Lemma Synthesis for Automating Induction over Algebraic Data Types

Weikun Yang, Grigory Fedyukovich, and Aarti Gupta

Princeton University, Princeton NJ 08544, USA
{weikuny,grigoryf,aartig}@cs.princeton.edu

Abstract. In this paper we introduce a new approach for proving quantified theorems over inductively defined data-types. We present an automated prover that searches for a sequence of simplifications and transformations to prove the validity of a given theorem, and in the absence of required lemmas, attempts to synthesize supporting lemmas based on terms and expressions witnessed during the search for a proof. The search for lemma candidates is guided by a user-specified template, along with many automated filtering mechanisms. Validity of generated lemmas is checked recursively by our prover, supported by an off-the-shelf SMT solver. We have implemented our prover called AdtInd and show that it is able to solve many problems on which a state-of-the-art prover fails.

1 Introduction

Program verification tasks are often encoded as queries to solvers for Satisfiability Modulo Theories (SMT). Modern solvers, such as Z3 [26] and CVC4 [3], are efficient and scalable mainly on quantifier-free queries. Formulas with universally quantified formulas, which could be obtained from programs with algebraic data types (ADT), are still challenging. While quantifier-instantiation strategies [14, 25, 18] and superposition-based theorem proving [10, 23] are effective in some cases, a native support for inductive reasoning is needed to handle the full range of problems. Inductive reasoning over universally quantified formulas has been partially implemented in CVC4, in particular, using a conjecture-generation feature [30]. However, CVC4 often generates too many unrelated conjectures and does not utilize a problem-specific information.

Automating induction over ADTs has also been the target for many theorem provers. Tools such as IsaPlanner [11], ACL2 [6], Zeno [31], and HipSpec [8] can make use of induction when proving goals, with varying capabilities of automatic lemma discovery based on rippling [5] and generalization. However, the heuristics for lemma discovery are baked into the prover as fixed rules that target a limited space. These rule-based approaches are often ineffective when the form of the required lemmas is significantly different from expressions encountered during the proof attempt. There is no automated support for exploring a larger search space for candidates, and the user has to manually guide the overall search.

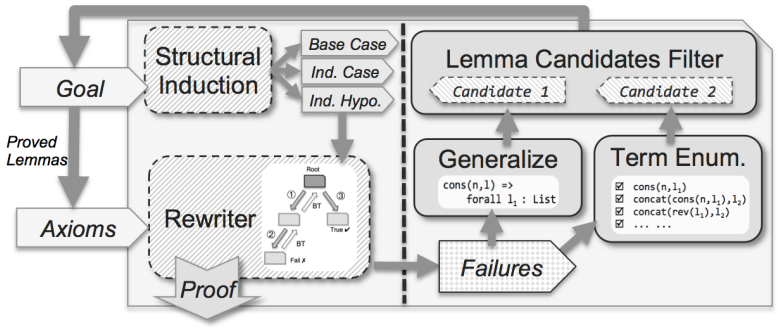

Fig. 1. AdtInd workflow.

A major challenge with both SMT-based tools and induction provers is that they often fail to produce crucial lemmas from which actual proofs follow. Technically this is due to the failure of cut-elimination in inductive theories [20], making the problem of finding proofs for many instances undecidable.

Our approach to automatic lemma discovery is inspired by the framework of syntax-guided synthesis (SyGuS) [1], applicable to program synthesis. To discover a program that meets a specification, SyGuS-based approaches take as additional input a formal grammar that defines the search space for the program. This framework has been successfully used in many applications and there exist dedicated SyGuS solvers that target various domains [2]. For example, SyGuS-based techniques have been used for program verification via generation of invariants [16, 17, 12] and termination arguments [13]. Although SyGuS has also inspired various SMT solver efforts [29, 28, 27] (described later in related work), to the best of our knowledge, none of these tackle automatic generation of lemmas for proofs by induction over ADTs.

An overview of our proposed framework is shown in Figure 1. It is built on top of an automated theorem prover based on inductive reasoning. The prover decomposes given theorems into the base-case and inductive-case subgoals and uses a backtracking rewriter that sequentially simplifies each of the subgoals toward *true*. When the prover is unable to succeed, our approach first *generalizes* the partially-rewritten formula (as done in prior efforts) by replacing certain concrete subterms in a formula with fresh variables and attempts to prove its validity from scratch. If successfully proved, such a lemma can then be used to help prove the original subgoal.

However, generalization can discover only a limited number of lemmas. Therefore, we also perform a *SyGuS-based lemma enumeration* driven by *templates*, i.e., formulas with unknowns potentially provided by the user. Our key contribution is an algorithm that instantiates these unknowns with terms generated from syntactic elements obtained automatically from the formulas encountered during the proof search. Thus, the formal grammars, provided as input to SyGuS in our case are automatically generated, goal-directed, and in many cases small. Furthermore, we contribute a set of built-in grammar templates and techniques to effectively filter out invalid formulas produced by the enumeration.

We have implemented our approach in an open-source tool called AdtInd, including the inductive reasoning module and the rewriter. We have evaluated the tool on challenging problems with ADTs and have demonstrated that AdtInd can successfully solve many of these problems on which CVC4 [30] failed, by discovering supporting lemmas through SyGuS-based lemma enumeration. As a sanity check for our tool, we have verified the validity of lemmas synthesized by AdtInd by using CVC4. We also provided our synthesized lemmas as axioms to CVC4, which can then succeed often in proving the original goal. This demonstrates the effectiveness of our lemma synthesis techniques in generating lemmas that can be used in other solvers and different environments, not just in combination with a rewriter that we have used here to implement our ideas.

