Model-Guided Synthesis of Inductive Lemmas for FOL with Least Fixpoints

ADITHYA MURALI, University of Illinois, Urbana-Champaign, USA
LUCAS PEÑA, University of Illinois, Urbana-Champaign, USA
EION BLANCHARD, University of Illinois, Urbana-Champaign, USA
CHRISTOF LÖDING, RWTH Aachen, Germany
P. MADHUSUDAN, University of Illinois, Urbana-Champaign, USA

Recursively defined linked data structures embedded in a pointer-based heap and their properties are naturally expressed in pure first-order logic with least fixpoint definitions (FO+lfp) with background theories. Such logics, unlike pure first-order logic, do not admit even complete procedures. In this paper, we undertake a novel approach for synthesizing inductive hypotheses 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.

CCS Concepts: • **Software and its engineering** → **Formal software verification**; • **Theory of computation** → **Automated reasoning**; **Logic and verification**; • **Computing methodologies** → *Machine learning*.

Additional Key Words and Phrases: Inductive Hypothesis Synthesis, Learning Logics, Counterexample-Guided Inductive Synthesis, First Order Logic with Least Fixpoints, Verifying Linked Data Structures

ACM Reference Format:
Adithya Murali, Lucas Peña, Eion Blanchard, Christof Löding, and P. Madhusudan. 2022. Model-Guided Synthesis of Inductive Lemmas for FOL with Least Fixpoints. *Proc. ACM Program. Lang.* 6, OOPSLA2, Article 191 (October 2022), 35 pages. https://doi.org/10.1145/3563354

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 [Barrett et al. 2011; Bradley and Manna 2007].

Authors' addresses: Adithya Murali, Department of Computer Science, University of Illinois, Urbana-Champaign, USA, adithya5@illinois.edu; Lucas Peña, Department of Computer Science, University of Illinois, Urbana-Champaign, USA, lpena7@illinois.edu; Eion Blanchard, Department of Mathematics, University of Illinois, Urbana-Champaign, USA, eionmb2@illinois.edu; Christof Löding, Department of Computer Science, RWTH Aachen, Germany, loeding@automata.rwth-aachen.de; P. Madhusudan, Department of Computer Science, University of Illinois, Urbana-Champaign, USA, madhu@illinois.edu.

2475-1421/2022/10-ART191
https://doi.org/10.1145/3563354

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 first-order logic (FOL) that can define data 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 [Calcagno et al. 2005; Löding et al. 2018; Madhusudan et al. 2012; Murali et al. 2020; Pek et al. 2014; Qiu et al. 2013; Suter et al. 2010]. The focus of this paper is 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 this paper, we use a particular technique called *natural proofs* that are based on systematic quantifier instantiation [Löding et al. 2018] 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[1] (i.e., sound proof systems for FO+*lfp* cannot admit proofs for every theorem). Indeed, on a number line, true addition and true multiplication over the natural numbers are definable using *lfp*. Hence by Gödel's incompleteness theorem [Enderton 2001], even quantifier-free logic with recursive definitions 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, which each identify a fairly strong property (the induction hypothesis) and its proof (the *induction step*).

In this paper, 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*.

The main proposal of this paper is 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:

[1] Quick proof: define a "number line" (discrete linear order) using a constant 0 and a unary function s (representing successor) with FO axioms expressing that the successor of no element is 0 and that the successor of no two different elements can be the same; second, encode the reachable configurations of a 2-counter machine (which is Turing powerful) as a relation defined using a least fixpoint, and express non-halting of the machine using this relation. This proof in fact shows that even a single recursive definition leads to validity being not recursively enumerable.

- 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 work under the assumption that each inductive 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 marks the only appeal to the least fixpoint semantics of recursive definitions 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.

We emphasize that proving inductive lemmas followed by pure FO reasoning to prove a theorem is itself not new. For example, the induction axiom schema in Peano arithmetic is:

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

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. This technique is prevalent, for example, in program verification. In this setting, 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 [Garg et al. 2014], 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 paper 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 [Ball and Rajamani 2002; Namjoshi and Kurshan 2000] or in program synthesis through the CEGIS paradigm [Alur et al. 2015; Solar Lezama 2008; Solar-Lezama et al. 2007]). The salient feature of our technique is the use of *finite first-order models* that act 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 [Löding et al. 2018; Pek et al. 2014; Qiu et al. 2013]. 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) cover all 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, simple 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 $SQI(k)$. By utilizing these three kinds of counterexample models, we narrow and guide the search space for lemma synthesis.

The main contribution of this paper 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 these three kinds of 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 synthesis engine 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 paper 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 [Löding et al. 2018] 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 the background sorts have quantifier-free fragments that are decidable using SMT solvers. Our tool framework 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 proposed in this paper are effective in synthesizing inductive lemmas and proving theorems. Apart from evaluating the efficiency of our tool, we evaluate the importance of 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.

Lemma synthesis has been studied for related logics, in particular for logics over algebraic datatypes (ADTs) [Reynolds and Kuncak 2015; Yang et al. 2019] and separation logic [Sighireanu et al. 2019; Ta et al. 2017]. Though these logics are very different in expressive power and comparisons across tools are hard, we provide a comparison of our tool against tools for these logics on our benchmark theorems using appropriate encodings whenever feasible.

Contributions. The main contributions of this paper 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 in the domain of heap data structures[2].

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.

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* 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_0. 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 [Enderton 2001].

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

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 2.1 (Counterexample models as conjunctive ground formulas). Consider the formula $(f(x) = y) \Rightarrow y > 3$ where $f : \sigma_0 \rightarrow Int$ 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 [Bradley and Manna 2007; de Moura and Bjørner 2008; Nelson 1980; Nelson and Oppen 1979] (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.

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

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

where $R \in \mathcal{R}^{rec}$, $\overline{x}$ are variables over the foregreound 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 [Grädel et al. 2007]). 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 [Tarski 1955])[3].

[3]Our definition of FO+*lfp* is similar to the one used in Finite Model Theory: see Libkin [Libkin 2004], 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 2.4.

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.

Example 2.2 (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

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

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 [Libkin 2004]. 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 2.3 (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):

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

$$htree(x) :=_{lfp} ite(x = nil,\ \emptyset, Singleton(x) \cup htree(l(x)) \cup htree(r(x)))$$

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 [Reynolds 2002].

We now state the usual notion of validity/entailment in FO+lfp in the language introduced above.

Definition 2.4 (FO+lfp Entailment). For a sentence α and a set Γ of formulas we write $\Gamma \cup \mathcal{D} \models_{\text{LFP}} \alpha$ 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 $\Phi \cup \mathcal{D}^{fp} \models_{\text{FO}} \alpha$ 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 2.3 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 $f : Int \rightarrow V$ 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 [Löding et al. 2018] for FOL validity for this restricted fragment in implementing the FOSSIL framework (see Section 2.4).

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

In this work 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 2.5 (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_*:

$$lseg(x, y_1) \Rightarrow \left(\mathsf{ite}(y_1 = nil, y_2 = y_1, y_2 = n(y_1)) \Rightarrow lseg(x, y_2)\right)$$

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 paper. 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 3.4.

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 2.6. *[Löding et al. 2018] If $\mathcal{A} \cup \mathcal{D}^{fp} \models_{FO} PFP(L)$, then $\mathcal{A} \cup \mathcal{D} \models_{LFP} L$.*

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 2.7 (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 omit them and just say that L is inductive.

Example 2.8 (Running Example: Inductive Lemma). Consider in the setting of Example 2.5 the following lemma L_*:

$$\forall x, y_1, y_2.\ lseg(x, y_1) \Rightarrow \Big(lseg(y_1, y_2) \Rightarrow lseg(x, y_2)\Big) \tag{L_*}$$

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 work.

Definition 2.9 (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 2.10 (**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 2.9).

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 2.11 (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 2.12 (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 2.5). 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 2.8), we have that L_* proves α_* in the sense of Definition 2.9. We illustrate a run of our synthesis algorithm that proves α_* by synthesizing L_* in Section 3.4.

In Section 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.

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 the work [Löding et al. 2018]. The results in this section are derived from the work in [Löding et al. 2018] and are not contributions of this paper.

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.,

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

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 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 2.13 (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 [Löding et al. 2018]:

Theorem 2.14 (From [Löding et al. 2018]). *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 THE FOSSIL ALGORITHM FOR SEQUENTIAL LEMMA SYNTHESIS

In this section, we present the fundamental contribution of this paper: FOSSIL (First-Order Solver with Synthesis of Inductive Lemmas), our algorithm for solving the Sequential Lemma Synthesis problem formulated in Definition 2.10. Figure 1 shows the components of our framework which we describe in Section 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 3.2. We then present the FOSSIL algorithm in Section 3.3. Finally, we illustrate a run of FOSSIL on our running example in Section 3.4. (the algorithm is guaranteed to find a proof if there is a set of independent lemmas that prove the goal).

