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 65% 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.

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 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. 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; Reynolds and Kuncak 2015; Yang et al. 2019]. For example, HipSpec [Claessen et al. 2013] enumerates many candidate lemmas and attempts to prove them with 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. 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; Paraskevopoulos 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.

We have implemented our approach as a tactic for Coq and call the resulting tool lfind. Coq users can invoke lfind as a tactic at any point in their proof, and it will produce a set of ranked
Lemma candidates. Lemma synthesis in lfind is targeted for use in proving properties of programs (as opposed to other uses of interactive theorem proving, such as formalizing mathematics). This is a common use case for Coq, aligns with the focus of prior lemma synthesis approaches, and is compatible with the data-driven style that our tool employs. 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 144/222 of these locations, with a median runtime of 4.8 minutes (see §5.3). At 109 of these locations lfind provides a full automated proof of the synthesized lemma and the goal; at the other 35 locations lfind produces a ranked list of lemma candidates where the human-written lemma is in the top 10. We also show that our approach significantly outperforms the prior technique of generalization as well as a version of lfind that employs type-guided 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 in the 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’ll 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 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:
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.

Proof.
induction l.
- reflexivity.
- simpl. (* I'm stuck! *)

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

\[ n : \text{nat} \]
\[ l : \text{lst} \]
\[ \text{IHl} : \text{rev} \ (\text{rev} \ l) = l \]
\[ \text{-----------------------------} \]
\[ \text{rev} \ (\text{app} \ (\text{rev} \ l) \ (\text{Cons} \ n \ \text{Nil})) = \text{Cons} \ n \ l \]

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

\( (\Lambda_1) \) Lemma lem1: forall l n, 
\[ \text{rev} \ (\text{app} \ l \ (\text{Cons} \ n \ \text{Nil})) = \text{Cons} \ n \ (\text{rev} \ l). \]

\( (\Lambda_2) \) Lemma lem2: forall l1 l2, 
\[ \text{rev} \ (\text{app} \ l1 \ l2) = \text{app} \ (\text{rev} \ l1) \ (\text{rev} \ l2). \]

\( (\Lambda_2) \) Lemma lem3: forall l1 l2, 
\[ \text{rev} \ (\text{app} \ (\text{rev} \ l1) \ 12) = \text{app} \ (\text{rev} \ l2) \ l1. \]

Each lemma is bucketed into one of three categories (\( \Lambda_1 \), \( \Lambda_2 \), or \( \Lambda_3 \)), and the categories are presented to the user in that order. \( \Lambda_1 \) lemmas are those in which 1find 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, 1find 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 \( \Lambda_1 \) lemma; the full proof of the theorem rev\_rev using lem1 is shown in Figure 3.

\( \Lambda_2 \) lemmas are those that are sufficient to automatically prove the original goal, but PROVERBOT9001 cannot automatically prove the auxiliary lemma. 1find indicates that the second and third lemmas in the above listing are \( \Lambda_2 \) 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
Lemma lem1: \(\forall l\ n, \text{rev (app } l\ \text{(Cons } n\ \text{Nil})) = \text{Cons } n\ \text{(rev } l)\).

Proof.
1. intros.
2. induction \(l\).
3. simpl.
4. eauto.
5. simpl.
6. rewrite IHl.
7. eauto.
8. Qed.

Lemma rev_rev: \(\forall l, \text{rev (rev } l) = l\).

Proof.
1. induction \(l\).
2. - reflexivity.
3. - simpl. rewrite <- IHl. unfold app.
4. rewrite IHl. rewrite lem1. rewrite IHl. easy.
5. Qed.

Fig. 3. A full proof provided by lfind.

to the original rev_rev lemma when \(l2\) is Nil. \(\lambda3\) 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. To begin, we produce all generalizations of the goal at the point where lfind is invoked, which are formed by replacing one or more terms within the goal with fresh variables [Kaufmann and Moore 1997]. 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 \(\text{rev } l\) with a fresh variable \(l1\) of type \(\text{lst}\) produces the following generalization:

\(\forall l\ n\ l1, \text{rev (app } l1\ \text{(Cons } n\ \text{Nil})) = \text{Cons } n\ l\).