In summary, this paper makes the following contributions:

- an algorithm for automating proofs by induction over ADTs, where lemmas are synthesized by term enumeration guided by user-specified templates.
- an optimized enumeration process to propose lemma candidates, taking into account the formulas encountered during the proof search. This process includes filtering to reduce overhead of considering candidates that are invalid, or lemmas that may be valid but are less likely to be useful in a proof. These lemma synthesis techniques could be potentially integrated with other solvers.
- an implementation of our lemma synthesis procedures along with an inductive reasoning module and a rewriter that work together as a theorem prover AdtInd. We demonstrate its effectiveness in handling many challenging problems that cannot be solved by a state-of-the-art automated solver.

2 Preliminaries

A many-sorted first-order theory is defined as a tuple $\langle \mathcal{S}, \mathcal{F}, \mathcal{P} \rangle$, where $\mathcal{S}$ is a set of sorts, $\mathcal{F}$ is a set of function symbols, and $\mathcal{P}$ is a set of predicate symbols, including equality. A formula φ is called satisfiable if there exists a model where φ evaluates to *true*. If every model of φ is also a model of ψ, then we write $\varphi \implies \psi$. A formula φ is called *valid* if $true \implies \varphi$.

An algebraic data type (ADT) is a tuple $\langle s, C \rangle$, where $s \in \mathcal{S}$ is a sort and C is a set of uninterpreted functions (called *constructors*), such that each $c \in C$ has some type $A \to s$. If for some s, A is s-free, we say that c is a *base* constructor (otherwise, an *inductive* constructor).

In this paper, we consider universally-quantified formulas over ADTs and uninterpreted functions. For proving validity of a formula $\forall x.\varphi(x)$, where variable x has sort s, we follow the well-known principle of *structural induction*:

Lemma 1. *Given an ADT* $\langle s, \{bc_s : s, ic_s : \underbrace{s \times \ldots \times s}_{n} \to s\} \rangle$ *and a formula* φ*, if the following two formulas (*base case *and* inductive case*) are valid:*

$$\varphi(bc_s) \qquad \textit{and} \qquad \forall x_1, \ldots, x_n. \Big(\bigwedge_{1 \le i \le n} \varphi(x_i) \Big) \implies \varphi(ic_s(x_1, \ldots, x_n))$$

then $\forall x.\varphi(x)$ *is valid.*

Lemma 1 is easily generalizable for ADTs with other constructor types. For instance, an inductive constructor *cons* of a single-linked list has an arity two (i.e., it takes an additional integer i as argument). Thus, to prove the inductive step, the validity of the following formula should be determined:

$$\forall x.\varphi(x) \implies \forall i.\varphi(cons(i,x))$$

We are interested in determining the validity of a universally-quantified formula $\forall x.\varphi(x)$, where φ may itself consist of universally quantified formulas:

$$\forall x.\Big(\forall y.\psi(y)\Big) \wedge \ldots \wedge \Big(\forall z.\gamma(z)\Big) \implies \theta(x) \tag{2.1}$$

We call the innermost universally-quantified formulas on the left side of the implication (2.1) *assumptions*. An assumption is called an *axiom* if it is not implied by any combination of other assumptions, and a *lemma* otherwise. We also assume that neither axioms nor lemmas have appearances of the variable x. The formula on the right side of the implication, which is the only formula over x, is called a *goal*.

A proof of a valid formula of the form (2.1) is derived by structural induction. In both the base and inductive cases, quantifier-free instances $Q(x)$ of axioms and lemmas are produced and used to (sequentially) rewrite the goal until it is simplified to *true*. In particular, we consider the following two simple proof rules (other rules on inequalities and user-proved predicates could also be added):

$$\frac{Q(x) \implies goal(x)}{true}\ [\text{apply}] \qquad \frac{Q(x) \equiv (P(x) = R(x))}{goal[P \mapsto R](x)}\ [\text{rewrite}]$$

If a (possibly transformed) goal cannot be further rewritten or simplified by the given axioms or lemmas, it is called a *failure* formula (if clear from the context, we drop "formula" and simply call it a failure). Clearly, when a goal is supplied by a larger number of assumptions, there is a wider room for possible simplifications, and thus a prover has more chances to succeed. We thus contribute a method that discovers new assumptions and enlarges the search space. An important condition for soundness of this method is that such newly introduced lemmas should themselves be derivable from given assumptions.

3 Motivating Example

Consider an ADT *Queue* defined as a tuple of two lists (the first one being the front, and the second one being the back but in reverse): $queue : list \times list \to Queue$. And some useful functions include: *concat* which concatenates two lists together, *len* which computes the length of a list, *qlen* which computes the length of a queue, *qpush* which appends one element to a queue, and finally *amrt* that balances the queue by concatenating the two lists together when the second list becomes longer than the first one. The given axioms and goal are shown below.

Axioms: definition of $concat, len, qlen, qpush, amrt$

$$\forall l.\, concat(nil, l) = l$$
$$\forall l_1, l_2, n.\, concat(cons(n, l_1), l_2) = cons(n, concat(l_1, l_2)) \tag{3.1}$$

$$\forall l.\, len(nil) = 0$$
$$\forall l, n.\, len(cons(n, l)) = 1 + len(l) \tag{3.2}$$

$$\forall l_1, l_2.\, qlen(queue(l_1, l_2)) = len(l_1) + len(l_2) \tag{3.3}$$

$$\forall l_1, l_2, n.\, qpush(queue(l_1, l_2), n) = amrt(l_1, cons(n, l_2)) \tag{3.4}$$

$$\forall l_1, l_2. amrt(l_1, l_2) = \begin{cases} queue(l_1, l_2) & \text{if } len(l_1) \geq len(l_2) \\ queue(concat(l_1, rev(l_2)), nil) & \text{Otherwise} \end{cases} \tag{3.5}$$

Goal: prove that length of queue increases by 1 after $qpush$

$$\forall l_1, l_2.\, qlen(qpush(queue(l_1, l_2), n)) = 1 + qlen(queue(l_1, l_2)) \tag{3.6}$$

Our approach performs several rewriting steps of applying function definitions, then the base case of induction on variable l_1 leads to the following formula:

$$\forall l_2, n.\, 1 + len(l_2) = len(concat(rev(l_2), cons(n, nil))) \tag{3.7}$$

The formula (3.7) cannot be further simplified with existing axioms, and this constitutes a failure. Before moving on to the inductive case for l_1, we first try to apply generalization as follows:

- We could replace nil with new list variable l_3 on the right hand side (RHS), but there is no place to introduce l_3 on the left hand side (LHS).
- We could also replace $cons(n, nil)$ with new list variable l_3, again there is no corresponding replacement on the LHS.
- Function applications are possible candidates as well, e.g., replacing $rev(l_2)$ with new list variable l_3, yet again it cannot be applied on the LHS.

