Data-Driven Abductive Inference of Library Specifications

ANONYMOUS AUTHOR(S)

Programmers often leverage data structure libraries that provide useful and reusable abstractions. Modular verification of programs that make use of these libraries naturally rely on specifications that capture important properties about how the library expects these data structures to be accessed and manipulated. However, these specifications are often missing or incomplete, making it hard for clients to be confident they are using the library safely. When library source code is also unavailable, as is often the case, the challenge to infer meaningful specifications is further exacerbated. In this paper, we present a novel data-driven abductive inference mechanism that infers specifications for library methods sufficient to enable verification of the library’s clients. Our technique combines a data-driven learning-based framework to postulate candidate specifications, and uses SMT-provided counterexamples to refine these candidates, taking special care to prevent generating specifications that overfit to sampled tests. The resulting specifications form a minimal set of requirements on the behavior of library implementations that ensures safety of a particular client program. Our solution thus provides a new multi-abduction procedure for precise specification inference of data structure libraries guided by client-side verification tasks. Experimental results on a wide range of realistic OCaml data structure programs demonstrate the effectiveness of the approach.

Additional Key Words and Phrases: Automated Verification, Data-Driven Specification Inference, Data Structures, Decision Tree Learning, Counterexample Guided Refinement

1 INTRODUCTION

Using a specification of a library’s methods in the verification of its clients is a hallmark of modular reasoning. Because these specifications act as a precise interface between the client and the library, it allows both to be independently verified, without requiring access to their respective implementations. This modularity is particularly beneficial when the library function is complex or its source code is unavailable. All too often though, such specifications are either missing or incomplete, preventing clients from being verified without making (oftentimes unwarranted) assumptions about the behavior of the method definitions. This problem is further exacerbated when libraries expose rich datatype functionality, which often lead to specifications that rely on inductive invariants [Dillig et al. 2013; Garg et al. 2016; Itzhaky et al. 2014; Miltner et al. 2020] and complex structural relations. One solution to this problem is to automatically infer missing specifications. Unfortunately, while significant progress has been made in specification inference over the past several years [Albarghouthi et al. 2016; Miltner et al. 2020; Padhi et al. 2016; Zhu et al. 2016], existing techniques have not considered inference in the context of client programs that make use of data structure libraries with unavailable implementations, whose inferred specifications must be consistent with their usage by the client.

To highlight the challenge, consider the following simple program, which concatenates two stacks together using four operations provided by a Stack library: push, top, is_empty and tail.

To ensure the correctness of this client function, its author may wish to verify that (a) the top element of the output stack is always the top element of one of the input stacks; and, (b) every element of the output stack is also an element of one of the input stacks and vice-versa. In order to express this behavior in a form amenable for automatic verification, we need some mechanism to
encode the semantics of stacks in a decidable logic. To do so, we rely on a pair of method predicates, "head of stack", \(hd\), and "member of stack", \(mem\), to write our postcondition:

\[
\forall u, (hd(v, u) \implies hd(s_1, u) \lor hd(s_2, u)) \land (mem(v, u) \iff mem(s_1, u) \lor mem(s_2, u))
\]

We assume these method predicates are associated with (possibly blackbox) implementations that we can use to check the specifications written using them. Here, the variable \(v\) is used to represent the output stack of \(\text{concat}\). This assertion claims that the head of the output stack must be the head of either \(s_1\) or \(s_2\) and that any element found in the output must be a member of either \(s_1\) or \(s_2\).

By treating these predicates as uninterpreted function symbols (as well as push, top, is_empty, and tail), it is straightforward to generate a verification condition using, e.g., weakest precondition inference, which can be handed off to an off-the-shelf SMT solver like Z3. However, the counter-examples returned by the theorem prover may be spurious, generated by incorrect assumptions about library method behavior in the absence of any constraints other than client-generated VCs. For example, the prover might assume the predicate \(\neg hd(s_1, v_{top})\); here, \(v_{top}\) represents the output of \(\text{top}\) when applied to \(s_1\). This predicate asserts that the output of \(\text{top}\) is not the head of \(s_1\), a claim inconsistent with the expected usage of the method in the client program. Using this assumption, Z3 may return the following counterexample: \(\exists s \; u. hd(s, u) \land \neg mem(s, u)\) This counterexample, which is obviously incongruous with the intended semantics of \(hd\) and \(mem\), was generated because the intuitive relationship between \(hd\) and \(mem\) was lost as a consequence of embedding these predicates as uninterpreted functions when translating the specifications to an SMT query.

To overcome this problem, we need stronger specifications for the library methods, defined in terms of these predicates, that are sufficient to imply the desired postcondition, i.e., specifications which rule out spurious unsafe executions such as the counter-example given above. In the (quite likely) scenario that such specifications are not provided, a reasonable fallback is to infer some specification for these functions that are strong enough to ensure the safety of the client. Usually, however, specification inference takes place using a closed-world assumption in which specifications of library methods are discovered in isolation, independent of the client context in which they are being used. Such assumptions are not applicable here since (a) we do not have access to the library’s method implementations and (b) the nature of the specifications we need to infer are impacted by the verification demands of the client. Some form of data-driven inference [Miltner et al. 2020; Padhi et al. 2016; Zhu et al. 2016] is therefore required, tailored to the client context in which the library methods are used, that can be used to postulate candidate specifications for library methods based on observing their input-output behavior. Unfortunately, completely black-box data-driven approaches are susceptible to overfitting on the set of observations used to train them—ruling out reasonable and safe behaviors of the underlying library functions.

To address the problem of overfitting, we might additionally consider attacking this problem from a purely logical standpoint, treating specification inference as an instance of a multi-abductive inference [Albarghouthis et al. 2016] problem that tries to find formulae \(R_{\text{push}}, R_{\text{top}}, R_{\text{tail}}\) and \(R_{\text{is\_empty}}\) such that \(\land R_i \not\models \bot\) yet is still sufficient to prove the desired verification condition. While such problems have been previously solved over linear integer arithmetic constraints [Albarghouthis et al. 2016] using quantifier elimination, these prior techniques cannot be directly applied to formulae with uninterpreted function symbols like the method predicates (e.g., \(hd\) and \(mem\)) used to encode library method specifications in our setting.

In this work, we combine aspects of these data-driven and abductive approaches that addresses the limitations each approach has when considered independently. Our technique uses SMT-provided counterexamples to generate infeasible interpretations of these predicates (similar to other abductive inference methods) while using concrete test data to generate feasible interpretations (similar to data-driven inference techniques). This combination yields a novel CEGIS-style inference.

J. ACM, Vol. 1, No. 1, Article . Publication date: July 2021.
methodology that allows us to postulate specifications built from method predicates sufficient to prove the postcondition in a purely blackbox setting. Because there may be many such specifications, we endeavor to choose one that is both consistent with the input-output behavior of the library and at least as weak as every other consistent one, to minimize overfitting specifications to test data. Our algorithm applies another data-driven weakening procedure to find such a specification.

To demonstrate the effectiveness of our approach, we have implemented a fully automated abductive specification inference pipeline in OCaml called Elrond (see Figure 1). This pipeline takes as input (a) an OCaml client program that may call blackbox library code defined over algebraic datatypes like lists, trees, heaps, etc.; (b) verification conditions derived from this program; and, (c) a set of method predicates (e.g., \texttt{hd} or \texttt{mem}), along with their (possibly blackbox) implementations, that are used to synthesize library method specifications. It combines tests and counterexample-guided refinement techniques to either generate a set of maximally-weak and consistent specifications for the library methods used by the program sufficient to verify the postcondition, or a concrete counterexample that demonstrates a violation of the postcondition. Our notion of "maximally-weak" is bounded by the "shape" of specifications (e.g., the number of quantified variables, the set of method predicates, etc.) and a time bound. Our results over a range of sophisticated data-structure manipulating programs, including those drawn from e.g., Okasaki [1999], show that Elrond is able to discover maximally-weak specifications (as determined by an oracle executing without any time constraints) for the vast majority of applications in our benchmark suite within one hour.

Our key contribution is thus a new abductive inference framework that is a fusion of automated data-driven methods and counterexample-guided refinement techniques, tailored to specification inference of libraries that make use of rich algebraic datatypes. Specifically, we:

1. Frame client-side verification as a multi-abduction inference problem that searches for library method specifications that are both consistent with the method’s implementation and sufficient to verify client assertions.
2. Devise a novel specification weakening procedure that yields the weakest specification among the collection of all safe and consistent ones, with respect to a given set of quantified variables and method predicates.
3. Evaluate our approach 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. In the next section, we present a detailed example motivating our ideas and informally presenting our approach. A formal characterization of the problem is given in Section 3. Section 4 defines how a data-driven learning strategy can be used to perform inference. A detailed presentation of the algorithm used to manifest these ideas in a practical implementation is given in Section 5. Details of our implementation and evaluation results are explained in Section 6. Related work and conclusions are given in Section 7 and Section 8.
2 MOTIVATION AND OVERVIEW

We begin by detailing how our algorithm works on the concat example from the previous section. Our goal is to find specifications for the Stack library functions that are sufficient to verify the desired postcondition while being permissive enough to admit most reasonable implementations.

The specification of a black-box library function $f$ is represented in our assertion logic using an uninterpreted predicate, $R_f$ that relates the parameters of $f$ to the value it returns in terms of constraints imposed on predicates like $hd$ and $mem$. We denote return values as $v_f$. Given the postcondition from the previous section, Elrond generates the following verification conditions for concat:

\[
\forall s_1, s_2, v, v_{top}, v_{tail}, v_{concat}, v_{is\_empty}:
\]

\[
((R_{is\_empty}(s_1, v_{is\_empty}) \land v_{is\_empty} = \top \land v = s_2) \lor
(R_{is\_empty}(s_1, v_{is\_empty}) \land v_{is\_empty} = \bot \land R_{top}(s_1, v_{top}) \land R_{tail}(s_1, v_{tail})) \land
(\forall u, (hd(v_{concat}, u) \Longrightarrow hd(v_{tail}, u) \lor hd(s_2, u)) \land
(mem(v_{concat}, u) \iff mem(v_{tail}, u) \lor mem(s_2, u))) \land
(R_{push}(v_{top}, v_{concat}, v) \land R_{push}(v_{top}, v_{concat}, v)) \Rightarrow
(\forall u, (hd(v, u) \iff hd(s_1, u) \lor hd(s_2, u)) \land (mem(v, u) \iff mem(s_1, u) \lor mem(s_2, u)))
\]

The conclusion of this implication encodes our desired postcondition, while its premise represents the possible final states, $v$, of concat $(s_1, s_2)$. The clauses of the disjunction in the premise encode the branches of the procedure’s conditional. The first clause, representing the true branch, states that if $s_1$ is empty, then the stack returned by concat is $s_2$. The second clause, capturing the behavior of the false branch, includes five sub-conjuncts, four of which encode the results of the various library functions (push, top, is_empty and tail) invoked by concat, with the remaining clause capturing the result of the recursive call to concat invoked as part of the call to push.

