Data-Driven Lemma Synthesis for Interactive Proofs

AISHWARYA SIVARAMAN, University of California, Los Angeles, USA
ALEX SANCHEZ-STERN, UMass Amherst, USA
BRETTON CHEN, University of California, Los Angeles, USA
SORIN LERNER, University of California, San Diego, USA
TODD MILLSTEIN, University of California, Los Angeles, USA

Interactive proofs of theorems often require auxiliary helper lemmas to prove the desired theorem. Existing approaches for automatically synthesizing helper lemmas fall into two broad categories. Some approaches are *goal-directed*, producing lemmas specifically to help a user make progress from a given proof state, but they have *limited expressiveness* in terms of the lemmas that can be produced. Other approaches are *highly expressive*, able to generate arbitrary lemmas from a given grammar, but they are completely *undirected* and hence not amenable to interactive usage.

In this paper, we develop an approach to lemma synthesis that is both goal-directed and expressive. The key novelty is a technique for reducing lemma synthesis to a *data-driven* program synthesis problem, whereby examples for synthesis are generated from the current proof state. We also describe a technique to systematically introduce new variables for lemma synthesis, as well as techniques for filtering and ranking candidate lemmas for presentation to the user. We implement these ideas in a tool called `lfind`, which can be run as a Coq tactic. In an evaluation on four benchmark suites, `lfind` produces useful lemmas in 68% of the cases where a human prover used a lemma to make progress. In these cases `lfind` synthesizes a lemma that either enables a fully automated proof of the original goal or that matches the human-provided lemma.

CCS Concepts: • **Theory of computation → Program verification**; • **Software and its engineering → Software verification and validation**.

Additional Key Words and Phrases: Lemma Synthesis, Data-Driven Synthesis, Type-Directed Synthesis, Interactive Theorem Proving

ACM Reference Format:
Aishwarya Sivaraman, Alex Sanchez-Stern, Bretton Chen, Sorin Lerner, and Todd Millstein. 2022. Data-Driven Lemma Synthesis for Interactive Proofs. *Proc. ACM Program. Lang.* 6, OOPSLA2, Article 143 (October 2022), 27 pages. https://doi.org/10.1145/3563306

1 INTRODUCTION

Interactive proof assistants [de Moura et al. 2015; Filliâtre et al. 1997; Paulson 1993] are powerful frameworks for writing code with strong guarantees. While various tools exist to perform automated proof search [Bansal et al. 2019; First et al. 2020; Gauthier et al. 2017; Paliwal et al. 2020; Sanchez-Stern et al. 2020; Sekiyama et al. 2017; Whalen 2016; Yang and Deng 2019] and to integrate external automated solvers [Blanchette et al. 2011; Czajka and Kaliszyk 2018; Kaliszyk and Urban 2015a,b], the

Authors' addresses: Aishwarya Sivaraman, University of California, Los Angeles, Los Angeles, California, USA, dcssiva@cs.ucla.edu; Alex Sanchez-Stern, UMass Amherst, Amherst, Massachusetts, USA, sanchezstern@cs.umass.edu; Bretton Chen, University of California, Los Angeles, Los Angeles, California, USA, bretton@g.ucla.edu; Sorin Lerner, University of California, San Diego, San Diego, California, USA, lerner@cs.ucsd.edu; Todd Millstein, University of California, Los Angeles, Los Angeles, California, USA, todd@cs.ucla.edu.

2475-1421/2022/10-ART143
https://doi.org/10.1145/3563306

manual proof burden remains high. One particular challenge is the need to identify auxiliary lemmas that are required to prove a desired theorem. For example, the theorem's induction hypothesis may be too weak, thereby necessitating a stronger lemma that is amenable to an inductive proof. As another example, a lemma may be required to rewrite a subgoal at a particular point in the proof into a form that allows the induction hypothesis to be applied.

Existing approaches to address this problem through a form of *lemma synthesis* fall into two categories [Johansson 2019]. In the first category, heuristic rewrites are performed on the proof state at the point where the user is stuck to identify potentially useful lemmas [Aubin 1976; Bundy et al. 1993; Castaing 1985; Dixon and Fleuriot 2003; Hesketh 1992; Hummel 1990; Johansson et al. 2010; Kapur and Subramaniam 1996; Kaufmann and Moore 1997; Sonnex et al. 2012]. For example, the *generalization* technique [Boyer and Moore 1979; Kaufmann and Moore 1997] from ACL2 heuristically replaces one or more terms in the current subgoal with fresh variables. In the second category of approaches, candidate lemmas are generated from a grammar through a form of enumeration-based synthesis [Claessen et al. 2013, 2010; Heras et al. 2013; Johansson et al. 2011, 2014; Montaño-Rivas et al. 2012; Reynolds and Kuncak 2015; Yang et al. 2019]. For example, HipSpec [Claessen et al. 2013] uses QuickSpec [Claessen et al. 2010] to generate many candidate equational lemmas and then proves as many as possible using an automated prover.

The strength of the heuristic rewriting approach is that it is *goal-directed*, producing candidate lemmas that are directly related to the current proof state. However, the approach has *limited expressiveness*, as the space of possible candidates is dependent on a particular set of rewrite rules. The enumeration approach has the opposite strengths and weaknesses. Because candidate lemmas are enumerated from a grammar, they can be *highly expressive*. However, candidate lemmas are generated in an *undirected* fashion, independent of the particular state where the user is stuck. Hence this approach will generate many irrelevant lemmas and so is ill-suited for an interactive setting. Indeed none of the enumeration-based tools cited above support interactive usage.

In this paper, we propose a new approach to lemma synthesis that combines the strengths of the existing approaches. We show how to reduce lemma synthesis to a *data-driven program synthesis* problem, which aims to synthesize an expression that meets a given set of input-output examples. The examples for synthesis are generated directly from the current proof state, ensuring that lemma candidates are targeted at the goal. At the same time, the approach enables the usage of off-the-shelf data-driven program synthesizers that generate expressions in a user-provided grammar [Albarghouthi et al. 2013; Feser et al. 2015; Frankle et al. 2016; Lubin et al. 2020; Miltner et al. 2022; Osera and Zdancewic 2015]. This new approach allows us to successfully synthesize helper lemmas for more stuck proofs than ever before.

Reducing lemma synthesis to data-driven program synthesis requires us to solve several technical challenges. While data-driven synthesis is a common approach to generating other kinds of program invariants [Ezudheen et al. 2018; Garg et al. 2014, 2016; Miltner et al. 2020; Padhi et al. 2016; Zhu et al. 2018], for instance, loop invariants, these prior settings have several advantages that our setting lacks. In prior settings, the desired invariant is a predicate over a fixed set of variables, for example, the variables that are in scope at a loop. In contrast, it's common for auxiliary lemmas to require new variables that do not appear in the current proof state. Further, prior approaches employ counterexample-guided inductive synthesis (CEGIS) [Solar-Lezama 2009], because there exists a clear behavioral specification for the desired invariant: each candidate invariant is verified against the specification, and counterexamples become new input-output examples for synthesis. In our setting, we lack such a specification since a proof state can require an auxiliary lemma for many different reasons. Further, a counterexample to validity applies to the current lemma candidate, but that same valuation of variables is not necessarily a counterexample to the validity of a different candidate lemma. Hence we cannot generate input-output examples using CEGIS. Finally, the lack of a specification also makes it difficult to determine whether any particular candidate lemma is useful.

To address the problem of lemmas that require variables not appearing in the proof state, we observe that the *generalization* technique [Boyer and Moore 1979; Kaufmann and Moore 1997] described above can be used not only to produce candidate lemmas but also as a systematic way to "lift" the current proof state to new variables for lemma synthesis. Hence our approach starts by producing all generalizations of the proof state, each formed by replacing one or more terms with fresh variables.

To generate examples for synthesis without counterexamples, we leverage the implicit observation underlying the heuristic rewriting approaches described earlier, that the necessary lemma often has a similar structure to the goal in the current proof state. We produce a set of *lemma sketches* for each generalized goal, each sketch consisting of a version of that goal but with one expression replaced by a *hole* to be synthesized. We sample valuations of the variables in the current goal to generate input examples, and the expected output value for each example is determined by the value of the hole's original expression. In this way, we require that the synthesized expression's behavior be consistent with that of the expression that it is replacing.

Finally, to address the lack of clear criteria for candidate lemmas to satisfy, we have developed techniques to *filter* candidate lemmas that are not useful and to *rank* the remaining candidates based on their likely utility to the user. Filtering removes lemmas that are determined to be either trivial, redundant, or invalid, the latter using existing tools for automated counterexample search [Claessen and Hughes 2000; Paraskevopoulou et al. 2015]. Since the ultimate utility of a lemma is based on whether it is provable and allows the user to complete the current proof, our ranking approach employs existing tools for automated proof search to categorize lemmas for user inspection.

Our example-based approach to lemma synthesis is targeted for use in proving properties of programs. In that setting we can leverage existing randomized testing tools to generate the necessary examples. Generating examples for arbitrary Coq propositions and types is difficult in general and an active area of research [Paraskevopoulou et al. 2022]. Therefore our approach is less applicable to other uses of Coq, for example to prove mathematical theorems.

We have implemented our approach as a tactic for Coq and call the resulting tool `lfind`[1]. Coq users can invoke `lfind` as a tactic at any point in their proof, and it will produce a set of ranked lemma candidates. Our approach is parameterized by a data-driven program synthesizer (for candidate lemma generation), counterexample searcher (for candidate filtering), and proof searcher (for candidate ranking). Our implementation uses the Myth [Osera and Zdancewic 2015] data-driven program synthesizer for OCaml, the QuickChick [Paraskevopoulou et al. 2015] tool for counterexample search, and the state-of-the-art Proverbot9001[Sanchez-Stern et al. 2020] tool for proof script search. Note that our approach is *agnostic* to the specific toolset we use for implementation; in fact, future improvements in data-driven program synthesis, counterexample search, and proof search can be directly leveraged to improve lemma synthesis.

We evaluate our approach on two benchmark suites from prior work on lemma synthesis, clam [Ireland and Bundy 1996] and lia [Yang et al. 2019], as well as two new benchmarks from diverse domains, full adder [cir 1995] and compiler correctness [Chlipala 2013]. Together, there are 222 evaluation locations from these benchmarks, where a human prover used an auxiliary lemma to progress. `lfind` synthesizes a useful lemma for 150/222 of these locations, with a median runtime of 3.3 minutes (see §5.3). At 117 of these locations `lfind` provides a full automated proof of the synthesized lemma and the goal; at the other 33 locations `lfind` produces a ranked list of lemma candidates where the human-written lemma is in the top 10. We also show that our

[1] Our code and full experimental results are available at https://github.com/AishwaryaSivaraman/lemmafinder.

```
Inductive lst : Type :=
  | Nil : lst
  | Cons : nat -> lst -> lst.

Fixpoint app (l1 : lst) (l2 : lst) : lst :=
  match l1 with
  | Nil => l2
  | Cons n l1' =>  Cons n (app l1' l2)
  end.

Fixpoint rev (l : lst) : lst :=
  match l with
  | Nil => Nil
  | Cons n l1' => app (rev l1') (Cons n Nil)
  end.

Lemma rev_rev : forall l, rev (rev l) = l.
Proof.
  induction l.
  - reflexivity.
  - simpl. (* I'm stuck! *)
```

Fig. 1. A partial proof of a theorem in Coq that requires an auxiliary lemma.

```
n : nat
l : lst
IHl : rev (rev l) = l
---------------------------------------------
rev (app (rev l) (Cons n Nil)) = Cons n l
```

Fig. 2. The proof state when the user gets stuck.

approach significantly outperforms the prior technique of generalization as well as a version of LIA that employs enumerative synthesis without examples (§5.4). Finally, in §5.5 we evaluate `lfind`'s sensitivity to different hyperparameters and timeouts.