As seen above, generalization does not give us any suitable candidates. However, useful lemmas could still be generated from ingredients occurring in the failure. In this case, we apply the last two generalization rules (shown above) to the RHS of (3.7), and then automatically construct a formal grammar using one of the user-provided templates (to be explained in detail in Section 4.2). In particular, the last line in (3.8) below is regarded as a template, where the undefined symbols (shown as $\langle ??? \rangle$) have to be filled automatically with suitable terms. This is done by enumeration of integer-typed terms with the function symbols len, $concat$ (as well as constructors nil and $cons$), the variables l_3, l_4, and integer constants such as 0, 1, etc.

$$
\begin{aligned}
\forall l_2, n.\ 1 + len(l_2) &= len(concat(rev(l_2), cons(n, nil))) \\
&\downarrow \text{replace } rev \text{ and } cons \text{ with new variables} \\
\forall l_2, n.\ 1 + len(l_2) &= len(concat(l_3, l_4)) \\
&\downarrow \text{create template for term enumeration} \\
\forall l_3, l_4.\ \langle ??? \rangle + \langle ??? \rangle &= len(concat(l_3, l_4))
\end{aligned}
\tag{3.8}
$$

For instance, our approach (explained in detail in Section 4) is able to discover the following lemma:

$$\forall l_3, l_4.\ len(concat(l_3, l_4)) = len(l_3) + len(l_4) \tag{3.9}$$

The lemma is proven valid by induction (i.e., a recursive invocation of our method), and then it can be used to prove the original goal.

4 Lemma Synthesis

In this section, we describe our key contributions on automated lemma synthesis. Algorithm 1 shows the top level procedure SolveWithInduction which applies structural induction to create the base-case and inductive-step subgoals for the rewriter to prove. If any subgoal cannot be proved with existing assumptions, the algorithm invokes Generalize and EnumerateLemmas to produce lemma candidates based on failures found in Rewrite. These procedures are further described in the following sections.

- Rewrite: a backtracking engine that attempts to rewrite a given goal towards *true*, using the provided assumptions (including discovered lemmas, as described in Section 2). For practical reasons, our implementation uses maximum limits on the depth of the recursive proof search and on the number of rewriting attempts using the same transformation. When a subgoal is not proved, the main output of this engine is a set of *failures*, i.e., formulas obtained during the search, to which no further rewriting rule can be applied (within the given limits). Our algorithm can utilize an external library of proven theorems while a set of heuristics must be developed to efficiently traverse a large search space, which is outside the scope of this work.
- Generalize: an engine, further described in Section 4.1, which takes the failures discovered by Rewrite and applies transformations to replace concrete values by universally quantified variables in order to produce lemma candidates that may support proving the original goal.
- EnumerateLemmas: a SyGuS-based lemma synthesis engine, further described in Section 4.2, which proposes a larger variety of lemma candidates than generalization. This incorporates more aggressive *mutation* of failures than generalization, to make the lemmas goal-oriented. This method is configurable by the choice of grammars, which can be guided by the user. In our implementation, we include grammars tailored to the most common applications appearing in practice in our benchmark examples.

For simplicity of the presentation, our pseudo-code in Algorithm 1 assumes only one quantified ADT variable, which is used to generate the base-case and

Algorithm 1: SolveWithInduction(*Goal*, *Assumptions*)

Input: *Goal*: quantified formula to be proved, *Assumptions*: set of formulas
Output: *result* $\in$ {QED, Unknown}

```
for subgoal ∈ {baseCase(goal), indStep(goal)} do
    if indStep then Assumptions ← Assumptions ∪ {indHypo}
    result, failures ← rewrite(subgoal, Assumptions)
    if result then continue
    candidates ← map(Generalize, failures) ∪ EnumerateLemmas(failures)
    for each ψ ∈ candidates do
        if SolveWithInduction(ψ, Assumptions) = QED then
            result ← SolveWithInduction(subgoal, Assumptions ∪ {ψ})
            if result = QED then break
    if baseCase and result = Unknown then return Unknown
return result
```

inductive-step subgoals. However, our implementation also supports multiple quantifiers and nested induction. For both the subgoals, the proving strategy is to find a sequence of rewriting attempts using the set of assumptions. For the inductive case, the inductive hypotheses are also included in the set of assumptions. In the case of nested induction (omitted from the pseudo-code), all assumptions from the outer-induction are inherited by the inner-induction.

If the algorithm falls short in rewriting any of the subgoals using the existing assumptions, it attempts to synthesize new lemmas by 1) applying Generalize to failures, and 2) identifying suitable terms from failures for applying SyGuS (inside EnumerateLemmas). Generated this way, a lemma candidate needs to be checked for validity which is performed by calling Algorithm 1 recursively.

4.1 Lemma Synthesis by Generalization

The approach of generalizing a failure is widely applied among induction solvers such as IsaPlanner [11], ACL2 [6], and Zeno [31], based on the observation that proving a formula that applies to some specific value is often more difficult than proving a more general version. In our setting, we replace suitable subterms of the formula with fresh quantified variables, effectively weakening the formula.

