Synthesizing Inductive Lemmas for Reasoning with First-Order Logic with Least Fixpoints

ADITHYA MURALI, University of Illinois, Urbana-Champaign, USA
LUCAS PENA, 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 datastructures embedded in a pointer-based heap and their properties are naturally expressed in pure first-order logic with least fixpoint definitions (FO+\textit{lfp}) combined with background theories. However, automated reasoning for such logics has not seen much progress. Such logics, unlike pure FOL, do not even admit complete procedures, let alone decidable ones. In this paper, we undertake a foundational study of automatically finding proofs that use induction to reason in these logics. By treating proofs as pure FO proofs that are punctuated by declarations of induction lemmas, we separate proofs into deductively reasoned components and statements of lemmas that need to be \textit{synthesized}. While humans divine such lemmas with intuition, we propose a technique that guides the synthesis of such lemmas using counterexamples that are finite first-order models that witness the help required for proving a goal theorem as well as non-provability and invalidity of lemmas. We develop relatively complete procedures for synthesizing lemmas for powerful FO+\textit{lfp} logics. We implement our procedures and evaluate them over a class of theorems involving heap datastructures that require inductive proofs.

1 INTRODUCTION

One of the key revolutions that has spurred program verification is \textit{automated reasoning of logics}. Particularly, in deductive verification, verification engineers write inductive invariants that punctuate recursive loops and contracts for methods and use logical analysis to reason with verification conditions that correspond to correctness of small, loop-free snippets. In this realm, automatic reasoning of combinations of quantifier-free theories using SMT solvers has been particularly useful; these tools in turn have been based on the fact that these logics often have a decidable validity (and satisfiability) problem [Barrett et al. 2011; Bradley and Manna 2007].

However, reasoning even with loop-free snippets of programs is challenging when reasoning with code manipulating \textit{linked datastructures} embedded in pointer-based heaps. Datastructures are finite but unbounded structures, often characterized using recursive definitions, whose semantics is defined using least fixpoints. Datastructure manipulation in imperative programming languages is also complex, as reasoning with them well requires capturing their footprint on the heap in order to do frame reasoning when code destructively modifies the heap.

First-order logic with least fixpoint definitions (FO+\textit{lfp}) that has access to various background sorts/theories (including integers and sets) is a powerful extension of first-order logic that can define datastructures and express properties about them. For example, there are fairly expressive dialects of separation logic that have been translated to FO+\textit{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 on automated reasoning for FO logics with least fixpoint definitions (or recursive definitions) (FO+\textit{lfp} or FO-RD).

Authors’ addresses: Adithya Murali, Department of Computer Science, University of Illinois, Urbana-Champaign, USA, adithya5@illinois.edu; Lucas Pena, 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.
Our automation of FO+\textit{lfp} reasoning builds over complete procedures for FO reasoning. Pure first-order logics (even with a finite or computable set of axiomatizations) admit complete proof systems (by Gödel’s completeness theorems [Enderton 2001]), and validity of formulas is recursively enumerable (though undecidable). In this paper, we are primarily interested in extending a particular complete reasoning technique for FOL, called natural proofs, that is based on systematic quantifier instantiation followed by SMT reasoning of the resulting quantifier-free formulas [Löding et al. 2018; Madhusudan et al. 2012; Pek et al. 2014; Qiu et al. 2013; Suter et al. 2010]. Given a theorem \( \alpha \) to prove valid, we take its negation, Skolemize it to get a universally quantified formula, and then instantiate universally quantified variables using a finite set of terms, systematically. Each such instantiation results in a quantifier-free formula which admits decidable satisfiability, and it is guaranteed that if \( \alpha \) is valid, there will be some instantiation of terms that results in an unsatisfiable formula for the negation of \( \alpha \). Recent work on natural proof techniques shows that formula-based quantifier instantiation, where we instantiate universally quantified variables using only terms that appear in instantiated formulas, iteratively, followed by quantifier-free reasoning, results in complete procedures [Löding et al. 2018]. This completeness result was also extended to certain first-order logics with background sorts that are constrained by theories such as arithmetic and sets [Löding et al. 2018]).

The Anatomy of Proofs for FO+\textit{lfp}: Proofs by Induction. In contrast to FO, FO+\textit{lfp} does not admit complete procedures.\footnote{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.} Note that one can define a number line, true addition, and true multiplication over naturals using least fixpoints. Hence even quantifier-free logic with recursive definitions has an undecidable validity (and satisfiability) problem, by Gödel’s incompleteness theorem [Enderton 2001].

Humans usually prove properties involving recursive definitions (or least fixpoints) using induction. A proof of a theorem by induction typically involves sub-proofs, where each sub-proof identifies a fairly strong property (called the induction hypothesis) and a proof of the induction hypothesis called an induction step. In this paper, we use a more general notion of induction proofs based on partial fixpoints and Park induction, which doesn’t require notions of size, etc. to induct on. We defer this notion to later, and instead encourage the reader to simply think of sub-proof claims as induction hypotheses and the sub-proofs as the induction step proof. Let us call inductive hypotheses lemmas (or inductive lemmas) and the statement of the induction step, whose proof establishes the lemma, as the PFP of the lemma (PFP stands for pre-fixpoint$^2$).

The main proposal of this paper is to build automated reasoning for FO+\textit{lfp} (with background theories) using a combination (a) complete procedures for FO reasoning to prove theorems and lemmas, and (b) counter-example guided expression synthesis for synthesizing lemmas (induction hypotheses) that aid proving a theorem.

A crucial observation we make in this paper is that in proofs by induction, the proof of the induction step (PFP) of the formula is typically reasoned with using pure first-order logic reasoning without induction. More precisely, we can think of a proof of a theorem in FO+\textit{lfp} as split into sub-proofs mediated by an induction principle but otherwise consisting of purely FO reasoning. The induction principle says that proving the PFP of the lemma proves the lemma (i.e., proving the induction step proves the lemma). We can even write the induction step as an FO axiom of the form

$^2$More precisely, the PFP of a formula \( \varphi : \forall x. R(x) \Rightarrow \psi(x) \), where \( R \) has a recursive definition \( R(x) :=_{\text{def}} \rho(x, R) \) is \( \text{PFP}(\varphi) : \forall x. \rho(x, R \leftarrow R \land \psi) \rightarrow \psi(x) \).
**PFP(Lemma) ⇒ Lemma**, where **Lemma** is a lemma and **PFP(Lemma)** is the PFP/induction-step of the lemma **Lemma**.

We can hence view proofs of a theorem $\alpha$ using induction structured as:

- We identify a finite set of lemmas $L = \{L_1, \ldots, L_n\}$.
- For each $i \in [1, n]$, there is a purely first-order logic proof of $PFP(L_i)$ using earlier lemmas $(\{L_1, \ldots, L_{i-1}\})$ as assumptions.
- There is a purely FO logic proof of $\alpha$ with the lemmas $L$ as assumptions.

Note that the above proofs do not have any induction proof explicitly, and the pure FO logic proofs work under the assumption that each inductive relation $R$ is simply a fixpoint definition (not least fixpoint) of the form $\forall x. R(\bar{x}) \iff \rho(\bar{x})$ instead of $\forall x. R(\bar{x}) :=_{lfp} \rho(\bar{x})$. The fact that proving the PFP of $L_i$ is sufficient proof of $L_i$ is implicit and is the only place where the least fixpoint definition of recursive definitions is appealed to in order to argue that the above constitutes a proof of the theorem.

The above structure of proving a theorem by having essentially FO proofs punctuated by lemmas that are inductive, and whose inductiveness is proved without using further induction (but which can use other lemmas that are in turn proved by induction), is quite natural in formal proof systems that use induction. For example, in Peano arithmetic, the induction axiom scheme is:

$$\forall \bar{y}. (\varphi(0, \bar{y}) \land (\forall x. \varphi(x, \bar{y}) \Rightarrow \varphi(S(x), \bar{y}))) \Rightarrow \forall x. \varphi(x, \bar{y})$$

for any formula $\varphi$. Any proof using this axiom can hence be seen as proving lemmas of the form $\forall x. \varphi(x, \bar{y})$, by proving $\forall \bar{y}. (\varphi(0, \bar{y}) \land (\forall x. \varphi(x, \bar{y}) \Rightarrow \varphi(S(x), \bar{y})))$, using purely first-order logic over the non-inductive axioms.

The view of the proof of an $FO+lfp$ formula as largely pure FO proofs mediated by induction principles has the clear advantage that we can automate the proof of each of the required FO proofs using established complete techniques such as quantifier instantiation. Consequently, the main challenge for automation becomes finding lemmas that are inductively provable using pure FO reasoning and help prove the main theorem, again using pure FO reasoning.

**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) that aid the proof of the target theorem. We embrace the paradigm of counterexample guided synthesis that has had an impressive success in automating verification and synthesis (e.g., in finding predicates for abstraction [Ball and Rajamani 2002; Namjoshi and Kurshan 2000], in program synthesis through the CEGIS paradigm [Alur et al. 2015; Solar Lezama 2008; Solar-Lezama et al. 2007], etc.). The salient feature of our technique is the use of finite first-order models that act as counterexamples that guide the search for lemmas.

Let us assume that the theorem to be proved, $\alpha$, is indeed valid. Let us also fix our method for automated FO-reasoning to be that based on systematic quantifier instantiation. Let the method $FO-Reason(k)$ denote the method that systematically instantiates terms of depth $k$ for quantified variables, followed by checking satisfiability of the resulting quantifier-free formula (which is a decidable problem). We know that the method is complete, in the sense that if $\beta$ is a valid formula in FOL, then there is some $k$ for which $FO-Reason(k)$ will prove it valid [Löding et al. 2018].

Our lemma synthesis procedure at any point would have synthesized a set of (potentially useful) lemmas that have been proven valid, and looks for a new lemma to help proving $\alpha$. We explore three kinds of counterexample models to guide the search for useful and provable lemmas. In our iterative framework for synthesizing lemmas, a prover and a synthesizer interact, where the synthesizer proposes lemmas, and the prover provides constraints for synthesizing new lemmas. When a lemma proposed by the synthesizer, the lemma can be (a) valid and provable using $FO-Reason(k)$ reasoning...
using existing lemmas, (b) invalid, but easily shown to be invalid using a small model, or (c) valid or invalid, but in any case not provable using \( \text{FO-Reason}(k) \) using existing lemmas. Note that (a) and (c) cover all cases, and (b) overlaps with (c).

- **Type 1** models guide the search of lemmas towards those that will help prove the theorem. When we fail to prove the target theorem \( \alpha \) from current lemmas (i.e., \( L_1 \land \ldots \land L_i \Rightarrow \alpha \)) using bounded quantifier instantiation and pure FO reasoning \( \text{FO-Reason}(k) \), it results in a satisfiable quantifier-free formula from which we can extract a finite first order model. We formulate synthesis constraints that demand that new lemmas help in some way in proving \( \alpha \) by insisting that this particular model at least will not be a counterexample model for non-provability of \( \alpha \) using the lemmas with the new one included.

- **Type 2** models stem when the prover examines a proposed lemma and finds that it is clearly invalid as it is refuted by a small model \( M \) (case (b)). In this case, we can propose a set of constraints that ensure that newly synthesized lemmas at least hold (with respect to true lfp semantics) on \( M \).

- **Type 3** models cater to the last case, where the prover is unable to prove a proposed lemma from current proven lemmas. In this case, the lemma’s PFP could not be proven using \( \text{FO-Reason}(k) \) reasoning. Note that we have no idea whether these lemmas are valid or not—we just know we can’t prove their PFP using \( k \)-depth instantiation reasoning. However, since the reasoning works using bounded quantifier instantiation, it turns out that we can always find a finite model that serves as a witness as to why the lemma wasn’t provable (even if it is valid!). We add constraints that demand that newly synthesized lemmas cannot have PFPs that are false in this counterexample model.

The three kinds of model-based restrictions above narrow the space of lemmas that we want to consider. The salient aspect of our synthesis procedure is that it uses the crucial power of the bounded quantifier instantiation technique followed by SMT reasoning in that it is capable of producing finite first-order models that serve as counterexamples that guide lemma search.

The first and third kind of counterexamples are sufficient to build a counterexample based procedure, and this leads to one version of our algorithm. The second kind of counterexample may not necessarily exist—it may be that all models of size \( n \), for a small \( n \), are not counterexamples to a lemma or it may be that the lemma is valid but not provable (it may require more instantiation depth or require more inductive lemmas). However, it turns out that small counterexamples, when they exist, can be very useful, and we can incorporate them in a hybrid technique—we propose the second kind of counterexamples if they can be found for a proposed lemma, but if they do not exist, then we resort to the third kind. We show empirically that this hybrid strategy does indeed perform better on our benchmarks.

