic assertion checking with separation logic," in *Proceedings of the 4th International Conference on Formal Methods for Components and Objects*, ser. FMCO'05. Berlin, Heidelberg: Springer-Verlag, 2006, pp. 115–137.

[109] J. Berdine, C. Calcagno, and P. W. O'Hearn, "A decidable fragment of separation logic," in *Proceedings of the 24th International Conference on Foundations of Software Technology and Theoretical Computer Science*, ser. FSTTCS'04, 2004, pp. 97–109.

[110] J. Berdine, C. Calcagno, and P. W. O'Hearn, "Symbolic execution with separation logic," in *Proceedings of the Third Asian Conference on Programming Languages and Systems*, ser. APLAS'05, 2005, pp. 52–68.

[111] B. Cook, C. Haase, J. Ouaknine, M. Parkinson, and J. Worrell, "Tractable reasoning in a fragment of separation logic," in *Proceedings of the 22nd International Conference on Concurrency Theory*, ser. CONCUR'11, 2011, pp. 235–249.

[112] J. A. Navarro Pérez and A. Rybalchenko, "Separation logic + superposition calculus = heap theorem prover," in *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '11. New York, NY, USA: ACM, 2011, pp. 556–566.

[113] J. A. N. Pérez and A. Rybalchenko, "Separation logic modulo theories," in *Programming Languages and Systems (APLAS)*. Cham: Springer International Publishing, 2013, pp. 90–106.

[114] J. Pagel, "Decision procedures for separation logic: beyond symbolic heaps," Ph.D. dissertation, Wien, 2020.

[115] R. Iosif, A. Rogalewicz, and J. Simacek, "The tree width of separation logic with recursive definitions," in *Automated Deduction – CADE-24*, M. P. Bonacina, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 21–38.

[116] M. Echenim, R. Iosif, and N. Peltier, "Unifying decidable entailments in separation logic with inductive definitions," in *Automated Deduction – CADE 28*, A. Platzer and G. Sutcliffe, Eds. Cham: Springer International Publishing, 2021, pp. 183–199.

[117] S. Itzhaky, A. Banerjee, N. Immerman, O. Lahav, A. Nanevski, and M. Sagiv, "Modular reasoning about heap paths via effectively propositional formulas," in *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '14. New York, NY, USA: ACM, 2014, pp. 385–396.

[118] S. Itzhaky, A. Banerjee, N. Immerman, A. Nanevski, and M. Sagiv, "Effectively-propositional reasoning about reachability in linked data structures," in *Proceedings of the 25th International Conference on Computer Aided Verification*, ser. CAV'13. Berlin, Heidelberg: Springer-Verlag, 2013, pp. 756–772.

[119] S. Itzhaky, N. Bjørner, T. Reps, M. Sagiv, and A. Thakur, "Property-directed shape analysis," in *Proceedings of the 16th International Conference on Computer Aided Verification*, ser. CAV'14. Berlin, Heidelberg: Springer-Verlag, 2014, pp. 35–51.

[120] R. Piskac, T. Wies, and D. Zufferey, "Automating separation logic using SMT," in *Proceedings of the 25th International Conference on Computer Aided Verification*, ser. CAV'13. Berlin, Heidelberg: Springer-Verlag, 2013, pp. 773–789.

[121] P. Madhusudan, G. Parlato, and X. Qiu, "Decidable logics combining heap structures and data," *SIGPLAN Not.*, vol. 46, no. 1, p. 611–622, jan 2011. [Online]. Available: https://doi.org/10.1145/1925844.1926455

[122] P. Madhusudan and X. Qiu, "Efficient decision procedures for heaps using strand," in *Static Analysis*, E. Yahav, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 43–59.

[123] R. Iosif, A. Rogalewicz, and J. Simacek, "The tree width of separation logic with recursive definitions," in *Automated Deduction – CADE-24*, M. P. Bonacina, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 21–38.

[124] S. Demri and M. Deters, "Separation logics and modalities: a survey," *Journal of Applied Non-Classical Logics*, vol. 25, pp. 50–99, 2015.

[125] I. T. Kassios, "Dynamic frames: Support for framing, dependencies and sharing without restrictions," in *FM 2006: Formal Methods*, J. Misra, T. Nipkow, and E. Sekerinski, Eds. Berlin, Heidelberg: Springer-Verlag, 2006, pp. 268–283.

[126] I. T. Kassios, "The dynamic frames theory," *Form. Asp. Comput.*, vol. 23, no. 3, pp. 267–288, May 2011.

[127] A. Banerjee, D. A. Naumann, and S. Rosenberg, "Local reasoning for global invariants, Part I: Region logic," *J. ACM*, vol. 60, no. 3, pp. 18:1–18:56, June 2013. [Online]. Available: http://doi.acm.org/10.1145/2485982

[128] A. Banerjee and D. Naumann, "Local reasoning for global invariants, Part II: Dynamic boundaries," *Journal of the ACM (JACM)*, vol. 60, 06 2013.

[129] A. Banerjee, D. A. Naumann, and S. Rosenberg, "Regional logic for local reasoning about global invariants," in *ECOOP 2008 – Object-Oriented Programming*, J. Vitek, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 387–411.

[130] J. Smans, B. Jacobs, and F. Piessens, "Implicit dynamic frames," *ACM Trans. Program. Lang. Syst.*, vol. 34, no. 1, pp. 2:1–2:58, May 2012.

[131] K. R. M. Leino and P. Müller, "A basis for verifying multi-threaded programs," in *Programming Languages and Systems*, G. Castagna, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 378–393.

[132] J. Smans, B. Jacobs, and F. Piessens, "Implicit dynamic frames: Combining dynamic frames and separation logic," in *ECOOP 2009 – Object-Oriented Programming*, S. Drossopoulou, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 148–172.

[133] M. J. Parkinson and A. J. Summers, "The relationship between separation logic and implicit dynamic frames," in *Programming Languages and Systems*, G. Barthe, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 439–458.

[134] F. Bobot and J.-C. Filliâtre, "Separation predicates: A taste of separation logic in first-order logic," in *Formal Methods and Software Engineering*, T. Aoki and K. Taguchi, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 167–181.

[135] P. W. O'Hearn, H. Yang, and J. C. Reynolds, "Separation and information hiding," in *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '04. New York, NY, USA: ACM, 2004, pp. 268–280.

[136] M. J. Parkinson and A. J. Summers, "The relationship between separation logic and implicit dynamic frames," in *Programming Languages and Systems*, G. Barthe, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 439–458.

[137] M. Sagiv, T. Reps, and R. Wilhelm, "Parametric shape analysis via 3-valued logic," *ACM Trans. Program. Lang. Syst.*, vol. 24, no. 3, p. 217–298, may 2002. [Online]. Available: https://doi.org/10.1145/514188.514190

[138] S. Kleene and M. Beeson, *Introduction to Metamathematics*. Ishi Press International, 2009. [Online]. Available: https://books.google.com/books?id=HZAjPwAACAAJ

[139] T. Lev-Ami, N. Immerman, T. Reps, M. Sagiv, S. Srivastava, and G. Yorsh, "Simulating reachability using first-order logic with applications to verification of linked data structures," in *Automated Deduction – CADE-20*, R. Nieuwenhuis, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 99–115.

[140] P. Müller, M. Schwerhoff, and A. J. Summers, "Automatic verification of iterated separating conjunctions using symbolic execution," in *Computer Aided Verification*, S. Chaudhuri and A. Farzan, Eds. Cham: Springer International Publishing, 2016, pp. 405–425.

[141] D. Distefano and M. Parkinson, "jstar: Towards practical verification for java," vol. 43, 09 2008, pp. 213–226.

