Data-driven Lemma Discovery

Anonymous Author(s)

Abstract
Programs often leverage data structure libraries to provide useful and reusable abstractions. Modular verification of these programs naturally rely on specifications associated with these libraries that capture important properties about how these data structures are expected to be accessed and manipulated. However, because these specifications are written without awareness of how they will actually be used in a client-side verification task, they are often incomplete. In particular, specifications rarely expose relevant relational properties among a library’s methods and the predicates that comprise their specifications. These properties, while not necessarily relevant to understanding the behavior of the library, are often critical to enabling verification of the client. We refer to these missing specifications as lemmas and, in this paper, propose an automated data-driven procedure for discovering them. Our approach enables modular verification of client programs by driving a learning process using samples and counter-examples from an SMT solver to discover relations among the predicates that define library method specifications. Our learning framework seeks to find lemmas sufficient to discharge verification conditions associated with the client program. We have applied our technique on a collection of challenging data structure programs written in OCaml. Our results provide strong evidence on the feasibility and effectiveness of data-driven lemma discovery.

1 Introduction
Using a specification of a library function’s behavior when verifying a client of that function is a hallmark of automated program analysis and verification. Such specifications act as an interface between the client and library, allowing both to be independently verified. This modularity is particular beneficial when the library function is complex or its source code is unavailable. Unfortunately, these benefits come at a cost: the specifications provided by libraries are often incomplete and can omit details necessary for the verification of client programs.

As a simple example, consider the following program that appends two lists together using the operations provided by a Stack library:

```ocaml
let rec concat s1 s2 =
  if Stack.is_empty s1 then s2
  else Stack.push (Stack.top s1)
    (concat (Stack.tail s1) s2)
```

Listing 1. A simple client program.

Suppose we want to use the specifications of stack operations given in Figure 1 to verify that (a) the top of the result stack corresponds to the top of one of the input stacks, and (b) the result stack only contains members of s1 and s2. The specifications of both `concat` and the library operations are expressed in terms of two predicates on lists: `hd(l, u)`, which holds when element `u` is the head of the list `l`, and `member(l, u)`, which holds when element `u` is a member of the list `l`. We assume the underlying `List` library provides implementations of methods `hd` and `member` against which the semantics of these predicates are defined. Given these specifications, it is straightforward to generate a verification condition (VC) using techniques like weakest precondition inference [6] which can (hopefully) be discharged via an off-the-shelf SMT solver like Z3.

Our hopes are, unfortunately, dashed when Z3 fails to discharge our VC, returning instead a counterexample that is morally equivalent to:

\[ \exists u. hd(l, u) \land \neg \text{member}(l, u) \]

This counterexample is obviously incongruous with the intended semantics of `hd` and `member`, since an element `u` cannot be the head of a list `l` without also being a member of `l`. What has gone wrong here? Informally, these predicates were embedded as uninterpreted functions when translating the specifications to an SMT query, obscuring the intuitive relationships between `hd` and `member`. Indeed, this counterexample is inconsistent with the following sensible lemma about the semantics of these predicates:

\[ \forall l x, \, hd(l, x) \Rightarrow \text{member}(l, x) \]

Given this additional fact, Z3 is able to eliminate the spurious counterexample and discharge our VC.

Asking users to manually inspect counterexamples and provide such lemmas is counter to our goal of automated verification. An alternative approach might be to provide a library of lemmas about operations over a fixed collection of data types [10, 11]. While possible, this approach still requires significant manual effort, and must be applied for each datatype and operation. More importantly, it relies on the library designer to identify a set of useful facts salient to the properties maintained by the library. Given the potentially large set of such facts that may be generated, the feasibility of this approach is questionable, especially because only a small fraction of possible facts will be relevant to aid specific client-side verification demands. Indeed, theorem exploration tools like QuickSpec [22] precisely automate this search process using random testing methods. However, as we show in Sec. 6, Quickspec was unable to generate any interesting lemmas.
for most of our benchmarks; the lemmas it did discover were no more sophisticated than the lemma given in this very simple example, and thus insufficient to enable verification under a more demanding correctness property. For similar reasons, axiomatizing these facts directly within the logic used by the theorem prover is also problematic, especially since properties we often wish to ascribe to these predicates are inductive (e.g., if \( x \) is a member of a list \( l \), then \( x \) is also a member of any list \( l' \) for which \( l \) is a prefix). Encoding these properties directly within a theorem prover typically requires additional axiomatizations.

Rather than requiring manual intervention from users or relying on a collection of predefined lemmas, we propose a very different solution: automatically learning lemmas that are sufficient for an SMT solver to discharge the desired VCs posed during client-side verification. Inspired by recent work in data-driven approaches to specification and invariant inference [14, 19, 25], we develop an algorithm to learn semantic relationships between combinations of method predicates, or arbitrary predicates used to specify recursive algebraic data types (e.g., lists, trees, etc.). Learning is driven by counterexamples provided by an SMT solver as well as the output of concrete tests, and is structured to ensure that discovered lemmas are tailored to the verification task at hand. The hypothesis space used by the learner consists of first-order prenex-quantified logical formulae built from constants, program variables, and the set of predicates (such as \( \text{hd} \) and \( \text{member} \)) used in the specification of library methods. We use decision tree learning [21] to discover hypotheses that separate all positive and negative samples, where a negative sample is a counterexample returned by the SMT solver reflecting an incorrect interpretation of these predicates and their relation to one another, and positive samples are the result of running the client program under different test cases, recording the values of these method predicates during program execution.

It is instructive to note that we cannot simply apply existing data-driven methods [14, 19] that would attempt to systematically enumerate the set of possible lemmas to this problem. Indeed, these tools timeout when asked to synthesize a lemma sufficient to discharge the VCs in the above example. This is because they use learning techniques solely to separate a set of test inputs that cause the post-condition to hold from a set of inputs that cause the post-condition to fail. However, in our setting, there are no such "bad" inputs - if the client program and library are written correctly, every concrete test will satisfy the program’s VCs and the post-condition, even though the desired implication cannot be verified by the theorem prover. In contrast to these efforts, our approach uses SMT-provided counterexamples to generate infeasible interpretations of these predicates, and concrete test data to generate feasible interpretations, thus enabling a CEGIS-style inference methodology that allows us to postulate relations among method predicates sufficient to prove the post-condition. Notably, running our tool on this program took less than one second to generate the desired candidate lemma. Subsequent verification of the lemma against a known implementation of the predicates was equally fast.

To show how our algorithm deals with modular verification of programs that make extensive use of algebraic datatypes and to demonstrate how it cooperates with other specification inference methods, we have implemented a fully automated lemma discovery pipeline in Ocaml. This pipeline takes as input Ocaml programs that may call library code which encapsulates uses of these datatypes. Specifications of library function calls are inferred by a third-party [14, 19, 25], and generated VCs are checked by an SMT solver. If the solver validates the VCs, verification is complete. If the solver provides a counter-example, then our lemma inference algorithm tries to strengthen the VCs by providing lemmas to fix the proof. To provide added assurance on the soundness of our approach, inferred lemmas are verified against library code using other verification tools [12] when the source is available. This pipeline is cleanly separated; there are no interdependencies between the specification inference algorithm used to postulate library specifications and post-discovery lemma verification. Our evaluation consists of a number of client programs that make extensive and sophisticated use of data structure libraries, for example those described by Okasaki [16].