The non-trivial false branch in $VC_{concat}$ can be simplified thus:

\[
\forall s_1, s_2, v, v_{top}, v_{tail}, v_{concat}, v_{is\_empty}:
\]

\[
(R_{is\_empty}(s_1, v_{is\_empty}) \land v_{is\_empty} = \bot \land R_{top}(s_1, v_{top}) \land R_{tail}(s_1, v_{tail}) \land R_{push}(v_{top}, v_{concat}, v)) \Rightarrow
(\forall u, (hd(v_{concat}, u) \Longrightarrow hd(v_{tail}, u) \lor hd(s_2, u)) \land
(mem(v_{concat}, u) \iff mem(v_{tail}, u) \lor mem(s_2, u))) \Rightarrow
(\forall u, (hd(v, u) \Longrightarrow hd(s_1, u) \lor hd(s_2, u)) \land (mem(v, u) \iff mem(s_1, u) \lor mem(s_2, u)))
\]

The remainder of this section refers to the premise and conclusion of this implication as $\Sigma_{concat}$ and $\Phi_{concat}$, respectively. Our goal is to find interpretations of the predicates $R_{push}$, $R_{top}$, $R_{is\_empty}$ and $R_{tail}$ that make this formula valid, i.e. to infer specifications of the Stack library functions that are sufficiently strong to ensure that concat is safe.

2.1 Specification Inference

Not every combination of function specifications that ensures the safety of concat is reasonable. At one extreme, interpreting every predicate as $\bot$ ensures the safety of the client, but we will never be able to find a sensible implementation of these specifications that still satisfies the expected behavior of a stack. Instead, our goal is to find interpretations that are consistent with some realizable implementation, while also being permissive enough to cover a range of possible implementations.

To further illustrate this point, consider the range of possible specifications for each single function $f$, as illustrated in Figure 2. We observe that such specifications can be thought of as a

---

1Simplification details are provided in the supplemental material.

J. ACM, Vol. 1, No. 1, Article . Publication date: July 2021.
separator that identifies executions of $R_f$ that a) are consistent with their intended semantics and b) preserve the safety of the client program $\Phi$. The challenge we face in building such a separator is that we need to construct a mapping ($\Delta$), that we refer to as a verification interface, that maps each placeholder ($R_f$) to a corresponding specification ($\phi_f$). But, naively foisting this task onto an SMT solver is problematic since it is not apparent how we might encode the behavior of an inductively-defined datatype like a stack sensibly using only the first-order decidable theories available to the theorem prover. Indeed, the problem we are faced with is an instance of an abductive inference problem that tries to find interpretations for $R_{\text{push}}$, $R_{\text{top}}$, $R_{\text{tail}}$ and $R_{\text{is empty}}$ in terms of predicates $\text{mem}$ and $\text{hd}$ such that $\land_i R_i \not\models \bot$ and which are sufficient to prove the desired verification condition $\Phi$. While such problems have been previously solved over linear integer arithmetic constraints [Albarghouthi et al. 2016] using quantifier elimination methods, adopting a similar approach to our setting is complicated by the fact that the predicates (e.g. $\text{hd}$ and $\text{mem}$) used to characterize the behavior of library methods are treated as uninterpreted functions by the theorem prover. An additional challenge in our setting is that we seek to infer specifications consistent with the library’s implementation, a requirement that is absent in [Albarghouthi et al. 2016].

We overcome these challenges by adopting a data-driven approach to building such a separator, training a classifier on a set of samples, or interpretations of a client program’s arguments, embedded as a formula that captures whether the method predicates hold at each call site for a particular execution of a program. Intuitively, a sample identifies the structure of the datatype arguments of a library function with respect to a set of method predicates. To train our classifier, we label the samples consistent with $\neg (\Sigma \Rightarrow \Phi)$ as “negative”, so that the learned specifications can help the solver rule out samples that are inconsistent with the semantics of the method predicates or which would produce unsafe executions (i.e., interpretations that would violate the postcondition). In addition to ruling out such safety violations, we also wish to bias our learner towards explanations that are consistent with the (unknown) implementation of these library functions with respect to the definitions of the underlying method predicates. To do so, we also generate “positive” samples that satisfy $\Sigma \Rightarrow \Phi$ using a test generation framework a la QuickCheck [Claessen and Hughes 2011]. Such samples correspond to the overlapping area between the blue and green circles in Figure 2.

Our learning algorithm is thus tasked with building a classifier that separates negative and positive samples for each library function. This separator is encoded as a logical formula defined in terms
of method predicates. Notably, our algorithm generalizes this data-driven abduction procedure
for individual functions to the multi-abduction case, ensuring that discovered interpretations are
globally consistent over all library methods.

The set of possible verification interfaces comprises the hypothesis space for our learner; Figure 2
shows four potential verification interfaces in this space. The dashed purple line, labelled $\Delta^{\text{unsafe}}$,
represents an unsafe verification interface that allows a client program to violate the desired
postcondition. The remaining red lines represent the range of safe verification interfaces. The
two dashed red lines represent the verification interfaces that are sufficient to verify $\Phi$, while
still being suboptimal. $\Delta^{\text{safe}}$ is safe but inconsistent with the behaviors admitted by the library
implementation, and is thus overly restrictive; $\Delta^{\text{consistent}}$ is safe and consistent, but not maximal:
this means there exists another safe and consistent verification interface ($\Delta^{\text{max}}$) in the hypothesis
space that is weaker than it.

To avoid inferring verification interfaces like $\Delta^{\text{safe}}$, our algorithm uses a test generation framework
to filter out inconsistent solutions. Relying on tests 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 library implementation. Because such reliance can also
lead to verification interfaces that are overfitted to the sampled points, we introduce a weakening
phase to generalize a candidate verification interface, e.g. $\Delta^{\text{consistent}}$.

Hypothesis space. Our data-driven inference algorithm limits the shape of specifications so
that they are both amenable to automatic verification and are strong enough to verify specified
postconditions. To enable automated verification, these specifications are expressed as prenex
universally-quantified propositional formulae over datatype values and variables representing
arguments to the predicates under consideration. Some possible specifications of the library function
push in our running example include:

\[
\begin{align*}
\text{push}(x, l) &= v : \\
\{ \forall u, \text{mem}(l, u) &\implies \text{mem}(v, u), \; \forall u, \neg\text{mem}(v, u), \; \forall u, \text{mem}(l, u) \land h\text{d}(l, u), \; \\
\forall u, u = x, \; \forall u, u = x &\iff h\text{d}(v, u), \ldots \}
\end{align*}
\]

which contains, among other candidates, the desired specification.

All atomic literals in generated formulae are applications of uninterpreted method predicates
and equalities over quantified variables, parameters, and return value of a function. The literals in
the above formulae are simply applications of $h\text{d}$ and $\text{mem}$ to $l$, $v$ and $u$, and the equality constraint
$u = x$. We automatically discard equalities between terms of different types, e.g. $l = x$. The feature
set for $R_{\text{push}}$, i.e., the set of atomic elements used to construct a specification formulae, used by our
algorithm is thus: $\{h\text{d}(l, u), \text{mem}(l, u), h\text{d}(v', u), \text{mem}(v, u), x = u\}$.

Samples and Feature vectors. Our goal is to find a combination of method predicates in the
hypothesis space that validates $\Sigma \implies \Phi$. Returning to our running example, our algorithm
initially begins by interpreting all function specifications as $\top$. Under this interpretation, verifying
$\Sigma_{\text{concat}} \implies \Phi_{\text{concat}}$ with an off-the-shelf theorem prover produces the following counterexample,
where $a$ is some constant:

\[
\forall l, u, \neg h\text{d}(l, u) \land v_{\top} = a \land (\text{mem}(l, u) \iff ((l = v \lor l = v_{\text{concat}} \lor l = v_{\text{tail}}) \land u = a)) \tag{CEX}
\]

To understand how we rule out this counterexample, we focus our attention on the clauses in
$\Sigma_{\text{concat}}$ involving function calls:

\[
R_{\text{is-empty}}(s_1, v_{\text{is-empty}}) \land v_{\text{is-empty}} = \bot \land R_{\top}(s_1, v_{\top}) \land R_{\text{tail}}(s_1, v_{\text{tail}}) \land R_{\text{push}}(v_{\top}, v_{\text{concat}}, v)
\]

J. ACM, Vol. 1, No. 1, Article . Publication date: July 2021.
The counterexample asserts that the stacks \( v, v_{\text{tail}}, v_{\text{concat}} \) contain exactly one element, \( a \), and the other two stacks, \( s_1 \) and \( s_2 \), are empty. This assertion violates the second conjunct of the postcondition, \( \text{mem}(v, u) \iff \text{mem}(s_1, u) \lor \text{mem}(s_2, u) \).

The counterexample arises because the current specifications of the functions are too permissive, and thus need to be strengthened. There are many ways to do so, however. One solution is to focus on a particular function at a time. For example, we could choose to refine the specification of \( R_{\text{top}} \) so that it guarantees that \( v_{\text{top}} \) is a member of \( s_1 \). Alternatively, we could focus on \( R_{\text{tail}} \), so the members of \( v_{\text{tail}} \) are also contained by \( s_1 \). In general, however, it may be necessary to strengthen multiple specifications at once, so our algorithm tries to refine all specifications simultaneously so that they are globally consistent with one another.

Our algorithm does so iteratively, using the assignments to the arguments of the placeholder in a counterexample to build feature vectors that describe the valuations of method predicates and equalities in the unsafe execution. Table 1 presents the feature vectors extracted from CEX. The first column of this table indicates the particular function calls we need to strengthen. The second column forms feature vectors for a particular instantiation of the quantified variables (e.g., \( u \)) in the current specification of the function. The subsequent columns list applications of predicates, with the rows underneath listing the valuation of these predicates in the offending run. The second row corresponds to the assertion that \( \neg \text{hd}(s_1, a) \land \neg \text{mem}(s_1, a) \land v_{\text{top}} = a \), for example. A strengthening of the specifications that disallows any one of these interpretations will also rule out the corresponding unsafe run of the program. Put another way, each row corresponds to a potential negative feature, and a classier (i.e., specification) for the corresponding placeholder that disallows this feature will disallow the counterexample.