[142] S. Krishna, A. J. Summers, and T. Wies, "Local reasoning for global graph properties," in *Programming Languages and Systems - 29th European Symposium on Programming, ESOP 2020, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2020, Dublin, Ireland, April 25-30, 2020, Proceedings*, ser. Lecture Notes in Computer Science, P. Müller, Ed., vol. 12075. Springer, 2020. [Online]. Available: https://doi.org/10.1007/978-3-030-44914-8_12 pp. 308–335.

[143] R. Meyer, T. Wies, and S. Wolff, "Make flows small again: Revisiting the flow framework," in *Tools and Algorithms for the Construction and Analysis of Systems: 29th International Conference, TACAS 2023, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2023, Paris, France, April 22–27, 2023, Proceedings, Part I.* Berlin, Heidelberg: Springer-Verlag, 2023. [Online]. Available: https://doi.org/10.1007/978-3-031-30823-9_32 p. 628–646.

[144] A. Hobor and J. Villard, "The ramifications of sharing in data structures," in *Proceedings of the 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '13. New York, NY, USA: Association for Computing Machinery, 2013. [Online]. Available: https://doi.org/10.1145/2429069.2429131 p. 523–536.

[145] A. Murali and P. Madhusudan, "Delta logics: Logics for change," *Technical Report*, 2024.

[146] T. Reps, M. Sagiv, and A. Loginov, "Finite differencing of logical formulas for static analysis," *ACM Trans. Program. Lang. Syst.*, vol. 32, no. 6, Aug. 2010. [Online]. Available: https://doi.org/10.1145/1749608.1749613

[147] T. Antonopoulos, N. Gorogiannis, C. Haase, M. Kanovich, and J. Ouaknine, "Foundations for decision problems in separation logic with general inductive predicates," in *Foundations of Software Science and Computation Structures*, A. Muscholl, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2014, pp. 411–425.

[148] R. Brochenin, S. Demri, and E. Lozes, "On the almighty wand," in *Computer Science Logic*, M. Kaminski and S. Martini, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 323–338.

[149] W. N. Chin, C. David, H. H. Nguyen, and S. Qin, "Automated verification of shape, size and bag properties," in *12th IEEE International Conference on Engineering Complex Computer Systems (ICECCS 2007)*, 2007, pp. 307–320.

[150] M. Sammler, R. Lepigre, R. Krebbers, K. Memarian, D. Dreyer, and D. Garg, "Refinedc: automating the foundational verification of c code with refined ownership types," in *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, ser. PLDI 2021. New York, NY, USA: Association for Computing Machinery, 2021. [Online]. Available: https://doi.org/10.1145/3453483.3454036 p. 158–174.

[151] J. Brotherston, D. Distefano, and R. L. Petersen, "Automated cyclic entailment proofs in separation logic," in *Proceedings of the 23rd International Conference on Automated Deduction*, ser. CADE'11. Berlin, Heidelberg: Springer-Verlag, 2011. [Online]. Available: http://dl.acm.org/citation.cfm?id=2032266.2032278 pp. 131–146.

[152] Q.-T. Ta, T. C. Le, S.-C. Khoo, and W.-N. Chin, "Automated mutual explicit induction proof in separation logic," in *FM 2016: Formal Methods*, J. Fitzgerald, C. Heitmeyer, S. Gnesi, and A. Philippou, Eds. Cham: Springer International Publishing, 2016, pp. 659–676.

[153] J. Brotherston, "Cyclic proofs for first-order logic with inductive definitions," in *Automated Reasoning with Analytic Tableaux and Related Methods*, B. Beckert, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 78–92.

[154] J. Brotherston, N. Gorogiannis, and R. L. Petersen, "A generic cyclic theorem prover," in *Proceedings of APLAS-10*, ser. LNCS. Springer, 2012, pp. 350–367.

[155] J.-C. Filliâtre and A. Paskevich, "Why3 — where programs meet provers," in *Programming Languages and Systems*, M. Felleisen and P. Gardner, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 125–128.

[156] F. Bobot, J.-C. Filliâtre, C. Marché, and A. Paskevich, "Why3: Shepherd Your Herd of Provers," in *Boogie 2011: First International Workshop on Intermediate Verification Languages*. Wroclaw, Poland: HAL-Inria, 2011. [Online]. Available: https://hal.inria.fr/hal-00790310 pp. 53–64.

[157] A. Banerjee, M. Barnett, and D. A. Naumann, "Boogie meets regions: A verification experience report," in *Verified Software: Theories, Tools, Experiments*, N. Shankar and J. Woodcock, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 177–191.

[158] P. Müller, M. Schwerhoff, and A. J. Summers, "Viper: A verification infrastructure for permission-based reasoning," in *Verification, Model Checking, and Abstract Interpretation (VMCAI)*, ser. LNCS, B. Jobstmann and K. R. M. Leino, Eds., vol. 9583. Springer-Verlag, 2016, pp. 41–62.

[159] S. Heule, I. T. Kassios, P. Müller, and A. J. Summers, "Verification condition generation for permission logics with abstract predicates and abstraction functions," in *ECOOP 2013 – Object-Oriented Programming*, G. Castagna, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 451–476.

[160] M. Eilers, M. Schwerhoff, and P. Müller, "Verification algorithms for automated separation logic verifiers," in *Computer Aided Verification*, A. Gurfinkel and V. Ganesh, Eds. Cham: Springer Nature Switzerland, 2024, pp. 362–386.

[161] B. Jacobs, J. Smans, P. Philippaerts, F. Vogels, W. Penninckx, and F. Piessens, "Verifast: A powerful, sound, predictable, fast verifier for c and java," in *Proceedings of the Third International Conference on NASA Formal Methods*, ser. NFM'11. Berlin, Heidelberg: Springer-Verlag, 2011, p. 41–55.

[162] D. Jost and A. J. Summers, "An automatic encoding from verifast predicates into implicit dynamic frames," in *Verified Software: Theories, Tools, Experiments*, E. Cohen and A. Rybalchenko, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2014, pp. 202–221.

[163] R. Piskac, T. Wies, and D. Zufferey, "Automating separation logic with trees and data," in *Proceedings of the 16th International Conference on Computer Aided Verification*, ser. CAV'14. Berlin, Heidelberg: Springer-Verlag, 2014, pp. 711–728.

[164] R. Piskac, T. Wies, and D. Zufferey, "Grasshopper," in *Tools and Algorithms for the Construction and Analysis of Systems*, E. Ábrahám and K. Havelund, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2014, pp. 124–139.

[165] G. Reger, M. Suda, and A. Voronkov, "Instantiation and pretending to be an smt solver with vampire," *CEUR Workshop Proceedings*, vol. 1889, pp. 63–75, Jan. 2017, 15th International Workshop on Satisfiability Modulo Theories, SMT 2017 ; Conference date: 22-07-2017 Through 23-07-2017.

[166] C. Barrett, P. Fontaine, and C. Tinelli, "The smt-lib standard: Version 2.6," 2017. [Online]. Available: http://smtlib.cs.uiowa.edu/papers/smt-lib-reference-v2.6-r2017-07-18.pdf

[167] C. Tinelli and M. Harandi, "A new correctness proof of the nelson-oppen combination procedure," in *Frontiers of Combining Systems: First International Workshop, Munich, March 1996*, F. Baader and K. U. Schulz, Eds. Dordrecht: Springer Netherlands, 1996. [Online]. Available: https://doi.org/10.1007/978-94-009-0349-4_5 pp. 103–119.

[168] T. Wies, R. Piskac, and V. Kuncak, "Combining theories with shared set operations," in *Frontiers of Combining Systems*, S. Ghilardi and R. Sebastiani, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 366–382.

[169] F. Baader and S. Ghilardi, "Connecting many-sorted theories," in *Automated Deduction – CADE-20*, R. Nieuwenhuis, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 278–294.