**Background theories and relative completeness:** The techniques for inductive reasoning we develop in this paper are considerably more complex than described above. First, for many applications like program verification it is important to handle some domains that are constrained to satisfy certain theories, like arithmetic and sets (sets are important in heap-based verification in order to talk about collections such as “set of keys stored in a list” and for heaplets for frame reasoning such as “the set of heap locations that constitute list”). Consequently, we work in a framework where there is a foreground sort modeling the heap with pointers and multiple background sorts, where background sorts are constrained by theories and admit decision procedures for quantifier-free formulas even for combined theories. In such settings, the work in [Löding et al. 2018] proved that for certain safe fragments FO logic, systematic quantifier instantiation is still a complete procedure. We study lemma synthesis for theorems where quantification is restricted only to the foreground sort; this falls into the safe fragment, and as argued in [Löding et al. 2018], captures a large class
of properties of heaps. Moreover, when quantifiers are instantiated using foreground terms, the resulting formula are quantifier free, and are hence amenable to Nelson-Oppen based reasoning of combined theories.

Second, we build lemma search in careful ways so that they admit relative completeness. We show that if there is a proof of theorem involving a finite set of lemmas (corresponding to a grammar of lemmas that the user gives), then our procedure is guaranteed to eventually find one. More precisely, there are two *infinities* to explore— one is the search for lemmas and one is the instantiation depth \( k \) chosen for finding proofs. As long as our procedure is run in a way that dovetails fairly between the two infinities (for any depth \( k \), the procedure for finding lemmas with depth \( k \) is run for an unbounded amount of time), then we are guaranteed to find a proof.

**Evaluation:** We implement and evaluate our procedure for a logic that combines a foreground uninterpreted sort with background sorts, where the background sorts have a decidable quantifier-free fragment using SMT solvers. Our tool framework can employ both generic SyGuS (syntax-guided synthesis) engines as well as a custom synthesis tool we built, both of which can synthesize lemmas using first-order counterexample models that are modeled using logical constraints for synthesis. The latter is a synthesis engine we develop that synthesizes lemmas from *grounded* constraints which are sufficient to express constraints on first-order models. We implement the term instantiation technique to prove FOL properties (used for proving both lemmas and the main theorem) as well as model generation for all counterexamples using an SMT solver. We evaluate our tools using a large class of benchmarks that call for reasoning over datastructures embedded in a pointer-based heap, where pointers are modeled using unary functions on the foreground sort. Our benchmarks are distilled from verification conditions and properties of datastructures that call for lemma synthesis, and cannot be proved just using FO reasoning. We evaluate several design decisions and optimizations that justify the effectiveness of the techniques that underlie our tool, including the use of our custom synthesis solver and the use of various kinds of counterexample models. Lemma synthesis has been studied for other related logics, in particular 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 is hard, we provide a comparison of our tool against tools for these logics on our benchmark theorems using appropriate encodings whenever feasible. Our experiments give evidence that the first-order counterexample based techniques proposed in this paper are effective in synthesizing inductive lemmas and proving theorems.

## 2 PRELIMINARIES AND OVERVIEW

### 2.1 First Order Logic with Recursive Definitions and Least Fixpoints

We work with first-order logic with recursive definitions that have least fixpoint semantics. The logic is over a universe that has a single *foreground* sort and multiple *background* sorts. The foreground sort (and functions and relations on it) is completely uninterpreted, while background sorts are restricted by certain theories. Further, we allow recursive definitions be defined only on the foreground sort. In our applications the foreground sort will be used to model heaps, with pointers modeled using uninterpreted unary functions, pointer variables modeled using first-order variables, and datastructures modeled using recursive definitions.

Note that the heap domain is *not* the same as that of algebraic datatypes (ADTs): heaps consist of locations with pointers we can have overlapping datastructures (two linked lists that merge), can have cycles (cyclic lists), and segments (a list segment between two pointer variables).

We encourage the reader to think of background sorts to include sorts such as arithmetic with only addition, sets of locations, sets of integers, etc. In practice in our experiments, these are theories
that admit decidable quantifier-free fragments both individually as well as in combination with each other using Nelson-Oppen type combinations [de Moura and Bjørner 2008; Nelson 1980]. In our applications for the heap domain we use background sorts to model primitive types (say keys or other integers stored in datastructures) and properties of datastructures (heaplets, sets of keys stored in a structure, length, height, etc.).

Quantified first-order logic (even without recursive definitions and least fixpoints) expressed as combinations of these background theories are typically undecidable. For example, the first-order logic over arithmetic with addition and uninterpreted functions is undecidable [Downey 1972]. The combination of these theories with a foreground uninterpreted universe is also typically undecidable, and with recursive definitions on the foreground sort it is not even complete. In fact, first-order logic with recursive definitions without any background theories is already incomplete and one cannot expect a proof system to prove all theorems in this theory (To see this, observe that we capture the theory of arithmetic in it by defining addition and multiplication using recursive definitions).

In this paper, we restrict ourselves to first order logic with recursive definitions where (a) universal quantification is only over the foreground sort, and (b) recursive definitions define properties of tuples over foreground sorts (though the definition itself can use functions that map to background sorts, for example to define sorted lists of integers).

Formally, we work with a signature of the form $\Sigma = (S;C;F;R)$, 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 $\tau \in S$. $F$ is a set of function symbols, where each function $f \in F$ has a type of the form $\tau_1 \times \ldots \times \tau_m \rightarrow \tau$ for some $m$, with $\tau_i, \tau \in S$. $R$ is a set of relation symbols, where each relation $R \in R$ has a type of the form $\tau_1 \times \ldots \times \tau_m$.

We assume that the set of sorts contains a designated foreground sort denoted by $\sigma_f$. All the other sorts in $S$ are the background sorts, and for each such background sort $\sigma$ we allow the constant symbols of type $\sigma_f$, function symbols that have type $\sigma^n \rightarrow \sigma$ for some $n$, and relation symbols have type $\sigma^m$ for some $m$, to be constrained using an arbitrary theory $T_{\sigma}$. Functions and relations that involve multiple background sorts (but not the foreground sort) can also assumed to be constrained by some (not necessarily complete) theory $T$. We consider standard first-order logic (FO) over such signatures (see [Löding et al. 2018] for more details).

Recursive Definitions. Some of the relations in $R$ over the foreground sort can have recursive definitions, which are given in a set $D$ of definitions the form $R(\vec{x}) := \text{lfp} \ \rho_R(\vec{x})$, where $R \in R$ and $\rho_R(\vec{x})$ is a first-order logic formula in which the relation symbols that have a recursive definition in $D$ occur only positively. We assume $D$ has at most one definition for any relation $R \in R$. By abuse of notation we write $R \in D$ to indicate that $R$ has a recursive definition.

The semantics of recursively defined relations are defined to be the least fixpoint (LFP) that satisfies the relational equations (the condition that each recursive definition only refers positively to recursively defined relations ensures that the least fixpoint exists, see [Löding et al. 2018] for more details).

Example 2.1. Let $n$ be a unary function symbol of type $\sigma_f \rightarrow \sigma_f$, i.e., from the foreground sort to the foreground sort. Let $\text{nil}$ be a constant of sort $\sigma_f$. Let $\text{list}$ be a unary relation with the recursive definition:

$$ \text{list}(x) := \text{lfp} \ (x = \text{nil}) \lor \text{list}(n(x)) $$

Then, in a model $M$ that correctly interprets $\text{list}$ as the LFP of this definition $\text{list}$ is true for precisely those elements that correspond to finite linked lists with $n$ as next pointer.
Similarly we can express that $x$ points to a tree using the recursive definitions:

$$
\text{tree}(x) := \text{lfp} \ (x = \text{nil}) \lor (\text{tree}(\text{left}(x)) \land \text{tree}(\text{right}(x)) \land \text{htree}(\text{left}(x)) \cap \text{htree}(\text{right}(x)) = \emptyset)
$$

$$
\text{htree}(x) := \text{if} \ (x = \text{nil}, 0, \ \text{htree}(\text{left}(x)) \cup \text{htree}(\text{right}(x)))
$$

Note that here we have two pointers left and right, and these pointers can ‘merge’, i.e., point to the same element (since our datastructures are not ADTs). Therefore to define trees we define a recursive definition for the heaplet of a tree $\text{htree} : \sigma_f \rightarrow \sigma_{st}$, where $\sigma_{st}$ is a background theory of sets, and use it demand that the left and right subtrees are disjoint. This is similar to constraints used in separation logic to express trees [Reynolds 2002]. Unlike separation logic, $\text{tree}(x)$ is true in any global heap where $x$ points to a tree (there can be other locations in the model that do not belong to the tree).

A model that correctly interprets the relations from $\mathcal{D}$ as the LFP of their recursive definitions is called an LFP model. As target theorems to prove we consider quantified FO formulas $\alpha$, where all quantifications in $\alpha$ are over the foreground sort and $\alpha$ is expressed using both the interpreted functions/relations as well as the recursively defined relations. For an FO formula $\alpha$ and a set $\Phi$ of FO formulas we write $\Phi \cup \mathcal{D} \models_{\text{LFP}} \alpha$ if $\alpha$ is true in all LFP models of $\Phi$.

**First-order abstractions of recursive definitions:** The recursive definitions in $\mathcal{D}$ can also be interpreted to use fixpoint (not lfp) semantics using standard FO formulas $\forall \bar{x}. \ R(\bar{x}) \leftrightarrow \rho_R(\bar{x})$. We write $\Phi \cup \mathcal{D} \models_{\text{FO}} \alpha$ if $\alpha$ is true in all models of $\Phi$ in which the relations $R \in \mathcal{D}$ satisfy the FO version of the recursive definitions. Note that if $\Phi \cup \mathcal{D} \models_{\text{FO}} \alpha$ holds, then $\Phi \cup \mathcal{D} \models_{\text{LFP}} \alpha$, but the converse is not necessarily true. Interpreting recursive definitions as fixpoint definitions rather than least fixpoint definitions is hence a form of sound abstraction.

### 2.2 Proofs for FO+Ifp based on Induction

Our goal is to provide algorithms that prove an FO formula $\alpha$ given a finite set $\mathcal{A}$ of assumptions (or axioms) and a set $\mathcal{D}$ of recursive definitions with lfp semantics. We want to show that $\mathcal{A} \cup \mathcal{D} \models_{\text{LFP}} \alpha$, but we want to do so using only first-order reasoning. Clearly, if $\mathcal{A} \cup \mathcal{D} \models_{\text{FO}} \alpha$, then also $\mathcal{A} \cup \mathcal{D} \models_{\text{LFP}} \alpha$ since a least fixpoint is also a fixpoint. However, the other direction is not true in general. The idea of our approach is to use intermediate inductive lemmas for finding a proof of $\alpha$.

In this work we only consider lemmas of the form $L = \forall \bar{x}. \ R(\bar{x}) \rightarrow \psi(\bar{x})$ for a quantifier-free formula $\psi$ and a recursively defined relation $R \in \mathcal{D}$. The following induction principle $IP(L)$ for the lemma $L$ expresses the fact that if $\psi(\bar{x})$ is a pre-fixpoint of the recursive definition of $R$, then the lemma is valid.

$$
IP(L) := \text{PFP}(L) \rightarrow L \quad \text{with} \quad \text{PFP}(L) := \forall \bar{x}. \ R(\bar{x}, R) \rightarrow (\psi \land R) \rightarrow \psi(\bar{x})
$$

where $\rho_R(\bar{x}, R \leftarrow (\psi \land R))$ is the formula obtained from $\rho_R(\bar{x})$ by replacing every occurrence of $R(t_1, \ldots, t_k)$ for terms $t_1, \ldots, t_k$ in $\rho_R$ by $\psi(t_1, \ldots, t_k) \land R(t_1, \ldots, t_k)$. The induction principle is valid for any lemma, and can be used to prove a lemma correct in the LFP semantics, as given by the following result:

**Theorem 2.2.** If $\mathcal{A} \cup \mathcal{D} \models_{\text{FO}} \text{PFP}(L)$ then $\mathcal{A} \cup \mathcal{D} \models_{\text{LFP}} L$. \hfill $\square$

This is an easy consequence of LFP semantics (see [Löding et al. 2018] for a formal proof).

**Definition 2.3 (Inductive Lemmas).** A lemma $L$ is inductive for $\mathcal{A} \cup \mathcal{D}$ if $\mathcal{A} \cup \mathcal{D} \models_{\text{FO}} \text{PFP}(L)$. If $\mathcal{A}$ and $\mathcal{D}$ are clear from the context, we omit them and just say that $L$ is inductive. \hfill $\square$
Definition 2.4 (Inductive lemmas that prove a theorem). A set of $L = \{L_1, \ldots, L_n\}$ is said to prove a theorem $\alpha$ if $\mathcal{A} \cup \mathcal{D} \cup L \models_{\text{FO}} \alpha$ and the lemmas are valid in LFP semantics, i.e., $\mathcal{A} \cup \mathcal{D} \models_{\text{LFP}} L_i$ for each $i \in \{1, \ldots, n\}$. □

There are two kinds of inductive lemmas that are relevant in this paper. The first one is the central problem that we wish to solve, namely synthesizing sequential lemmas that result in proving a theorem:

Definition 2.5 (Sequential Lemmas that Prove a Theorem). A sequence $(L_1, \ldots, L_n)$ of lemmas is said to provide an inductive proof of $\alpha$ if the set $L = \{L_1, \ldots, L_n\}$ proves $\alpha$ in the sense of Definition 2.4 and $L_i$ (for $1 \leq i \leq n$) is inductive for $\mathcal{A} \cup \mathcal{D} \cup \{L_1, \ldots, L_{i-1}\}$ in the sense of Definition 2.3.