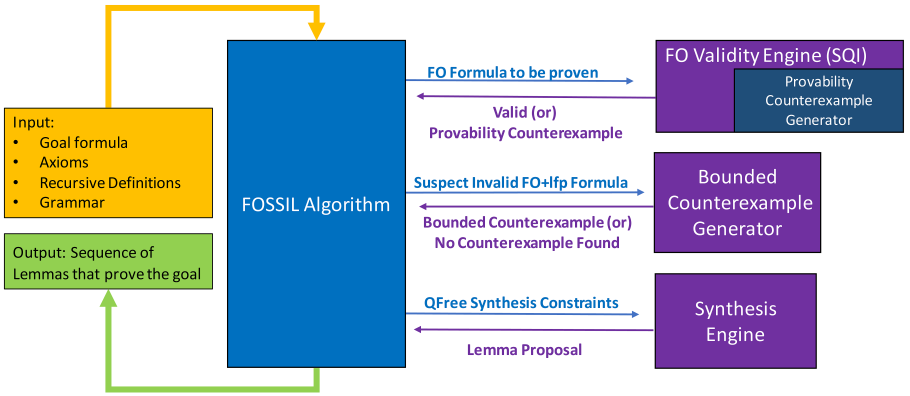


Fig. 1. Components of FOSSIL.

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 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 2.9. The high-level external components of FOSSIL are shown as purple boxes/arrows in Figure 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 3.2. 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 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** Counterexample(φ, 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 3.2.
(3) **Bounded Counterexample Generator** 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** 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.

We now turn to the definition of the three kinds of counterexamples used in FOSSIL.

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 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 8b and 8c of the FOSSIL algorithm (Figure 2). 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 8a.

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 8a in Figure 2). 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 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 $|\overline{c}| = |\overline{x}|$. 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 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 4.2 and Section 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 $|\overline{c}| = |\overline{x}|$ 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 The FOSSIL Algorithm

We now present the main contribution of this paper, the FOSSIL algorithm, which synthesizes lemmas in order to prove a theorem in FO+*lfp* .

Figure 2 shows the pseudocode of FOSSIL using the external components SQI, Counterexample, BoundedCex, and Synthesize described in Section 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 4 and an *inner loop* for discovering valid lemmas on line 7. At a general point in the execution on line 4, 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 6 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 7, 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: (8a) the lemma is false on the *Type–1* model, i.e., false on some tuple of elements from C (in line (8a), $L[C]$ denotes the set of all instantiations of L by elements from C, and $\bigwedge L[C]$ their conjunction); (8b) 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); and (8c) 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.

FOSSIL $(\mathcal{A}, \mathcal{D}, \mathcal{G}, \alpha; k, h)$
INPUT: axioms $\mathcal{A}$, recursive definitions $\mathcal{D}$, grammar $\mathcal{G}$, goal formula α, natural proofs depth parameter k, lemma production height parameter h
OUTPUT: Sequence of valid lemmas $\mathcal{L} \in L(\mathcal{G})$ (of height at most h) that prove α FO+*lfp* -valid using $SQI(k)$
IMPORTS: **SQI, Counterexample, BoundedCex, Synthesize**

(1) Compute $\mathcal{G}_h \subseteq \mathcal{G}$ such that $Lang(\mathcal{G}_h)$ does not contain any formulas whose parse-tree in $\mathcal{G}$ has a height greater than h.
(2) $\mathcal{L} := ()$, *Type–2* $:= \emptyset$, and *Type–3* $:= \emptyset$ for each $R \in \mathcal{D}$
(3) $\Phi_\alpha := \left(\bigwedge \mathcal{A} \cup \mathcal{D}^{fp}\right) \Rightarrow \alpha$
(4) WHILE (**SQI**$(\Phi_\alpha, k) \neq VALID$)
(5) $\quad gf_1(C) =$ **Counterexample**(Φ_α, k)
(6) $\quad$ *Type–1* $= gf_1(C)$
(7) $\quad$ WHILE (True)
(8) $\quad\quad L =$ **Synthesize**$(S, \mathcal{G}_h)$ such that $L(\overline{x}) = \forall \overline{x}.R(\overline{x}) \rightarrow \psi(\overline{x})$
and constraints S are:
(a) $\models_{\text{FO}} gf_1(C) \Rightarrow \neg(\bigwedge L[C])$, where $gf_1(C)$ is the current *Type–1* model
(b) $\models_{\text{FO}} gf_2(\overline{c}) \Rightarrow L(\overline{c})$ for all $(gf_2(\overline{c}), R) \in$ *Type–2*
(c) $\models_{\text{FO}} gf_3(\overline{c}) \Rightarrow PFP(L)(\overline{c})$ for all $(gf_3(\overline{c}), R) \in$ *Type–3*
(9) $\quad\quad$ If no lemma found, call FOSSIL$(\mathcal{A}, \mathcal{D}, \mathcal{G}, \alpha; k{+}1, h{+}1)$
(10) $\quad\quad \Phi_L := \left(\bigwedge \mathcal{A} \cup \mathcal{D}^{fp} \cup \mathcal{L}\right) \Rightarrow PFP(L)$
(11) $\quad\quad$ IF (**SQI**$(\Phi_L, k) = VALID$) THEN // Valid Lemma
(12) $\quad\quad\quad \mathcal{L} := \mathcal{L} \circ (L)$ (sequence extension)
(13) $\quad\quad\quad \Phi_\alpha := \left(\bigwedge \mathcal{A} \cup \mathcal{D}^{fp} \cup \mathcal{L}\right) \Rightarrow \alpha$
(14) $\quad\quad\quad$ *Type–3* $:= \emptyset$
(15) $\quad\quad\quad$ CONTINUE LOOP ON LINE 4
(16) $\quad\quad$ ELSE // Unprovable Lemma
(17) $\quad\quad\quad gf_2(\overline{c}) =$ **BoundedCex**$(L,$ *size*$)$
(18) $\quad\quad\quad$ IF ($gf_2(\overline{c})$ found) // Invalid Lemma
(19) $\quad\quad\quad\quad$ *Type–2* $:=$ *Type–2* $\cup \{(gf_2(\overline{c}), R)\}$
(20) $\quad\quad\quad$ ELSE // Irrefutable and Unprovable Lemma
(21) $\quad\quad\quad\quad gf_3(\overline{c}) =$ **Counterexample**(Φ_L, k)
(22) $\quad\quad\quad\quad$ *Type–3* $:=$ *Type–3* $\cup \{(gf_3(\overline{c}), R)\}$
(23) $\quad\quad\quad$ CONTINUE LOOP ON LINE 7

Fig. 2. The FOSSIL algorithm.

If no such lemma is found, we halt and restart the FOSSIL algorithm with higher values for k and h. If a lemma L is found, we try to prove Φ_L valid on line 11 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, then we add L to our assumptions and the current sequence of lemmas, discard *Type–3* counterexamples, stop the inner loop, and finally retry the proof of the theorem on line 4. We discard *Type–3* counterexamples since previously non-provable lemmas may now be provable.

If Φ_L is unprovable, we try to obtain a *size*-bounded *Type–2* counterexample $gf_2(\overline{c})$ on line 17 such that L does not hold on the tuple $\overline{c}$ of the foreground universe. If we cannot obtain a *Type–2* counterexample, then we obtain a *Type–3* counterexample $gf_3(\overline{c})$ such that $PFP(L)$ does not hold at $\overline{c}$ in the model gf_3. We add these counterexamples to their respective sets and continue searching for valid lemmas on line 7.

3.4 Running Example: List Segments

In this section, we present a full execution of our algorithm on the running example introduced in Example 2.5. Let us recall the Verification Condition (VC) α_* introduced earlier:

$$lseg(x, y_1) \Rightarrow \left(\text{ite}(y_1 = nil, y_2 = y_1, y_2 = n(y_1)) \Rightarrow lseg(x, y_2)\right)$$

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., $\mathcal{D}_*^{fp} \not\models_{\mathsf{FO}} PFP(\alpha_*)$ 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 6 in Figure 2):

$$u_1 \mapsto u_2 \mapsto u_3 \mapsto u_3 \quad u_4 \mapsto u_3,\ u_5 \mapsto u_5 \qquad \textit{and} \qquad x = u_1,\ y_1 = u_4,\ y_2 = u_3,\ nil = u_5 \quad lseg(u_1, u_3) = \textit{false}, \text{ and } lseg \text{ is } \textit{true} \text{ otherwise}$$

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 $lseg(u_1, u_4) = \textit{true}$ 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 $k = 1$[4].

Type–2 *Counterexample*. We now search for a lemma using the Synthesize module (line 8), 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 11). We now give it to the BoundedCex module (line 17) which returns the *Type–2* counterexample $\mathcal{M}_2$:

$$v_1 \mapsto v_2 \mapsto v_2 \qquad \textit{and} \qquad x = v_1,\ y = v_2,\ nil = v_2 \quad lseg(v_2, v_1) = \textit{false}, \text{ and } lseg \text{ is } \textit{true} \text{ otherwise}$$

$\mathcal{M}_2$ is a model of a one-element linked list where the interpretation of *lseg* is consistent with the *lfp* semantics. We add $\mathcal{M}_2$ to the set of *Type–2* models (line 19) 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 11 leads to the generation of a *Type–3* counterexample[5] (line 21) $\mathcal{M}_3$ 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 22) 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 2.8 in Section 2.3). We know from Examples 2.8 and 2.12 that L_* is inductive and proves α_*, and in fact it is provable with SQI at $k = 1$. Therefore, the checks on line 11 and subsequently on line 4 both succeed, whereupon FOSSIL terminates and reports that α_* is valid along with the lemma L_* used to prove it.