Our key contribution is an automated data-driven method to discover relationships among various predicates found in the specification of data-structure manipulating libraries, expressing these relations as lemmas that enable verification of client programs that use themx. Specifically, we:

1. Frame client-side verification as a counterexample-guided search over a space of relations that ascribe an interpretation to the predicates found in the specification of libraries the client uses.
2. Devise a data-driven sampling-based learning technique to make this search process practical and scalable. Our learning framework assumes no programmer involvement in defining features, It is also modular, requiring no analysis of library code to enable inference.

\[
is_{\text{empty}}(s) \equiv \forall u, \neg \text{member}(s, u)
\]

\[
\text{top}(s, v) \equiv \forall u, (v = u) \implies \text{hd}(s, u)
\]

\[
\text{tail}(s, v) \equiv \forall u, \text{member}(v, u) \implies \text{member}(s, u)
\]

\[
\text{push}(x, s, v) \equiv \forall u, (\text{hd}(v, u) \iff (u = x)) \land \\
(\text{member}(v, u) \iff (u = x) \lor \text{member}(s, u))
\]

Figure 1. Specifications of Stack operations. The special variable \( v \) represents the result of the method.
3. Verify the correctness of discovered specifications against library implementations when their sources are available.

4. Evaluate our ideas in a tool, Elrond, which we use to analyze a comprehensive set of realistic and challenging functional (OCaml) data structure programs.

The remainder of the paper is structured as follows: we motivate our ideas in Sec. 2. A formal characterization of the problem is given in Sec. 3. A detailed presentation of the algorithm used to manifest these ideas in a practical implementation is given in Sec. 4. A description of the implementation is provided in Sec. 5; details of our evaluation results are explained in Sec. 6. Related work and conclusions are given in Sec. 7 and Sec. 8.

2 Motivation and Overview

To motivate our approach, we revisit the example introduced in the previous section. Our goal is to verify the correctness of the following post-condition holds after a call to \texttt{concat}(s_1, s_2):

\[ \forall u, (hd(l, u) \implies \text{hd}(s_1, u) \lor \text{hd}(s_2, u)) \land (\text{member}(v, u) \implies \text{member}(s_1, u) \lor \text{member}(s_2, u)) \]

The left conjunct requires that the top element of the output stack be the top element of one of the input stacks, and the right conjunct requires that elements in the output stack be found in at least one of the input stacks. Using this post-condition and the implementation of \texttt{concat}, we generate a set of verification conditions \( \Phi_{\text{concat}} \). We then require specifications of the Stack library, which we will call \( \Sigma_{\text{Stack}} \), to entail the verification conditions: \( \Sigma_{\text{Stack}} \implies \Phi_{\text{concat}} \). In this case, the verification conditions \( \Phi_{\text{concat}} \) are captured by the following formula:

\[
\begin{align*}
\forall v, v_{\text{top}}, v_{\text{tail}}, v_{\text{concat}}, & \quad (\text{is\_empty}(s_1) \implies v = s_2) \land \\
(\neg \text{is\_empty}(s_1)) & \implies (\text{top}(s_1, v_{\text{top}}) \land \text{tail}(s_1, v_{\text{tail}})) \land \\
(v_{\text{concat}}, (\text{hd}(v_{\text{concat}}, u) & \implies \text{hd}(v_{\text{tail}}, u) \lor \text{hd}(s_2, u)) \land \\
\text{member}(v_{\text{concat}}, u) & \implies \text{member}(v_{\text{tail}, u} \lor \text{member}(s_2, u))) \land \\
\text{push}(v_{\text{top}}, v_{\text{concat}}, v) & \implies \\
(v_{\text{concat}}, (\text{hd}(v, u) & \implies \text{hd}(s_1, u) \lor \text{hd}(s_2, u)) \land \\
\text{member}(v, u) & \implies \text{member}(s_1, u) \lor \text{member}(s_2, u))) \\
(\Phi_{\text{concat}})
\end{align*}
\]

These conditions can be derived using a standard weakest precondition analysis. The first conjunct captures the true branch of the conditional in \texttt{concat}’s definition, which concludes that if \( s_1 \) is empty, then the list returned by \texttt{concat} is \( s_2 \); the second conjunct captures the behavior of the false branch. Its four sub-conjuncts reflect the following four properties about \texttt{concat}’s implementation when \( s_1 \) is not empty: (1) the top element of stack \( s_1 (v_{\text{top}}) \) is the head of its list representation; (2) elements in the tail of \( s_1 (v_{\text{tail}}) \) are members of \( s_1 \)'s list representation; (3) the result of the recursive call to \texttt{concat} is a list whose head element is either the head of \( s_1 \)'s tail or the head of \( s_2 \), and every element of this list is either a member of \( s_1 \)'s tail or a member of \( s_2 \); and (4) the head element in the result list \( v \) is the head element of \( s_1 (v_{\text{top}}) \) and a member in this result is either the head element of \( s_1 \) or a member of the list returned by the recursive call to \texttt{concat} (\( v_{\text{concat}} \)).

Suitably simplified, the property we wish to verify (\( \Sigma_{\text{Stack}} \implies \Phi_{\text{concat}} \)) is:

\[
\begin{align*}
((\forall u, \text{member}(v_{\text{tail}}, u) \implies \text{member}(s_1, u)) \land \\
((\forall u, \text{member}(v, u) \implies (\Sigma_{\text{Stack}})) \land \\
(hd(s_1, u) \lor \text{member}(v_{\text{tail}}, u) \lor \text{member}(s_2, u))) \implies \\
(\forall u, \text{member}(v, u) \implies (\text{member}(s_1, u) \lor \text{member}(s_2, u))) \\
(\Phi_{\text{concat}})
\end{align*}
\]

Verifying this property with an off-the-shelf theorem prover returns the following counterexample, where \( a \) is some unknown integer constant:

\[
\begin{align*}
\forall l, (hd(l, u) & \iff ((l = s_1) \land (u = a)) \land \\
\text{member}(l, u) & \iff ((l = v) \land (u = a)) \\
(\text{Cex})
\end{align*}
\]

To understand this counterexample, we focus our attention on the encoding of \texttt{concat}’s false branch in \( \Phi_{\text{concat}} \). The counterexample asserts the top element of \( s_1 \) is some value which is not a member of \( s_1 \) or \( s_2 \). The counterexample additionally asserts that \( v \) is a member of the list returned by \texttt{concat}, thus violating the second conjunct of the post-condition, \( \text{member}(v, u) \implies \text{member}(s_1, u) \lor \text{member}(s_2, u) \). In other words, the counterexample posits that \( \text{member}(s_1, a) \) is false, \( \text{member}(v, a) \) is true, and \( \text{hd}(s_1, a) \) is true. These claims clearly violate the intended semantics of \text{member} \ and \text{hd}; it should not be possible for \( s_1 \) to have a head which is not also a member. However, absent additional facts regarding these kinds of relationships between \text{hd} \ and \text{member}, the counterexample is indeed valid. One solution is to augment \( \Sigma_{\text{Stack}} \) to disallow this particular counterexample, but this quickly results in a large set of overly specific counterexamples, each of which considers specific input configurations, but which fail to generalize.