The designation of these features as potentially negative is deliberate, as we only want to disallow features that are inconsistent with the implementation of the library functions. To identify actual negative training data, our algorithm generates random inputs of the client program to get inputs representing positive samples. For our current example, running blackbox implementations of \( \text{hd} \) and \( \text{mem} \) on these samples allows us to also extract positive feature vectors. For example, letting \( s_1 = [a] \) and \( s_2 = [b] \), we get the assignment:

\[
v_{\text{is_empty}} = \bot, v_{\text{top}} = a, v_{\text{tail}} = [], v_{\text{concat}} = [b], v = [a; b]
\]
Table 3 shows the partition of positive and negative feature vectors for $R_{\text{top}}$. Table 2 illustrates corresponding feature vectors of the positive sample above. Consider the second row where $u$ is instantiated with $a$: under this assignment, $hd(s_1, u) \equiv hd([a], a)$ is true, as are $mem(s_1, u) \equiv mem([a], a)$ and $v_{\text{top}} = u \equiv a = a$.

In general, the feature vectors extracted from counterexamples and those extracted from testing can overlap. We can observe this for the vectors of $R_{\text{top}}$ in Table 1 and Table 2: the interpretation $hd(s_1, u) \mapsto false; mem(s_1, u) \mapsto false; v_{\text{top}} = u \mapsto false$; occurs in both tables, for example. Intuitively, we do not want to strengthen the specification of $R_{\text{top}}$ to rule out this scenario, as the positive sample is a witness that this execution is consistent with the implementation of $R_{\text{top}}$. This strategy is similar to that taken by Miltner et al. [2020] to deal with inductiveness counterexamples.

As long as the counterexample set contains at least one feature vector that describes an inconsistent execution, however, we can strengthen the specifications to disallow it. The first row of $R_{\text{top}}$ in Table 1 represents one such infeasible execution. This vector encodes the case when $u$ is the output of $top$ but is not a member or head of the input stack; clearly, no positive sample would support such an interpretation. We use this observation to label as “negative” those feature vectors that are extracted from a counterexample but do not appear in the set drawn from a concrete execution. Table 3 shows the partition of positive and negative feature vectors for $R_{\text{top}}$.

When all feature vectors extracted from a counterexample overlap with the set of existing positive feature vectors, we cannot use these potential negative feature vectors to learn a new specification mapping to reject the counterexample. This situation can happen when the assertion made by the client is actually wrong, or if the size of the number of quantified variables is limited, preventing...
is_empty(s, v) \equiv \forall u, (v \implies \neg mem(s, u)) \land (\neg v \land hd(s, u) \implies mem(s, u))

top(s, v) \equiv \forall u, mem(s, v) \land (v = u \iff hd(s, u))

tail(s, v) \equiv \forall u, (mem(s, u) \implies (mem(v, u) \lor hd(s, u))) \land 

\left( (mem(v, u) \lor hd(v, u)) \implies mem(s, u) \right)

push(x, s, v) \equiv \forall u, (mem(v, u) \land mem(s, u) \implies \neg(x = u \lor hd(v, u))) \land 

\left( (x = u \lor hd(v, u) \lor hd(s, u) \lor mem(s, u)) \implies mem(v, u) \right)

Fig. 3. Specifications of Stack operations. The special variable \( v \) represents the result of the method.

discovery of sufficiently expressive specifications. In this case, we try to convert the counterexample to a concrete-valued counterexample, that is a counterexample that manifests with specific test inputs of the client program: if we succeed, then we can claim the provided assertion is wrong; if we fail, we increase the number of quantified variables (e.g. from \( u \) to \( v \)) to this behavior: e.g. 

\begin{align*}
\forall u, mem(v, u) & \implies mem(s_2, u) \\
\forall l, v_{top} = a & \land (mem(l, u) \iff ((l = v \lor l = v_{concat} \lor l = s_1) \land u = a))
\end{align*}

We now try to discover concrete values of \( s_1 \) and \( s_2 \) consistent with this CEX’, that is, \( s_1 \) only contains \( a \) and \( s_2 \) is empty. When run on one such possible input, \( s_1 \equiv [a] \) and \( s_2 \equiv [] \), concat will return \( v \equiv [a] \), exhibiting a concrete violation of the postcondition.

Classification. Given this labelled set of positive and negative feature vectors, the role of the classifier is to build a separator over the training data. One such classifier (formula) for the data in Table 3 is \( v_{top} = u \implies hd(s_1, u) \). Substituting similarly learned specifications for the other library functions in \( \Sigma \) equips the SMT solver with enough constraints to rule out Cex. Our algorithm uses a counterexample-guided refinement loop to find safe specifications for the library functions, iteratively gathering potential negative feature vectors from counterexamples and positive feature vectors from automated (random) testing until the inferred specification are strong enough to remove all counterexamples. Figure 3 shows one such choice of specifications that enable a SMT solver to verify the verification conditions for concat.

Weakening. While safe, this solution may still be suboptimal, as Figure 2 illustrates. Our reliance on testing to identify and label the “actual” negative feature vectors may result in specifications that are overfitted to the sample space enumerated by the test generator. As an example, the first conjunct of the specification push in Figure 3 states that any member of both the input and output stacks is not the same as the element added to the stack, i.e. push always produces a stack with no duplicates. This specification is too restrictive, as it disallows an implementation where, e.g. 

\begin{align*}
\text{push}([1; 2], 1) & = [1; 1; 2] \text{. However, if our sampler never generates a positive sample corresponding to this behavior: e.g. } x \equiv 1, s \equiv [1; 2] \text{ and } v_{top} \equiv [1; 1; 2], \text{ our inferred specification will disallow this (safe) behavior.}
\end{align*}

Note that there are two potential reasons such a positive sample might be missed: (1) the input space of the program might be too large for a test generator to effectively explore, and (2) the
provided implementation may simply not exhibit this behavior (e.g., it may be the case that the implementation of push that we are trying to verify against does indeed remove duplicates). While exhaustive or more effective enumeration can address the first cause, it cannot remedy the second. To ameliorate both issues, we have implemented an additional weakening phase after the refinement loop. Our algorithm iteratively weakens candidate specifications, focusing on one library function \( f \) at a time. To weaken the specification of push for example, we instantiate the other specifications in \( \Sigma_{\text{concat}} \implies \Phi_{\text{concat}} \) with their inferred specifications from Figure 3:

\[
\begin{align*}
\Delta(R_{\text{push}})(\nu_{\text{top}}, \nu_{\text{concat}}, \nu) & \implies \\
(\Delta(R_{\text{is_empty}})(s_1, \bot) \land \Delta(R_{\text{top}})(s_1, \nu_{\text{top}}) \land \Delta(R_{\text{tail}})(s_1, \nu_{\text{tail}})) & \implies \Phi_{\text{concat}} \\
(\Sigma_{\text{concat}}[\Delta] & \implies \Phi_{\text{concat}})
\end{align*}
\]

We denote the instantiation of a Stack function \( f \) with its specification from Figure 3 as \( \Delta(R_f) \). Our goal is to find a maximal weakening of \( R_{\text{push}} \) that admits a larger set of implementations of push. To do so, we employ our data-driven driven approach to infer a new weakened formula, \( W_{\text{push}} \), from our hypothesis space such that \( W_{\text{push}} \lor \Delta(R_{\text{push}}) \) implies \( \Sigma_{\text{concat}}[\Delta] \implies \Phi_{\text{concat}} \).

Intuitively, inference of \( W_{\text{push}} \) is similar to the specification inference process given earlier, insofar as it also gathers potential negative feature vector from the query:

\[
\neg((W_{\text{push}} \lor \Delta(R_{\text{push}})) \implies (\Sigma_{\text{concat}}[\Delta] \implies \Phi_{\text{concat}}))
\]

On the other hand, inference here differs from our earlier approach in its treatment of positive feature vectors; such vectors must now satisfy:

\[
\neg((\Sigma_{\text{concat}}[\Delta] \implies \Phi_{\text{concat}}) \implies (W_{\text{push}} \lor \Delta(R_{\text{push}})))
\]

Such vectors must be consistent with \( \Sigma_{\text{concat}}[\Delta] \implies \Phi_{\text{concat}} \) but should not be included in \( W_{\text{push}} \lor \Delta(R_{\text{push}}) \). The algorithm terminates when the learned new classifier (i.e. \( W_{\text{push}} \)) make both of the above queries unsat. In this case, \( W_{\text{push}} \lor \Delta(R_{\text{push}}) \) is the expected maximal specification. Notice that this formula always entails \( \Delta(R_{\text{push}}) \), ensuring that an inferred specification never becomes inconsistent through weakening.

\[
\begin{align*}
is_{\text{empty}}(s, \nu) & \equiv \forall u, (\nu \implies \neg \text{mem}(s, u)) \\
top(s, \nu) & \equiv \forall u, \nu = u \iff \text{hd}(s, u) \land \text{hd}(\nu, u) \\
tail(s, \nu) & \equiv \forall u, (\text{mem}(s, u) \implies (\text{mem}(\nu, u) \lor \text{hd}(s, u))) \land \\
& \quad ((\text{mem}(\nu, u) \lor (\text{hd}(\nu, u) \land \text{hd}(s, u))) \implies \text{mem}(s, u)) \\
push(x, s, \nu) & \equiv \forall u, (\text{mem}(\nu, u) \land \neg x = u \implies \text{mem}(s, u) \land \neg \text{hd}(\nu, u)) \land \\
& \quad (x = u \lor \text{mem}(s, u) \implies \text{mem}(\nu, u)) \land (\neg \text{mem}(\nu, u) \land \text{hd}(\nu, u) \implies \text{hd}(s, u))
\end{align*}
\]

Fig. 4. Maximal specifications for Stack operations. The special variable \( \nu \) represents the result of the method.

Our algorithm iteratively weakens each function specification until a fixpoint is reached. Figure 4 shows the weakened specifications for our running example. Compared with Figure 3, push now permits duplicate elements in stacks; moreover, the specification of is_empty is simplified by removing the redundant conjunction \( \neg \nu \land \text{hd}(s, u) \implies \text{mem}(s, u) \), as \( \text{hd}(s, u) \implies \text{mem}(s, u) \) can never be violated by a concrete stack value thanks to the observed semantics of \( \text{hd} \) and \( \text{mem} \).
3 PROBLEM FORMULATION

Having completed a high-level tour of Elrond in action, we now present a precise description of the specification synthesis problem and our data-driven inference procedure. We consider functional programs that use data structure libraries that provide functions to access and construct instances of inductively-defined algebraic datatypes (e.g., list, stacks, trees, heaps, tries, etc.).

In the remainder of the paper, we use \((\Sigma, \Phi)\) to refer to the verification query whose validity we are attempting to establish. The first component of this query, \(\Sigma\), is a conjunction of applications of specification placeholders \((R_f)\) to arguments; these represent the library method calls made by the client program. The second component, \(\Phi\), represents the client program’s pre- and post-conditions, encoded as sentences built from logical connectives \((\land, \lor, \implies)\) over prenex universally-quantified propositional formulae.

