Synthesizing Implication Lemmas for Interactive Theorem Proving

ANA BRENDEL, University of California, Los Angeles, USA
AISHWARYA SIVARAMAN, University of California, Los Angeles, USA
TODD MILLSTEIN, University of California, Los Angeles, USA

Interactive theorem provers (ITP) enable programmers to formally verify properties of their software systems. One burden for users of ITPs is identifying the necessary *helper lemmas* to complete a proof, for example those that define key inductive invariants. Existing approaches to *lemma synthesis* for ITPs have limited, if any, support for synthesizing *implications*: lemmas of the form $P_1 \wedge \cdots \wedge P_n \implies Q$. In this paper, we propose a technique and associated tool for synthesizing useful implication lemmas. Our approach employs a form of data-driven invariant inference to explore strengthenings of the current proof state, based on sample valuations of the current goal and assumptions. We have implemented our approach in a Rocq tactic called `dilemma`. We demonstrate its effectiveness in synthesizing necessary helper lemmas for proofs from the Verified Functional Algorithms textbook as well as from prior benchmark suites for lemma synthesis.

CCS Concepts: • **Software and its engineering** → **Formal software verification**.

Additional Key Words and Phrases: Interactive Theorem Prover, Lemma Synthesis, Data-Driven Synthesis

ACM Reference Format:
Ana Brendel, Aishwarya Sivaraman, and Todd Millstein. 2025. Synthesizing Implication Lemmas for Interactive Theorem Proving. *Proc. ACM Program. Lang.* 9, OOPSLA2, Article 353 (October 2025), 25 pages. https://doi.org/10.1145/3763131

1 Introduction

Interactive theorem provers (ITPs) are powerful tools that allow software engineers to prove important properties of their code [13, 17, 43]. While these tools have been used to build real systems that have strong correctness guarantees [27, 35, 36], the time, effort, and expertise required remains a prohibitive barrier to their broad adoption.

The obvious way to lower the cost of using ITPs is through forms of automation. Accordingly there are a variety of tools and techniques for automated proof construction, for example by integrating external automated solvers (e.g., [5, 12, 31, 32]) and recently by using a machine-learning model to predict the next proof tactic (e.g., [18, 19, 33, 46]). However, these approaches have a common limitation: while they make use of lemmas that have already been defined in order to make progress in the current proof, they do not devise new lemmas that may be required and will instead get stuck in such cases. It is not reasonable to expect the user to have already defined all of the lemmas that are required to prove a given theorem. Indeed, determining the required helper lemmas is often one of the hardest parts in a proof; for example, the theorem's induction hypothesis may be too weak, thereby necessitating a stronger lemma that defines the key inductive invariant to be proven.

Authors' Contact Information: Ana Brendel, anabrendel@cs.ucla.edu, University of California, Los Angeles, Los Angeles, California, USA; Aishwarya Sivaraman, University of California, Los Angeles, Los Angeles, California, USA; Todd Millstein, todd@cs.ucla.edu, University of California, Los Angeles, Los Angeles, California, USA.

ACM 2475-1421/2025/10-ART353
https://doi.org/10.1145/3763131

In this paper we present an approach for synthesizing lemmas that are useful for making progress from a given proof state. Standard approaches to this problem generate candidate lemmas through a pre-defined set of rewrite rules on the current proof state [6, 8, 14], which is useful but provides limited expressiveness. A recent approach [48] instead reduces lemma synthesis to a form of data-driven program synthesis (e.g., [39]), whereby variable valuations from the proof state are used to define examples for synthesis. While this technique enables a much broader range of candidate lemmas to be explored, it is restricted to producing *atomic* lemmas related to the current proof goal, i.e., lemmas of the form $\forall \overline{v}.Q(\overline{t})$, where $\overline{v}$ are variables (e.g., `x` and `y`), $\overline{t}$ are terms over those variables (e.g., `2 * x + y`) and Q is a logical proposition over those terms (e.g., $\leq$).

We describe a new approach to lemma synthesis which is data-driven but targets a rich space of *implication lemmas*, i.e., lemmas of the form $\forall \overline{v}.(\bigwedge P_i(\overline{t_i}) \implies Q(\overline{t}))$. In our setting the implicand $Q(\overline{t})$ of a candidate lemma is either the original goal or some generalization of it, and our core problem is one of *implicant synthesis*, determining the implicants that are required for a given implicand. Our approach reduces implicant synthesis to a form of *data-driven invariant synthesis* [16, 20, 21, 37, 40, 53], which synthesizes propositions that respect a given set of positive and negative examples. We describe how to use valuations of the current proof state to derive the examples for synthesis.

Our approach is based on three key ideas. First, it is natural for the implicants in a candidate lemma to be based on the assumptions in the current proof state, but there can be many such assumptions. To reduce the search space for useful implicants, we initially use valuations of the proof state to isolate the assumptions that are most relevant for proving the goal. We identify a minimal set S of assumptions such that every sample valuation either falsifies at least one of the assumptions in S or satisfies the current goal G. While simple, this step already yields a candidate lemma that can be useful in practice, namely $\bigwedge S \implies G$.

However, often a useful lemma must strengthen the current proof state, in order to provide a stronger induction hypothesis. As our second contribution, we explore a space of such strengthenings by *weakening* the relevant assumptions. Intuitively, producing a proposition B that is weaker than an existing assumption A allows the proof to be decomposed into 1) proving that B and other relevant assumptions together imply the goal; and 2) proving that $A \implies B$. We use a form of data-driven synthesis to produce candidate propositions that weaken existing assumptions.

Finally, a commonly used form of lemma synthesis is *generalization*, which strengthens a proof state by replacing specific terms with fresh universally quantified variables [6]. Our approach also performs generalization in order to identify candidate lemmas, and we use sample valuations to identify generalizations that are likely to be valid lemmas. However, in cases where a generalization is not valid (meaning that the assumptions do not imply the goal), rather than simply discarding this lemma candidate, we employ data-driven synthesis to produce an additional implicant to make the lemma valid. Intuitively this process has the effect of re-introducing relevant information from the original proof state that was abstracted away by the generalization process.

We have implemented our approach as a tactic called `dilemma`[1] for the Rocq prover (formerly known as the Coq proof assistant) [45]. Users can invoke the tactic at a point in their proof where they are stuck, and the tool will suggest a ranked list of candidate lemmas to consider. Due to the large search space that it explores, `dilemma` takes on the order of minutes to run (average time of 207 seconds and median time of 100 seconds in our main experiment) and so is not currently suitable for a tight interaction loop with the user. At the same time, without a tool like `dilemma` users are simply on their own to determine what lemma is required in order to make progress in their proof, so it can be well worth the wait.

[1] https://github.com/ana-brendel/dilemma

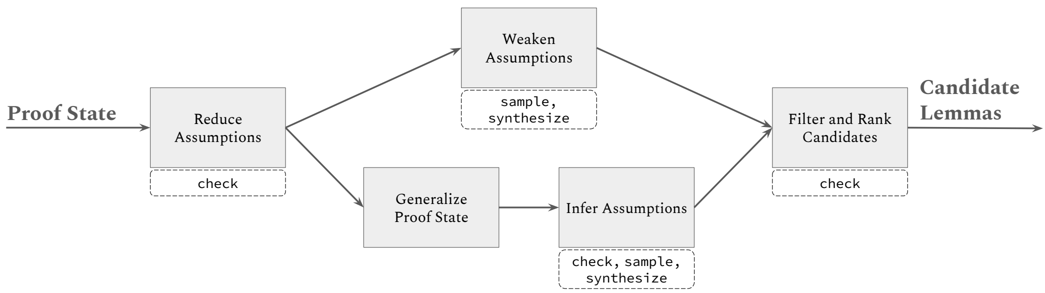

Fig. 1. The overall architecture of `dilemma`. The approach depends on three black-box functions, which respectively check for counterexamples to validity of candidate lemmas, `sample` variable valuations for a given proof state, and `synthesize` propositions from examples. Each component of `dilemma` is labeled with the black-box functions on which it depends.

We have performed two experimental studies to assess the effectiveness of `dilemma`. First, we evaluate `dilemma`'s ability to synthesize the helper lemmas required for proofs in the Verified Functional Algorithms book [1], which includes several formally verified data structures. In 43% of the locations where a helper lemma was used to complete a proof, one of the top-5 lemmas returned by `dilemma` is either an exact match to the human-provided lemma or an otherwise useful lemma for completing the proof. Second, we evaluate `dilemma` on the benchmark suites used to evaluate `lfind` [48], which almost exclusively require atomic equality lemmas. We find that despite targeting implication lemmas, which `lfind` is unable to produce by design, `dilemma`'s implicant synthesis is also expressive enough to produce an exact match to the human-provided atomic lemma in nearly 30% of the evaluation locations.

In summary, we present a data-driven technique for synthesizing implication lemmas to complete an interactive proof. After over viewing our approach and illustrating it through two examples (Section 2), we detail the algorithms that underlie it (Section 3). We have implemented our approach in a Rocq tactic called `dilemma` (Section 4) and have demonstrated its capabilities in practice on several benchmark suites (Section 5).

2 Motivation and Overview

This section motivates the need for lemma synthesis and overviews `dilemma`'s approach. After a general description of the approach, we describe two examples that come from the book Verified Functional Algorithms (VFA) [1], which contains Rocq implementations of common data structures and algorithms along with their proofs of correctness.