[4]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.
[5]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.

4 SYNTHESIS AND COUNTEREXAMPLE GENERATION ENGINES

In this section, we provide details of the individual modules from Figure 1. We refer the reader to Section 2.4 for the SQI module and only describe the synthesis and counterexample generation modules below.

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 [Alur et al. 2015, 2018] 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 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 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 [de Moura and Bjørner 2008; Nelson 1980]. 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.

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

We implement the above technique in a custom synthesis engine (see Section 6.1) and evaluate its efficacy in Section 6.

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.

Among these, the *Type−1* and *Type−3* counterexamples are generated using the Counterexample module as shown on lines 6 and 21 in Figure 2. 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 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 2.3) 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.

5 SOUNDNESS AND RELATIVE COMPLETENESS

The soundness of FOSSIL is clear from the problem description and the termination conditions in Figure 2: the branch on line 11 is only taken when a lemma is proved valid, and the loop condition on line 4 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 2.11). 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 5.1 (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 2.11, 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 Gist. Assume that there exists some set of independent lemmas $\{L_1, L_2, \ldots, L_n\}$ that proves α. We establish that 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 9 or runs forever without choosing one of the lemmas L_i.

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 that in each round the candidate proposal L will either: (i) be prevented from being proposed again in the inner loop (line 7) 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. Therefore we can eliminate the possibility that the algorithm will run forever without choosing a lemma from $\mathcal{L}$.

This leaves us with the possibility that the algorithm reaches line 9 without finding a new candidate lemma. In particular, this means that none of the L_i satisfies the constraints in line 8. It is easy to see that each $L_i, 1 \leq i \leq n$ satisfies constraints 8b and 8c 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. This leaves us with constraint 8a. 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 $M \models (\mathcal{A} \cup \mathcal{D} \cup \{\neg\alpha\} \cup \{L_i | 1 \leq i \leq n\})[T_k]$, which contradicts our initial assumption that $\{L_1, \ldots, L_n\}$ collectively prove α at depth k, i.e., $(\mathcal{A} \cup \mathcal{D} \cup \{\neg\alpha\} \cup \{L_i | 1 \leq i \leq n\})[T_k]$ is unsatisfiable. Therefore some L_i satisfies the constraint on line 8a and will eventually be proposed. Finally, we use induction on the number of lemmas n to reduce the given problem to a smaller one. See Appendix A.1 for a detailed proof. □

There are several possibilities for extending FOSSIL to achieve completeness for sequential lemma synthesis. One particular extension is an algorithm called FOSSIL-IP. The key idea is that when a lemma is neither provable nor refutable we add the *induction principle* $PFP(L) \Rightarrow L$ as an assumption (instead of adding *Type−3* counterexamples). The induction principle can be added because it is always valid. We discuss the possible extensions, describe FOSSIL-IP, and prove its

relative completeness for sequential lemma synthesis in Appendix A.2. We do not pursue these extensions in our work any further as they are significantly more expensive than FOSSIL.

6 IMPLEMENTATION AND EVALUATION

In this section, we describe our implementation and evaluation of FOSSIL (see Section 3). We also compare with lemma synthesis tools over ADTs and Separation Logic.

6.1 Implementation

We implement FOSSIL in Python, building the components given in Figure 1 using Z3Py (an API for the SMT solver Z3 [de Moura and Bjørner 2008]) 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 2.4). As far as we know, this is the first implementation of systematic quantifier instantiation [Löding et al. 2018; Pek et al. 2014; Qiu et al. 2013] 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 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 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 (Figure 2), utilizing the components above.

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?

6.3 Benchmarks

We curate two classes of benchmarks. The first suite consists of 50 theorems that were distilled from the work on VCDryad [Pek et al. 2014] repository[6] 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 [Reynolds 2002]. The goal was to retain 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

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

lfp to determine *reachable* states, demanding that unsafe states are not reached. Table 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 mentioned 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 benchmark 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 and disequalities between foreground sort terms, and the fixed predicate P from the benchmark. Finally, we allow implication and conjunction as Boolean operators.

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 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 9 in Figure 2). 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 30 s (with a maximum of 167 s). 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 Appendix A.3 for valid lemmas discovered in proving each theorem. For example, for `bst-left-right`, the tool proposes 27 lemmas of which 6 were proved

Table 1. Experiment results of the FOSSIL tool. The Syn column is the number of lemmas synthesized; the Val column is the number of valid lemmas synthesized; the Time column is the runtime in seconds.

Theorem	Syn	Val	Time (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	Syn	Val	Time (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

valid, including complex lemmas such as

$$bst(x) \Rightarrow (y \in hbst(x) \Rightarrow minr(x) \leq minr(y))$$

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.

Benchmark Suite #2. Figure 3 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.

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 (Figure 2) in two ways. First, the Synthesize module can *skip* solutions, proceeding to

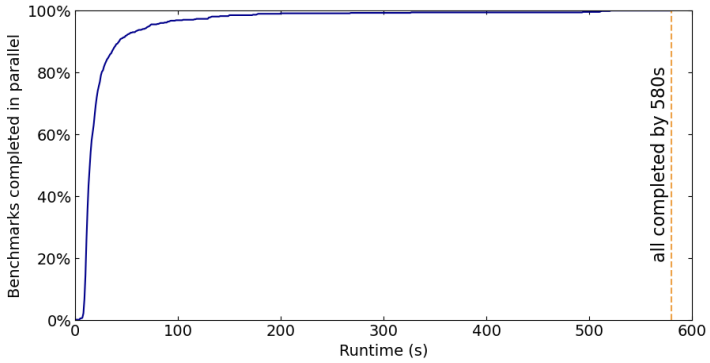


Fig. 3. Cumulative sum graph of FOSSIL on the synthetic benchmark suite of 673 theorems.

others. Second, when a lemma is not provable (line 16 in Figure 2) 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 4) 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 [Reynolds et al. 2019]. We used a timeout of 1 hour for the ablated algorithm.

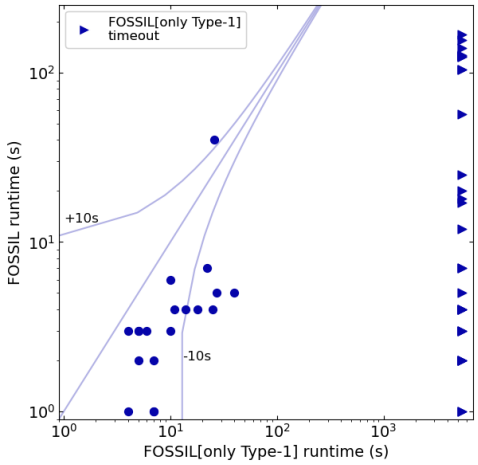
Figure 4a compares the ablated tool against our tool (with all types of counterexamples) on Suite #1 benchmarks. Apart from a few outliers where the lemmas proposed are very simple, FOSSIL with only *Type–1* counterexamples performs drastically worse than FOSSIL with all three counterexamples. 31 of the 50 benchmarks did not terminate with the ablated tool before the timeout. 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.

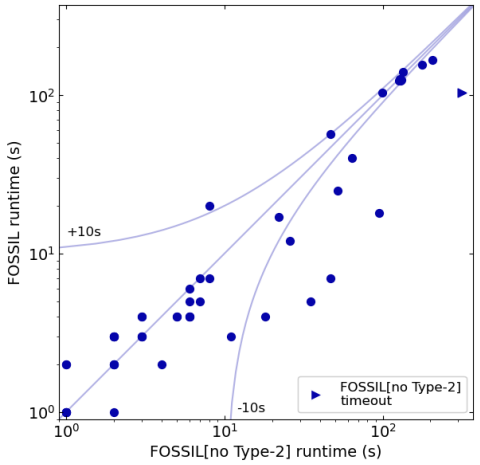
Efficacy of Type–2 counterexamples. We evaluate the efficacy of *Type–2* countermodels in FOSSIL by building a version of FOSSIL that does not use *Type–2* counterexamples.

The ablated algorithm is similar to the one in Figure 2 except for the case where a lemma is not provable (line 16). If a lemma cannot be proven valid, we do not try to generate a *Type–2* counterexample (lines 17- 19) and skip directly to generating a *Type–3* counterexample (line 21). A *Type–3* counterexample can always be generated since it witnesses the non-provability of a lemma (see Section 3.2). It also ensures that such unprovable lemmas will not be re-proposed.