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 \( n \) 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):

\[
\forall l_1 n. \text{rev} (\text{app} l_1 (\text{Cons} n \text{Nil})) = □.
\]

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 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 \( l_1 \) replaced \( \text{rev} l \) from the original goal. Hence we evaluate \( \text{rev} l \) for each of our three examples to produce the following \( l_1 \) 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 \( \text{Cons} n l \). 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 \((l_1, n)\) pairs to the expected output value of the term to be synthesized:

\[
\begin{align*}
(\[] , 4) & \mapsto [4] \\
([1; 0], 2) & \mapsto [2; 0; 1] \\
([1; 2], 1) & \mapsto [1; 2; 1]
\end{align*}
\]

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 \( \text{lst} \star \text{nat} \rightarrow \text{lst} \)) and a grammar to use for term generation. \text{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 \text{Nil} and \text{Cons} and the functions \text{app} and \text{rev}. One term that the synthesizer generates from these inputs is \( \text{Cons} \ n \ (\text{rev} \ 11) \). Substituting this expression into the hole in our sketch yields exactly the lemma \text{lem1} shown earlier, which enables a fully automated proof of the original lemma.

6
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 \( \forall l, \text{rev } l = \text{rev } l \). 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 \( \forall n \ l, \text{rev } (\text{rev } (\text{Cons } n \ l)) = \text{Cons } n \ l \), 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: \( \Lambda_1 \) lemmas that are automatically provable and enable automatic proof of the user’s stuck proof state; \( \Lambda_2 \) lemmas that are not automatically provable but enable automatic proof of the user’s stuck proof state; and the remaining \( \Lambda_3 \) 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 \( \Lambda_1 \), which yields a fully automated proof, is \( \text{lem1} \) shown earlier, so it is ranked first. Lemmas \( \text{lem2} \) and \( \text{lem3} \) are the smallest lemmas in category \( \Lambda_2 \) 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 \( \Psi \), which is a tuple \( \langle \mathcal{H}, g, \Gamma, D \rangle \), where \( \mathcal{H} \) is a set of logical formulas that are the current hypotheses, \( g \) is a logical formula that is the current goal, \( \Gamma \) is a type environment for all free variables in \( \mathcal{H} \) and \( g \), and \( D \) 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
where we define the generalization we assume the existence of both automated theorem provers and disprovers. A prover $t$ where $e = \langle x_1 : v_1, \ldots, x_n : v_n \rangle$ such that $e$ is a model of $\mathcal{H} \rightarrow g$, denoted $e \models_\mathcal{H} \mathcal{H} \rightarrow g$. 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 $G$, consisting of typed constants and functions; a type signature $\tau_1 \rightarrow \tau_2$; and input-output examples of the form $(v_1, v_2)$, where $v_1$ has type $\tau_1$ and $v_2$ has type $\tau_2$. This synthesizer returns a list of functions $f$ of type $\tau_1 \rightarrow \tau_2$ in the grammar $G$ 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 $\text{SAMPLE}(\Psi)$ that produces a set of samples. Last, we assume the existence of both automated theorem provers and disprovers. A prover $R(\phi, \bar{\phi}, D)$ 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 $\text{VALID}$ or $\text{DONT KNOW}$. A disprover $C(\phi, D)$ searches for concrete counterexamples to $\phi$ and returns either $\text{INVALID}$ or $\text{DON’T KNOW}$.

### 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: $G$) Given a goal $g$, a type environment $\Gamma$, and a set $T = \{t_1, \ldots, t_n\}$, we define the generalization of $g$ with respect to $\Gamma$ and $T$, denoted $G(g, \Gamma, T)$, as the tuple $\langle g', \Sigma \rangle$, where $\Sigma = \langle x_1 \mapsto t_1, \ldots, x_n \mapsto t_n \rangle$ 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 $\Gamma$, and $g' = g[t_i \mapsto x_i]$.

`find` 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 $\square$.

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

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 $\Sigma = \langle x'_1 \mapsto t_1, \ldots, x'_m \mapsto t_m \rangle$, and term $t_s$, denoted $\text{IO}(e, \Sigma, t_s)$, is defined as

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

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

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 $\Psi = \langle \mathcal{H}, g, \Gamma, D \rangle$, a set of terms $T = \{t_1, \ldots, t_m\}$ for generalization, and a sketch term $t_s$, we produce
We invoke the synthesizer with these inputs and ask for up to $t$ functions. This is a subtle notion. For example, it is not a form of logical implication, and it is not a form of logical implication. 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, it’s common for there to be many duplicate lemmas. 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 `simp` tactic and then comparing the results for syntactic equivalence. Second, we use the disprover $C$ to search for counterexamples, filtering out any candidate $\phi$ such that $C(\phi, D) = \text{INVALID}$. 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 $\leq$, which says that one lemma is an instantiation of (or equivalent to) another, defined as follows:

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

1. intros. apply l2. Qed.
We then filter out any candidate lemma that is \( \leq \) 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 \( \phi \) we use the prover to determine if the candidate can be used to automatically prove the goal \( g \leftarrow \mathcal{R}(g, \{H, \phi\}, D) \) — and whether the candidate itself is automatically provable — \( \mathcal{R}(\phi, \emptyset, D) \). Based on the results we partition the lemmas into three groups, \( \Lambda_1 \), \( \Lambda_2 \), and \( \Lambda_3 \). The \( \Lambda_1 \) group contains the lemmas for which both calls to \( \mathcal{R} \) return \texttt{Valid}, meaning that we have obtained a fully automated proof of the user’s original goal. The \( \Lambda_2 \) group contains the lemmas for which the first call to \( \mathcal{R} \) return \texttt{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 \( \Lambda_3 \) 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 \texttt{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, \texttt{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 \texttt{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 \texttt{lfind} is 3.2 KLOC of OCaml code.

#### 4.1 Example Generation

To synthesize candidate lemmas, our approach relies on a \texttt{SAMPLE} function that can produce samples for the variables in the stuck state \( g \). We leverage \texttt{QUICKCHICK} [Paraskevopoulou et al. 2015], a state-of-the-art property-based randomized tester for Coq, for this purpose. While \texttt{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, \texttt{lfind} first generates the following Coq code, which enables the usage of \texttt{QUICKCHICK} for that type:

```coq
1  Derive Show for T.
2  Derive Arbitrary for T.
3  Instance Dec_Eq_T : Dec_Eq T.
```
Fig. 5. Given a stuck goal, lfind implements generalization, synthesis, filtering and ranking in conjunction with existing tools to generate candidate lemmas.

4 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: \( H \rightarrow g \). We also create a function collect_data whose input type \( V \) is the tuple of the types of all free variables in \( H \rightarrow g \). The function logs the input values to a file and returns the valuation of \( H \rightarrow g \) on those input values:

1 Definition collect_data (n:V) :=
2 in let _ := print_to_file (show n)
3 in stuckState n.
4 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 $\Lambda_1$ 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 human-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.
Table 1. Results

<table>
<thead>
<tr>
<th></th>
<th>CLAM</th>
<th>Full Adder</th>
<th>Compiler</th>
<th>LIA</th>
</tr>
</thead>
<tbody>
<tr>
<td>Setup</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td># Theorems</td>
<td>86</td>
<td>40</td>
<td>1</td>
<td>9</td>
</tr>
<tr>
<td># Evaluation Locations</td>
<td>140</td>
<td>62</td>
<td>1</td>
<td>19</td>
</tr>
<tr>
<td>Results</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td># fully proven lemma and goal</td>
<td>68</td>
<td>34</td>
<td>0</td>
<td>7</td>
</tr>
<tr>
<td># else human match in top 1</td>
<td>14</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td># else human match in top 5</td>
<td>9</td>
<td>1</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td># else human match in top 10</td>
<td>6</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td># else more general than human lemma in top 1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Summary</td>
<td>98/140</td>
<td>35/62</td>
<td>1/1</td>
<td>10/19</td>
</tr>
</tbody>
</table>

- **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 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 100 minutes.
the large search space, in § 5.3 we show that the tool is performant with a median runtime of only 4.8 min. The tool has a 12s timeout for each call MYTH and a 15s 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 $\text{lfind}$ at an evaluation location to be successful in three scenarios. First, we say that $\text{lfind}$ 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 $\text{lfind}$ has produced a complete proof for the user. Second, we say that $\text{lfind}$ is successful if a lemma that matches the human-provided lemma is ranked highly (top-10) by the tool. Third, we say that $\text{lfind}$ is successful if a lemma that is more general than the human-provided lemma is ranked highly by the tool. We use the $\leq$ 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 $l \leq h$ and $h \leq l$, and we say that $l$ is more general than $h$ if $h \leq l$ 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 $\text{lfind}$ 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 $\text{lfind}$ succeeds in 144 (64.9%) of the 222 evaluation locations across all benchmarks. Further, as shown in the third row of the table, in 109 (75.7%) of these successful 144 locations, $\text{lfind}$ 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 35 successful locations. Notably, for 15 of these evaluation locations, the top-ranked candidate lemma produced by $\text{lfind}$ 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.