In summary, this paper makes the following contributions:

(1) We present the first approach that reduces the general lemma synthesis problem to a data-driven program synthesis problem. The approach derives both lemma sketches as well as examples for synthesis from a given stuck proof state, and it uses the existing generalization technique to lift the proof state to new variables for synthesis.
(2) We describe a suite of filtering and ranking strategies for candidate lemmas, which is necessary for an interactive verification setting.
(3) We have instantiated our approach in a tactic called `lfind` for Coq.
(4) Our experimental evaluation demonstrates the practical utility of our approach and tool, quantifies the benefits over multiple alternative approaches to lemma synthesis, and investigates the sensitivity of `lfind` to different parameter values.

2 OVERVIEW

2.1 Motivating Example

To illustrate how `lfind` works, we will start with an example. Figure 1 shows Coq code that tries to prove a simple theorem: that reversing a list twice returns the same list. It starts by defining lists

```
Lemma lem1: forall l n,
  rev (app l (Cons n Nil)) = Cons n (rev l).
Proof.
    intros.
    induction l.
    simpl.
    eauto.
    simpl.
    rewrite IHl.
    eauto.
Qed.

Lemma rev_rev : forall l, rev (rev l) = l.
Proof.
  induction l.
  - reflexivity.
  - simpl. rewrite <- IHl. unfold app.
    rewrite IHl. rewrite lem1. rewrite IHl. easy.
Qed.
```

Fig. 3. A full proof provided by lfind.

of nats along with definitions for appending and reversing lists. Following that is an attempt to prove the theorem, named rev_rev.

The proof proceeds by induction on the list l. The Nil case is easily proven, but the Cons case is trickier. After simplification, the user is stuck because the goal is not in a form that enables direct use of the induction hypothesis. Figure 2 shows the proof state at that point, including the current assumptions and goal.

To get unstuck, the user can invoke our tool lfind as a tactic at this point. In this example, the top three lemmas that lfind produces are as follows:

```
(Λ1) Lemma lem1: forall l1 n,
      rev (app l1 (Cons n Nil)) = Cons n (rev l1).
(Λ2) Lemma lem2: forall l1 l2,
      rev (app l1 l2) = app (rev l1) (rev l2).
(Λ2) Lemma lem3: forall l1 l2,
      rev (app (rev l1) l2) = app (rev l2) l1.
```

Each lemma is bucketed into one of three categories (Λ_1, Λ_2, or Λ_3), and the categories are presented to the user in that order. Λ_1 lemmas are those in which lfind can automatically find a complete proof of the original goal using the generated lemma and Proverbot9001, a state-of-the-art automated prover. In other words, lfind has successfully generated an appropriate auxiliary lemma, proven that lemma, and used the lemma to complete the original proof. The lemma lem1 is such an Λ_1 lemma; the full proof of the theorem rev_rev using lem1 is shown in Figure 3.

Λ_2 lemmas are those that are not disprovable by QuickChick and are sufficient to automatically prove the original goal, but Proverbot9001 cannot automatically prove the auxiliary lemma. lfind indicates that the second and third lemmas in the above listing are Λ_2 lemmas; indeed, each of them in turn depends on its own auxiliary lemmas, for example, the associativity of app. However, both lemmas are also still good options for the user: the lemma lem2 is a more general version of lem1, while lemma lem3 reduces to the original rev_rev lemma when l2 is Nil. Λ_3 lemmas are ones that are not disprovable by a tester like QuickChick but automation using Proverbot9001 can't be used to prove either the goal or the auxiliary lemma; since they are similar to the goal and not disprovable, they might still be useful to the user.

In the rest of this section we explain how `lfind` produces these results.

2.2 Approach

As mentioned in Section 1, the generality of our setting induces several technical challenges. Lemma synthesis in `lfind` has four steps that are targeted at these challenges, as shown in Figure 4. We start by **generalizing** the goal state, in order to systematically introduce new variables that can be used in candidate lemmas. From each generalization, we create sketches and sample variable valuations from the current goal in order to reduce lemma synthesis to **data-driven program synthesis**. Finally, we **filter** the resulting lemma candidates to remove those that cannot be useful and **rank** and categorize the remaining candidates for user inspection.

Generalization. In Coq, helper lemmas are generally used as arguments to the `apply` and `rewrite` tactics. To use the `apply` tactic, the consequent of the lemma must structurally match the goal state to which it is applied. Similarly, to use the `rewrite` tactic, the lemma needs to be an equality or similar relation, one of whose operands structurally matches a portion of the goal state. It is for these reasons that prior techniques for lemma synthesis in the interactive setting [Bundy et al. 1993; Kaufmann and Moore 1997] work by making heuristic rewrites to the goal state. The most common such technique is *generalization*, which replaces terms in the goal state with fresh variables [Kaufmann and Moore 1997].

Our approach starts from the same intuition but aims to use data-driven synthesis instead of heuristic rewrites in order to transform the goal state. However, we observe that generalization provides a systematic way to introduce new variables for the synthesis process. Since we are not sure in advance how many and which variables a useful lemma might need, we exhaustively generate generalizations, one per subset of terms within the goal state. In our example, there are six non-variable terms in the goal (Figure 2). While in principle there are 2^6 possible generalizations using these terms, there are only 16 unique ones, since some terms are only present as subterms of other ones.

For example, replacing `rev l` with a fresh variable `l1` of type `lst` produces the following generalization:

```
forall l n l1, rev (app l1 (Cons n Nil)) = Cons n l.
```

Alone, this generalization does not produce a valid lemma, as it does not hold when `l1` is not the reverse of `l`. Typically generalization is only applied on terms that appear more than once in a goal [Kaufmann and Moore 1997], to avoid these cases. In our example, there are no such terms, and in fact, all lemmas generated by generalization alone are easily disprovable.

Nonetheless, these generalizations play a crucial role in our approach. In addition to being treated as candidate lemmas themselves, we use each generalization as a starting point from which to produce many more candidate lemmas via data-driven program synthesis. Each generalization introduces new variables that can be leveraged as part of that synthesis process.

Synthesis. From each generalization, we create a set of *sketches*, where each sketch is a version of that generalization with one term replaced by a *hole*. For example, if we replace the term `Cons n l` in the generalization above with a hole, then we end up with the following sketch (note that we remove variable `l` from the quantifier since it is no longer used):

```
forall l1 n, rev (app l1 (Cons n Nil)) = □.
```

Intuitively, we would like the expression that fills the hole to behave consistently with the expression that it is replacing. To that end, we generate concrete examples of the original goal

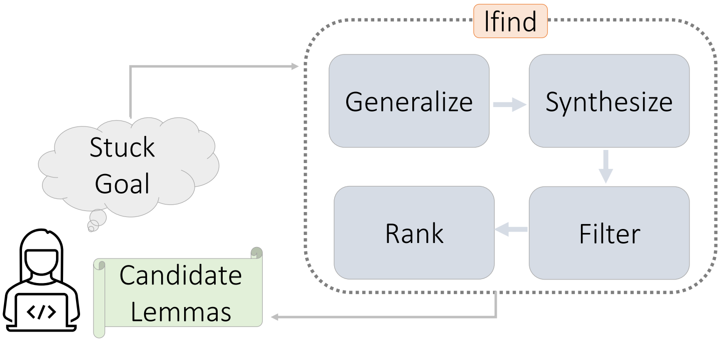

Fig. 4. Overview of `lfind`.

in the stuck state and then map them to input-output examples for data-driven synthesis. In our running example, the original goal has two variables, `l` and `n`, so suppose we randomly generate the following `(l, n)` pairs (using regular list notation for clarity):

`{([], 4), ([0; 1], 2), ([2; 1], 1)}`.

Next, we map these examples to our sketch. We do so by leveraging the fact that the new variable `l1` replaced `rev l` from the original goal. Hence we evaluate `rev l` for each of our three examples to produce the following `l1` values: `{[], [1; 0], [1; 2]}`. Similarly, we produce the expected value of the hole for each example, by leveraging the fact that the hole replaced `Cons n l`. This yields the values `{[4], [2; 0; 1], [1; 2; 1]}`.

As a result of this mapping, we can now produce a set of input-output examples that act as a specification for synthesis, each mapping `(l1, n)` pairs to the expected output value of the term to be synthesized:

```
([], 4) ↦ [4]
([1; 0], 2) ↦ [2; 0; 1]
([1; 2], 1) ↦ [1; 2; 1]
```

Finally, we pass these input-output examples to a data-driven synthesizer. In addition to the examples, we provide the type of the function to be synthesized (which in this case is `lst * nat → lst`) and a grammar to use for term generation. `lfind` automatically creates a grammar consisting of the definitions that appear in the stuck proof state along with definitions that they recursively depend upon. In our example the grammar includes the constructors `Nil` and `Cons` and the functions `app` and `rev`. One term that the synthesizer generates from these inputs is `Cons n (rev l1)`. Substituting this expression into the hole in our sketch yields exactly the lemma `lem1` shown earlier, which enables a fully automated proof of the original lemma.

Note that synthesis is much more expensive than the generalization process described above, which simply replaces some terms with variables. Furthermore, the need for lemmas to structurally match the goal state limits how many parts of that state can be usefully rewritten. For these reasons, we consider only one hole per lemma sketch, but `lfind`'s algorithm conceptually does not limit the number of holes per sketch. Our technique can be extended to synthesize terms for each hole one at a time and then induce candidate lemmas from their combinations.

In summary, we have shown how to generate candidate lemmas in a targeted way, based on the current proof state, using a novel combination of generalization and data-driven program synthesis. While the expressions that are generated by synthesis can make use of a general grammar, the form of the lemmas that we generate are still limited to the structure of the sketches that we produce. As we demonstrate in §5, our approach can generate useful lemmas for a variety of interesting benchmarks.

Filtering. As described above, our approach induces many generalizations of each goal, multiple sketches for each generalization, and multiple synthesis results for filling each sketch's hole. Hence, the set of candidate lemmas that are generated is quite large. In our running example, with default settings for the number of sketches to produce for each generalization and the number of synthesis results to produce for each sketch (see Section 5.2), `lfind` generates 276 candidate lemmas. While the ability to explore a large space of candidates is a strength of the approach, we must organize these candidates in a manner that is understandable and beneficial to users.

To that end, we filter out extraneous candidates in multiple ways. First, we filter out candidates for which we can find a counterexample; we search for counterexamples using QuickChick, an existing counterexample-generating tool [Paraskevopoulou et al. 2015]. Second, we filter out candidates representing trivial facts, for example `forall l, rev l = rev l`. We identify such cases using Coq's `trivial` tactic.

Finally, we filter out candidates that "follow directly" from the user's original lemma, a notion we explore in more detail in 3.4. For instance, in our running example, one candidate lemma is `forall n l, rev (rev (Cons n l)) = Cons n l`, which is a special case of the original `rev_rev` lemma and hence is discarded in this step.

Ranking. After filtering, there are 21 candidate lemmas remaining in our running example. While that constitutes a 92.4% reduction, it is still too many candidates to require the user to examine. Hence, we rank candidates based on their likely utility to the user and present them in ranked order. Since ultimately the utility of a lemma is based on whether it allows the user to prove the original goal, our ranking leverages a state-of-the-art automatic prover for Coq, Proverbot9001, which searches the space of Coq tactics to try to prove a given goal [Sanchez-Stern et al. 2020].