[170] S. Ghilardi, "Model-theoretic methods in combined constraint satisfiability," *Journal of Automated Reasoning*, vol. 33, no. 3, pp. 221–249, Oct 2004. [Online]. Available: https://doi.org/10.1007/s10817-004-6241-5

[171] P. Fontaine, "Combinations of Theories and the Bernays-Schönfinkel-Ramsey Class," in *4th International Verification Workshop - VERIFY'07*, ser. CEUR Workshop Proceedings, B. Beckert, Ed., vol. 259. Bremen, Germany: HAL-Inria, July 2007, uRL : http://sunsite.informatik.rwth-aachen.de/Publications/CEUR-WS/Vol-259/paper06.pdf. [Online]. Available: https://hal.inria.fr/inria-00186639 pp. 37–54.

[172] S. Krstic, A. Goel, J. Grundy, and C. Tinelli, "Combined satisfiability modulo parametric theories," in *Proceedings of the 13th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, ser. TACAS'07. Berlin, Heidelberg: Springer-Verlag, 2007, p. 602–617.

[173] C. Tinelli and C. G. Zarba, "Combining nonstably infinite theories," *Journal of Automated Reasoning*, vol. 34, no. 3, pp. 209–238, Apr 2005. [Online]. Available: https://doi.org/10.1007/s10817-005-5204-9

[174] C. Ihlemann, S. Jacobs, and V. Sofronie-Stokkermans, "On local reasoning in verification," in *Tools and Algorithms for the Construction and Analysis of Systems*, C. R. Ramakrishnan and J. Rehof, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 265–281.

[175] V. Sofronie-Stokkermans, "Locality results for certain extensions of theories with bridging functions," in *Automated Deduction – CADE-22*, R. A. Schmidt, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 67–83.

[176] Z. Manna, H. B. Sipma, and T. Zhang, "Verifying balanced trees," in *Logical Foundations of Computer Science*, S. N. Artemov and A. Nerode, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 363–378.

[177] T. Zhang, H. B. Sipma, and Z. Manna, "Decision procedures for term algebras with integer constraints," *Information and Computation*, vol. 204, no. 10, pp. 1526–1574, 2006, combining Logical Systems. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0890540106000630

[178] H. Hojjat and P. Rümmer, "Deciding and interpolating algebraic data types by reduction," in *19th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing, SYNASC 2017, Timisoara, Romania, September 21-24, 2017*, T. Jebelean, V. Negru, D. Petcu, D. Zaharie, T. Ida, and S. M. Watt, Eds. Los Alamitos, CA, USA: IEEE Computer Society, 2017. [Online]. Available: https://doi.org/10.1109/SYNASC.2017.00033 pp. 145–152.

[179] D. Kapur, R. Majumdar, and C. G. Zarba, "Interpolation for data structures," in *Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. SIGSOFT '06/FSE-14. New York, NY, USA: Association for Computing Machinery, 2006. [Online]. Available: https://doi.org/10.1145/1181775.1181789 p. 105–116.

[180] D. Monniaux, "Quantifier elimination by lazy model enumeration," in *Computer Aided Verification*, T. Touili, B. Cook, and P. Jackson, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 585–599.

[181] R. Stansifer, "Presburger's article on integer airthmetic: Remarks and translation," Cornell University, Computer Science Department, Tech. Rep. TR84-639, September 1984. [Online]. Available: http://techreports.library.cornell.edu:8081/Dienst/UI/1.0/Display/cul.cs/TR84-639

[182] S. Cruanes, S. Schulz, and P. Vukmirović, "Faster, Higher, Stronger: E 2.3," in *TACAS 2019*, ser. LNAI, vol. 11716, Prague, Czech Republic, Apr. 2019. [Online]. Available: https://inria.hal.science/hal-02296188 pp. 495–507.

[183] P. Backeman and P. Rümmer, "Theorem proving with bounded rigid e-unification," in *Automated Deduction – CADE-25 :*, ser. Lecture Notes in Computer Science, no. 9195, 2015, pp. 572–587.

[184] J. Otten, "leancop 2.0 and ileancop 1.2: High performance lean theorem proving in classical and intuitionistic logic (system descriptions)," in *Proceedings of the 4th International Joint Conference on Automated Reasoning*, ser. IJCAR '08. Berlin, Heidelberg: Springer-Verlag, 2008. [Online]. Available: https://doi.org/10.1007/978-3-540-71070-7_23 p. 283–291.

[185] D. Detlefs, G. Nelson, and J. B. Saxe, "Simplify: A theorem prover for program checking," *J. ACM*, vol. 52, no. 3, p. 365–473, May 2005. [Online]. Available: https://doi.org/10.1145/1066100.1066102

[186] L. de Moura and N. Bjørner, "Efficient e-matching for smt solvers," in *Automated Deduction – CADE-21*, F. Pfenning, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 183–198.

[187] Y. Ge, C. Barrett, and C. Tinelli, "Solving quantified verification conditions using satisfiability modulo theories," in *Automated Deduction – CADE-21*, F. Pfenning, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 167–182.

[188] Y. Ge and L. de Moura, "Complete instantiation for quantified formulas in satisfiabiliby modulo theories," in *Computer Aided Verification*, A. Bouajjani and O. Maler, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 306–320.

[189] A. Reynolds, C. Tinelli, and L. de Moura, "Finding conflicting instances of quantified formulas in smt," in *Proceedings of the 14th Conference on Formal Methods in Computer-Aided Design*, ser. FMCAD '14. Austin, Texas: FMCAD Inc, 2014, p. 195–202.

[190] H. Barbosa, P. Fontaine, and A. Reynolds, "Congruence closure with free variables," in *Tools and Algorithms for the Construction and Analysis of Systems*, A. Legay and T. Margaria, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2017, pp. 214–230.

[191] A. Reynolds, H. Barbosa, and P. Fontaine, "Revisiting Enumerative Instantiation," in *TACAS 2018 - 24th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, ser. LNCS, D. Beyer and M. Huisman, Eds., vol. 10806. Thessaloniki, Greece: Springer, Apr. 2018. [Online]. Available: https://hal.science/hal-01877055 p. 20.

[192] A. R. Bradley, Z. Manna, and H. B. Sipma, "What's decidable about arrays?" in *Verification, Model Checking, and Abstract Interpretation*, E. A. Emerson and K. S. Namjoshi, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 427–442.

[193] I. Garcia-Contreras, V. K. H. Govind, S. Shoham, and A. Gurfinkel, "Fast approximations of quantifier elimination," in *Computer Aided Verification*, C. Enea and A. Lal, Eds. Cham: Springer Nature Switzerland, 2023, pp. 64–86.

[194] K. R. M. Leino, "This is boogie 2," June 2008, technical Report. [Online]. Available: https://www.microsoft.com/en-us/research/publication/this-is-boogie-2-2/

[195] T. dafny-lang community, 2022. [Online]. Available: https://dafny-lang.github.io/dafny/DafnyRef/DafnyRef

[196] M. Moskal, "Programming with triggers," in *Proceedings of the 7th International Workshop on Satisfiability Modulo Theories*, ser. SMT '09. New York, NY, USA: Association for Computing Machinery, 2009. [Online]. Available: https://doi.org/10.1145/1670412.1670416 p. 20–29.

[197] K. R. M. Leino and C. Pit-Claudel, "Trigger selection strategies to stabilize program verifiers," in *Computer Aided Verification*, S. Chaudhuri and A. Farzan, Eds. Cham: Springer International Publishing, 2016, pp. 361–381.

[198] Y. M. Y. Feldman, O. Padon, N. Immerman, M. Sagiv, and S. Shoham, "Bounded quantifier instantiation for checking inductive invariants," in *Tools and Algorithms for the Construction and Analysis of Systems*, A. Legay and T. Margaria, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2017, pp. 76–95.