**Definition 3.1 (Problem Definition).** A given verification query \((\Sigma, \Phi)\) with unknown library functions \(F\) has the form \(\Sigma \implies \Phi\), where:

\[
\Sigma \equiv ( \bigwedge_{f \in F, i \geq 0} R_f(\bar{x}_i) \land \bigwedge_{x, x' \in \cup \{0, \ldots, n\}, \cup \text{constant}} x = x')
\]

Here, the equality constraints are either between program variables \((x, x')\) or between variables and constants of some base type (e.g., Booleans and integers). Each \(R_f(\bar{x}_i)\) in \(\Sigma\) is a placeholder application; the conjunction of placeholder applications and equality constraints represents a sequence of library method invocations in one control-flow path of the client.

To model the input and output behaviors of the black-box implementations of library functions and method predicates, our formalization relies on a pair of partial functions with the same signature as the implementations. We use partial functions to reflect the fact that we can only observe a subset of the full behaviors of these implementations when searching for specifications.

**Definition 3.2 (Specification Configuration).** Let \(P\) be a predicate set, and \(F\) be the set of methods in a library used by the client. Let \(\Gamma_f\) be a partial function from the domain of \(f \in F\) to its codomain, and \(\Gamma_p\) be a partial function with the same signature as \(p \in P\). Let \(\Gamma_P = \bigcup_{p \in P} \Gamma_p\) and \(\Gamma_F = \bigcup_{f \in F} \Gamma_f\). A specification configuration is a 5-tuple \((\Sigma, \Phi, P, F, \Gamma_P, \Gamma_F)\), where \((\Sigma, \Phi)\) is the verification query extracted from the client.

Given a specification configuration as input, the output of our verification pipeline is a verification interface \((\Delta)\), a logical interpretation of the method predicates that maps each specification placeholder for a function \(f \in F\) to a universally-quantified propositional formula over the parameters and result of \(f\). We impose two requirements on \(\Delta\). The first is safety: an underlying theorem prover (e.g., a SMT solver) must be able to prove \(\Sigma[\Delta] \implies \Phi\), where \(\Sigma[\Delta]\) denotes the formula constructed by replacing all occurrences of specification placeholders with their interpretation in \(\Delta\), and \(\Sigma \implies \Phi\) is the verification query built from a client program:

**Definition 3.3 (Safe Verification Interface).** For a given verification query \((\Sigma, \Phi)\), a verification interface \(\Delta\) is safe when:

1. it makes the VC valid: \(\Sigma[\Delta] \models \Phi\), and
2. is not trivial: \(\Sigma[\Delta] \not\models False\)

Besides safety, we also desire that any proposed mapping \(\Delta\) be consistent with the provided implementations of method predicates and library functions, i.e. that \(\Delta\) must accurately represent their observed behavior. Formally:
**Definition 3.4 (Mapping Consistency).** A verification interface $\Delta$ is consistent with $\Gamma_P$ and $\Gamma_F$ when all specifications in $\Delta$ are consistent with the inputs on which $\Gamma_P$ and $\Gamma_F$ are defined. Formally,

$$\forall f \in F, \Gamma_f \in \Gamma_F, \Delta_f(\vec{a}) = v \Rightarrow \Delta(R_f)(\vec{a}, v)[\Gamma_P]$$

$\Delta(R_f)(\vec{a}, v)$ denotes the instantiation of the formula bound to $R_f$ in $\Delta$ with the input arguments $\vec{a}$ and observed output $v$. The expression $\Delta(R_f)(\vec{a}, v)[\Gamma_P]$ replaces all free occurrences of $p$ in $\Delta(R_f)(\vec{a}, v)$ with $\Gamma_P$ where $\Gamma_P \in \Gamma_P$.

This definition thus relates the observed behavior of a library method on test data, encoded by $\Gamma_P$ and $\Gamma_F$, with its logical characterization provided by $\Delta$. Note that there may be many possible verification interfaces for a given specification configuration. In order to identify the best such interface, we use an ordering based on a natural logical inclusion property:

**Definition 3.5 (Partial order $\succ$).** The verification interface $\Delta'$ is weaker ($\succ$) than $\Delta$ when,

1. The two interfaces contain the same functions: $\text{dom}(\Delta) = \text{dom}(\Delta')$
2. They are not equal: $\exists R_f \in \text{dom}(\Delta), \Delta(R_f) \nsucc \Delta'(R_f)$
3. The specifications in $\Delta'$ are logically weaker that those in $\Delta$: $\forall R_f \in \text{dom}(\Delta), \Delta(R_f) \nsucceq \Delta'(R_f)$

Intuitively, weaker verification interfaces are preferable because they place fewer restrictions on the behavior of the underlying implementation. Given an ordering over verification interfaces, we seek to find the weakest safe and consistent interface, i.e. one that imposes the fewest constraints while still enabling verification of the client program:

**Definition 3.6 (Maximal Safe and Consistent Verification Interface).** For a specification configuration $(\Sigma, \Phi, P, F, \Gamma_P, \Gamma_F)$, $\Delta$ is the maximal and safe and consistent verification interface when:

1. $\Delta$ is safe for the verification query $(\Sigma, \Phi)$.
2. $\Delta$ is consistent with $\Gamma_P$ and $\Gamma_F$.
3. For a given bound on the number of quantified variables $k$ allowed in specifications, there is no safe and consistent mapping $\Delta'$ with $k$ quantified variables, such that $\Delta' \succ \Delta$.

Given this notion of maximality, we refine our expectation for the output of our verification pipeline to be not just any safe and consistent verification interface, but a *maximally* safe and consistent one. Notice that our notion of maximality is parameterized by the number of quantified variables in the interpretation. As this bound increases, we can always find a weaker specification mapping. Thus we frame our definition of maximality to be relative to the number of quantified variables in the specification.

## 4 LEARNING LIBRARY SPECIFICATIONS

As Section 2 outlined, Elrond frames the search for a safe verification interface as a data-driven learning problem. At a high level, the goal of learning is to learn a classifier from a set of labeled data, i.e. a function from unlabeled data to a label. More precisely, our goal is to learn classifiers for each of the library functions in a specification configuration that can correctly identify any input and output behavior that could induce an unsafe execution in the client.

Our first challenge is to find an encoding of program executions that is amenable to a data-driven learning framework. To begin, we need to identify the salient features used by a classifier to make its decisions:

**Definition 4.1 (Feature).** A feature of a function $f$ for a set of variables $\vec{x}$ is a method predicate $p$ applied to elements of $\vec{x}$ or equalities between variables in $\vec{x}$.
A feature is similar to a literal in first-order logic, but does not allow for method predicates as arguments (e.g. $hd(l, mem(l, u))$) or constant arguments (e.g. $hd(l, 3)$).

**Definition 4.2 (Feature Set).** The feature set of a function $f$ with method predicates $P$ and quantified variables $\vec{u}$, denoted as $S \equiv FSet(P, f(\vec{a}) = v, \vec{u})$, is a list of all well-typed features in $P$ for the set of variables $\vec{a}_f \cup \{v_f\} \cup \vec{u}$, which is minimally linearly independent:

$$\forall \eta' \notin S, \exists \vec{\eta} \in S, \eta' \iff \bigwedge \vec{\eta}$$

**Example.** The feature set for the function $\top: \text{list}_a \to a$ from the Stack library ($a$ is some base type) for predicate set $P = \{hd, mem\}$, equality operation $=_{a}$ and quantified variables $\vec{u} = \{u : a\}$ is $FSet(P, \top(l) = v, \vec{u}) = [hd(l, u), mem(l, u), v =_{a} u]$. Note that the features $mem(v, u)$ and $l =_{a} u$ are not included in this set because they are not well-typed. The feature $hd(l, v)$, on the other hand, is omitted because it can be represented by $hd(l, u) \land v =_{a} u$, and is thus not linearly independent with the other features in the set. We use feature vectors to encode the features of observed tests:

**Definition 4.3 (Feature Vector).** A feature vector $fv$ is a vector of Booleans that represents the value of each feature in the feature set for some test.

We also need to define the hypothesis space of possible solutions considered by our learning system. To easily integrate learned classifiers into the underlying theorem prover, we choose to represent such solutions as Boolean combinations over terms consisting of applications of interpreted base relations and uninterpreted functions. In order to preserve decidability, we limit this space to a subset of effectively propositional sentences.

**Definition 4.4 (Specification Hypothesis Space).** The specification hypothesis space for a library function $f$, method predicate set $P$, and quantified variables $\vec{u}$ is the set of formulas in prenex normal form with the quantifier prefix $\forall \vec{u}$, and whose bodies are built from $FSet(P, f(\vec{a}) = v, \vec{u})$, the logical connectives $\{\land, \lor, \neg, \implies\}$, and Boolean constants $\top$ (TRUE) and $\bot$ (FALSE). The hypothesis space of $f$ over $P$ and $\vec{u}$ is denoted $Hyp(P, f(\vec{a}) = v, \vec{u})$.

In order to classify feature vectors, we ascribe them a semantics in logic:

**Definition 4.5 (Unitary classifier).** For a given feature vector $fv$ in library function $f$’s feature set $S \equiv FSet(P, f(\alpha) = v, \vec{u})$, the logical embedding of $fv$ is a formula encoding the assignment to its features:

$$[[fv]] \equiv \forall \vec{u}, \bigwedge_{i=0}^{\vert S \vert} S[i] \iff fv[i]$$

We say that a classifier $\phi$ labels a feature vector as positive when $[[fv]] \implies \phi$, and negative otherwise.

**Example.** Given the classifier $\phi \equiv \forall u, hd(s_1, u) \implies v_{top} = u$ for the top function from Section 2, $fv_{\neg} \equiv \{hd(s_1, u) \implies \text{true}; mem(s_1, u) \implies \text{true}; v_{top} = u \implies \text{false}\}$ is a negative feature vector. The unitary classifier for $fv$ is $[[fv_{\neg}]] \equiv \forall u, hd(s_1, u) \land mem(s_1, u) \land \neg v_{top} = u$. The other two feature vectors in Table 3 are labeled as positive by this classifier.

**Definition 4.6 (Classification).** For a given classifier $\phi$ and feature set $S$, it is straightforward to partition the feature vectors of $S$ into positive ($\phi^+$) and negative ($\phi^-$) sets:

$$\phi^+ \equiv \{fv \in 2^S \mid [[fv]] \implies \phi\}$$

$$\phi^- \equiv \{fv \in 2^S \mid [[fv]] \not\implies \phi\}$$

Notice that these two sets are trivially disjoint: $\phi^+ \cap \phi^- = \emptyset$. 