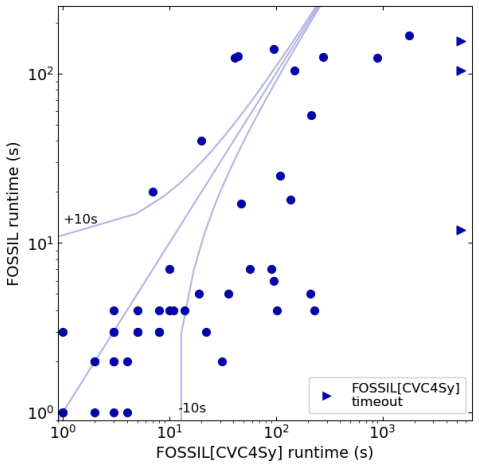
Figure 4b shows the running time comparison between the FOSSIL tool and the FOSSIL tool without *Type–2* counterexamples. The ablated tool does not solve one of the benchmarks and is slower in general for many benchmarks, especially those that require more than 10 seconds to solve. *Type–2* countermodels seem to have a higher impact in pruning the search space for more complex theorems. Figure 5 shows a comparison in the number of proposed lemmas for FOSSIL vs.



(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.

Fig. 4. Ablation studies of the FOSSIL tool. The timeout is 1 hour. In 4a, 4b, and 4c, 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.

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.

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 4c shows the results of this evaluation

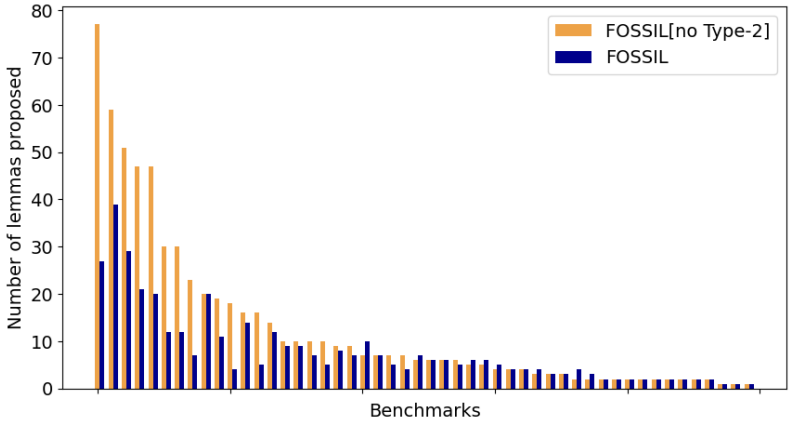


Fig. 5. Comparison of lemma proposal counts by FOSSIL vs. FOSSIL without *Type–2* counterexamples.

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.

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 [Reynolds and Kuncak 2015] and ADTInd [Yang et al. 2019], 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) [Sighireanu et al. 2019] (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) [Ta et al. 2017] 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).

7 RELATED WORK

Quantifier Instantiation. Quantifier instantiation is a common tool for reasoning using SMT solvers [Reynolds 2017]. E-matching is an instantiation technique used in the Simplify theorem prover [Detlefs et al. 2005], which chooses instantiations based on matching pattern terms. Similar methods are implemented in other SMT solvers [Barrett et al. 2011; de Moura and Bjørner 2008; Rümmer 2012], as well as methods for combining term instantiation with background SMT solvers [Ge and de Moura 2009]. The work in [Feldman et al. 2017] considers bounded quantifier instantiation in a pure FO setting (EPR) without background theories.

Natural Proofs. Our work directly builds off work related to natural proofs [Löding et al. 2018; Madhusudan et al. 2012; Pek et al. 2014; Qiu et al. 2013; Suter et al. 2010]. VCs similar to the theorems in our experiments are present in [Qiu et al. 2013], though lemmas needed to be *user-provided*. Work in [Löding et al. 2018] provided foundations for the work on natural proofs that preceded it. Completeness results in [Löding et al. 2018] directly contribute to our completeness results in this paper, and the techniques outlined in [Löding et al. 2018] are directly implemented in our tool.

Reasoning with Recursive Definitions. There is vast literature on reasoning with recursive definitions. The NQTHM prover developed by Boyer and Moore [Boyer and Moore 1988] and its successor ACL2 [Kaufmann and Moore 1997; Kaufmann et al. 2000] had support for recursive functions and had several induction heuristics to find inductive proofs. Recent works on cyclic proofs [Brotherston et al. 2011; Ta et al. 2016] also use heuristics for reasoning about recursive definitions. Additionally, an ongoing area of research involves decidable logics for recursive data structures [Le et al. 2017]. Naturally, the expressive power of these logics is restricted in order to

obtain a decidable validity problem. Further techniques in Dafny [Leino 2012] and Verifast [Jacobs et al. 2011] allow for verification via unfolding (or folding) recursive definitions, potentially based on user suggestions. These are instances of "unfold-and-match" [Madhusudan et al. 2012; Nguyen and Chin 2008; Pek et al. 2014; Qiu et al. 2013; Suter et al. 2010], a common heuristic for reasoning with recursive definitions that involves unfolding a recursive definition a few times and finding a proof of validity with the unfolded formulas, treating recursive definitions as uninterpreted.

Lemma Synthesis. The work in [Chu et al. 2015] 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 [Zhang et al. 2021], which also uses SyGuS for lemma generation, but operates over the simpler domain of bitvector problems. SLS (Songbird+Lemma Synthesis) [Ta et al. 2016] 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. SLS only supports a constrained version of SL, disallowing, for example, conjunctions of heap formulas. As a result, many FO+*lfp* formulas are inexpressible. Refer to Section 6.7 for a detailed comparison between FOSSIL and SLS on our set of benchmarks.

Formula Synthesis. The problem of synthesizing or learning first-order formulas from first-order models has seen recent development. In this space, our synthesis technique is specialized; we only learn universally quantified prenex formulas and hence use a mechanism that synthesizes the quantifier-free matrix using constraints that implicitly assume universal quantification. The work in [Krogmeier and Madhusudan 2022] tackles the harder 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. The work in [Koenig et al. 2020] develops a synthesis technique that reduces to SAT, but does not handle grammars; therefore, whether the technique can be used in our work is unclear. Further, whether either of these two works can be extended to effectively synthesize formulas involving background theories like integers and sets, which we require in FOSSIL, is also unclear.

ICE Learning. Our counterexamples and framework bear resemblance to the work on ICE Learning [Garg et al. 2014] for invariant synthesis, as invariants in imperative program verification are similar to inductive hypotheses. FOSSIL cannot handle invariant synthesis problems, however, despite the fact that inductive invariants are similar to inductive lemmas when programs are written as FO+*lfp* formulae. This is because we do not synthesize lemmas that quantify over background sorts such as integers (this distinction also applies to other methods catering to loop invariant synthesis [Neider et al. 2018], and as such we do not compare with such works). Our synthesis algorithm that uses essentially Boolean constraint solving exploits the fact that expressions synthesized do not have constants over the background sort. Second, while negative counterexamples in ICE correspond roughly to *Type–1* models (though the latter are *non-provability* counterexamples, similar to [Neider et al. 2018]), the positive and implication counterexamples in ICE do not seem to have a strict counterpart in our framework. However, *Type–3* counterexamples come close to implication counterexamples. In the ICE setting programs change configurations, leading to implication counterexamples. 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. However, *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, which makes them similar in spirit to implication counterexamples in ICE.

ADTs and Term Algebras. Turning to related work in proving properties of *term algebras* and *algebraic datatypes* (ADTs), the work in [Kovács et al. 2017] focuses on automating logics over arbitrary term algebras using FO approximations. For lemma synthesis, the work in [Yang et al. 2019] is another effort to synthesize inductive lemmas and also uses SyGuS (but without counterexample-guidance). The work in [Reynolds and Kuncak 2015] also aims to synthesize inductive lemmas, and we provide a detailed comparison with this work in Section 6.7. The work in [Govind V K et al. 2022] infers lemmas for synthesizing invariants but does not use counterexamples.

We emphasize again, however, that work on ADTs/term algebras and our work here on FO+*lfp* are very different and hard to compare both theoretically and experimentally. First, a term algebra universe (ADTs) (without background universe) is a *single* universe/model (with fixed interpretation of functions such as constructors/destructors) that is negation-complete. Our universes model heaps and admit a *multitude* of universes. Second, the universe of a term algebra has a complete recursive axiomatization [Hodges 1997; Mal'tsev 1962], and hence FO properties of ADTs are in fact decidable, while FO+*lfp* does not even admit complete procedures, let alone decidable ones. Third, several data structures we work with do not even have analogous structures in the ADT world— e.g., list segments between two locations, doubly-linked lists, cyclic lists. And destructive pointer updates on them are not expressible in the world of ADTs. Also defining data structures common in ADTs in the heap world are considerably more difficult, as we need to express separation (for example, even the definition of a tree requires such separation constraints; see Section 2). Consequently, a fair experimental comparison of our tools against those developed for ADTs [Boyer and Moore 1988; Claessen et al. 2013; Cruanes 2017; Govind V K et al. 2022; Hajdú et al. 2020; Johansson 2019; Kaufmann and Moore 1997; Passmore et al. 2020; Sonnex et al. 2012] is difficult. Still, when the logic admits uninterpreted or higher-order functions, an encoding is possible (Section 6.7). Extending our techniques to build a lemma synthesis technique/tool with built-in support for ADTs, especially to reason with functional programs, is an interesting future direction.