Algorithm 2 shows the pseudocode of our generalization procedure that, given a formula φ, outputs a lemma candidate ψ. It starts by gathering common subterms in φ (e.g., when φ is an equality, it is possible that the same terms occur on both its sides). Then, it replaces occurrences of subterms by fresh variables and universally quantifies them. In our implementation, we prefer to generalize applications of inductive constructors first. If no lemma was discovered, we proceed to generalizing uninterpreted functions, and our last choice is to generalize base constructors.

Algorithm 2: GENERALIZE(φ)

Input: φ: formula to be generalized
Output: ψ: generalized formula

```
while ∃t ∈ terms(φ), which occurs in φ twice do
    let v be such that v ∉ vars(φ)
    ψ ← ∀v.φ[t ↦ v]
return ψ
```

Algorithm 3: ENUMERATELEMMAS($Failures$)

Input: $Failures$ in the proof search
Output: $Candidates$ formulas

```
Φ ← terms(Failures)
while |Candidates| < THRESHOLD do
    φ ← LARGEST(Φ)
    G ← CREATEGRAMMAR(functions(φ), predicates(φ), vars(φ))
    for each ψ ∈ G do
        ψ ← ∀vars(φ) . ψ
        if ¬REFUTED(ψ) then Candidates ← Candidates ∪ {ψ}
    Φ ← Φ \ {φ}
return Candidates
```

4.2 SyGuS-based Lemma Synthesis

Applying generalization alone may not yield the desired supporting lemma at times. Algorithm 3 shows our SyGuS-style approach for synthesis of lemma candidates from formal grammars. These formal grammars are themselves generated on-the-fly by our procedure. Specifically, in each iteration of the outer loop, the algorithm picks a term which occurs in some failure, and then uses its parse tree to extract function and predicate symbols to construct a formal grammar. This grammar is then used to generate the desired candidate lemmas automatically. Our key contribution is the grammar construction algorithm (outlined in Section 4.3) that uses these function and predicate symbols in combination with user-provided templates. We also provide a set of built-in templates that have worked well on our practical benchmarks.

Finally, in Section 4.4, we describe how to enumerate lemma candidates (up to a certain size) from the grammar, and how to filter likely successful candidates for the original proof goal in Algorithm 1. These candidates must be proven correct first, as shown in Section 4.5.

4.3 Automatic Construction of Grammars

Although our algorithm does not depend on any particular grammar for lemma generation, it is practically important to consider grammars that are relevant for the failures (one or many), so that the generated lemmas have a higher likelihood of success in proving the original goal. Therefore, we focus on various elements (e.g., uninterpreted functions and predicates) that can be extracted from the

parse trees of failures, to automate the process of grammar creation. Elements that do not appear in the failure are not considered to save efforts.

At the same time, a user might specify some *higher-level templates* that provide additional guidance for this process. Essentially, a higher-level template provided by a user can be viewed as a *partially defined grammar* that involves a set of *undefined nonterminals* (i.e., where the corresponding rules are still undefined). Our algorithm automatically constructs missing rules for these nonterminals by using the syntactic patterns obtained from failures, thus constructing fully-defined grammars. These fully-defined grammars are then used for automatically generating candidate lemmas.

To additionally optimize this process, our grammar construction algorithm focuses on individual subterms occurring in failures. Our particular strategy is to pick the largest subterm (referred to as φ in the pseudo-code and later in the text), but other heuristics could be used here as well.

Furthermore, we identified three useful higher-level templates that have been applied to solve our benchmarks[1]. These templates are in the form of an equality, they use undefined nonterminals (shown as $\langle ??? \rangle$), and interestingly, two of them have occurrences of φ on the left side of the equality:

$$\varphi = \langle ??? \rangle + \langle ??? \rangle \tag{4.1}$$

$$\varphi = \langle ??? \rangle \tag{4.2}$$

$$\langle ??? \rangle = \langle ??? \rangle \tag{4.3}$$

The first template is chosen when φ has an integer type, and the second one is chosen for all algebraic data types. Lemma candidates generated from the first two templates inherit information from the failure, having the subterm φ on one side. The third template is chosen as a last resort, when no valid lemmas are discovered after using the first two (as explained in Sections 4.4 and 4.5).

After choosing one of these templates, our algorithm defines the rules for nonterminals $\langle ??? \rangle$, based on the variables, uninterpreted functions, and predicates occurring in φ.

Additionally, we identified two higher-level templates, applicable when the same function occurs in a failure multiple times. Intuitively, they correspond to the commutativity and the associativity of certain uninterpreted functions. After such functions are determined by a syntactic analysis of a failure, they immediately give instantiations of nonterminals $\langle ??? \rangle$ in templates (4.4) and (4.5).

$$\langle ??? \rangle(a, b) = \langle ??? \rangle(b, a) \tag{4.4}$$

$$\langle ??? \rangle(a, \langle ??? \rangle(b, c)) = \langle ??? \rangle(\langle ??? \rangle(a, b), c) \tag{4.5}$$

Returning back to our motivating example, for failure (3.7), both sides of the equality have integer type. Thus, our algorithm chooses template (4.1), and we use the right side of (3.7) as φ, since it is the larger (more complex) expression.

[1] These templates are referred to as *built-in* templates, which need not be specified by the user. Furthermore, our current implementation automatically chooses a built-in template based on φ.