J. ACM, Vol. 1, No. 1, Article . Publication date: July 2021.
For a particular configuration, we can straightforwardly lift this partitioning to verification interfaces:

\[
\Delta(R_f)(\vec{a}, v)^+ \equiv \{fv \in 2^{\text{FSet}(P, f(\vec{a}) = v, \vec{u})} \mid [fv] \Rightarrow \Delta(R_f)(\vec{a}, v)\}
\]

\[
\Delta(R_f)(\vec{a}, v)^- \equiv \{fv \in 2^{\text{FSet}(P, f(\vec{a}) = v, \vec{u})} \mid [fv] \not\Rightarrow \Delta(R_f)(\vec{a}, v)\}
\]

4.1 Verification Interface Inference

We now confront the challenge of how to generate training data from a specification configuration in a way that guarantees the safety of the learned formulas (classifiers). To do so, we build feature vectors from a set of logical samples:

Definition 4.7 (Sample). A sample \( s \) of a formula \( \phi \) is an instantiation of its quantified variables and a Boolean-valued interpretation for each application of a method predicate to those variables in \( \phi \), which we denote as \( s \models \phi \). The positive and negative samples of a verification query \((\Sigma, \Phi)\) are samples of \( \Phi \) and \( \neg \Phi \), respectively.

Intuitively, the positive samples of a verification query correspond to safe executions of a client program, while negative samples represent potential violations that safe verification interfaces need to prevent. For example, CEx from Section 2 corresponds to the following negative sample of \( \Phi_{\text{concat}} \):

\[
\{s_1 \mapsto 1_0; s_2 \mapsto 1_1; v \mapsto 1_2; v_{\text{top}} \mapsto a; v_{\text{tail}} \mapsto 1_3; v_{\text{concat}} \mapsto 1_4; v_{\text{is empty}} \mapsto \bot; \hspace{1cm} (s^-)
\]

\[
\text{hd}(l, u) \equiv \{\}; \text{mem}(l, u) \equiv \{(1_2, a); (1_3, a); (1_4, a)\}
\]

and the following sample, extracted from a concrete input and client execution result, is positive:

\[
\{s_1 \mapsto 1_0; s_2 \mapsto 1_1; v \mapsto 1_2; v_{\text{top}} \mapsto a; v_{\text{tail}} \mapsto 1_3; v_{\text{concat}} \mapsto 1_4; v_{\text{is empty}} \mapsto \bot; \hspace{1cm} (s^+)
\]

\[
\text{hd}(l, u) \equiv \{(1_0, a); (1_1, b); (1_4, b); (1_2, a)\}; \text{mem}(l, u) \equiv \{(1_0, a); (1_1, b); (1_4, b); (1_2, a); (1_2, b)\}
\]

Although they come from different sources, both samples provide the value of variables (e.g. the value of \( v_{\text{is empty}} \)) and the value of predicate applications (e.g. \( \text{hd}(v_{\text{concat}}, v_{\text{top}}) \)).

Using \([\cdot]\), we can extract a collection of feature vectors under a feature set \( S \) from a sample \( s \):

\[
\chi_S(s) \equiv \{fv \in 2^S \mid s \models [fv]\}
\]

For example, the feature vectors extracted under \( S_{\text{top}} \) from \( s^- \) and \( s^+ \) are shown in the second and third rows of Table 1 and Table 2, resp.

Definition 4.8 (Classifier Consistency). For a verification query \((\Sigma, \Phi)\), we say that a verification interface \( \Delta \) is consistent with a negative sample \( s^- \) if at least one of the specification placeholders in \( \Sigma \) classifies a feature extracted from that sample as negative:

\[
\exists R_f(\vec{a}, v) \in \Sigma, \exists fv \in \chi_{\Sigma}(P, f(\vec{a}) = v, \vec{u}) (s^-), fv \in \Delta(R_f)(\vec{a}, v)^-
\]

Similarly, \( \Delta \) is consistent with a positive sample \( s^+ \) if all specifications in \( \Delta \) positively identify every feature vector extracted from \( s^+ \):

\[
\forall R_f(\vec{a}, v) \in \Sigma, \forall fv \in \chi_{\Sigma}(P, f(\vec{a}) = v, \vec{u}) (s^+), fv \in \Delta(R_f)(\vec{a}, v)^+
\]

Example. The verification interface \( \Delta \) from Figure 3 is consistent with \( (s^-) \), as the specification of the top function labels as negative the following feature vector of \( s^- \), \( fv^- \equiv \{hd(s_1, u) \mapsto true; mem(s_1, u) \mapsto true; v_{\text{top}} = u \mapsto false\} \) to negative. Furthermore, \( \Delta \) is also consistent with all the feature vectors extracted from \( s^+ \).
Theorem 4.9. For a given specification configuration \((\Sigma, \Phi), P, F, \Gamma_P, \Gamma_F\) and verification interface \(\Delta\), \(\Sigma[\Delta] \Rightarrow \Phi\) is valid iff \(\Delta\) is consistent with all negative samples \(s^-\); \(\Delta\) is a consistent interface iff \(\Delta\) is consistent with all positive samples \(s^+\) entailed by \(\Gamma_F\) and \(\Gamma_P\).

4.2 Weakening Inference

Note that Theorem 4.9 guarantees that a verification interface is safe and consistent, but does not ensure that it is maximal. We again frame the search for such a maximal solution as a learning problem.

Definition 4.10 (Sample with respect to library function). For a verification query \((\Sigma, \Phi)\), and safe verification interface \(\Delta\), a sample \(s\) is positive (resp. negative) with respect to library function \(f\) when \(s\) is positive (resp. negative) and consistent with the specifications of all other library functions in the domain of \(\Delta\):

\[
s \models \Sigma[\Delta[R_f \mapsto \top]] \land \Phi
\]

\[
s \models \Sigma[\Delta[R_f \mapsto \top]] \land \neg \Phi
\]

Ideally, the maximal specification of \(f\) would be able to positively classify every such positive sample. This is not possible in general due to the intrinsic granularity of the hypothesis space, and we must limit ourselves to covering some subset of this space instead. The weakening relation between classifiers can be viewed as the difference between the sets of positive and negative feature vectors induced by classifiers:

Definition 4.11 (Weakening feature vector and samples). For a verification query \((\Sigma, \Phi)\) and safe verification interface \(\Delta\), a weakening feature vector \(fv\) distinguishes between a weaker safe specification and \(\Delta(R_f)\):

\[
fv \in \Delta(R_f)^- \land \Sigma[\Delta[R_f \mapsto \top] \lor \Delta(R_f)] \Rightarrow \Phi
\]

A sample \(s\) is a weakening sample when \(s\) is positive with respect to \(f\) and includes some weakening feature vector. Intuitively, such weakening samples can be used to safely generalize \(f\). If we cannot find any weakening sample, then the specification must have converged to a maximal one.

Example. The sample

\[
\{s_1 \mapsto 1_0; s_2 \mapsto 1_1; v \mapsto 1_2; u_{\text{top}} \mapsto a; u_{\text{tail}} \mapsto 1_3; v_{\text{concat}} \mapsto 1_4; v_{\text{is-empty}} \mapsto \bot; \}
\]

\[
\text{hd}(l, u) \equiv \{(1_0, a); (1_1, a); (1_2, a); (1_4, a)); \text{mem}(l, u) \equiv \{(1_0, a); (1_1, a); (1_2, a); (1_4, a))\}
\]

is a positive sample with respect to push since it makes both input stacks and the output stack contain \(a\) and also have the head element \(a\), which entails \(\Phi_{\text{concat}}\). It is also consistent with all the specifications in the verification interface from Figure 3, outside of push. The following feature vector extracted from the sample is not included in \(\Delta(R_{\text{push}})^+\),

\[
\{\text{hd}(v_{\text{concat}}, u) \mapsto \text{true}; \text{mem}(v_{\text{concat}}, u) \mapsto \text{true}; \}
\]

\[
\text{hd}(v, u) \mapsto \text{true}; \text{mem}(v, u) \mapsto \text{true}; v_{\text{top}} = u \mapsto \text{true}; \}
\]

Moreover, \(\Sigma[\Delta[R_{\text{push}} \mapsto [fv] \lor \Delta(R_{\text{push}})]\] \(\Rightarrow \Phi\), thus this is a weakening feature vector for push, and the sample above is a weakening sample for push.

Theorem 4.12. For a given specification configuration \((\Sigma, \Phi), P, F, \Gamma_P, \Gamma_F\) and safe verification interface \(\Delta\), there exists a weaker specification \(\phi\) of library function \(f\) where \(\Delta(R_f) \Rightarrow \phi\) and \(\Delta[R_f \mapsto \phi]\) is safe if and only if there exists a weakening sample \(s\) for \(f\).

\(^2\)Proofs for all theorems are provided in the supplemental material.
5 ALGORITHM

Our specification inference algorithm assumes three key components: a property-based random sampler Sampler, a satisfiability checker SMTCheck, and a classifier learner Learner. The Sampler takes a first-order formula as input and attempts to find a counterexample by random sampling, returning data instances which violate this specification based on these counterexamples. SMTCheck 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 Learner takes two feature vector sets π and ω as inputs where π ∩ ω = ∅, and returns a specification φ that classifies elements of π and ω as positive and negative, respectively. We use a decision tree algorithm [Ruggieri 2002] as the underlying learning framework.

5.1 Multi-Abductive Inference

Algorithm 1:

<table>
<thead>
<tr>
<th>Algorithm 1: Multi-Abductive Inference Algorithm.</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Inputs</strong>: Specification configuration ((Σ, Φ), F, P, ΓF, Γp) and set variables for weakening ( \bar{u}_w )</td>
</tr>
<tr>
<td><strong>Output</strong>: Maximal consistent safe specification Δ or concrete counterexample Cex</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>3</td>
</tr>
<tr>
<td>4</td>
</tr>
<tr>
<td>5</td>
</tr>
</tbody>
</table>

Algorithm 1 is the main algorithm for our abductive inference procedure. It is based on two sub-algorithms, SpecificationInference and Weakening. The algorithm first tries to find an initial solution that is safe and consistent with the specification configuration by calling SpecificationInference on line 1. If it fails, a concrete counter-example is returned. Otherwise, the procedure returns a consistent and safe verification interface Δ, and the algorithm proceeds to finding a maximally-weak verification interface using the Weakening procedure. The algorithm returns the weakened solution as its result.

5.2 Specification Inference

Algorithm 2 presents our specification inference algorithm. The algorithm relies on a subroutine DiscoverMapping to infer specifications for a fixed feature set mapping \( \Psi \) (lines 10 - 29), and a main loop (lines 2 – 9) that iteratively grows the hypothesis space used by DiscoverMapping when the existing hypothesis space is insufficient to find a classifier.