A weaker class of inductive proofs is when a set of lemmas are independently inductive and help prove a theorem:

Definition 2.6 (Independent Lemmas that Prove a Theorem). A set of lemmas $L = \{L_1, \ldots, L_n\}$ is said to provide an inductive proof of $\alpha$ if $L$ proves $\alpha$ in the sense of Definition 2.4 and $L_i$ (for each $1 \leq i \leq n$) is inductive for $\mathcal{A} \cup \mathcal{D}$ in the sense of Definition 2.3.

The difference in the above two definitions is that the inductiveness of lemmas in a sequential proof can depend on previous lemmas, while in a proof based on independent lemmas, the proof of inductiveness of each lemma is independent of the other lemmas. Note that if a set of lemmas $\{L_1, \ldots, L_n\}$ is a set of independent lemmas for a theorem $\alpha$, then $(L_1, \ldots, L_n)$ is a sequence of lemmas for $\alpha$ as well, while the converse is not always true.

We can now define the lemma synthesis problem we study:

Definition 2.7 (Sequential Lemma Synthesis Problem). Given a grammar $G$ for expressing lemmas and a theorem $\alpha$, find a sequence of lemmas admitted by $G$ that proves $\alpha$.

We present, in the next section, our core algorithm FOSSIL for solving the above 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 theorem as long as there is a set of independent lemmas that prove the theorem.

Note that the FOSSIL algorithm does not guarantee finding a proof when sequential lemmas exist that prove the theorem, though as we show in our evaluation that it works on all our benchmarks. In Section 3.6, we discuss several strategies to refine the FOSSIL algorithm to achieve relative completeness with respect to sequential lemmas as well, though these algorithms are considerably more complex.

2.3 Proofs for FOL without lfp: Natural Proofs/Quantifier Instantiation

Given a theorem and a sequence of lemmas to prove the theorem, the proof of the theorem and proof of the inductiveness of the lemmas (using their PFP) is done using pure FO reasoning (interpreting recursive definitions using fixpoints and not lfp). We now describe the automated technique that we use for FO proofs, namely the mechanism of natural proofs based on systematic quantifier instantiation. Natural proofs is a technique for proving theorems in many-sorted first-order logic [Löding et al. 2018; Pek et al. 2014; Qiu et al. 2013] that is based on instantiation of universally quantified formulas. This technique operates over a logical universe of foreground and background sorts similar to this work.

In order to prove a theorem $\alpha$ under a finite set of axioms and recursive definitions (which are universally quantified), we negate the theorem, and Skolemize the axioms and definitions to
get a set of purely universally quantified formula. Validity of the theorem under the axioms and
definitions then reduces to checking that this set of formulas is unsatisfiable.

Let $\Phi$ be a set of formulas of the form $\forall \bar{x}. \eta(\bar{x})$ with all variables in $\bar{x}$ of foreground sort, and $\eta(\bar{x})$ quantifier free. We sometimes omit the quantifiers, and just refer to the formula as $\eta(\bar{x})$, implicitly assuming that the free variables in $\eta$ are quantified universally.

For a set $T$ of ground terms of foreground sort, we denote by $\Phi[T]$ the set of all quantifier free
formulas that are obtained by instantiating the formulas from $\Phi$ by the terms in $T$, i.e.,

$$\Phi[T] := \{\eta(\bar{t}/\bar{x}) \mid \eta(\bar{x}) \in \Phi \text{ and } \bar{t} \text{ a tuple of terms from } T\}$$

By $T_k(\Phi)$ we denote the set of all ground terms of depth at most $k$ over the signature of $\Phi$. Natural proofs reduces the unsatisfiability of FO formulae where the universally quantified variables are only over the foreground sort to the unsatisfiability of $\Phi[T_k(\Phi)]$, which is a quantifier-free formula
that can be decided using Nelson-Oppen combination of the foreground and background theories
(and typically amenable to reasoning using SMT solvers). The work in [Löding et al. 2018] shows
that this natural proof technique is complete:

**Theorem 2.8 (From [Löding et al. 2018]).** If a formula $\Phi$ has quantification only over the
foreground sort is unsatisfiable, then there exists a $k$ such that $\Phi[T_k(\Phi)]$ is unsatisfiable.

A solver can hence choose increasingly larger bounds $k$ and attempt to prove the given quantified
goal by generating instantiations using terms of depth $k$. We will refer to this method as proof
by using natural proofs of depth $k$. Theorem 2.8 proved in [Löding et al. 2018]] shows that natural
proofs is a complete technique for proving formulas in FO for formulas that quantify only over the
foreground sort. We use this result to prove the completeness of our own algorithms.

### 3 ALGORITHMS FOR SEQUENTIAL LEMMA SYNTHESIS

In this section we present FOSSIL(First-Order Solver with Synthesis of Inductive Lemmas), our
algorithm for solving the Sequential Lemma Synthesis problem as defined by Definition 2.7. FOS-
SIL is a counterexample-based lemma synthesis algorithm that orchestrates interactions between
three components as shown in Figure 1: a first-order verifier based on natural proofs, a synthesis
engine (for synthesizing lemmas given logical constraints on them), and a bounded counterexample
generator (for proposed lemmas). We present the algorithm and describe it in Section 3.1. We follow
this description with a fully worked-out illustrative example in Section 3.2. Then, we describe each
of the components in Figure 1 in detail in Sections 3.3 and 3.4.

In Section 3.5 we state and prove the completeness of FOSSIL for the problem of independent
lemmas that prove the goal (see Definition 2.6). The completeness result states that if there is a set
of independent lemmas that prove the goal, then our algorithm will eventually prove it. Finally, we
conclude the discussion with some thoughts on the problem of finding an algorithm that is complete
for the full problem of sequential lemma synthesis and discuss several alternatives, including a
natural extension of FOSSIL that we call FOSSIL-IP.

#### 3.1 FOSSIL

In this section we will first provide a bird’s eye view, and next provide a more detailed step-
by-step description of FOSSIL. Given an FO+lfp theory specified by a set $\mathcal{A}$ of axioms, a set
$\mathcal{D}$ of recursively defined functions, a goal formula $\alpha$, and a grammar $G$ over a many-sorted FO
signature with recursive definitions, FOSSIL finds a proof of $\alpha$ by synthesising a sequence of lemmas
$L = (L_1, L_2, \ldots, L_k)$ belonging to $L(G)$ such that $L$ is a sequence of lemmas that prove $\alpha$ as given
by Definition 2.5. The provability is established in the FO+lfp theory given by $\mathcal{A}$ and $\mathcal{D}$. 
Adithya Murali, Lucas Pena, Eion Blanchard, Christof Löding, and P. Madhusudan

Fig. 1. Components of FOSSIL

**Bird's Eye View.** The high-level components of FOSSIL and the interaction between them is shown in Figure 1. FOSSIL is a recursively enumerable search for a sequence of lemmas that prove $\alpha$. We present the pseudocode for the orchestration module labeled by 'FOSSIL Algorithm' (Figure 1) in Figure 2 which uses the following abstractions corresponding to the high-level components:

- **NaturalProofs($\Phi$, $T$)**, a First-Order verification engine based on Natural Proofs [Löding et al. 2018; Qiu et al. 2013], corresponding to the module labeled by 'First-Order Verifier' in Figure 1. It takes in a formula $\Phi$ with quantification over the foreground sort and a set of terms $T$, and outputs either that $\Phi$ is valid or unprovable.

- **GenerateCounterexample($\phi$)**: this is a counterexample generation module whose input is a quantifier-free formula $\phi$ (over a combination of SMT theories [de Moura and Björner 2008; Nelson 1980]) and the output is a finite model that witnesses the satisfiability of $\phi$ if it is satisfiable, and otherwise returns no model. It is depicted as a submodule of the verifier module in Figure 1. When the natural proof engine instantiates the quantified variables in the negation of the theorem with terms, we get a quantifier-free formula, which if found satisfiable, we extract a finite model using this module.

- **Synthesize($C$, $G$)** corresponding to a synthesis module, labeled by 'Synthesis Engine' in Figure 1. It takes a set of constraints $C$ over a variable $L$ referring to a quantified formula and a grammar $G$, and either returns a formula belonging to the language $\text{Lang}(G)$ satisfying the constraints $C$, or outputs that such a formula does not exist in the given grammar.

- **GenerateTrueCounterexample($\neg \phi$, $Th$, size_bound)**, which abstracts a module that generates FO+$\text{Ifp}$ models of bounded size. It corresponds to the module labeled 'Bounded True Counterexample Generator' in Figure 1. The module is given a negated formula $\neg \phi$, a set of axioms and recursive definitions that define an FO+$\text{Ifp}$ theory $Th$, and a positive integer size_bound. It either finds an FO+$\text{Ifp}$ model of the given theory bounded in size by the given integer that witnesses the invalidity of $\phi$, or returns that no such model can be found.

FOSSIL takes as input an FO+$\text{Ifp}$ theory defined using a set of axioms and recursively defined functions as well as a goal formula for which a proof must be found. It is designed as a recursively enumerable (r.e.) procedure that either returns a proof in the form of a sequence of lemmas, or does not terminate.
Counterexamples. FOSSIL is a counterexample guided algorithm that uses the verification and synthesis modules in rounds of lemma proposals. Recall that the problem requires finding a sequence \( \mathcal{L} \) of lemmas such that: (i) the lemmas help prove the goal: \( \mathcal{A} \cup \mathcal{D} \cup \mathcal{L} \models \text{LFP } \alpha \), and (ii) the lemmas are valid: \( \mathcal{A} \cup \mathcal{D} \cup \{ \mathcal{L}_j \mid j < i \} \models \text{LFP } \mathcal{L}_i \) for \( 1 \leq i \leq |\mathcal{L}| \). The algorithm uses three kinds of counterexamples to guide the synthesis module towards lemmas satisfying (i) and (ii):

- A Type−1 finite FO model that witnesses the current non-FO-provability of the goal \( \alpha \) (assuming lemmas proven thus far). This guides the synthesis towards new lemmas that help prove the goal as required by condition (i).
- A set of Type−2 models, which are finite FO+\text{lfp} models of the given theory that witness the invalidity of a lemma proposed by the synthesis module. They satisfy condition (ii) as they help the synthesis module propose fewer invalid lemmas. These are the models returned by the True Counterexample Generator component in Figure 1 and are referred to as ‘true’ because the valuation of recursive definitions is consistent with least-fixpoint semantics throughout the model, unlike the other two types of counterexamples.
- A set of Type−3 finite FO models that witness the failure of an induction proof of a proposed lemma. They also correspond to condition (ii), but these models do not show that the proposed lemma is invalid, only that we cannot prove it by induction and the current discovered and proven lemmas. They help the synthesis module propose fewer unprovable (and therefore undesirable) lemmas. Unlike Type−2 models, these models can always be found for unprovable lemmas even if we are not able to check if they are truly invalid. They also constrain the synthesis less strongly than Type−2 models since they are not counterexamples to validity, but rather are only counterexamples to non-provability.

Observe that all the counterexample models are finite, i.e., they have finite foreground universes (see Section 2 for a description of First-Order models over multiple sorts as used in this work).

Algorithm Description. We shall now provide a step-by-step description of main orchestration module labeled by ‘FOSSIL Algorithm’ in Figure 1. We provide the pseudocode for the algorithm over abstractions of the high-level components in Figure 2 and illustrate it on a concrete example in Section 3.2. The inputs to the algorithm are a set of axioms \( \mathcal{A} \) and a set of recursively defined functions \( \mathcal{D} \), along with a grammar \( \mathcal{G} \) whose language potentially contains the lemmas of interest, and the goal \( \alpha \) which is a formula quantified universally and only over the foreground sort. We also use two additional parameters in our presentation. Observe from the description in Section 2.3 that Natural Proofs uses an integer for depth of instantiation, used to control the expressive power of the formula-driven quantifier instantiation procedure. We add this depth parameter \( k \) to the input of FOSSIL, as well as another parameter \( h \) that refers to a measure of subsets of \( \mathcal{G} \) that we use to systematically explore the entire grammar. Since FOSSIL is to be an r.e. procedure, sometimes the presented algorithm is simply restarted with larger values for the parameters \( k \) and \( h \) to search for increasingly complex proofs of \( \alpha \).

For ease of presentation, instead of using formula-driven instantiation for Natural Proofs we compute a set of terms \( T^* \) that overapproximates of the set of instantiation terms that may be used in any possible verification query at the given depth \( k \). We then only use these terms for instantiation uniformly for every verification query. We do this (see lines 1 and 2 in Figure 2) by first using the grammar height parameter \( h \) to compute a finite sub-grammar \( \mathcal{G}_h \) of the grammar \( \mathcal{G} \) and use that to derive the total set of terms that may occur in any verification query. The following presentation therefore elides details of formula-driven instantiation techniques in FOSSIL.

