Data-Driven Abductive Inference of Library Specifications

ZHE ZHOU, Purdue University, USA
ROBERT DICKERSON, Purdue University, USA
BENJAMIN DELAWARE, Purdue University, USA
SURESH JAGANNATHAN, Purdue University, USA

116

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, along with 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.

CCS Concepts: • **Computing methodologies → Classification and regression trees**; • **Software and its engineering → Software verification and validation**; **Automated static analysis**.

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

ACM Reference Format:
Zhe Zhou, Robert Dickerson, Benjamin Delaware, and Suresh Jagannathan. 2021. Data-Driven Abductive Inference of Library Specifications. *Proc. ACM Program. Lang.* 5, OOPSLA, Article 116 (October 2021), 29 pages. https://doi.org/10.1145/3485493

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 encapsulate the interface between the client and the library, each may be independently verified without access to the other's implementation. 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 the verification of clients without making (often unwarranted) assumptions about the behavior of the library. This problem is further exacerbated when libraries expose rich datatype functionality, which often leads to specifications that rely on inductive invariants [Dillig et al. 2013; Itzhaky et al. 2014] and complex

Authors' addresses: Zhe Zhou, Purdue University, USA; Robert Dickerson, Purdue University, USA; Benjamin Delaware, Purdue University, USA; Suresh Jagannathan, Purdue University, USA.

This work is licensed under a Creative Commons Attribution 4.0 International License.
© 2021 Copyright held by the owner/author(s).
2475-1421/2021/10-ART116
https://doi.org/10.1145/3485493

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 frequently occurring case of client programs that make use of data structure libraries with unavailable implementations.

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.

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

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*, "a is the head of stack s", $hd(s, a)$, and "a is a member of stack s", $mem(s, a)$, to write our postcondition:

$$\forall u, (hd(v, u) \implies hd(\mathsf{s}_1, u) \vee hd(\mathsf{s}_2, u)) \wedge (mem(v, u) \iff mem(\mathsf{s}_1, u) \vee mem(\mathsf{s}_2, u)) \qquad (\phi_{\mathsf{concat}})$$

We assume these method predicates are associated with (possibly blackbox) implementations that we can use to check the specifications in which they appear. For example, *hd* may be defined in terms of the stack operations top and is_empty, while the implementation of *mem* might additionally use tail. The variable v in ϕ_{concat} is used to represent the output stack of concat. The above 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 method predicates (*hd* and *mem*) and library functions (push, top, is_empty, and tail) as uninterpreted function symbols, it is straightforward to generate verification conditions (VCs), e.g. using weakest precondition inference, which can be handed off to an off-the-shelf SMT solver like Z3 to check. 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 on these behaviors outside the client VCs. For example, the prover might assume the formula $\neg hd(\mathsf{s}_1, \mathsf{top}(\mathsf{s}_1))$ is valid, i.e. that the result of $\mathsf{top}(\mathsf{s}_1)$ is not the head of s_1. This claim is obviously inconsistent with the client's expectation of top's semantics, but it is not disallowed by any constraints in the SMT query. Using this assumption, Z3 may return the following counterexample: $\exists s\ u. hd(s, u) \wedge \neg mem(s, u)$. This counterexample, which is obviously incongruous with the intended semantics of *hd* and *mem*, occurs because the expected relationship between *hd* and *mem* is lost when the predicates are embedded as uninterpreted functions in the SMT query.

To overcome this problem, we need stronger specifications for the library methods, defined in terms of these method predicates, that are sufficient to imply the desired client postcondition. In particular, these specifications should rule out spurious unsafe executions such as the counter-example given above. In the (quite likely) scenario that such library specifications are not already available, a reasonable fallback is to infer *some* specifications for these functions that are strong enough to ensure the safety of the client. Traditional approaches to specification inference usually adopt 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

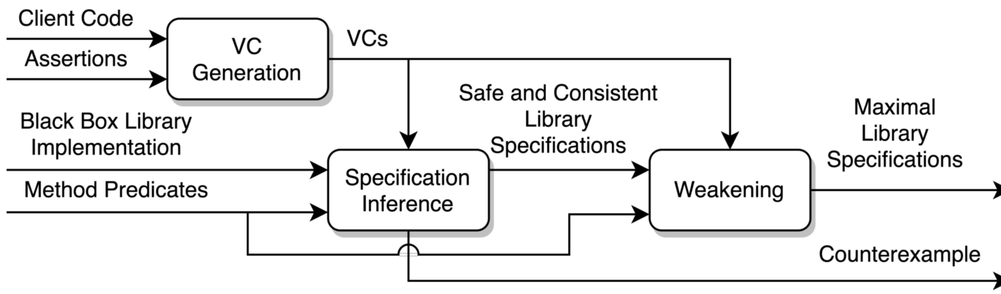

Fig. 1. Elrond pipeline.

nature of the specifications we need to infer are impacted by the verification demands of the client. In this setting, some form of data-driven inference [Miltner et al. 2020; Padhi et al. 2016; Zhu et al. 2016] can be beneficial. Such an approach may be tailored to the client context in which the library methods are used, postulating candidate specifications for library methods based on observations of their input-output behavior. Unfortunately, completely blackbox data-driven approaches are susceptible to overfitting on the set of observations used to train them, and can thus discount reasonable and safe behaviors of the underlying library functions.

To address the problem of overfitting, we might instead consider attacking this problem from a purely logical standpoint, treating specification inference as an instance of a multi-abductive inference problem [Albarghouthi et al. 2016] that tries to find formulae $\mathsf{R}_{\mathsf{push}}$, $\mathsf{R}_{\mathsf{top}}$, $\mathsf{R}_{\mathsf{tail}}$ and $\mathsf{R}_{\mathsf{is\_empty}}$ such that $\bigwedge R_i \not\models \bot$ and yet which are sufficient to prove the desired verification condition. While such problems have been previously solved over linear integer arithmetic constraints [Albarghouthi 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 in a way 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 methodology that allows us to postulate specifications built from method predicates sufficient to prove the postcondition in a purely blackbox setting. The specifications learned by this procedure are guaranteed to be both *consistent* with the observed input-output behavior of the blackbox library implementations and *safe* with respect to the postcondition of the client program. As there may be many such specifications, we also endeavor to find a *maximal* one that is at least as weak as every other safe and consistent specification, in order to avoid overfitting to observed library behaviors. Our algorithm applies another data-driven weakening procedure to find these maximal specifications.

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) assertions about the behavior of this client program; and, (c) a set of method predicates (e.g., *hd* or *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 maximal specifications for the library methods used by the client program, or a counterexample that demonstrates a violation of the postcondition. The notion of "weakest" used in our definition of maximal 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 for 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. An artifact containing this tool and our benchmark suite is publicly available [Zhou et al. 2021b].

The remainder of the paper is structured as follows. The next section presents an overview of our approach using a detailed example to motivate its key ideas . 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 OVERVIEW AND MOTIVATION

We divide the inference of maximal library specifications into two stages, which are represented as the "Specification Inference" and "Weakening" components in Figure 1. Both stages leverage data-driven learning to overcome the lack of a purely logical abduction procedure for our specification language. The initial inference stage learns a set of safe and consistent specifications from a combination of concrete tests and verifier-provided counterexamples. The next stage then weakens these specifications by iteratively augmenting this data set with additional safe behaviors until a set of maximal specifications are found.

Figure 2 provides a more detailed depiction of the initial specification inference stage in Elrond. Starting from an initial set of maximally permissive specifications, this stage iteratively refines the set of candidate specifications until either a set of safe and consistent solutions or a counterexample witnessing an unsafe execution is found.

Each iteration first uses a property-based sampler, e.g. QuickCheck [Claessen and Hughes 2000], to look for executions of the blackbox library implementations that are inconsistent with the current set of inferred specifications. The reliance on a generator to provide high-quality tests provides yet more motivation for the subsequent weakening phase, in order to ensure that the final abduced specifications are not overfitted to or otherwise biased by the tests provided by the generator. At the same time, we also observe that the sorts of shape properties (e.g., membership and ordering) used in 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 "*a list whose* 116^{th} *element is equal to 5.*"

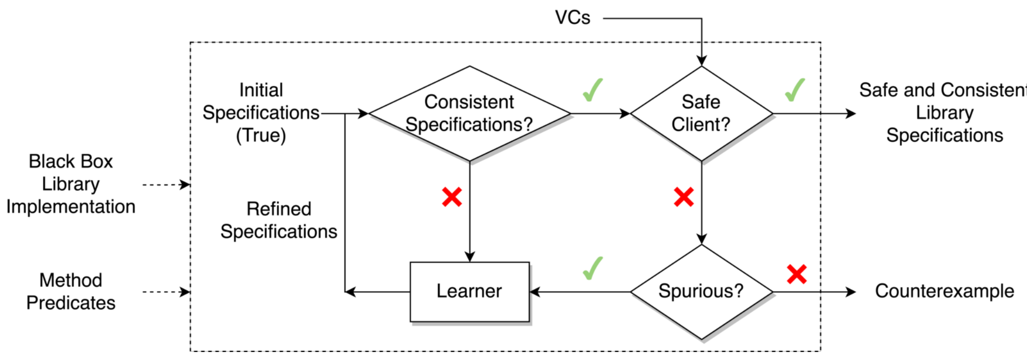

Fig. 2. The details of Elrond's Specification Inference component from Figure 1. The blackbox library implementation and method predicates are used by multiple components; their corresponding arrows within the diagram are omitted for clarity.

Any tests that are disallowed by the current solution are passed to a learner which uses them to generalize the current specification. If no inconsistencies are detected, Elrond attempts to verify the client against the candidate specifications using a theorem prover. If the inferred specifications are sufficient to prove client safety, the loop exits, returning the discovered solution. If not, the verifier has identified a model that represents a *potential* safety violation. The model is then analyzed in an attempt to extract test inputs that trigger a safety violation. If we are unable to find such a counterexample, the model is most likely incongruous with the semantics of the method predicates and is thus spurious. In this case, the model is passed to the learner so that it can be used to strengthen candidate specifications, preventing this and similar spurious counterexamples from manifesting in subsequent iterations.