SpecificationInference maintains a set of quantified variables \( \bar{u} \) shared by all library function specifications and initialized as empty on line 1. The algorithm calls DiscoverMapping with a fixed feature set; if the procedure returns an unsolvable negative sample \( \bar{s}^{-} \), the algorithm tries to convert the SMT model to a concrete counter-example using ConcreteCexGen, a subroutine that generates random tests consistent with a model (line 8). If ConcreteCexGen fails (line 9) or DiscoverMapping cannot find a verification interface in the current hypothesis space (line 4), the algorithm increases the hypothesis space by adding a new quantified variable to \( \bar{u} \). Otherwise SpecificationInference returns the verification interface returned (line 5).

DiscoverMapping uses a classifier-based learning method to infer a verification interface that is safe and consistent. It maintains a pair of mappings from placeholder \( R_f \) to "positive" and "negative"...
Algorithm 2: Specification Inference

Inputs: Specification configuration ($\Sigma, \Phi$), $F, P, \Gamma_F, \Gamma_P$
Output: Consistent safe verification interface $\Delta$ or concrete counterexample $Cex$

1. $\bar{u} \leftarrow \emptyset$

2. while true do
   
   3. match DiscoverMapping($\{R_f \mapsto FSet(P, f(\bar{u}_f) = v_f, \bar{u})\}, \{R_f \mapsto \forall \bar{u}, \top\}$):
      
      4. case None do $\bar{u} \leftarrow \bar{u} \cup$ NewVariable();
      
      5. case $\Delta$ do return $\Delta$;
      
      6. case $s^-$ do
         
         7. match ConcreteCexGen($s^-$):
            
            8. case Cex do return Cex;
            
            9. case None do $\bar{u} \leftarrow \bar{u} \cup$ NewVariable();
       
   10. Procedure DiscoverMapping($\Psi, \Delta$)
      
      11. $\Pi = \Omega \leftarrow \{R_f \mapsto \emptyset\}$
      
      12. while true do
          
          13. match SMTCheck($\neg (\Sigma [\Delta] \Rightarrow \Phi)$):
              
              14. case Unsat do
                  
                  15. match SMTCheck($\Sigma [\Delta]$):
                      
                      16. case Unsat do return None;
                      
                      17. case Sat do return $\Delta$;
                 
                 18. case Sat $s^-$ do
                    
                    19. for $f \in F$ do $C(R_f) \leftarrow FVecFromModel(\Psi(R_f), s^-) \setminus \Pi(R_f)$;
                    
                    20. if $C = \emptyset$ then return None;
                    
                    21. $\Omega \leftarrow \Omega \cup C$;
               
               22. $\Delta \leftarrow \text{Learner}(\Pi, \Omega)$;
               
               23. match Sampler($\Sigma, \Phi, \Delta(R_f), \Gamma_F, \Gamma_P$):
                   
                   24. case None do break;
                   
                   25. case Some $s^+$ do
                       
                       26. for $f \in F$ do $\Pi(R_f) \leftarrow FvecFromSampler(\Psi(R_f), \Gamma_P, s^+) \cup \Pi(R_f)$;
                       
                       27. $\Omega \leftarrow \Omega \setminus \Pi$;
                
                28. while true;

feature vector sets, $\Pi$ and $\Omega$. These are initialized to empty on line 11. DiscoverMapping populates these mappings with negative feature vectors from a counter-example of $\Sigma [\Delta] \Rightarrow \Phi$ provided by the SMT solver (lines 13 - 21) and positive feature vectors from the blackbox execution of $F$ and $P$ (lines 22 - 29).

DiscoverMapping implements a refinement loop which first uses the solver to check if the verification query ($\Sigma [\Delta] \Rightarrow \Phi$) is valid (line 13); if so, then the current $\Delta$ is sufficient to prove the verification query. If $\Sigma [\Delta]$ is also satisfiable (line 15), then $\Delta$ is a consistent and safe verification interface (line 17); otherwise, the algorithm returns None, forcing SpecificationInference to increase the size of the hypothesis space (line 16). Given a counterexample from the SMT solver $s^-$ (line 18), the algorithm extracts potential negative feature vectors from $s^-$ for each library function for each placeholder (lines 19 - 21). For each library function, it extracts all feature vectors consistent with
the negative sample \(s^-\) that are not included in the positive feature vector mapping \(\Pi\) (line 19). If all potential negative feature vectors overlap with positive feature vectors (line 20), \textit{DiscoverMapping} cannot infer a verification interface for the current hypothesis space, and thus returns \texttt{None}. Otherwise, the algorithm updates the negative feature vector mapping \(\Omega\) with all gathered potential negative feature vectors (line 21). Notice that the algorithm still guarantees \(\Pi \cap \Omega = \emptyset\) since \(C\) is always filtered by \(\Pi\) (line 19). After gathering negative feature vectors, the algorithm infers a verification interface with respect to \(\Gamma_F\) and \(\Gamma_P\) by gathering positive feature vectors from \textit{Sampler}. The algorithm tests the current candidate verification interface \(\Delta\) against \(\Gamma_F\) and \(\Gamma_P\) to find if there exists a concrete valued assignment in which a positive sample \(s^+\) violates \(\Delta(R_F)\) (line 24). If there is such a positive sample, the algorithm adds the feature vectors of the sample to \(\Pi\) (line 27). It then removes any feature vectors appearing in \(\Pi\) from \(\Omega\) in order to preserve the constraint that \(\Pi \cap \Omega = \emptyset\) (line 28). Sampling ends when \textit{Sampler} cannot find any new positive samples and \textit{DiscoverMapping} continues the refinement looping.

**Theorem 5.1 (Inference Soundness).** Any result returned by Algorithm 2 is either a consistent and safe verification interface or a concrete valued counter-example.

**Theorem 5.2 (Inference Decidability).** \textit{DiscoverMapping} in Algorithm 2 always halts.

### 5.3 Weakening

Algorithm 3 shows the \textit{Weakening} algorithm that is used to generalize an initial solution. \textit{Weakening} takes a verification query, method predicate set, verification interface, library function \(f\), and a list of quantified variables as input, and infers a maximal specification for \(f\). \textit{Weakening} maintains a candidate weakening formula \(W_f\) for \(f\); the start of the function initializes this formula to false (line 1). The algorithm also maintains a two sets of feature vectors: \(\pi\), which holds weakening feature vectors, and \(\omega\) which holds non-weakening feature vectors. Both sets are initially empty (line 1). The body of \textit{WeakeningLoop}, implements a counterexample-guided refinement loop. This loop first uses a SMT solver to check if the current specification is maximal (line 3). If so, the procedure returns \(W_f \lor \Delta(R_F)\) (line 5) if \(W_f\) is not false, and \(\Delta(R_F)\) otherwise. If not, the SMT solver has provided a weakening sample \(s\) (line 6), which is used to update \(\pi\) and \(\omega\) via a call to \textit{Update} (line 7). The \textit{Update} function extracts feature vectors from a sample \(s\), inserting any weakening feature vectors into \(\pi\) and all other vectors into \(\omega\). The updated specification \(W_f \lor \Delta(R_F)\) may violate \(\Phi\), so \textit{WeakeningLoop} invokes \textit{SafetyLoop} to filter any unsafe weakenings (line 8). \textit{SafetyLoop} is similar to \textit{WeakeningLoop}, and progressively refines \(W_f\) until it guarantees safety (lines 10-14). After this filtering step, the loop restarts, continuing to examine additional weakening possibilities.

**Theorem 5.3 (Weakening Soundness).** For given \((\Sigma, \Phi), P, \Delta, f, \bar{u}, \Delta\) is safe. algorithm 3 returns the weakest specification \(W_f \lor \Delta(R_F)\) other than \(\Delta(R_F)\) which makes \(\Delta[R_f \Rightarrow W_f \lor \Delta(R_F)]\) safe if \(W_f\) exists; otherwise return \texttt{None}.

**Theorem 5.4 (Weakening Decidability).** Algorithm 3 always halts.

**Theorem 5.5 (Soundness).** Any result returned by Algorithm 1 is either a maximal consistent safe verification interface or a concrete valued counter-example.

### 6 IMPLEMENTATION AND EVALUATION

We have implemented a verification pipeline based on the above approach, called Elrond, that targets OCaml programs that rely on libraries to manipulate algebraic data types. Elrond consists

---

\footnote{For conciseness, we lift set union(\(\cup\)), set minus (\(\setminus\)) and classification (\textit{Learner}) to the feature vector set mappings. For example, \(\Omega \setminus \Pi\) builds a new feature vector set mappings that maps each \(R_F\) to \(\Omega(R_F) \setminus \Pi(R_F)\).}
of 7267 lines of OCaml and uses Z3 [De Moura and Bjørner 2008] as its backend solver. Elrond’s frontend generates verification queries from client programs via weakest liberal precondition predicate transformers. Elrond does not automatically infer inductive invariants for recursive client functions, and it expects client programs to provide such invariants, in addition to any pre- and post-conditions.

Our experimental evaluation of Elrond addresses five key questions:

Q1: Is Elrond able to find specifications sufficient to verify a range of properties and client programs in a reasonable amount of time?

Q2: Can Elrond identify unsafe client programs?

Q3: Can Elrond efficiently find maximal solutions?

Q4: Is Elrond able to find useful intermediate generalizations of initial specifications?

Q5: Does weakening improve the quality of the inferred specifications?

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.