[199] M. Y. Vardi, "The complexity of relational query languages (extended abstract)," in *Proceedings of the Fourteenth Annual ACM Symposium on Theory of Computing*, ser. STOC '82. New York, NY, USA: Association for Computing Machinery, 1982. [Online]. Available: https://doi.org/10.1145/800070.802186 p. 137–146.

[200] N. Immerman, "Relational queries computable in polynomial time (extended abstract)," in *Proceedings of the Fourteenth Annual ACM Symposium on Theory of Computing*, ser. STOC '82. New York, NY, USA: Association for Computing Machinery, 1982. [Online]. Available: https://doi.org/10.1145/800070.802187 p. 147–152.

[201] A. V. Aho and J. D. Ullman, "Universality of data retrieval languages," in *Proceedings of the 6th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, ser. POPL '79. New York, NY, USA: Association for Computing Machinery, 1979. [Online]. Available: https://doi.org/10.1145/567752.567763 p. 110–119.

[202] A. K. Chandra and D. Harel, "Structure and complexity of relational queries," in *Proceedings of the 21st Annual Symposium on Foundations of Computer Science*, ser. SFCS '80. USA: IEEE Computer Society, 1980. [Online]. Available: https://doi.org/10.1109/SFCS.1980.41 p. 333–347.

[203] M. Kaufmann and J. S. Moore, "An industrial strength theorem prover for a logic based on common lisp," *IEEE Trans. Softw. Eng.*, vol. 23, no. 4, p. 203–213, Apr. 1997. [Online]. Available: https://doi.org/10.1109/32.588534

[204] M. Kaufmann, J. S. Moore, and P. Manolios, *Computer-Aided Reasoning: An Approach*. USA: Kluwer Academic Publishers, 2000.

[205] H. H. Nguyen and W.-N. Chin, "Enhancing program verification with lemmas," in *Proceedings of the 20th International Conference on Computer Aided Verification*, ser. CAV '08. Berlin, Heidelberg: Springer-Verlag, 2008. [Online]. Available: https://doi.org/10.1007/978-3-540-70545-1_34 p. 355–369.

[206] N. Amin, K. R. M. Leino, and T. Rompf, "Computing with an smt solver," in *Tests and Proofs*, M. Seidl and N. Tillmann, Eds. Cham: Springer International Publishing, 2014, pp. 20–35.

[207] R. Leino and N. Polikarpova, "Verified calculations," March 2013. [Online]. Available: https://www.microsoft.com/en-us/research/publication/verified-calculations/

[208] H. R. Chamarthi, P. Dillinger, P. Manolios, and D. Vroon, "The acl2 sedan theorem proving system," in *Proceedings of the 17th International Conference on Tools and Algorithms for the Construction and Analysis of Systems: Part of the Joint European Conferences on Theory and Practice of Software*, ser. TACAS'11/ETAPS'11. Berlin, Heidelberg: Springer-Verlag, 2011, p. 291–295.

[209] G. Passmore, S. Cruanes, D. Ignatovich, D. Aitken, M. Bray, E. Kagan, K. Kanishev, E. Maclean, and N. Mometto, "The imandra automated reasoning system (system description)," in *Automated Reasoning*, N. Peltier and V. Sofronie-Stokkermans, Eds. Cham: Springer International Publishing, 2020, pp. 464–471.

[210] K. R. M. Leino, "Automating induction with an smt solver," in *Proceedings of the 13th International Conference on Verification, Model Checking, and Abstract Interpretation*, ser. VMCAI'12. Berlin, Heidelberg: Springer-Verlag, 2012. [Online]. Available: https://doi.org/10.1007/978-3-642-27940-9_21 p. 315–331.

[211] B. Jacobs, J. Smans, P. Philippaerts, F. Vogels, W. Penninckx, and F. Piessens, "Verifast: A powerful, sound, predictable, fast verifier for c and java," in *Proceedings of the Third International Conference on NASA Formal Methods*, ser. NFM'11. Berlin, Heidelberg: Springer-Verlag, 2011, p. 41–55.

[212] Q. L. Le, M. Tatsuta, J. Sun, and W.-N. Chin, "A decidable fragment in separation logic with inductive predicates and arithmetic," 07 2017, pp. 495–517.

[213] A. Gurfinkel, "Program verification with constrained horn clauses (invited paper)," in *Computer Aided Verification*, S. Shoham and Y. Vizel, Eds. Cham: Springer International Publishing, 2022, pp. 19–29.

[214] A. Gurfinkel, T. Kahsai, A. Komuravelli, and J. A. Navas, "The seahorn verification framework," in *Computer Aided Verification*, D. Kroening and C. S. Păsăreanu, Eds. Cham: Springer International Publishing, 2015, pp. 343–361.

[215] A. Komuravelli, A. Gurfinkel, and S. Chaki, "Smt-based model checking for recursive programs," in *Computer Aided Verification*, A. Biere and R. Bloem, Eds. Cham: Springer International Publishing, 2014, pp. 17–34.

[216] E. De Angelis, F. Fioravanti, A. Pettorossi, and M. Proietti, "Solving horn clauses on inductive data types without induction," *Theory and Practice of Logic Programming*, vol. 18, no. 3-4, p. 452–469, 2018.

[217] H. Govind V K, S. Shoham, and A. Gurfinkel, "Solving constrained horn clauses modulo algebraic data types and recursive functions," *Proc. ACM Program. Lang.*, vol. 6, no. POPL, jan 2022. [Online]. Available: https://doi.org/10.1145/3498722

[218] A. Reynolds and J. C. Blanchette, "A decision procedure for (co)datatypes in smt solvers," *J. Autom. Reason.*, vol. 58, no. 3, p. 341–362, mar 2017. [Online]. Available: https://doi.org/10.1007/s10817-016-9372-6

[219] T. Rybina and A. Voronkov, "A decision procedure for term algebras with queues," *ACM Trans. Comput. Logic*, vol. 2, no. 2, p. 155–181, apr 2001. [Online]. Available: https://doi.org/10.1145/371316.371494

[220] L. Wos, G. A. Robinson, and D. F. Carson, "Efficiency and completeness of the set of support strategy in theorem proving," *J. ACM*, vol. 12, no. 4, p. 536–541, oct 1965. [Online]. Available: https://doi.org/10.1145/321296.321302

[221] F. Haifani, S. Tourret, and C. Weidenbach, "Generalized completeness for sos resolution and its application to a new notion of relevance," in *Automated Deduction – CADE 28*, A. Platzer and G. Sutcliffe, Eds. Cham: Springer International Publishing, 2021, pp. 327–343.

[222] M. E. Stickel, "Automated deduction by theory resolution," *J. Autom. Reason.*, vol. 1, no. 4, p. 333–355, Dec. 1985.

[223] H. Ganzinger and K. Korovin, "New directions in instantiation-based theorem proving," in *Proceedings of the 18th Annual IEEE Symposium on Logic in Computer Science*, ser. LICS '03. USA: IEEE Computer Society, 2003, p. 55.

[224] K. Korovin and C. Sticksel, "iprover-eq: An instantiation-based theorem prover with equality," in *Automated Reasoning*, J. Giesl and R. Hähnle, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 196–202.

[225] S. Gulwani, "Automating string processing in spreadsheets using input-output examples," *SIGPLAN Not.*, vol. 46, no. 1, p. 317–330, jan 2011. [Online]. Available: https://doi.org/10.1145/1925844.1926423

[226] O. Polozov and S. Gulwani, "Flashmeta: A framework for inductive program synthesis," in *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, ser. OOPSLA 2015. New York, NY, USA: Association for Computing Machinery, 2015. [Online]. Available: https://doi.org/10.1145/2814270.2814310 p. 107–126.

[227] E. Torlak and R. Bodik, "Growing solver-aided languages with rosette," in *Proceedings of the 2013 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software*, ser. Onward! 2013. New York, NY, USA: Association for Computing Machinery, 2013. [Online]. Available: https://doi.org/10.1145/2509578.2509586 p. 135–152.