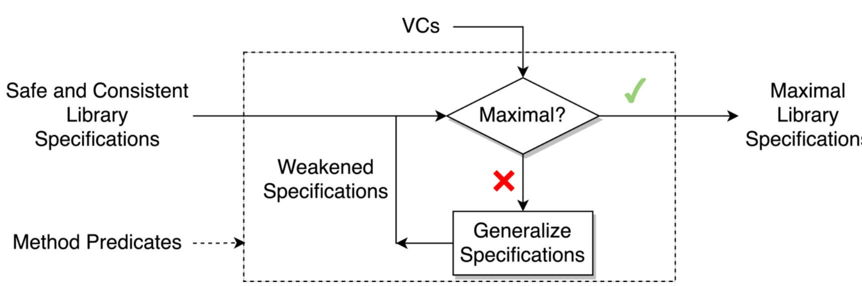

Fig. 3. The details of Elrond's Weakening component from Figure 1. Arrows for method predicates are again omitted for clarity.

While the previous loop is guaranteed to return safe and consistent solutions, it may find specifications that are nonetheless *too strong* with respect to the underlying library implementation. This occurs when the property-based sampler fails to find a test that identifies an inconsistent specification, which may happen when the input space of a library function is very large. To combat overfitting specifications to test data, candidate solutions are iteratively weakened using the data-driven counterexample-guided refinement loop depicted in Figure 3. The data in this phase is supplied by the underlying theorem prover rather than a concrete test generator. Each iteration of the refinement loop first attempts to find a safe execution of the client program that is disallowed by the current set of specifications. If no such execution can be found, the specifications are maximal and the loop terminates. Otherwise, the identified execution is passed to a learner, which uses it to generalize the candidate solution so that the execution is permitted before continuing the refinement loop. The learner always generalizes candidate specifications, maintaining the invariant that the current solution is consistent with all previously observed library behaviors.

2.1 Elrond **in Action**

To illustrate our approach in more detail, we apply it to the stack concatenation example from the introduction. Given the postcondition ϕ_{concat} and the implementation of concat from Section 1, Elrond generates a formula that can be simplified to the following implication:

$$
\begin{array}{l}
\forall \mathsf{s}_1, \mathsf{s}_2, \nu, \nu_{\text{top}}, \nu_{\text{tail}}, \nu_{\text{concat}}, \nu_{\text{is\_empty}}, \\
(\mathsf{R}_{\text{is\_empty}}(\mathsf{s}_1, \nu_{\text{is\_empty}}) \land \mathsf{R}_{\text{top}}(\mathsf{s}_1, \nu_{\text{top}}) \land \mathsf{R}_{\text{tail}}(\mathsf{s}_1, \nu_{\text{tail}}) \land \mathsf{R}_{\text{push}}(\nu_{\text{top}}, \nu_{\text{concat}}, \nu)) \\
\quad \implies \\
((\nu_{\text{is\_empty}} = \bot \land \forall u, (hd(\nu_{\text{concat}}, u) \implies hd(\nu_{\text{tail}}, u) \lor hd(\mathsf{s}_2, u)) \land \\
\qquad (mem(\nu_{\text{concat}}, u) \iff mem(\nu_{\text{tail}}, u) \lor mem(\mathsf{s}_2, u))) \implies \\
\quad \forall u, (hd(\nu, u) \implies hd(\mathsf{s}_1, u) \lor hd(\mathsf{s}_2, u)) \land (mem(\nu, u) \iff mem(\mathsf{s}_1, u) \lor mem(\mathsf{s}_2, u)))
\end{array}
$$

The four predicates in the premise of this formula correspond to the four library functions (push, top, is_empty and tail) invoked in a recursive call to concat. The specification of a blackbox library function f in our assertion logic is represented as a *placeholder predicate*: an uninterpreted predicate that relates the parameters of f to its return value. For a library function f, we adopt a naming convention of R_{f} and ν_f for its placeholder predicates and return values, respectively. The predicate $\mathsf{R}_{\text{top}}(\mathsf{s}_1, \nu_{\text{top}})$ in the above formula, for example, says the variable ν_{top} holds the return value of the call to `Stack.top s1` in concat. The result of the recursive call to concat is similarly denoted as ν_{concat}. The conclusion of the formula encodes the expected verification condition for a recursive call to concat: namely, that if the result of the call to `Stack.is_empty s1` is false and the recursive call to `concat (Stack.tail s1) s2` satisfies ϕ_{concat}, then the result of concat must also satisfy ϕ_{concat}. The remainder of this section refers to the premise and conclusion of this implication as Σ_{concat} and Φ_{concat}, respectively.

Method Predicates. From a logical standpoint, the method predicates used in Φ_{concat} are simply uninterpreted function symbols which have no intrinsic semantics. This representation allows our specifications to use predicates whose semantics may be difficult to encode directly in the logic. Embedding recursively defined predicates like mem, for example, requires particular care [Zhu et al. 2016]. In order to ensure that the specifications inferred by Elrond are tethered to reality, users must also supply Elrond with implementations (possibly blackbox) of these predicates. One possible implementation for *hd* and *mem* is:

```
let hd(s, u) =
   if Stack.is_empty s then false else (Stack.top s) == u

let rec mem(s, u) =
   if Stack.is_empty s then false else (Stack.top s = u) ||
                                       (mem(Stack.tail s, u))
```

where `Stack.is_empty`, `Stack.top` and `Stack.tail` refer to blackbox implementations of stack library methods.

While it is possible to naïvely include a method predicate for each library method, such an approach may not be useful for verification. Some functions may be irrelevant to client assertions, unnecessarily increasing the set of possible specifications that must be considered. Conversely, the library may not include functions for desirable predicates; e.g., the Stack library does not provide a Stack.mem function, although it is quite relevant for verifying our running example.

$$
\begin{aligned}
&\mathsf{R}_{\mathsf{is\_empty}}(\mathsf{s}, \nu) \mapsto \forall u, (\nu \implies \neg mem(\mathsf{s}, u)) \land (\neg \nu \land hd(\mathsf{s}, u) \implies mem(\mathsf{s}, u))\\
&\mathsf{R}_{\mathsf{top}}(\mathsf{s}, \nu) \mapsto \forall u, mem(\mathsf{s}, \nu) \land (\nu = u \iff hd(\mathsf{s}, u))\\
&\mathsf{R}_{\mathsf{tail}}(\mathsf{s}, \nu) \mapsto \forall u, (mem(\mathsf{s}, u) \implies (mem(\nu, u) \lor hd(\mathsf{s}, u))) \land\\
&\qquad\qquad\qquad ((mem(\nu, u) \lor hd(\nu, u)) \implies mem(\mathsf{s}, u))\\
&\mathsf{R}_{\mathsf{push}}(\mathsf{x}, \mathsf{s}, \nu) \mapsto \forall u, (mem(\nu, u) \land mem(\mathsf{s}, u) \implies \neg(\mathsf{x} = u \lor hd(\nu, u))) \land\\
&\qquad\qquad\qquad\quad (mem(\nu, u) \land \neg mem(\mathsf{s}, u) \implies (\mathsf{x} = u \land hd(\nu, u))) \land\\
&\qquad\qquad\qquad\quad ((\mathsf{x} = u \lor hd(\nu, u) \lor hd(\mathsf{s}, u) \lor mem(\mathsf{s}, u)) \implies mem(\nu, u))
\end{aligned}
$$

Fig. 4. Candidate verification interface for the concat example.

Solution Space. Our ultimate goal is to find a mapping from each placeholder predicate in Σ_{concat} to an interpretation that entails Φ_{concat}. We refer to such a mapping as a *verification interface*; Figure 4 presents a potential verification interface for our running example. Not every mapping that ensures the safety of concat is reasonable, however. At one extreme, interpreting every predicate as $\perp$ ensures the safety of the client, but does not capture the behavior of any sensible stack implementation. Our goal, then, is to find interpretations that are *general* enough to cover a range of possible implementations. From a purely logical perspective, this an instance of a multi-abductive inference problem that tries to find the weakest interpretations of $\mathsf{R}_{\mathsf{push}}$, $\mathsf{R}_{\mathsf{top}}$, $\mathsf{R}_{\mathsf{tail}}$ and $\mathsf{R}_{\mathsf{is\_empty}}$ in terms of predicates *mem* and *hd* such that the interpretations are self-consistent (i.e. $\bigwedge R_i \not\models \perp$) and which are sufficient to prove the desired verification condition. While solutions to the multi-abduction problem have been developed for domains that admit quantifier elimination, e.g. linear integer arithmetic constraints Albarghouthi et al. [2016], there is no purely logical solution for formulae involving equalities over uninterpreted functions. 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].

2.2 Data-Driven Abduction

We overcome these challenges by adopting a data-driven approach to abducing maximal library specifications, framing the problem as one of training a Boolean classifier on a set of *example behaviors* for each function f. Under this interpretation, a classifier represents a specification of the "acceptable" behaviors of f. Thus, the goal of the specification inference stage of our algorithm is to learn a set of classifiers that recognize the behaviors of each f that a) are consistent with f's underlying implementation and b) preserve the safety of the client program. As discussed at the start of this section, this algorithm uses both an SMT-based verifier and a property-based test generator as sources of training data for our learner. The former identifies example behaviors consistent with $\neg(\Sigma_{\mathsf{concat}} \implies \Phi_{\mathsf{concat}})$; these are labelled as "negative" examples, so that the learned specifications can help the solver rule out behaviors that are inconsistent with the semantics of the method predicates or that produce unsafe executions (i.e., interpretations that would violate the postcondition). Example behaviors drawn from tests are labelled as "positive", so that our learner is biased towards explanations that are consistent with the (unknown) implementations of library functions. 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.

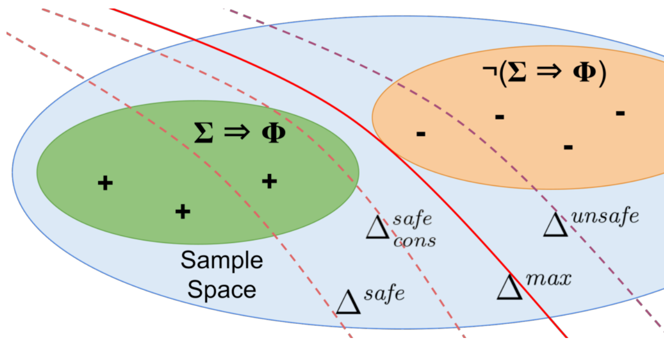

Fig. 5. Elements in the circle labelled $\Sigma \Rightarrow \Phi$ represent positive examples, while the orange $\neg(\Sigma \Rightarrow \Phi)$ circle contains negative examples. Our algorithm generates a mapping Δ, which maps placeholder predicates to specifications, separating the two.