2.1 Architecture and Key Ideas

Figure 1 shows the architecture of `dilemma`. The tool is invoked as a Rocq tactic from a particular proof state, containing a current goal and set of assumptions. After first minimizing the set of assumptions to consider, `dilemma` explores two approaches to generating candidate implication lemmas, via weakening an existing assumption and inferring a new implicant to augment the existing assumptions. Finally, `dilemma` filters out invalid and redundant lemmas and rank orders the remaining lemmas.

The `dilemma` architecture is parameterized by three black-box functions. The `sample` function is a *random input generator*, producing sample valuations for a given set of typed variables. The `check` function is a *property-based tester* (e.g., [42]), which samples variable valuations in order to identify validity counterexamples to a given candidate lemma. Finally, the `synthesize` function is a *data-driven invariant synthesizer* (e.g., [40]), which is given two sets of variable valuations $\mathcal{E}^+$ and $\mathcal{E}^-$ and returns propositions that satisfy each valuation in $\mathcal{E}^+$ and falsify each valuation in $\mathcal{E}^-$. In Figure 1, each component of `dilemma` is labeled with the black-box functions that it invokes. Both property-based testing and data-driven synthesis are rich areas of research in their own right, and `dilemma` can directly leverage any future improvements in these external tools.

Our approach is based on three key ideas for generating useful implication lemmas. First, it is natural for the implicants in a required implication lemma to be related to the assumptions in the current proof state, but in proofs of any complexity there are many such assumptions. We observe that we can use valuations of the current proof state to identify a subset of *relevant* assumptions: those that are needed in order to prevent the `check` function from identifying a counterexample to the proof state's validity (i.e., a variable valuation that satisfies the state's assumptions but falsifies its goal). This is the role of the Reduce Assumptions component in Figure 1.

Once the assumptions are reduced, the resulting minimized proof state is itself a candidate lemma. However, a common issue is that the proof state is not directly provable because the induction hypothesis is too weak. To address this problem, `dilemma` explores two different approaches to strengthening the minimized proof state in order to generate candidate lemmas, respectively represented by the two paths in the figure.

On the top path, `dilemma` explores a space of *weakenings* of existing assumptions, which has the effect of strengthening the overall proof state. `dilemma` reduces the problem of assumption weakening to data-driven invariant synthesis, using sample valuations of the proof state to produce targeted examples for synthesis. On the bottom path through the figure, `dilemma` explores a space of *generalizations* of the proof state, whereby one or more terms are replaced by fresh variables. This is a standard technique [6], but in many cases the resulting proof state is *over-strengthened*, to the point where it is no longer valid. In such cases, `dilemma` infers additional implicants in order to rule out the identified validity counterexamples. As with assumption weakening, `dilemma` reduces the problem of implicant inference to data-driven invariant synthesis, with appropriate examples generated from the proof state.

Once `dilemma` has generated a set of candidate lemmas, it filters out candidates that are found to be either invalid or redundant. It then ranks the remaining candidates in order of rough complexity, based on the means by which they were created. For example, candidate lemmas that are produced solely by generalization are ranked higher than those that require an additional implicant to be synthesized. Within each rank, candidates are ordered by size from smallest to greatest. The top five lemma candidates are reported to the user.

In the remainder of this section we walk through two examples that illustrate these key ideas.

2.2 Example 1: Weakening Existing Assumptions

Figure 2 shows a portion of the VFA chapter on selection sort.[2] The function `selsort` produces a sorted list of the smallest `n` elements in the list `l`; this is the main function in the implementation of selection sort. The lemma `cons_of_small_maintains_sort` at the bottom of the figure proves a key property about `selsort` that is required to prove the overall correctness of selection sort, and it refers to the two propositions defined above it. Roughly, the lemma says that if `y` is less than

[2]Throughout this section we elide unnecessary details in the Rocq code and also use standard mathematical and logical notation where possible.

```
(* Selection sort for the n smallest elements of list l *)
Definition selsort (l : list nat) (n : nat) : list nat := ...

(* A proposition that is True if x is less than or equal to each element of xs. *)
Definition le_all (x : nat) (xs : list nat) : Prop := ...

(* A proposition that is True if the elements of xs are in non-decreasing order. *)
Definition sorted (xs : list nat) : Prop := ...

Lemma cons_of_small_maintains_sort: ∀ (n : nat) (bl : list nat) (y : nat),
    (n = length bl) ∧ (le_all y bl) ∧ (sorted (selsort bl n))
        ⟹ (sorted (y :: selsort bl n)).
Proof.
    intros. destruct (selsort bl n) eqn:Eq.
    - apply sorted_1.
    - apply sorted_cons.
      (* STUCK *)
```

Fig. 2. Partial proof in Rocq for the lemma `cons_of_small_maintains_sort` from VFA's chapter on selection sort.

```
n = length bl
le_all y bl
selsort bl n = a :: l
sorted (a :: l)
_________________________
y ≤ a
```

Fig. 3. Interactive proof state for motivating example 1 (Figure 2).

each element of the list `bl`, then adding `y` to the front of the sorted version of `bl` maintains its sortedness.

The figure also shows a partial proof of the lemma, which uses Rocq *tactics* to iteratively transform the proof state (the details are irrelevant). Figure 3 shows the proof state at the end of our partial proof. Above the line are four *assumptions*, propositions that can be assumed to be true, and below the line is the *goal* that we must prove. A bit of thought shows that the given assumptions do indeed imply the goal: `y` is less than or equal to each element of `bl`, and `a` is in the sorted version of `bl`, so `y` must be less than or equal to `a`. But how can we prove it?

Fortunately, the VFA book provides a lemma earlier in the chapter, whose proof is left as an exercise for the reader and which exactly fits our needs here:

```
Lemma le_all__le_one : ∀ (bl : list nat) (y : nat) (a : nat),
  (le_all y bl) ∧ (In a bl) ⟹ y ≤ a.
```

Here the proposition `In a bl` is true if the list `bl` contains the element `a`. The above lemma allows us to make progress in the proof by decomposing it into two parts. First, we have to prove the above lemma, which follows easily from the definitions of `le_all` and `In`. Second, after applying the lemma in our proof state, we are left to prove `In a bl`; this follows from the assumption in our proof state that `selsort bl n = a :: l`.

The VFA book provides this lemma for us, but in a more typical interactive proof setting the user would simply be stuck at this point in the proof and left to figure out a useful lemma on their own. Our work aims to fill this gap by identifying useful implication lemmas like the one above, and indeed our tool `dilemma` automatically suggests that lemma when invoked from the proof state in Figure 3.

How does `dilemma` do this? As shown in Figure 1, first it identifies the *relevant* assumptions in the proof state: those assumptions that are necessary in order to prove the goal. In our proof state there are four assumptions but it turns out that only the second and third of them are relevant to the goal `y ≤ a`. The `dilemma` tool determines a likely set of relevant assumptions in a data-driven manner using the black-box property testing function `check`, which takes a set of assumptions and a goal and tries to identify *validity counterexamples* — variable valuations that satisfy the assumptions but falsify the goal. Specifically, `dilemma` identifies a minimal subset S of the current assumptions such that `check(`S`, `G`)` returns no counterexamples, where G is the current goal. Note that in general propositions can be undecidable, so like other tools that rely on sampling valuations of the proof state [42] `dilemma` is restricted to code whose specifications are decidable.

That first step above already yields a potentially useful candidate lemma, namely that the relevant assumptions imply the goal. In our example, the induced candidate lemma is as follows:

```
Lemma relevant_state: ∀ (bl : list nat) (l : list nat) (y : nat) (n : nat) (a : nat),
    (le_all y bl) ∧ (selsort bl n = a :: l)  ⟹  y ≤ a.
```

Notice that the above lemma `relevant_state` is very similar to the lemma that we actually need, differing only in the second implicant. Specifically, our target lemma `le_all__le_one` is a stronger version of `relevant_state`, due to the fact that `In a bl` is *weaker* than (implied by) the proposition `selsort bl n = a :: l`. Based on this observation, `dilemma` explores a space of candidate lemmas by systematically weakening the identified relevant assumptions. We use a form of data-driven invariant synthesis for this purpose, which takes positive and negative examples and produces propositions that satisfy the positive examples and falsify the negative examples [16, 20, 21, 37, 40, 53].

Continuing the example, `dilemma` will use the black-box data-driven synthesizer `synthesize` to produce weakenings of the relevant assumption `selsort bl n = a :: l`. The black-box `sample` function is used to obtain random valuations of the variables in the current proof state, and `dilemma` uses these valuations to produce the examples for synthesis. The positive examples for synthesis are valuations that satisfy `selsort bl n = a :: l`. For instance, {`(bl, [5]), (n, 1), (a, 5), (l, []), (y, 3)`} is a positive example, so any synthesized proposition must satisfy it. Intuitively, the positive examples drive the synthesizer to explore weakenings of the original assumption. The negative examples for synthesis are valuations of the variables that satisfy all other relevant assumptions — in this case just `le_all y bl` — but falsify the goal `y ≤ a`. For instance, {`(bl, [5]), (n, 1), (a, 0), (l, []), (y, 3)`} is a negative example, so any synthesized proposition must falsify it. Intuitively, the negative examples help to ensure that the synthesized weakenings are still strong enough to prove the goal.