[228] E. Torlak and R. Bodik, "A lightweight symbolic virtual machine for solver-aided host languages," *SIGPLAN Not.*, vol. 49, no. 6, p. 530–541, June 2014. [Online]. Available: https://doi.org/10.1145/2666356.2594340

[229] Z. Manna and R. Waldinger, "A deductive approach to program synthesis," *ACM Trans. Program. Lang. Syst.*, vol. 2, no. 1, p. 90–121, Jan. 1980. [Online]. Available: https://doi.org/10.1145/357084.357090

[230] J. Kim, Q. Hu, L. D'Antoni, and T. Reps, "Semantics-guided synthesis," *Proc. ACM Program. Lang.*, vol. 5, no. POPL, jan 2021. [Online]. Available: https://doi.org/10.1145/3434311

[231] L. D'Antoni, Q. Hu, J. Kim, and T. Reps, "Programmable program synthesis," in *Computer Aided Verification*, A. Silva and K. R. M. Leino, Eds. Cham: Springer International Publishing, 2021, pp. 84–109.

[232] D. Beyer, T. A. Henzinger, R. Majumdar, and A. Rybalchenko, "Invariant synthesis for combined theories," in *Verification, Model Checking, and Abstract Interpretation*, B. Cook and A. Podelski, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007. [Online]. Available: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.65.4364&rep=rep1&type=pdf pp. 378–394.

[233] S. Gulwani, S. Srivastava, and R. Venkatesan, "Program analysis as constraint solving," in *Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation, Tucson, AZ, USA, June 7-13, 2008*, 2008. [Online]. Available: http://doi.acm.org/10.1145/1375581.1375616 pp. 281–292.

[234] A. Gupta and A. Rybalchenko, "Invgen: An efficient invariant generator," in *Computer Aided Verification*, A. Bouajjani and O. Maler, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 634–640.

[235] M. A. Colón, S. Sankaranarayanan, and H. B. Sipma, "Linear invariant generation using non-linear constraint solving," in *Computer Aided Verification*, W. A. Hunt and F. Somenzi, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2003, pp. 420–432.

[236] A. Gupta, R. Majumdar, and A. Rybalchenko, "From tests to proofs," in *Tools and Algorithms for the Construction and Analysis of Systems*, S. Kowalewski and A. Philippou, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 262–276.

[237] C. Flanagan and K. R. M. Leino, "Houdini, an annotation assistant for esc/java," in *FME 2001: Formal Methods for Increasing Software Productivity*, J. N. Oliveira and P. Zave, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2001, pp. 500–517.

[238] M. D. Ernst, A. Czeisler, W. G. Griswold, and D. Notkin, "Quickly detecting relevant program invariants," in *Proceedings of the 22nd International Conference on Software Engineering*, ser. ICSE '00. New York, NY, USA: Association for Computing Machinery, 2000. [Online]. Available: https://doi.org/10.1145/337180.337240 p. 449–458.

[239] M. M. Zloof, "Query by example," in *Proceedings of the May 19-22, 1975, National Computer Conference and Exposition*, ser. AFIPS '75. New York, NY, USA: Association for Computing Machinery, 1975. [Online]. Available: https://doi.org/10.1145/1499949.1500034 p. 431–438.

[240] A. Thakkar, A. Naik, N. Sands, R. Alur, M. Naik, and M. Raghothaman, "Example-guided synthesis of relational queries," in *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, ser. PLDI 2021. New York, NY, USA: Association for Computing Machinery, 2021. [Online]. Available: https://doi.org/10.1145/3453483.3454098 p. 1110–1125.

[241] C. Wang, A. Cheung, and R. Bodik, "Synthesizing highly expressive sql queries from input-output examples," *SIGPLAN Not.*, vol. 52, no. 6, p. 452–466, June 2017. [Online]. Available: https://doi.org/10.1145/3140587.3062365

[242] J. R. Koenig, O. Padon, N. Immerman, and A. Aiken, "First-order quantified separators," in *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI 2020. New York, NY, USA: Association for Computing Machinery, 2020. [Online]. Available: https://doi.org/10.1145/3385412.3386018 p. 703–717.

[243] P. Krogmeier and P. Madhusudan, "Learning formulas in finite variable logics," *Proc. ACM Program. Lang.*, vol. 6, no. POPL, Jan 2022. [Online]. Available: https://doi.org/10.1145/3498671

[244] X. Wang, "An efficient programming-by-example framework," Ph.D. dissertation, The University of Texas at Austin, 2019.

[245] X. Wang, I. Dillig, and R. Singh, "Program synthesis using abstraction refinement," *Proc. ACM Program. Lang.*, vol. 2, no. POPL, Dec. 2017. [Online]. Available: https://doi.org/10.1145/3158151

[246] T. Hance, M. Heule, R. Martins, and B. Parno, "Finding invariants of distributed systems: It's a small (enough) world after all," in *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI '21)*. Boston, MA, USA: USENIX Association, Apr. 2021. [Online]. Available: https://www.usenix.org/conference/nsdi21/presentation/hance pp. 115–131.

[247] J. Yao, R. Tao, R. Gu, J. Nieh, S. Jana, and G. Ryan, "DistAI: Data-Driven automated invariant learning for distributed protocols," in *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*. USENIX Association, July 2021. [Online]. Available: https://www.usenix.org/conference/osdi21/presentation/yao pp. 405–421.

[248] G. Ryan, J. Wong, J. Yao, R. Gu, and S. Jana, "Cln2inv: Learning loop invariants with continuous logic networks," in *International Conference on Learning Representations*, 2020. [Online]. Available: https://openreview.net/forum?id=HJlfuTEtvB

[249] M. Fischer, M. Balunovic, D. Drachsler-Cohen, T. Gehr, C. Zhang, and M. Vechev, "DL2: Training and querying neural networks with logic," in *Proceedings of the 36th International Conference on Machine Learning*, ser. Proceedings of Machine Learning Research, K. Chaudhuri and R. Salakhutdinov, Eds., vol. 97. PMLR, 09–15 Jun 2019. [Online]. Available: https://proceedings.mlr.press/v97/fischer19a.html pp. 1931–1941.

[250] M. Janota, J. Piepenbrock, and B. Piotrowski, "Towards Learning Quantifier Instantiation in SMT," in *25th International Conference on Theory and Applications of Satisfiability Testing (SAT 2022)*, ser. Leibniz International Proceedings in Informatics (LIPIcs), K. S. Meel and O. Strichman, Eds., vol. 236. Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2022. [Online]. Available: https://drops.dagstuhl.de/entities/document/10.4230/LIPIcs.SAT.2022.7 pp. 7:1–7:18.

[251] J. Jakubův, M. Janota, J. Piepenbrock, and J. Urban, "Machine learning for quantifier selection in cvc5," 2024. [Online]. Available: https://arxiv.org/abs/2408.14338

[252] A. Niemetz, M. Preiner, A. Reynolds, C. Barrett, and C. Tinelli, "Syntax-guided quantifier instantiation," in *Tools and Algorithms for the Construction and Analysis of Systems*, J. F. Groote and K. G. Larsen, Eds. Cham: Springer International Publishing, 2021, pp. 145–163.

[253] R. Alur, P. Černý, P. Madhusudan, and W. Nam, "Synthesis of interface specifications for java classes," in *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '05. New York, NY, USA: Association for Computing Machinery, 2005. [Online]. Available: https://doi.org/10.1145/1040305.1040314 p. 98–109.

[254] J. M. Cobleigh, D. Giannakopoulou, and C. S. Păsăreanu, "Learning assumptions for compositional verification," in *Proceedings of the 9th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, ser. TACAS'03. Berlin, Heidelberg: Springer-Verlag, 2003, p. 331–346.