Specifically, we use the automatic prover to partition the candidate lemmas into the three groups introduced in Section 2.1: Λ_1 lemmas that are automatically provable and enable automatic proof of the user's stuck proof state; Λ_2 lemmas that are not automatically provable but enable automatic proof of the user's stuck proof state; and the remaining Λ_3 lemmas. Next, we sort each group in order of size from least to greatest, since we expect smaller lemmas to be easier for users to understand and evaluate. Finally, we concatenate these sorted groups to form the final ranked list.

In our running example, there are 2, 2, and 17 lemmas respectively in each of these three categories. The first lemma in category Λ_1, which yields a fully automated proof, is `lem1` shown earlier, so it is ranked first. Lemmas `lem2` and `lem3` are the smallest lemmas in category Λ_2 and hence are ranked next in our results.

3 ALGORITHMS

In this section we formally describe the core algorithms that make up our approach.

3.1 Preliminaries

Our approach synthesizes lemmas for a given proof state Ψ, which is a tuple $\langle \mathcal{H}, g, \Gamma, \mathcal{D} \rangle$, where $\mathcal{H}$ is a set of logical formulas that are the current *hypotheses*, g is a logical formula that is the current *goal*, Γ is a type environment for all free variables in $\mathcal{H}$ and g, and $\mathcal{D}$ is a set of type and term definitions that are recursively referred to in $\mathcal{H}$ and g. We require that the goal g be unquantified, which in practice typically means that the original lemma/theorem should have all variables universally quantified at the front.

We use ϕ to denote logical formulas, x for variables, v for values, t for terms of sort `Type`, and τ for the types of terms. A *sample* for a proof state $\Psi = \langle \mathcal{H}, g, \langle x_1 : \tau_1, \ldots, x_n : \tau_n \rangle, \mathcal{D} \rangle$ is an environment $e = \langle x_1 : v_1, \ldots, x_n : v_n \rangle$ such that e is a model of $\mathcal{H} \to g$, denoted $e \vDash_{\mathcal{D}} \mathcal{H} \to g$. We also use the notation $t \Downarrow_e v$ to denote the evaluation of term t to value v under environment e.

Finally, we assume the existence of several black-box functions that have been created by others in prior work. We assume the availability of a black-box synthesizer that takes as input a grammar $\mathscr{G}$, consisting of typed constants and functions; a type signature $\tau_1 \longrightarrow \tau_2$; and input-output examples of the form (v_1, v_2), where v_1 has type τ_1 and v_2 has type τ_2. This synthesizer returns a list of functions f of type $\tau_1 \longrightarrow \tau_2$ in the grammar $\mathscr{G}$ such that $f(v_1) = v_2$ for all examples; or it fails after some time limit. We also assume the existence of a function Sample(Ψ) that produces a set of samples. Last, we assume the existence of both automated theorem provers and disprovers. A *prover* $\mathcal{R}(\phi, \bar{\phi}, \mathcal{D})$ attempts to prove a given formula in the context of a set of auxiliary lemmas $\bar{\phi}$ as well as a set of definitions, returning either Valid or Dont Know. A *disprover* $C(\phi, \mathcal{D})$ searches for concrete counterexamples to ϕ and returns either Invalid or Don't Know.

3.2 Lemma Synthesis

First, we describe how we reduce lemma synthesis to data-driven program synthesis. As described in the previous section, the first step is to produce *generalizations* of the current goal g, by replacing some set of terms in g with fresh variables. The following definition formalizes this notion of generalization.

Definition 3.1. (Generalization: $\mathcal{G}$) Given a goal g, a type environment Γ, and a set $\mathbf{T} = \{t_1, \ldots, t_n\}$ whereby no term in $\mathbf{T}$ is a subterm of any other term in the set, we define the *generalization* of g with respect to Γ and $\mathbf{T}$, denoted $\mathcal{G}(g, \Gamma, \mathbf{T})$, as the tuple $\langle g', \Sigma \rangle$, where $\Sigma = \langle x_1 \mapsto t_1, ..., x_n \mapsto t_n \rangle$ records the mapping from each new variable to the term that it replaces, variables $x_1, \ldots, x_n$ are not in the domain of Γ, and $g' = g[\bar{t_i} \mapsto \bar{x_i}]$.

As in `lfind`, the definition of generalization above replaces *all* occurrences of a particular subterm with the same variable, though it is possible to relax this requirement. The restriction that no term in $\mathbf{T}$ can be a subterm of any other term in the set ensures that their simultaneous substitution is well defined. In `lfind` any term that is a subterm of another term to be generalized is simply ignored.

For example, the goal state:

```
(atan2(x, y) - z) + atan2(x, y) = 2 * (atan2(x, y) - z)
```

can generalize to the tuple:

```
⟨ (a - z) + a = 2 * (a - z), ⟨ a ↦ atan2(x, y) ⟩⟩
```

Here, g' is `(a - z) + a = 2 * (a - z)`, and Σ is the mapping from `a` to the term it replaced in the original expression.

`lfind` uses the generalizations that it constructs as candidate lemmas. In addition, generalizations are used as the basis for creating *sketches* for data-driven synthesis. Each sketch is simply a version of a generalized goal that has one term replaced by a *hole*, denoted □.

Definition 3.2. (Sketch: $\mathcal{S}$) A *sketch* of goal g with respect to term t, denoted $\mathcal{S}(g, t)$, is $g[t \mapsto \square]$.

For example, a sketch of the generalized goal

```
(a - z) + a = 2 * (a - z)
```

with respect to `2 * (a - z)` would be

```
(a - z) + a = □
```

Technically the definition above replaces all occurrences of the term t with a hole, but in `lfind` each sketch has a single hole. In particular, `lfind` creates one sketch per occurrence of a non-variable term in the generalized goal state.

In order to produce a data-driven program synthesis problem, we must generate input-output examples. The following definition shows how we extend an environment to an input-output example, given a set of terms (which are used for generalization), and a term (used for creating a sketch). Intuitively, the new variables created by generalization become additional input variables, and the term used to create a sketch defines the expected output.

Definition 3.3. (Input-output example: IO) The input-output example corresponding to a given environment $e = \langle x_1 \mapsto v_1, \ldots, x_n \mapsto v_n \rangle$, term mapping $\Sigma = \langle x'_1 \mapsto t_1, ..., x'_m \mapsto t_m \rangle$, and term t_s, denoted $\mathrm{IO}(e, \Sigma, t_s)$, is defined as

$$\langle \langle x_1 \mapsto v_1, \ldots, x_n \mapsto v_n, x'_1 \mapsto v'_1, ..., x'_m \mapsto v'_m \rangle, v_s \rangle$$

where $t_i \Downarrow_e v'_i$ for each t_i in $t_1, \ldots, t_m$ and $t_s \Downarrow_e v_s$.

For example, given (1) the environment $e = \langle x \mapsto 2, y \mapsto 3, z \mapsto 4 \rangle$, which are values for the variables in our original example goal state, (2) the term mapping $\Sigma = \langle a \mapsto$ `atan2(x,y)`$\rangle$ from our example generalization, and (3) the term `2 * (a - z)` that was replaced with a hole in our example sketch, we produce an input-output example whose input tuple extends the environment e with a mapping from a to the value of `atan2(2,3)` and whose expected output is the value of `2 * (atan2(2,3) - 4)`.

Finally, we can put all of this together to specify how to reduce lemma synthesis to data-driven program synthesis.

Definition 3.4. (Lemma synthesis as data-driven program synthesis) Given a proof state $\Psi = \langle \mathcal{H}, g, \Gamma, \mathcal{D} \rangle$, a set of terms $\mathbf{T} = \{t_1, \ldots, t_m\}$ for generalization, and a sketch term t_s, we produce a data-driven program synthesis problem as follows. Let $\mathcal{G}(g, \Gamma, \mathbf{T}) = \langle g', \Sigma \rangle$, where $\Sigma = \langle x'_1 \mapsto t_1, ..., x'_m \mapsto t_n \rangle$. Let $\mathcal{S}(g', t_s) = g_s$.

- The grammar $\mathscr{G}$ for synthesis is defined by the type and term definitions in $\mathcal{D}$.
- Let Γ_s be Γ restricted to the variables that appear free in g_s. The input variables and associated types for the function to be synthesized are Γ_s@$\langle x'_1 : \tau_1, ..., x'_m : \tau_n \rangle$, where $\Gamma \vdash t_i : \tau_i$ for each $t_1, \ldots, t_m$.
- The output type for the function to be synthesized is τ_s, where $\Gamma \vdash t_s : \tau_s$.
- The input-output examples for synthesis are produced as follows. First we generate a set of samples $\textsc{Sample}(\Psi) = \langle e_1, \ldots, e_p \rangle$ for the given proof state. Then the set of input-output examples is $\langle \mathrm{IO}(e_1, \Sigma, t_s), \ldots, \mathrm{IO}(e_p, \Sigma, t_s) \rangle$.

We invoke the synthesizer with these inputs and ask for the k smallest functions (§ 5 reports sensitivity analysis for k) that meet this specification. For each such function f, with body expression t_f, the induced *candidate lemma* is created by universally quantifying all free variables in the term $g_s[\square \mapsto t_f]$.

Above we have formalized the process of lemma candidate generation from a single set of terms to be generalized and a single term to be used for creating a sketch. `lfind` performs this process many times, for many different generalizations and many different sketch terms. Various approaches to exploring this space are possible. `lfind` *exhaustively* explores the generalization space, producing one generalization for each subset of terms in the goal g. For each such generalization, `lfind` employs terms that have sort `Type` for creating sketches. There are several ways to pick a synthesis term for a sketch, and in §5 we carry out sensitivity analysis for two natural approaches to choosing such terms.

3.3 Filtering

The approach described so far generates *a lot* of candidate lemmas. If there are t subterms in a given goal to use for generalization, m sketches per generalization, and we ask the synthesizer for k results, then without any filtering `lfind` would produce a maximum of $2^{t+1}mk$ candidates, including all generalizations and the lemmas derived from them using data-driven synthesis. Exploring a large space of candidates is advantageous, but clearly we require techniques to filter out candidates that are not going to help the user.

We employ four different filtering techniques. First, duplicates among candidate lemmas are common. For example, it's possible for synthesis from two different sketches to produce the same result. It's also possible for synthesis from a single sketch to produce syntactically distinct results that are behaviorally equivalent. We identify and filter duplicates by applying Coq's `simpl` tactic and then comparing the results for syntactic equivalence. Second, we use the disprover C to search for counterexamples, filtering out any candidate ϕ such that $C(\phi, \mathcal{D}) = \textsc{Invalid}$. Third, we remove lemmas that can be solved using Coq's `trivial` tactic, since they are self-evident and hence never needed as explicit auxiliary lemmas.

Finally, we filter lemmas that "follow directly" from the original lemma, as they will not help in proving that lemma. This is a subtle notion. For example, it is not a form of logical implication, since if the candidate lemma is valid then *any* other lemma implies it. Instead, we formalize this filter via a binary relation $\preceq$, which says that one lemma is an instantiation of (or equivalent to) another, defined as follows:

Definition 3.5. ($\preceq$-operator) Given lemmas $l1$ and $l2$, we say $l1 \preceq l2$ if we can prove $l1$ using either of the following proof scripts:

```
intros. apply l2. Qed.
intros. rewrite <- l2. reflexivity. Qed.
intros. rewrite -> l2. reflexivity. Qed.
```

We then filter out any candidate lemma that is $\preceq$ than the original lemma.