Figure 5 depicts the space of example behaviors for our learner, as well four potential verification interfaces. Each of these represents a potential solution in the *hypothesis space* for this learner, which is tasked with building a classifier that separates negative (-) and positive examples (+) for each library function. The dashed purple line, labelled Δ^{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 the client, but which are suboptimal. Δ^{safe} is safe but inconsistent with the observed behaviors of the library implementation, and is thus overly restrictive. Δ^{safe}_{cons} is safe and consistent, but not maximal, as there exists a weaker verification interface (Δ^{max}) in the hypothesis space that is still safe and consistent. Intuitively, the goal of our first phase is to identify Δ^{safe}_{cons}, which is then weakened by the second phase to produce Δ^{max} .

Hypothesis space. Our learner limits the shape of solutions it considers so that inferred specifications are both amenable to automatic verification and strong enough to verify specified postconditions. To enable automated verification of client programs, potential specifications are required to be 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:

$$\mathsf{push}(x, l) = v : \left\{ \begin{array}{ll} \forall u, mem(l, u) \implies mem(v, u), & \forall u, \neg mem(v, u), \\ \forall u, u = x, & \forall u, u = x \iff hd(v, u), \\ \forall u, mem(l, u) \land hd(l, u), & \ldots \end{array} \right\}$$

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 values of functions. The literals in the above formulae are simply applications of hd and mem to l, v and u, and the equality $u = x$. We automatically discard equalities between terms of different types, e.g. $l = x$. The *feature set* for the predicate $\mathsf{R}_{\mathsf{push}}$, i.e. the set of atomic elements used to construct its specification, is thus: $\{hd(l, u),\ mem(l, u),\ hd(v', u),\ mem(v, u),\ x = u\}$.

Table 1. Potential negative feature vectors extracted from Cex.

$\mathsf{R}_{\mathsf{top}}(\mathsf{s}_1, v_{\mathsf{top}})$	$\forall \mathsf{u}$	$hd(\mathsf{s}_1, \mathsf{u})$	$mem(\mathsf{s}_1, \mathsf{u})$	$v_{\mathsf{top}} = \mathsf{u}$		
	$\mathsf{u} = \mathsf{a}$	false	false	true		
	$\mathsf{u} \neq \mathsf{a}$	false	false	false		
$\mathsf{R}_{\mathsf{tail}}(\mathsf{s}_1, v_{\mathsf{tail}})$	$\forall \mathsf{u}$	$hd(\mathsf{s}_1, \mathsf{u})$	$mem(\mathsf{s}_1, \mathsf{u})$	$hd(v_{\mathsf{tail}}, \mathsf{u})$	$mem(v_{\mathsf{tail}}, \mathsf{u})$	
	$\mathsf{u} = \mathsf{a}$	false	false	false	true	
	$\mathsf{u} \neq \mathsf{a}$	false	false	false	false	
$\mathsf{R}_{\mathsf{push}}(v_{top}, v_{concat}, v)$	$\forall \mathsf{u}$	$hd(v_{\mathsf{concat}}, \mathsf{u})$	$mem(v_{\mathsf{concat}}, \mathsf{u})$	$hd(v, \mathsf{u})$	$mem(v, \mathsf{u})$	$v_{\mathsf{top}} = \mathsf{u}$
	$\mathsf{u} = \mathsf{a}$	false	true	false	true	true
	$\mathsf{u} \neq \mathsf{a}$	false	false	false	false	false
$\mathsf{R}_{\mathsf{is\_empty}}(\mathsf{s}_1, v_{\mathsf{is\_empty}})$	$\forall \mathsf{u}$	$hd(\mathsf{s}_1, \mathsf{u})$	$mem(\mathsf{s}_1, \mathsf{u})$	$v_{\mathsf{is\_empty}}$		
	$\mathsf{u} = \mathsf{a}$	false	false	false		
	$\mathsf{u} \neq \mathsf{a}$	false	false	false		

Training Data. We now consider how to represent library behaviors in a form that is useful for learning a solution in our hypothesis space. To illustrate our chosen representation, consider the counterexample produced by an off-the-shelf theorem prover when asked to verify the formula $\Sigma_{\mathsf{concat}} \implies \Phi_{\mathsf{concat}}$ from our running stack example, where the set of candidate specifications are initialized to true (i.e., $\forall \mathsf{f}.\, \mathsf{R}_{\mathsf{f}} \mapsto \top$):

$$\forall l\ u, \neg hd(l, u) \land v_{\mathsf{top}} = \mathsf{a} \land (mem(l, u) \iff ((l = v \lor l = v_{\mathsf{concat}} \lor l = v_{\mathsf{tail}}) \land u = \mathsf{a})) \qquad (\textsc{Cex})$$

Intuitively, this counterexample asserts that the stacks v, $v_{\mathtt{tail}}$, and $v_{\mathtt{concat}}$ contain exactly one element, the constant a, and the other two stacks, s_1 and s_2, are empty. This assertion indeed violates the second conjunct of the postcondition, $mem(v, u) \iff mem(\mathsf{s}_1, u) \lor mem(\mathsf{s}_2, u)$, but it is inconsistent with the expected semantics of the library functions, and can thus be safely ignored. The verifier generates this counterexample because the interpretations of the placeholder predicates in Σ_{concat} are too permissive. In order for the verifier to rule out this counterexample, the placeholder predicates need to be strengthened to rule out this inconsistent behavior.

Ignoring how we identify this counterexample as spurious for now, note that there are many ways to strengthen these specifications. One approach is to focus on one particular function at a time. For example, we could choose to refine the specification of $\mathsf{R}_{\mathsf{top}}$ so that it guarantees that v_{top} is a member of s_1. Alternatively, we could focus on $\mathsf{R}_{\mathsf{tail}}$, ensuring the members of v_{tail} are also contained by s_1. In general, however, it may be necessary to strengthen multiple specifications at once. Therefore, instead of focusing on one specification at a time, we learn refined specifications simultaneously.

The first step to refining our placeholder specifications is to extract data from Cex in a form that can be used to train a classifier. We do this by using the assignments to the arguments of the placeholder predicates 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 placeholder predicates that can be strengthened to rule out this counterexample. The second column gives the feature vectors for a particular instantiation of the quantified variables (*e.g.*, u) of the placeholders. The subsequent columns list applications of method predicates, with the rows underneath listing the valuation of these predicates in the offending run. The second row corresponds to the assertion

Table 2. Positive feature vectors extracted from executing `concat([a], [b])`.

$\mathsf{R_{top}}(\mathsf{s_1}, \nu_{\mathsf{top}})$	$\forall \mathsf{u}$	$hd(\mathsf{s_1}, \mathsf{u})$	$mem(\mathsf{s_1}, \mathsf{u})$	$\nu_{\mathsf{top}} = \mathsf{u}$		
	$\mathsf{u} = \mathsf{a}$	true	true	true		
	$\mathsf{u} = \mathsf{b}$	false	false	false		
$\mathsf{R_{tail}}(\mathsf{s_1}, \nu_{\mathsf{tail}})$	$\forall \mathsf{u}$	$hd(\mathsf{s_1}, \mathsf{u})$	$mem(\mathsf{s_1}, \mathsf{u})$	$hd(\nu_{\mathsf{tail}}, \mathsf{u})$	$mem(\nu_{\mathsf{tail}}, \mathsf{u})$	
	$\mathsf{u} = \mathsf{a}$	true	true	false	false	
	$\mathsf{u} = \mathsf{b}$	false	false	false	false	
$\mathsf{R_{push}}(\nu_{top}, \nu_{concat}, \nu)$	$\forall \mathsf{u}$	$hd(\nu_{\mathsf{concat}}, \mathsf{u})$	$mem(\nu_{\mathsf{concat}}, \mathsf{u})$	$hd(\nu, \mathsf{u})$	$mem(\nu, \mathsf{u})$	$\nu_{\mathsf{top}} = \mathsf{u}$
	$\mathsf{u} = \mathsf{a}$	false	false	true	true	true
	$\mathsf{u} = \mathsf{b}$	true	true	false	true	false
$\mathsf{R_{is\_empty}}(\mathsf{s_1}, \nu_{\mathsf{is\_empty}})$	$\forall \mathsf{u}$	$hd(\mathsf{s_1}, \mathsf{u})$	$mem(\mathsf{s_1}, \mathsf{u})$	$\nu_{\mathsf{is\_empty}}$		
	$\mathsf{u} = \mathsf{a}$	true	true	false		
	$\mathsf{u} = \mathsf{b}$	false	false	false		

Table 3. Disjoint positive and negative feature vectors of $\mathsf{R_{top}}$.

$\mathsf{R_{top}}(\mathsf{s_1}, \nu_{\mathsf{top}})$	$hd(\mathsf{s_1}, \mathsf{u})$	$mem(\mathsf{s_1}, \mathsf{u})$	$\nu_{\mathsf{top}} = \mathsf{u}$
-	false	false	true
+	true	true	true
+	false	false	false

that $\neg hd(\mathsf{s_1}, \mathsf{a}) \wedge \neg mem(\mathsf{s_1}, \mathsf{a}) \wedge \nu_{\mathsf{top}} = \mathsf{a}$, 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 vector*, and a classifier (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. As an example, the first feature vector for $\mathsf{R_{top}}$ (the second row of Table 1) states that the result of `top s1` is not the head element of the input stack (since $\nu_{\mathsf{top}} = \mathsf{u}$ is true and $hd(\mathsf{s_1}, \mathsf{u})$ is false), and thus is inconsistent with *any* reasonable implementation of top. In contrast, the next feature vector is compatible with an execution of top where, e.g. $\mathsf{top}([1]) = 1$ and $\mathsf{u} = 2$. The second feature vector represents a behavior that is consistent with the underlying library implementation and that should be allowed by the learned specification. The consistency checker generates this positive training data via random testing of the client program. To see how we extract positive feature vectors from training data, consider the execution of `concat` with the inputs $\mathsf{s_1} = [\mathsf{a}]$ and $\mathsf{s_2} = [\mathsf{b}]$, which produces the following assignment to program variables:

$$\nu_{\mathsf{is\_empty}} = \bot, \nu_{\mathsf{top}} = \mathsf{a}, \nu_{\mathsf{tail}} = [], \nu_{\mathsf{concat}} = [\mathsf{b}], \nu = [\mathsf{a}; \mathsf{b}]$$

Similar to how we built negative feature vectors from Cex, we can construct feature vectors for each function specification from these assignments. Table 2 illustrates the feature vectors corresponding to this assignment. Consider the second row where u is instantiated with a: under this assignment, $hd(\mathsf{s_1}, \mathsf{u}) \equiv \mathsf{hd}([\mathsf{a}], \mathsf{a})$ is true, as are $mem(\mathsf{s_1}, \mathsf{u}) \equiv \mathsf{mem}([\mathsf{a}], \mathsf{a})$ and $\nu_{\mathsf{top}} = \mathsf{u} \equiv \mathsf{a} = \mathsf{a}$.