[255] A. Astorga, S. Saha, A. Dinkins, F. Wang, P. Madhusudan, and T. Xie, "Synthesizing contracts correct modulo a test generator," *Proc. ACM Program. Lang.*, vol. 5, no. OOPSLA, Oct. 2021. [Online]. Available: https://doi.org/10.1145/3485481

[256] P. Ezudheen, D. Neider, D. D'Souza, P. Garg, and P. Madhusudan, "Horn-ice learning for synthesizing invariants and contracts," *Proc. ACM Program. Lang.*, vol. 2, no. OOPSLA, Oct. 2018. [Online]. Available: https://doi.org/10.1145/3276501

[257] Z. Zhou, R. Dickerson, B. Delaware, and S. Jagannathan, "Data-driven abductive inference of library specifications," *Proc. ACM Program. Lang.*, vol. 5, no. OOPSLA, Oct. 2021. [Online]. Available: https://doi.org/10.1145/3485493

[258] G. Fedyukovich, Y. Zhang, and A. Gupta, "Syntax-guided termination analysis," in *Computer Aided Verification*, H. Chockler and G. Weissenbacher, Eds. Cham: Springer International Publishing, 2018, pp. 124–143.

[259] Y. Sarita, A. Singh, S. Gomber, G. Singh, and M. Vishwanathan, "Syndicate: Synergistic synthesis of ranking function and invariants for termination analysis," 2024. [Online]. Available: https://arxiv.org/abs/2404.05951

[260] C. Urban, A. Gurfinkel, and T. Kahsai, "Synthesizing ranking functions from bits and pieces," in *Tools and Algorithms for the Construction and Analysis of Systems*, M. Chechik and J.-F. Raskin, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2016, pp. 54–70.

[261] J. Kim, L. D'Antoni, and T. Reps, "Unrealizability logic," *Proc. ACM Program. Lang.*, vol. 7, no. POPL, Jan. 2023. [Online]. Available: https://doi.org/10.1145/3571216

[262] S. Nagy, J. Kim, T. Reps, and L. D'Antoni, "Automating unrealizability logic: Hoare-style proof synthesis for infinite sets of programs," *Proc. ACM Program. Lang.*, vol. 8, no. OOPSLA2, Oct. 2024. [Online]. Available: https://doi.org/10.1145/3689715

[263] D. Angluin, "Queries and concept learning," *Mach. Learn.*, vol. 2, no. 4, p. 319–342, Apr. 1988. [Online]. Available: https://doi.org/10.1023/A:1022821128753

[264] S. Jha and S. A. Seshia, "A theory of formal synthesis via inductive learning," *Acta Inf.*, vol. 54, no. 7, p. 693–726, Nov. 2017. [Online]. Available: https://doi.org/10.1007/s00236-017-0294-5

[265] S. Jha, S. Gulwani, S. A. Seshia, and A. Tiwari, "Oracle-guided component-based program synthesis," in *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ser. ICSE '10. New York, NY, USA: Association for Computing Machinery, 2010. [Online]. Available: https://doi.org/10.1145/1806799.1806833 p. 215–224.

[266] R. P. Kurshan, *Computer-Aided Verification of Coordinating Processes: The Automata-Theoretic Approach*. USA: Princeton University Press, 1994.

[267] R. P. Kurshan, "Model checking and abstraction," in *Abstraction, Reformulation, and Approximation*, S. Koenig and R. C. Holte, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2002, pp. 1–17.

[268] C. Löding, P. Madhusudan, and D. Neider, "Abstract learning frameworks for synthesis," in *Proceedings of the 22nd International Conference on Tools and Algorithms for the Construction and Analysis of Systems - Volume 9636*. Berlin, Heidelberg: Springer-Verlag, 2016. [Online]. Available: https://doi.org/10.1007/978-3-662-49674-9_10 p. 167–185.

[269] A. Ireland and A. Bundy, "Productive use of failure in inductive proof," in *Automated Mathematical Induction*, H. Zhang, Ed. Dordrecht: Springer Netherlands, 1996. [Online]. Available: https://doi.org/10.1007/978-94-009-1675-3_3 pp. 79–111.

[270] M. Johansson, L. Dixon, and A. Bundy, "Case-analysis for rippling and inductive proof," in *Proceedings of the First International Conference on Interactive Theorem Proving*, ser. ITP'10. Berlin, Heidelberg: Springer-Verlag, 2010. [Online]. Available: https://doi.org/10.1007/978-3-642-14052-5_21 p. 291–306.

[271] K. Claessen, M. Johansson, D. Rosén, and N. Smallbone, "Automating inductive proofs using theory exploration," in *Automated Deduction – CADE-24*, M. P. Bonacina, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 392–406.

[272] M. Hajdú, P. Hozzová, L. Kovács, J. Schoisswohl, and A. Voronkov, "Induction with generalization in superposition reasoning," in *Intelligent Computer Mathematics: 13th International Conference, CICM 2020, Bertinoro, Italy, July 26–31, 2020, Proceedings*. Berlin, Heidelberg: Springer-Verlag, 2020. [Online]. Available: https://doi.org/10.1007/978-3-030-53518-6_8 p. 123–137.

[273] S. Cruanes, "Superposition with structural induction," in *Frontiers of Combining Systems*, C. Dixon and M. Finger, Eds. Cham: Springer International Publishing, 2017, pp. 172–188.

[274] H. Unno, S. Torii, and H. Sakamoto, "Automating induction for solving horn clauses," in *Computer Aided Verification*, R. Majumdar and V. Kunčak, Eds. Cham: Springer International Publishing, 2017, pp. 571–591.

[275] D.-H. Chu, J. Jaffar, and M.-T. Trinh, "Automatic induction proofs of data-structures in imperative programs," in *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '15. New York, NY, USA: Association for Computing Machinery, 2015. [Online]. Available: https://doi.org/10.1145/2737924.2737984 p. 457–466.

[276] H. Zhang, A. Gupta, and S. Malik, "Syntax-guided synthesis for lemma generation in hardware model checking," in *Verification, Model Checking, and Abstract Interpretation - 22nd International Conference, VMCAI 2021, Copenhagen, Denmark, January 17-19, 2021, Proceedings*, ser. Lecture Notes in Computer Science, F. Henglein, S. Shoham, and Y. Vizel, Eds., vol. 12597. Springer, 2021. [Online]. Available: https://doi.org/10.1007/978-3-030-67067-2_15 pp. 325–349.

[277] W. Sonnex, S. Drossopoulou, and S. Eisenbach, "Zeno: An automated prover for properties of recursive data structures," in *Tools and Algorithms for the Construction and Analysis of Systems*, C. Flanagan and B. König, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 407–421.

[278] M. Hajdú, P. Hozzová, L. Kovács, J. Schoisswohl, and A. Voronkov, "Induction with generalization in superposition reasoning," in *Intelligent Computer Mathematics - 13th International Conference, CICM 2020, Bertinoro, Italy, July 26-31, 2020, Proceedings*, ser. Lecture Notes in Computer Science, C. Benzmüller and B. R. Miller, Eds., vol. 12236. Springer, 2020. [Online]. Available: https://doi.org/10.1007/978-3-030-53518-6_8 pp. 123–137.

[279] K. Claessen, M. Johansson, D. Rosén, and N. Smallbone, "Automating inductive proofs using theory exploration," in *Automated Deduction - CADE-24 - 24th International Conference on Automated Deduction, Lake Placid, NY, USA, June 9-14, 2013. Proceedings*, ser. Lecture Notes in Computer Science, M. P. Bonacina, Ed., vol. 7898. Springer, 2013. [Online]. Available: https://doi.org/10.1007/978-3-642-38574-2_27 pp. 392–406.