3.4 Ranking

We rank the remaining candidate lemmas using the automated prover $\mathcal{R}$ we introduced earlier. For each candidate ϕ we use the prover to determine if the candidate can be used to automatically prove the goal g — $\mathcal{R}(g, \{\mathcal{H}, \phi\}, \mathcal{D})$ — and whether the candidate itself is automatically provable — $\mathcal{R}(\phi, \emptyset, \mathcal{D})$. Based on the results we partition the lemmas into three groups, Λ_1, Λ_2, and Λ_3. The Λ_1 group contains the lemmas for which both calls to $\mathcal{R}$ return $\textsc{Valid}$, meaning that we have obtained a fully automated proof of the user's original goal. The Λ_2 group contains the lemmas for which the first call to $\mathcal{R}$ return $\textsc{Valid}$, meaning that the lemma enables the goal to be automatically proven but the lemma is not itself automatically provable. The remaining lemmas go in the Λ_3 group. We sort the lemmas in each group in order of size from smallest to largest, since we expect smaller lemmas to be easier for users to understand and evaluate. Finally, we concatenate these sorted groups to form the ranked list.

3.5 Discussion

We note that `lfind`'s approach to candidate lemma generation imposes some important restrictions on its usage. We have already mentioned that the goal in the proof state must be unquantified. Further, the approach relies on the ability to generate examples for the stuck state, which limits it to the capabilities of current test generation techniques. Because we reduce lemma synthesis to program synthesis we require the ability to extract necessary definitions as code and translate code back to Coq. Finally, because sketches for synthesis are derived from a generalization of the original goal, the generated lemmas will always have the same top-level structure as the goal. For example, if the original goal has the form `A = B` then the candidate lemmas will also have this form. §5.3 shows that despite these limitations, `lfind` can successfully identify non-trivial helper lemmas for a variety of examples. In addition, all of these limitations represent useful avenues of investigation in future work.

4 IMPLEMENTATION

Figure 5 illustrates the overall architecture of `lfind`, which leverages three black-box components: a data-driven synthesizer for candidate lemma generation; an automatic disprover for candidate filtering; and an automatic prover for candidate ranking. Our implementation of `lfind` is 3.2 KLOC of OCaml code.

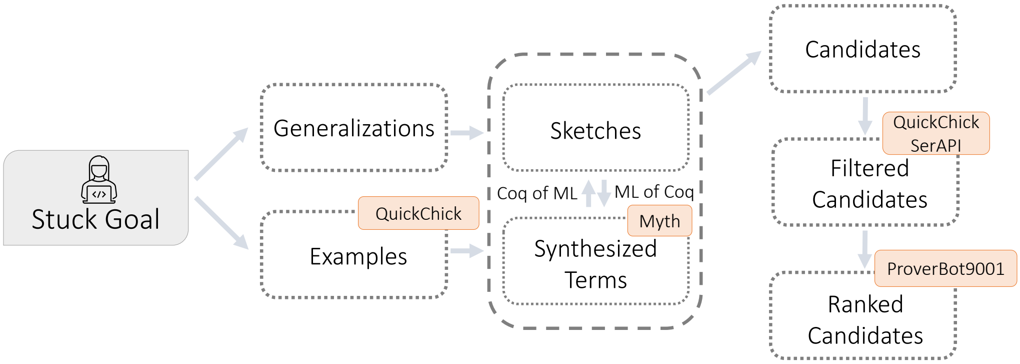

Fig. 5. Given a stuck goal, `lfind` implements generalization, synthesis, filtering, and ranking in conjunction with existing tools to generate candidate lemmas.

4.1 Example Generation

To synthesize candidate lemmas, our approach relies on a Sample function that can produce samples for the variables in the stuck state g. We leverage QuickChick [Paraskevopoulou et al. 2015], a state-of-the-art property-based randomized tester for Coq, for this purpose. While QuickChick is intended as a testing tool, we log all of the test inputs that it generates and use them as the samples from which to produce examples for synthesis.

Specifically, for each user-defined type `T` in the stuck goal, `lfind` first generates the following Coq code, which enables the usage of QuickChick for that type:

```
Derive Show for T.
Derive Arbitrary for T.
Instance Dec_Eq_T : Dec_Eq T.
    Proof. dec_eq. Qed.
```

The `Show` typeclass is required for printing test cases and the `Arbitrary` typeclass is required to combine test-case generation with an operation for shrinking test inputs. QuickChick supports automatic derivation of instances of these type classes for simple types. QuickChick also requires that types have decidable equality, so we derive an instance of the `Dec_Eq` typeclass for `T`.

Next, to produce examples for the stuck proof state g, we create a Coq lemma for that state, defined as `Lemma stuckState:` $\mathcal{H} \longrightarrow g$. We also create a function `collect_data` whose input type `V` is the tuple of the types of all free variables in $\mathcal{H} \longrightarrow g$. The function logs the input values to a file and returns the valuation of $\mathcal{H} \longrightarrow g$ on those input values:

```
Definition collect_data  (n:V) :=
 in let _ := print_to_file (show n)
 in stuckState n.
QuickChick collect_data.
```

Finally, we run QuickChick on this function, thereby logging samples to use for data-driven synthesis and also searching for counterexamples to the stuck proof state. If QuickChick returns any such counterexamples, then there is no way to complete the proof so we report this to the user and halt `lfind`. Otherwise, we proceed with synthesis.

4.2 Synthesis

To our knowledge, there are no data-driven synthesizers that work directly for Coq. We chose Myth [Osera and Zdancewic 2015] as our synthesizer because it accepts and generates OCaml code, for which tools exist to convert to/from Coq's language Gallina; it has a simple interface that is easy to use, and it has worked well for us in the past. Myth requires an input grammar in OCaml, so we use Coq's `Extraction` feature to recursively extract reachable definitions and types from the stuck goal to OCaml. Additionally, we adapt Myth slightly in two ways. First, Myth supports only a subset of OCaml and does not support common syntactic sugars. For example, Myth does not support the `function` keyword. To get around these limitations, we wrote a translator that desugars the definitions extracted from Coq into a form acceptable by Myth. Second, we modified Myth to return a set of candidate functions sorted by size, instead of just one. This enables the generation of multiple candidate lemmas as described earlier. Finally, to substitute the synthesized OCaml function body back into our lemma sketch, we use an open-source tool, `coq-of-ocaml` [coq 2003].

4.3 Filtering and Ranking

In §3.3, we described multiple filters to remove extraneous candidate lemmas. To implement these filters, we declare each candidate as a Coq lemma and use QuickChick to remove lemmas that have counterexamples. The remaining filters are implemented by running proof tactics using `SerAPI` [Gallego Arias et al. 2020], a library for machine-to-machine interaction with Coq. To rank the filtered lemmas, we use Proverbot9001 [Sanchez-Stern et al. 2020], a state-of-the-art proof-synthesis tool that uses machine learning to produce proofs of Coq theorems. Proverbot9001 takes as input definitions, a theorem that needs to be proven, and a set of axioms that can be assumed, and returns a proof script or Don't Know.

4.4 Discussion

In our implementation, we try to disprove each generalization eagerly, and we only carry out synthesis from generalizations for which the disprover finds a counterexample. Intuitively, if a generalization is not disprovable then it is itself a candidate lemma, and so we would rather spend our synthesis resources elsewhere. Candidate lemmas are produced incrementally, as generalization and synthesis proceed. Hence the algorithm is *any-time*: we can stop at any point, collect up the current set of candidates, and filter and rank them. Furthermore, we stop synthesis as soon as we get a category Λ_1 lemma since we will have a fully automated proof of the user's original goal.

Our implementation inherits the limitations of the black-box tools we rely on. Notably, Myth only supports a small subset of OCaml. As described above, we mitigate this limitation by implementing a translator, but this is not a solution that works for the full OCaml language, and so in some cases `lfind` can fail to produce code that Myth accepts. Myth also does not support polymorphic types.

5 EXPERIMENTAL RESULTS

In this section we perform experiments to answer the following research questions:

RQ1. (§5.3) How effective is `lfind` in synthesizing useful helper lemmas? How fast can the tool synthesize these helper lemmas? What is the impact of its filtering and ranking techniques?
RQ2. (§5.4) How does `lfind`'s data-driven approach compare in effectiveness to prior approaches to lemma synthesis?
RQ3. (§5.5) How sensitive is `lfind` to hyperparameters and timeouts?

5.1 Benchmark Suite

Our approach generates candidate helper lemmas from a given proof context, and our tool is implemented as a *tactic*. Hence, to evaluate `lfind` we need to invoke `lfind` at each point in the proofs where a user-provided helper lemma was used. These are called *evaluation locations*. Concretely, a proof state is an *evaluation location* if a human prover has used either the `apply` or `rewrite` tactics with a helper lemma that they created. We evaluate `lfind` on a total of 222 evaluation locations. These benchmark locations are drawn from the following sources.

- CLAM (140): This benchmark suite consists of 86 theorems about natural numbers as well as various data structures, including lists, queues, and trees, and it has been used to evaluate prior forms of lemma synthesis [Ireland and Bundy 1996; Yang et al. 2019]. These benchmarks lack associated proofs, so we converted them to Coq and manually proved each theorem (more details on this process below). Out of the 86 CLAM theorems, 67 require at least one helper lemma, with many requiring multiple lemmas. In total, the CLAM suite contains 184 unique evaluation locations that employ a helper lemma. Implementation limitations mentioned in §4.4 preclude 44 locations from CLAM from being used for evaluation, leaving 140 remaining evaluation locations.
- Full Adder (62): This project [cir 1995] from the `coq-contribs` collection formalizes a full adder and proves it correct [cir 1995]. The program first builds a half-adder circuit (which takes two binary digits, and outputs two binary digits) and proves properties about it. Then the half-adder circuit is used to build a full-adder circuit (which takes two binary digits, plus a "carry" digit, and outputs two binary digits). Finally, the program chains together a sequence of full adders to create an adder circuit, which is proven correct. All of the 40 theorems in this project require at least one helper lemma, and the project contains 62 evaluation locations in total.
- Compiler (1): This benchmark is the compiler example from Chapter 2 of Chlipala's CPDT textbook [Chlipala 2013], which is a certified compiler from a source language of expressions to a target language of a stack machine. The final theorem formalizes the correctness of the compiler. This benchmark contains one theorem, which uses one helper lemma, which is the evaluation location. Though it contains only a single evaluation location, we chose this example as a benchmark because it showcases a different application and the required helper lemma is relatively large and complex.
- LIA (19): This benchmark suite consists of 9 theorems about data structures that require linear integer arithmetic, from a prior work on lemma synthesis for fully automated proofs about data structures (see Table 1 in [Yang et al. 2019]). As with the CLAM benchmarks, we converted them to Coq and manually proved each theorem. Each proof requires at least one helper lemma, and there are a total of 19 evaluation locations.

The Full Adder and Compiler benchmark suites already contain full Coq proofs written by others, which in turn determine our evaluation locations. The theorems in the CLAM and LIA benchmark suites lack proofs, so each theorem was manually proven by one of three of us, with

Table 1. Results

	CLAM	Full Adder	Compiler	LIA
Setup				
# Theorems	86	40	1	9
# Evaluation Locations	140	62	1	19
Results				
# fully proven lemma and goal	73	35	0	9
# else human match in top 1	17	0	1	1
# else human match in top 5	6	2	0	1
# else human match in top 10	4	0	0	0
# else more general than human lemma in top 1	1	0	0	0
Summary	101/140	37/62	1/1	11/19

varying experience from novice to expert in interactive theorem proving. Specifically, one person had only done a small class project with Coq previously, one has been using Coq for the past few years on a research project, and one has used it on and off for a decade. The proofs were completed independently of `lfind`'s evaluation, and helper lemmas were used wherever the human prover deemed necessary. In § 5.4, we show that the vast majority of these helper lemmas are indeed necessary, in the sense that a state-of-the-art automated prover cannot complete the proof of the theorem without a helper lemma. We have provided all the benchmarks as part of the anonymous supplementary material.