One of the propositions that `dilemma` synthesizes through this process is `In a bl`. This induces a proposed decomposition of the remaining proof that `dilemma` suggests to the user in the form of a pair of lemmas to prove. One lemma is exactly the target lemma `le_all__le_one` that VFA

```
(* A type for binary trees of natural numbers, with constructors Leaf and Node. *)
Inductive tree : Type := ...
Definition priqueue := list tree.

(* True if t is a power-of-two heap of depth n. *)
Definition pow2heap (n: nat) (t: tree) : Prop := ...
(* True if l is the ith tail of a binomial heap. *)
Definition priq'  (i: nat) (l: priqueue) : Prop := ...

(* Combine two trees into a single tree. *)
Definition smash (t1 : tree) (t2 : tree) : tree := ...
(* Add tree t to the priority queue q. *)
Definition carry (q: priqueue) (t: tree) : priqueue := ...

Theorem carry_valid: ∀ (n : nat) (q : priqueue),
    (priq' n q) ⟹ (∀ (t : tree), (t=Leaf ∨ pow2heap n t) ⟹ priq' n (carry q t)).
Proof.
    intros n q. generalize dependent n. induction q.
    + intros. destruct t. { simpl. inversion H0. auto. auto. } { auto. }
    + intros. destruct H0.
    ++ simpl. destruct a. { subst. assumption. } { subst. assumption. }
    ++ destruct a.
    - simpl. destruct t.
    -- split.
    * left. reflexivity.
    * apply IHq. inversion H. assumption. right.
    (* STUCK *)
```

Fig. 4. Partial proof in Rocq for the theorem `carry_valid` from VFA's chapter on binomial queues.

provides, which is produced by replacing `(selsort bl n = a :: l)` with `(In a bl)` in the lemma `relevant_state`. The other lemma prompts the proof that `In a bl` is indeed a valid weakening, namely that `∀ bl l n a, (selsort bl n = a :: l) ⟹ (In a bl)`.

2.3 Example 2: Synthesizing Missing Assumptions

Figure 4 shows part of the VFA chapter that implements priority queues using the binomial heap data structure. A priority queue in this representation is a list of binary trees (as shown on the second line of the figure), but they have a very special form. Specifically, the ith element of the list is either a `Leaf` or it is a *power-of-2 heap* that has exactly 2^i nodes. In the chapter this property is formalized by the `priq'` proposition, and its definition leverages the pow2heap proposition, which indicates whether a tree is a power-of-2 heap.

The `carry` function shown in the figure incorporates a new tree into a priority queue and is an important subroutine for both inserting new elements into a priority queue and merging two

```
∀ (n : nat), priq' n q ⟹
    ∀ (t : tree), (t = Leaf ∨ pow2heap n t) ⟹ (priq' n (carry q t))
priq' n ((Node n0 a1 a2) :: q)
pow2heap n (Node n1 t1 t2)
_____________________________________________________________________
pow2heap (n+1) (smash (Node n1 t1 t2) (Node n0 a1 a2))
```

Fig. 5. Interactive proof state for motivating example 2 (Figure 4).

```
priq' n (gv1 :: q)
pow2heap n gv0
______________________________
pow2heap (n+1) (smash gv0 gv1)
```

Fig. 6. Generalized proof state for motivating example 2 (Figure 4).

priority queues. That function relies on the `smash` function shown above it, which combines two trees into a single tree. The theorem `carry_valid` shown in the figure is the key correctness property for `carry`. Intuitively, the theorem says that `carry q t` preserves the `priq'` property for the given priority queue `q` whenever the tree `t` being added is a valid power-of-2 heap of the right size.

Figure 4 also shows a partial proof of the `carry_valid` theorem (whose details are not important to understand), and Figure 5 shows the proof state at that point, including three assumptions. Roughly, the user has to prove that smashing together two trees produces a valid power-of-2 heap of the right size, given the assumptions about those trees. As in the previous example, VFA provides a lemma that allows the user to make progress, as an earlier exercise:

```
Lemma smash_valid: ∀ (n : nat) (t : tree) (u : tree),
    (pow2heap n t) ∧ (pow2heap n u) ⟹ (pow2heap (S n) (smash t u)).
```

With that said, if the lemma were not provided, then the user would be left to find it on their own. As in the previous example, `dilemma` can automatically synthesize this lemma for the user.

First `dilemma` determines the relevant assumptions in the proof state, as shown for the previous example. For the state in Figure 5, only the second and third assumptions are relevant. While `dilemma` will then explore weakening these assumptions like we did in the previous example, `dilemma` also pursues a standard approach to lemma synthesis called *generalization* [6, 34], which strengthens the proof state by replacing terms with fresh universally quantified variables. As an example, consider the generalized proof state shown in Figure 6, where `(Node n1 t1 t2)` and `(Node n0 a1 a2)` are respectively replaced by fresh variables `gv0` and `gv1`, and only the relevant assumptions are shown. This proof state induces the following candidate lemma:

```
Lemma relevant_generalized: ∀ (q: list tree) (n: nat) (gv0: tree) (gv1: tree),
    (priq' n (gv1 :: q)) ∧ (pow2heap n gv0) ⟹ (pow2heap (S n) (smash gv0 gv1)).
```

If this lemma is valid, then it can be helpful for the user. Therefore, we use the check function to try to find counterexamples to validity and include the lemma as a candidate to provide the user if no counterexamples are found. Unfortunately, in this case, the lemma is not valid; for example, when `gv1` is a `Leaf` we can find values of the other variables that satisfy the implicants but falsify the implicand.

However, instead of simply giving up on this generalization, we observe that the identified counterexamples indicate that we need additional assumptions about `gv0` and/or `gv1` — important information that was lost in the generalization process. We therefore employ data-driven invariant synthesis, via the `sample` and `synthesize` functions, to produce candidate implicants that regain validity. The negative examples for synthesis are exactly the counterexamples to validity for this lemma, which will have the effect of producing implicants that rule out these counterexamples. The positive examples for synthesis are sample valuations from the original proof state that satisfy both the assumptions and goal, but then lifted to also give values to the generalized variables (e.g., `gv0` gets the value `(Node n1 t1 t2)` for specific values of `n1`, `t1`, and `t2`). The positive examples help to ensure that the synthesized implicant is weak enough to be provable from the original proof state's assumptions.

One of the propositions that `dilemma` synthesizes through this process is `(pow2heap n gv1)`. After incorporating this additional assumption to the proof state shown in Figure 6 and using `check` to reduce once more to the relevant assumptions, we obtain exactly the lemma `smash_valid` shown earlier, which the VFA book provides.

3 Algorithms

In this section we describe the core algorithms underlying `dilemma`. Our approach depends on access to the following three *black-box* functions:

`sample`$(\{v_1 : \tau_1, \ldots, v_n : \tau_n\})$ Given a set of typed variables, `sample` generates a set of *valuations*. Each valuation is a mapping of the form $\{v_1 : t_1, \ldots, v_n : t_n\}$ where each term t_i has type τ_i.

`check`$(\mathcal{A}, G)$ Given a proof state consisting of a set of assumptions $\mathcal{A}$ and a goal G, the `check` function uses sample valuations of the variables in the proof state to assess its validity, i.e., whether the assumptions imply the goal. Specifically, the function generates a set $\mathcal{E}$ of valuations of the variables in the proof state and returns false if there exists a valuation $e \in \mathcal{E}$ such that $e \models \bigwedge \mathcal{A}$ but $e \not\models G$; otherwise `check` returns true.

`synthesize`$(\mathcal{E}^+, \mathcal{E}^-)$ This function performs data-driven invariant synthesis. Given two sets of valuations $\mathcal{E}^+$ and $\mathcal{E}^-$, `synthesize` returns a set of propositions P over the variables in the valuations such that for each $p \in P$ we have $\forall e \in \mathcal{E}^+, e \models p$ and $\forall e \in \mathcal{E}^-, e \not\models p$. We refer to $\mathcal{E}^+$ as *positive examples* and $\mathcal{E}^-$ as *negative examples* for synthesis.

Our approach is independent of the algorithms and implementations of these functions and can benefit from future improvements in these capabilities. In §4, we'll describe our specific implementations of these functions.

Algorithm 1 details our overall approach to lemma synthesis, which follows the informal discussion from the previous section. First, we reduce the assumptions $\mathcal{A}$ to a subset $\mathcal{S}$ of assumptions that are deemed necessary. Algorithm 2 shows how this is done: the `check` function is used to find a minimal subset of the assumptions that is necessary to imply the goal.[3] This yields our first candidate lemma, as shown in line 3 in Algorithm 1.[4] We then sample valuations of the variables

[3]If there is no such subset, that implies that the current proof state is invalid and so cannot lead to a proof. In that case, we halt the algorithm and notify the user.

[4]All free variables in the propositions will be universally quantified at the front of the lemma, but we elide this detail in the algorithm for simplicity.