Classification. In order to train a classifier, we need to label the extracted feature vectors as either positive or negative. In other words, we need to identify behaviors that should (and should not) be allowed by the inferred specification. Assigning labels is not as straightforward as labelling the feature vectors extracted from counterexamples as negative and those extracted from testing as positive, as the two sets can overlap. We observe this in the vectors of $\mathsf{R}_{\mathsf{top}}$ from Table 1 and Table 2: the interpretation $hd(\mathsf{s}_1, \mathsf{u}) \mapsto \mathsf{false}; mem(\mathsf{s}_1, \mathsf{u}) \mapsto \mathsf{false}; \nu_{\mathsf{top}} = \mathsf{u} \mapsto \mathsf{false};$ occurs in both tables. Intuitively, we do not want to strengthen the specification of $\mathsf{R}_{\mathsf{top}}$ to rule out this interpretation, as the positive sample is a witness that this execution is consistent with the implementation of $\mathsf{R}_{\mathsf{top}}$. Ultimately, therefore, the specification must be relaxed to allow the execution. In cases where a negative feature vector conflicts with a positive feature vector, we identify the potential negative feature vector as positive and remove it from the learner's negative feature vector set. This strategy is similar to the one used by Miltner et al. [2020] to deal with inductiveness counterexamples.

Thankfully, as long as the counterexample set contains at least one feature vector not known to be consistent with the underlying library implementation, we can strengthen the specifications to disallow it. The first feature vector for $\mathsf{R}_{\mathsf{top}}$ in Table 1 represents one such infeasible execution. This vector encodes the case where u is the output of top but is not a member or head of the input stack. Clearly, no reasonable implementation 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 $\mathsf{R}_{\mathsf{top}}$.

Given this labelled set of positive and negative feature vectors, our data-driven learner builds a separator over the training data. One such classifier (formula) for the data in Table 3 is $\nu_{\mathsf{top}} = \mathsf{u} \implies hd(\mathsf{s}_1, \mathsf{u})$. Substituting similarly learned specifications for the other library functions in Σ_{concat} equips the SMT solver with enough constraints to rule out CEX while maintaining the invariant that the learned specifications are also consistent with the underlying library implementations . Additional iterations of this counterexample-guided refinement loop gather additional positive and negative features, eventually producing the specifications for the library functions presented in Figure 4.

Identifying Spurious Counterexamples. Thus far, we have only considered spurious counterexamples generated by the safety checker. Counterexamples, however, can also result from an incorrect client assertion. For example, suppose the client (unsoundly) asserts :

$$\forall u, mem(\nu, u) \implies mem(\mathsf{s}_2, u)$$

This assertion is wrong, assuming reasonable implementations of top and push, as the elements of the result stack ν can also come from s_1. We distinguish counterexamples corresponding to actual safety violations by first checking if all the feature vectors extracted from the counterexample are included in the set of known positive feature vectors. For example, given this unsound assertion, the verifier may produce the following counterexample:

$$\begin{array}{c} \forall l\ u, (hd(l, u) \iff ((l = \nu \lor l = \mathsf{s}_1) \land u = \mathsf{a})) \land \nu_{\mathsf{top}} = \mathsf{a} \land \\ (mem(l, u) \iff ((l = \nu \lor l = \mathsf{s}_1) \land u = \mathsf{a})) \end{array} \qquad \text{(CEX')}$$

Table 4 shows all of the feature vectors for $\mathsf{R}_{\mathsf{top}}$ that are extracted from this counterexample . Since these are a subset of the positive feature vectors from Table 2, there are no feature vectors that can be labeled as negative, and there are thus no new bad behaviors that the learner can use to generate a refined specification mapping that rejects the counterexample. In this scenario, our algorithm tries to discover concrete values of s_1 and s_2 consistent with CEX'; that is, s_1 only

Table 4. Feature vectors for R_{top} extracted from CEX'.

$R_{top}(s_1, \nu_{top})$	$\forall u$	$hd(s_1, u)$	$mem(s_1, u)$	$\nu_{top} = u$
	$u = a$	true	true	true
	$u \neq a$	false	false	false

Table 5. A weakening feature vector of R_{push} in Figure 4

$R_{push}(\nu_{top}, \nu_{concat}, \nu)$	$\forall u$	$hd(\nu_{concat}, u)$	$mem(\nu_{concat}, u)$	$hd(\nu, u)$	$mem(\nu, u)$	$\nu_{top} = u$
	$u = 1$	true	true	true	true	true

contains a and s_2 is empty. When called with these parameters, concat([a], []) will return $\nu \equiv [a]$, which Elrond returns as a witness of an unsafe execution.

Note that this situation may also occur when the feature set is not large enough, as the specifications in the corresponding hypothesis space are not expressive enough to identify a spurious counterexample. Thus, if we are not able to find inputs that trigger a safety violation, we grow the feature set by increasing the number of quantified variables (*e.g.* from u to u, v) so that Elrond can explore a richer space of specifications.

Weakening. While the above strategy is guaranteed to find a safe and consistent verification interface when one exists, the solutions it produces may still be suboptimal, as illustrated in Figure 5. For example, the first conjunct of the specification for `push` in Figure 4 states that any existing member of both the input and output stacks should not be the same as the element being added to the stack; that is, `push` always produces a stack with no duplicates. This specification is too restrictive, however, as it disallows reasonable behaviors such as `push([1;2], 1) = [1;1;2]`. However, if our sampler never generates an observation corresponding to this behavior, e.g. $x \equiv 1$, $s \equiv [1; 2]$ and $\nu_{push} \equiv [1; 1; 2]$, the candidate specification for `top` produced by Elrond's first phase will incorrectly disallow it.

In other words, our reliance on testing to identify and label negative feature vectors may result in initial specifications that are overfitted to the examples enumerated by the test generator. There are two potential reasons such a positive example 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. Elrond's weakening phase helps ameliorate both issues.

Our weakening algorithm iteratively weakens candidate specifications, focusing on one library function at a time. To weaken the specification of `push`, for example, we fix the specifications of the other library functions to their assignments in the current verification interface, and then try to find a maximal weakening of R_{push} that admits a larger set of implementations of `push`. To do so, Elrond attempts to discover additional *weakening feature vectors* for R_{push}, or feature vectors corresponding to behaviors disallowed by the current specification but which would not lead to a violation of client safety. One possible weakening feature vector for our current example is shown in Table 5. Here, the head of both s_1 and the result of the recursive call is 1; this scenario is mistakenly disallowed by the specification of `push` in Figure 4.

Elrond repeatedly queries the verifier to identify weakening feature vectors for push, indicating that the current specification is maximal when none can be found. It then moves on to the next function specification, iteratively weakening each until a fixpoint is reached. Figure 6 shows the maximal verification interface Elrond builds by weakening the candidate specifications in Figure 4 . Compared with Figure 4, the specification of push now permits duplicate elements in stacks. In addition, the specification of is_empty has been simplified by removing the redundant conjunction $\neg\nu \wedge hd(\mathsf{s}, u) \implies mem(\mathsf{s}, u)$, as $hd(\mathsf{s}, u) \implies mem(\mathsf{s}, u)$ can never be violated by a concrete stack value thanks to the observed semantics of *hd* and *mem*.

$$
\begin{aligned}
&\mathsf{R}_{\mathsf{is\_empty}}(\mathsf{s}, \nu) \mapsto \forall u, (\nu \implies \neg mem(\mathsf{s}, u))\\
&\mathsf{R}_{\mathsf{top}}(\mathsf{s}, \nu) \mapsto \forall u, \nu = u \iff (hd(\mathsf{s}, u) \wedge mem(\mathsf{s}, u))\\
&\mathsf{R}_{\mathsf{tail}}(\mathsf{s}, \nu) \mapsto \forall u, (mem(\mathsf{s}, u) \implies (mem(\nu, u) \vee hd(\mathsf{s}, u)))\wedge\\
&\qquad\qquad\qquad ((mem(\nu, u) \vee (hd(\nu, u) \wedge hd(\mathsf{s}, u))) \implies mem(\mathsf{s}, u))\\
&\mathsf{R}_{\mathsf{push}}(\mathsf{x}, \mathsf{s}, \nu) \mapsto \forall u, (mem(\nu, u) \wedge \neg \mathsf{x} = u \implies mem(\mathsf{s}, u) \wedge \neg hd(\nu, u))\wedge\\
&\qquad\qquad\qquad (\mathsf{x} = u \vee mem(\mathsf{s}, u) \implies mem(\nu, u)) \wedge (\neg mem(\nu, u) \wedge hd(\nu, u) \implies hd(\mathsf{s}, u))
\end{aligned}
$$

Fig. 6. Maximal specifications for Stack operations. The special variable ν represents the result of the method.

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 providing 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 (Σ, Φ) to refer to the *verification query* whose validity we are attempting to establish. These structures serve the same role as verification conditions in a typical verification framework. The first component of this query, Σ, 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, Φ, represents the client program's pre- and post-conditions, encoded as sentences built from logical connectives ($\wedge, \vee, \implies$) over prenex universally-quantified propositional formulae. Each verification query corresponds to a control flow path in the client program; the full algorithm considers the conjunction of all these verification queries at once. To keep the formalization and the description of our algorithms concise, our description considers a single verification query in isolation. The extension to sets of verification queries is provided in the full version of the paper [Zhou et al. 2021a] .

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

$$
\Sigma \equiv (\bigwedge_{f \in F, i=0}^{n} R_f(\vec{x_i}) \wedge \bigwedge_{x, x' \in \cup_{0 \le j \le i} \vec{x_j} \cup C} x = x')
$$

Here, the equality constraints are either between program variables (x, x') or between variables and constants C of some base type (e.g., Booleans and integers). Each $R_f(\vec{x_i})$ in Σ is an application of a placeholder predicate to some arguments; the conjunction of these 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 blackbox 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 set of method predicates and F be the set of functions in a library used by the client. Let Γ_f be a partial function from the domain of $f \in F$ to its codomain, and Γ_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 (Σ, Φ) is the verification query extracted from the client.