At a general point in the execution on line 5, we have the following: a formula \( \Phi_\alpha \) which is the current goal (namely, that the valid lemmas found thus far imply the theorem), a set of terms \( T^* \)
FOSSIL($\mathcal{A}, \mathcal{D}, \mathcal{G}, \alpha, k, h$)

**Input:** axioms $\mathcal{A}$, recursive definitions $\mathcal{D}$, grammar $\mathcal{G}$, goal formula $\alpha$, natural proofs depth parameter $k$, lemma production height parameter $h$.

1. Compute $\mathcal{G}_h \subseteq \mathcal{G}$ such that $\text{Lang}(\mathcal{G}_h)$ does not contain any formulas whose parse-tree in $\mathcal{G}$ has a height greater than $h$.
2. $T^* := T_h(\mathcal{A} \cup \mathcal{D} \cup \{\lnot \alpha\} \cup \{\text{pfp}_L \mid \text{pfp}_L = \text{PFP}(L) \text{ for } L \in \mathcal{G}_h\})$
3. $\mathcal{L} := ()$, $\text{Type-2} := \emptyset$, and $\text{Type-3} := \emptyset$ for each $R \in \mathcal{D}$
4. $\Phi_\alpha := \mathcal{A} \cup \mathcal{D} \cup \{\lnot \alpha\}$
5. While (NaturalProofs($\Phi_\alpha, T^* \neq \text{VALID}$))
6. $\text{Type-1} = \text{GenerateCounterexample}(\Phi_\alpha[T^*])$
7. **While (True)**
8. $L = \text{Synthesize}(C, \mathcal{G}_h)$ such that $L(\bar{x}) = \forall \bar{x}.R(\bar{x}) \rightarrow \psi(\bar{x})$ satisfies constraints $C$ given by:
   a. $\text{Type-1} \not\models L[T^*]$
   b. $M \models L$ for every $M \in \text{Type-2}$
   c. $M_R \models \text{PFP}(L)[T^*]$ for each $(M, R) \in \text{Type-3}$
9. If no such lemma exists, call FOSSIL($\mathcal{A}, \mathcal{D}, \mathcal{G}, \alpha, k+1, h+1$)
10. $\Phi_L := \mathcal{A} \cup \mathcal{D} \cup \mathcal{L} \cup \{\lnot \text{PFP}(L)\}$
11. If (NaturalProofs($\Phi_L, T^* \neq \text{VALID}$) THEN
12. // Unprovable Lemma
13. $M_L = \text{GenerateTrueCounterexample}(\lnot L, \mathcal{A} \cup \mathcal{D}, \text{size_bound})$
14. If ($M_L$ found)
15. // Invalid Lemma
16. $\text{Type-2} := \text{Type-2} \cup \{M_L\}$
17. **ELSE**
18. // Irrefutable and Unprovable Lemma
19. $M_R = \text{GenerateCounterexample}(\Phi_L[T^*])$
20. $\text{Type-3} := \text{Type-3} \cup \{(M, R)\}$
21. **CONTINUE LOOP ON LINE 7**
22. **ELSE**
23. // Valid Lemma
24. $\Phi_\alpha := \Phi_\alpha \cup \{L\}; \mathcal{L} := \mathcal{L} \circ (L)$ (sequence extension)
25. $\text{Type-3} := \emptyset$
26. **BREAK; GOTO LINE 5**

**Output:** sequence $\mathcal{L}$ of lemmas that prove $\alpha$

Fig. 2. FOSSIL Algorithm

used for instantiating the quantified variables in the goal, a grammar $\mathcal{G}_h$ to search for lemmas, a sequence $\mathcal{L}$ of valid lemmas in $\text{Lang}(\mathcal{G}_h)$ discovered hitherto, and countermodels $\text{Type-1}, \text{Type-2}$ and $\text{Type-3}$ whose meaning is as explained above. We first try to prove the goal with the current bag of lemmas on line 5 using the Natural proofs verifier module. If the module returns that the goal is provable (i.e., that the query is Valid), then we exit and declare that a proof has been found. Otherwise, we seek a $\text{Type-1}$ countermodel from the counterexample generation module on line 6. This model witnesses that the goal is not currently provable at the current depth of instantiation (using $T^*$).

We use the obtained $\text{Type-1}$ model along with the currently available $\text{Type-2}$ and $\text{Type-3}$ models and query the synthesis module Synthesize on line 8 for a lemma $L$ from the grammar $\mathcal{G}_h$ satisfying the following constraints: (a) $L$ is false on the $\text{Type-1}$ model on the sub-universe $T^*$ (line 8a) — this is because the $\text{Type-1}$ model represents the non-provability of the goal, and a lemma that eliminates this model could help prove the goal; (b) $L$ holds on every model in the $\text{Type-2}$
models (line 8b) — since Type−2 models are ‘true’ models of the FO+\textit{lfp} theory, this constraint enforces that the proposed lemma be a valid formula in the given theory; (c) \( L \) is inductive on every relevant model from the Type−3 models (line 8c) — since Type−3 models witness the failure of an induction proof of earlier proposal, this constraint enforces that the proposed lemma be inductively provable.

If the synthesis module returns that no such lemma can be found, we stop and call FOSSIL again with larger parameters \( k \) for depth of instantiation for verification and \( h \) for the subset of the grammar to be explored for synthesis on line 9. If a lemma \( L \) is proposed, we first construct the first-order formula \( \Phi_L \) that represents an induction proof of \( L \) on line 10. This involves using the PFP of \( L \) as described in Section 2.3. Then, we query the NaturalProofs module on line 12 with \( \Phi_L \).

If the lemma cannot be proven valid, we query the GenerateTrueCounterexample module for a Type−2 model that refutes \( L \) on line 13, fixing a bound \textit{size\_bound} prior to the algorithm. If such a model is found, then we take the ‘Invalid Lemma’ branch on line 14 and add it to the set of Type−2 models on line 16. We continue searching for a valid lemma on line 21. If such a ‘true’ model cannot be found, we take the ‘Unprovable Lemma’ branch on line 17 and obtain a Type−3 model to witness the failure of the induction proof \( \Phi_L \) on line 19 using the GenerateCounterexample module. We add it to the current set of Type−3 models and once again continue the search for valid lemmas on line 21.

Finally, if a the proposal \( L \) can be proved valid on line 12 we take the ‘Valid Lemma’ branch on line 22 add it to the sequence \( \mathcal{L} \) of synthesized lemmas on line 24. We also drop the set of Type−3 models and reset it to the empty set on line 25 since lemmas whose induction proofs failed earlier may now be provable and we would like for those to be proposed again. We break out of the inner synthesis loop and try to prove the goal again with the new valid lemma included on line 26.

We will now demonstrate the algorithm and the many possible branches in it on a concrete example. We detail the internals of the verification, synthesis, and counterexample generation components later in Sections 3.3 and 3.4.

### 3.2 Illustrative Example

Let us consider the following example of an actual execution of our tool: given the following list and list segment recursive definitions

\[
\text{list}(x) := \text{lfp} (x = \text{nil}) \lor \text{list}(n(x)) \quad \text{lseg}(x, y) := \text{lfp} (x = y) \lor \text{lseg}(n(x), y)
\]

and the following short program:

\[
\text{if (x == nil) \{ ret := nil \} else \{ ret := n(x) \}}
\]

with precondition \( \text{lseg}(x, y) \land \text{list}(y) \) and postcondition \( \text{list}(\text{ret}) \), we will prove the validity of this triple. Note that we are working in a theory where we model \text{nil} as a heap location. We first generate the verification condition for the triple as an FO+\textit{lfp} formula:

\[
\text{lseg}(x, y) \implies (\text{list}(y) \implies (\text{ite}(x = \text{nil} : \text{ret} = \text{nil}, \text{ret} = n(x)) \implies \text{list}(\text{ret})))
\]

Despite the relative simplicity of this formula, not only is it not provable using FO techniques, but it is also not provable using natural proofs and not provable by induction with it itself as the induction hypothesis. Indeed, we feed this formula to the module NaturalProofs, where we get back a model which sets \( x = 0, y = 2 \), and \( \text{ret} = 1 \), with \( 0 \implies 1 \implies 5 \implies 6 \implies 1 \). We use \( u \mapsto v \) to represent \( n(u) = v \). This corresponds to the Type−1 countermodel from line 6 in Figure 2. Note that this cycle shows that \( \text{lseg}(x, y) \) is not true when we fully evaluate the \textit{lfp} definitions. However, our natural proof depth does not unfold the definitions enough to witness that the ‘true’ meaning of list segment in this case.
With the model generated, we next search for a lemma using the Synthesize module satisfying the properties from line 8 in Figure 2. The first lemma proposed is \( \text{list}(x) \implies \text{lseg}(y, x) \). This lemma is not true (and hence not provable), and when we feed it to the GenerateTrueCounterexample module (line 13) to find a \( \text{Type} - 2 \) model exhibiting its invalidity. The model generated sets \( x = 1 \), and \( y = \text{nil} = 2 \), with \( 1 \mapsto 2 \). The antecedent is true as \( x \) represents a single node immediately pointing to \( \text{nil} \). However, the consequent is equivalent to \( \text{lseg}(\text{nil}, x) \) which is false as \( x \) is not \( \text{nil} \). This concrete model is added to our set of \( \text{Type} - 2 \) models and the process continues, where new lemmas proposed must also be true on this model.

This repeats, with new models being added, when the Synthesize module proposes the lemma \( \text{list}(x) \implies \text{lseg}(x, \text{nil}) \). The prefixpoint of this lemma is valid and provable, so we query the NaturalProofs module with this lemma added to check if the original goal is now provable. It still is not provable, so the process continues, with this lemma added to our set \( \mathcal{L} \) of valid lemmas.

Invalid lemmas continue to be proposed, with \( \text{Type} - 2 \) models being generated, until the valid lemma \( \text{lseg}(x, y) \implies (\text{lseg}(y, \text{nil}) \iff \text{list}(x)) \) is proposed. The PFP of this lemma is:

\[
\text{ite}(x = y, \top, \text{lseg}(y, \text{nil}) \iff \text{list}(n(x)) \land \text{lseg}(n(x), y)) \implies (\text{lseg}(y, \text{nil}) \iff \text{list}(x))
\]

However, unlike the previous lemma proposed, we are unable to prove the PFP of this lemma. So the GenerateCounterexample module is called (line 19), and a \( \text{Type} - 3 \) model is generated, in this case where \( x = y = 0, \text{nil} = 2 \), and \( 0 \mapsto 1 \mapsto 6 \mapsto 3 \mapsto 4 \mapsto 5 \mapsto 1 \). The antecedent of the PFP is trivially true since \( x = y \). However, similar to the first model discussed in this example, the looping behavior shows that \( \text{lseg}(y, \text{nil}) \) and \( \text{list}(x) \) are both untrue with the recursive definitions fully evaluated. But, the first-order reasoning depth is again not large enough to exhibit this equivalence. The \( \text{Type} - 3 \) model is then added on line 20, and the process continues.

Finally, the Synthesize module proposes the lemma \( \text{lseg}(x, y) \land \text{list}(y) \implies \text{list}(x) \). The prefix-point of this lemma is proved valid using the NaturalProofs module. This time, the NaturalProofs module is able to prove the original goal with this lemma (and the other valid lemmas at this point), so we break from the loop on line 5 and our proof is complete.

We will now provide details of the individual modules in Figure 1. We refer the reader to the discussion in Section 2.3 and only describe the synthesis and counterexample generation modules below.

### 3.3 Synthesis Engine

The synthesis module Synthesize in FOSSIL takes a grammar for expressing formulas (lemmas) that generates a finite language along with a set of \textit{ground} constraints \( \psi(e) \). It then produces a formula \( \varphi \) conforming to the grammar such that \( \psi \) is true when \( e \) is replaced with \( \varphi \).

This problem formulation is similar to SyGuS problems [Alur et al. 2015, 2018] in that we have a grammar and constraints on the synthesized expression. However, in SyGuS, specifications can be more complex, of the form \( \forall x. \psi(e, x) \). In contrast, the constraints we generate are \textit{grounded constraints} that have no quantification/variables. However, we can of course use SyGuS solvers as synthesis engines for our problem, and indeed we do so in a version of our implementation (see Section 4.1).

We now describe a custom synthesis engine that is particularly suited for ground constraints. The advantage of having a finite grammar and ground constraints is that the synthesis problem can then be reduced to a quantifier-free query over a combination of theories, which can be effectively handled by modern SMT solvers [de Moura and Björner 2008; Nelson 1980]. A quantifier-free query can be obtained since the grammar is finite and therefore has productions of finite height. We can then encode any expression in the language as a set of boolean variables representing the decision of production rules chosen for each nonterminal in the grammar.
The interesting aspect of the synthesis is then the fact that all the constraints in lines 8a, 8b, and 8c can be encoded as ground constraints. As we have mentioned earlier and will explain in Section 3.4, all our counterexample models have finite foreground universes. Therefore, they can be encoded as conjunctions over atomic formulas representing the valuation of all functions over the foreground universe. If the constraints on the model are themselves quantifier-free, a condition of the form \( M \models \psi \) can be written as Encode(M) \( \equiv \psi[t \leftarrow M(t)] \) where Encode(M) is the encoding of M using atomic formulas and every term in \( \psi \) is replaced with its valuation in M.