To answer these questions, we have evaluated Elrond on a corpus\(^5\) of client programs drawn from Okasaki [1999], the OCaml standard library [Leroy et al. 2014], Verified Functional Algorithms [Appel 2018] and Software Foundations [Pierce et al. 2010]. Our benchmarks cover a range

---

\(^5\)The supplemental material contains all benchmarks, post-conditions, and inferred library specifications from our evaluation.
| (|F|, |R|) | |P| | |ui| |cex| |time_e(s) | #Gather / ||φ^*| |time_w(s) | time_d(ms) |
|---|---|---|---|---|---|---|---|---|
| (6, 10) | 2 | 2 | 26 | 1.85 | 366 / 1828 | 47.9 | 123.5 |
| (4, 5) | 2 | 1 | 10 | 0.28 | 235 / 1536 | 9.2 | 18.2 |
| (4, 5) | 3 | 2 | 23 | 0.87 | 2076 / 30054 | 318.9 | 42.7 |
| (6, 10) | 1 | 1 | 8 | 0.22 | 53 / 114 | 1.9 | Max |
| (4, 5) | 2 | 1 | 11 | 0.61 | 29 / 13 | 0.5 | 18.5 |
| (4, 5) | 3 | 2 | 39 | 2.94 | 3220 / 22716 | Limit | 169.3 |
| (4, 5) | 2 | 1 | 4 | 0.08 |
| (3, 8) | 2 | 2 | 71 | 11.21 | 1790 / 26588 | Limit | 895.8 |
| (2, 21) | 1 | 1 | 10 | 0.37 | 172 / 432 | 21.5 | 21.3 |
| (2, 21) | 3 | 2 | 141 | 107.18 | 702 / 177388 | Limit | 328.3 |
| (2, 21) | 3 | 2 | 192 | 152.99 | 512 / 190215 | Limit | 149.7 |
| (3, 8) | 1 | 1 | 10 | 0.37 | 464 / 3308 | 46.0 | Max |
| (4, 10) | 1 | 1 | 4 | 0.15 | 50 / 110 | 1.3 | 21.1 |
| (4, 10) | 2 | 2 | 23 | 1.57 | 640 / 1376 | 1038.0 | 70.6 |
| (4, 10) | 1 | 1 | 8 | 0.14 |
| (4, 10) | 3 | 2 | 18 | 1.69 | 1755 / 31276 | 741.8 | 24.3 |
| (4, 10) | 3 | 2 | 12 | 0.37 |
| (1, 21) | 4 | 2 | 91 | 77.75 | 1369 / 59797035 | Limit | 1268.7 |
| (2, 8) | 1 | 1 | 11 | 0.30 | 181 / 448 | 11.2 | 21.1 |
| (2, 8) | 2 | 1 | 21 | 0.81 | 3327 / 27609 | 1741.9 | 23.2 |
| (2, 8) | 2 | 2 | 63 | 8.28 | 2496 / 28495 | Limit | 314.1 |
| (2, 6) | 3 | 1 | 15 | 0.39 | 3644 / 22526 | 1203.3 | 19.5 |
| (2, 6) | 3 | 1 | 25 | 0.69 | 3091 / 20090 | 968.7 | 19.2 |
| (1, 21) | 1 | 1 | 10 | 0.27 | 228 / 772 | 34.0 | Max |
| (2, 8) | 2 | 1 | 8 | 0.14 |
| (2, 6) | 3 | 3 | 29 | 4.69 |

Table 4. Experimental results. The columns in the table can be divided into three groups. The first group presents the number of distinct library functions (|F|), the number of library function applications (|R|), and the size of the method predicate set (|P|) for each benchmark. The second column describes the number of quantified variables (|ui|), the number of counter-examples generated (|cex|) and the time in seconds (time_e) needed for Elrond to find a consistent and safe verification interface. Times indicate how long it took for counter-examples generation, sampling, feature vectors extraction and labelling and DT learning; they do not include the time taken to generate the initial verification query. Red entries in the time column indicate how long it took to identify safety violations in the five unsafe benchmarks. The last group lists the number of gathered (#Gather) and total positive (|φ^*|) feature vectors in the space of weakenings, the time needed by the weakening phase (time_w), and the time needed for the SMT solver to find a sample allowed by a weakened solution but not the initial one (time_d). Blue entries in the |φ^*| column indicate a lower bound. "Limit" entries in the time_w column indicate that the one hour time bound was reached. "Max" entries in the time_d column indicate that the initial solution was already maximally weak.

of abstract data types manipulating a diverse set of algebraic data types, including queues (bankers queues and batched queues), list-based stacks, heaps (leftist heap and splay heap), streams, sets (trees, red black trees and lists) and tries. The underlying representation of each algebraic data type provides a set of method predicates related to ordering, membership, and uniqueness (no duplicate elements) that can be queried as part of a test. We used Elrond to verify several different properties for each data type, including membership, ordering, distinct elements, and sorting.
Example 6.1. One of our benchmarks uses the following specification for an insert program that inserts an element $x$ into an unbalanced set $s$ using a binary tree for its underlying representation:

$$(\forall u, (\text{root}(s, u) \implies (u < x \implies \text{root}(\text{vinsert}, u)) \land (u \geq x \implies \text{root}(\text{vinsert}, x))))$$

$$(\land (\text{mem}(\text{vinsert}, u) \iff \text{mem}(s, u) \lor x = u))$$

Given this specification, Elrond infers the following specification for the $\text{maket}(x, l, r) = v$ function used by add that constructs a new tree from $x$ and the left and right subtrees $l$ and $r$:

$$\forall u, (\text{mem}(v, u) \iff \text{mem}(1, u) \lor \text{mem}(r, u) \lor (x = u)) \land (\text{root}(v, u) \iff x = u)$$

$$\land (\text{root}(1, u) \iff \text{mem}(1, u)) \land (\text{root}(r, u) \iff \text{mem}(r, u))$$

The first line of this specification captures the key semantic properties of the tree, while the second line encodes a key relationship between the $\text{mem}$ and $\text{hd}$ method predicates needed by the solver to verify the specification.

The detailed results of our experiments are shown in Table 4. The first group of columns in Table 4 describes the salient features of our benchmarks. Each client specification uses between 1 and 4 member predicates, and the client programs make between 5 and 21 calls to library functions. In order to evaluate Elrond’s ability to identify faulty specifications, our experiments also included five unsafe client programs. Elrond can infer specifications for all the benchmarks with valid assertions, and returns concrete counter-examples for clients with unsafe post-conditions.

The second group of columns in Table 4 presents our evaluation of Elrond’s ability to discover an initial safe and consistent verification interface (Q1 and Q2). These columns show that Elrond is relatively efficient at finding safe specifications, with none of the benchmarks taking longer than three minutes to learn an initial solution (Q1). As expected, more complicated benchmarks (i.e. those with more function calls or member predicates) required more SMT-provided counter-examples and took longer to complete, as did benchmarks requiring a larger hypothesis space (i.e. specifications with more quantified variables). Notably, Elrond was able to quickly generate concrete counter-examples witnessing safety violations (Q2) in the five benchmarks with invalid postconditions (these benchmarks have red entries in the time column).

The final group of columns in Table 4 addresses the questions dealing with Elrond’s weakening phase (Q3 - Q5). Unsurprisingly, weakening requires more time than the initial inference phase: while the latter needs to identify a single solution, the former needs to account for all possible solutions in order to select the best one. When evaluating this phase, we choose to use a one hour time bound for each experiment; if this time bound was reached, we had Elrond return the current weakened solution. Three-fourths (16/22) of our safe benchmarks were able to find maximal solutions from the initial solution within this limit (Q3). In general, the time taken to weaken a solution was correlated with the complexity of the benchmark.

To further investigate how effective Elrond was at exploring the space of candidate weakenings (Q4), we calculated the ratio of the total number of feature vectors gathered during weakening (#Gather) against the total number of positive feature vectors admitted by the final maximal specification (|$\phi^*$|). The latter number represents the set of vectors that a naïve exhaustive enumeration would need to consider in order to find our solution. In our experiments, Elrond only needed to consider at most 40% of the full search space. To get |$\phi^*$| for the six benchmarks on which Elrond returned a partially weakened solution, we increased the time bound to 24 hours and ran Elrond until it converged on the maximal solution. Elrond was unable to converge under this longer bound

---

6In the first stack benchmark, Elrond gathers more feature vectors than the total number because the algorithm may generate false feature vectors, as discussed in Section 5
for three of these benchmarks; for these three, we report the total number of feature vectors in the partial solution, which is indicated using a blue entry in the $|\phi^*|$ column.

In an attempt to quantify how well Elrond was able to generalize an initial solution, we asked the SMT solver to identify samples permitted under the weakened solution but not the original (Q5). The more general a weakened solution, the larger this space should be, so the solver should be able to more readily identify one of its elements. The results of this experiment are presented under the time$_d$ column. The initial solution was already maximal for three of our benchmarks, which is indicated via a “Max” entry in the time$_d$ column. In general, the solver was able to quickly find such samples for the remaining nineteen benchmarks. This search took longer for the benchmarks that hit the time bound, suggesting those solutions are either closer to the initial specifications or are otherwise more complicated for the SMT solver to handle.

![Fig. 5. run time(s) of first 1000 weakening iterations of benchmarks over time bound.](image)

When running our experiments without a time bound, we observed that individual iterations of the weakening loop took longer to complete over time. As an example, in the first hour of the second stack benchmark, Elrond performed 46 weakening steps per minute on average, but only averaged 33 after that. Based on this observation, we conjectured that it became harder for Elrond to find further generalizations as the weakening phase progresses. To test this hypothesis, we measured the time required by each of the first 1000 iterations of the weakening loop for the six benchmarks which hit the time bound. The results of this experiment are presented in Figure 5; the y-axis uses a logarithmic scale in order to account for times that range from 0.1s to 50s. The trajectory agrees with our hypothesis, and suggests diminishing returns for running the weakening procedure for long periods of time.

Finally, in order to show that Elrond produces useful specifications (Q1 and Q5), we used the Coq proof assistant to verify that implementations of the library functions from our benchmarks satisfied their inferred specifications. In total, Elrond inferred 68 specifications across all our experiments; we were able to verify 64 of these in Coq. Of these, three had initial specifications that were too strong; it was only after the weakening phase that the specifications could be used for verification. The four specifications that we were not able to verify came from the benchmarks whose weakening phase timed out, i.e. these specifications were not maximal. Taken together, these points suggest that the weakening phase results in more general (and thus more useful) specifications, and that Elrond is highly effective in finding meaningful specifications.

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. [2016] automatically infer specifications which

7The corresponding Coq developments are provided in the supplemental material.
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. [2016] use program synthesis to automatically learn
features on demand when inferring preconditions for data-structure manipulating library functions.
This approach is further extended by Miltner et al. [2020] to synthesize [Osera and Zdancewic 2015]
representation invariants that are sufficient to verify specifications of the operations of abstract data
types using. Rather than inferring specifications of libraries in isolation, this paper considers the
complementary problem of discovering those specifications in service of verifying 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 [Itzhaky et al. 2014, 2013]. 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. [2017] 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 specifications without the need to reflect
the implementation, which in our setting are unavailable, into the solver’s underlying logic.

**Abductive Inference.** As noted in Section 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 [Dillig et al. 2013] and specifications of functions
in a client program [Albarghouthi et al. 2016]. 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 post-condition. 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. 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 approach provides an alternative solution that combines data-driven methods with SMT-based
counterexample-guided refinement to discover library method specifications that effectively serve
as client-side 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. We thank the
reviewers for their detailed comments, and we look forward to incorporating their feedback in the
revised version of the paper.

Overview: ———
We emphasize that the primary novelty of our approach is the use of data to infer plausible explanations (i.e. specifications) of the behaviors of black-box library functions that ensure the safety of a client program. In this sense, we are tackling the same high-level multi-abduction problem as Albarghouthi et al. Unlike their work, however, our setting considers specifications involving algebraic data types, and must thus work in a different theory (EUF) that unlike theirs (linear integer arithmetic, (LIA)), lacks a complete abduction procedure.