This allows more information from the failure to be retained, thereby enabling the enumerated lemma candidates to be goal-directed. The following grammar is then automatically extracted from φ:

$$\begin{aligned} \langle \textbf{int-term} \rangle &::= n \mid len(\langle \textbf{list-term} \rangle) \\ \langle \textbf{list-term} \rangle &::= nil \mid l_2 \mid cons(\langle \textbf{int-term} \rangle, \langle \textbf{list-term} \rangle) \mid \\ &\quad concat(\langle \textbf{list-term} \rangle, \langle \textbf{list-term} \rangle) \mid rev(\langle \textbf{list-term} \rangle) \end{aligned} \tag{4.6}$$

Note that this grammar is recursive and relatively large in scope. For performance reasons, we try to reduce the grammar. We do this by heuristically generalizing φ first, where we replace function applications by fresh variables (i.e., similar to the strategy in Section 4.1), as shown in (3.8). The generalized φ gives rise to the following grammar:

$$\begin{aligned} \langle \textbf{int-term} \rangle &::= len(\langle \textbf{list-term} \rangle) \\ \langle \textbf{list-term} \rangle &::= l_3 \mid l_4 \mid concat(\langle \textbf{list-term} \rangle, \langle \textbf{list-term} \rangle) \end{aligned} \tag{4.7}$$

Finally, the resulting production rules are embedded into the chosen template to generate a complete grammar, where $\langle ??? \rangle$ is instantiated by $\langle \textbf{int-term} \rangle$.

4.4 Producing Terms from Grammar

Given a grammar, constructed as shown in the previous subsection, our algorithm enumerates various candidate lemmas and checks their validity. Since larger candidate lemmas are typically more expensive to deal with, our algorithm starts by enumerating small formulas with terms upto some size. We define the *size* of an expression as the height of its parse tree. For example, variables and base constructors of data types, such as x, y, nil have size 1, while $cons(1, nil)$ and $rev(x)$ have size 2. By Ψ_k, we denote the set of expressions of size k, and by $\Psi_k[ty]$, we denote the set of expressions of size k that has type ty.

Given Ψ_k, it is straight-forward to enumerate expressions of size $k+1$: for each function (including inductive constructors) f with m parameters typed $ty_1, \ldots, ty_m$, we first enumerate expressions $\tau_1, \ldots, \tau_m$ from sets $\Psi_k[ty_1], \ldots \Psi_k[ty_m]$, respectively, and second, we create a new expression $f(\tau_1, \ldots, \tau_m)$ which is inserted into Ψ_{k+1}. This process is repeated iteratively until we reach the desired size limit.

Our algorithm enforces the following two constraints on the generated candidate formulas. First, it checks the generated formula for *non-triviality*: there should be no application of a function on only base constructors of an ADT, e.g., $concat(nil, nil)$. Such candidates are usually invalid for any non-trivial instantiation of a universally quantified variable. Second, a generated lemma candidate should cover as many variables occurring in the subterm φ from which we derived the templates (4.1) – (4.3) as possible. In our experience, prioritizing candidates with full coverage leads to significant performance gains.

To further reduce the number of candidates, we leverage symmetry of operators in a template (e.g., commutativity of integer addition) whenever possible.

4.5 Filtering by Refutation

We apply an additional filtering step on lemma candidates where we search for inexpensive counterexamples to validity. Given a candidate lemma, our algorithm instantiates quantified variables with concrete values, creates quantifier-free expressions, and repeatedly simplifies them by applying assumptions. In addition to the rules mentioned in Section 2, we also apply the following *refutation* rule:

$$\frac{goal(x) \implies false}{false}\ [\text{apply}^r]$$

In our implementation, we limit the number of refutation attempts for each candidate and the complexity of the concrete instantiations. The concrete values of variables are produced by applying constructors of ADTs repeatedly.

For example, formula (4.8) is one of the possible candidates based on the template in (3.8). This lemma is shown invalid by instantiating l_3 and l_4 with concrete lists $cons(1, cons(2, nil))$ and $cons(3, nil)$, and then applying the given axioms to the resulting quantifier-free expression, as shown below.

$$\begin{aligned}
&\forall l_3, l_4.\ len(cons(len(l_3), nil)) + len(l_4) = len(concat(l_3, l_4)) \\
&\qquad \downarrow \texttt{instantiate quantified variables} \\
&len(cons(len(cons(1, cons(2, nil))), nil)) + len(cons(3, nil)) \\
&= len(concat(cons(1, cons(2, nil)), cons(3, nil))) \\
&\qquad \downarrow \texttt{apply axiom}\ (3.2) \\
&1 + 1 = len(concat(cons(1, cons(2, nil)), cons(3, nil))) \\
&\qquad \downarrow \texttt{apply axiom}\ (3.1) \\
&1 + 1 = len(cons(1, cons(2, cons(3, nil)))) \\
&\qquad \downarrow \texttt{apply axiom}\ (3.2) \\
&1 + 1 = 1 + 1 + 1\ (\texttt{False})
\end{aligned} \tag{4.8}$$

If a lemma candidate passes (some number of) refutation tests, then a new instance of SolveWithInduction is created in an attempt to prove its validity. This recursive nature of our procedure allows proving lemma candidates that may further require discovering new supporting lemmas. However, creating a subgoal to prove a lemma candidate is a fairly expensive procedure. Therefore, we would like the filtering to be aggressive, to minimize the number of lemmas to be proved. Although the refutation tests are relatively cheap to perform, too many tests may result in wasted effort and delay lemma application in proving the original goal. Thus, we must strike a balance between testing for refutations and proof attempts. In our implementation, we perform three refutation tests by default (and the user can optionally set the number of such tests).

5 Implementation and Evaluation

We have implemented our algorithm in a prototype tool named AdtInd on top of Z3 [26]. Our backtracking rewrite procedure uses the "apply" and "rewrite"

proof rules repeatedly to simplify the goals and invokes Z3 to determine the validity of quantifier-free expressions encountered during such rewriting. Our implementation allows the user to specify the maximal depth of the backtracking search (15 steps by default); it also avoids divergence by limiting the consecutive applications of the same rewrite rules.

Our lemma synthesis procedures are also configurable. In GENERALIZE, the user can adjust the aggressiveness of generalization, opting to replace smaller or larger terms in failures (recall Section 4.1). In EnumerateLemmas, the user sets a larger limit on sizes of enumerated terms to explore a larger space of lemma candidates. The number of refutation attempts is also configurable (3 times by default).