Finally, it is easy to see that the constraints are quantifier-free. The constraints on lines 8a and 8c are obtained by instantiating away universal quantifiers using a set of terms \( T^* \) and the constraint on line 8b, which refers to a universally quantified formula L, is specified over a finite Type\(2\) model and can therefore be written as a quantifier-free formula by expanding each quantifier into a conjunction over all infinitely many instantiations from the model’s finite foreground universe.

We implement this synthesis technique (see Section 4.1) in a custom synthesizer and evaluate its efficacy in Section 4.

### 3.4 Counterexample Generators

FOSSIL uses three kinds of finite counterexample models to guide the lemma synthesis. The Type\(1\) model witnesses the non-provability of the goal with the current set of lemmas synthesized and makes the synthesis goal-directed. The Type\(2\) models witness the invalidity of proposals and guide the synthesis towards producing valid lemmas, and the Type\(3\) models witness the non-inductiveness of proposals and help guide the synthesis towards producing provable lemmas.

Among these, the counterexamples for Type\(1\) and Type\(3\) cases are generated by using the GenerateCounterexample module as shown on lines 6 and 19 in Figure 2. These are obtained as a consequence of using the NaturalProofs module that reduces verification of a quantified formula \( \Phi \) to the satisfiability of a quantifier-free formula \( \phi \) as indicated in Section 2.3.

The case of Type\(2\) models produced by the GenerateTrueCounterexample module is more involved. We achieve the generation of such models by using an SMT solver. Given the bound \( n \) on the size of the model required, we can construct a formula that represents the existence of \( n \) elements such that the valuation of functions, including recursively defined functions, satisfies the axioms and falsifies the given lemma. In order to ensure that the valuation of recursively defined functions is respects the least-fixpoint semantics, we add the notion of the rank of \( R \) for every \( R \) in \( \mathcal{D} \).

Let us consider the simple case of a recursively defined unary predicate \( R \) whose definition \( R(x) := \text{lfp} \ \rho(x, R) \). Assume that for any \( x \), \( \rho \) refers to \( R \) over a particular set of terms— say \( R(t_1(x)), R(t_2(x)), \ldots R(t_n) \). The rank function \( \text{Rank}_R \) for \( R \) is a function from the domain of \( R \) to natural numbers along with \( -1 \). We ensure several constraints: (a) \( R \) holds on \( u \) iff the rank of \( u \) is not \( -1 \), (b) if rank of \( u \) is non-negative, then the witnessing atomic formulæ \( R(t_i(u)) \) that make \( \rho(u, R) \) true are such that each \( t_i \) gets a smaller rank non-negative than the rank of \( u \), and vice-versa, (c) if the rank of \( u \) is \( -1 \), then no matter which set of witnessing atomic formulæ \( R(t_i(u)) \) we pick such that their truth would make \( \rho(u, R) \) true, there is at least one \( t_i \) whose rank is \( -1 \).

It is easy to see that if we assign ranks according to the standard iterative computation of least fixpoints, assigning the rank of any element \( u \) 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 exist that satisfies the above constraints, then we can be assured that \( R \) evaluates to the true least fixpoint. The above constraints over a bounded model can be expressed in SMT in order produce true counterexamples to lemmas.
3.5 Relative Completeness of FOSSIL wrt Independent Lemmas

The soundness of FOSSIL is clear from the problem description and the termination conditions in Figure 2: the branch on line 22 is only taken when a lemma is proved valid, and the loop condition on line 5 establishes that if FOSSIL terminates it does so with a sequence of lemmas that prove \( \alpha \). But it is unclear whether the algorithm will always find a sequence of lemmas in \( G \) that prove \( \alpha \) if one exists. Indeed, FOSSIL is not complete for the problem of sequential lemma synthesis, but it turns out FOSSIL is complete wrt independent lemmas. That is, if there is a set of independent lemmas that prove \( \alpha \), then there is a depth of instantiation and a grammar height such that FOSSIL will find a proof of \( \alpha \). (Note that FOSSIL however does produce sequential lemmas.)

**Theorem 3.1 (Relative completeness of FOSSIL wrt independent lemmas).** If \( \alpha \) is provable from \( \mathcal{A} \) and \( \mathcal{D} \) by a finite set of independent inductive lemmas in \( G \) in the sense of Definition 2.6, then there is an instantiation depth \( k \) and a grammar height \( h \) such that FOSSIL(\( \mathcal{A}, \mathcal{D}, G, \alpha, k, h \)) (see Figure 2) terminates and returns a sequence \( L \) of lemmas that proves \( \alpha \).

**Proof Gist.** Assume that there exists some set of independent lemmas \( \{L_1, L_2, \ldots, L_n\} \) that proves \( \alpha \). We induct on the number \( n \) of lemmas in the set. 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 \( G_h \) that \( \text{Lang}(G_h) \) is finite for any \( h \). Observe from the description of the algorithm in Section 3.1 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 \( \Phi_\alpha \) 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 \( L \).

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 \( M \models (\mathcal{A} \cup \mathcal{D} \cup \{\neg \alpha\} \cup \{L_i\})[T^*] \) for any \( L_i, 1 \leq i \leq n \). This yields that \( M \models (\mathcal{A} \cup \mathcal{D} \cup \{\neg \alpha\} \cup \{L_i|1 \leq i \leq n\})[T^*] \), which contradicts our initial assumption that \( \{L_1, \ldots, L_n\} \) collectively prove \( \alpha \) at depth \( k \), i.e., \( (\mathcal{A} \cup \mathcal{D} \cup \{\neg \alpha\} \cup \{L_i|1 \leq i \leq n\})[T^*] \) is unsatisfiable. Therefore some \( L_i \) satisfies the constraint on line 8a and will eventually be proposed, which concludes our proof gist. See Appendix A.1 for a detailed proof.

3.6 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 3.1). 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 3.2 (FOSSIL is not complete for sequence of lemmas).** Consider the case where \( \alpha \) 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 \( \alpha \) is provable assuming \( L_2 \). At the beginning of the algorithm on line 6 in Figure 2,
$L_2$ would be false on Type $\text{-} 1$ since it helps prove $\alpha$. But there is nothing that prevents $L_1[T]$ from being true on Type $\text{-} 1$, 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 20 witnessing that $L_2$ has no inductive proof. However, the Type $\text{-} 1$ 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 $\alpha$ 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 $\text{-} 1$ models. This eliminates the problem described in Example 3.2 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 $\text{-} 2$ and Type $\text{-} 3$ 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 3.2, it would be useful if we could update Type $\text{-} 1$ 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 2.2: the induction principle of a lemma $L(\bar{x})$ is given by $(\forall \bar{x}.\text{PFP}(L(\bar{x}))) \to (\forall \bar{x}.L(\bar{x})) \equiv \neg(\forall \bar{x}.\text{PFP}(L(\bar{x}))) \lor (\forall \bar{x}.L(\bar{x}))$ where PFP represents the condition that $L$ is inductive. Therefore the induction principle captures the two possibilities of $L$ 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 $\Phi_\alpha$ and $\Phi_\ell$ on lines 4 and 10. Then, given a proposal $L$ that we can neither prove nor establish as being invalid using a Type $\text{-} 2$ model (line 17), we would eliminate falling back to a Type $\text{-} 3$ model on lines 19 and 20 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 3.3 (Relative completeness of FOSSIL-IP wrt sequential lemmas).** If $\alpha$ 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 (see Figure 4 in Appendix A.1.1) terminates and returns a set $\mathcal{L}$ of lemmas and a set $\mathcal{IP}$ of induction principles proving $\alpha$. 
We detail the formulation of proving a theorem using induction principles, the pseudocode for the FOSSIL-IP algorithm and the proof of its relative completeness in Appendix A.1.1. Admittedly, despite the elegance of being able to handle unprovable lemmas uniformly and being a natural extension to FOSSIL using induction principles, 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.

4 IMPLEMENTATION AND EVALUATION
In this section we will describe our implementation and evaluation of the FOSSIL algorithm described in Section 3.1.

4.1 Implementation
We implement our algorithm 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 as described in Sections 3.3 and 3.4. Our implementation makes two major contributions.

The first major contribution is an implementation of the NaturalProofs module (see Section 3.1). To our knowledge this is the first implementation of Natural Proofs [Löding et al. 2018; Pek et al. 2014; Qiu et al. 2013] that gives a general tool operating over a multi-sorted first-order logic with recursive definitions for modeling heaps. We added to this implementation a mechanism for generating induction templates/induction principles for formulae that can use Natural Proofs to prove formulae using induction as well as a mechanism to provide finite counterexample models, implementing the GenerateCounterexample module. Lastly, we also implement the mechanism described in Section 3.4 to generate bounded true counterexamples.

A second major contribution is the implementation of a custom SyGuS solver based on constraint solving that synthesizes quantified expressions from a grammar given ground constraints. This is also to our knowledge the first implementation that solves the problem of (finite) model-based synthesis. We implement this solver on top of Z3, using the technique described in Section 3.3 that reduces synthesis to the satisfiability of quantifier-free formulas.

4.2 Research Questions
We will now turn to the problem of evaluating our implementation of FOSSIL as well as the custom synthesis solver. In particular, we wish to answer the following Research Questions (RQs):

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

We first describe the benchmark suite that we created for the problem of sequential lemma synthesis before discussing the results on experiments that evaluate the above RQs.

4.3 Benchmarks
We curate a class of 50 theorems that are valid in FO+\textit{lfp}, not valid in FO (with recursive definitions taken as fixpoints and not least fixpoints) and are not are easily provable by induction (using their own statement as the induction hypothesis) and hence require new lemmas. Many of the benchmarks were distilled from the VCDryad [Pek et al. 2014] repository of heap verification using Dryad, a variant of Separation Logic. 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]. We then converted the remaining VCs that required lemma synthesis (the VCDryad work required lemmas to be provided manually) into FO+lfp benchmarks. Some other benchmarks are inspired from from the work in [Löding et al. 2018] which used an induction principle to prove FO+lfp theorems, and some theorems capture interesting relations between properties of datastructures. We list these benchmarks in Table 1 and the lemmas synthesized by our tool in Appendix A.2.

A final class of benchmarks involves partial correctness of scalar programs with loops. We write the computation of the program as a recursive definition of reachability over program configurations, and seek lemmas corresponding to loop invariants in order to prove that any state satisfying the precondition will reach a state satisfying the postcondition when the loop exits. Our suite contains six such benchmarks.

### 4.4 Effectiveness of FOSSIL in proving theorems

| Theorem         | Lemmas proposed | Lemmas proved | Time |
|-----------------|-----------------|---------------|------|
| dlist-list      | 1               | 1             | 1s   |
| slist-list      | 1               | 1             | 1s   |
| sdlist-dist     | 1               | 1             | 2s   |
| sdlist-dist-slist | 3            | 2             | 3s   |
| listen-list     | 1               | 1             | 1s   |
| even-list       | 1               | 1             | 2s   |
| odd-list        | 4               | 2             | 4s   |
| list-even-or-odd | 7             | 3             | 9s   |
| lseg-list       | 9               | 2             | 11s  |
| lseg-next       | 6               | 1             | 9s   |
| lseg-next-dyn   | 1               | 1             | 2s   |
| lseg-trans      | 5               | 1             | 8s   |
| lseg-trans2     | 8               | 1             | 16s  |
| lseg-ext        | 7               | 1             | 5s   |
| lseg-nil-list   | 8               | 1             | 10s  |
| lseg-nil-slist  | 9               | 1             | 13s  |
| list-hlist-list | 5               | 1             | 3s   |
| list-hlist-lseg | 14              | 4             | 13s  |
| list-lseg-keys  | 15              | 3             | 20s  |
| list-lseg-keys2 | 17             | 3             | 19s  |
| rlist-list      | 1               | 1             | 1s   |
| rlist-black-height | 4            | 1             | 3s   |
| rlist-red-height | 2             | 1             | 2s   |
| cyclic-next     | 7               | 1             | 10s  |
| tree-dag        | 1               | 1             | 1s   |

| Theorem         | Lemmas proposed | Lemmas proved | Time |
|-----------------|-----------------|---------------|------|
| bst-tree        | 1               | 1             | 3s   |
| maxheap-dag     | 2               | 1             | 3s   |
| maxheap-tree    | 2               | 1             | 2s   |
| tree-p-tree     | 1               | 1             | 1s   |
| tree-p-reach    | 13              | 2             | 24s  |
| tree-p-reach-tree | 6            | 2             | 12s  |
| tree-reach      | 10              | 2             | 48s  |
| tree-reach2     | 11              | 2             | 35s  |
| dag-reach       | 7               | 2             | 6s   |
| dag-reach2      | 8               | 2             | 7s   |
| reach-left-right | 19             | 6             | 89s  |
| bst-left        | 11              | 3             | 82s  |
| bst-right       | 6               | 1             | 119s |
| bst-leftmost    | 14              | 3             | 59s  |
| bst-left-right  | 25              | 5             | 132s |
| bst-maximal     | 5               | 1             | 5s   |
| bst-minimal     | 6               | 1             | 7s   |
| maxheap-htree-key | 5         | 1             | 4s   |
| maxheap-keys    | 5               | 1             | 14s  |
| reachability    | 8               | 1             | 10s  |
| reachability2   | 3               | 1             | 4s   |
| reachability3   | 3               | 1             | 4s   |
| reachability4   | 3               | 1             | 3s   |
| reachability5   | 11              | 1             | 16s  |
| reachability6   | 3               | 1             | 4s   |

Table 1. FOSSIL experimental results

Table 1 displays the benchmarks, along with the total time taken by the FOSSIL tool to prove each theorem, to study RQ1. The FOSSIL tool refers to our main tool, with all three kinds of counterexamples, and with our own synthesis algorithm from grounded constraints. We also report the number of lemmas proposed during the process and the subset of them that were proved valid.
Evidently, FOSSIL is very effective on these benchmarks. The total time per theorem varies from 1 second to 132 seconds. The number of total lemmas proposed varied from 1 (i.e. the first proposed lemma was sufficient) to 25, with up to 6 lemmas proven valid while proving a single theorem. We refer the reader to Appendix A.2 for a list of all lemmas proven valid during the process of proving each theorem. Several of our benchmarks required the synthesis of sequential lemmas.