Instead, we propose to augment \( \Sigma_{\text{Stack}} \) with a stronger and more general fact that prohibits this and similar counterexamples, while also being simpler, making it easier for users to understand (and potentially to verify implementations of the predicate when their implementations are available).

Of course, it is possible to suggest a fact that is too strong: observe that we can reject all possible assignments of the top element \( v_{\text{top}} \) of \( s_1 \) by adding the following stipulation to \( \Sigma_{\text{Stack}} \) that simply mandates that \( s_1 \) be empty:

\[
\forall u, \neg \text{hd}(s_1, u) \\
(\text{L}')
\]

\[\text{1} \] Simplification details are provided in the supplemental material.
When a formula \( \Sigma \Rightarrow \Phi \) cannot be proven, we try to find a consistent strengthening of \( \Sigma \). This is an example of an abductive inference problem which tries to find a formula \( L \) such that \( \Sigma \land L \not\models \bot \) and which is sufficient to prove the desired verification condition \( \Phi \). To do so, we observe that such a formula can be thought of as a separator that identifies which interpretations of method predicates are consistent with their intended semantics. Figure 2 is a visual depiction of our approach.

We construct this classifier using a set of samples, or interpretations of the free variables in \( \Sigma \) and \( \Phi \), and a formula capturing whether the method predicates hold for each of these interpretations. Intuitively, a sample identifies which semantic relationships defined by the method predicates hold between the variables in a counterexample identified by the verifier. To train our classifier, we label the samples consistent with \( \neg(\Sigma \Rightarrow \Phi) \) as "negative", so that the solver can rule out the associated counterexamples. In addition to ruling out spurious counterexamples, we wish to bias our learner towards explanations that are consistent with the semantics of the underlying method predicates. To do so, we generate "positive" samples that are consistent with \( \Sigma \Rightarrow \Phi \) and the execution of method predicates on concrete data values using a test generation framework like QuickCheck [4]. Such samples correspond to the overlapping area between the blue Datatype Instance and green \( \Sigma \Rightarrow \Phi \) circles in Figure 2. Our learning algorithm is thus tasked with building a classifier that separates negative and positive samples. This separation is encoded as a logical formula defined in terms of the set of method predicates used to specify library functions.

The set of possible classifiers comprises the hypothesis space for our learner. Figure 2 shows three classifiers, the weakest one given by a solid red line and the two stronger ones given by dashed red lines. These dashed lines represent candidate lemmas that are sufficient to verify \( \Phi \), but which are inconsistent with the semantics of our method predicates, as with \( L' \) above. Our algorithm uses a test generation framework to filter out these sorts of candidate lemmas. Relying on a test-generation framework to identify positive samples means that the set used to train our learner might be incomplete, as can happen when there are only few concrete data values that are inconsistent with our method predicates. While a large sample size can give us confidence in the correctness of a candidate lemma, guaranteeing that it covers all possible instantiations of the type ultimately requires verifying it against the library implementation itself. We drop the "candidate" qualifier from those facts that are so verified and call them verified lemmas.

**Hypothesis space** Our data-driven inference algorithm limits the shape of candidate lemmas so that they are both amenable to automatic verification and are strong enough to verify many desirable post-conditions. To enable automated verification, the shape of lemmas considered are restricted to prenex universally-quantified propositional formulas over datatype values and variables representing arguments to the predicates under consideration. Some example candidate lemmas for our running example are:

\[
\{ \forall u, member(l, u), \quad \forall u, \neg member(l, u), \\
\forall u, member(l, u) \land hd(l, u), \quad \forall u, \neg hd(l, u), \\
\forall u, hd(l, u) \implies member(l, u), \quad \ldots \} \]

which contains, among other candidates, the desired lemma. All atomic literals in generated formulae are applications of method predicates to quantified variables; i.e., the quantifier-free parts of the lemma are propositions whose literals are.

The literals in the above formulas are simply applications of \( hd \) and \( member \) to \( l \) and \( u \), for example. These literals form the algorithm’s feature set:

\[
\{ hd(l, u), member(l, u) \}
\]

**Samples and Feature vectors.** The candidate lemmas in our hypothesis space allow us to disallow combinations of method predicates which are never consistent. For example,
the (invalid) candidate lemma \( \forall l, u, \neg \text{member}(l, u) \) would entail that there is no list which contains an element (i.e. all lists are empty). Our algorithm uses the assignments to the variables of a negative sample to build a feature vector that describes the uses of the method predicates in that sample. As an example, the last two columns of Table 1 form a feature vector for a particular instantiation of variables drawn from a negative sample. The three rows shown in the table are satisfiable under some interpretation of the method predicates, which means at least one of them should be a negative feature vector. The first row corresponds to the assertion that \( \mathit{hd}(s_1, a) \wedge \neg \text{member}(s_1, a) \), for example. Each row corresponds to a potential negative feature, which our classifier should learn to disallow (thus disallowing the entire counterexample).

As we have seen though, we only want to disallow features that are truly incompatible with the semantics of the method predicates. To identify which are feasible, our algorithm generates random inputs of the client program to get inputs representing positive samples. For our current example, running these inputs using blackbox implementations of \( \mathit{hd} \) and \( \text{member} \) allows us to also extract positive feature vectors from these samples. For example, letting \( s_1 = [a; b; a; b] \) and \( s_2 = [b] \), we get the assignment:

\[
    v = [a; b; a; b], v_{\text{top}} = a, v_{\text{tail}} = [b; a], v_{\text{concat}} = [b; a; b]
\]

These assignments collectively form a positive sample. Similar to how we constructed a negative feature vector for C\text{sex}, we can assign concrete values \( l \) and \( u \) to build a positive feature vector. Table 2 illustrates some positive samples and their corresponding feature vectors. Consider the row \( l = s_2 \) (which is \( [b] \)) and \( u = a; \) under this assignment, \( \mathit{hd}(l, u) = \mathit{hd}([b], a) \) is false; \( \text{member}(l, u) = \text{member}([b], a) \) is also false.

If \( \Sigma \implies \Phi \) is valid, only positive samples exist (since its negation would not be satisfiable). This is generally not the case, however; in fact, it is quite possible that positive and negative feature vectors may overlap. We can observe this for the vectors in Table 1 and Table 2: the interpretation of \( \mathit{hd}(l, u) = \text{false} \) and \( \text{member}(l, u) = \text{true} \) occurs in both tables, for example.

However, notice that the negative sample is truly negative because there exists at least one negative feature vector which describes an impossible situation, i.e. the first row of Table 1, which represents the case when \( u \) is the head of \( l \) but is not a member of \( l \). The other feature vectors (second and third rows) are reasonable and can manifest in a positive sample. On the other hand, all positive feature vectors are always sensible. We use this observation to identify the actual negative feature vectors as those arising from a negative sample but not from a positive sample. Table 3 shows the partition of positive and negative feature vectors for our running example. When all the potential negative feature vectors are included in the set of positive feature vectors, our hypothesis space is not rich enough to separate the two samples. In this scenario, our algorithm enriches the hypothesis space in an attempt to remedy the situation by searching for a more refined (i.e. complex) lemma.