8 CONCLUSIONS

The primary contribution of this paper is an inductive lemma synthesis technique for FO+*lfp* with background theories that learns from semantically-rich counterexample FO models that witness non-provabilitySuch a search for inductive lemmas based on the semantics of theorems/lemmas can be useful in other contexts— e.g, in identifying lemmas from a large corpus to help prove theorems. For instance, the work in [Bansal et al. 2019] uses machine learning to find proofs, but currently little semantic information is used in learning. Extending our work to synthesizing lemmas for other logics, especially over ADTs (see Section 7) as well as Separation Logic is also interesting. We also believe that building general lemma synthesis engines that extend SMT solvers can be valuable for researchers who use automated theorem proving in a variety of application domains.

DATA AVAILABILITY STATEMENT

The code and data artifacts required to reproduce the experiments on the FOSSIL tool and various ablation studies are available via ACM DL at [Murali et al. 2022].

ACKNOWLEDGMENTS

This work is supported in part by a research grant from Amazon and a Discovery Partners Institute (DPI) science team seed grant.

REFERENCES

Rajeev Alur, Rastislav Bodík, Eric Dallal, Dana Fisman, Pranav Garg, Garvit Juniwal, Hadas Kress-Gazit, P. Madhusudan, Milo M. K. Martin, Mukund Raghothaman, Shambwaditya Saha, Sanjit A. Seshia, Rishabh Singh, Armando Solar-Lezama, Emina Torlak, and Abhishek Udupa. 2015. *Syntax-Guided Synthesis*. IOS Press, 1–25. https://doi.org/10.3233/978-1-61499-495-4-1

Rajeev Alur, Rishabh Singh, Dana Fisman, and Armando Solar-Lezama. 2018. Search-Based Program Synthesis. *Commun. ACM* 61, 12 (Nov. 2018), 84–93. https://doi.org/10.1145/3208071

Thomas Ball and Sriram K. Rajamani. 2002. The SLAM Project: Debugging System Software via Static Analysis. (2002), 1–3. https://doi.org/10.1145/503272.503274

Kshitij Bansal, Sarah M. Loos, Markus N. Rabe, Christian Szegedy, and Stewart Wilcox. 2019. HOList: An Environment for Machine Learning of Higher-Order Theorem Proving. https://doi.org/10.48550/ARXIV.1904.03241

Clark Barrett, Christopher L. Conway, Morgan Deters, Liana Hadarean, Dejan Jovanović, Tim King, Andrew Reynolds, and Cesare Tinelli. 2011. CVC4. In *Computer Aided Verification*, Ganesh Gopalakrishnan and Shaz Qadeer (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 171–177. https://doi.org/10.1007/978-3-642-22110-1_14

Robert S. Boyer and J. Strother Moore. 1988. *A Computational Logic Handbook*. Academic Press Professional, Inc., USA.

Aaron R. Bradley and Zohar Manna. 2007. *The Calculus of Computation: Decision Procedures with Applications to Verification*. Springer-Verlag, Berlin, Heidelberg. https://doi.org/10.1007/978-3-540-74113-8

James Brotherston, Dino Distefano, and Rasmus Lerchedahl Petersen. 2011. Automated Cyclic Entailment Proofs in Separation Logic. In *Automated Deduction – CADE-23*, Nikolaj Bjørner and Viorica Sofronie-Stokkermans (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 131–146. https://doi.org/10.1007/978-3-642-22438-6_12

Cristiano Calcagno, Philippa Gardner, and Matthew Hague. 2005. From Separation Logic to First-Order Logic. In *Foundations of Software Science and Computational Structures*, Vladimiro Sassone (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 395–409. https://doi.org/10.1007/978-3-540-31982-5_25

Duc-Hiep Chu, Joxan Jaffar, and Minh-Thai Trinh. 2015. Automatic Induction Proofs of Data-Structures in Imperative Programs. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Portland, OR, USA) *(PLDI '15)*. Association for Computing Machinery, New York, NY, USA, 457–466. https://doi.org/10.1145/2737924.2737984

Koen Claessen, Moa Johansson, Dan Rosén, and Nicholas Smallbone. 2013. 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 (Lecture Notes in Computer Science, Vol. 7898)*, Maria Paola Bonacina (Ed.). Springer, 392–406. https://doi.org/10.1007/978-3-642-38574-2_27

Simon Cruanes. 2017. Superposition with Structural Induction. In *Frontiers of Combining Systems*, Clare Dixon and Marcelo Finger (Eds.). Springer International Publishing, Cham, 172–188. https://doi.org/10.1007/978-3-319-66167-4_10

Leonardo de Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *Tools and Algorithms for the Construction and Analysis of Systems*, C. R. Ramakrishnan and Jakob Rehof (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 337–340. https://doi.org/10.1007/978-3-540-78800-3_24

David Detlefs, Greg Nelson, and James B. Saxe. 2005. Simplify: A Theorem Prover for Program Checking. *J. ACM* 52, 3 (May 2005), 365–473. https://doi.org/10.1145/1066100.1066102

H.B. Enderton. 2001. *A Mathematical Introduction to Logic*. Elsevier Science Publishers Ltd. https://doi.org/10.1016/C2009-0-22107-6

Yotam M. Y. Feldman, Oded Padon, Neil Immerman, Mooly Sagiv, and Sharon Shoham. 2017. Bounded Quantifier Instantiation for Checking Inductive Invariants. In *Tools and Algorithms for the Construction and Analysis of Systems*, Axel Legay and Tiziana Margaria (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 76–95. https://doi.org/10.1007/978-3-662-54577-5_5

Pranav Garg, Christof Löding, P. Madhusudan, and Daniel Neider. 2014. ICE: A Robust Framework for Learning Invariants. In *Computer Aided Verification*, Armin Biere and Roderick Bloem (Eds.). Springer International Publishing, Cham, 69–87. https://doi.org/10.1007/978-3-319-08867-9_5

Yeting Ge and Leonardo de Moura. 2009. Complete Instantiation for Quantified Formulas in Satisfiabiliby Modulo Theories. In *Computer Aided Verification*, Ahmed Bouajjani and Oded Maler (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 306–320. https://doi.org/10.1007/978-3-642-02658-4_25

Hari Govind V K, Sharon Shoham, and Arie Gurfinkel. 2022. Solving Constrained Horn Clauses modulo Algebraic Data Types and Recursive Functions. *Proc. ACM Program. Lang.* 6, POPL, Article 60 (Jan 2022), 29 pages. https://doi.org/10.1145/3498722

Erich Grädel, Phokion G. Kolaitis, Leonid Libkin, Maarten Marx, Joel Spencer, Moshe Y. Vardi, Yde Venema, and Scott Weinstein. 2007. *Finite Model Theory and Its Applications*. Springer. https://doi.org/10.1007/3-540-68804-8

Márton Hajdú, Petra Hozzová, Laura Kovács, Johannes Schoisswohl, and Andrei Voronkov. 2020. Induction with Generalization in Superposition Reasoning. In *Intelligent Computer Mathematics - 13th International Conference, CICM 2020, Bertinoro, Italy, July 26-31, 2020, Proceedings (Lecture Notes in Computer Science, Vol. 12236)*, Christoph Benzmüller and

Bruce R. Miller (Eds.). Springer, 123–137. https://doi.org/10.1007/978-3-030-53518-6_8

Wilfrid Hodges. 1997. *A Shorter Model Theory.* Cambridge University Press, USA.

Bart Jacobs, Jan Smans, Pieter Philippaerts, Frédéric Vogels, Willem Penninckx, and Frank Piessens. 2011. VeriFast: A Powerful, Sound, Predictable, Fast Verifier for C and Java. In *NASA Formal Methods*, Mihaela Bobaru, Klaus Havelund, Gerard J. Holzmann, and Rajeev Joshi (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 41–55. https://doi.org/10.1007/978-3-642-20398-5_4

Moa Johansson. 2019. Lemma Discovery for Induction - A Survey. In *Intelligent Computer Mathematics - 12th International Conference, CICM 2019, Prague, Czech Republic, July 8-12, 2019, Proceedings (Lecture Notes in Computer Science, Vol. 11617)*, Cezary Kaliszyk, Edwin C. Brady, Andrea Kohlhase, and Claudio Sacerdoti Coen (Eds.). Springer, 125–139. https://doi.org/10.1007/978-3-030-23250-4_9

Matt Kaufmann and J. S. Moore. 1997. An Industrial Strength Theorem Prover for a Logic Based on Common Lisp. *IEEE Trans. Softw. Eng.* 23, 4 (April 1997), 203–213. https://doi.org/10.1109/32.588534

Matt Kaufmann, J. Strother Moore, and Panagiotis Manolios. 2000. *Computer-Aided Reasoning: An Approach.* Springer New York, NY. https://doi.org/10.1007/978-1-4615-4449-4

Jason R. Koenig, Oded Padon, Neil Immerman, and Alex Aiken. 2020. First-Order Quantified Separators. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation* (London, UK) *(PLDI 2020)*. Association for Computing Machinery, New York, NY, USA, 703–717. https://doi.org/10.1145/3385412.3386018