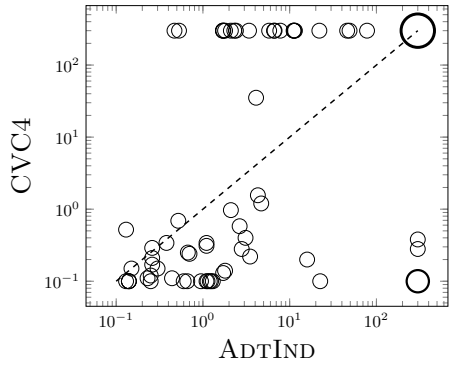

Fig. 2. Evaluation comparison (sec×sec): points above the diagonal represent runtimes for benchmarks on which AdtInd outperformed CVC4; points on the boundaries represent timeouts. The diameter of a circle represents the number of overlapping circles.

AdtInd has been evaluated on benchmarks from the CLAM [20] suite[2] consisting of 86 quantified theorems over common operations of natural numbers, lists and other data types. We have compared AdtInd against the CVC4 SMT solver (v1.7, which supports induction and subgoal generation). We used a timeout of 300 seconds. The scatter plot in Figure 2 summarizes the results. In total, AdtInd proved 62, and CVC4 proved 47 benchmarks. These numbers include respectively 22 and 7 theorems proven only by the corresponding tool and not by the other (shown in Figure 2 as crosses on the top horizontal line and as crosses clustered around the bottom right corner, respectively). Interestingly, there are not many cases when CVC4 takes a significant amount of time before delivering a successful result, i.e., it either terminates in less than a second or diverges. This is possibly due to an inability to discover a meaningful lemma candidate for these benchmarks. In contrast, AdtInd is often able to enumerate useful lem-

[2] The source code of AdtInd and benchmarks are available at: `github.com/wky/aeval/tree/adt-ind`.

mas, but sometimes it requires a number of iterations (see, e.g., crosses on the bottom horizontal line). We hope to improve runtime performance of AdtInd in the future by adopting certain successful optimizations and heuristics from CVC4.

Of the 62 theorems proven by AdtInd, 29 did not require extra lemmas, 12 were proven with lemmas discovered through generalization, and 21 were proven with lemmas discovered through SyGuS. For the SyGuS-generated 21 theorems, AdtInd created on average 171 lemma candidates. In our experiments, 93% of processed candidates were refuted by tests, leaving only a small number of lemmas to be validated by the more expensive SolveWithInduction.

Experiment over ADTs and LIA. To fully demonstrate the power of SMT solvers, we considered several additional benchmarks involving linear integer arithmetic (LIA). With this capability, the benchmarks do not require specifying assumptions over natural numbers (like in CLAM). These benchmarks also motivate the usefulness of having a specialized lemma template for integers shown in Section 4.3. The results are listed in Table 1. The `list_rev2` benchmark took more solving time than others due to the large search space (about 500 lemma candidates were rejected before a sufficient lemma was found). For comparison, CVC4 failed to prove `list_rev` and exceeded a timeout of 300 seconds on the other 8 problems. The interactive prover ACL2 was only able to prove only 2 out of 9 problems in Table 1, namely `list_rev_concat` and `list_rev_len`.

Table 1. AdtInd on ADT+LIA problems.

Goal	**AdtInd result**	**Goal**	**AdtInd result**
`list_rev`	Proved, 10.6s	`list_rev2_len`	Proved, 0.95s
`list_rev_concat`	Proved, 2.52s	`queue_push`	Proved, 33.9s
`list_rev2_concat`	Proved, 1.95s	`queue_len`	Proved, 7.6s
`list_rev2`	Proved, 1m59s	`tree_insert_all`	Proved, 1.9s
`list_rev_len`	Proved, 2.30s		

Future Work. There are other categories of theorems, mainly in the Leon [4] and TIP [7] suites, that require more advanced techniques for automated proving. Many theorems require non-trivial case-splitting transformations (some implemented in Why3 [15]) for *if-then-else* blocks in given axioms, which is not fully supported in our prototype yet. However, our lemma synthesis algorithms can be used in combination with a different solver or environment that handles case-splitting and other forms of goal decomposition. Thus, our tool can focus on producing lemma candidates without being dependent on current capabilities of our prototype rewriting engine.

Of the theorems that we cannot prove in the TIP set, many are mathematically challenging (e.g., Fermat's Last Theorem), involve high-order functions, contain sortedness properties, or require some form of pumping lemma to solve

(e.g., proving equivalence of regular languages). These instances are currently outside the scope of our work.

6 Related Work

There is a wide range of approaches for proving quantified theorems defined on algebraic data types. These include SMT-based inductive reasoning in tools such as Dafny [24] and CVC4 [30]; Horn Clause solvers [33]; generic theorem provers such as ACL2 [6], and induction provers such as CLAM [20], IsaPlanner [22] and HipSpec [8]. The main issue with these tools is that even with the help of built-in heuristics such as rippling [5] and generalization of failures, they still require human interaction to discover necessary lemmas to complete a proof end-to-end. Our proposal to use term enumeration for lemma discovery (after failure of generalization) as a SyGuS-style synthesis task leverages information available at proof failures and explores a much larger space of possible lemma candidates. As shown in our evaluation in Section 5, this was enough to eliminate the need for human input in many practical cases.

On lemma discovery within induction provers, machine learning techniques have also been attempted in works such as ACL2(ml) [19] and Multi-Waterfall [21]. ACL2(ml) uses statistical machine learning algorithms to extract features present in the proof goal, and uses that to find similar patterns in a library of proven theorems to in order to suggest new lemmas. Multi-Waterfall runs multiple strategies in parallel, while a machine learning module trained by previous proofs in a library is used to select lemmas candidates based on their likelihood of advancing the current proof. The machine learning components in these tools typically require a sufficiently large set of proven theroems to learn from, whereas our tool uses term enumeration that does not depend on an external library.