5.2 Experimental Setup

For each evaluation location, `lfind` generates 50 input-output examples from the current proof state and is allowed to generate candidate lemmas with a maximum timeout of 120 minutes. Despite the large search space, in §5.3 we show that the tool is performant with a median runtime of 3.3 min. The tool has a 12s timeout for each call to Myth and a 30s timeout for each call to Proverbot9001. In addition to the timeout parameters, two key hyperparameters to our algorithm are the choice of subterms to use for generating sketches and the number of synthesis results k to obtain per sketch. In our experiments, we generate sketches from *all subterms* of sort `Type`, and we ask for *5* synthesis terms per sketch. Empirically we have found these choices to provide good results, but we also present a sensitivity analysis of other choices for timeout and hyperparameters in § 5.5.

All evaluations were performed on a machine that runs MacOS (10.15.6) in a 2.3 GHz-Quad-Core Intel Core i7, with 32GB memory.

5.3 Synthesized Helper Lemmas

Table 1 summarizes the results for all our benchmarks. We consider the use of `lfind` at an evaluation location to be successful in three scenarios. First, we say that `lfind` is successful if it can produce a candidate helper lemma that is automatically proven by Proverbot9001 and this helper lemma enables Proverbot9001 to automatically prove the user's goal. This is the best-case scenario, as `lfind` has produced a complete proof for the user. Second, we say that `lfind` is successful if a lemma that matches the human-provided lemma is ranked highly (top-10) by the tool. Third, we say that `lfind` is successful if a lemma that is more general than the human-provided lemma is ranked highly by the tool. We use the $\preceq$ operator defined in §3.3 to automatically identify if a candidate lemma l matches or is more general than the human-provided lemma h. Specifically we say that l matches h if both $l \preceq h$ and $h \preceq l$, and we say that l is more general than h if $h \preceq l$ but not vice versa. These are reasonable success metrics for our tool, as we expect versions of the human-provided lemma to be "natural" for people to understand, and also we know that the human-provided lemma does indeed lead to a full proof of the goal. Note however that the metrics are conservative, as there could be other lemmas produced by `lfind` that are natural and appropriate but do not fall into one of the above three categories.

In total, based on our evaluation metrics we see that `lfind` succeeds in 150 (67.5%) of the 222 evaluation locations across all benchmarks. Further, as shown in the third row of the table, in 117 (78%) of these successful 150 locations, `lfind` was able to synthesize a lemma that led to a fully automated proof of the user's goal. Rows 4-7 of Table 1 show a breakdown of the remaining 33 successful locations. Notably, for 19 of these evaluation locations, the top-ranked candidate lemma produced by `lfind` matches the helper lemma provided by the human prover. These results demonstrate the effectiveness of our filtering and ranking strategies in surfacing relevant lemmas toward the top, and often as *the* top result. Further, 83.3% of the successful locations employed data-driven synthesis, rather than solely using generalization. In those cases, we found that on average 43.4% of the candidate lemma comes from the synthesized term, and the rest comes from the lemma sketch.

Examples. Table 2 shows examples of lemmas synthesized by `lfind` along with their rank and category (see §3.4 for category notations). We describe the first four of them in detail.

The first example from the `Compiler` benchmark formalizes the correctness of a compiler from a source language of expressions to a target language of a stack machine. In this case, type `exp` defines the source language of arithmetic expressions. `evalExp` function evaluates the programs in this language. The target language's instructions are of type `instr`, which are executed on a stack machine. The function `execI` takes an instruction and a stack (represented as a list of `nat`s) and returns an updated stack, and `execIs` uses this function to execute a list of instructions. Finally, the `compiler` function translates source programs to a list of instructions. The theorem itself is not inductive, necessitating an inductive helper lemma that implies the theorem [Chlipala 2013]. `lfind` was not able to identify a helper lemma that leads to a fully automated proof of the theorem. However, it produces candidate lemmas in categories Λ_2 and Λ_3, and the top-ranked candidate in category Λ_2, shown in the table, *exactly matches* the human-provided lemma. The lemma is non-trivial as it involves multiple calls to `execIs`, an arbitrary list of stack instructions `l`, and an arbitrary stack `s`.

The second example is from the FULL ADDER benchmark. The theorem says that if we convert to a natural number the result of adding a binary number, we get the same natural number we would if we converted that input to a natural number. We present a synthesized helper lemma in table 2, which belongs to category Λ_1 and hence led to a full proof of the theorem.

The third example in the table is from the CLAM benchmark suite and proves the equivalence of two functions for converting a binary tree into a list. For this example, `lfind` produced candidate helper lemmas in both categories Λ_2 and Λ_3. The tenth-ranked candidate, shown in the table, matches the human-provided lemma.

The fourth example in the table is from the LIA benchmark suite and reasons about how pushing onto a queue affect its length. This is a case in which our evaluation does not deem `lfind` to have succeeded, since it does not produce a fully automated proof and does not produce a match for the human-provided lemma in the top ten results. However, the top-ranked result, shown in the figure, is *very* close to the human-provided lemma, which simply replaces the term `(rev l1)` with `l1`. Further, this lemma is itself equally useful in completing the proof, despite being slightly more complex.

Runtime Performance. Figure 6 plots the runtime distribution of `lfind` across all 222 evaluation locations. The tool ran to completion on each of these benchmarks with a median runtime of 3.3 min (shown in the plot where the curve labeled TOTAL TIME reaches a CDF of 0.50). Recall that `lfind`

Table 2. A sample of lfind synthesized lemmas and their associated rank and category.

#	Original Theorem & Proof State	lfind Synthesized Lemma	Λ
1.	Theorem correct_compilation:∀ (e :exp), execIs (compile e) Nil = (evalExp e) :: Nil. Proof State: execIs (compile e2) ++ (compile e1) ++ (Add :: Nil) Nil = (evalExp e1) + (evalExp e2)::Nil	Lemma lem1:∀ (e: exp) (l: list instr) (s: list nat), execIs (compile e ++ l) s = execIs l (evalExp e :: s).	Λ_2
2.	Theorem BV_full_adder_nil_ok :∀ (v:BoolList) (cin:bool), BV_to_nat (BV_full_adder v Nil cin) = BV_to_nat v + Bool_to_nat cin. Proof State: BV_to_nat (BV_full_adder_sum (b :: b0) Nil false ++ BV_full_adder_carry (b :: b0) Nil false :: Nil) = BV_to_nat (b :: b0) + bool_to_nat false	Lemma lem1:∀ (l:BoolList), BV_to_nat (BV_full_adder_sum l Nil false ++ BV_full_adder_carry l Nil false::Nil) = BV_to_nat l.	Λ_1
3.	Theorem app_revflat:∀ (x:tree) (y:list nat), (revflat x) ++ y = qrevaflat x y. Proof State: (revflat x1 ++ n :: revflat x2) ++ y = qrevaflat x1 n :: qrevaflat x2 y	Lemma lem10:∀ (l l1 l2:list nat), (l ++ l1) ++ l2 = l ++ (l1 ++ l2).	Λ_3
4.	Theorem queue_push:∀ (q:queue) (n:nat), qlen (qpush q n) = 1 + (qlen q). Proof State: len l + len (rev l0 ++ n :: Nil) = S (len l + len l0)	Lemma lem1:∀ (l l1:list nat), len l + len l1 = len (l ++ rev l1).	Λ_2
5.	Theorem qreva_qreva:∀ (x:list nat), (qreva (qreva x Nil) Nil) = x. Proof State: qreva (qreva x Nil) ++ (n :: Nil) Nil = n :: x	Lemma lem9:∀ (n:nat) (l:list nat), qreva (l ++ n :: Nil) Nil = n :: (qreva l Nil).	Λ_2
6.	Theorem rotate_len:∀ (x:list nat), rotate (len x) x = x. Proof State: rotate (len x) (x ++ n :: Nil) = n ++ x	Lemma lem2:∀ (l l1:list nat), rotate (len l) l ++ l1 = l1 ++ l.	Λ_2
7.	Theorem drop_elem:∀ (v w x y:nat) (z:list nat), drop (S v) (drop w (drop x y::z)) = drop v (drop w (drop x z)). Proof State: drop (S v) (drop w (drop x y :: z)) = drop v (drop w (drop x z))	Lemma lem1:∀ (n x:nat) (l:list nat), drop (S x) (drop n l) = drop x (drop (S n) l)).	Λ_1

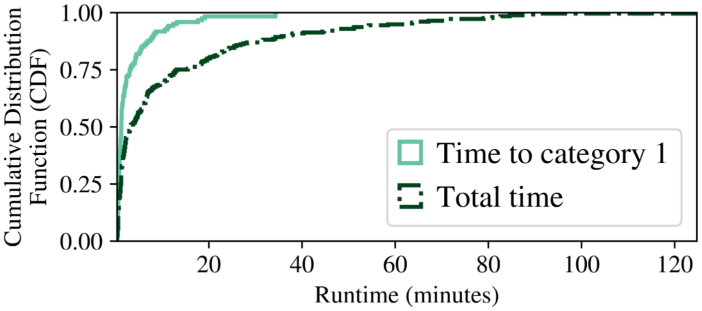

Fig. 6. `lfind` has a median total runtime of 3.3 min. Further, the tool has a median runtime of 1.2 min for the 117 cases (see table 1) where it was able to find a full automated proof (Λ_1).

produces a full automated proof (category Λ_1) in 78% (see Table 1) of the successful evaluation locations. As shown by the curve labeled TIME TO CATEGORY 1 in Figure 6, the median and 75th percentile runtime of the tool were only 1.2 min and 3.0 min respectively. These runtimes indicate the viability of our approach and its instantiation in `lfind` to support interactive usage.

Impact of Filtering and Ranking. Figure 7 provides a detailed view of how many candidate lemmas were generated and filtered, for the results presented in Table 1. As explained in §3.3, our approach indeed generates a *lot* of candidate lemmas. For example, `lfind` generates a median of 150 candidate lemmas for each evaluation location from the benchmarks (shown where the solid curve reaches a CDF of 0.50). However, our filtering techniques are very effective in removing useless lemmas. As mentioned in §4, we filter INVALID candidates (labeled `Filter 1` in the figure) as we generate candidate lemmas. We then filter lemmas (labeled `Filter 2`) that are either syntactically similar to each other, or trivial, or restatements or special versions of the theorem statement. After `Filter 1`, the median number of lemmas is reduced to 100. Further, after `Filter 2` there is a median of 16 candidate lemmas. Hence on average, `Filter 1` reduced the candidate lemmas by 33%, and `Filter 2` reduced the remaining candidates by 84%.

Finally, as mentioned above even after filtering we are left with a median of 16 lemmas for each benchmark suite. This highlights the importance of our ranking strategy, which was already shown to be effective in the results of Table 1.

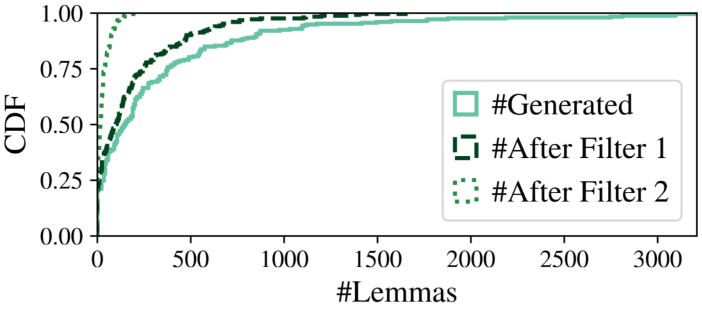

Fig. 7. `lfind` reduces the number of lemmas by 89.3% on average after application of both filters.