Some simple lemmas assert that one datastructure is more general than another. For example, consider again the following short program from 3.2

```plaintext
if (x == nil) { ret := nil } else { ret := n(x) }
```

this time with precondition \( dlist(x) \) (stating that \( x \) points to a doubly linked list) and postcondition \( list(x) \). When converted to a VC, this is not provable in first-order logic. Further, the PFP of this theorem is also not provable. However, FOSSIL quickly synthesize the lemma \( \forall x. \ dlist(x) \Rightarrow list(x) \) (in fact it is the first lemma proposed). The PFP of this lemma is provable using the natural proof solver of FOSSIL, and with this lemma assumed, the VC of the above triple is provably valid in FOL (this is the \( dlist\)-list example in Table 1).

Other more complex lemmas proved include transitivity of list segments, combining lists and list segments (Section 3.2), and those involving loop invariants for partial correctness as mentioned in the previous section. We also include lemmas involving more complex datastructures, heaplets with sets as a background theory, and integer arithmetic. As a more interesting example, consider the theorem: \( tree_p(x) \implies (parent(x) = nil \land reach(x, y)) \Rightarrow tree(y) \). This theorem (\( tree\-p\-reach\-tree \) in Table 1) involves three recursive definitions: a tree (with left and right pointers) \( tree \), a tree with an additional parent pointer \( tree_p \), and a reachability definition on heaps \( reach \) that expresses that node \( y \) in a tree is reachable from \( x \) (using left and right pointers). Recall that the recursive definition for \( tree \) (and \( tree_p \)) must contain a clause stating that left and right pointers do not intersect: \( htree(left(x)) \cap htree(right(x)) = \emptyset \). The above theorem states that any node reachable in a tree with parent pointers itself points to a tree. The \( parent(x) = nil \) clause states that \( x \) is the root. As before, this theorem is not first-order provable, nor can it be proved on its own statement as the induction hypothesis. FOSSIL fairly quickly proposes and proves these lemmas which together are sufficient to prove the original theorem:

\[
\forall x, y. \ reach(x, y) \implies (tree_p(x) \Rightarrow tree_p(y)) \\
\forall x. \ tree_p(x) \implies tree(x)
\]

4.5 Comparison to synthesis without counterexample models

We now consider RQ2, testing the efficacy of using counterexample models during synthesis in FOSSIL. We do this in two different ways. First, we test FOSSIL using no counterexample models. We compare our tool with a version of FOSSIL integrated with the state-of-the-art SyGuS solver in CVC4 (CVC4Sy). We used an enumerative mode of CVC4Sy (which is quite optimized and does symmetry and semantic reduction), asking it to synthesize lemmas without giving counterexample first-order models. The enumerative mode of CVC4Sy is designed to avoid re-proposing equivalent yet structurally different lemmas. CVC4Sy will continue to propose potential lemmas until either no additional lemmas are possible or a timeout is reached. The potential benefit of this method is that time to create countermodels is avoided, though at the cost of proposing additional lemmas that would otherwise be eliminated.

Comparison of our tool with all counterexamples enabled and disabled can be seen in Figure 3a. Besides some outliers when the lemma or lemmas proposed are very simple, FOSSIL without counterexample models performs drastically worse than the FOSSIL tool. In fact, 29 of our 50 benchmarks did not terminate in the given timeout of 15 minutes. This shows that search for lemmas using (inequivalent) enumeration is not a feasible strategy.
Next, we evaluate the efficacy of the use of $Type^{-2}$ countermodels in FOSSIL. As discussed in Section 3, the $Type^{-2}$ countermodels are given priority over $Type^{-3}$ in FOSSIL; when the former are not found, we rely on the latter. We can, however, build a version of the tool with $Type^{-3}$ countermodels only, omitting $Type^{-2}$ countermodels.

See Figure 3b for running time comparison between the FOSSIL tool and the FOSSIL tool without $Type^{-2}$. For the majority of benchmarks, the runtime is improved when $Type^{-2}$ countermodels are utilized. In fact, even on theorems where turning off $Type^{-2}$ countermodels resulted in an
improvement in runtime, this improvement is very minimal, and are theorems that take 10s or less with both modes. In fact, 20 of the 50 benchmarks took > 10s to run with Type−2 countermodels turned off. Of these 20, 17 terminate faster with Type−2 countermodels turned on. This shows that for more complex theorems, the Type−2 ‘true’ countermodels become more impactful in stemming the search space.

Figure 3c shows a comparison in the number of proposed lemmas for FOSSIL vs. FOSSIL without Type−2 counterexample models. As expected, fewer lemmas are proposed for most benchmarks in the FOSSIL tool (which utilizes Type−2 countermodels). The instances where fewer lemmas are proposed with Type−2 countermodels disabled had a small difference in the number of lemmas proposed, and only for the simpler benchmarks.

4.6 Comparison with CVC4 SyGuS solver

To evaluate RQ3, the efficacy of our custom synthesis tool that learns from first-order models, we compare our synthesis tool with CVC4Sy, utilizing them in an identical fashion the FOSSIL tool. This time, we use the standard mode of CVC4Sy to propose lemmas, generating Type−2 and Type−3 models from the lemmas proposed.

Results from these experiments can be seen in Figure 3d. Similar to the results from Figure 3b, there is no significant difference in the running time of each approach for the simpler theorems. However, again, once theorems become more complex, FOSSIL with our custom constraint-based synthesis solver outperforms CVC4Sy. Specifically, of the 22 tests that took > 10s to run with CVC4Sy, 19 of these tests terminate faster with constraint-based synthesis. In addition, given our timeout of 15 minutes, FOSSIL with CVC4Sy resulted in two benchmarks timing out without synthesizing sufficient valid lemmas to prove the goal. With constraint-based synthesis, both of these benchmarks were proved valid in under two minutes. Thus, the constraint-based synthesis approach that is based on SAT/SMT solving seems to scale much better with more complex examples, greatly improving the versatility of our FOSSIL tool.

5 COMPARISON WITH TOOLS OVER ADTS AND SEPARATION LOGIC

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+lpf, especially one that can handle foreground and background sorts as in our setting. As far as we know, our technique is the first one that uses finite first-order models as counterexamples to help synthesizing lemmas.

Comparing tools that work for different logics (FO+lpf, algebraic datatypes, separation logic) is inherently hard and poses at least the following challenges: (a) since the logics are different, automated encodings from one logic to another are hard; manual encoding of formulas of one logic in another can call for considerable human intelligence in translation, and there are often many choices in this encoding, making fair comparisons hard; (b) tools typically support only specific restricted forms of the general logic; many a time, there may be a natural translation from a logic A to a logic B, but not necessarily to the fragment of B supported by a tool, (c) in the context of proving theorems using lemmas, the translation of a formula $\varphi$ in logic A to $\varphi'$ in logic B can result in very different problems; the lemmas required for proving $\varphi$ (with lemmas expressed in logic A) may be very different from the lemmas required for proving $\varphi'$ in logic B (with lemmas expressed in logic B), and sometimes the translated formula may not require a lemma at all.

In this section, we attempt to compare our tool with tools for algebraic datatypes (ADTs) and separation logic on our benchmarks, making the best translation effort. The reader should, however, take these comparisons with a pinch of salt as it is hard to establish the fairness of these comparisons. In each of these comparisons, we describe the challenges we face and motivate the limited
comparison we make. Our comparative study does show empirical evidence that our benchmarks in FO+lfp, when translated as well as we can to other logics, is not amenable to effective reasoning using existing tools that synthesize inductive lemmas (in particular, CVC4+ig [Reynolds and Kuncak 2015] over ADTs and SLS [Ta et al. 2017] for the SL-COMP competition for separation logic).

5.1 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 datastructures, which are different from pure mathematical algebraic datatypes (heaps admit a spaghetti of pointers that have overlapping datastructures embedded in them). 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. The challenge now is in defining least fixpoint definitions using pure first order logic over ADTs.

We encode finite pointer-linked datastructures, such as linked lists and linked trees, using ADTs such as lists and trees, respectively, that store locations that constitute the linked datastructure. 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 datastructures 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 our benchmarks in the format of CVC4+ig [Reynolds and Kuncak 2015], which allows for inductive proofs as well as subgoal generation (akin to our lemma synthesis) in the CVC4 tool. CVC4 was unable to solve any of our benchmarks.

For example, even the VC for the simple example

\[
\text{if } (x == \text{nil}) \{ \text{ret} := \text{nil} \} \text{ else } \{ \text{ret} := n(x) \}
\]

with precondition \(dlist(x)\) and postcondition \(list(x)\) mentioned above is not provable by CVC4+ig (though it can be easily proved with the inductive lemma \(\forall x. dlist(x) \Rightarrow list(x)\); in fact, CVC4+ig with this lemma given is able to prove it).

We encoded all of our benchmarks in Table 1 in CVC4+ig in a similar fashion as above. None (i.e., \(0/50\)) of these examples terminated within our timeout of 10 minutes.

5.2 Comparison with separation logic tools in SL-COMP
We consider the set of tools that participate in the the Separation Logic Competition (SL-COMP) [Sighireanu et al. 2019]. Note that though several tools do support lemma synthesis, they do not take in a specific grammar for lemmas, which is a difference from our tool.

There are many restrictions imposed by the various divisions and tools that make encoding of our benchmarks challenging. First, the symbolic heap fragment with inductive definitions (qf_-shid_entl) of the SL-COMP language is the one most closely related to our work. However, all tools that participated for this fragment do not support conjunction of heap formulas. Formulations of several of our problems required such conjunctions. A more complex encoding without using such conjunctions required changing the recursive definitions themselves and resulted in fundamentally different problems that require different and/or fewer lemmas.
Second, the CVC4 tool (which participated in the “Boolean separation logic” divisions) supports conjunctions of heap formulas, but supports only satisfiability (not entailment) and also does not support recursive definitions.

In our benchmarks, we frequently have formulas that describe different properties of the same structure. For example, the formula $\text{tree}(x) \land \text{maxr}(x) = k$, where $\text{maxr}(x) = k$ represents that $k$ is the maximum value in the heap rooted at $x$. However, since the heaplets of $\text{tree}(x)$ and $\text{maxr}(x)$ overlap, we are unable to represent such expressions in any of the SL-COMP solvers. This eliminates many of our benchmark examples, including those involving binary search trees and other complex datastructures (25 of our benchmark formulas). Some of our examples (20 of them, though only 2 that did not also have conjunction of heap formulas) have properties about heaplets expressed explicitly. These do not translate naturally to separation logic.

We consider the solver SLS (Songbird+Lemma Synthesis) [Ta et al. 2017] that won the 2019 SL-COMP competition for the \texttt{qf_shid_entl} division. SLS has support for synthesizing inductive lemmas. As mentioned above, many of our examples cannot be translated faithfully into SLS. Some formulas in our benchmarks that involve synthesizing lemmas that relate a more specific datastructure to a general one (such as \texttt{dlist-list}, \texttt{slist-list}, etc.) were solved by SLS efficiently. Other examples encoded over simple structures like list and list segments (\texttt{lseg-trans}, \texttt{lseg-nil-list}) were also solved by SLS efficiently.

We report on encodings of some additional examples that SLS failed to prove. First, \texttt{cyclic-next} in Table 2 states that if $x$ is a cyclic list, then so is the value $\text{n}(x)$ that $x$ points to. This requires a lemma about list segments, which SLS does not synthesize. Next, \texttt{list-even-or-odd} states that a list will either have an even or odd length. For this, lemmas are needed relating odd and even lists, such as $\text{odd-list}(x) \Rightarrow \text{even-list}(\text{n}(x))$. While SLS is able to prove other properties about even and odd lists or list segments, it returns \textit{unknown} for this example. Finally, all the program reachability examples (\texttt{reachability-reachability6} above) were faithfully encodable in SLS, though SLS returned \textit{unknown} for all 6 examples. Of our 50 theorems, we were able to faithfully encode and prove in SLS only 14 theorems (which were all relatively simple).