### Classification

Given this set of positive and negative feature vectors, the role of the classifier is to build a separator over the training data. One such classifier for the set in Table 3 is, \( \mathit{hd}(l, x) \implies \text{member}(l, x) \) Augmenting \( \Sigma \) with the candidate lemma, \( \forall x, \mathit{hd}(l, x) \implies \text{member}(l, x) \), is sufficient to remove all negative feature vectors and discharge the verification conditions. Using an alternative verification tool, we can verify verify that this lemma holds for given implementations of \( \mathit{hd} \) and \( \text{member} \), allowing us to drop the "candidate" label from this lemma.

### 3 Formalization

**Setup.** The input to our verification pipeline has three components: data structure libraries, a client program, and assertions (pre- and post-conditions) characterizing the client’s input and output. Because we are primarily interested in functional data structure verification, we assume library and client programs are pure, i.e. have no mutable references or data structures. We also assume that specifications for library methods are available, either provided by the programmer or inferred via specification inference [19, 25], and that verification conditions for the client program can be generated.
Our expectation is that an underlying theorem prover (e.g., a SMT solver) can prove \( \Sigma \implies \Phi \), where \( \Sigma \) is the library specification and \( \Phi \) are the verification conditions (VCs) built from a client program and its pre- and post-conditions. In the remainder of the paper, we refer to \((\Sigma, \Phi)\) as the verification query whose validity we wish to establish. Both \( \Sigma \) and \( \Phi \) are sentences built from logical connectives \((\wedge, \vee, \implies)\) over prenex universally-quantified propositional formulae.

We restrict our attention to the automated verification of functional programs that use data structure libraries whose functions access and construct instances of inductively-defined algebraic datatypes (e.g., list, trees, heaps, tries, etc.). Method predicates over these types are used in both specifications and VCs, and are treated as uninterpreted function symbols by the underlying solver.

If the solver fails to prove \((\Sigma, \Phi)\), the verification task seeks to find a formula \( L \) that strengthens \( \Sigma \) and which is sufficient to prove \( \Phi \). Superficially, this problem bears similarity to maximal specification synthesis [1], as the client program may make multiple calls to library functions. However, given that our proof obligations are expressed over predicates describing inductively-defined data structures, abductive inference methods based on e.g., quantifier elimination [3, 8] are not applicable. We therefore choose instead to attack this problem using data-driven methods, using data to ensure that whatever \( L \) is discovered must be consistent with the underlying semantics of method predicates.

### 3.1 Learning System

To preserve decidability, we limit the hypothesis space of our learning system to a subset of effectively propositional sentences, defined in terms of universal quantified formula whose quantifier-free component serves as a classifier, expressed as a Boolean combination over a term consisting of applications of interpreted base relations and uninterpreted functions.

**Definition 3.1 (Feature).** A feature is an uninterpreted predicate applied to quantified variables and whose sort is expressed in terms of base types and the algebraic datatypes manipulated by the library. Notice that a feature is similar to a literal in first-order logic without support for nested applications (e.g. \( \text{hd}(l, \text{member}(l, u)) \)) or applications of constants (e.g. \( \text{hd}(l, S) \)).

**Definition 3.2 (Feature Set).** For a given set of predicates \( \mathcal{P} \) and variables \( X \), the feature set \( \mathcal{F}(\mathcal{P}, X) \) is the set of type correct features of \( \mathcal{P} \) over \( X \).

In our motivating example, we use the predicate set \( \mathcal{P} \equiv \{ \text{hd}, \text{member}, =_a \}, X \equiv \{ l : \text{list}_a, u : a \} \), yielding a corresponding feature set, \( \{ \text{hd}(l, u), \text{member}(l, u) \} \). Here, \( a \) stands for some base type, preventing the equality predicate \( =_a \) from being applied to an instance of another algebraic datatype. Thus \( l=_{a} l' \) is not considered to be a valid application.

**Definition 3.3 (Hypothesis Space).** A hypothesis space is the set of all universally quantified Boolean combinations of elements drawn from a feature set. Specifically, the hypothesis space for a given verification query \((\Sigma, \Phi)\) and predicate set \( \mathcal{P} \) is the set of formulas in prenex normal form which have quantifier prefix \( \forall X \), where \( X \) is a sequence of variables, and quantifier-free parts built from \( \mathcal{F}(\mathcal{P}, X) \), logical connectives \((\wedge, \vee, \neg, \implies)\), and constants \( T \) (true) and \( \bot \) (false). The hypothesis space over \( \mathcal{P} \) and \( X \) is denoted \( \mathcal{H}(\mathcal{P}, X) \).

The hypothesis space grows along with the number of free variables. For a fixed predicate set, the hypothesis space with different universal quantified variables \( X \) will build a hierarchy that preserves the following property: \( X_1 \subseteq X_2 \implies \mathcal{H}(\mathcal{P}, X_1) \subseteq \mathcal{H}(\mathcal{P}, X_2) \). Every candidate lemma can be converted to a candidate lemma in a hypothesis space with more quantified variables. On the other hand, some candidate lemmas can only be represented with a minimum cardinality for \( X \). For example, consider a predicate \( \text{order}(l, u, v) \) that relates an element \( v \) that appears in the tail of the list rooted at \( u \). Capturing the behavior of this predicate requires the following axiomatization of transitive closure that is defined by three variables:

\[
\forall l u v w. \text{order}(l, u, v) \land \text{order}(l, v, w) \land \neg\text{order}(l, u, v) \implies \text{order}(l, u, w)
\]

This property cannot be encoded in a hypothesis space with only two variables, because the statement of transitivity relates three distinct elements.

### 3.2 Lemma Inference

**Definition 3.4 (Candidate Lemma).** For a verification query \((\Sigma, \Phi)\) suppose that \( \Sigma \implies \Phi \) is invalid, and a predicate set \( \mathcal{P} \) contains all the method predicates in \( \Sigma \) and \( \Phi \). Then, a formula \( L \) drawn from \( \mathcal{H}(\mathcal{P}, X) \) where \( X \) is a set of quantified variables, is a candidate lemma for \( \Phi \) if:

1. \( L \land \Sigma \) is satisfiable
2. \( L \land \Sigma \implies \Phi \) is valid

As illustrated in Figure 2, our goal is additionally to discover candidate lemmas that rule out all samples in \( \neg(\Sigma \implies \Phi) \).

**Definition 3.5 (Sample).** A sample \( s \) of a formula \( \phi \) which involves unknown method predicates is a satisfying instantiation of all quantified variables and a Boolean-valued interpretation of each method predicate applied to those variables in \( \phi \), denoted as \( s \models \phi \).

**Definition 3.6 (Feature vector).** Let \( \mathcal{P} \) be the predicates in a verification query. A feature vector \( v \) is a Boolean vector that encodes an assignment to all features in the feature set \( \mathcal{F}(\mathcal{P}, X) \).