Algorithm 1 Overall Pipeline for `dilemma`

Input: set of proof assumptions $\mathcal{A}$, proof goal G
Output: set of ranked candidate lemmas

1: **procedure** DILEMMA($\mathcal{A}$,G):
2: $\quad\mathcal{S} \leftarrow$`ReduceAssumptions`($\mathcal{A}$,$G$) ▷ $\mathcal{S}$ stores a minimal set of assumptions
3: $\quad\mathcal{R} \leftarrow \{(\bigwedge \mathcal{S} \implies G)\}$ ▷ $\mathcal{R}$ stores the candidate lemmas
4:
5: $\quad\mathcal{E} \leftarrow$`sample(vars(`$\mathcal{S}$`)` $\cup$ `vars(`G`))`
6:
7: $\quad$**for** $a \in \mathcal{S}$ **do**
8: $\quad\quad\mathcal{W} \leftarrow$ `synthesize(WeakenAssumption(`$\mathcal{E}, \mathcal{S}, G, a$`))`
9: $\quad\quad\mathcal{R} \leftarrow \mathcal{R} \cup \{((a \implies w), (w \wedge (\bigwedge(\mathcal{S} \setminus \{a\})) \implies G)) \mid w \in \mathcal{W}\}$
10:
11: $\quad\mathcal{G} \leftarrow$`Generalize(`$\mathcal{S}$,$G$`)`
12: $\quad$**for** $(\mathcal{S}', G', \mu) \in \mathcal{G}$ **do** ▷ μ maps variables to the terms that they generalize
13: $\quad\quad$**if** `check(`$\mathcal{S}', G'$`)` **then**
14: $\quad\quad\quad\mathcal{R} \leftarrow \mathcal{R} \cup \{(\bigwedge \mathcal{S}' \implies G')\}$
15: $\quad\quad$**else** ▷ $\mathcal{I}$ holds the missing assumptions that are synthesized
16: $\quad\quad\quad\mathcal{I} \leftarrow$ `synthesize(InferImplicants(`$\mathcal{E}, \mathcal{S}', G', \mu$`))`
17: $\quad\quad\quad\mathcal{R} \leftarrow \mathcal{R} \cup \{\bigwedge$ `ReduceAssumptions(`$\mathcal{S}' \cup \{i\}, G'$`)` $\implies G' \mid i \in \mathcal{I}\}$
18: $\quad\quad\quad\mathcal{R} \leftarrow \mathcal{R} \cup \{t = t' \mid v = t' \in \mathcal{I}$ and $(v, t) \in \mu\}$
19:
20: $\quad$**return** `Rank(Filter(`$\mathcal{R}$`))`

Algorithm 2 Reducing to Only Necessary Assumptions

Input: set of proof assumptions $\mathcal{A}$, proof goal G
Output: a minimal subset of $\mathcal{A}$ that imply G

procedure REDUCEASSUMPTIONS($\mathcal{A}$,G):
$\quad$**for** $i \in [0, \ldots, |\mathcal{A}|]$ **do**
$\quad\quad$**for** each $\mathcal{S} \subseteq \mathcal{A}$ such that $|\mathcal{S}| = i$ **do**
$\quad\quad\quad$**if** `check(`$\mathcal{S}, G$`)` **then**
$\quad\quad\quad\quad$**return** $\mathcal{S}$

in the given proof state, which will be used for data-driven implicant synthesis in the rest of the algorithm.

3.1 Assumption Weakening

Next the algorithm tries to weaken each assumption a in $\mathcal{S}$. On line 8 in Algorithm 1, we use the `WeakenAssumption` function to generate positive and negative examples for weakening, which are passed to the data-driven synthesizer in order to produce candidate weakenings $\mathcal{W}$. Each candidate weakening w yields two lemmas, as shown on line 9: one states that w is in fact a weakening of a and the other is the lemma derived by replacing a with w in the set of necessary assumptions. We add these two lemmas as a pair into our set $\mathcal{R}$ of candidate lemmas, so that they will be filtered and ranked together in the last step of the algorithm.

Algorithm 3 shows how we produce the positive and negative examples to weaken an assumption a. For a candidate implicant w to be a weakening of a, any valuation that satisfies a should also

Algorithm 3 Positive and Negative Examples for Assumption Weakening

Input: valuations of proof variables $\mathcal{E}$, set of assumptions $\mathcal{S}$, goal G, assumption a
Output: positive and negative examples for synthesis to weaken assumption a

procedure WeakenAssumption($\mathcal{E}$,$\mathcal{S}$,G,a):
 $\mathcal{E}^+ \leftarrow \{e \in \mathcal{E} \mid e \models a\}$
 $\mathcal{E}^- \leftarrow \{e \in \mathcal{E} \mid e \models \bigwedge(\mathcal{S} \setminus \{a\})$ and $e \not\models G\}$
 return $(\mathcal{E}^+,\mathcal{E}^-)$

satisfy w. Hence all valuations in $\mathcal{E}$ that satisfy a are used as the positive examples for synthesis. But the implicant w must also be strong enough to imply the goal G in conjunction with the other assumptions in $\mathcal{S}$. Hence, the negative examples for synthesis are the valuations in $\mathcal{E}$ that w must falsify in order to ensure that the proof state remains valid if a is replaced by w.

3.2 Generalization and Implicant Inference

After weakening existing assumptions, Algorithm 1 employs the generalization [6] technique to explore additional candidate lemmas. On line 11, the algorithm creates a set of generalizations of the reduced proof state. The definition of the function `Generalize` is not shown but is straightforward: each produced generalization introduces fresh variables v_i for one or more terms t_i that appear in the goal G. Each generalization in the resulting set $\mathcal{G}$ is a triple consisting of $\mathcal{S}[t_i \mapsto v_i]$, $G[t_i \mapsto v_i]$, and the mapping $\{v_i : t_i\}$ that defines this generalization.

Next we consider each generalization in turn (line 12). If the generalization is not found to be invalid by the `check` function, then it is simply added to the set of candidate lemmas (lines 13-14). Otherwise, we use the `InferImplicants` function to generate positive and negative examples to infer implicants that can make the generalization valid; these examples are passed to the data-driven synthesizer in order to produce candidate implicants $\mathcal{I}$. Each synthesized implicant $i \in \mathcal{I}$ yields a candidate lemma by adding the implicant to the existing set $\mathcal{S}'$ of assumptions and removing any assumptions that are no longer needed (line 17).

Algorithm 1 also includes special handling for the case when the candidate implicant is an equality of the form $v = t'$ where v is a fresh variable introduced for generalization (line 18). Since v was introduced for generalization, it is a stand-in for some existing term t from the proof state. Hence in this case we also yield a candidate lemma $t = t'$.

Algorithm 4 Positive and Negative Examples to Infer Implicants

Input: valuations of original proof variables $\mathcal{E}$, set of generalized assumptions $\mathcal{S}'$, generalized goal G', generalization mapping μ
Output: positive and negative examples for synthesis to infer missing assumptions

procedure InferImplicants($\mathcal{E}$,$\mathcal{S}'$,G',μ):
 $\mathcal{E}' \leftarrow \mathrm{Lift}(\mathcal{E},\mu)$
 $\mathcal{E}^+ \leftarrow \{e \in \mathcal{E}' \mid e \models \bigwedge \mathcal{S}'$ and $e \models G'\}$
 $\mathcal{E}'' \leftarrow \texttt{sample}(\texttt{vars}(\mathcal{S}') \cup \texttt{vars}(G'))$
 $\mathcal{E}^- \leftarrow \{e \in \mathcal{E}'' \mid e \models \bigwedge \mathcal{S}'$ and $e \not\models G'\}$
 return $(\mathcal{E}^+,\mathcal{E}^-)$

Algorithm 4 details how we produce the positive and negative examples for synthesis to infer new implicants. First, we lift each valuation e in $\mathcal{E}$ to include values for the generalized variables. The `Lift` function is not shown but is straightforward: each generalized variable v is set to the value of $\mu(v)$ when evaluated under the variable valuation e, where μ is the mapping of variables to the terms they've generalized. The positive examples for synthesis are then simply the lifted valuations that satisfy all assumptions and the goal. This ensures that the inferred implicants will be weak enough to include the satisfiable valuations of the original proof state. To obtain negative examples for synthesis, we sample from the generalized proof state (line 4) and collect those valuations that represent violations of validity (the assumptions are satisfied but the goal is falsified). Including these valuations as negative examples will help ensure that the synthesized implicants are strong enough to restore validity.

3.3 Filtering and Ranking

The final step in Algorithm 1 is to filter the lemma candidates that are less likely to be useful, and then to rank the remaining lemmas for presentation to the user.

Filtering. The `Filter` function (definition not shown) removes candidate lemmas that are either invalid or redundant, similar to the filtering done by the `lfind` data-driven lemma synthesis tool [48]. Specifically:

- **Invalid.** We remove candidate lemmas for which `check` returns false, indicating that the lemma is not valid. Additionally, if either lemma in a pair resulting from weakening an existing assumption is invalid, then the entire pair is removed.
- **Redundant.** We remove candidate lemmas that are clearly unhelpful. This includes weakenings that are syntactically equal to the assumption that they weaken and inferred implicants that simply assign a generalized variable to be equal to the term that it generalized. To go beyond syntactic equality, we say that two propositions are *redundant* with one another if Rocq's `rewrite`, `reflexivity`, and/or `apply` tactics can be used to prove that the propositions imply each other. We remove candidate lemmas that are redundant with either the goal or with the assumption being weakened (where applicable).