6 RELATED WORK

Quantifier instantiation is a common tool for reasoning using SMT solvers [Reynolds 2016]. 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 Björner 2008; Rümmer 2012], as well as methods for combining term instantiation with background SMT solvers [Ge and de Moura 2009].

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 in Table 1 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.

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. More recent work 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 a common heuristic for reasoning with recursive definitions known as “unfold-and-match” [Chu et al. 2015; Madhusudan et al. 2012; Nguyen and Chin 2008; Pek et al. 2014; Qiu et al. 2013; Suter et al. 2010], which involve unfolding a recursive definition a few times and finding a proof of validity with the unfolded formulas, treating recursive definitions as uninterpreted.

Other lemma synthesis approaches include the work in [Zhang et al. 2021], which also uses SyGuS for lemma generation uses models to guide the lemma search, but operates over bitvector problems, a much simpler domain. Additionally, 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 proof to generate constraints on top of a lemma template, and 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, and as a result many FO+lfp formulas are not expressible. Refer to Section 5 for a detailed comparison between FOSSIL and SLS on our set of benchmarks.

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] also is an effort to synthesize inductive lemmas, and, similar to our work, 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 between with this work in Section 5.

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), which hence 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 datastructures 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 datastructures 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; 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 5). Extending our techniques to build a lemma synthesis technique/tool with built-in support for ADTs, especially to reason with functional programs, is in fact an interesting future direction.

7 CONCLUSIONS

The primary contribution of this paper is an inductive lemma synthesis technique for FO+lfp with background theories that works by learning from semantically-rich counterexample first-order models that witness nonprovability of theorems and lemmas. We believe that such a search for lemmas that is based on the semantics of theorems/lemmas is very interesting and can be useful in other contexts— for example, in helping identify lemmas from a large corpus to help prove theorems (such as the work in [Bansal et al. 2019], where machine learning is used to find proofs, and where there is little semantic information used in learning). We believe extending our work to synthesizing lemmas for other logics, especially logics for ADTs (see Section 6) as well
separation logic, is interesting. We also believe that general lemma synthesis engines incorporated as extensions to SMT solvers can be valuable routines for researchers who wish to use such theorem proving for a variety of application domains.

REFERENCES

Rajeev Alur, Rastislav Bodík, Eric Dallal, Dana Fisman, Pranav Garg, Garavit Juniwal, Hadas Kress-Gazit, P. Madhusudan, Milo M. K. Martin, Mukund Raghothaman, Shambwadiya Saha, Sanjit A. Seshia, Rishabh Singh, Armando Solar-Lezama, Emina Torlak, and Abhishek Upadhyaya. 2015. Syntax-Guided Synthesis. 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 S-LAM Project: Debugging System Software via Static Analysis. In Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (Portland, Oregon) (POPL '02). Association for Computing Machinery, New York, NY, USA, 1–3. https://doi.org/10.1145/503272.503274

Kshitij Bansal, Sarah Loos, Markus Rabe, Christian Szegedy, and Stewart James Wilcox. 2019. HOList: An Environment for Machine Learning of Higher Order Logic Theorem Proving. In Thirty-sixth International Conference on Machine Learning (ICML). https://arxiv.org/abs/1904.03241

Clark Barrett, Christopher L. Conway, Morgan Deters, Liana Hadarean, Dejan Jovanović, Tim King, Andrew Reynolds, and Cesare Tinelli. 2011. CVC4. In Proceedings of the 23rd International Conference on Computer Aided Verification (Snowbird, UT) (CAV’11). Springer-Verlag, Berlin, Heidelberg, 171–177.

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.

James Brotherston, Dino Distefano, and Rasmus Lerchedahl Petersen. 2011. Automated Cyclic Entailment Proofs in Separation Logic. In Proceedings of the 23rd International Conference on Automated Deduction (CADE’11). Springer-Verlag, Berlin, Heidelberg, 131–146. http://dl.acm.org/citation.cfm?id=2033266.2033278

Cristiano Calcagno, Philippa Gardner, and Matthew Hague. 2005. From Separation Logic to First-Order Logic. In Foundations of Software Science and Computational Structures, Vladimir Sassone (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 395–409.

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 Berlin Heidelberg, Berlin, Heidelberg, 337–340.

Simon Cruanes. 2017. Superposition with Structural Induction. In FroCoS.

Leonardo de Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (Budapest, Hungary) (TACAS’08/ETAPS’08). Springer-Verlag, Berlin, Heidelberg, 337–340.

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

P. Downey. 1972. Undecidability of Presburger Arithmetic with a Single Monadic Predicate Letter. https://books.google.com/books?id=jXnCtgAACAAJ

H.B. Enderton. 2001. A Mathematical Introduction to Logic. Elsevier Science. https://books.google.com/books?id=dVncClEtUkC

Yeting Ge and Leonardo de Moura. 2009. Complete Instantiation for Quantified Formulas in Satisfiability Modulo Theories. In Computer Aided Verification, Ahmed Bouajjani and Oded Maler (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 306–320.

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.
Synthesizing Inductive Lemmas for Reasoning with First-Order Logic with Least Fixpoints

Bart Jacobs, Jan Smans, Pieter Philippaerts, Frédéric Vogels, Willem Penninckx, and Frank Piessens. 2011. VeriFast: A Powerful, Sound, Fast Verifier for C and Java. In Proceedings of the Third International Conference on NASA Formal Methods (Pasadena, CA) (NFM ’11). Springer-Verlag, Berlin, Heidelberg, 41–55.

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/10.1109/38.588534

Matt Kaufmann, J. Strother Moore, and Panagiotis Manolios. 2000. Computer-Aided Reasoning: An Approach. Kluwer Academic Publishers, USA.

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

Quang Loc Le, Makoto Tatsuta, Jun Sun, and Wei-Ngan Chin. 2017. A Decidable Fragment in Separation Logic with Inductive Predicates and Arithmetic. 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

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/10.1101/3158098

P. Madhusudan, Xiaokang Qiu, and Andrei Ștefănescu. 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.

Aditya 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.

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.

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

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 Atkin, 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.

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 Ștefănescu, 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. 2016. Conflicts, Models and Heuristics for Quantifier Instantiation in SMT. In Vampire@IJCAR. Berlin, Heidelberg, 80–98.

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), 55–74.

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.

Mihaela Sighireanu, Juan A. Navarro Pérez, Andrey Rybalchenko, Nikos Gorogiannis, Radu Iosif, Andrew Reynolds, Cristina Serban, Jens Katalaen, Christoph Matheja, Thomas Noll, Florian Zuleger, Wei-Ngan Chin, Quang Loc Le, Quang-Trung Ta, Ton-Chanh Le, Thanh-Toan Nguyen, Siu-Cheng Khoo, Michal Cypryan, 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, Marielke Huisman, Fabrice Kordon, and Bernhard
\section{Appendix}

\subsection{Further details for Section~3}
\textbf{Proof of Theorem 3.1.}

\textsc{Proof}. Assume that there exists some set of independent lemmas \(\{L_1, L_2, \ldots, L_n\}\) that proves \(\alpha\). 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 \(G\) 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 \(\alpha\).

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 \(\alpha\). 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 \(\Phi_\alpha\) (see line 24) 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 \(G_h\) that \(\text{Lang}(G_h)\) is finite for any \(h\). Observe from the description of the algorithm in Section~3.1 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 \(\Phi_\alpha\) 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 \(M \models (\mathcal{A} \cup \mathcal{D} \cup \{\neg \alpha\} \cup \{L_i\})[T^+]\) for any \(L_i, 1 \leq i \leq n\). This yields that \(M \models (\mathcal{A} \cup \mathcal{D} \cup \{\neg \alpha\} \cup \{L_i\} \mid 1 \leq i \leq n\})[T^+]\), which contradicts our initial assumption that \(\{L_1, \ldots, L_n\}\) collectively prove \(\alpha\) at depth \(k\), i.e., \((\mathcal{A} \cup \mathcal{D} \cup \{\neg \alpha\} \cup \{L_i\} \mid 1 \leq i \leq n\})[T^+]\) is unsatisfiable. Therefore some \(L_i\) satisfies the constraint on line 8a and will eventually be proposed, which concludes our proof. \(\Box\)

\textbf{A.1.1 Discussion about induction principles and description of FOSSIL-IP.} As illustrated in Example 3.2, 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 $\alpha$ without the algorithm being able to prove the actual lemmas valid. We illustrate the difference between induction principles and lemmas proving $\alpha$ for an (artificial) example situation.

**Example A.1.** Consider the definition of $\text{list}$ from above. Add two constants $c_1, c_2$ to the signature, and two recursive definitions $\text{list}_1$ and $\text{list}_2$:

$$
\text{list}_1(x) := \text{ifp} \quad (x = \text{nil}) \lor ((\text{list}_1(n(x)) \land (c_1 = c_2 \rightarrow x \neq c_1))
$$

$$
\text{list}_2(x) := \text{ifp} \quad (x = \text{nil}) \lor ((\text{list}_2(n(x)) \land (c_1 \neq c_2 \rightarrow x \neq c_1))
$$

So both are defined as $\text{list}$ with the only difference that the recursion stops at $c_1$ for $\text{list}_1$ if $c_1 = c_2$, and for $\text{list}_2$ if $c_1 \neq c_2$.

Take $\alpha = \forall x. \text{list}(x) \rightarrow ((\text{list}_1(x) \lor \text{list}_2(x)))$. This is certainly true in LFP semantics because if $c_1 = c_2$, then $\text{list}_2$ is the same as $\text{list}_1$; otherwise $\text{list}_2$ is the same as $\text{list}$. Consider the lemmas $L_1 = \forall x. \text{list}(x) \rightarrow \text{list}_1(x)$ and $L_2 = \forall x. \text{list}(x) \rightarrow \text{list}_2(x)$. For each lemma, there are clearly LFP models in which the lemma does not hold (if $c_1 = c_2$ and $\text{list}_1$, then $L_1$ is false, similarly for $L_2$). However, we have that $\mathcal{A} \cup \mathcal{D} \cup \{\mathcal{I}P(L_1), \mathcal{I}P(L_2)\} \models_{\mathcal{F}O} \alpha$ because on each model either $\mathcal{P}FP(L_1)$ or $\mathcal{P}FP(L_2)$ 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.2.** If $(L_1, \ldots, L_n)$ is a sequence of inductive lemmas that prove $\alpha$ then the set $\mathcal{I}P = \{\mathcal{I}P(L_1), \ldots, \mathcal{I}P(L_n)\}$ proves $\alpha$.

**Proof.** If $\mathcal{I}P$ does not prove $\alpha$, then there is a model $M$ of $\mathcal{A} \cup \mathcal{D} \cup \mathcal{I}P \cup \{\lnot \alpha\}$ (in the FO semantics). Since the lemmas from the sequence $(L_1, \ldots, L_n)$ prove $\alpha$, one of the lemmas $L_i$ has to be false on $M$. Since $\mathcal{I}P(L_i)$ is true on $M$, we obtain that $\mathcal{P}FP(L_i)$ 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 $\mathcal{A} \cup \mathcal{D} \cup \{L_1, \ldots, L_{i-1}\} \models_{\mathcal{F}O} \mathcal{P}FP(L_i)$. \qed

The algorithm FOSSIL-IP is shown in Figure 4. It has a similar structure as FOSSIL but it does not use the models $\mathcal{M}_R$ anymore. Instead it collects the induction principles for lemmas that it cannot prove, and creates a new $\text{Type} \rightarrow 1$ model in each round, as opposed to FOSSIL, where a new $\text{Type} \rightarrow 1$ model is only created when a new provably valid lemma is discovered.

The induction principles are collected in the set $\mathcal{I}P$ in line (20) of the algorithm. The induction principle for a lemma $L$ is of the form

$$
\forall \bar{x}. \lnot \mathcal{P}FP(L[\bar{c}]) \lor L(x)
$$

The formula $\lnot \mathcal{P}FP(L[\bar{c}])$ is quantifier free because the negation turns the universal quantifiers of $\mathcal{P}FP(L)$ into existential quantifiers, which are then replaced by fresh constants $\bar{c}$ (Skolemization). These constants are added to the signature, and are collected in the set $\mathcal{C}_R$ (line (21)) because they are used in the new constraint $7c$ as explained below. In order to make the constants that are used in the induction principles explicit, we use the notation $\mathcal{I}P(L, \bar{c}_L)$ in line 20, where $\bar{c}_L$ is a tuple of fresh constants introduced for the lemma $L$. Since the number of constants increases with each induction principle that is added to $\mathcal{I}P$, we cannot assume them to be present in the terms in $T^*$ from the beginning. We therefore have to extend $T^*$ with the corresponding terms involving the new constants (lines 11 and 22).
FOSSIL-IP(𝒜, Đ, G, α, k, h)

Input: axioms 𝒜, recursive definitions Đ, grammar G, goal formula α, natural proofs depth parameter k, lemma production height parameter h.