Example. The specification configuration of our running example consists of a verification query $(\Sigma_{concat}, \Phi_{\mathsf{concat}})$, a method predicate set $\{memhd\}$, and library functions $\{\mathsf{push}, \mathsf{is\_empty}, \mathsf{top}, \mathsf{tail}\}$. The partial functions in Γ_F and Γ_P abstract over observations on their corresponding blackbox implementations; for an execution that produces the feature vectors shown in Table 2, they are:

$$
\begin{aligned}
\Gamma_{mem} &\equiv \lambda(l \in \{\mathsf{s}_1, \mathsf{s}_2, v_{\mathsf{tail}}, v_{\mathsf{concat}}, v\}), u. \\
&\quad (u = \mathsf{a} \land (l = \mathsf{s}_1 \lor l = v)) \lor (u = \mathsf{b} \land (l = \mathsf{s}_2 \lor l = v_{\mathsf{concat}} \lor l = v)) \\
\Gamma_{hd} &\equiv \lambda(l \in \{\mathsf{s}_1, \mathsf{s}_2, v_{\mathsf{tail}}, v_{\mathsf{concat}}, v\}), u. \\
&\quad (u = \mathsf{a} \land (l = \mathsf{s}_1 \lor l = v)) \lor (u = \mathsf{b} \land (l = \mathsf{s}_2 \lor l = v_{\mathsf{concat}})) \\
\Gamma_{\mathsf{push}} &\equiv \lambda x, (l \in \{v_{\mathsf{concat}})\}).v \\
\Gamma_{\mathsf{is\_empty}} &\equiv \lambda(l \in \{\mathsf{s}_1\}), \mathit{false} \\
\Gamma_{\mathsf{top}} &\equiv \lambda(l \in \{\mathsf{s}_1\}), \mathsf{a} \\
\Gamma_{\mathsf{tail}} &\equiv \lambda(l \in \{\mathsf{s}_1\}), v_{\mathsf{tail}}
\end{aligned}
$$

where stack arguments are limited to values in this particular execution, e.g. s_1 or v_{tail}.

Given a specification configuration as input, the output of our verification pipeline is a *verification interface* (Δ), a logical interpretation of the method predicates that maps each placeholder predicate for a function $f \in F$ to a universally-quantified propositional formula over the parameters and result of f. We impose two requirements on Δ. 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 Δ, and $\Sigma \implies \Phi$ is the verification query built from a client program:

Definition 3.3 (Safe Verification Interface). For a given verification query (Σ, Φ), a verification interface Δ is *safe* when:

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

In addition to safety, we also desire that any proposed mapping Δ be *consistent* with the provided implementations of method predicates and library functions, *i.e.* that Δ must accurately represent their observed behavior. Formally:

Definition 3.4 (Interface Consistency). A verification interface Δ is consistent with Γ_P and Γ_F when all specifications in Δ are consistent with the inputs on which Γ_P and Γ_F are defined. Formally,

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

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

This definition thus relates the observed behavior of a library method on test data, encoded by Γ_P and Γ_F with its logical characterization provided by Δ. 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 (Interface Order $\succ$). The verification interface Δ' is weaker ($\succ$) than Δ when,

(1) The two interfaces contain the same functions: $dom(\Delta) = dom(\Delta')$
(2) They are not equal: $\exists R_f \in dom(\Delta), \Delta(R_f) \not\Longleftrightarrow \Delta'(R_f)$
(3) The specifications in Δ' are logically weaker that those in Δ: $\forall R_f \in dom(\Delta), \Delta(R_f) \implies \Delta'(R_f)$

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 Verification Interface). For a specification configuration $((\Sigma, \Phi), P, F, \Gamma_P, \Gamma_F)$, Δ is a maximal verification interface when:

(1) Δ is safe for the verification query (Σ, Φ).
(2) Δ is consistent with Γ_P and Γ_F.
(3) For a given bound on the number of quantified variables k used by the specifications in Δ, there is no safe and consistent interface Δ' whose specifications use at most k quantified variables such that $\Delta' \succ \Delta$.

We now refine our expectation for the output of our verification pipeline to be not just any safe and consistent verification interface, but also a *maximal* one. Notice that our notion of maximality is parameterized by the number of quantified variables used 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 build a *classifier* (a function from unlabeled data to a label) from a set of labeled data. 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 for a set of variables $\vec{x}$ is a method predicate 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 $\mathcal{S} \equiv FSet(P, f(\vec{\alpha}_f) = v_f, \vec{u})$, is a list of all well-typed features in P for the set of variables $\vec{\alpha}_f \cup \{v_f\} \cup \vec{u}$ which is *minimally linearly independent*:

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

Example. The feature set for the function top : $list_a \rightarrow a$ from the `Stack` library, where a is some base type, for predicate set $P \equiv \{hd, mem\}$, equality operation $=_a$ and quantified variables $\vec{u} \equiv \{u : a\}$ is $FSet(P, \mathrm{top}(l) = v, \vec{u}) \equiv [hd(l, u),\ mem(l, u),\ v{=_a}u]$. 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 $hd(l, u) \wedge v{=_a}u$ and is thus not linearly independent with respect to 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 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. This limitation was expressive enough for all of our benchmarks.[1]

Definition 4.4 (Hypothesis Space). The *hypothesis space* of specifications 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 $FSet(P, f(\vec{\alpha}) = v, \vec{u})$, the logical connectives $\{\wedge, \vee, \neg, \implies\}$, and Boolean constants $\top$ (TRUE) and $\bot$ (FALSE). The hypothesis space of f over P and $\vec{u}$ is denoted $Hyp(P, f(\vec{\alpha}) = v, \vec{u})$.

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 a feature set $\mathcal{S} \equiv FSet(P, f(\alpha) = v, \vec{u})$, the *logical embedding* of fv is a formula encoding the assignment to its features:

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

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

Example. Given the classifier $\phi \equiv \forall u, hd(\mathsf{s}_1, \mathsf{u}) \implies v_{\mathrm{top}} = \mathsf{u}$ for the top function from Section 2, the first row in Table 3 corresponds to the feature vector $fv^- \equiv \{hd(\mathsf{s}_1, \mathsf{u}) \mapsto \mathsf{false};\ mem(\mathsf{s}_1, \mathsf{u}) \mapsto \mathsf{false};\ v_{\mathrm{top}} = \mathsf{u} \mapsto \mathsf{true}\}$. The unitary classifier for fv is $[\![fv^-]\!] \equiv \forall u, \neg hd(\mathsf{s}_1, \mathsf{u}) \wedge \neg mem(\mathsf{s}_1, \mathsf{u}) \wedge v_{\mathrm{top}} = \mathsf{u}$. fv is labelled negative by ϕ, as $[\![fv^-]\!] \not\Longrightarrow \phi$. The other two feature vectors in Table 3 are labeled as positive by ϕ.

[1] 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 it: $(\forall u, \exists v, mem(l, u) \implies (mem(l, v) \wedge ord(l, u, v) \wedge u \leq v))$.

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

$$\begin{aligned}\phi^+ \equiv &\quad \{fv \in 2^{\mathcal{S}} \mid [\![fv]\!] \implies \phi\}\\ \phi^- \equiv &\quad \{fv \in 2^{\mathcal{S}} \mid [\![fv]\!] \not\Longrightarrow \phi\}\end{aligned}$$

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

For a particular configuration, we can straightforwardly lift this partitioning to verification interfaces:

$$\begin{aligned}\Delta(R_f)(\vec{\alpha}, \nu)^+ \equiv &\quad \{fv \in 2^{FSet(P, f(\vec{\alpha})=\nu, \vec{u})} \mid [\![fv]\!] \implies \Delta(R_f)(\vec{\alpha}, \nu)\}\\ \Delta(R_f)(\vec{\alpha}, \nu)^- \equiv &\quad \{fv \in 2^{FSet(P, f(\vec{\alpha})=\nu, \vec{u})} \mid [\![fv]\!] \not\Longrightarrow \Delta(R_f)(\vec{\alpha}, \nu)\}\end{aligned}$$

4.1 Learning Safe and Consistent Verification Interfaces

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 extract feature vectors from a set of *logical samples*:

Definition 4.7 (Sample). A *sample* s of a formula ϕ is an instantiation of its quantified variables and a Boolean-valued interpretation for each application of a method predicate to those variables in ϕ, which we denote as $s \models \phi$. The positive and negative samples of a verification query (Σ, Φ) are samples of Φ 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 Φ_{concat}[2]:

$$\begin{aligned}&\{\mathsf{s}_1 \mapsto \mathtt{l}_0; \mathsf{s}_2 \mapsto \mathtt{l}_1; \nu \mapsto \mathtt{l}_2; \nu_{\mathsf{top}} \mapsto \mathsf{a}; \nu_{\mathsf{tail}} \mapsto \mathtt{l}_3; \nu_{\mathsf{concat}} \mapsto \mathtt{l}_4; \nu_{\mathsf{is\_empty}} \mapsto \bot;\\ &\ hd(l,u) \equiv \{\}; mem(l,u) \equiv \{(\mathtt{l}_2, \mathsf{a}); (\mathtt{l}_3, \mathsf{a}); (\mathtt{l}_4, \mathsf{a})\}\}\end{aligned} \qquad (s^-)$$

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

$$\begin{aligned}&\{\mathsf{s}_1 \mapsto \mathtt{l}_0; \mathsf{s}_2 \mapsto \mathtt{l}_1; \nu \mapsto \mathtt{l}_2; \nu_{\mathsf{top}} \mapsto \mathsf{a}; \nu_{\mathsf{tail}} \mapsto \mathtt{l}_3; \nu_{\mathsf{concat}} \mapsto \mathtt{l}_4; \nu_{\mathsf{is\_empty}} \mapsto \bot;\\ &\ hd(l,u) \equiv \{(\mathtt{l}_0, \mathsf{a}); (\mathtt{l}_1, \mathsf{b}); (\mathtt{l}_4, \mathsf{b}); (\mathtt{l}_2, \mathsf{a})\}; mem(l,u) \equiv \{(\mathtt{l}_0, \mathsf{a}); (\mathtt{l}_1, \mathsf{b}); (\mathtt{l}_4, \mathsf{b}); (\mathtt{l}_2, \mathsf{a}); (\mathtt{l}_2, \mathsf{b})\}\}\end{aligned} \qquad (s^+)$$

Although they come from different sources, both samples provide the values of variables (e.g., the value of $\nu_{\mathsf{is\_empty}}$ in both s^- and s^+ is $\bot$) and the values of predicate applications (e.g., $hd(\nu_{\mathsf{concat}}, \nu_{\mathsf{top}})$ is true in s^- sample and false in s^+).