Ranking. The `Rank` function orders the remaining lemma candidates in order of their likely utility. The prior `lfind` tool ranks candidate lemmas based on provability — whether the lemma itself is provable and whether it can be used to fully prove the goal — by invoking an automated prover [46]. However, this ranking approach degrades as proofs get more complex, because it becomes less likely that an automated prover can complete the proofs automatically. Indeed, our experiment with the Proverbot9001 automated prover [46] in Section 5.4 shows several instances of useful lemmas that cannot be automatically proven.

For this reason, we propose a new ranking approach that is specific to the way in which `dilemma` constructs candidate lemmas. As defined in Algorithm 1, our algorithm constructs lemmas in five different ways: (1) reducing the assumptions of the current proof, (2) weakening an existing assumption, (3) generalizing the proof state, (4) inferring a necessary implicant for a generalization, and (5) creating an equality lemma for a generalized term. These different means of construction are given different levels of priority in our ranking based on perceived complexity. We order them as follows: 3,1,5,2,4. For example, valid generalizations come first in our ranking, as they tend to be the simplest, while lemmas that require an additional implicant come last. We order all lemmas and lemma pairs that are in the same priority level based on the size of their abstract syntax tree representations, from least to greatest.

4 Implementation

We've instantiated our approach in a Rocq tactic called `dilemma`, which is implemented in Rocq and OCaml.

4.1 Sampling, Validity, and Synthesis

As described in the previous section, `dilemma`'s approach depends on black-box functions for sampling valuations of a proof state; searching for validity counterexamples to a proof state; and synthesizing propositions from positive and negative examples.

`dilemma` uses the property-based tester `QuickChick` [42] as its `check` function. Given a proof state with assumptions $\mathcal{A}$ and goal G, we invoke `QuickChick` on the proposition $\bigwedge \mathcal{A} \implies G$. `QuickChick` will then sample valuations of the variables in the given proposition and use them to search for counterexamples to the proposition's validity.

`dilemma` inherits the requirements for running `QuickChick`. Specifically, users must provide a proof of decidability for any propositions used in their code. In addition, each type used in the code must be *showable*, meaning that it has a function to produce a string representation of a value, and it also must have a generator for producing random values of the type. `QuickChick` includes some support for automatically generating these functions for inductive types.

We also use `QuickChick` to implement the `sample` function, by creating a dummy property of the right type for `QuickChick` to test and then recording the values that it generates. We do this instead of directly invoking the underlying generators for each type in order to leverage `QuickChick`'s sampling heuristics.

We implemented a simple enumerative type-based synthesizer in OCaml to perform the data-driven implicant synthesis as defined by the `synthesize` function. `dilemma` passes the synthesizer the positive and negative examples along with a set of typed constructors, functions, and propositions (extracted from Rocq) that may be used to construct implicants. The synthesizer enumerates well-typed propositions in order of size, up to a specified bound on the heights of their abstract syntax trees, and it returns all those that respect the given positive and negative examples. When `dilemma` invokes the synthesizer from a particular point in a proof, it passes all constructors, functions, and propositions that are defined in the surrounding file, along with all of those that are used in the file but imported from other libraries.

`dilemma` currently does not include support for sampling values of a function type. Similarly, the synthesizer does not currently support synthesizing values of a function type. These limitations are not fundamental but would require some experimentation to determine the most effective approaches.

4.2 Hyperparameters and Bounds

The search space for useful implicants is large. `dilemma` uses several hyperparameters to practically bound the search space (and hence the time required to run `dilemma`) while retaining effectiveness.

- **Generalization** A generalized proof state can introduce any number of fresh variables, each replacing some term in the original proof state. `dilemma` only considers generalized proof states with up to three fresh variables, and not all of them are considered. Specifically, generalized proof states are enumerated in order of the number of fresh variables that they contain. If a generalized proof state is found to be invalid, then `dilemma` tries to infer a missing assumption, yielding new candidate lemmas. If the generalized proof state is valid, then `dilemma` generates a candidate lemma from the proof state and then also moves on to consider extensions of the generalization that include an additional fresh variable, if we are not already at the limit of three. Due to the limitations described above, terms of function type are never generalized.
- **Sampling** When using `QuickChick` to generate sample valuations, we set a limit of 9000 samples, to ensure sufficient diversity. When we use these valuations to produce positive and negative examples for synthesis, we limit each set of examples to at most 500. Across all

of the tests that we ran, a synthesis query had an average of 329 positive examples and 354 negative examples.

Synthesis As mentioned above, `dilemma` uses a simple enumerative synthesizer for Rocq propositions. The synthesizer takes as arguments the sets of positive and negative examples, as well as the set of functions, constructors, and propositions to use for synthesis. In addition it takes a bound on the *height* of synthesized propositions, meaning the maximum number of nodes from the root to a leaf in the proposition's AST representation. In our experiments we use a maximum height of 3. For example, `Permutation (a ++ b) (b ++ c)` has a height of 3 (and is considered), but `Permutation (a ++ b) (b ++ a ++ c)` has a height of 4 (and is not considered).

These hyperparameter values were determined empirically, attempting to balance expressiveness against time. For example, we observed that the number of distinct examples generated by QuickChick plateaus after 9000 samples or so. The bounds are fairly modest, for example a maximum height of only 3 for synthesized propositions. With more computational resources and/or time allowance we could increase them, and of course they can also be made user-configurable.

4.3 Filtering and Ranking

`dilemma` uses `QuickChick` to filter invalid candidate lemmas and a simple Rocq script to check for a candidate lemma's redundancy. Each candidate lemma is tagged to indicate by which of the five possible means in `dilemma`'s algorithm it was constructed, and our ranking function (defined in OCaml) leverages these tags and the size of each lemma to totally order them. Currently `dilemma` returns only the top five ranked candidates; however, it also produces a full log containing all of the candidate lemmas that were synthesized.

5 Evaluation and Results

In this section, we evaluate the capabilities of `dilemma` through two experiments, as well as compare its performance to that of a prior state-of-the-art tool, `lfind`. First, we evaluate `dilemma`'s (and `lfind`'s) ability to synthesize the lemmas used in the correctness proofs for data structures from the Verified Functional Algorithms (VFA) book [1]. These proofs heavily rely on implication lemmas. Second, we compare the tools' performances on the benchmarks used to evaluate `lfind` [48]. These benchmarks almost exclusively require *atomic equality* lemmas, i.e., lemmas of the form $\forall \overline{v}.t_1 = t_2$, where t_1 and t_2 are terms over the variables in $\overline{v}$.

5.1 Experimental Setup

Our experiments assess `dilemma`'s ability to synthesize the lemmas used by a human to prove the lemmas and theorems in each benchmark. For each proof of a theorem or lemma in a benchmark, every point in the proof that uses either the `apply` or `rewrite` tactic to use a lemma becomes an evaluation point for `dilemma`— we execute `dilemma` from that point in the proof and compare the synthesized lemmas against the actual one that was used in the proof. For the VFA benchmark, we started from the proof sketches that the book provides in each chapter, which includes the statement of key helper lemmas [1]. We manually completed the proofs of all lemmas and theorems, adding new helper lemmas where needed, to create the benchmark.

All `dilemma` evaluations were performed on a machine that runs MacOS using an Apple M1 Pro chip with 10 cores and 16GB memory. We used the artifact submitted with the `lfind` paper [47] in order to run `lfind` on the VFA benchmark.

We deem `dilemma` to be successful at a particular proof point if one of the top 5 lemmas produced by `dilemma` is either an *exact match* for the lemma used by the human at that point or is otherwise

Table 1. Results for `dilemma` on the VFA benchmark

Chapter	Total	Exact Match	Step in Right Direction	Weaker Version of Target	Average Runtime (s)
bagperm	11	5	0	0	22.1
binom	46	17	5	6	406.1
merge	17	10	0	0	138.9
perm	1	0	0	0	7.1
priqueue	8	2	2	0	128.1
redblack	32	10	0	0	258.0
searchtree	59	1	3	2	206.1
selection	24	15	2	0	70.3
sort	11	7	0	0	40.4
trie	17	10	0	0	62.9
TOTALS	226	77	12	8	206.6

useful for completing the proof. Specifically, we define two classes of useful lemmas that are not an exact match:

A step in the right direction From the reduced proof state, `dilemma` produces candidate lemmas either by weakening a single assumption in the state or inferring a single new implicant. However, some lemmas require that multiple assumptions be weakened or inferred. For example, suppose that a target lemma requires both that assumption A be weakened to A' and assumption B be weakened to B', yielding the lemma $A' \land B' \implies G$, where G is the current goal. If `dilemma` instead produces the lemma $A' \land B \implies G$, we consider this a useful result since it helps the user to make progress by identifying the implicant A'.

A weaker version of the target lemma that is used equivalently A lemma that falls in this category is weaker that the human lemma that is used, but it is able to make the same step in the provided proof context when applied, yielding the same resulting proof state. For example, suppose the target lemma was `(In x l` $\lor$ `In x m)` $\implies$ `(In x (l ++ m))` but `dilemma` instead found `(In x l)` $\implies$ `(In x (l ++ m))`. If the latter can be applied to the current goal in the same way as the former, and if proving the latter lemma is of similar difficulty as proving the former, then we declare the latter lemma useful.