1. Compute 𝐺ℎ ⊆ 𝐺 such that Lang(𝐺ℎ) does not contain any formulas whose parse-tree in 𝐺 has a
   height greater than ℎ.
2. 𝑇 ∗ := 𝑇 ℎ(𝒜 ∪ Đ ∪ {¬α})
3. ℋ := {}, Type−2 := 0, IP := 0, and 𝐶 𝑅 := ∅ for each 𝑅 ∈ Đ
4. Φα := 𝒜 ∪ Đ ∪ {¬α}
5. While (NaturalProofs(Φα, 𝑇 ∗) ≠ VALID)
   6. Type−1 = GenerateCounterexample(Φα[𝑇 ∗])
   7. ℋ = Synthesize(C, 𝐺ℎ) such that ℋ(𝑥) = ∀𝑥. 𝑅(𝑥) → 𝜓(𝑥) satisfies constraints C given by:
      a. Type−1 ⊬ ℋ[𝑇 ∗]
      b. 𝑀 ⊨ ℋ for every 𝑀 ∈ Type−2
      c. Type−1 ⊬ PFP(ℒ[𝑐]) for each 𝑐 ∈ 𝐶 𝑅
8. If no such lemma exists, call FOSSIL-IP(𝒜, Đ, G, α, k + 1, h + 1)
9. ℎ𝑖 tuple of fresh constants of same arity as 𝑅
10. Φℓ := Φα ∪ {IP(ℒ, ℎ𝑖)}; IP := IP ∪ {IP(ℒ, ℎ𝑖)}
11. 𝑇 ℎ := 𝑇 ∗ extended by depth 𝑘 terms involving the new constants ℎ𝑖
12. If (NaturalProofs(Φℓ, 𝑇 ℎ) ≠ VALID) Then
   13. // Unprovable Lemma
   14. 𝑀 ℎ = GenerateTrueCounterexample(¬L, 𝒜 ∪ Đ, size_bound)
   15. If (𝑀 ℎ found)
   16. // Invalid Lemma
   17. Type−2 := Type−2 ∪ {𝑀 ℎ}
   18. Else
   19. // Irrefutable and Unprovable Lemma
   20. Φα := Φα ∪ {IP(L, ℎ𝑖)}; IP := IP ∪ {IP(L, ℎ𝑖)}
   21. 𝐶 𝑅 := 𝐶 𝑅 ∪ {ℎ𝑖}
   22. 𝑇 ∗ := 𝑇 ℎ
   23. Else
   24. // Valid Lemma
   25. Φα := Φα ∪ {𝐿}; 𝐿 := 𝐿 ◦ (𝐿) (sequence extension)

Output: sequence ℋ of lemmas and set of induction principles IP that prove α

Fig. 4. FOSSIL-IP Algorithm

Proof of Theorem 3.3. The proof is along the same lines as the one for Theorem 3.1. All lemmas that are added to ℋ are valid in LFP semantics, as for the first algorithm. So if the algorithm terminates it has found a set ℋ and a set IP as claimed in the statement of the theorem.

Assume that α is provable from 𝒜 and Đ by a finite sequence (𝐿₁, … , 𝐿𝑛) of inductive lemmas. Let

- Φ𝑖 := 𝒜 ∪ Đ ∪ {𝐿𝑖, … , 𝐿𝑖−1} ∪ {¬PFP(𝐿𝑖)} for each 𝑖 ∈ {1, … , 𝑛}, and
- Φ := 𝒜 ∪ Đ ∪ {𝐿₁, … , 𝐿𝑛} ∪ {¬α}.

The sets Φ𝑖 are unsatisfiable because (𝐿₁, … , 𝐿𝑛) is a sequence of inductive lemmas. The set Φ is unsatisfiable because (𝐿₁, … , 𝐿𝑛) is a sequence of inductive lemmas that proves α (see Lemma A.2).

By completeness of natural proofs (Theorem 2.8), there is an instantiation depth 𝑘 such that Φ[𝑇 ∗] and Φ𝑖[𝑇 ∗] for each 𝑖 are unsatisfiable (with 𝑇 ∗ as in the algorithm). We choose this 𝑘 as instantiation depth for the algorithm, and ℎ such that the lemmas 𝐿₁, … , 𝐿𝑛 can be derived in 𝐺ℎ
We first argue that no lemma \( L \) is proposed twice. If \( L \) is shown invalid by a Type\(-2\) model, it will not satisfy constraint 7b in future rounds. If \( L \) is shown valid, it is added to \( \Phi_\alpha \) and will not satisfy constraint 7b for the new Type\(-1\) models in future rounds. Otherwise, if \( L \) cannot be shown valid nor invalid, \( IP(L, \bar{c}_L) \) is added to \( IP \). The instantiation of \( IP(L, \bar{c}_L) \) with \( T^* \) is of the form \( \neg PFP(\bar{c}_L) \lor L[T^*] \). Hence, the next Type\(-1\) model either is a model of \( \neg PFP(\bar{c}_L) \) or a model of \( L[T^*] \), and therefore \( L \) will not satisfy constraint 7a or 7c in future rounds.

It remains to show that in each round one of the lemmas \( L_i \) is a candidate (satisfies all constraints). Constraint 7b is satisfied for all \( L_i \) by the fact that valid lemmas hold in all the Type\(-2\) models.

Now assume that no \( L_i \) satisfies 7a. Then the Type\(-1\) model would be a model of \( \Phi[T^*] \), which is unsatisfiable as explained above. So there is at least one \( L_i \) that satisfies constraint 7a. Take the minimal such \( i \). Assume that \( L_i \) does not satisfy 7c. Then there is a tuple \( \bar{c} \in C_R \) for the appropriate \( R \) such that Type\(-1\) \( \models \neg PFP(L_i[\bar{c}]) \). By the choice of \( i \), Type\(-1\) \( \models L_j[T^*] \) for each \( j < i \). Thus, Type\(-1\) \( \models \Phi_i[T^*] \), contradicting the unsatisfiability of \( \Phi_i[T^*] \).

Hence, one of the \( L_i \) satisfies the constraints, and thus is eventually proposed. Once all \( L_i \) are in \( L \), the algorithm terminates (or it terminates earlier with a different proof of \( \alpha \)).

A.2 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 2. 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 | \(sdlist(x) \Rightarrow dlist(x)\)                                       |
| lseg-list       | \(lseg(x, y) \Rightarrow list(x) \Leftrightarrow list(y)\)                |
| lseg(x, y) \Rightarrow list(x) \Leftrightarrow list(y) |
| listlen-list    | \(lseg(x, y) \Rightarrow list(x) \Leftrightarrow list(y)\)                |
| even-list       | \(even\text{-}lst(x) \Rightarrow list(x)\)                               |
| odd-list        | \(odd\text{-}lst(x) \Rightarrow list(x)\)                                |
| list-even-or-odd| \((even\text{-}lst(x) \Rightarrow list(x)) \Leftrightarrow odd\text{-}lst(n(x))\) |
| list(x) \Rightarrow ((even\text{-}lst(x) \Rightarrow odd\text{-}lst(n(nil)) \Rightarrow even\text{-}lst(n(x))) |
| lseg-nil-list   | \(lseg(x, y) \Rightarrow list(y) = list(x)\)                             |
| slseg-nil-list  | \(slseg(x, y) \Rightarrow (slist(y) \Leftrightarrow slist(x))\)           |
| list-hlist-list | \(list(x) \Rightarrow (y \in hlist(x) \Rightarrow list(y))\)            |
| list-hlist-lseg | \(list(x) \Rightarrow lseg(x, nil)\)                                      |
| \(list(x) \Rightarrow (y \in hlist(x) \Rightarrow lseg(y, nil))\) |
| \(list(x) \Rightarrow (y \in hlist(x) \Rightarrow lseg(x, y))\) |
| \(list(x) \Rightarrow (y \in hlist(x) \Rightarrow list(y))\) |
| list-lseg-keys | \(lseg(x, y) \Rightarrow (k \in keys(y) \Rightarrow k \in keys(x))\)     |
| \(lseg(x, y) \Rightarrow lseg(y, nil))\) |
| list-lseg-keys2| \(lseg(x, y) \Rightarrow (list(x) \Rightarrow list(y))\)                 |
| \(lseg(x, y) \Rightarrow (k \in keys(y) \Rightarrow k \in keys(x))\) |
Table 3. Valid lemmas synthesized and proven correct by our tool.

| theorem                  | valid lemmas                                                                 |
|--------------------------|-----------------------------------------------------------------------------|
| tree-p-reach             | \( \text{tree}_p(x) \imp \text{reach}(x, \text{nil}) \) \<br> \( \text{reach}(x, y) \imp (\text{tree}_p(x) \imp \text{tree}_p(y)) \) |
| tree-p-reach-tree        | \( \text{tree}_p(x) \imp \text{tree}(x) \) \<br> \( \text{reach}(x, y) \imp (\text{tree}_p(x) \imp \text{tree}_p(y)) \) |
| tree-reach               | \( \text{reach}(x, y) \imp (\text{tree}(x) \imp \text{tree}(y)) \) \<br> \( \text{tree}(x) \imp \text{reach}(x, \text{nil}) \) |
| tree-reach2              | \( \text{reach}(x, y) \imp (\text{tree}(x) \imp \text{tree}(y)) \) \<br> \( \text{tree}(x) \imp \text{reach}(x, \text{nil}) \) |
| dag-reach                | \( \text{reach}(x, y) \imp (\text{dag}(x) \imp \text{dag}(y)) \) \<br> \( \text{dag}(x) \imp \text{reach}(x, \text{nil}) \) |
| dag-reach2               | \( \text{reach}(x, y) \imp (\text{dag}(x) \imp \text{dag}(y)) \) \<br> \( \text{dag}(x) \imp \text{reach}(x, \text{nil}) \) |
| reach-left-right         | \( \text{reach}(x, y) \imp (\text{tree}(x) \imp \text{reach}(y, \text{nil})) \) \<br> \( \text{reach}(x, y) \imp (y \in \text{htree}(y) \imp x \in \text{htree}(x)) \) \<br> \( \text{reach}(x, y) \imp (y \in \text{htree}(y) \imp y \in \text{htree}(x)) \) \<br> \( \text{tree}(x) \imp (y \in \text{htree}(x) \imp \text{reach}(y, \text{nil})) \) \<br> \( \text{tree}(x) \imp (\text{reach}(\text{nil}, y) \imp \text{reach}(x, y)) \) \<br> \( \text{tree}(x) \imp \text{reach}(x, \text{nil}) \) |
| bst-left                 | \( \text{bst}(x) \imp \text{maxr}(x) \leq \text{minr}(\text{nil}) \) \<br> \( \text{bst}(x) \imp (k \in \text{keys}(x) \imp \text{minr}(x) \leq k) \) \<br> \( \text{bst}(x) \imp (\text{minr}(\text{nil}) \leq \text{maxr}(x) \imp k \in \text{keys}(\text{nil})) \) |
| bst-right                | \( \text{bst}(x) \imp (k \in \text{keys}(x) \imp k \leq \text{maxr}(x)) \) |
| bst-leftmost             | \( \text{bst}(x) \imp \text{minr}(\text{leftmost}(x)) = \text{minr}(x) \) \<br> \( \text{bst}(x) \imp \text{bst}(\text{leftmost}(x)) \) \<br> \( \text{bst}(x) \imp ((\text{bst}(\text{leftmost}(x))) \imp x \neq \text{nil}) \imp \text{minr}(\text{leftmost}(x)) = \text{key}(\text{leftmost}(x)) \) |
| bst-left-right           | \( \text{bst}(x) \imp \text{maxr}(\text{nil}) \leq \text{minr}(x) \) \<br> \( \text{bst}(x) \imp (y \in \text{hbst}(x) \imp \text{minr}(x) \leq \text{minr}(y)) \) \<br> \( \text{bst}(x) \imp (y \in \text{hbst}(x) \imp \text{maxr}(y) \leq \text{maxr}(x)) \) \<br> \( \text{bst}(x) \imp (\text{nil} \in \text{hbst}(x) \imp \text{nil} \in \text{hbst}(\text{nil})) \) \<br> \( \text{bst}(x) \imp \text{maxr}(x) \leq \text{minr}(x) \) |
| bst-maximal              | \( \text{bst}(x) \imp (y \in \text{hbst}(x) \imp \text{bst}(y)) \) |
| bst-minimal              | \( \text{bst}(x) \imp (y \in \text{hbst}(x) \imp \text{bst}(y)) \) |
| maxheap-htree-key        | \( \text{maxheap}(x) \imp (y \in \text{htree}(x) \imp \text{key}(y) \leq \text{key}(x)) \) |
| maxheap-keys             | \( \text{maxheap}(x) \imp (k \in \text{keys}(x) \imp k \leq \text{key}(x)) \) |
| reachability             | \( \text{reach}(z) \imp (c = y(z) \lor n(x(z)) = n(y(z))) \) |
| reachability2            | \( \text{reach}(z) \imp y(z) = x(z) \) |
| reachability3            | \( \text{reach}(z) \imp x(z) = y(z) \) |
| reachability4            | \( \text{reach}(z) \imp y(z) = x(z) \) |
| reachability5            | \( \text{reach}(z) \imp (n(y(z)) = x(z) \lor y(z) = c) \) |
| reachability6            | \( \text{reach}(z) \imp n(y(z)) = x(z) \) |