Using $[\![\cdot]\!]$, we can *extract* a collection of feature vectors under a feature set $\mathcal{S}$ from a sample s:

$$\chi_{\mathcal{S}}(s) \equiv \{fv \in 2^{\mathcal{S}} \mid s \models [\![fv]\!]\}$$

For example, the feature vectors extracted from s^- and s^+ for the feature set of top are shown in the second and third rows of Table 1 and Table 2, respectively.

Definition 4.8 (Classifier Consistency). For a verification query (Σ, Φ), we say that a verification interface Δ is *consistent* with a negative sample s^- if *at least one* of the library specifications in Δ classifies *one or more* features extracted from that sample as negative:

$$\exists R_f(\vec{\alpha}, \nu) \in \Sigma,\ \exists fv \in \chi_{FSet(P, f(\vec{\alpha})=\nu, \vec{u})}(s^-), fv \in \Delta(R_f)(\vec{a}, \nu)^-$$

[2]The interpretation of method predicates are represented as binary relations.

Similarly, Δ is consistent with a positive sample s^+ if *all* specifications in Δ positively identify *every* feature vector extracted from s^+:

$$\forall R_f(\vec{\alpha}, \nu) \in \Sigma,\ \forall fv \in \chi_{FSet(P, f(\vec{\alpha})=\nu, \vec{u})}(s^+), fv \in \Delta(R_f)(\vec{a}, \nu)^+$$

Example. The verification interface Δ from Figure 4 is consistent with (s^-), as the specification of the top function labels as negative the following feature vector of s^-, $fv^- \equiv \{hd(\mathsf{s}_1, \mathsf{u}) \mapsto \mathsf{false}; mem(\mathsf{s}_1, \mathsf{u}) \mapsto \mathsf{false}; \nu_{\mathsf{top}} = \mathsf{u} \mapsto \mathsf{true}\}$ to negative. Furthermore, Δ 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* Δ, $\Sigma[\Delta] \implies \Phi$ *is valid iff* Δ *is consistent with all negative samples* s^-*;* Δ *is a consistent interface iff* Δ *is consistent with all positive samples* s^+ *entailed by* Γ_F *and* Γ_P*.* [3][4]

4.2 Learning Maximal Verification Interfaces

While Theorem 4.9 identifies the conditions under which a verification interface is safe and consistent, it does not ensure that it is maximal. We frame the search for a maximal solution as a learning problem for a single function specification assuming all other specifications are fixed .

Definition 4.10 (Weakest safe specification). For a given verification query (Σ, Φ) and safe and consistent verification interface Δ, ϕ is the weakest safe specification of f iff

(1) $\Delta[R_f \mapsto \phi]$ is safe
(2) For a given bound on the number of quantified variables k allowed in the specification of f, there is no other specification ϕ' with k quantified variables that makes $\Delta[R_f \mapsto \phi']$ safe such that $\phi \implies \phi'$.

Definition 4.11 (Sample with respect to library function). For a verification query (Σ, Φ), and safe verification interface Δ, 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 Δ:

$$s \models \Sigma[\Delta[R_f \mapsto \top]] \land \Phi$$
$$s \models \Sigma[\Delta[R_f \mapsto \top]] \land \neg\Phi \quad \text{(resp.)}$$

Ideally, the weakest safe specification of f would be able to positively classify every such positive sample. Because this is not possible in general due to the intrinsic granularity of the hypothesis space, 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 4.12 (Weakening feature vector and samples). For a verification query (Σ, Φ) and safe verification interface Δ, a weakening feature vector fv distinguishes between a weaker safe specification and $\Delta(R_f)$:

$$fv \in \Delta(R_f)^- \text{ and } \Sigma[\Delta[R_f \mapsto [\![fv]\!] \lor \Delta(R_f)]] \implies \Phi$$

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.

[3] All positive samples s^+ entailed by Γ_F and Γ_P means every positive sample consistent with the observations encoded by Γ_F and Γ_P.
[4] Proofs for all theorems are available in the full version of the paper [Zhou et al. 2021a] .

Algorithm 1: Multi-Abductive Inference Algorithm.

Inputs : Specification configuration $((\Sigma, \Phi), F, P, \Gamma_F, \Gamma_P)$
Output : Maximal verification interface Δ or counterexample Cex

1 $\vec{u} \leftarrow \emptyset$;
2 **while** *true* **do**
3   **match** $\mathit{SpecInfer}((\Sigma, \Phi), F, P, \Gamma_F, \Gamma_P, \vec{u})$**:**
4   **case** $\mathit{Fail}\ s^-$ **do**
5   **match** $\mathit{ExtractCex}(s^-)$**:**
6   **case** *Cex* **do** **return** *Cex* ;
7   **case** *None* **do** $\vec{u} \leftarrow \vec{u} \cup \mathit{FreshVariable}()$;
8   **case** *Fail None* **do** $\vec{u} \leftarrow \vec{u} \cup \mathit{FreshVariable}()$;
9   **case** Δ **do**
10   **repeat**
11         $\Delta_0 \leftarrow \Delta$;
12   **for** $f \in F$ **do** $\Delta(R_f) \leftarrow \mathit{Weaken}((\Sigma, \Phi), P, \Delta, f, \vec{u})$;
13   **until** $\Delta = \Delta_0$;
14   **return** Δ;

Example. The sample

$$\{\mathsf{s}_1 \mapsto \mathtt{l}_0; \mathsf{s}_2 \mapsto \mathtt{l}_1; \nu \mapsto \mathtt{l}_2; \nu_{\mathrm{top}} \mapsto \mathsf{a}; \nu_{\mathrm{tail}} \mapsto \mathtt{l}_3; \nu_{\mathrm{concat}} \mapsto \mathtt{l}_4; \nu_{\mathrm{is\_empty}} \mapsto \bot;$$
$$hd(l, u) \equiv \{(\mathtt{l}_0, \mathsf{a}); (\mathtt{l}_1, \mathsf{a}); (\mathtt{l}_2, \mathsf{a}); (\mathtt{l}_4, \mathsf{a})\}; mem(l, u) \equiv \{((\mathtt{l}_0, \mathsf{a}); (\mathtt{l}_1, \mathsf{a}); (\mathtt{l}_2, \mathsf{a}); (\mathtt{l}_4, \mathsf{a})\}\}$$

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 Φ_{concat}. It is also consistent with all the specifications in the verification interface from Figure 4 outside of the one for push. The feature vector fv shown in Table 5 is extracted from this sample, and it is not included in $\Delta(R_{\mathrm{push}})^+$. Moreover, $\Sigma[\Delta[R_{\mathrm{push}} \mapsto [\![fv]\!] \vee \Delta(R_{\mathrm{push}})]] \implies \Phi$, thus fv is a weakening feature vector for push, and the above sample is a weakening sample for push.

Theorem 4.13. *For a given specification configuration* $((\Sigma, \Phi), P, F, \Gamma_P, \Gamma_F)$ *and safe verification interface* Δ, $\Delta(R_f)$ *is weakest safe specification of* f *if and only if there are no weakening samples for* f.

5 ALGORITHM

Algorithm 1 presents the abductive inference procedure depicted in Figure 1. The algorithm maintains a list of variables $\vec{u}$ that defines the hypothesis space it is currently exploring. The main loop of the algorithm searches for a maximal verification interface in this hypothesis space, starting from an initially empty set of variables (line 1). We first try to use the *SpecInfer* subalgorithm (Algorithm 2) to infer a verification interface that is safe and consistent with the specification configuration. If inference fails with a sample, we search for corresponding inputs that trigger a safety violation using random testing via the *ExtractCex* subroutine (line 8). If we are unable to find such unsafe inputs, we extend $\vec{u}$ with a fresh variable (lines 7-8) and restart the loop under this enhanced hypothesis space. Otherwise, *SpecInfer* has found a safe and consistent verification interface, Δ, which we then refine to a maximal solution via a weakening loop. This loop iteratively weakens the specification of each library function in Δ using the *Weaken* subalgorithm (Algorithm 3), and returns Δ once it has reached a fixed-point.

5.1 Specification Inference

Algorithm 2: Safe and Consistent Specification Inference (*SpecInfer*)

Inputs : Specification configuration $(\Sigma, \Phi), F, P, \Gamma_F, \Gamma_P$, variables in hypothesis space $\vec{u}$
Output : Consistent safe verification interface or reports failure

1 **for** $f \in F$ **do** $\pi_f, \omega_f \leftarrow \emptyset, \emptyset$;
2 $\Delta \leftarrow \{R_f \mapsto \forall \vec{u}, \top\}$;
3 **while** *true* **do**
4   **while** *true* **do**
5   **match** $\mathit{Sample}(\Delta(R_f), \Gamma_F, \Gamma_P)$**:**
6   **case** *None* **do break**;
7   **case** *Some* s^+ **do**
8   **for** $f \in F$ **do**
9                     $\pi_f \leftarrow \mathit{FvecFromSample}(\mathit{FSet}(P, f(\vec{\alpha}_f) = \nu_f, \vec{u}), s^+) \cup \pi_f$;
10                     $\omega_f \leftarrow \omega_f \setminus \pi_f$
11   **for** $f \in F$ **do** $\Delta(R_f) \leftarrow \mathit{Learner}(\pi_f, \omega_f)$;
12   **match** $\mathit{Verify}((\Sigma[\Delta] \implies \Phi))$**:**
13   **case** *OK* **do**
14   **match** $\mathit{Verify}(\neg\Sigma[\Delta])$**:**
15   **case** *OK* **do return** *Fail None* ;
16   **case** *Sat* _ **do return** Δ;
17   **case** *Sat* s^- **do**
18   **for** $f \in F$ **do** $\omega'_f \leftarrow \mathit{FVecFromSample}(\mathit{FSet}(P, f(\vec{\alpha}_f) = \nu_f, \vec{u}), s^-) \setminus \pi_f$;
19   **if** $\bigwedge_{f \in F} \omega'_f = \emptyset$ **then return** *Fail* s^- ;
20   **else for** $f \in F$ **do** $\omega_f \leftarrow \omega_f \cup \omega'_f$;
21   **for** $f \in F$ **do** $\Delta(R_f) \leftarrow \mathit{Learner}(\pi_f, \omega_f)$;

Our data-driven multi-abduction specification inference algorithm (*SpecInfer*) given by Algorithm 2 assumes three key components: a property-based random sampler *Sample*, a satisfiability checker *Verify*, and a classifier learner *Learner*. *Sample* takes a first-order formula as input and attempts to find a counterexample via random sampling, returning data instances which violate this specification based on these counterexamples. *Verify* returns OK when given a valid formula, and produces an assignment of variables and member predicates (i.e. a sample) that invalidates its input formula otherwise. *Learner* takes two disjoint sets of feature vectors, π and ω, as inputs, and returns a formula that classifies elements of π and ω as positive and negative, respectively. Elrond uses a decision tree algorithm [Ruggieri 2002] as the underlying learning framework.