Specifically, CVC4 [30] supports induction natively to solve quantified SMT queries with custom data types. The tool implements Skolemization with inductive strengthening to prove conjectures, and uses enumeration to find adequate subgoals (inspired by QuickSpec [9]). CVC4 employs filtering of candidates based on activation of function symbols, canonicity of terms and counterexamples, which is roughly analogous to our filtering techniques. However, our lemma candidates arise from grammars that combine user-provided (or built-in) templates with elements from failures in rewriting proof attempts and seem to have a better chance of proving the original goal.

ACL2 (a Boyer-Moore prover) is based on rewriting of terms and a number of induction heuristics. The tool identifies "key checkpoints" as subgoals to prove on its way to prove the outer theorem, and has rules to perform generalization similar to our approach described in Section 4.1. However, ACL2 does not have the ability to enumerate lemma candidates, although users can provide their own proof tactics or plug-ins to this theorem prover.

On the lemma synthesis front, the SLS framework [32] employs different techniques to automatically generate and validate lemmas, but within an interactive theorem prover environment. For symbolic heap verification using separation

logic, the tool generates lemma templates with the heap structures from the goal entailment, and proposes unknown relations as constraints over the templates' variables, which are later solved to discover the desired lemmas. We were unable to experimentally compare with SLS because it works in an interactive theorem prover environment, targets a distinct type of problems (proof entailments in separation logic) and requires a different input format which is prohibitive for us to translate to.

Among SyGuS applications for solving quantified formulas, in another recent effort with CVC4 [28], a user can provide a grammar and a correctness specification to a synthesis task, whose goal is to find *rewrite rules* that transform and simplify SMT queries. The similarity here is that our tool also uses a SyGuS-style user-provided template to search for supporting lemmas, which will be used just like rewrite rules. However, the purpose of their technique is primarily goal-agnostic simplification, and it does not track information such as failures in proof search. More importantly, their grammars are not generated automatically from problem instances, but are fixed by the user. Another recent effort [27] uses SyGuS to synthesize invertibility conditions under which quantified bit-vector problems can be converted to quantifier-free problems, to be solved by an SMT solver. However, the purpose and specific techniques are different from our approach.

Finally, SyGuS was recently applied to verification of program safety and termination in the FreqHorn framework [12, 13]. These works exploit the syntax of given programs to automatically generate grammar, from which the candidates for inductive invariants and ranking functions are produced. While their main insight is similar to ours, their approach does not support ADTs and hardly exploits any failures. In the future, we believe that our tool could be integrated to FreqHorn and help verify programs which are currently out of its scope.

7 Conclusions and Future Work

We have presented a new approach for automating induction over algebraic datatypes that uses lemma synthesis based on automatic grammar generation and term enumeration guided by user-specified templates. Our prover AdtInd incorporates these ideas in a rewriting engine built on top of Z3. We demonstrated that it successfully solves many challenging problem instances that a state-of-the-art prover failed to solve.

So far, the proof goals in the examples that we considered (i.e., *List*, *Queue*, *Tree*) are mostly in the form of equalities. We intend to apply our ideas to support inequalities and other relations that demand non-trivial inductive reasoning and lemma discovery. Incorporating our lemma synthesis procedures into other theorem proving frameworks (such as CVC4) would allow us to leverage existing heuristics and proof tactics to deliver results on more complex problems. Also we will consider additional criteria for usefulness of lemma candidates to better filter the large number of candidates in certain benchmarks.

Acknowledgments. This work is supported in part by NSF Grant 1525936.

References

1. Alur, R., Bodík, R., Juniwal, G., Martin, M.M.K., Raghothaman, M., Seshia, S.A., Singh, R., Solar-Lezama, A., Torlak, E., Udupa, A.: Syntax-guided synthesis. In: FMCAD. pp. 1–17. IEEE (2013)
2. Alur, R., Fisman, D., Singh, R., Solar-Lezama, A.: Sygus-comp'17: Results and analysis (2017), `http://sygus.seas.upenn.edu/`
3. Barrett, C., Conway, C.L., Deters, M., Hadarean, L., Jovanovic, D., King, T., Reynolds, A., Tinelli, C.: CVC4. In: Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings. pp. 171–177 (2011)
4. Blanc, R., Kuncak, V., Kneuss, E., Suter, P.: An overview of the leon verification system: Verification by translation to recursive functions. In: Proceedings of the 4th Workshop on Scala. pp. 1:1–1:10. SCALA '13, ACM, New York, NY, USA (2013). https://doi.org/10.1145/2489837.2489838
5. Bundy, A., Stevens, A., van Harmelen, F., Ireland, A., Smaill, A.: Rippling: A heuristic for guiding inductive proofs. Artificial Intelligence **62**(2), 185 – 253 (1993)
6. Chamarthi, H.R., Dillinger, P., Manolios, P., Vroon, D.: The acl2 sedan theorem proving system. In: Proceedings of the 17th International Conference on Tools and Algorithms for the Construction and Analysis of Systems: Part of the Joint European Conferences on Theory and Practice of Software. pp. 291–295. TACAS'11/ETAPS'11, Springer-Verlag, Berlin, Heidelberg (2011)
7. Claessen, K., Johansson, M., Rosén, D., Smallbone, N.: Tip: Tons of inductive problems. In: Kerber, M., Carette, J., Kaliszyk, C., Rabe, F., Sorge, V. (eds.) Intelligent Computer Mathematics. pp. 333–337. Springer International Publishing, Cham (2015)
8. Claessen, K., Johansson, M., Smallbone, N.: Hipspec: Automating inductive proofs of program properties. In: In Workshop on Automated Theory eXploration: ATX 2012 (2012)
9. Claessen, K., Smallbone, N., Hughes, J.: Quickspec: Guessing formal specifications using testing. In: Proceedings of the 4th International Conference on Tests and Proofs. pp. 6–21. TAP'10, Springer-Verlag, Berlin, Heidelberg (2010)
10. Cruanes, S.: Superposition with structural induction. In: Dixon, C., Finger, M. (eds.) Frontiers of Combining Systems. pp. 172–188. Springer International Publishing, Cham (2017)
11. Dixon, L., Fleuriot, J.: Isaplanner: A prototype proof planner in isabelle. In: Baader, F. (ed.) Automated Deduction – CADE-19. pp. 279–283. Springer Berlin Heidelberg, Berlin, Heidelberg (2003)
12. Fedyukovich, G., Kaufman, S., Bodík, R.: Sampling Invariants from Frequency Distributions. In: FMCAD. pp. 100–107. IEEE (2017)
13. Fedyukovich, G., Zhang, Y., Gupta, A.: Syntax-Guided Termination Analysis. In: CAV, Part I. LNCS, vol. 10981, pp. 124–143. Springer (2018)
14. Feldman, Y.M.Y., Padon, O., Immerman, N., Sagiv, M., Shoham, S.: Bounded quantifier instantiation for checking inductive invariants. In: Legay, A., Margaria, T. (eds.) Tools and Algorithms for the Construction and Analysis of Systems. pp. 76–95. Springer Berlin Heidelberg, Berlin, Heidelberg (2017)
15. Filliâtre, J.C., Paskevich, A.: Why3 — where programs meet provers. In: Felleisen, M., Gardner, P. (eds.) Proceedings of the 22nd European Symposium on Programming. Lecture Notes in Computer Science, vol. 7792, pp. 125–128. Springer (Mar 2013)