For example, \( \{ \text{true}, \text{false} \} \) is a feature vector for the feature set \( \{ \text{hd}(l, u), \text{member}(l, u) \} \) corresponding to the second row of Table 1.
Definition 3.7 (Sample consistency). A feature vector $v$ under feature set $F$ is consistent with a sample $s$ of $\Sigma \Rightarrow \Phi$ if the variables in the feature can be instantiated with variables in $s$ such that:

$$s \models \bigwedge_{f_i \in F_{\text{subst}}} f_i \iff v[i]$$

where $F_{\text{subst}}$ is $F_{x \in X/y \in s}$, denoted as $s \rightsquigarrow_F v$.

For our motivating example, each of the rows in Table 1 are feature vectors consistent with $\text{Cex}$.

Definition 3.8 (Lemma consistency). Let $FVec^+$ and $FVec^-$ be a set of feature vectors under feature set $F$. $FVec^+$ is strongly consistent with a candidate lemma $L$ (written $\Sigma' \Rightarrow^+ FVec^+$) if $L$ holds for all feature vectors in $FVec^+$. We say that $FVec^-$ is strongly inconsistent with $L$ (written $\Sigma' \Rightarrow^- FVec^-$) if $L$ does not hold for all feature vectors in $FVec^-$. The positive feature vectors in Table 3 are strongly consistent with candidate lemma $\forall x, \text{hd}(l, x) \Rightarrow \text{member}(l, x)$; and the negative feature vectors in Table 3 are strongly inconsistent with this lemma.

Corollary 3.9. For a candidate lemma $L$, sample $s \models L$ iff all feature vectors consistent with $s$ are also strongly consistent with $L$.  

Theorem 3.10. Suppose a verification query $(\Sigma, \Phi)$ involves method predicates $P$. $L$ which is drawn from hypothesis space of $P$ is a candidate lemma of $\Phi$ if $L \land \Sigma$ is satisfiable and for all samples $s$ of $\neg(\Sigma \Rightarrow \Phi)$, there exists a feature vector $v$ where $s \rightsquigarrow_F v$ and $L \Rightarrow^+_F v$.

3.3 Verified Lemmas

Observe that determining whether a formula is a candidate lemma depends on the feature vectors used to ascertain lemma consistency. While our algorithm attempts to bias the learned classifier against generating incoherent candidate lemmas by providing positive samples that are consistent with some blackbox implementation of the method predicates, the validity of discovered candidate lemmas cannot be established merely by the availability of tests. However, if the implementation of the methods from which these predicates are derived is available, it is possible to verify that the candidate lemma is indeed a true lemma with respect to the predicates’ semantics. In our evaluation, we have been able to show that the inferred candidate lemmas from our tool can be easily and mechanically verified, using existing verification tools [12]. We note that even if these implementations are not available, lemma discovery can still play a useful role in a developer’s verification workflow, generating relations among datatype and library methods that reveal interesting properties about the library which may otherwise have been difficult to glean.

4 Algorithm

Our lemma inference algorithm assumes the three key components: a property-based random sampler $\text{Sampler}$, a lemma satisfiability checker $\text{IsSat}$ and a classifier learner $\text{Learner}$. The $\text{Sampler}$ takes a first-order formula as input, and attempts to find a counterexample by random sampling, returning data instances which violate this specification as samples based on these counterexamples. $\text{IsSat}$ checks whether there is a satisfying assignment to variables and member predicates (i.e. a sample) if one exists for the negation of the verification query, returning unsat otherwise. The $\text{Learner}$ takes positive and negative feature vectors, $FVec^+$ and $FVec^-$ as input and returns a candidate lemma that splits $FVec^+$ and $FVec^-$ i.e., ensures that $FVec^+ \land FVec^- = \emptyset$. We use a decision tree algorithm [21] as the underlying learning framework.

Algorithm 1 presents our lemma learning algorithm. The algorithm has two subroutines: DiscoverLemma infers candidate lemmas for a verification query under a fixed feature set $F$ (lines 15–38), and SearchHyp iteratively grows the hypothesis space used by DiscoverLemma when the existing hypothesis space is insufficient to find a classifier.

Lines 4–5 initialize the hypothesis space using the method predicates from the verification query and a single fresh variable. The subsequent loop iteratively grows this space by adding a new quantified variable to $X$ (line 13) each time a candidate lemma cannot be learned. The feature set grows (line 8) with the set of quantified variables. $\text{AllFeatureSet}$ generates the feature set under all possible type assignments of $X$. For example, given a variable set $X, \{x_1, x_2, x_3\}$, and $P, \{\text{hd, member, >}\}$, $\text{AllFeatureSet}$ generates the following feature sets:

- $\{\text{hd}(x_1, x_3), \text{hd}(x_2, x_3), \text{member}(x_1, x_3), \text{member}(x_2, x_3)\}$
- $\{\text{hd}(x_1, x_2), \text{hd}(x_1, x_3), \text{member}(x_1, x_2), \text{member}(x_1, x_3), x_2 > x_3, x_3 > x_2\}$
- $\{x_1 > x_2, x_2 > x_1, x_1 > x_3, x_3 > x_1, x_2 > x_3, x_3 > x_2\}$
- $\{x_1 > x_2, x_2 > x_1, x_1 > x_3, x_3 > x_1, x_2 > x_3, x_2 > x_3\}$

where $x_1, x_2$ and $x_3$ are elements of a list

The algorithm iterates over the multiple feature sets generated by this procedure, using DiscoverLemma to identify a suitable candidate lemma. If one is found, it is returned and the algorithm terminates. If not, the algorithm grows the hypothesis space by increasing the set of variables and continues looping.

Lines 15–38 defines the DiscoverLemma procedure which tries to find a candidate lemma using the feature set $F$. Lines 16 and 17 initializes the positive feature vectors $FVec^+$ and negative feature vectors $FVec^-$ to $\emptyset$, and sets $L$, the candidate lemma to be generated, to true. Then, the counterexample guided loop shown in line 18–38 refines $L$ based on these feature vectors.

The beginning of this loop checks to see if $L$ is strong enough to establish $\Sigma \land L \Rightarrow \Phi$. If so, it is sufficient to prove
Algorithm 1: Lemma Learning

1. Input: $\Sigma \implies \Phi$, the formula expected be verified;
2. Output: $L$, the candidate lemma;
3. Algorithm SearchHyp() 
   4. $P = \text{predicates in } \Sigma \implies V$;
   5. $X = \{\text{NewVariable}\}$;
   6. while true do
      7. for $F$ in $\text{AllFeatureSet}(P, X)$ do
         8. if $\text{is lemma}$, $L = \text{DiscoverLemma}(F)$;
         9. if $\text{is lemma}$ then
            10. return $L$;
         11. end
      12. end
      13. $X = X \cup \{\text{NewVariable}\}$;
   14. end
15. Procedure DiscoverLemma($F$)
   16. $FVec^+ = FVec^- = \emptyset$;
   17. $L = \top$;
   18. while true do
      19. $s^- = \text{IsSAT}((\Sigma \wedge L) \implies \Phi)$;
      20. if $s^- = \text{unsat}$ then
         21. if $\text{IsSAT}((\Sigma \wedge L) \neq \text{unsat}$ then
            22. return false, $L$;
         23. end
      24. return true, $L$;
      25. end
26. $FVec_{cex} = FVec_{FromModel}(s^-, F)$;
27. $FVec^- = FVec^- \cup FVec_{cex}$;
28. do
29. $FVec^- = FVec^- - FVec^+$;
30. $L = \text{Learner}(FVec^+, FVec^-)$;
31. $s_u = \text{Sampler}(L)$;
32. $FVec_{campler} = FVec_{FromSample}(s_u, F)$;
33. $FVec^+ = FVec^+ \cup FVec_{campler}$;
34. while $s_u \neq \emptyset$;
35. if $\forall v \in FVec_{cex}, v \notin FVec^- \text{ then}$
36. return false, $L$;
37. end
38. end