Laura Kovács, Simon Robillard, and Andrei Voronkov. 2017. Coming to Terms with Quantified Reasoning. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages* (Paris, France) *(POPL '17)*. ACM, New York, NY, USA, 260–270. https://doi.org/10.1145/3009837.3009887

Paul Krogmeier and P. Madhusudan. 2022. Learning Formulas in Finite Variable Logics. *Proc. ACM Program. Lang.* 6, POPL, Article 10 (Jan 2022), 28 pages. https://doi.org/10.1145/3498671

Quang Loc Le, Makoto Tatsuta, Jun Sun, and Wei-Ngan Chin. 2017. A Decidable Fragment in Separation Logic with Inductive Predicates and Arithmetic. In *Computer Aided Verification*, Rupak Majumdar and Viktor Kunčak (Eds.). Springer International Publishing, Cham, 495–517. https://doi.org/10.1007/978-3-319-63390-9_26

K. Rustan M. Leino. 2012. Automating Induction with an SMT Solver. In *Proceedings of the 13th International Conference on Verification, Model Checking, and Abstract Interpretation* (Philadelphia, PA) *(VMCAI'12)*. Springer-Verlag, Berlin, Heidelberg, 315–331. https://doi.org/10.1007/978-3-642-27940-9_21

Leonid Libkin. 2004. *Elements of Finite Model Theory.* Springer Berlin, Heidelberg. https://doi.org/10.1007/978-3-662-07003-1

Christof Löding, P. Madhusudan, and Lucas Peña. 2018. Foundations for natural proofs and quantifier instantiation. *PACMPL* 2, POPL (2018), 10:1–10:30. https://doi.org/10.1145/3158098

P. Madhusudan, Xiaokang Qiu, and Andrei Ştefănescu. 2012. Recursive Proofs for Inductive Tree Data-structures. In *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Philadelphia, PA, USA) *(POPL '12)*. ACM, New York, NY, USA, 123–136. https://doi.org/10.1145/2103656.2103673

A. I. Mal'tsev. 1962. Axiomatizable classes of locally free algebras of certain types. *Sibirsk. Mat. Zh.* 3 (1962), 729–743. Issue 5. http://mi.mathnet.ru/eng/smj/v3/i5/p729

Adithya Murali, Lucas Peña, Eion Blanchard, Christof Löding, and P. Madhusudan. 2022. *Artifact for OOPSLA 2022 Article Model-Guided Synthesis of Inductive Lemmas for FOL with Least Fixpoints.* https://doi.org/10.1145/3554331

Adithya Murali, Lucas Peña, Christof Löding, and P. Madhusudan. 2020. A First-Order Logic with Frames. In *Programming Languages and Systems*, Peter Müller (Ed.). Springer International Publishing, Cham, 515–543. https://doi.org/10.1007/978-3-030-44914-8_19

Kedar S. Namjoshi and Robert P. Kurshan. 2000. Syntactic Program Transformations for Automatic Abstraction. In *Computer Aided Verification*, E. Allen Emerson and Aravinda Prasad Sistla (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 435–449. https://doi.org/10.1007/10722167_33

Daniel Neider, Pranav Garg, P. Madhusudan, Shambwaditya Saha, and Daejun Park. 2018. Invariant Synthesis for Incomplete Verification Engines. In *Tools and Algorithms for the Construction and Analysis of Systems*, Dirk Beyer and Marieke Huisman (Eds.). Springer International Publishing, Cham, 232–250. https://doi.org/10.1007/978-3-319-89960-2_13

Charles Gregory Nelson. 1980. *Techniques for Program Verification.* Ph. D. Dissertation. Stanford University, Stanford, CA, USA. AAI8011683.

Greg Nelson and Derek C. Oppen. 1979. Simplification by Cooperating Decision Procedures. *ACM Trans. Program. Lang. Syst.* 1, 2 (Oct 1979), 245–257. https://doi.org/10.1145/357073.357079

Huu Hai Nguyen and Wei-Ngan Chin. 2008. Enhancing Program Verification with Lemmas. In *Proceedings of the 20th International Conference on Computer Aided Verification* (Princeton, NJ, USA) *(CAV '08)*. Springer-Verlag, Berlin, Heidelberg, 355–369. https://doi.org/10.1007/978-3-540-70545-1_34

Grant Passmore, Simon Cruanes, Denis Ignatovich, Dave Aitken, Matt Bray, Elijah Kagan, Kostya Kanishev, Ewen Maclean, and Nicola Mometto. 2020. The Imandra Automated Reasoning System (System Description). In *Automated Reasoning*, Nicolas Peltier and Viorica Sofronie-Stokkermans (Eds.). Springer International Publishing, Cham, 464–471. https:

//doi.org/10.1007/978-3-030-51054-1_30

Edgar Pek, Xiaokang Qiu, and P. Madhusudan. 2014. 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* (Edinburgh, United Kingdom) *(PLDI '14)*. ACM, New York, NY, USA, 440–451. https://doi.org/10.1145/2594291.2594325

Xiaokang Qiu, Pranav Garg, Andrei Ştefănescu, and P. Madhusudan. 2013. Natural Proofs for Structure, Data, and Separation. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Seattle, Washington, USA) *(PLDI '13)*. ACM, New York, NY, USA, 231–242. https://doi.org/10.1145/2491956.2462169

Andrew Reynolds. 2017. Conflicts, Models and Heuristics for Quantifier Instantiation in SMT. In *Vampire 2016. Proceedings of the 3rd Vampire Workshop (EPiC Series in Computing, Vol. 44)*, Laura Kovacs and Andrei Voronkov (Eds.). EasyChair, 1–15. https://doi.org/10.29007/jmd3

Andrew Reynolds, Haniel Barbosa, Andres Nötzli, Clark Barrett, and Cesare Tinelli. 2019. cvc4sy: Smart and Fast Term Enumeration for Syntax-Guided Synthesis. In *Computer Aided Verification*, Isil Dillig and Serdar Tasiran (Eds.). Springer International Publishing, Cham, 74–83. https://doi.org/10.1007/978-3-030-25543-5_5

Andrew Reynolds and Viktor Kuncak. 2015. Induction for SMT Solvers. In *Verification, Model Checking, and Abstract Interpretation*, Deepak D'Souza, Akash Lal, and Kim Guldstrand Larsen (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 80–98. https://doi.org/10.1007/978-3-662-46081-8_5

John C. Reynolds. 2002. Separation Logic: A Logic for Shared Mutable Data Structures. In *Proceedings of the 17th Annual IEEE Symposium on Logic in Computer Science (LICS '02)*. IEEE Press, 55–74. https://doi.org/10.1109/LICS.2002.1029817

Philipp Rümmer. 2012. E-Matching with Free Variables. In *Logic for Programming, Artificial Intelligence, and Reasoning*, Nikolaj Bjørner and Andrei Voronkov (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 359–374. https://doi.org/10.1007/978-3-642-28717-6_28

Mihaela Sighireanu, Juan A. Navarro Pérez, Andrey Rybalchenko, Nikos Gorogiannis, Radu Iosif, Andrew Reynolds, Cristina Serban, Jens Katelaan, Christoph Matheja, Thomas Noll, Florian Zuleger, Wei-Ngan Chin, Quang Loc Le, Quang-Trung Ta, Ton-Chanh Le, Thanh-Toan Nguyen, Siau-Cheng Khoo, Michal Cyprian, Adam Rogalewicz, Tomas Vojnar, Constantin Enea, Ondrej Lengal, Chong Gao, and Zhilin Wu. 2019. SL-COMP: Competition of Solvers for Separation Logic. In *Tools and Algorithms for the Construction and Analysis of Systems*, Dirk Beyer, Marieke Huisman, Fabrice Kordon, and Bernhard Steffen (Eds.). Springer International Publishing, Cham, 116–132. https://doi.org/10.1007/978-3-030-17502-3_8

Armando Solar Lezama. 2008. *Program Synthesis By Sketching*. Ph. D. Dissertation. EECS Department, University of California, Berkeley. http://www2.eecs.berkeley.edu/Pubs/TechRpts/2008/EECS-2008-177.html

Armando Solar-Lezama, Gilad Arnold, Liviu Tancau, Rastislav Bodík, Vijay A. Saraswat, and Sanjit A. Seshia. 2007. Sketching stencils. In *Proceedings of the ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation, San Diego, California, USA, June 10-13, 2007*, Jeanne Ferrante and Kathryn S. McKinley (Eds.). ACM, 167–178. https://doi.org/10.1145/1250734.1250754

William Sonnex, Sophia Drossopoulou, and Susan Eisenbach. 2012. Zeno: An Automated Prover for Properties of Recursive Data Structures. In *Tools and Algorithms for the Construction and Analysis of Systems*, Cormac Flanagan and Barbara König (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 407–421. https://doi.org/10.1007/978-3-642-28756-5_28

Philippe Suter, Mirco Dotta, and Viktor Kunćak. 2010. Decision Procedures for Algebraic Data Types with Abstractions. In *Proceedings of the 37th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Madrid, Spain) *(POPL '10)*. ACM, New York, NY, USA, 199–210. https://doi.org/10.1145/1706299.1706325