Our approach, therefore, uses data in two ways in order to compensate for the lack of such a procedure: 1) we use concrete runs of black-box implementations to generate reasonable explanations (i.e. explanations that are consistent with implementation behavior); and, 2) we limit overfitting to these samples by using symbolic data from the SMT solver to ensure they are sufficiently general (i.e. maximally weak). The use of SMT-generated data for weakening specifications is in sharp contrast to other data-driven approaches that typically employ heuristics or rely on a learning procedure (e.g., decision trees) to drive generalization, and are thus not concerned with finding weakest specifications.

Our approach guarantees that even in the absence of library source code, inferred specifications are: 1) *safe* in terms of client assertions - executing a client using a library implementation that satisfies these specifications is always safe, and 2) *consistent* with any observations of library behavior produced from concrete tests.

Proposed Changes:

We propose to make the following modifications to the revision to address the reviewer’s concerns:

- To clarify our presentation, we will add more examples (especially for weakening), making space by moving some definitions and theorems to a technical report. - We will refine the presentation of the algorithm in Section 5 to provide better intuitions about its design. - We will add comparisons to the suggested related works (Reviewer-B and -C) per the discussion below.

Detailed Response:

**Reviewer A:**

**Dependence on random test generator, and sensitivity/robustness of the technique.**

We concur that, like all data-driven approaches, our algorithm is dependent on the quality of the underlying data (i.e. the tests generated by QuickCheck in the case of our initial solutions). It is precisely for this reason that our algorithm uses a second weakening phase to ensure that the final abduced specifications are not overfitted to or otherwise biased by the test generator. Our evaluation demonstrates how (lines 1067-1069) weakening can meaningfully combat such biases.

In addition, we observed that the sorts of shape properties (e.g., membership, ordering) used by our specifications and assertions are relatively under-constrained and are thus amenable to property-based random sampling. We do not ask, for example, for QuickCheck to generate inputs satisfying non-structural properties like "the 116th element of a list is equal to 5".

**Weakening procedure in Algorithm 1:**

In order to make Algorithm 1 concise, we adopted the convention that the "forall" on line 4 continues until Δ converges. To clarify this, we will add an explicit loop that performs this convergence check in the revised version of the paper.

**Theorem 5.2:**

At a high level, DiscoverMapping always halts because it considers a finite space of possible specifications, i.e. those drawn from Definition 4.4. At each iteration of the loop, the loop either terminates or strengthens one of the inferred specifications. Since there are only a finite number of such specifications, the loop must eventually terminate. Note that Algorithm 2 is not guaranteed to terminate, as it can always increase the number of quantified variables used by the hypothesis space.
The full proof of Theorem 5.2 can be found in the supplemental material (lines 1336 - 1344). Even though we do not limit the number quantified variables in our implementation, our experimental results (the $|\vec{u}|$ column in Table 4) show that typically 3 quantified variables are enough to solve our benchmarks.

**Line 3 of Weakening algorithm (Algorithm 3):**

Intuitively, this query can be interpreted as "there exists a safe execution (i.e. one that satisfies $\Phi$) consistent with the candidate specifications besides $\Delta(R_f)$ (the first part of implication), that is not currently allowed by the specification of $f$ (the second part of implication)". If this statement holds (i.e. the negation of this implication is UNSAT), then by Definition 3.5, the weakened specification is maximal. If it does not hold, the corresponding model represents a safe execution that should be allowed by the specification of $f$. Such models are examples of the weakening samples described in Section 4.2.

**Reviewer B:**

**Related Work:**

Thank you for providing these references. We will be sure to provide a detailed comparison in the revised version. To clarify the differences between our work and these:

- Like our setup, J. Su et al’s work extended from H.A. Nguyen et al. infers specifications of library APIs from client information. However, they try to infer correct library specifications under the assumption that client is correct. On the other hand, we do abduction to provide guarantees that any library implementation must satisfy to ensure the client is safe. Our approach does not assume clients are always safe - e.g., Elrond was able to identify safety violations in 5 of our benchmarks (lines 1007-1010). Another distinguishing feature in our work is the focus on generating maximally weak specifications to overcome generating overfitted specifications.

- S. Qin et al. also infer specifications for unknown procedures in a rich domain involves the shape property of datatypes. However, because their technique does not involve any queries to the library implementation and does not do weakening like ours, inferred specifications may be over-fitted to the client safety property; e.g., their approach may fail to infer a specification that admits duplicate stack elements (lines 432-435). In contrast to our work, their method also requires users to understand the underlying representation of the datatype used by the library method, and present explicit interpretations of predicates used in specifications sufficient to verify the client. Finally, unlike Qin, we provide an algorithmic weakening procedure to refine inferred specifications.

- The work of R. Pandita et al. learns specifications from comments in natural language but does not provide safety and consistency guarantees, as discussed in the overview section above.

1. **Method Predicates:**

Method predicates are provided by the user. In our example, we might implement predicates $hd$ and $mem$ as:

```plaintext
fun hd(s, u) = (Stack.top s = u)
fun mem(s, u) = (Stack.top s = v) || mem(Stack.pop s, v)
```

While it is possible to naively translate each library method as a predicate, such an approach may not be useful for verification. Some functions may be irrelevant to client assertions, unnecessarily increasing the hypothesis space from which specifications are drawn. Conversely, the library may not include functions for desirable predicates; e.g., the Stack library does not provide a $Stack.mem$ function that is nonetheless relevant for verifying our running example. Notably, we impose no restriction on the structure of method predicates - observe that the definition of $mem$ is recursive, and that both $Stack.top$ and $Stack.pop$ refer to black-box implementations of Stack library methods.

2. **Prenex Universally-Quantified Formula:**

As with any automated verification exercise, we face a tradeoff between expressive power and algorithm efficiency; we chose to constrain our specifications in this way in order to ensure that
our SMT queries were in the decidable EPR fragment. As verification technology improves, we could certainly relax this restriction. EPR-based encodings are commonly used in practice, and was expressive enough for all our benchmarks.

The main sorts of properties that we do not support as a consequence of this choice are those which use quantifier alternation, e.g. "for every element in a stream, there exists another larger element that appears after than it" ($\forall u, \exists v, \text{mem}(l, u) \implies (\text{mem}(l, v) \land \text{ord}(l, u, v) \land u \leq v)$).

**Reviewer C:**

1. Understanding the counter-example (CEX) and method predicates

CEX is a model produced by the Z3, and it provides interpretations to the query’s uninterpreted predicates (in this case $\text{hd}$ and $\text{mem}$) as anonymous functions (see below). Even though we have implementations of these predicates, directly encoding their semantics in logic is problematic given e.g., the recursive structure of $\text{mem}$. Thus, the theorem prover views these predicates as uninterpreted and therefore the models it generates may well be inconsistent with their actual semantics.

As an example, one possible SMT model for the CEX given on line 288 is:

- $v_{\text{top}} = 19$
- $v = 1$
- $v_{\text{concat}} = 2$
- $v_{\text{tail}} = 3$
- $\text{hd} = \lambda l, u. \text{false}$
- $\text{mem} = \lambda l, u. \text{if } (l = 1 \lor l = 2 \lor l = 3) \land u = 19 \text{ then } \text{true } \text{ else } \text{false}$

Algebraic datatypes are encoded as integers, and we replace the 1, 2, and 3 in the interpretation of $\text{mem}$ with $v$, $v_{\text{concat}}$ and $v_{\text{tail}}$ when presenting CEX in the paper.

2. Role of quantified variables:

Many of our method predicates are relational; increasing the number of quantified variables broadens our hypothesis space to include richer relationships: expressing transitivity, for example, require at least three quantified variables: $\forall u, v, w. \text{ord}(l, u, v) \land \text{ord}(l, v, w) \implies \text{ord}(l, u, w)$.

Similarly, any binary predicate, e.g. $\text{mem}(l, u)$, requires 2 quantified variables. Using a grammar of predicates with increasing size bounds to encode the hypothesis space is equivalent to our approach, we choose increasing quantified variables to more closely align with our definition of maximality (Definitions 3.5 and 3.6).

3. Concrete tests in weakening:

The weakening phase does not check if there is a concrete test corresponding to the logical samples produced by the SMT solver; since weakened specifications are guaranteed to preserve safety, this will not compromise the soundness of our algorithm. Where there is no corresponding test, we will have weakened the specification to cover a vacuously true scenario.

Comparison to prior work on data-driven invariant inference:

Broadly speaking, our approach differs from prior work on data-driven invariant inference insofar as we cannot appeal to any source of ground truth (e.g., SMT verification (e.g., Padhi et al., PLDI’16), or bounded model checking (Miltner et al., PLDI’20)). This leads to the need for a weakening procedure to facilitate generalization to ultimately aid client-side verification. We believe that techniques like Miltner et al. can be complementary to ours, using our abduced specifications to infer and verify representation invariants.

Comparison to CHC solving:

While it is possible to frame our problem as an instance of CHC solving (e.g., Zhu et al. (PLDI’18)), we would still be left with the important challenge of generating maximally weak specifications. As observed by Albaghouthi et. al., "...Horn clauses does not address the question of optimality and would allow trivial solutions that set unconstrained predicates to false, as allowed by Horn-clause
semantics. In contrast, our multi-abduction solver ensures that we compute maximal, non-trivial solutions..." (p. 800)

VC generation:

We infer VCs for each control flow path, and our full algorithm considers the conjunction of all these VCs at once. To keep the description of the algorithms in Sections 3-5 concise, we only consider a single VC on Sec 3-5; the complete algorithm for multiple VCs is a straightforward generalization of this approach and is provided in the supplemental material. This is currently noted in the footnote on page 16, but we will clarify this point in the revised version.

Negative/positive samples:

The statement of Theorem 4.9 quantifies over the entire space of negative samples, and every positive sample generated by random testing (this is what the phrase "entailed by $Γ_Γ$ and $Γ_p$ means). The former ensures that our specifications rule out every unsafe execution, while the latter ensures they are consistent with our observations of the libraries' behavior.

Comparison to Bastani et. al (POPL '15)

While their work considers a similar problem to ours, analysis of clients when some pieces of code are missing, discovering minimal sufficient assumptions, there are several major differences between our approaches: 1) their specification domain is limited to CFL reachability (alias and taint specifications), while our hypothesis space involves shape properties (membership, ordering...) over algebraic datatypes; 2) we guarantee specification consistency with respect to our observations on the black-box implementation of the libraries; and, (3) they do not provide an explicit weakening procedure to generalize their solution.

REFERENCES