5.2 Evaluation on Verified Functional Algorithms

The VFA benchmark consists of 10 out of the 16 chapters in the book. Two chapters, *Multiset* and *Maps*, were skipped because their data structures are defined using function types, and our implementation does not currently support sampling or synthesis of functions. The other four chapters (*ADT, Extract, Decide and Color*) were skipped because they don't define or verify a data structure. In total across the chapters that we used there are 226 points in the proofs where a helper lemma is used, and hence from where we evaluate `dilemma`.

5.2.1 Evaluation of `dilemma`. The results of running `dilemma` on the VFA benchmark are shown in Table 1. In 34.1% of the evaluation points (77 out of 226), `dilemma` finds an *exact match* to the

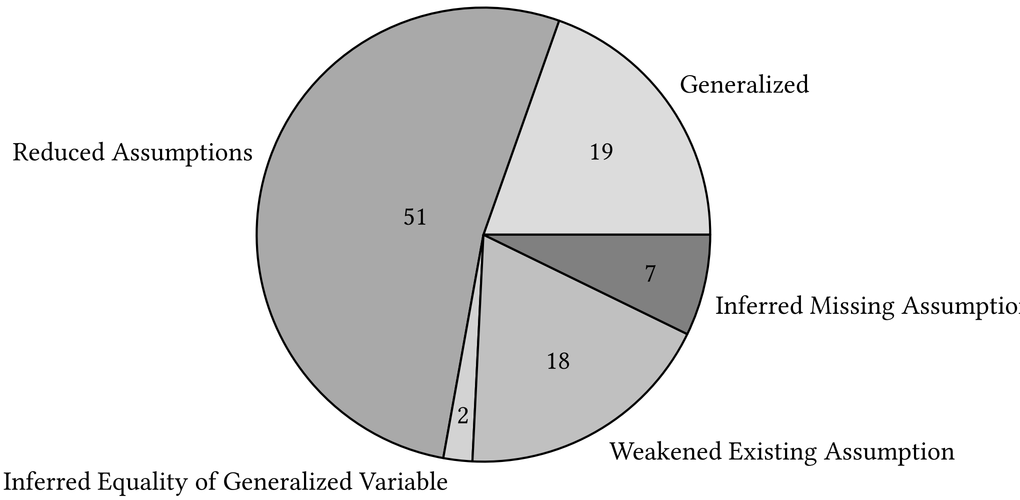

Fig. 7. Breakdown of the exact matches and useful lemmas from dilemma for VFA.

```
∀ (n : nat) (m : nat) (q : list tree) (q' : list tree),
    (priqq n p) ∧ (delete_max_aux m p = (q, q')) ⟹ (priq' n q ∧ priq q')
priq' n (Node n0 a1 Leaf :: p)
(Leaf :: p, unzip a1 (fun u : list tree ⇒ u)) = (q, q')
Leaf :: p = q
unzip a1 (fun u : list tree ⇒ u) = q'
________________________________________________________________________________
priq (unzip a1 (fun u : list tree ⇒ u))
```

Fig. 8. Interactive proof state for proof of `delete_max_aux_priq` from VFA's chapter on binomial queues.

helper lemma that is used at that point, within the top 5 lemmas returned by `dilemma`. Further, 97.4% of the exact matches are within the top 3, and 76.6% of the exact matches are the top lemma returned by `dilemma`. If we include the two categories of other useful lemmas, as described earlier, then `dilemma` returns an exact match or useful lemma in the top 5 in 42.9% of the evaluation points.

Figure 7 breaks down the manner in which each successful lemma was constructed by `dilemma` (referring to the categories described in §3). Most of the lemmas come from reducing the assumptions to the necessary ones. This is a common case in general but also makes sense for these benchmarks, where very general properties of data structures are being proven. While relatively simpler than the other parts of `dilemma`, reducing the assumptions still inherently leverages `dilemma`'s data-driven approach. Further, 27 of the 97 successes employ a form of data-driven invariant synthesis. Finally, 64 of the 97 successes are implication lemmas (this is not shown in the figure).

Example. The following example illustrates how `dilemma` can provide value in making progress from complex proof states, even when it does not find the exact target lemma. Figure 8 shows a proof state from the chapter on binomial queues where the following helper lemma (provided by VFA) is used:

```
priq' n (Node n0 a1 Leaf :: p)
_______________________________________________
priq (unzip a1 (fun u : list tree ⇒ u))
```

Fig. 9. Reduced interactive proof state for proof of `delete_max_aux_priq` from VFA's chapter on binomial queues found by `dilemma`.

Table 2. Breakdown of failures for `dilemma` on the VFA benchmark.

Failure Reason	Quantities
Forward Reasoning	59
Resource Limit	19
Implementation Restriction	29
Ranked Exact Match Too Low	3
Beyond Scope	19

```
Lemma unzip_preq: ∀ (t : tree) (f : list tree -> list tree) (n : nat) (k : nat),
    (pow2heap' n k t) ∧ (priq' n (f [])) ⟹ (priq (unzip t f)).
```

The details of the `unzip` function and propositions like `pow2heap'` are not necessary to understand what `dilemma` does on this example. First, `dilemma` reduces the assumptions to the necessary ones, resulting in the proof state shown in Figure 9. In order to find the target lemma, `dilemma` would need to both weaken an existing assumption and infer a missing assumption, but `dilemma` performs at most one of these tasks per candidate lemma. Furthermore, because of `dilemma`'s current limitations on sampling valuations of functions, it does not support generalization for terms of a function type, which is needed in this case. Nonetheless, `dilemma` is able to find the following lemma through assumption weakening:

```
Lemma useful_onestep_lemma: ∀ (n : nat) (k : nat) (t : tree),
    (pow2heap' n k t) ⟹ (priq (unzip t (fun u : list tree ⇒ u))).
```

This lemma is a special case of the target lemma for when the function `f` is the specific function `(fun u : list tree ⇒ u)`, i.e. the identity function. This is an example of *a step in the right direction* because `dilemma` correctly inferred the required assumption about the `pow2heap'` proposition. This example also illustrates that despite `dilemma`'s limitations on sampling valuations for variables of function type, it can still provide value for proof states that involve functions.

Failure analysis. We've broken down the reasons that `dilemma` fails to find a useful lemma into five broad categories which are shown in Table 2. Specifically:

Forward Reasoning These are cases where the target lemma is applied in an assumption rather than in the goal. `dilemma`'s approach to assumption weakening yields a pair of lemmas, one of which applies to an assumption, as shown in the first example in Section 2. However, even in that case, the lemmas are derived from the needs of the current goal. Hence proof strategies that derive new facts from the assumptions before simplifying the goal are not well supported by `dilemma`. This also includes cases where the assumptions are contradictory and so completing the proof requires this contradiction to be proven from the assumptions alone.

Resource Limit These are cases where the resource bounds described in §4 prevent `dilemma` from finding the target lemma. The two most common limitations that lead to failures are

Table 3. The average percentage of time spent by `dilemma` on different tasks, per evaluation point in the VFA benchmark.

Group	*Reduce Assumptions*	*Synthesize*	*Rank + Filter*
All	56.37%	16.34%	27.29%
Slowest 10%	49.79%	37.47%	12.74%
Middle 10%	59.12%	12.06%	28.82%
Fastest 10%	65.32%	10.06%	24.62%

the limit on the number of generalized terms considered and the limit on the height of synthesized propositions.

Implementation Restriction These are cases that are not beyond the scope of our approach but fail due to implementation limitations. The most common one is the lack of support for generalizing function types.

Ranked Exact Match Too Low These are cases where `dilemma` found an exact match for the lemma that was used, however, the lemma was ranked beyond the top 5 candidates. The fact that there are only three such cases validates the effectiveness of our ranking technique in general.

Beyond Scope Finding these lemmas appears to be beyond the scope of our technique.

Runtime analysis. Table 1 also shows the average time that `dilemma` takes at a single evaluation point, both on a per-chapter basis and over all evaluation points. The time taken varies quite a lot based on the complexity of the chapter. The average runtime per test is 206.6 seconds, as listed in bottom right of the table 1. However, the *median* runtime across the entire set of evaluation points is significantly less at 100.0 seconds (not shown in the figure). Given these running times, `dilemma` is not currently suitable for a tight interaction loop with the user. However, it is still quite reasonable for the user to wait a few minutes to obtain guidance on how to make progress from a point at which they are stuck, since without a tool like `dilemma` users are simply on their own to figure out how to proceed.

Table 3 breaks down the time that `dilemma` spends on different tasks, determined as a percentage of time per evaluation point and then averaged over all such points. The majority of the time is spent on reducing the assumptions to just those that are relevant to the goal. Currently `dilemma` finds a *minimal* set of relevant assumptions by considering all subsets of assumptions in order of size. Obviously this becomes expensive as the number of relevant assumptions increases, so to improve usability it is worth considering heuristics that give up on minimality in exchange for increased efficiency. The table also shows that the percentage of time spent on synthesis is dramatically higher on the slowest benchmarks versus on the fastest or median-time benchmarks. Our synthesizer enumerates all propositions up to some maximum size and so is quite sensitive to the number of possible propositions, as induced by the given set of constants and functions. Hence it will also be helpful to improve the black-box synthesizer, for example by adding heuristics for searching the space or by exploring synthesis techniques other than enumeration (e.g., LLM-based synthesis).