16. Garg, P., Löding, C., Madhusudan, P., Neider, D.: ICE: A robust framework for learning invariants. In: CAV. LNCS, vol. 8559, pp. 69–87. Springer (2014)
17. Garg, P., Neider, D., Madhusudan, P., Roth, D.: Learning invariants using decision trees and implication counterexamples. In: POPL. pp. 499–512. ACM (2016)
18. Ge, Y., Barrett, C., Tinelli, C.: Solving quantified verification conditions using satisfiability modulo theories. In: Pfenning, F. (ed.) Automated Deduction – CADE-21. pp. 167–182. Springer Berlin Heidelberg, Berlin, Heidelberg (2007)
19. Heras, J., Komendantskaya, E.: Acl2(ml): Machine-learning for ACL2. In: Proceedings Twelfth International Workshop on the ACL2 Theorem Prover and its Applications, Vienna, Austria, 12-13th July 2014. pp. 61–75 (2014)
20. Ireland, A., Bundy, A.: Productive use of failure in inductive proof. Journal of Automated Reasoning **16**, 79–111 (01 1996)
21. Jiang, Y., Papapanagiotou, P., Fleuriot, J.: Machine learning for inductive theorem proving. In: Fleuriot, J., Wang, D., Calmet, J. (eds.) Artificial Intelligence and Symbolic Computation. pp. 87–103. Springer International Publishing, Cham (2018)
22. Johansson, M., Dixon, L., Bundy, A.: Case-analysis for rippling and inductive proof. In: Kaufmann, M., Paulson, L.C. (eds.) Interactive Theorem Proving. pp. 291–306. Springer Berlin Heidelberg, Berlin, Heidelberg (2010)
23. Kersani, A., Peltier, N.: Combining superposition and induction: A practical realization. In: Fontaine, P., Ringeissen, C., Schmidt, R.A. (eds.) Frontiers of Combining Systems. Springer Berlin Heidelberg, Berlin, Heidelberg (2013)
24. Leino, K.R.M.: Automating induction with an smt solver. In: Proceedings of the 13th International Conference on Verification, Model Checking, and Abstract Interpretation. pp. 315–331. VMCAI'12, Springer-Verlag, Berlin, Heidelberg (2012)
25. de Moura, L.M., Bjørner, N.: Efficient e-matching for SMT solvers. In: Automated Deduction - CADE-21, 21st International Conference on Automated Deduction, Bremen, Germany, July 17-20, 2007, Proceedings. pp. 183–198 (2007)
26. de Moura, L.M., Bjørner, N.: Z3: An Efficient SMT Solver. In: TACAS. LNCS, vol. 4963, pp. 337–340. Springer (2008)
27. Niemetz, A., Preiner, M., Reynolds, A., Barrett, C., Tinelli, C.: Solving quantified bit-vectors using invertibility conditions. In: Chockler, H., Weissenbacher, G. (eds.) Computer Aided Verification. pp. 236–255. Springer International Publishing, Cham (2018)
28. Reynolds, A., Barbosa, H., Tinelli, C., Niemetz, A., Nötzli, A., Preiner, M., Barrett, C.W.: Rewrites for smt solvers using syntax-guided enumeration. In: SMT Workshop (2018)
29. Reynolds, A., Deters, M., Kuncak, V., Tinelli, C., Barrett, C.W.: Counterexample-guided quantifier instantiation for synthesis in SMT. In: CAV. LNCS, vol. 9206, pp. 198–216. Springer (2015)
30. Reynolds, A., Kuncak, V.: Induction for SMT solvers. In: VMCAI. LNCS, vol. 8931, pp. 80–98. Springer (2015)
31. Sonnex, W., Drossopoulou, S., Eisenbach, S.: Zeno: An automated prover for properties of recursive data structures. In: Flanagan, C., König, B. (eds.) Tools and Algorithms for the Construction and Analysis of Systems. pp. 407–421. Springer Berlin Heidelberg, Berlin, Heidelberg (2012)
32. Ta, Q., Le, T.C., Khoo, S., Chin, W.: Automated lemma synthesis in symbolic-heap separation logic. PACMPL **2**(POPL), 9:1–9:29 (2018)
33. Unno, H., Torii, S., Sakamoto, H.: Automating induction for solving horn clauses. In: Majumdar, R., Kunčak, V. (eds.) Computer Aided Verification. pp. 571–591. Springer International Publishing, Cham (2017)