[280] M. Johansson, "Lemma discovery for induction - A survey," in *Intelligent Computer Mathematics - 12th International Conference, CICM 2019, Prague, Czech Republic, July 8-12, 2019, Proceedings*, ser. Lecture Notes in Computer Science, C. Kaliszyk, E. C. Brady, A. Kohlhase, and C. S. Coen, Eds., vol. 11617. Springer, 2019. [Online]. Available: https://doi.org/10.1007/978-3-030-23250-4_9 pp. 125–139.

[281] A. Sivaraman, A. Sanchez-Stern, B. Chen, S. Lerner, and T. Millstein, "Data-driven lemma synthesis for interactive proofs," *Proc. ACM Program. Lang.*, vol. 6, no. OOPSLA2, oct 2022. [Online]. Available: https://doi.org/10.1145/3563306

[282] C. Kurashige, R. Ji, A. Giridharan, M. Barbone, D. Noor, S. Itzhaky, R. Jhala, and N. Polikarpova, "Cclemma: E-graph guided lemma discovery for inductive equational proofs," *Proc. ACM Program. Lang.*, vol. 8, no. ICFP, Aug. 2024. [Online]. Available: https://doi.org/10.1145/3674653

[283] D. Neider, P. Garg, P. Madhusudan, S. Saha, and D. Park, "Invariant synthesis for incomplete verification engines," in *Tools and Algorithms for the Construction and Analysis of Systems*, D. Beyer and M. Huisman, Eds. Cham: Springer International Publishing, 2018, pp. 232–250.

[284] A. Tarski, "Ein beitrag zur axiomatik der abelschen gruppen," *Fundamenta Mathematicae*, vol. 30, no. 11, p. 253–256, 1938.

[285] A. Rezus, *Witness Theory: Notes on λ-calculus and Logic*. London, UK: College Publications, 03 2020.

[286] J. Łukasiewicz and A. Tarski, *Untersuchungen über den Aussagenkalkül*. Warsaw, Poland: Comptes Rendus des Séances de la Sociélé des Scierices et des Lettres des Varsovie Classe III, 1930, vol. 23.

[287] K. Kunen, "Single axioms for groups," *Journal of Automated Reasoning*, vol. 9, no. 3, pp. 291–308, Dec 1992. [Online]. Available: https://doi.org/10.1007/BF00245293

[288] W. W. Mccune, "Single axioms for groups and abelian groups with various operations," *Journal of Automated Reasoning*, vol. 10, pp. 1–13, 1993.

[289] W. McCune and A. D. Sands, "Computer and human reasoning: Single implicative axioms for groups and for abelian groups," *The American Mathematical Monthly*, vol. 103, no. 10, pp. 888–892, 1996. [Online]. Available: http://www.jstor.org/stable/2974613

[290] W. McCune, R. Veroff, B. Fitelson, K. Harris, A. Feist, and L. Wos, "Short single axioms for boolean algebra," *Journal of Automated Reasoning*, vol. 29, no. 1, pp. 1–16, Mar 2002. [Online]. Available: https://doi.org/10.1023/A:1020542009983

[291] W. McCune and R. Padmanabhan, "Single identities for lattice theory and for weakly associative lattices," *Algebra Universalis*, vol. 36, pp. 436–449, 12 1996.

[292] W. McCune, R. Padmanabhan, M. A. Rose, and R. Veroff, "Automated discovery of single axioms for ortholattices," *algebra universalis*, vol. 52, no. 4, pp. 541–549, Feb 2005. [Online]. Available: https://doi.org/10.1007/s00012-004-1902-0

[293] W. McCune, R. Padmanabhan, and R. Veroff, "Yet another single law for lattices," *algebra universalis*, vol. 50, no. 2, pp. 165–169, Dec 2003. [Online]. Available: https://doi.org/10.1007/s00012-003-1832-2

[294] B. Neumann, "Another single law for groups," *Bulletin of the Australian Mathematical Society*, vol. 23, no. 1, p. 81–102, 1981.

[295] R. Padmanabhan, "On single equational-axiom systems for abelian groups," *Journal of the Australian Mathematical Society*, vol. 9, no. 1-2, p. 143–152, 1969.

[296] I. L. Valbuena and M. Johansson, "Conditional lemma discovery and recursion induction in hipster," *Electronic Communications of the EASST*, vol. 72, pp. 1–15, 2015. [Online]. Available: https://doi.org/10.14279/tuj.eceasst.72.1009

[297] M. Johansson, "Automated theory exploration for interactive theorem proving:," in *Interactive Theorem Proving*, M. Ayala-Rincón and C. A. Muñoz, Eds. Cham: Springer International Publishing, 2017, pp. 1–11.

[298] M. Johansson, D. Rosén, N. Smallbone, and K. Claessen, "Hipster: Integrating theory exploration in a proof assistant," in *Intelligent Computer Mathematics*, S. M. Watt, J. H. Davenport, A. P. Sexton, P. Sojka, and J. Urban, Eds. Cham: Springer International Publishing, 2014, pp. 108–122.

[299] B. Buchberger, A. Crăciun, T. Jebelean, L. Kovács, T. Kutsia, K. Nakagawa, F. Piroi, N. Popov, J. Robu, M. Rosenkranz, and W. Windsteiger, "Theorema: Towards computer-aided mathematical theory exploration," *Journal of Applied Logic*, vol. 4, no. 4, pp. 470–504, 2006. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1570868305000716

[300] I. Drâmnesc, T. Jebelean, and S. Stratulat, "Theory exploration of binary trees," in *2015 IEEE 13th International Symposium on Intelligent Systems and Informatics (SISY)*. Subotica, Serbia: IEEE, 2015, pp. 139–144.

[301] I. Drămnesc and T. Jebelean, "Theory exploration in theorema: Case study on lists," in *2012 7th IEEE International Symposium on Applied Computational Intelligence and Informatics (SACI)*. Timisoara, Romania: IEEE, 2012, pp. 421–426.

[302] R. L. Mccasland, A. Bundy, and P. F. Smith, "Mathsaid: Automated mathematical theory exploration," *Applied Intelligence*, vol. 47, no. 3, p. 585–606, oct 2017. [Online]. Available: https://doi.org/10.1007/s10489-017-0954-8

[303] E. Singher and S. Itzhaky, "Theory exploration powered by deductive synthesis," in *Computer Aided Verification*, A. Silva and K. R. M. Leino, Eds. Cham: Springer International Publishing, 2021. [Online]. Available: https://link.springer.com/chapter/10.1007/978-3-030-81688-9_6 pp. 125–148.

[304] A. Pal, B. Saiki, R. Tjoa, C. Richey, A. Zhu, O. Flatt, M. Willsey, Z. Tatlock, and C. Nandi, "Equality saturation theory exploration à la carte," *Proc. ACM Program. Lang.*, vol. 7, no. OOPSLA2, Oct. 2023. [Online]. Available: https://doi.org/10.1145/3622834

[305] G. Poesia, D. Broman, N. Haber, and N. D. Goodman, "Learning formal mathematics from intrinsic motivation," *arXiv preprint arXiv:2407.00695*, 2024.

[306] A. Grayeli, A. Sehgal, O. Costilla-Reyes, M. Cranmer, and S. Chaudhuri, "Symbolic regression with a learned concept library," 2024. [Online]. Available: https://arxiv.org/abs/2409.09359

[307] C. Singh, J. X. Morris, J. Aneja, A. M. Rush, and J. Gao, "Explaining patterns in data with language models via interpretable autoprompting," 2022. [Online]. Available: https://arxiv.org/abs/2210.01848

[308] P. Langley, "Data-driven discovery of physical laws," *Cognitive Science*, vol. 5, no. 1, pp. 31–54, 1981. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0364021381800250

[309] T. Wu and M. Tegmark, "Toward an artificial intelligence physicist for unsupervised learning." *Physical review. E*, vol. 100 3-1, p. 033311, 2019.