Failure Analysis. Table 3 summarizes the failure reasons for the 72 locations where `lfind` did not identify a successful candidate lemma. Broadly there are two classes of failures. First, there

Table 3. Failure Analysis

	CLAM	Full Adder	Compiler	LIA
Not ranked in top-10	6	0	0	2
Not in k smallest Myth results	4	9	0	2
Requires generalization of variables	4	0	0	0
Incomplete grammar	4	0	0	0
Lemma rewrites a subterm of the goal	22	15	0	4
Total	40	24	0	8

are 23 cases where lfind failed to identify a candidate lemma due to the specific choice of tool hyperparameters (rows 1-2). Recall that lfind is successful if it can produce a category Λ_1 lemma or if a lemma ranked in the top 10 matches the human-provided lemma. As shown in the first row of Table 3, in 8 cases lfind identifies a candidate lemma that matches a human lemma, but it is not ranked in the top 10. The second row contains 15 cases where Myth synthesizes a term that leads to the required candidate lemma, but that term is not among the k (5 in our case) smallest terms that Myth produces and so we do not use it (see §5.2).

Second, there are 49 cases where lfind failed due to algorithmic limitations (rows 3-5 of Table 3), which represent useful avenues of investigation in future work. lfind produces generalizations by replacing non-variable terms in the goal state with fresh variables (see §3.2). However, in four cases (row 3), the generation of a successful candidate lemma requires a generalization that is formed by replacing repeated variables in the goal state with fresh variables. Next, in four cases (row 4), the human-provided helper lemma contains functions that are not part of the goal state, and therefore the grammar generated by lfind for synthesis does not contain those functions. Finally, recall that lfind produces lemma sketches from a generalization of the original goal, so candidate lemmas will always have the same top-level structure as the entire goal. However, 41 cases (row 5) require a helper lemma that is used to rewrite a particular subterm of the goal, and lfind is unable to generate such lemmas.

5.4 Comparison with Other Approaches

In this section, we compare lfind against the three most relevant approaches. First, we compare against a state-of-the-art automated prover to try to complete the proof from the evaluation location (proof context). Second, we compare against generalization [Boyer and Moore 1979; Kaufmann and Moore 1997], which is the most common lemma synthesis technique in interactive theorem provers. Finally, we compare against ADTInd [Yang et al. 2019] which is a state-of-the-art lemma synthesis technique in a non-interactive setting.

No Synthesis. In this study, we ran Proverbot9001 on each evaluation location across all benchmarks, without providing any synthesized lemma from lfind. Proverbot9001 can automatically prove only 27.5% of the evaluation locations. In contrast, with a lemma synthesized by lfind, Proverbot9001 can automatically prove 52.7% of the evaluation locations (117 out of 222), and as shown earlier overall lfind provides a useful lemma in 67.5% of the cases. This experiment highlights the need for lemma synthesis and shows how our work complements existing work on automated proofs. These results also serve as a measure of the quality of the human proofs, as the human-provided lemmas are required in the vast majority of cases. Situations where a lemma is used but not needed could arise due to the inexperience of the human prover or simply for readability purposes.

Generalization. To our knowledge, there is no existing implementation of generalization for Coq. As part of `lfind`, however, we have implemented generalization in Coq, and furthermore, we have implemented an exhaustive version of it: each subset of terms in the current goal state induces a candidate lemma through generalization. This implementation allows us to perform an apples-to-apples comparison between generalization and our approach. Hence, for this comparison, we disable `lfind`'s synthesis process, so Myth is not used at all, but all other parts of `lfind` work as described earlier. This version of `lfind` can be seen as a best-case version of the generalization technique [Boyer and Moore 1979; Kaufmann and Moore 1997], since we *exhaustively* consider all possible generalizations, while in prior tools typically only one or a small number of generalizations are heuristically chosen [Chamarthi et al. 2011; Yang et al. 2019]. According to our success metrics defined in § 5.3, a generalization is deemed useful in only 11.3% of all evaluation locations, as compared with 67.5% of locations for `lfind`.

Enumerative Synthesis. We compare against `ADTInd` [Yang et al. 2019], a state-of-the-art lemma synthesis technique for fully automated theorem proving. Like `lfind`, `ADTInd` employs generalization as well as lemma sketches, but `ADTInd` fills these sketches via grammar-based term enumeration. Unfortunately, a direct tool comparison against `ADTInd` would not be informative due to several important differences between the tools: (1) Unlike `lfind`, which automatically generates lemma sketches from the current proof state, `ADTInd` requires user-provided lemma sketches. The choice of sketches has a large impact on which theorems are able to be successfully proven. (2) Since `lfind` is intended for an interactive setting, the user indicates the proof state from which to carry out lemma synthesis, whereas `ADTInd` automatically decides where to invoke lemma synthesis. (3) `ADTInd` is a fully automated prover, so in addition to lemma synthesis, it also has its own proof search algorithm. Hence whether `ADTInd` can prove a theorem or not depends in large part on the power of that proof search algorithm. Further, that algorithm is very different from the neural-based prover that `lfind` uses for filtering and ranking. To avoid these confounding differences and enable a fair comparison of `ADTInd`'s and `lfind`'s lemma synthesis techniques, we substitute `ADTInd`'s enumeration-based synthesis approach for our data-driven approach but keep everything else the same in `lfind`: the same automatically generated lemma sketches, filtering and ranking, and success metrics.

To perform this comparison, we have created a version of `lfind` that does not provide any examples to Myth whenever it is invoked, but is otherwise identical to `lfind`. Without examples, all terms of the desired type will be considered by Myth to meet the given specification, so the effect is that Myth will perform a type-guided enumeration through the given grammar. This `ADTInd`-like version of `lfind` synthesizes a successful helper lemma according to our success metric in 79 evaluation locations, whereas the unmodified `lfind` does so in 125 evaluation locations. Note that these results exclude cases where generalization produces the useful lemma for an evaluation location since the two versions of `lfind` are identical in those cases. These results demonstrate the benefits of data-driven synthesis: the examples act as a specification that allows for early filtering of candidate lemmas, which in turn enables the synthesizer to provide higher-quality candidates.

5.5 Sensitivity

As described in § 3, `lfind` has two hyperparameters: (1) number of synthesis results per sketch, and (2) which terms to select for generating sketches. Further, as described in § 4, `lfind` uses Proverbot9001 to rank candidate lemmas and the Myth synthesis engine for term generation. We limit the time spent on each of these tools to efficiently search over the large space of candidate lemmas using available resources. We carry out four separate experiments on the largest benchmark suite (clam, with 140 evaluation locations) to understand `lfind`'s sensitivity to each of these parameters. To quantify the sensitivity of a parameter, in each experiment we vary one parameter while fixing all others.

Number of Synthesis Terms. In the first experiment, we vary the number of synthesis results (k) that we ask of Myth per sketch. We generate sketches from maximal subterms, and use 10s and 12s timeout for Proverbot9001 and Myth respectively. We study the sensitivity to this parameter by varying k to be 5, 15, and 25. Respectively for these settings, `lfind` is successful in 85, 89, and 80 clam evaluation locations. There is a modest 4.7% increase in effectiveness from $k = 5$ to $k = 15$, since there is a large search space of candidate lemmas as k increases. However, there is a significant drop in effectiveness from $k = 15$ to $k = 25$ — as the search space increases, the useful candidates can more easily fail to be highly ranked. Figure 8 plots the total runtime for different k values, and as expected, the median total time increases with increasing k. Median total time of $k = 5$ is 4.4 min (labeled Top 5), while it is 8.0 min and 10.9 min for $k = 15$ and $k = 25$ respectively (labeled Top 15, Top 25). We pick $k = 5$ as the optimal number of synthesis terms for the remaining experiments, since the increase to $k = 15$ has a large time cost and only a modest effectiveness benefit.

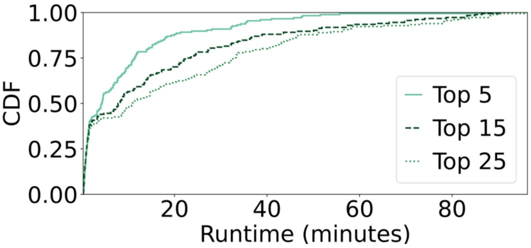

Fig. 8. Total runtime of `lfind` increases when increasing the number of synthesis terms per sketch. Runtime almost doubles when k increases from 5 to 15, while it is 1.3x more when it is increases from 15 to 25.

Proverbot Timeout. In the second experiment, we vary Proverbot9001 timeout to be 5s, 10s, 15s, 30s, and 60s setting $k = 5$ and keeping other parameters similar to the previous experiment. Respectively for these settings, `lfind` is successful in 50, 85, 94, 97, and 102 clam evaluation locations. The tool performs poorly with a 5s timeout, since Proverbot9001 spends the first few seconds in setup, leaving too little time for the actual proof search. Figure 9 plots the runtimes for the 10s, 15s, 30s, and 60s timeout cases. Median total runtime for 10s (labeled 10 seconds) is 4.4 min, while it is only 3.4 min for 15 seconds (labeled 15 seconds), 2.6 min for 30 seconds (labeled 30 seconds), and 3.2 min for 60 seconds (labeled 60 seconds). It is perhaps unintuitive that allowing Proverbot9001 more time leads to lower time overall, but the additional time for Proverbot9001 can allow it to complete a proof that would otherwise not be possible, thereby finding a category Λ_1 lemma sooner. Therefore, we pick 30s as the optimal timeout parameter for Proverbot9001 for experiments in §5.3.

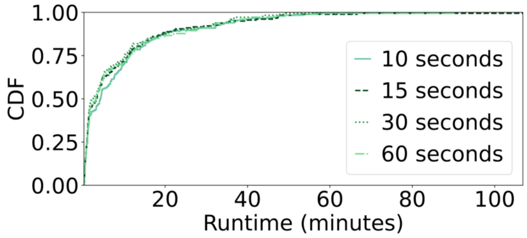

Fig. 9. Median runtime of `lfind` decreases with an increase in Proverbot9001 timeout. While this is unintuitive, this is because the prover is allocated more time per call, enabling it to prove a candidate lemma earlier, which was otherwise not provable using a smaller timeout.

Myth Timeout. The third experiment varies the Myth timeout to be 8s, 12s, and 16s, updating Proverbot9001 timeout to 15s and keeping other parameters similar to the second experiment. Respectively for these settings, `lfind` is successful in 87, 94 and 94 clam evaluation locations. Figure 10 plots the total runtime for these timeout values. Despite increasing timeouts, the total runtime is very similar among the three settings, with a median timeout of 3.1 min, 3.4 min, and

3.5 min for 8s, 12s, and 16s respectively. Therefore, we pick 12s as the optimal timeout parameter for Myth.

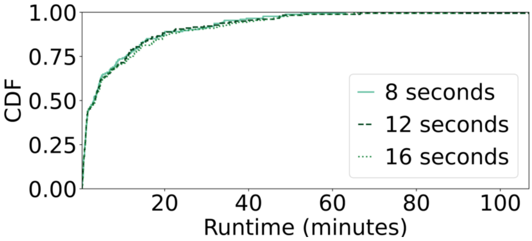

Fig. 10. Median runtime of `lfind` is unaffected when increasing Myth timeout.