In more detail, for each library function f, *SpecInfer* maintains a pair of positive and negative sets of feature vectors π_f and ω_f; these are initially empty (line 1). The algorithm holds the current candidate verification interface in the variable Δ, which initially stores trivial specifications for each library function. *SpecInfer* implements a two-step refinement loop which first calls *Sample* to test the current candidate verification interface Δ against Γ_F and Γ_P in order to identify any library function behaviors that are inconsistent with their current specifications (line 5). If so, *Sample* returns a positive sample s^+ that we then use to augment the set of positive feature vectors (line 7). We subsequently remove any feature vectors appearing in π_f from ω_f in order to ensure the

Algorithm 3: Weakening (*Weaken*)

Inputs : $(\Sigma, \Phi), P, \Delta, f, \vec{u}$

Output : Weakest safe specification of f

$W_f \leftarrow \forall \vec{u}, \bot;\ \pi, \omega \leftarrow \emptyset, \emptyset$;

while *true* **do**

    match $\mathit{Verify}(\Sigma[\Delta[R_f \mapsto \forall \vec{u}, \top]] \land \Phi \implies \Sigma[\Delta[R_f \mapsto \Delta(R_f) \lor W_f \lor [\![\omega]\!]]])$**:**

        case *OK* **do**

            if $W_f = \forall \vec{u}, \bot$ **then return** $\Delta(R_f)$ **else return** $W_f \lor \Delta(R_f)$;

        case *Sat s* **do**

            $W_f, \pi, \omega \leftarrow \mathit{Update}\ (s, W_f, \pi, \omega)$;

            $W_f, \pi, \omega \leftarrow \mathit{SafetyLoop}\ (W_f, \pi, \omega)$;

Procedure $\mathit{Update}(s, W_f, \pi, \omega)$

    for $fv \in \mathit{FVecFromModel}(\mathit{Fset}(P, f(\alpha_f) = v_f, \vec{u}),\ s)$ **do**

        match $\mathit{Verify}((\Sigma[\Delta[R_f \mapsto W_f \lor \Delta(R_f) \lor [\![fv]\!]]] \implies \Phi))$**:**

            case *OK* **do** $\pi \leftarrow \pi \cup \{fv\}$;

            case *Sat* _ **do** $\omega \leftarrow \omega \cup \{fv\}$;

        $W_f \leftarrow \mathit{Learner}(\pi, \omega)$

    return W_f, π, ω

Procedure $\mathit{SafetyLoop}(W_f, \pi, \omega)$

    match $\mathit{Verify}((\Sigma[\Delta[R_f \mapsto W_f \lor \Delta(R_f)]] \implies \Phi))$**:**

        case *OK* **do** **return** W_f, π, ω ;

        case *Sat s* **do**

            $W_f, \pi, \omega \leftarrow \mathit{Update}(s, W_f, \pi, \omega)$;

            return $\mathit{SafetyLoop}(W_f, \pi, \omega)$

two sets are disjoint before invoking *Learner* to ascertain whether Δ is consistent with the newly observed behaviors (lines 9-11).

SpecInfer then queries *Verify* to determine if the current verification interface is sufficient to ensure the desired safety property (line 12). We then verify that the specifications are not contradictory (line 14) before returning them, in order to ensure that they are not vacuously safe. Alternatively, if the verifier returns a sample s^- witnessing an unsafe execution that Δ was not strong enough to disallow, we extract sets of feature vectors for each library function f from s^-, filtering out any that also occur in π_f (line 18) to maintain the invariant that π_f and ω_f are disjoint. If no new negative feature vectors are found, *SpecInfer* cannot ensure that s^- is spurious in the current hypothesis space, and it thus fails, returning Fail s^- to the main algorithm. Otherwise, the algorithm updates ω_f with the new negative feature vectors and strengthens Δ to rule out s^- by invoking *Learner* and the refinement loop restarts (line 21).

Theorem 5.1 (SpecInfer is Total and Sound). *Algorithm 2 always halts and returns a verification interface that is safe and consistent.*

5.2 Weakening

Algorithm 3 shows the *Weaken* procedure that is used to generalize the specification of a library function f from the initial solution returned by *SpecInfer*. This procedure takes a verification query, method predicate set, verification interface, library function, and a list of quantified variables as input, and infers a weakest safe specification for f. Notably, the arguments of *Weaken* do not include Γ_P or Γ_P, reflecting that our approach to weakening is purely logical.

Weaken maintains a formula W_f which is used to weaken the specification of f; the start of the function initializes this formula to false (line 1). Like *SpecInfer*, *Weaken* maintains two sets of feature vectors: π, which holds weakening feature vectors, and ω, which holds feature vectors representing library behaviors that cause client safety violations. Both sets are initially empty (line 1). The body of *Weaken* uses a counterexample-guided refinement loop to iteratively construct W_f. The loop first queries the verifier to ensure that every safe execution consistent with the candidate specifications besides $\Delta(R_f)$ is either a) currently allowed by $W_f \vee \Delta(R_f)$ or b) involves a behavior of f that can trigger a safety violation in other contexts (line 3). The clauses of the formula representing each of these pieces are highlighted with the corresponding color. When this query is valid, there is no weakening sample for f; thus, by Theorem 4.13, the current specification for f is the weakest safe one, and *Weaken* returns $W_f \vee \Delta(R_f)$ (or $\Delta(R_f)$ when $W_f \equiv \perp$) (line 5). Alternatively, *Verify* will provide a sample s witnessing a safe execution; this sample is passed to the *Update* subroutine to further weaken W_f. *Update* first uses *Verify* to identify feature vectors extracted from s that trigger safety violations in other contexts. After inserting safe and unsafe feature vectors into π and ω, respectively (lines 12-13), *Update* uses *Learner* to refine W_f. The weakened specification $W_f \vee \Delta(R_f)$ may violate client program safety, however. We use a learning based procedure for its intrinsic generalization ability: when we add a new feature vector to π, the learned classifier will not only classify *that* new feature vector to positive, but also potentially others. This ability results in faster convergence, since we do not need to explicitly enumerate each weakening feature vector. However, to avoid the learned classifier from being weakened to the point that it mistakenly classifies unsafe feature vectors as safe, *Weaken* then invokes *SafetyLoop* to filter any unsafe weakenings (line 8). *SafetyLoop* implements a similar strategy as Algorithm 2 to progressively refine W_f until it guarantees safety. After this step, the loop restarts in order to identify additional weakening possibilities .

Theorem 5.2 (Weaken is Sound). *For given $(\Sigma, \Phi), P, \Delta, f, \vec{u}$, if Δ is safe, Algorithm 3 will halt with the weakest safe specification for f.*

Theorem 5.3 (Relative Completeness). *Algorithm 1 either returns either a maximal verification interface or a concrete counterexample.*

6 IMPLEMENTATION AND EVALUATION

We have implemented a verification pipeline based on the above approach, called Elrond[Zhou et al. 2021b], that targets OCaml programs which rely on libraries to manipulate algebraic data types. Elrond consists 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?

Table 6. 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 ($|\vec{u}|$), the number of counter-examples generated ($|cex|$), and the time in seconds (time_c) needed for Elrond to find a consistent and safe verification interface. Times indicate how long it took for counter-example generation, sampling, feature vector 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 ($|\phi^+|$) 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 $|\phi^+|$ 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.

	$(\lvert F\rvert, \lvert R\rvert)$	$\lvert P\rvert$	$\lvert\vec{u}\rvert$	$\lvert cex\rvert$	time_c(s)	#Gather / $\lvert\phi^+\rvert$	time_w(s)	time_d(ms)
queue	(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
stack	(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			
heap	(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
stream	(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)	3	2	18	1.69	1755 / 31276	741.8	24.3
	(4, 10)	3	2	12	0.37			
set	(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			
trie	(4, 18)	2	1	12	0.47	167 / 404	13.6	28.0
	(4, 18)	2	1	15	0.31			

All reported data was collected on a Linux server 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] All benchmarks, post-conditions, and inferred library specifications from our evaluation are provided in the full version of the paper [Zhou et al. 2021a] .

of abstract data types manipulating a diverse set of algebraic data types, including queues (bankers queues and batched queues), list-based stacks, heaps (leftist heaps and splay heaps), streams, sets (backed by 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. These sorts of shape properties are relatively under-constrained and are thus amenable to property-based random sampling. 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:

$$(\forall u, (root(\mathsf{s}, u) \implies (u < \mathsf{x} \implies root(\nu_{\mathsf{insert}}, u)) \land (u \geq \mathsf{x} \implies root(\nu_{\mathsf{insert}}, \mathsf{x})))$$
$$\land\ (mem(\nu_{\mathsf{insert}}, u) \iff mem(\mathsf{s}, u) \lor \mathsf{x} = u)$$

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

$$\forall u, (mem(\nu, u) \iff mem(\mathtt{l}, u) \lor mem(\mathsf{r}, u) \lor (\mathsf{x} = u)) \land (root(\nu, u) \iff \mathsf{x} = u)$$
$$\land (root(\mathtt{l}, u) \implies mem(\mathtt{l}, u)) \land (root(\mathsf{r}, u) \implies mem(\mathsf{r}, u))$$

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

The detailed results of our experiments are shown in Table 6. The first group of columns in Table 6 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 6 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 counterexamples 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$_c$ column).

The final group of columns in Table 6 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.[6] 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 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.