**Examples.** Table 2 shows examples of lemmas synthesized by $\text{lfind}$ 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 $\text{exp}$ defines the source language of arithmetic expressions. $\text{evalExp}$ function evaluates the programs in this language. The target language’s instructions are of type $\text{instr}$, which are executed on a stack machine. The function $\text{execI}$ takes an instruction and a stack (represented as a list of $\text{nats}$) and returns an updated stack, and $\text{execIs}$ uses this function to execute a list of instructions. Finally, the $\text{compiler}$ 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]. $\text{lfind}$ 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 $\Lambda_2$ and $\Lambda_3$, and the top-ranked candidate
Table 2. A sample of `lfind` synthesized lemmas and their associated rank and category.

<table>
<thead>
<tr>
<th>#</th>
<th>Original Theorem</th>
<th><code>lfind</code> Synthesized Lemma</th>
<th>(\Lambda)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>Theorem <code>correct_compilation</code> :∀ (e:exp), execIs (compile e) Nil = (evalExp e) :: Nil.</td>
<td>Lemma <code>lem1</code> :∀ (e: exp) (l: list instr) (s: list nat), execIs (compile e ++ l) s = execIs l (evalExp e :: s).</td>
<td>(\Lambda_2)</td>
</tr>
<tr>
<td>2.</td>
<td>Theorem <code>BV_full_adder_nil_ok</code> :∀ (v:BoolList) (cin:bool), BV_to_nat (BV_full_adder v Nil cin) = BV_to_nat v + Bool_to_nat cin.</td>
<td>Lemma <code>lem1</code> :∀ (l: BoolList), BV_to_nat (BV_full_adder_sum l Nil false ++ BV_full_adder_carry l Nil false::Nil) = BV_to_nat l.</td>
<td>(\Lambda_1)</td>
</tr>
<tr>
<td>3.</td>
<td>Theorem <code>app_revflat</code> :∀ (x:tree) (y:list nat), (revflat x) ++ y = qrevaflat x y.</td>
<td>Lemma <code>lem10</code> :∀ (l l1 l2:list nat), (l ++ l1) ++ l2 = l ++ (l1 ++ l2).</td>
<td>(\Lambda_3)</td>
</tr>
<tr>
<td>4.</td>
<td>Theorem <code>queue_push</code> :∀ (q:queue) (n:nat), qlen (qpush q n) = 1 + (qlen q).</td>
<td>Lemma <code>lem1</code> :∀ (l l1:list nat), len l + len l1 = len (l ++ rev l1).</td>
<td>(\Lambda_2)</td>
</tr>
<tr>
<td>5.</td>
<td>Theorem <code>qreva_qreva</code> :∀ (x:list nat), (qreva (qreva x Nil) Nil) = x.</td>
<td>Lemma <code>lem9</code> :∀ (n:nat) (l:list nat), qreva (append l n::Nil) Nil = n::(qreva l Nil).</td>
<td>(\Lambda_2)</td>
</tr>
<tr>
<td>6.</td>
<td>Theorem <code>rotate_len</code> :∀ (x:list nat), rotate (len x) x = x.</td>
<td>Lemma <code>lem2</code> :∀ (l l1:list nat), rotate (len l) 1 ++ l1 = l1 ++ 1.</td>
<td>(\Lambda_2)</td>
</tr>
<tr>
<td>7.</td>
<td>Theorem <code>add_even</code> :∀ (x y:nat), even(x+y) = even(y+x).</td>
<td>Lemma <code>lem1</code> :∀ (n x:nat), negb (even(n+x)) = even(n+(S x)).</td>
<td>(\Lambda_1)</td>
</tr>
<tr>
<td>8.</td>
<td>Theorem <code>drop_elem</code> :∀ (v w x y: nat) (z:list nat), drop (S v) (drop w (drop x y::z)) = drop v (drop w (drop x z)).</td>
<td>Lemma <code>lem1</code> :∀ (n x:nat) (l:list nat), drop (S x) (drop n l) = drop x (drop (S n) l)).</td>
<td>(\Lambda_1)</td>
</tr>
</tbody>
</table>

in category \(\Lambda_2\), 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 \(l1\), and an arbitrary stack \(l2\).