[310] J. C. Reynolds, "Definitional interpreters for higher-order programming languages," in *Proceedings of the ACM Annual Conference - Volume 2*, ser. ACM '72. New York, NY, USA: Association for Computing Machinery, 1972. [Online]. Available: https://doi.org/10.1145/800194.805852 p. 717–740.

[311] N. Lehmann, A. T. Geller, N. Vazou, and R. Jhala, "Flux: Liquid types for rust," *Proc. ACM Program. Lang.*, vol. 7, no. PLDI, June 2023. [Online]. Available: https://doi.org/10.1145/3591283

[312] C. Enea, M. Sighireanu, and Z. Wu, "On automated lemma generation for separation logic with inductive definitions," in *Automated Technology for Verification and Analysis*, B. Finkbeiner, G. Pu, and L. Zhang, Eds. Cham: Springer International Publishing, 2015, pp. 80–96.

[313] J. C. Blanchette and K. Claessen, "Generating counterexamples for structural inductions by exploiting nonstandard models," in *Logic for Programming, Artificial Intelligence, and Reasoning*, C. G. Fermüller and A. Voronkov, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 127–141.

[314] N. Elad, O. Padon, and S. Shoham, "An infinite needle in a finite haystack: Finding infinite counter-models in deductive verification," *Proc. ACM Program. Lang.*, vol. 8, no. POPL, Jan. 2024. [Online]. Available: https://doi.org/10.1145/3632875

[315] A. Lattuada, T. Hance, C. Cho, M. Brun, I. Subasinghe, Y. Zhou, J. Howell, B. Parno, and C. Hawblitzel, "Verus: Verifying rust programs using linear ghost types," *Proc. ACM Program. Lang.*, vol. 7, no. OOPSLA1, apr 2023. [Online]. Available: https://doi.org/10.1145/3586037

[316] Adithya Murali and Madhusudan Parthasarathy, "Automated Datastructure Verification using Unfoldings and SMT Solving: Foundations and FO-Completeness (POPL 2024 - TutorialFest) - POPL 2024," publisher: POPL 2024. [Online]. Available: https://popl24.sigplan.org/details/POPL-2024-tutorialfest/10/-Automated-Datastructure-Verification-using-Unfoldings-and-SMT-Solving-Foundations-a

[317] A. Q. Jiang, S. Welleck, J. P. Zhou, T. Lacroix, J. Liu, W. Li, M. Jamnik, G. Lample, and Y. Wu, "Draft, sketch, and prove: Guiding formal theorem provers with informal proofs," in *The Eleventh International Conference on Learning Representations*, 2023. [Online]. Available: https://openreview.net/forum?id=SMa9EAovKMC

[318] K. Yang, A. M. Swope, A. Gu, R. Chalamala, P. Song, S. Yu, S. Godil, R. Prenger, and A. Anandkumar, "Leandojo: theorem proving with retrieval-augmented language models," in *Proceedings of the 37th International Conference on Neural Information Processing Systems*, ser. NIPS '23. Red Hook, NY, USA: Curran Associates Inc., 2024.

[319] H. Wu, C. Barrett, and N. Narodytska, "Lemur: Integrating large language models in automated program verification," in *The Twelfth International Conference on Learning Representations*, 2024. [Online]. Available: https://openreview.net/forum?id=Q3YaCghZNt

[320] A. Kamath, A. Senthilnathan, S. Chakraborty, P. Deligiannis, S. Lahiri, A. Lal, A. Rastogi, S. Roy, and R. Sharma, "Leveraging llms for program verification," in *Formal Methods in Computer-Aided Design (FMCAD)*, October 2024. [Online]. Available: https://www.microsoft.com/en-us/research/publication/finding-inductive-loop-invariants-using-large-language-models/

[321] E. Lohn and S. Welleck, "minicodeprops: a minimal benchmark for proving code properties," 2024. [Online]. Available: https://arxiv.org/abs/2406.11915

[322] S. Klabnik and C. Nichols, *The Rust Programming Language*. USA: No Starch Press, 2018.

[323] N. D. Matsakis and F. S. Klock, "The rust language," *Ada Lett.*, vol. 34, no. 3, p. 103–104, Oct. 2014. [Online]. Available: https://doi.org/10.1145/2692956.2663188

[324] C. Enea and E. Koskinen, "Scenario-based proofs for concurrent objects," *Proc. ACM Program. Lang.*, vol. 8, no. OOPSLA1, Apr. 2024. [Online]. Available: https://doi.org/10.1145/3649857

[325] C. Cho, Y. Zhou, J. Bosamiya, and B. Parno, "A framework for debugging automated program verification proofs via proof actions," in *Computer Aided Verification*, A. Gurfinkel and V. Ganesh, Eds. Cham: Springer Nature Switzerland, 2024, pp. 348–361.

[326] P. Krogmeier and P. Madhusudan, "Languages with decidable learning: A meta-theorem," *Proc. ACM Program. Lang.*, vol. 7, no. OOPSLA1, Apr. 2023. [Online]. Available: https://doi.org/10.1145/3586032

[327] S. Heule, E. Schkufza, R. Sharma, and A. Aiken, "Stratified synthesis: Automatically learning the x86-64 instruction set," in *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '16. New York, NY, USA: Association for Computing Machinery, 2016. [Online]. Available: https://doi.org/10.1145/2908080.2908121 p. 237–250.

[328] S. Dinesh, G. Garrett-Grossman, and C. W. Fletcher, "Synthct: Towards portable constant-time code," in *29th Annual Network and Distributed System Security Symposium (NDSS '22)*. San Diego, CA, USA: The Internet Society, 2022, pp. 1–18.

[329] Y. Wang, I. Dillig, S. K. Lahiri, and W. R. Cook, "Verifying equivalence of database-driven applications," *Proc. ACM Program. Lang.*, vol. 2, no. POPL, dec 2018. [Online]. Available: https://doi.org/10.1145/3158144

[330] C. Smith, G. Ferns, and A. Albarghouthi, "Discovering relational specifications," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2017. New York, NY, USA: Association for Computing Machinery, 2017. [Online]. Available: https://doi.org/10.1145/3106237.3106279 p. 616–626.

[331] A. Fariha, A. Tiwari, A. Radhakrishna, S. Gulwani, and A. Meliou, "Conformance constraint discovery: Measuring trust in data-driven systems," in *Proceedings of the 2021 International Conference on Management of Data*, ser. SIGMOD '21. New York, NY, USA: Association for Computing Machinery, 2021. [Online]. Available: https://doi.org/10.1145/3448016.3452795 p. 499–512.

[332] A. Fariha, A. Tiwari, A. Meliou, A. Radhakrishna, and S. Gulwani, "Coco: Interactive exploration of conformance constraints for data understanding and data cleaning," in *Proceedings of the 2021 International Conference on Management of Data*, ser. SIGMOD '21. New York, NY, USA: Association for Computing Machinery, 2021. [Online]. Available: https://doi.org/10.1145/3448016.3452750 p. 2706–2710.

[333] A. Murali, A. Sehgal, P. Krogmeier, and P. Madhusudan, "Composing neural learning and symbolic reasoning with an application to visual discrimination," in *Proceedings of the Thirty-First International Joint Conference on Artificial Intelligence, IJCAI-22*, L. D. Raedt, Ed. International Joint Conferences on Artificial Intelligence Organization, 7 2022, main Track. [Online]. Available: https://doi.org/10.24963/ijcai.2022/466 pp. 3358–3365.

[334] L. Dunlap, Y. Zhang, X. Wang, R. Zhong, T. Darrell, J. Steinhardt, J. E. Gonzalez, and S. Yeung-Levy, "Describing differences in image sets with natural language," in *Conference on Computer Vision and Pattern Recognition (CVPR)*, 2024.