Our feature vector space is finite because feature vectors are Boolean vectors with size $|F|$. We can enumerate all possible feature vectors and ask if $s$ is consistent with each of them. For example, a possible query of $\{T, \top\}$ might ask: “for all list instances $l$ in the model, does there exist an integer $u$ that makes both $hd(l, u)$ and $\text{member}(l, u)$ true?”. In other words, a feasible query might be: $\exists u, \text{head}(l, u) \wedge \text{member}(l, u)$. Then we can evaluate this formula under sample $m$ to check if $(T, \top)$ is consistent with $s^-$.

According to Theorem 3.10, there exists at least one negative feature vector in $FVec_{cex}$. The algorithm updates the current $FVec^-$ by greedily adding all feature vectors consistent with $s^-$ to $FVec^-$ on line 27. To avoid adding feature vectors which should not be added to $FVec^-$ (e.g. the feature vector in $FVec^+$), $FVec^-$ is refined before each call to Learner. On line 29, we remove feature vectors appearing in $FVec^+$ from $FVec^-$ to preserve the constraint that $FVec^- \cap FVec^+ = \emptyset$.

There is a loop on line 28–34 that updates $FVec^+$ with randomly sampled positive data. The Learner learns a classifier based on positive and negative feature vectors and updates $L$ on line 30. The property-based sampler Sampler tests $L$ to find if there exist an assignment $s_{-L}$ to the variables in $X$ which violates $L$. DiscoverLemma assumes the existence of a runnable implementation $I(P)$ of $P$. Function $FVec_{FromSample}$ extracts feature vectors by simply evaluating $F$ under the assignment $s_{-L}$ and implementation $I(P)$. As these feature vectors are the result of running $I(P)$, they should be added to the set of positive feature vectors (line 33). If the $s_{-L}$ returned by Sampler is empty, which means Sampler believes $L$ is valid for all assignments to $X$, then the loop ends.

Line 35 checks for progress after this inner loop ends. If the counter-example $s$ did not provide new feature vectors for $FVec^-$, we conclude that the candidate lemma does not exist in the current hypothesis space and DiscoverLemma returns false.

Theorem 4.1 (Soundness). For a given verification query $(\Sigma, \Phi)$, Algorithm 1 always returns a candidate lemma of $\Sigma \implies \Phi$, or loops forever.

Theorem 4.2 (Decidability). For a given verification query $(\Sigma, \Phi)$, DiscoverLemma in Algorithm 1 always halts.

5 Implementation

We have implemented a verification pipeline, called Erond, for the modular verification of OCaml programs that are clients of libraries of functions manipulating algebraic data types, shown in Figure 3. Verification conditions for client programs are generated by weakest liberal precondition predicate transformers. Erond does not infer automatically inductive loop invariants for recursive functions in client programs, and expects client programs to provide, in addition to any pre- and post-conditions, any necessary loop invariants.
Figure 3. Overview of Elrond’s workflow.

We derive specifications for library methods used by the client using DOrder, a specification inference framework tailored for functional data structures [25]. We attempt to verify a client-supplied post-condition given library specifications (Σ), along with any pre-conditions, and VCs (collectively referred to as Φ in our discussion) using an SMT solver. If verification fails, Elrond attempts to learn a lemma using Algorithm 1. If verification succeeds, the candidate lemma can be verified against an implementation of its method predicates. Elrond uses the Z3 SMT solver [5] for its satisfiability checking component, IsSat. It implements Learner using the C4.5 decision tree algorithm [21], and Sampler using the QuickCheck property-based testing framework [4].

5.1 Predicates

We assume specifications are given using a fixed set of predicates for the various datatypes supported by system. For example, our specification inference algorithm for libraries whose methods manipulate lists need to be expressed in terms of the following predicates:

1.  \(hd(l, u)\): element \(u\) is the head of the list \(l\).
2.  \(member(l, u)\): element \(u\) is a member of the list \(l\).
3.  \(order(l, u, v)\): element \(v\) appears after element \(u\) in the list \(l\), separated by an arbitrary number of elements less than or equal to length \((l) – 2\).

Our implementation supports inference over stacks, binary trees, heaps, queues, splay trees, leftist trees, tries, streams and sets, and provides different predicate sets tailored for that type used in the specification of libraries.

5.2 A Decision Tree Classifier

A decision tree is a classifier that fully separates positive and negative samples. Decision tree learning uses “information gain” as a measure to learn simpler and more general classifiers that guide the inference algorithm to infer weaker (aka more general) candidate lemmas given a set of positive and negative samples.

A decision tree (DT) in lemma inference is a binary tree that represents a Boolean combination of a feature set \(\mathcal{F}\). Each inner node of the tree is labeled a feature \(f \in \mathcal{F}\); this use of DTs is similar to their use in other learning-based program verification and specification inference tasks [9, 24]. Each leaf of the tree is labeled either positive + or negative −.

To evaluate an input feature vector \(v\) which is an assignment of \(\mathcal{F}\), we trace a path down from the root node of the tree, going to a true branch or a false branch at each inner node whose labeled feature is \(f\) depending on \(v(f)\) which is the value of its feature under assignment \(v\). The output of the tree on \(v\) is the label of the leaf reached by this process.

To evaluate an input feature vector \(v\) which is an assignment of \(\mathcal{F}\), we trace a path down from the root node of the tree, going to a true branch or a false branch at each inner node whose labeled feature is \(f\) depending on \(v(f)\) which is the value of its feature under assignment \(v\). The output of the tree on \(v\) is the label of the leaf reached by this process.

There is an one-to-one mapping between the DT, which is a Boolean combination of features in \(\mathcal{F}(\mathcal{P}, X)\), and the hypothesis space \(\mathcal{H}(\mathcal{P}, X)\) realized by adding or dropping universal quantified variables \(X\) from the prenex.