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 \(\Lambda_1\) 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 \(\Lambda_2\) and \(\Lambda_3\). 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

in category \(\Lambda_2\), 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 \(l1\), and an arbitrary stack \(l2\).

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 \(\Lambda_1\) 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 \(\Lambda_2\) and \(\Lambda_3\). 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
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 4.8 min (shown in the plot where the curve labeled **Total Time** reaches a CDF of 0.50). Recall that lfind produces a full automated proof (category $\Lambda_1$) in 75.7% (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 3.8 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 168 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 112. Further, after Filter 2 there is a median of 17 candidate lemmas. Hence on average, Filter 1 reduced the candidate lemmas by 33.3%, and Filter 2 reduced the remaining candidates by 84.8%.

Finally, as mentioned above even after filtering we are left with a median of 17 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.

**5.4 Comparison with Other Approaches**

In order to understand how our approach compares with other approaches to lemma synthesis, we performed an ablation study in which we compare lfind against versions of it that have certain features disabled. First, we compare against a version of the tool that generates no lemmas, instead simply using a state-of-the-art automated prover to try to complete the proof from the evaluation location (proof context). Second, we compare against a version of lfind that only generates candidate lemmas through generalization, without performing any synthesis. This version of the tool allows us to compare against the commonly used generalization technique [Boyer and Moore 1979; Kaufmann and Moore 1997]. Finally, we compare against a version of lfind that is identical to the original version except that it provides no examples to Myth for synthesis. This change has
the effect of forcing MYTH to do type-guided synthesis, thereby providing a closer comparison with the term enumeration approach to lemma synthesis [Claessen et al. 2013; Yang et al. 2019].

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 22.1% of the evaluation locations. In contrast, with a lemma synthesized by lfind, PROVERBOT9001 can automatically prove 49.1% of the evaluation locations (109 out of 222), and as shown earlier overall lfind provides a useful lemma in 64.9% 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. 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 19.4% of all evaluation locations, as compared with 64.9% of locations for lfind.

Type-guided Synthesis. For this comparison, we use 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 synthesis through the given grammar, Hence this version of lfind is related to the enumeration techniques from prior work on lemma synthesis, like HipSpec [Claessen et al. 2013] and AdtInd [Yang et al. 2019]. This version synthesizes a successful helper lemma according to our success metric in 67 evaluation locations, whereas lfind does so in 109 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.

**Proverbot Timeout.** In the second experiment, we vary PROVERBOT9001 timeout to be 5s, 10s, and 15s, setting $k = 5$ and keeping other parameters similar to the previous experiment. Respectively for these settings, lfind is successful in 85, 89, and 80 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 and 15s 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). 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 $A_1$ lemma sooner. Therefore, we pick 15s as the optimal timeout parameter for PROVERBOT9001 in the remaining experiments.

---

**Fig. 8.** Total runtime of lfind increases when increasing 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.

**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, \texttt{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.

**Sketch Generation.** In this final experiment, we explore two choices for sketch generation, using the optimal choices for other parameters based on the previous experiments. 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.

6 RELATED WORK

6.1 Lemma Synthesis

As described in §1, there are a variety of existing approaches to lemma synthesis, 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 \cite{Aubin1976,Boyer1979,Castaing1985,Dixon2003,Hesketh1992,Hummel1990,Kaufmann1997}, whereby selected terms are replaced by fresh variables. Other works go beyond generalizing variables to a broader set of rewrites \cite{Bundy1993,Johansson2010,Kapur1996,Sonnex2012}. For example, the rippling technique \cite{Bundy1993} 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. QuickSpec \cite{Claessen2010} employs this approach and filters candidates by searching for counterexamples \cite{Claessen2000}. HipSpec \cite{Claessen2013} combines QuickSpec with an automated prover in order to synthesize a set of provably correct lemmas. A similar enumerate-and-filter strategy is used to automate induction in the CVC4 solver \cite{Reynolds2015}. Finally, ADTIND \cite{Yang2019} employs bottom-up enumeration in order to search
for candidate lemmas in the context of an automated prover for abstract datatypes. Notably, like AdtInd, lfind 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.

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.

REFERENCES
Jacqueline Castaing. 1985. How to Facilitate the Proof of Theorems by Using the Induction-matching, and by Generalization. In IJCAI.
ACM SIGPLAN Conference on Programming Languages,


22


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