5.2.2 Evaluation of `lfind`. `lfind` is a state-of-the-art prior tool that performs data-driven lemma synthesis for interactive proofs [48]. `lfind` is designed to generate atomic lemmas but is not able to synthesize implication lemmas, so we ran `lfind` on just the VFA evaluation points where an

Table 4. `lfind` and `dilemma` results for the equality benchmarks. `dilemma`'s results include only exact matches with the target lemma.

Benchmark Suite	`lfind`		`dilemma`	
clam	101/140	72.1%	54/140	38.6%
lia	11/19	57.9%	6/19	31.6%
full adder	37/62	60.0%	7/62	11.3%
compiler	1/1	100%	0/1	0%

atomic lemma is used. Out of 226 evaluation points in VFA, 54 of them use an atomic lemma, and 33 of these lemmas are atomic equality lemmas.

Unfortunately, `lfind` has significant limitations on the syntax of functions and terms accepted by its synthesizer, as discussed in §4.4 of the `lfind` paper [48]. For example, it does not support parametric polymorphism and only supports a small subset of the OCaml language, to which Rocq functions and terms get extracted for purposes of synthesis. The original evaluation of `lfind` skipped 44 evaluation points in their benchmarks due to these limitations.

The VFA data structures are all polymorphic and employ polymorphic functions and definitions from the Roqc standard library. We created monomorphic versions of all required types and definitions in order to avoid the `lfind` limitation. Nonetheless, in 49 out of the 54 evaluation locations, `lfind` still failed due to a syntactic limitation of its synthesizer. Hence we were only able to get results for the remaining five evaluation points. None of them yielded a useful lemma, according to either our success metrics for `dilemma` or the success metrics used for `lfind`'s evaluation [48].

5.3 Evaluation on Equality Benchmarks

The existing benchmarks for lemma synthesis contain evaluation points that largely require atomic equality lemmas. We ran `dilemma` on the four benchmarks used to evaluate `lfind` [48]. The `clam` and `lia` benchmarks contain proofs of equalities about numbers and simple data structures. The `full adder` benchmark creates a full-adder circuit and proves its correctness, and the `compiler` benchmark has a single evaluation point in the proof of a simple compiler's correctness.

Table 4 lists the results reported on these benchmarks in the `lfind` paper alongside the results of running `dilemma` on the same benchmarks and same evaluation points. For `dilemma` we conservatively only report success if there is an exact match to the target lemma; the success criteria used for `lfind`'s evaluation are somewhat broader [48].

`dilemma` synthesizes an exact match to the target lemma for a bit more than 30% of the evaluation points. This constitutes about 45% of the number of evaluation points for which `lfind` succeeds, but `lfind` is designed specifically to synthesize atomic equality lemmas. The evaluation demonstrates that `dilemma` is also able to express such lemmas, despite its focus on implicant synthesis. We give an example below to illustrate this expressiveness.

Finally, the `clam` benchmark has 18 proof locations where an implication lemma was used, and which were not included in the `lfind` evaluation since that tool does not support synthesizing implications. Similarly, there are 9 such locations in the `lia` benchmark. We therefore additionally evaluated `dilemma` on these locations, and it was able to synthesize an exact match to the target lemma in 13 out of the 27 evaluation points (result not shown in the table).

Example. Consider the example in Figure 10, which is used as a motivating example in the `lfind` paper [48]. The figure shows a proof state in the middle of a proof that `rev (rev l) = l` for all lists `l`, which is one of the `clam` benchmarks. At this proof state, the following lemma is used:

```
(rev (rev l)) = l
________________________________________
(rev (append (rev l) [n])) = (n :: l)
```

Fig. 10. Interactive proof state for proof of `rev_rev` from the `clam` benchmarks.

```
Lemma target : ∀ (l' : Lst) (n : Nat),
    rev (append l' [n]) = n :: (rev l').
```

`lfind` identifies this helper lemma through two rewrites on the goal in Figure 10. First, it generalizes `(rev l)` to a fresh variable `l'`. Then, it performs a form of data-driven program synthesis to identify the term `n :: (rev l')` as a replacement for `n :: l`.

Despite the fact that `dilemma` does not perform goal rewrites, it is nonetheless able to find this equivalent implication lemma:

```
Lemma found : ∀ (l : Lst) (n : nat) (l' : Lst),
    (rev l' = l)  ⟹  (rev (append l' [n]) = n :: l).
```

As in `lfind`, `dilemma` first generalizes `(rev l)` to a fresh variable `l'`. However, it then infers the missing assumption `rev l' = l` which is needed to make the generalized proof state valid.

This example illustrates that `dilemma`'s approach is expressive enough to support the goal rewrites that `lfind` considers. In general, if `lfind` would replace a term `t` with another term `t'` in the goal, then this can be equivalently represented as an additional assumption `t = t'` that is needed to regain validity of the proof state. Hence in principle, `dilemma` subsumes the behavior of `lfind` while additionally supporting synthesis of complex implication lemmas. In practice, `dilemma` finds fewer atomic lemmas than `lfind` due to its resource bounds, for example on the number of generalized terms to consider. `lfind`'s exclusive focus on goal rewrites allows it to consider *all possible* generalizations of terms in the goal.

5.4 Leveraging `dilemma` for Proof Automation

We performed a small experiment to gauge the value of `dilemma` in aiding automated proof generation. Specifically, we combine `dilemma` with the ProverBot9001 [46] automated neural prover for Rocq. In all runs of ProverBot9001 we used the same hyperparameters that were used for ProverBot9001 in the evaluation of `lfind` [48]. Specifically, we used the pre-trained weights that are provided with ProverBot9001 and set the `max-proof-time` field to 15 seconds. Additionally, we imposed a 5-minute timeout for each entire run of ProverBot9001.

As mentioned in the previous subsection, there are 27 proof locations across the `clam` and `lia` benchmarks where an implication lemma is used. We first asked ProverBot9001 to complete the proofs from these 27 locations, but without access to the particular implication lemma that was appealed to in the original proof. In this way, we are simulating a scenario where the required helper lemma is not yet written. As expected, ProverBot9001 was unable to complete a single one of these 27 proofs, since it can use existing lemmas but lacks the ability to synthesize new ones.

We then ran a variant of this experiment where we simulate a version of ProverBot9001 that can appeal to `dilemma`. Specifically, we run `dilemma` at each of the 27 proof locations, apply the top-ranked lemma from `dilemma` to the current proof state, and then ask ProverBot9001 to complete the proof. We then do the same thing but with the second-ranked lemma that `dilemma` provides. ProverBot9001 successfully completes the proof in at least one of the two runs for 14 out of the 27 proof locations.

Finally, we investigated the extent to which the `dilemma`-provided lemmas in those 14 successful proof locations are useful in simplifying the overall proof. First we asked Proverbot9001 to prove each of those lemmas, and it was able to do so for one of the top-2 `dilemma`-provided lemmas at two of the 14 proof locations. Hence at two proof locations, Proverbot9001 can use `dilemma` to provide a full proof to the user, which otherwise would not have been possible. For the remaining 12 locations, the helper lemmas are themselves complex enough (e.g., requiring their own helper lemmas) that Proverbot9001 cannot prove them automatically. However, in five of these cases one of the `dilemma`-provided lemmas is an exact match to the lemma used by the human in the original proof. Hence the combination of `dilemma` and Proverbot9001 is still very useful in these cases: it provides the user with a simpler lemma to be proven, along with a full proof of the original goal that uses that lemma.

While preliminary, we believe that this experiment demonstrates the promise of combining lemma synthesis with automated proof search. Without lemma synthesis, Proverbot9001 cannot complete any of the 27 proofs when the needed lemma is not available. With lemma synthesis from `dilemma`, Proverbot9001 can provide a useful result in 7 cases, including a complete proof in 2 cases. In the future, we are interested to see how a more powerful automated prover, for example one that employs LLMs, would benefit from `dilemma`. To fully integrate `dilemma` with an automated prover, we also need to devise approaches to determine when `dilemma` should be invoked as part of the proof search.

6 Related Work

6.1 Lemma Synthesis

Our approach to lemma synthesis is closest to that of `lfind` [48], which also employs a form of data-driven synthesis based on sample valuations of the current proof state. However, both the class of lemmas being targeted and the form of data-driven synthesis are quite distinct. The `lfind` approach targets lemmas that can be used to rewrite a subterm within the current goal with a new term, and furthermore these lemmas must be *atomic*, i.e. true unconditionally. In `lfind` such synthesis is reduced to a form of data-driven program synthesis that produces candidate terms for goal rewriting. In contrast, `dilemma` targets lemmas that apply directly to the current goal, without rewriting, but that are only true conditional on a set of assumptions. Hence our key challenge is to synthesize candidate propositions to use as lemma *implicants*, and we describe how to reduce this task to data-driven invariant synthesis from positive and negative examples. We have demonstrated that `dilemma` can express some of the goal rewrites performed by `lfind` despite our focus on synthesizing implicants. In the future, it could be useful to combine `dilemma` and `lfind` for broader effectiveness in practice.