For example, consider the positive and negative feature vectors over the feature set

\[
\{order(l, u, v), order(l, v, u), member(l, u), member(l, v)\}
\]

shown in Table 4. One DT is shown in the left-hand side of Figure 4. Here, we first pick the feature \(member(l, u)\) as a root to separate the samples \(\{p2, p3, p4, p5, n1, n4\}\) and \(\{p1, n2, n3\}\). Neither of these sets are purely positive or purely negative, which means they still need to be refined. This DT uses four inner nodes and can be mapped to a formula in

<table>
<thead>
<tr>
<th>positive</th>
<th>(ord(l, u, v))</th>
<th>(ord(l, v, u))</th>
<th>(mem(l, u))</th>
<th>(mem(l, v))</th>
</tr>
</thead>
<tbody>
<tr>
<td>(p1)</td>
<td>false</td>
<td>false</td>
<td>true</td>
<td>true</td>
</tr>
<tr>
<td>(p2)</td>
<td>true</td>
<td>true</td>
<td>true</td>
<td>true</td>
</tr>
<tr>
<td>(p3)</td>
<td>true</td>
<td>false</td>
<td>true</td>
<td>true</td>
</tr>
<tr>
<td>(p4)</td>
<td>false</td>
<td>false</td>
<td>true</td>
<td>true</td>
</tr>
<tr>
<td>(p5)</td>
<td>false</td>
<td>false</td>
<td>true</td>
<td>false</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>negative</th>
<th>(ord(l, u, v))</th>
<th>(ord(l, v, u))</th>
<th>(mem(l, u))</th>
<th>(mem(l, v))</th>
</tr>
</thead>
<tbody>
<tr>
<td>(n1)</td>
<td>true</td>
<td>false</td>
<td>true</td>
<td>false</td>
</tr>
<tr>
<td>(n2)</td>
<td>true</td>
<td>false</td>
<td>true</td>
<td>false</td>
</tr>
<tr>
<td>(n3)</td>
<td>true</td>
<td>true</td>
<td>false</td>
<td>false</td>
</tr>
<tr>
<td>(n4)</td>
<td>true</td>
<td>true</td>
<td>false</td>
<td>false</td>
</tr>
</tbody>
</table>

Table 4. Decision-tree learning samples.
our hypothesis space by adding the prenex:

\[ \forall l \ u \ v, (\text{member}(l, u) \implies (\text{order}(l, u, v) \implies \text{member}(l, v))) \land \\
(\neg \text{order}(l, u, v) \implies \tau)) \land \\
(\neg \text{member}(l, u) \implies \neg \text{order}(l, u, v)) \]

A simpler DT is shown in the right hand side of Figure 4; this DT uses fewer inner nodes and thus yields a correspondingly shorter formula:

\[ \forall l \ u \ v, (\text{member}(l, v) \implies \tau) \land \\
(\neg \text{member}(l, v) \implies \text{order}(l, u, v)) \]

This lemma has the same meaning as the one shown in the left-hand side in Figure 4. However, it uses different features for its nodes.

Thus, central to our learning task is the question of quantifying a notion of the goodness of feature. Informally, the information gain of a feature evaluates how homogeneous the feature vectors are after choosing the feature as a separator. We prefer high information gain features that lead to a split that causes two partitions, one with more positive feature vectors and the other with more negative. However, there are no guarantees on learning a high information gain split at every internal classification task.

The Shannon Entropy $\epsilon$ of a set of feature vectors $FVec = FVec^+ \cup FVec^-$ is:

\[ \epsilon(FVec) = -\frac{|FVec^+|}{|FVec|} \log_2 \frac{|FVec^+|}{|FVec|} - \frac{|FVec^-|}{|FVec|} \log_2 \frac{|FVec^-|}{|FVec|} \]

which yields a value that rates the ratio of positive and negative feature vectors. A small entropy value indicates that $FVec$ contains significantly more of one than the other. Formally, the information gain $IG$ of a feature $f \in F$ on $FVec$ is:

\[ IG(FVec, f) = \epsilon(FVec) - \left( \frac{|FVec_f| \epsilon(FVec_f)}{|FVec|} + \frac{|FVec_{\neg f}| \epsilon(FVec_{\neg f})}{|FVec|} \right) \]

where $FVec_f$ and $FVec_{\neg f}$ are feature vectors in $FVec$ that assign $f$ to true and false.

For example, the feature $\text{member}(l, v)$ splits the samples in Table 4 to $\{p_2, p_3, p_4\}$ and $\{p_1, p_5, n_1, n_2, n_3, n_4\}$. This partition is preferable to $\text{member}(l, u)$, which splits its samples into $\{p_2, p_3, p_4, p_5, n_1, n_4\}$ and $\{p_1, n_2, n_3\}$, because $\text{member}(l, v)$ separates more positive samples from negative ones. The decision tree algorithm will choose $\text{member}(l, v)$ as a root and will generate the DT on the right-hand side of Figure 4 because $\text{member}(l, v)$ has higher information gain.

### 6 Evaluation

We evaluate Elrond along four dimensions:

**Q1:** Is Elrond able to learn candidate lemmas sufficient to verify a range of properties and client programs?

**Q2:** Is Elrond able to verify client programs in a reasonable amount of time?

**Q3:** How does the shape postcondition impact the learned lemmas?

**Q4:** Are the candidate lemmas generated by Elrond reasonable?

All reported data was collected on a Linux server machine with an Intel(R) Core(TM) i7-8700 CPU @ 3.20GHz and 64GB of RAM. Note that Elrond is a single-threaded application because OCaml does not naively take advantage of multiple CPU cores.

To answer (Q1), we used Elrond to verify a corpus of benchmarks which include abstract datatype implementations from Okasaki [16], the OCaml standard library [13], and examples from Verified Functional Algorithms (VFA) [2] and Software Foundations [20]. The functions in our benchmarks manipulate a diverse set of algebraic data types, including search trees, tries, heaps, lists, stacks, queues, and sets. Each
type is equipped with its own set of method predicates, including ordering, membership, uniqueness (no duplicate elements), and position (e.g., the first and last element in the list, or the root and leaf of the trees). Each benchmark uses between 3 – 8 predicates, and the client programs call between 2 – 5 library functions. There are several verification tasks in each abstract data type, leading to different lemmas necessary to complete verification. Specifications for library methods were automatically inferred using DOrder [25], a specification inference tool for OCaml. We note that none of these benchmarks could be directly verified without the additional lemmas provided by Elrond, and that all of the candidate lemmas discovered were verified to be correct.

As an example, the rbset-2 benchmark attempts to verify that a `balance` function preserves the self-balancing property of a red-black tree [17] that implements a set. This function calls 2 library methods, `make_tree` and `is_empty`. As `balance` only changes the structure of a red-black tree, and not its content, the benchmark’s postcondition specifies that (1) every element of the input set is also a member of the output set, and (2) the input set has no duplicate elements, neither does the output set. Elrond discovered the following candidate lemma L, which enabled Z3 to verify that `balance` indeed ensures this postcondition:

\[
\forall u, v, (hd(t, u) \land hd(t, v) \implies u = v) \land \quad (c_1)
\]

\[
\land (once(t, u) \implies \neg(\text{left}(t, u) \lor \text{right}(t, u) \lor \text{left}(t, v) \land \text{right}(t, v)) \land \text{member}(t, u)) \land \quad (c_2)
\]

\[
\land (\text{left}(t, u) \lor \text{right}(t, u) \lor \text{left}(t, v) \land \neg\text{hd}(t, u)) \land \quad (c_3)
\]

\[
\land (\text{left}(t, u) \lor \text{right}(t, u) \lor \text{left}(t, v) \land \text{member}(t, u) \land \text{member}(t, v)) \land \quad (c_4)
\]

\[
\land (\text{left}(t, u) \lor \text{right}(t, u) \lor \text{left}(t, v) \land \text{member}(t, u) \land \text{member}(t, v) \land u \neq v) \land \quad (c_5)
\]

In addition to `hd` (which identifies the root of a tree) and `member`, `L` includes four additional method predicates. The predicates `left`, `right` and `par` describe three ordering relations between pairs of elements in a tree: `left(t, u)` and `right(t, u)` capture that `u` is an ancestor of `v` and `v` is in the left and right sub-tree of `u`, respectively. `par(t, u)` holds when `u` and `v` have the same ancestor. `once(t, u)` is true if `u` appears in the tree `t` exactly once. These method predicates reflect natural properties for describing behaviors of a set implemented using red-black trees.