Quang-Trung Ta, Ton Chanh Le, Siau-Cheng Khoo, and Wei-Ngan Chin. 2016. Automated Mutual Explicit Induction Proof in Separation Logic. In *FM 2016: Formal Methods*, John Fitzgerald, Constance Heitmeyer, Stefania Gnesi, and Anna Philippou (Eds.). Springer International Publishing, Cham, 659–676. https://doi.org/10.1007/978-3-319-48989-6_40

Quang-Trung Ta, Ton Chanh Le, Siau-Cheng Khoo, and Wei-Ngan Chin. 2017. Automated Lemma Synthesis in Symbolic-Heap Separation Logic. *Proc. ACM Program. Lang.* 2, POPL, Article 9 (Dec 2017), 29 pages. https://doi.org/10.1145/3158097

Alfred Tarski. 1955. A lattice-theoretical fixpoint theorem and its applications. *Pacific J. Math.* 5, 2 (1955), 285 – 309. https://projecteuclid.org/euclid.pjm/1103044538

Weikun Yang, Grigory Fedyukovich, and Aarti Gupta. 2019. Lemma Synthesis for Automating Induction over Algebraic Data Types. In *Principles and Practice of Constraint Programming*, Thomas Schiex and Simon de Givry (Eds.). Springer International Publishing, Cham, 600–617. https://doi.org/10.1007/978-3-030-30048-7_35

Hongce Zhang, Aarti Gupta, and Sharad Malik. 2021. 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 (Lecture Notes in Computer Science, Vol. 12597)*, Fritz Henglein, Sharon Shoham, and Yakir Vizel (Eds.). Springer, 325–349. https://doi.org/10.1007/978-3-030-67067-2_15

A APPENDIX

A.1 Further details for Section 3

Proof of Theorem 5.1.

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 9 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 12) before all the variables are reset, which reduces the problem to discovering at most $n-1$ independent lemmas whereupon we will appeal to the induction on number of lemmas to be discovered.

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 that in each round the candidate proposal L will either: (i) be prevented from being proposed again in the inner loop (line 7) 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 9.

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

This leaves us with the possibility that the algorithm reaches line 9 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 8b and 8c 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 8a. 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 $M \models (\mathcal{A} \cup \mathcal{D} \cup \{\neg\alpha\} \cup \{L_i | 1 \leq i \leq n\})[T_k]$, which contradicts our initial assumption that $\{L_1, \ldots, L_n\}$ collectively prove α at depth k, i.e., $(\mathcal{A} \cup \mathcal{D} \cup \{\neg\alpha\} \cup \{L_i | 1 \leq i \leq n\})[T_k]$ is unsatisfiable. Therefore some L_i satisfies the constraint on line 8a and will eventually be proposed, which concludes our proof. □

A.2 Lemma Synthesis Algorithms Relatively Complete wrt Sequential Lemmas

In this section we briefly discuss the problem of designing algorithms for sequential lemma synthesis that are also relatively complete wrt sequential lemmas (instead of just being relatively complete wrt independent lemmas as in Theorem 5.1). To do this we must first see why FOSSIL is not already

complete for sequential lemmas. They key obstacle is the *Type−1* model that makes the lemma synthesis goal-directed. Consider the following scenario:

Example A.1 (FOSSIL is not complete for sequence of lemmas). 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 the beginning of the algorithm on line 6 in Figure 2, L_2 would be false on *Type−1* since it helps prove α. But there is nothing that prevents $L_1[T^*]$ 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 22 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 guarantee that a proof of α will be found by FOSSIL.

We propose three different strategies to address the above issue:

(1) The simplest way to achieve the relative completeness is to utilize FOSSIL as described in Figure 2, but eliminate constraints corresponding to *Type−1* models. This eliminates the problem described in Example A.1 where we need to necessarily synthesize lemmas that help prove the goal, and instead reduces the algorithm to only generating lemma proposals and eliminating spurious proposals using *Type−2* and *Type−3* models. This approach has the obvious disadvantage of not being goal-directed and could lead to large execution time for proof if the sequence of lemmas needed consists of large lemmas (by size) and smaller lemmas could be easily eliminated if given the goal.
(2) A second approach is to have the algorithm branch into two-subroutines (both branches searched fairly, dovetailing between them) when given a lemma that is unprovable, one assuming that the lemma is valid and other assuming that it is not. We can then pursue each subroutine until we find a proof or reach a contradiction. However, this algorithm could quickly explode in the number of subroutines even with a few unprovable lemmas and likely impractical.
(3) We propose a third alternative that generalises FOSSIL. Looking at the Example A.1, it would be useful if we could update *Type−1* to include the failure to prove L_2 so that the lemma synthesis is guided towards L_1. What should be the constraint with which we update the model? The updated model could be such that L_2 holds (on the instantiated terms), or it could witness that L_2 is not inductive, i.e., cannot be proved by induction. However, these two possibilities are precisely those expressed by the induction principle for L_2. Recall the definition from Section 2.3: the induction principle of a lemma $L(\bar{x})$ is given by $(\forall \bar{x}.PFP(L(\bar{x}))) \rightarrow (\forall \bar{x}.L(\bar{x})) \equiv \neg(\forall \bar{x}.PFP(L(\bar{x}))) \vee (\forall \bar{x}.L(\bar{x}))$ where PFP represents the condition that L is inductive. Therefore the induction principle captures the two possibilities of L either being valid or not inductive. Our third alternative proposal is thus to use the induction principle to address the problem of completeness for sequences of lemmas in an algorithm we call FOSSIL-IP.

FOSSIL-IP. Let us discuss the third strategy in more detail. Simply put, we would like to add the induction principle for any lemmas that we cannot prove to our axioms and retain the rest of the algorithm. In particular, with respect to the algorithm description in Figure 2 we would maintain a set $\mathcal{IP}$ of induction principles starting out with an empty set and include it in the construction of Φ_α and Φ_L on lines 3 and 10. Then, given a proposal L that we can neither prove nor establish as being invalid using a *Type−2* model (line 20), we would eliminate falling back to a *Type−3* model on lines 21 and 22 and replace it with the update of $\mathcal{IP}$ with the induction principle of L. This

algorithm, which we call FOSSIL-IP, is relatively complete for the problem of sequential lemma synthesis:

Theorem A.2 (Relative completeness of FOSSIL-IP wrt sequential lemmas). *If α is provable from $\mathcal{A}$ and $\mathcal{D}$ by a finite sequence of inductive lemmas, then there is an instantiation depth k and grammar height h such that FOSSIL-IP terminates and returns a set $\mathcal{L}$ of lemmas and a set $\mathcal{IP}$ of induction principles proving α.*

We detail the formulation of proving a theorem using induction principles in Appendix A.2.1. Admittedly, our solution could still create large synthesis queries that could be difficult to handle and therefore has potential disadvantages as do the other two strategies.

A.2.1 Discussion about induction principles and description of FOSSIL-IP. As illustrated in Example A.1, we cannot guarantee that FOSSIL finds a sequence of inductive lemmas proving the goal if such a sequence exists. We need the stronger assumption that a set of independent lemmas exists for proving the goal.

The algorithm FOSSIL-IP is a modification of FOSSIL that is guaranteed to find a proof of the goal if it can be proven by a sequence of inductive lemmas. In addition to the sequence of valid lemmas that is constructed in a similar way as FOSSIL does, FOSSIL-IP additionally uses induction principles of lemmas for which it does not find an inductive proof. It might happen that these induction principles help proving α without the algorithm being able to prove the actual lemmas valid. We illustrate the difference between induction principles and lemmas proving α for an (artificial) example situation.

Example A.3. Consider the definition of *list* from above. Add two constants c_1, c_2 to the signature, and two recursive definitions $list_1$ and $list_2$:

$$\begin{aligned} list_1(x) &:=_{lfp} (x = nil) \vee ((list_1(n(x)) \wedge (c_1 = c_2 \rightarrow x \neq c_1)) \\ list_2(x) &:=_{lfp} (x = nil) \vee ((list_2(n(x)) \wedge (c_1 \neq c_2 \rightarrow x \neq c_1)) \end{aligned}$$

So both are defined as *list* with the only difference that the recursion stops at c_1 for $list_1$ if $c_1 = c_2$, and for $list_2$ if $c_1 \neq c_2$.

Take $\alpha = \forall x.list(x) \rightarrow (list_1(x) \vee list_2(x))$. This is certainly true in LFP semantics because if $c_1 = c_2$, then $list_2$ is the same as *list*, otherwise $list_1$ is the same as *list*. Consider the lemmas $L_1 = \forall x.list(x) \rightarrow list_1(x)$ and $L_2 = \forall x.list(x) \rightarrow list_2(x)$. For each lemma, there are clearly LFP models in which the lemma does not hold (if $c_1 = c_2$ and $list(c_1)$, then L_1 is false, similarly for L_2). However, we have that $\mathcal{A} \cup \mathcal{D} \cup \{IP(L_1), IP(L_2)\} \models_{\mathsf{FO}} \alpha$ because on each model either $PFP(L_1)$ or $PFP(L_2)$ is satisfied.