The most common prior technique for lemma synthesis in interactive theorem proving is *generalization* [3, 6, 9, 15, 24, 26, 34], whereby selected terms are replaced by fresh variables. Another class of approaches uses a fixed set of rewrite rules to derive candidate lemmas from the current state [8, 14, 28]. Compared to these approaches, our work is distinguished by the gain in expressiveness that results from reducing the problem to one of data-driven invariant synthesis. We also directly employ the generalization technique as part of our approach.

6.2 Theory Exploration

A related class of tools performs *theory exploration* to infer candidate specifications for a given API, typically through a form of bottom-up enumeration. This has been done in the context of both automated and interactive verification [10, 11, 23, 29, 30, 38, 44, 52]. The key difference versus `dilemma` is that these approaches are fundamentally *undirected* — they are not given a particular proof state from which to make progress. By leveraging sample valuations of the proof state, we are able to perform synthesis in a targeted way for the given state, including making use of the specific assumptions that hold in that state. However, `dilemma`'s underlying synthesizer is currently implemented as a simple bottom-up enumeration and thus could potentially benefit from the kinds of optimizations used in these tools.

6.3 Automated Proofs for Interactive Theorem Provers

As described in Section 1, there has been significant work on proof automation for interactive theorem provers. For example, *hammers* [5, 12, 31, 32] employ external constraint solvers to try to complete proofs, and neural theorem provers use machine learning models to predict the next tactic in a proof or to generate full proofs [4, 18, 19, 22, 25, 33, 41, 46, 49–51]. These techniques are increasingly powerful and can leverage existing lemmas, but they do not identify new lemmas that may be required in order to make progress. Hence it would be natural to integrate `dilemma` into an existing automated prover in future work. A key technical question is to determine when the prover is "stuck" and may need a new lemma to be synthesized.

6.4 Data-Driven Invariant Synthesis

Invariant synthesis from positive and negative examples is a well-studied technology, with various approaches and applications [2, 16, 20, 21, 37, 40, 53]. These past techniques are all employed in the context of fully automated verification (e.g., to infer sufficient loop invariants); to our knowledge, ours is the first application of data-driven invariant synthesis to interactive theorem proving. In the prior work there is a clear logical specification for the required invariant (e.g., a loop invariant must be inductive and imply a given post condition), while in our setting there is no such specification. Hence we have developed novel criteria for synthesizing candidate lemmas, based on the ideas of weakening existing assumptions and synthesizing missing assumptions, as well as criteria for filtering and ranking such candidates.

7 Conclusion

A key burden in interactive theorem proving today is the need for users to identify the right implementation-specific lemmas to define and prove, which will enable them to prove the high-level correctness properties that they really care about. We present a novel form of data-driven lemma synthesis that addresses an important gap in prior such tools, namely the lack of support for *implications*. Absent resource limitations, our approach subsumes the expressiveness of both the *generalization* technique for lemma synthesis [6] and the `lfind` data-driven lemma synthesis technique [48]. We have implemented our approach as a Rocq tactic called `dilemma` and have demonstrated its effectiveness on several benchmark suites.

There are several fruitful avenues for future work. First, as shown in Section 5, because `dilemma` targets lemmas that apply to the current goal state, it fails in many cases that involve forward reasoning, where a lemma is applied to an assumption. It would be interesting to extend our approach to explicitly search for such lemmas as well. Second, we would like to integrate `dilemma` into an existing proof automation tool, such as a neural theorem prover that leverages LLMs. This integration has the potential to greatly enhance the capabilities of automated provers by enabling them to synthesize new lemmas whenever they would otherwise not be able to make progress. Achieving this goal will require us to improve the efficiency of `dilemma` and devise heuristics for when `dilemma` should be invoked during an automated proof search.

8 Data-Availability Statement

As discussed in §4, we've implemented our approach in a Rocq tactic, `dilemma`. The source code of `dilemma` is on GitHub at https://github.com/ana-brendel/dilemma, along with documentation on how to install and run it. The benchmarks and result logs for `dilemma`'s evaluation can be found at https://github.com/ana-brendel/dilemma-benchmarks. Finally, a Docker image containing all of the above and scripts for running the benchmarks is available as an artifact[7].

Acknowledgments

Thanks to Yousef Alhessi, Sorin Lerner, and Alex Sanchez-Stern for early discussions on lemma synthesis and to Remy Wang for feedback on a draft of this paper. This work is supported by the National Science Foundation under grant CCF-2220891.

References

[1] Andrew W. Appel. 2024. *Verified Functional Algorithms*. Software Foundations, Vol. 3. Electronic textbook. http://softwarefoundations.cis.upenn.edu Version 1.5.5.

[2] 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.

[3] R Aubin. 1976. Mechanising Structural Induction. (1976).

[4] 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

[5] 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.

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

[7] Ana Brendel, Aishwarya Sivaraman, and Todd Millstein. 2025. Synthesizing Implication Lemmas for Interactive Theorem Proving. doi:10.5281/zenodo.16595407

[8] 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.

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

[10] 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.

[11] 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.

[12] 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. doi:10.1007/s10817-018-9458-4

[13] 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.

[14] M. Demba and K. Bsaies. 2004. Appropriate lemmae discovery. *Information Sciences* 163, 4 (2004), 221–237. doi:10.1016/j.ins.2003.06.012 Information Technology.

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

[16] 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.

[17] 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.

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

[19] Emily First, Markus N. Rabe, Talia Ringer, and Yuriy Brun. 2023. Baldur: Whole-Proof Generation and Repair with Large Language Models. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2023, San Francisco, CA, USA, December 3-9, 2023*, Satish Chandra, Kelly Blincoe, and Paolo Tonella (Eds.). ACM, 1229–1241. doi:10.1145/3611643.3616243

[20] 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.

[21] 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

[22] 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. doi:10.29007/ntlb

[23] 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.

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

[25] Daniel Huang, Prafulla Dhariwal, Dawn Song, and Ilya Sutskever. 2018. GamePad: A Learning Environment for Theorem Proving. *CoRR* abs/1806.00608 (2018). arXiv:1806.00608 http://arxiv.org/abs/1806.00608

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

[27] Dongseok Jang, Zachary Tatlock, and Sorin Lerner. 2012. Establishing Browser Security Guarantees Through Formal Shim Verification. In *Proceedings of the 21st USENIX Conference on Security Symposium* (Bellevue, WA) *(Security'12)*. USENIX Association, Berkeley, CA, USA, 8–8. http://dl.acm.org/citation.cfm?id=2362793.2362801

[28] 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.

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

[30] 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.

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

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

[33] Saketh Ram Kasibatla, Arpan Agarwal, Yuriy Brun, Sorin Lerner, Talia Ringer, and Emily First. 2024. Cobblestone: Iterative Automation for Formal Verification. *CoRR* abs/2410.19940 (2024). doi:10.48550/ARXIV.2410.19940 arXiv:2410.19940

[34] 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.

[35] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. 2009. seL4: Formal Verification of an OS Kernel. In *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles* (Big Sky, Montana, USA) *(SOSP '09)*. ACM, New York, NY, USA, 207–220. doi:10.1145/1629575.1629596

[36] Xavier Leroy. 2009. Formal verification of a realistic compiler. *Commun. ACM* 52, 7 (2009), 107–115. http://xavierleroy.org/publi/compcert-CACM.pdf

[37] 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.

[38] 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.

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

[40] 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.

[41] Aditya Paliwal, Sarah M. Loos, Markus N. Rabe, Kshitij Bansal, and Christian Szegedy. 2019. Graph Representations for Higher-Order Logic and Theorem Proving. *CoRR* abs/1905.10006 (2019). arXiv:1905.10006 http://arxiv.org/abs/1905.10006

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

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

[44] 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.

[45] Rocq Team 2025. Rocq Prover. https://rocq-prover.org/.

[46] 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. doi:10.1145/3394450.3397466

[47] Aishwarya Sivaraman, Alex Sanchez-Stern, Bretton Chen, Sorin Lerner, and Todd D. Millstein. [n. d.]. Data-Driven Lemma Synthesis for Interactive Proofs. Zendo. https://doi.org/10.5281/zenodo.7088869

[48] Aishwarya Sivaraman, Alex Sanchez-Stern, Bretton Chen, Sorin Lerner, and Todd D. Millstein. 2022. Data-driven lemma synthesis for interactive proofs. *Proc. ACM Program. Lang.* 6, OOPSLA2 (2022), 505–531. doi:10.1145/3563306

[49] Amitayush Thakur, George Tsoukalas, Yeming Wen, Jimmy Xin, and Swarat Chaudhuri. 2024. An In-Context Learning Agent for Formal Theorem-Proving. In *First Conference on Language Modeling*. https://openreview.net/forum?id=V7HRrxXUhN

[50] Kyle Thompson, Nuno Saavedra, Pedro Carrott, Kevin Fisher, Alex Sanchez-Stern, Yuriy Brun, João F Ferreira, Sorin Lerner, and Emily First. 2024. Rango: Adaptive Retrieval-Augmented Proving for Automated Software Verification. *arXiv preprint arXiv:2412.14063* (2024).

[51] Kaiyu Yang and Jia Deng. 2019. Learning to Prove Theorems via Interacting with Proof Assistants. *CoRR* abs/1905.09381 (2019). arXiv:1905.09381 http://arxiv.org/abs/1905.09381

[52] 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.

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

Received 2025-03-25; accepted 2025-08-12