Sketch Generation. In this final experiment, we explore two choices for sketch generation, using $k = 5$, 15s timeout for Proverbot9001 and 12s timeout for Myth. We generate synthesis sketches from (1) all subterms of sort `Type` or (2) only from maximal subterms of sort `Type`. To make the use of maximal terms more feasible, for that setting we also use a heuristic that requires the synthesized expression to refer to all generalized variables from the sketch. The use of all terms is successful in 98 evaluation locations while the use of maximal terms is successful in 94 locations. Figure 11 plots the total runtime for these settings, and as expected, the total runtime is more when generating sketches from all subterms compared to only maximal subterms. However, the difference in the median runtime is only one minute. Therefore, we pick *all subterms* as the optimal parameter for sketch generation.

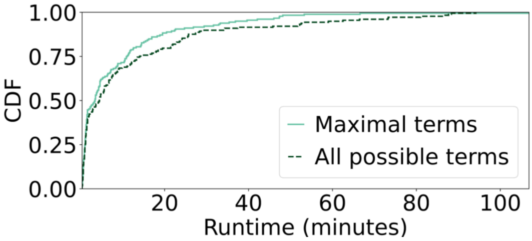

Fig. 11. There is a modest increase in median runtime of `lfind` from 3.4 min to 4.5 min when generating synthesis sketches from maximal terms compared to all terms.

6 RELATED WORK

6.1 Lemma Synthesis

As described in §1, there are a variety of existing approaches to lemma synthesis [Johansson 2019], and they broadly fall into two categories. Many techniques perform rewrites on the target theorem or the current proof state, in order to identify stronger induction hypotheses and helper lemmas. Most common among these is the *generalization* technique [Aubin 1976; Boyer and Moore 1979; Castaing 1985; Dixon and Fleuriot 2003; Hesketh 1992; Hummel 1990; Kaufmann and Moore 1997], whereby selected terms are replaced by fresh variables. Other works go beyond generalizing variables to a broader set of rewrites [Bundy et al. 1993; Johansson et al. 2010; Kapur and Subramaniam 1996; Sonnex et al. 2012]. For example, the *rippling* technique [Bundy et al. 1993] employs a set of rewrite rules in order to make the current goal match the induction hypothesis.

The other category synthesizes candidate lemmas from a grammar using bottom-up enumeration [Claessen et al. 2013, 2010; Heras et al. 2013; Johansson et al. 2011, 2014; Montaño-Rivas et al. 2012; Reynolds and Kuncak 2015; Yang et al. 2019]. Candidate lemmas are typically filtered by searching for counterexamples [Claessen and Hughes 2000], and in many systems an automated prover is used to try to prove the remaining candidates. Closest to our work is AdtInd [Yang et al. 2019], which employs bottom-up enumeration in order to search for candidate lemmas in the context of an automated prover for abstract datatypes. Like `lfind`, AdtInd leverages both generalization and sketches (which they call *templates*) for synthesis, but it is unclear how generalizations are chosen and the sketches are user-provided. Heras *et al.* also combine enumeration with lemma sketches [Heras et al. 2013], but the sketches are automatically learned from a set of existing theorems.

`lfind`'s key innovation over these prior works is showing how to reduce the problem of lemma synthesis to a form of *data-driven* program synthesis. Versus the first category of approaches, `lfind` explores a wider space of potential lemmas via grammar-based synthesis and can leverage off-the-shelf program synthesizers. Versus the second category of approaches, `lfind` generates candidates that are directly targeted toward the current goal, which is critical in an interactive setting. However, our approach borrows several techniques from these prior works. First, `lfind` also employs generalization, but it is used not only to directly produce candidate lemmas but also as the basis for producing sketches for program synthesis. Second, `lfind` employs counterexample search to filter candidates, which has been previously used for filtering in both of the earlier approaches [Chamarthi et al. 2011; Claessen et al. 2010]. Third, `lfind` also employs automated provers, though due to the interactive setting we use them to rank rather than verify candidates.

6.2 Data-Driven Invariant Inference

Data-driven invariant inference has been widely used for various software engineering tasks, at least since Ernst's dissertation on inferring likely program invariants from data [Ernst 2000]. In this approach, data about concrete program executions is used to generate *positive* and/or *negative* examples, and the goal is to synthesize a predicate that separates these two sets of examples. Recently these techniques have become state of the art for automated program specification and verification [Astorga et al. 2019; Ezudheen et al. 2018; Garg et al. 2014, 2016; Padhi et al. 2016; Zhu et al. 2018]. For example, prior work has shown how to generate examples for data-driven synthesis of *loop invariants* that are sufficient to prove that a function meets its specification [Garg et al. 2014, 2016; Padhi et al. 2016]. Closest to our work is the Hanoi tool [Miltner et al. 2020], which infers likely *representation invariants* to aid users of interactive theorem provers in proving that a data structure implementation meets its specification.

As described in Section 1, the existing data-driven verification techniques fundamentally exploit the specific kind of invariant being targeted, which has a clear logical specification over a fixed set of variables. This enables a natural approach based on CEGIS [Solar-Lezama 2009] for both generating examples and verifying candidate invariants. Our setting of lemma synthesis is more general and poses a challenge for data-driven inference, as we lack both a fixed set of variables for the lemma and clear criteria upon which to classify examples as positive or negative. Hence, we have devised a new reduction to data-driven program synthesis: `lfind` produces sketches from generalizations of the goal state and generates examples for synthesis using the heuristic that a synthesized term should behave consistently to the term that it replaces. We have also developed new approaches to filtering and ranking lemma candidates, to address the lack of clear success criteria in our setting.

6.3 Automated Proofs for Interactive Theorem Provers

A variety of tools exist for automatically generating proofs in interactive settings, both in Coq and other languages. Recent techniques use a form of machine learning, for example a neural network, to guide a heuristic proof search, given a set of proof tactics as well as a set of existing lemmas/theorems [Bansal et al. 2019; First et al. 2020; Gauthier et al. 2017; Huang et al. 2019; Paliwal et al. 2020; Sanchez-Stern et al. 2020; Yang and Deng 2019]. Another class of techniques serialize the proof context into a format for input to an external automated solver and then serialize the resulting proof back into the interactive theorem prover [Blanchette et al. 2011; Czajka and Kaliszyk 2018; Kaliszyk and Urban 2015a,b].

Our contribution is orthogonal to these works, which do not perform lemma synthesis. For example, while the machine-learning-based approaches leverage existing lemmas as part of the proof search, they will fail if the existing lemmas are not sufficient. As we showed in §5.3, `lfind` can improve the capabilities of Proverbot9001 [Sanchez-Stern et al. 2020], a state-of-the-art automated prover for Coq based on neural networks, synthesizing lemmas that allow it to prove goals that it otherwise could not. `lfind` uses Proverbot9001 to rank candidate lemmas and produce proofs for ones that are fully automatable. However, our approach is independent of the particular prover used and so for example could instead employ a solver-based prover like CoqHammer [Czajka and Kaliszyk 2018] or even employ multiple provers to leverage their relative strengths.

7 CONCLUSION

In this paper, we developed a new approach to lemma synthesis for interactive proofs that is both goal-directed and expressive. The key technical contribution is a new reduction from the general lemma synthesis problem to a data-driven program synthesis problem. The approach leverages the information available in a given stuck proof state in multiple ways: sampling variable valuations for example generation, generalizing the state to systematically introduce new variables for synthesis, and deriving synthesis sketches from the current goal. We also describe several techniques for filtering and ranking candidate lemmas, which are critical in an interactive setting. While the problem of lemma synthesis is hard in general, the experimental evaluation of our resulting tool `lfind` demonstrates the promise of the approach and quantifies the benefits over other approaches.

ACKNOWLEDGMENTS

The authors would like to thank Murali Ramanujam for helpful feedback on this research. This work is supported in part by DARPA grant HR0011-22-9-0063, NSF grants CCF-1837129 and CCF-2220891, and a PhD fellowship from Google Research.

REFERENCES

1995. Circuits. https://github.com/coq-contribs/circuits.

2003. Coq-of-Ocaml. https://github.com/foobar-land/coq-of-ocaml.

Aws Albarghouthi, Sumit Gulwani, and Zachary Kincaid. 2013. Recursive program synthesis. In *International Conference on Computer Aided Verification*. Springer, 934–950.

Angello Astorga, P Madhusudan, Shambwaditya Saha, Shiyu Wang, and Tao Xie. 2019. Learning stateful preconditions modulo a test generator. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 775–787.

R Aubin. 1976. Mechanising Structural Induction. (1976).

Kshitij Bansal, Sarah M. Loos, Markus N. Rabe, Christian Szegedy, and Stewart Wilcox. 2019. HOList: An Environment for Machine Learning of Higher-Order Theorem Proving (extended version). *CoRR* abs/1904.03241 (2019). arXiv:1904.03241 http://arxiv.org/abs/1904.03241

Jasmin Christian Blanchette, Sascha Böhme, and Lawrence C. Paulson. 2011. Extending Sledgehammer with SMT Solvers. In *Automated Deduction – CADE-23*, Nikolaj Bjørner and Viorica Sofronie-Stokkermans (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 116–130.

Robert S Boyer and J Strother Moore. 1979. A Computational Logic. *ACM Monograph Series* (1979).

Alan Bundy, Andrew Stevens, Frank Van Harmelen, Andrew Ireland, and Alan Smaill. 1993. Rippling: A heuristic for guiding inductive proofs. *Artificial intelligence* 62, 2 (1993), 185–253.

Jacqueline Castaing. 1985. How to Facilitate the Proof of Theorems by Using the Induction-matching, and by Generalization. In *IJCAI*.

Harsh Raju Chamarthi, Peter C. Dillinger, Matt Kaufmann, and Panagiotis Manolios. 2011. Integrating Testing and Interactive Theorem Proving. In *Proceedings 10th International Workshop on the ACL2 Theorem Prover and its Applications, ACL2 2011, Austin, Texas, USA, November 3-4, 2011 (EPTCS, Vol. 70)*, David Hardin and Julien Schmaltz (Eds.). 4–19. http://arxiv.org/abs/1110.4473

Adam Chlipala. 2013. *Certified Programming with Dependent Types: A Pragmatic Introduction to the Coq Proof Assistant.* MIT Press.

Koen Claessen and John Hughes. 2000. QuickCheck: a lightweight tool for random testing of Haskell programs. In *5th ACM SIGPLAN International Conference on Functional Programming (ICFP) (ICFP).* ACM, 268–279. http://www.eecs.northwestern.edu/~robby/courses/395-495-2009-fall/quick.pdf

Koen Claessen, Moa Johansson, Dan Rosén, and Nicholas Smallbone. 2013. Automating Inductive Proofs Using Theory Exploration. In *Automated Deduction - CADE-24 - 24th International Conference on Automated Deduction, Lake Placid, NY, USA, June 9-14, 2013. Proceedings (Lecture Notes in Computer Science, Vol. 7898)*, Maria Paola Bonacina (Ed.). Springer, 392–406.

Koen Claessen, Nicholas Smallbone, and John Hughes. 2010. QuickSpec: Guessing Formal Specifications Using Testing. In *TAP@TOOLS (Lecture Notes in Computer Science, Vol. 6143)*, Gordon Fraser 0001 and Angelo Gargantini (Eds.). Springer, 6–21.

Łukasz Czajka and Cezary Kaliszyk. 2018. Hammer for Coq: Automation for Dependent Type Theory. *Journal of Automated Reasoning* 61, 1 (01 Jun 2018), 423–453. https://doi.org/10.1007/s10817-018-9458-4

Leonardo de Moura, Soonho Kong, Jeremy Avigad, Floris van Doorn, and Jakob von Raumer. 2015. The Lean Theorem Prover (System Description). In *Automated Deduction - CADE-25*, Amy P. Felty and Aart Middeldorp (Eds.). Springer International Publishing, Cham, 378–388.