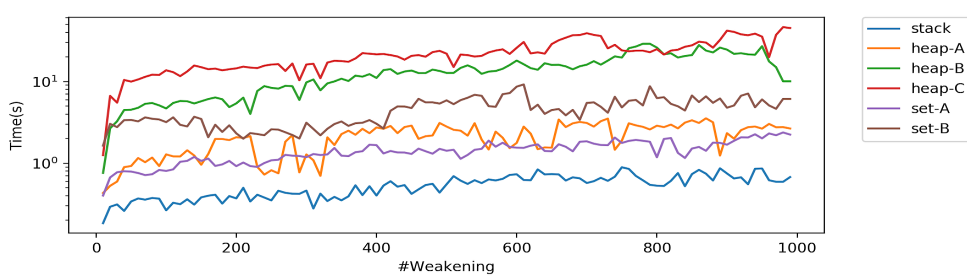

Fig. 7. Run times of the first 1000 weakening iterations in benchmarks.

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 that hit the time bound. The results of this experiment are presented in Figure 7; 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.[7] 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

[6] In 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.

[7] The corresponding Coq developments are provided in the full version of the paper [Zhou et al. 2021a] .

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 that 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 abstract datatype operations. Rather than inferring specifications of libraries in isolation, we consider the complementary problem of discovering those specifications in service of verifying library clients. Broadly speaking, this change in perspective differentiates our approach from these prior works insofar as we cannot appeal to any source of ground truth (e.g, SMT verification as in Padhi et al. [2016] or bounded model checking as in Miltner et al. [2020]). This leads to the need for a weakening procedure to facilitate generalization to ultimately aid client-side verification. However, techniques like Miltner et al. [2020] could be used alongside ours, for example, by leveraging abduced specifications to infer and verify representation invariants.

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. Informally, 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 are 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 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 is 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 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 to aid client-side verification.

While Bastani et al. [2015] consider a similar abductive inference problem – discovering minimal sufficient assumptions in order to analyze client programs when some portion of the code is missing – there are several major differences between our two approaches: 1) their specification domain is limited to CFL reachability (alias and taint specifications), while our hypothesis space involves shape properties (membership, ordering, etc.) over algebraic datatypes; 2) we guarantee specification consistency with respect to our observations on the blackbox implementation of the libraries; and (3) our approach incorporates an explicit weakening procedure to generalize candidate solutions.

Specification Inference. Similar to our problem setup, Nguyen et al. [2014] and Su et al. [2018] infer specifications of library APIs from client information. However, both works try to infer specifications of correct library usage under the assumption that the client is itself correct. In contrast, we leverage abductive methods 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. Another distinguishing feature of our work is our focus on generating maximally weak specifications to overcome overfitted specifications. The work of Pandita et al. [2012] learns specifications from comments in natural language but does not provide safety and consistency guarantees.

Qin et al. [2010] also infers specifications for unknown procedures in a rich domain involving shape properties of datatypes. However, their technique does not query library implementations and does not generalize inferred specifications, which can potentially lead to specifications that are overfitted to the client safety property. 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, we apply an additional algorithmic weakening procedure to refine inferred specifications.

CHC solving. While it is possible to frame our problem as an instance of data-driven CHC solving à la Zhu et al. [2018], this would not address the important challenge of generating maximally weak specifications, a concern that is not considered by Horn-clause semantics [Albarghouthi et al. 2016].

8 CONCLUSIONS

This paper presents a novel data-driven approach to infer specifications using a minimal set of assumptions that are nonetheless consistent with provided blackbox library implementations and sufficient to verify client assertions . We demonstrate that our technique, manifested in a tool called Elrond, is highly effective in identifying sophisticated specifications that enable verification of challenging functional data structure programs.

ACKNOWLEDGEMENTS

We thank Pedro Abreu and the anonymous reviewers for their detailed comments and suggestions. This material is based upon work supported by the NSF under Grants CCF-SHF 1717741, CCF-FMiTF 2019263, and CCF-1755880.

REFERENCES

Aws Albarghouthi, Isil Dillig, and Arie Gurfinkel. 2016. Maximal Specification Synthesis. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (St. Petersburg, FL, USA) *(POPL '16)*. Association for Computing Machinery, New York, NY, USA, 789–801. https://doi.org/10.1145/2837614.2837628

Andrew Appel. 2018. Software Foundations Volume 3: Verified Functional Algorithms.

Osbert Bastani, Saswat Anand, and Alex Aiken. 2015. Specification Inference Using Context-Free Language Reachability. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Mumbai, India) *(POPL '15)*. Association for Computing Machinery, New York, NY, USA, 553–566. https://doi.org/10.1145/2676726.2676977

Koen Claessen and John Hughes. 2000. QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs. In *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming (ICFP '00)*. Association for Computing Machinery, New York, NY, USA, 268–279. https://doi.org/10.1145/351240.351266

Leonardo de Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *Tools and Algorithms for the Construction and Analysis of Systems*, C. R. Ramakrishnan and Jakob Rehof (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 337–340. https://doi.org/10.1007/978-3-540-78800-3_24

Isil Dillig, Thomas Dillig, Boyang Li, and Ken McMillan. 2013. Inductive Invariant Generation via Abductive Inference. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages and Applications* (Indianapolis, Indiana, USA) *(OOPSLA '13)*. Association for Computing Machinery, New York, NY, USA, 443–456. https://doi.org/10.1145/2509136.2509511

Shachar Itzhaky, Anindya Banerjee, Neil Immerman, Ori Lahav, Aleksandar Nanevski, and Mooly Sagiv. 2014. Modular Reasoning about Heap Paths via Effectively Propositional Formulas. *SIGPLAN Not.* 49, 1 (Jan. 2014), 385–396. https://doi.org/10.1145/2578855.2535854

Shachar Itzhaky, Anindya Banerjee, Neil Immerman, Aleksandar Nanevski, and Mooly Sagiv. 2013. Effectively-Propositional Reasoning about Reachability in Linked Data Structures. In *Proceedings of the 25th International Conference on Computer Aided Verification - Volume 8044* (Saint Petersburg, Russia) *(CAV 2013)*. Springer-Verlag, Berlin, Heidelberg, 756–772. https://doi.org/10.5555/2958031.2958053

Xavier Leroy, Damien Doligez, Alain Frisch, Jacques Garrigue, Didier Rémy, and Jérôme Vouillon. 2014. The OCaml system release 4.02. *Institut National de Recherche en Informatique et en Automatique* 54 (2014).

Anders Miltner, Saswat Padhi, Todd Millstein, and David Walker. 2020. Data-Driven Inference of Representation Invariants. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation* (London, UK) *(PLDI 2020)*. Association for Computing Machinery, New York, NY, USA, 1–15. https://doi.org/10.1145/3385412.3385967

Hoan Anh Nguyen, Robert Dyer, Tien N. Nguyen, and Hridesh Rajan. 2014. Mining Preconditions of APIs in Large-Scale Code Corpus. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering* (Hong Kong, China) *(FSE 2014)*. Association for Computing Machinery, New York, NY, USA, 166–177. https://doi.org/10.1145/2635868.2635924

Chris Okasaki. 1999. *Purely Functional Data Structures.* Cambridge University Press, USA.

Peter-Michael Osera and Steve Zdancewic. 2015. Type-and-Example-Directed Program Synthesis. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Portland, OR, USA) *(PLDI '15)*. Association for Computing Machinery, New York, NY, USA, 619–630. https://doi.org/10.1145/2737924.2738007

Saswat Padhi, Rahul Sharma, and Todd Millstein. 2016. Data-Driven Precondition Inference with Learned Features. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Santa Barbara, CA, USA) *(PLDI '16)*. Association for Computing Machinery, New York, NY, USA, 42–56. https://doi.org/10.1145/2908080.2908099

Rahul Pandita, Xusheng Xiao, Hao Zhong, Tao Xie, Stephen Oney, and Amit Paradkar. 2012. Inferring method specifications from natural language API descriptions. In *2012 34th International Conference on Software Engineering (ICSE)*. 815–825. https://doi.org/10.1109/ICSE.2012.6227137

Benjamin C Pierce, Chris Casinghino, Marco Gaboardi, Michael Greenberg, Cătălin Hriţcu, Vilhelm Sjöberg, and Brent Yorgey. 2010. Software foundations. *Webpage: http://www. cis. upenn. edu/bcpierce/sf/current/index. html* (2010).

Shengchao Qin, Chenguang Luo, Guanhua He, Florin Craciun, and Wei-Ngan Chin. 2010. Verifying Heap-Manipulating Programs with Unknown Procedure Calls. In *Formal Methods and Software Engineering*, Jin Song Dong and Huibiao Zhu (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 171–187. https://doi.org/10.1007/978-3-642-16901-4_13

S. Ruggieri. 2002. Efficient C4.5 [classification algorithm]. *IEEE Transactions on Knowledge and Data Engineering* 14, 2 (2002), 438–444. https://doi.org/10.1109/69.991727

Jingyi Su, Mohd Arafat, and Robert Dyer. 2018. Using Consensus to Automatically Infer Post-Conditions. In *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings* (Gothenburg, Sweden) *(ICSE '18)*. Association for Computing Machinery, New York, NY, USA, 202–203. https://doi.org/10.1145/3183440.3195096

Niki Vazou, Anish Tondwalkar, Vikraman Choudhury, Ryan G. Scott, Ryan R. Newton, Philip Wadler, and Ranjit Jhala. 2017. Refinement Reflection: Complete Verification with SMT. *Proc. ACM Program. Lang.* 2, POPL, Article 53 (Dec. 2017), 31 pages. https://doi.org/10.1145/3158141

Zhe Zhou, Robert Dickerson, Benjamin Delaware, and Suresh Jagannathan. 2021a. Data-Driven Abductive Inference of Library Specifications (Full Version). arXiv:2108.04783 [cs.PL]

Zhe Zhou, Robert Dickerson, Benjamin Delaware, and Suresh Jagannathan. 2021b. OOPSLA2021 Artifact: Data-Driven Abductive Inference of Library Specifications. https://doi.org/10.5281/zenodo.5130646

He Zhu, Stephen Magill, and Suresh Jagannathan. 2018. A Data-Driven CHC Solver. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Philadelphia, PA, USA) *(PLDI 2018)*. Association for Computing Machinery, New York, NY, USA, 707–721. https://doi.org/10.1145/3192366.3192416

He Zhu, Gustavo Petri, and Suresh Jagannathan. 2016. Automatically Learning Shape Specifications. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Santa Barbara, CA, USA) *(PLDI '16)*. Association for Computing Machinery, New York, NY, USA, 491–507. https://doi.org/10.1145/2908080.2908125