This illustrates that provability by induction principles does not yield provability by the corresponding lemmas. The other direction, however, is always true, as stated in the following lemma.

Lemma A.4. *If $(L_1, \ldots, L_n)$ is a sequence of inductive lemmas that prove α then the set $\mathcal{IP} = \{IP(L_1), \ldots, IP(L_n)\}$ proves α.*

Proof. If $\mathcal{IP}$ does not prove α, then there is a model M of $\mathcal{A} \cup \mathcal{D} \cup \mathcal{IP} \cup \{\neg\alpha\}$ (in the FO semantics). Since the lemmas from the sequence $(L_1, \ldots, L_n)$ prove α, one of the lemmas L_i has to be false on M. Since $IP(L_i)$ is true on M, we obtain that $PFP(L_i)$ is false on M. If i is the smallest index such that L_i is false on M, then we get a contradiction to the fact that $(L_1, \ldots, L_n)$ is an inductive sequence of lemmas, and hence $\mathcal{A} \cup \mathcal{D} \cup \{L_1, \ldots, L_{i-1}\} \models_{\mathsf{FO}} PFP(L_i)$. □

Table 2. Sample valid lemmas synthesized and proven correct by our tool.

Theorem	Valid Lemmas
dlist-list	$dlist(x) \Rightarrow list(x)$
slist-list	$slist(x) \Rightarrow list(x)$
sdlist-dlist	$sdlist(x) \Rightarrow dlist(x)$
sdlist-dlist-slist	$sdlist(x) \Rightarrow dlist(x)$ $sdlist(x) \Rightarrow slist(x)$
listlen-list	$list(v, l) \Rightarrow list(x)$
even-list	$even{-}lst(x) \Rightarrow list(x)$
odd-list	$odd{-}lst(x) \Rightarrow list(x)$
list-even-or-odd	$even{-}lst(x) \Rightarrow (n(x) \neq nil \Rightarrow odd{-}lst(n(x)))$ $odd{-}lst(x) \Rightarrow even{-}lst(n(x))$ $list(x) \Rightarrow ((even{-}lst(next(x)) \Rightarrow false) \Rightarrow even{-}lst(x))$
lseg-list	$lseg(x, y) \Rightarrow (list(x) \Rightarrow list(y))$
lseg-next	$lseg(x, y) \Rightarrow (lseg(y, z) \Rightarrow lseg(x, z))$
lseg-next-dyn	$lsegy(x) \Rightarrow lsegz_p(x)$
lseg-trans	$lseg(x, y) \Rightarrow (lseg(y, z) \Rightarrow lseg(x, z))$
lseg-trans2	$lseg(x, y) \Rightarrow (lseg(y, z) \Rightarrow lseg(x, z))$
lseg-ext	$lseg(x, y) \Rightarrow (lseg(y, z) \vee (lseg(x, z) \Rightarrow lseg(z, y)))$
lseg-nil-list	$lseg(x, y) \Rightarrow (list(y) \Rightarrow list(x))$
slseg-nil-slist	$slseg(x, y) \Rightarrow (slist(y) \Rightarrow slist(x))$
list-hlist-list	$list(x) \Rightarrow (y \in hlist(x) \Rightarrow list(y))$
list-hlist-lseg	$list(x) \Rightarrow (y \in hlist(x) \Rightarrow lseg(x, y))$
list-lseg-keys	$lseg(x, y) \Rightarrow (k \in keys(y) \Rightarrow k \in keys(x))$
list-lseg-keys2	$lseg(x, y) \Rightarrow (list(x) \Rightarrow list(y))$ $lseg(x, y) \Rightarrow (k \in keys(y) \Rightarrow k \in keys(x))$
rlist-list	$rlist(x) \Rightarrow list(x)$
rlist-black-height	$rlist(x) \Rightarrow red{-}height(n(x)) \leq black{-}height(x) + 1$ $rlist(x) \Rightarrow 1 \leq red{-}height(n(x)) + 1$ $rlist(x) \Rightarrow red{-}height(x) = 1 + black{-}height(n(x))$ $rlist(x) \Rightarrow black{-}height(n(x)) + black{-}height(x) \leq red{-}height(n(x)) + black{-}height(n(x))$
rlist-red-height	$rlist(x) \Rightarrow red{-}height(x) = 1 + black{-}height(n(x))$ $rlist(x) \Rightarrow (black(x) \Rightarrow red(next(x))$ $rlist(x) \Rightarrow 1 \leq red{-}height(x)$
cyclic-next	$lseg(x, y) \Rightarrow lseg(n(x), n(y))$
tree-dag	$tree(x) \Rightarrow dag(x)$
bst-tree	$bst(x) \Rightarrow tree(x)$
maxheap-dag	$maxheap(x) \Rightarrow dag(x)$
maxheap-tree	$maxheap(x) \Rightarrow tree(x)$
tree-p-tree	$tree_p(x) \Rightarrow tree(x)$

A.3 Lemmas Proved

Tables 2 and 3 represent all the lemmas proved valid by our tool. All variables (x, y, z, k, etc.) are implicitly universally quantified. Notably, different runs of our tool may produce different valid lemmas. Additionally, not all lemmas are guaranteed to be useful in proving the given theorem.

Table 3. Sample valid lemmas synthesized and proven correct by our tool.

Theorem	Valid Lemmas
tree-p-reach	$reach(x, y) \Rightarrow (tree_p(x) \Rightarrow tree_p(y))$
tree-p-reach-tree	$tree_p(x) \Rightarrow tree(x)$ $reach(x, y) \Rightarrow (tree_p(x) \Rightarrow tree(y))$ $reach(x, y) \Rightarrow (y \neq nil \Rightarrow y \in htree(x))$
tree-reach	$reach(x, y) \Rightarrow (tree(x) \Rightarrow tree(y))$
tree-reach2	$reach(x, y) \Rightarrow (tree(x) \Rightarrow tree(y))$
dag-reach	$reach(x, y) \Rightarrow (dag(x) \Rightarrow dag(y))$
dag-reach2	$reach(x, y) \Rightarrow (dag(x) \Rightarrow dag(y))$
reach-left-right	$reach(x, y) \Rightarrow (y \in htree(y) \Rightarrow x \in htree(x))$ $reach(x, y) \Rightarrow (y \in htree(y) \Rightarrow y \in htree(x))$ $reach(x, y) \Rightarrow (tree(x) \Rightarrow tree(y))$ $tree(x) \Rightarrow (y \in htree(x) \Rightarrow tree(y))$
bst-left	$bst(x) \Rightarrow (k \in keys(x) \Rightarrow minr(x) \leq k)$
bst-right	$bst(x) \Rightarrow (k \in keys(x) \Rightarrow k \leq maxr(x))$
bst-leftmost	$bst(x) \Rightarrow minr(leftmost(x)) = minr(x)$ $bst(x) \Rightarrow maxr(leftmost(x)) \leq maxr(x)$ $bst(x) \Rightarrow bst(leftmost(x))$ $bst(x) \Rightarrow key(leftmost(x)) \leq key(x)$ $bst(x) \Rightarrow (x \neq nil \Rightarrow leftmost(x) \neq nil)$ $bst(x) \Rightarrow ((bst(leftmost(x)) \Rightarrow x \in hbst(x)) \Rightarrow key(leftmost(x)) \leq minr(leftmost(x)))$
bst-left-right	$bst(x) \Rightarrow (y \in hbst(x) \Rightarrow minr(x) \leq minr(y))$ $bst(x) \Rightarrow (y \in hbst(x) \Rightarrow maxr(y) \leq maxr(x))$ $bst(x) \Rightarrow (y \in hbst(x) \Rightarrow bst(y))$ $bst(x) \Rightarrow (y \in hbst(x) \Rightarrow y \neq nil)$ $bst(x) \Rightarrow (minr(x) \leq maxr(y) \Rightarrow y \in hbst(y))$ $bst(x) \Rightarrow (minr(y) \leq maxr(x) \Rightarrow (bst(y) \Rightarrow y \in hbst(y)))$
bst-maximal	$bst(x) \Rightarrow (y \in hbst(x) \Rightarrow bst(y))$
bst-minimal	$bst(x) \Rightarrow (y \in hbst(x) \Rightarrow bst(y))$
maxheap-htree-key	$maxheap(x) \Rightarrow (y \in htree(x) \Rightarrow key(y) \leq key(x))$ $maxheap(x) \Rightarrow (y \in htree(x) \Rightarrow maxheap(y))$
maxheap-keys	$maxheap(x) \Rightarrow (k \in keys(x) \Rightarrow k \leq key(x))$
reachability	$reach(z) \Rightarrow (c = y(z) \vee n(x(z)) = n(y(z)))$
reachability2	$reach(z) \Rightarrow y(z) = x(z)$
reachability3	$reach(z) \Rightarrow x(z) = y(z)$
reachability4	$reach(z) \Rightarrow y(z) = x(z)$
reachability5	$reach(z) \Rightarrow (n(y(z)) = x(z) \vee y(z) = c)$
reachability6	$reach(z) \Rightarrow n(y(z)) = x(z)$