Lucas Dixon and Jacques Fleuriot. 2003. IsaPlanner: A prototype proof planner in Isabelle. In *International Conference on Automated Deduction.* Springer, 279–283.

Michael D. Ernst. 2000. *Dynamically Discovering Likely Program Invariants.* Ph.D. University of Washington Department of Computer Science and Engineering, Seattle, Washington.

P Ezudheen, Daniel Neider, Deepak D'Souza, Pranav Garg, and P Madhusudan. 2018. Horn-ICE learning for synthesizing invariants and contracts. *Proceedings of the ACM on Programming Languages* 2, OOPSLA (2018), 1–25.

John K. Feser, Swarat Chaudhuri, and Isil Dillig. 2015. Synthesizing Data Structure Transformations from Input-Output Examples. *SIGPLAN Not.* 50, 6 (jun 2015), 229–239. https://doi.org/10.1145/2813885.2737977

Jean-Christophe Filliâtre, Hugo Herbelin, Bruno Barras, Bruno Barras, Samuel Boutin, Eduardo Giménez, Samuel Boutin, Gérard Huet, César Muñoz, Cristina Cornes, Cristina Cornes, Judicaël Courant, Judicael Courant, Chetan Murthy, Chetan Murthy, Catherine Parent, Catherine Parent, Christine Paulin-mohring, Christine Paulin-mohring, Amokrane Saibi, Amokrane Saibi, Benjamin Werner, and Benjamin Werner. 1997. *The Coq Proof Assistant - Reference Manual Version 6.1.* Technical Report.

Emily First, Yuriy Brun, and Arjun Guha. 2020. TacTok: Semantics-Aware Proof Synthesis. In *Object-oriented Programming, Systems, Languages, and Applications.*

Jonathan Frankle, Peter-Michael Osera, David Walker, and Steve Zdancewic. 2016. Example-Directed Synthesis: A Type-Theoretic Interpretation. *SIGPLAN Not.* 51, 1 (jan 2016), 802–815. https://doi.org/10.1145/2914770.2837629

Emilio Jesús Gallego Arias, Karl Palmskog, and Vasily Pestun. 2020. SerAPI:Machine-Friendly, Data-Centric Serialization for Coq. https://github.com/ejgallego/coq-serapi.

Pranav Garg, Christof Löding, P Madhusudan, and Daniel Neider. 2014. ICE: A robust framework for learning invariants. In *International Conference on Computer Aided Verification.* Springer, 69–87.

Pranav Garg, Daniel Neider, P. Madhusudan, and Dan Roth. 2016. Learning invariants using decision trees and implication counterexamples. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*, Rastislav Bodík and Rupak Majumdar (Eds.). ACM, 499–512. http://dl.acm.org/citation.cfm?id=2837614

Thibault Gauthier, Cezary Kaliszyk, and Josef Urban. 2017. TacticToe: Learning to Reason with HOL4 Tactics. In *LPAR-21. 21st International Conference on Logic for Programming, Artificial Intelligence and Reasoning (EPiC Series in Computing, Vol. 46)*, Thomas Eiter and David Sands (Eds.). EasyChair, 125–143. https://doi.org/10.29007/ntlb

Jónathan Heras, Ekaterina Komendantskaya, Moa Johansson, and Ewen Maclean. 2013. Proof-Pattern Recognition and Lemma Discovery in ACL2. In *Logic for Programming, Artificial Intelligence, and Reasoning - 19th International Conference, LPAR-19, Stellenbosch, South Africa, December 14-19, 2013. Proceedings (Lecture Notes in Computer Science, Vol. 8312)*, Kenneth L. McMillan, Aart Middeldorp, and Andrei Voronkov (Eds.). Springer, 389–406.

Jane Thurmann Hesketh. 1992. *Using Middle-Out Reasoning to Guide Inductive Theorem Proving.* Ph. D. Dissertation. University of Edinburgh.

Daniel Huang, Prafulla Dhariwal, Dawn Song, and Ilya Sutskever. 2019. GamePad: A Learning Environment for Theorem Proving. In *7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019.*

OpenReview.net. https://openreview.net/forum?id=r1xwKoR9Y7

B Hummel. 1990. Generation of induction axioms and generalisation. (1990).

Andrew Ireland and Alan Bundy. 1996. Productive use of failure in inductive proof. In *Automated Mathematical Induction*. Springer, 79–111.

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.

Moa Johansson, Lucas Dixon, and Alan Bundy. 2010. Dynamic Rippling, Middle-Out Reasoning and Lemma Discovery. In *Verification, Induction, Termination Analysis - Festschrift for Christoph Walther on the Occasion of His 60th Birthday (Lecture Notes in Computer Science, Vol. 6463)*, Simon Siegler and Nathan Wasser (Eds.). Springer, 102–116.

Moa Johansson, Lucas Dixon, and Alan Bundy. 2011. Conjecture Synthesis for Inductive Theories. *J. Autom. Reason* 47, 3 (2011), 251–289.

Moa Johansson, Dan Rosén, Nicholas Smallbone, and Koen Claessen. 2014. Hipster: Integrating Theory Exploration in a Proof Assistant. In *Intelligent Computer Mathematics - International Conference, CICM 2014, Coimbra, Portugal, July 7-11, 2014. Proceedings (Lecture Notes in Computer Science, Vol. 8543)*, Stephen M. Watt, James H. Davenport, Alan P. Sexton, Petr Sojka, and Josef Urban (Eds.). Springer, 108–122.

Cezary Kaliszyk and Josef Urban. 2015a. HOL(y)Hammer: Online ATP Service for HOL Light. *Mathematics in Computer Science* 9, 1 (01 Mar 2015), 5–22. https://doi.org/10.1007/s11786-014-0182-0

Cezary Kaliszyk and Josef Urban. 2015b. MizAR 40 for Mizar 40. *Journal of Automated Reasoning* 55, 3 (01 Oct 2015), 245–256. https://doi.org/10.1007/s10817-015-9330-8

Deepak Kapur and Mahadevan Subramaniam. 1996. Lemma Discovery in Automated Induction. In *Automated Deduction - CADE-13, 13th International Conference on Automated Deduction, New Brunswick, NJ, USA, July 30 - August 3, 1996, Proceedings (Lecture Notes in Computer Science, Vol. 1104)*, Michael A. McRobbie and John K. Slaney (Eds.). Springer, 538–552.

Matt Kaufmann and J S. Moore. 1997. An Industrial Strength Theorem Prover for a Logic Based on Common Lisp. *IEEE Transactions on Software Engineering* 23, 4 (April 1997), 203–213.

Justin Lubin, Nick Collins, Cyrus Omar, and Ravi Chugh. 2020. Program Sketching with Live Bidirectional Evaluation. *Proc. ACM Program. Lang.* 4, ICFP, Article 109 (aug 2020), 29 pages. https://doi.org/10.1145/3408991

Anders Miltner, Adrian Trejo Nuñez, Ana Brendel, Swarat Chaudhuri, and Isil Dillig. 2022. Bottom-up Synthesis of Recursive Functional Programs Using Angelic Execution. *Proc. ACM Program. Lang.* 6, POPL, Article 21 (jan 2022), 29 pages. https://doi.org/10.1145/3498682

Anders Miltner, Saswat Padhi, David Walker, and Todd Millstein. 2020. Data-driven inference of representation invariants. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM.

Omar Montaño-Rivas, Roy L. McCasland, Lucas Dixon, and Alan Bundy. 2012. Scheme-based theorem discovery and concept invention. *Expert Syst. Appl* 39, 2 (2012), 1637–1646.

Peter-Michael Osera and Steve Zdancewic. 2015. Type-and-example-directed program synthesis. *ACM SIGPLAN Notices* 50, 6 (2015), 619–630.

Saswat Padhi, Rahul Sharma, and Todd Millstein. 2016. Data-driven precondition inference with learned features. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, 42–56.

Aditya Paliwal, Sarah M. Loos, Markus N. Rabe, Kshitij Bansal, and Christian Szegedy. 2020. Graph Representations for Higher-Order Logic and Theorem Proving. In *The Thirty-Fourth AAAI Conference on Artificial Intelligence, AAAI 2020, The Thirty-Second Innovative Applications of Artificial Intelligence Conference, IAAI 2020, The Tenth AAAI Symposium on Educational Advances in Artificial Intelligence, EAAI 2020, New York, NY, USA, February 7-12, 2020*. AAAI Press, 2967–2974. https://ojs.aaai.org/index.php/AAAI/article/view/5689

Zoe Paraskevopoulou, Aaron Eline, and Leonidas Lampropoulos. 2022. Computing correctly with inductive relations. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. 966–980.

Zoe Paraskevopoulou, Cătălin Hriţcu, Maxime Dénès, Leonidas Lampropoulos, and Benjamin Pierce. 2015. Foundational Property-Based Testing, Vol. 9236. https://doi.org/10.1007/978-3-319-22102-1_22

Lawrence C. Paulson. 1993. Natural Deduction as Higher-Order Resolution. *CoRR* cs.LO/9301104 (1993). http://arxiv.org/abs/cs.LO/9301104

Andrew Reynolds and Viktor Kuncak. 2015. Induction for SMT Solvers. In *VMCAI (Lecture Notes in Computer Science, Vol. 8931)*, Deepak D'Souza, Akash Lal, and Kim Guldstrand Larsen (Eds.). Springer, 80–98.

Alex Sanchez-Stern, Yousef Alhessi, Lawrence K. Saul, and Sorin Lerner. 2020. Generating correctness proofs with neural networks. In *Proceedings of the 4th ACM SIGPLAN International Workshop on Machine Learning and Programming Languages, MAPL@PLDI 2020, London, UK, June 15, 2020*, Koushik Sen and Mayur Naik (Eds.). ACM, 1–10. https://doi.org/10.1145/3394450.3397466

Taro Sekiyama, Akifumi Imanishi, and Kohei Suenaga. 2017. Towards Proof Synthesis Guided by Neural Machine Translation for Intuitionistic Propositional Logic. *CoRR* abs/1706.06462 (2017). arXiv:1706.06462 http://arxiv.org/abs/1706.06462

Armando Solar-Lezama. 2009. The Sketching Approach to Program Synthesis. In *Programming Languages and Systems, 7th Asian Symposium, APLAS 2009, Seoul, Korea, December 14-16, 2009. Proceedings (Lecture Notes in Computer Science, Vol. 5904)*, Zhenjiang Hu (Ed.). Springer, 4–13.

William Sonnex, Sophia Drossopoulou, and Susan Eisenbach. 2012. Zeno: An automated prover for properties of recursive data structures. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 407–421.

Daniel Whalen. 2016. Holophrasm: a neural Automated Theorem Prover for higher-order logic. arXiv:1608.02644 [cs.AI]

Kaiyu Yang and Jia Deng. 2019. Learning to Prove Theorems via Interacting with Proof Assistants. In *Proceedings of the 36th International Conference on Machine Learning, ICML 2019, 9-15 June 2019, Long Beach, California, USA (Proceedings of Machine Learning Research, Vol. 97)*, Kamalika Chaudhuri and Ruslan Salakhutdinov (Eds.). PMLR, 6984–6994. http://proceedings.mlr.press/v97/yang19a.html

Weikun Yang, Grigory Fedyukovich, and Aarti Gupta. 2019. Lemma synthesis for automating induction over algebraic data types. In *International Conference on Principles and Practice of Constraint Programming*. Springer, 600–617.

He Zhu, Stephen Magill, and Suresh Jagannathan. 2018. A data-driven CHC solver. *ACM SIGPLAN Notices* 53, 4 (2018), 707–721.