The candidate lemma is a conjunction of five clauses, each of which captures an interesting property of these six method predicates: `c_1` states that the root of the tree is unique. `c_2` says that if `u` appears once in tree `t`, `u` is a member of `t` and cannot be an ancestor of itself or appear in different subtrees. Alternatively, if `u` appears more than once in `t`, then `c_3` requires `u` to be its own ancestor or else reside in a sub-tree of a tree for which `u` is not the root. `c_4` states that if `u` and `v` are ordered by `left`, `right` or `par`, then both `u` and `v` are elements of the tree. Finally `c_5` encodes that distinct elements of the tree are always related by at least one ordering predicate.

To address Q2, we also ran an experiment to quantify how the performance of Elrond was impacted by the number of quantified variables and size of predicate sets. In this experiment (see Table 6), we fixed both parameters instead of iteratively increasing them, choosing values in excess of what was needed to find a candidate lemma for the batched-1 benchmark. The fourth row shows that when the feature set grows to 40, which means there are $2^{40} \approx 10^{12}$ feature vectors and over $2^{40}$ different classifiers possible in the hypothesis space, a size too large to naively enumerate, the algorithm only takes 10 minutes to find a candidate lemma. By way of comparison, QuickSpec [22], a theorem discovery algorithm only takes 10 minutes to find a candidate lemma for the others, even when allowed to run for over 100 minutes.

To answer Q3, we used Elrond to verify a `partition` function for a splay heap against two different specifications (splayhp-1 and splayhp-2). This function divides a splay tree `t` into two sub-trees `small` and `big` according to a pivot value such that `small` contains all elements of `t` less than or equal to pivot and `big` contains all those greater than pivot. If the `small` and `big` should only contain members of `t` (splayhp-1), the candidate lemma learned is:

\[
\forall t, u, v, (hd(t, u) \implies \text{member}(t, u)) \land \\
\neg\text{member}(t, u) \implies \\
\neg(\text{left}(t, u, v) \land \text{right}(t, u, v) \land \text{par}(t, u, v)) 
\]

The two clauses in this lemma assert that the root of a tree is also a member and that when `u` is not the member of a tree, the ordering relations (left, right, par) will not hold between `u` and any `v`. Alternatively, if we change the assertion in the postcondition to require that all members of `small` must also be less than or equal to pivot and that the members of `big` are similarly greater than pivot (splayhp-2), Elrond learns the following candidate lemma:

\[
\forall t, u, v, ((\text{left}(t, u, v) \lor \text{right}(t, u, v)) \implies \\
\neg\text{member}(t, u) \land \text{member}(t, v)) \land \\
\neg(\text{hd}(t, u) \land \text{member}(t, u) \land \text{member}(t, v) \land u \neq v) \implies \\
(\text{left}(t, u, v) \lor \text{right}(t, u, v) \lor \text{par}(t, u, v)) 
\]

that incorporates an ordering constraint deemed to be necessary to verify the assertion. Note that its first conjunction does not establish any relation between `par` and `member` unlike the second conjunct of the candidate lemma for splayhp-1. By conjuncting these two post-conditions (splayhp-3), Elrond correctly yields the conjunction of the two candidate lemmas above.
Finally, to judge whether Elrond was able to learn reasonable candidate lemmas (Q4), we used Dafny [12] to verify all the candidate lemmas generated for our benchmarks against implementations of our method predicates. Of these, Dafny was able to verify two-thirds (15/23) directly, without any additional user intervention. For the remaining third, simple inductive invariants needed to be supplied to allow Dafny to properly unroll recursive procedures.

7 Related Work

Data-Driven Approaches. There have been several recent data-driven approaches that use learning to infer specifications of library functions. Zhu et al. [25] automatically infer specifications which use a fixed set of features (analogous to our method predicates) to identify relationships between the input and outputs of a function. Padhi et al. [19] use program synthesis to automatically learn features on demand when inferring preconditions for data-structure manipulating library functions. This approach is further extended by Mültnner et al. [15] to synthesize [18] representation invariants that are sufficient to verify specifications of the operations of abstract data types. In contrast, rather than inferring specifications of libraries in isolation, this paper considers the complementary problem of leveraging and extending those specifications to verify library clients.

Automated Verification. Encoding verification conditions in a logic for which efficient solvers exist (e.g. SMT) is ubiquitous in the automatic program verification community. In this setting, the standard approach to reasoning about clients of user-defined functions is to rely on some manually written axiomatization of those functions, paying particular care in order to ensure that the underlying solver will terminate [10, 11]. In the case that a specification is incomplete, e.g. when defining new functions, manual intervention is required to extend the axiomatization. Our approach can be thought of as filling in the missing parts of specifications by using the latent semantics of method predicates. More recently, Vazou et al. [23] introduced refinement reflection in order to enable SMT-based reasoning about arbitrary user-defined functions. There, the semantics of a function is embedded directly into the logic as a set of equations, and users can manually construct equational proofs about their behavior using a library of proof combinators. The authors introduce a proof-search algorithm to help automate the construction of these proofs. While this algorithm is complete when a proof exists for a bounded unfolding of function definitions, users are still required to provide provide instantiations of lemmas and induction hypotheses to completely automate program verification. Our approach uses data-driven methods and counterexample-guided search to generate lemmas without the need to reflect the implementation into the solver’s underlying logic.

Abductive Inference. As noted in Sec. 2, our logical formulation of specification inference is an instance of an abductive inference problem. This observation has been previously exploited to develop inference algorithms for loop invariants [7] and specifications of functions in a client program [1]. For a given program, both algorithms rely on an abduction procedure to iteratively strengthen the loop invariants (resp. function specifications) until they are strong enough to prove a user-provided postcondition. While completely automated, these approaches critically rely on an abduction procedure for the underlying specification logic, in particular the first-order theory of linear integer arithmetic in their experiments. Unfortunately, to the best of our knowledge, no such abduction procedure exists for the theory of equalities with uninterpreted function symbols that is commonly used to specify recursive functions over algebraic datatypes. Our data-driven approach provides an alternative solution that uses learning-enabled classifiers to discover candidate lemmas that effectively serve as verifiable abducibles.

8 Conclusions

This paper presents a novel data-driven approach to lemma discovery, the search for relational properties about a library’s methods necessary to verify a client program. We demonstrate that our technique, manifested in a tool called Elrond, is highly effective in identifying sophisticated lemmas to enable verification of challenging functional data structure programs.

References


Data-driven Lemma Discovery

PL’21, January 01–03, 2020, New York, NY, USA

(2011), 53–64.


