Plucking Buggy Inputs Out of Thin Errors: Synthesizing Test Generators via Perturbation Learning

ANONYMOUS AUTHOR(S)

Given a single witness to a fault in a program (in the form of a buggy input), we often wish to discover related inputs that can also trigger the same fault. This kind of error generalization is important to help document API misuse, better localize faults, provide crucial detail in bug reports, and facilitate data-driven program analyses, verification, and inference techniques that require both meaningful positive and negative inputs to a program. Error generalization is particularly challenging, however, when the identified fault occurs in blackbox components whose source code is either unavailable or too complex to understand or effectively analyze. To facilitate error generalization in such contexts, we present a generative learning-based mechanism that synthesizes error-producing test generators for a program under test given one or more known buggy inputs. Our learned test generators are input *perturbations*, functions implemented as sequential compositions of datatype operations that transform one erroneous input into another. These perturbations can be thus used to generate additional error-producing inputs from some initial set of buggy inputs. Our results demonstrate that perturbation learning can effectively and systematically generalize from a small set of known errors in the presence of blackbox components, providing significant benefits to data-driven analysis and verification tools.

1 INTRODUCTION

Consider the following scenario: the client of a program observes some unexpected behavior and reports the issue to the developer, helpfully including the input which triggered the bug. After running the system on this input, and confirming the existence of the bug, the developer observes that the execution trace over the provided input makes calls to external libraries and methods they did not author, and concludes that the root cause of the bug lies within these components and not the program itself. However, if it is not possible to directly access and modify these components or if their complexity makes it difficult to understand how exactly they manifest the bug, diagnosing the root cause of the problem becomes impossible. In this case, providing a *family* of inputs that yield similar errors would help provide a more accurate characterization of the problem that can assist the author of the offending component(s) diagnose and repair the fault.

```
let rec sort s =
  if Stack.length s <= 1 then s else
  let (s1, s2) = split s in
  merge (sort s1) (sort s2)
```

```
let split s =
  let rec helper x y z = ...

let rec merge s1 s2 =
  if Stack.is_empty s1 then s2 else ...
```

Fig. 1. Merge sort over stacks.

To illustrate this problem concretely, consider the example shown in Figure 1. Here the program `sort` tries to sort an input stack. It calls two external helper functions `split` and `merge`, which in turn make calls to a `Stack` library that provides methods like `Stack.length`. Unfortunately, when the developer applies `sort` to the list $[1; 2; 3; 4]$, they are surprised to get the result $[1; 3; 2; 4]$ in response. Due to the program's dependence on these helper functions and library methods, it is not readily apparent exactly how and where the bug occurs when `sort` behaves unexpectedly. In this scenario, additional buggy inputs can help the developer isolate the problem.

Besides aiding fault localization, error generalization is also useful to guide data-driven specification inference and verification tasks. For example, inferring a precondition for `merge` in a blackbox

2022. XXXX-XXXX/2022/4-ART $15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

way [Padhi et al. 2016; Zhu et al. 2016] requires having examples of bugs to help strengthen candidate specifications. Similarly, having additional examples can help abductive inference tools reason about `sort` by guiding the construction of a more precise specifications for `merge` [Zhou et al. 2021].

Property-based testing tools like Quickcheck [Claessen and Hughes 2000] provide a well-studied technique for automatically finding bugs in a program. This approach relies on *data generators* to systematically explore the input space of a program, reporting any errors triggered by those inputs. These tools are usually accompanied by a set of default test generators that sample from a uniform random distribution. They additionally allow users to build their own property-based generators that achieve finer control over the distribution of generated values [Lampropoulos et al. 2017]. Unfortunately, neither the use of default generators nor the ability to write customized ones are likely to work well in the above scenario, where the desired input distribution is tightly coupled to the particular error the developer is trying to debug. Default generators are likely to explore uninteresting regions of the input space unrelated to the bug, while constructing a customized generator requires more information about the nature of the bug, the very problem the developer is trying to solve! Using either approach, the amount of time required to find additional buggy inputs can be impractical. In the worst case, a test generator with an undesirable sampling distribution can try thousands of inputs without triggering any bugs [Lampropoulos et al. 2019], failing to meaningfully grow the family of buggy inputs at all. Intuitively, we would like to explore buggy inputs *similar* to a reported error, but sampling from a uniform distribution does not encapsulate any notion of generating novel inputs "close" to an input known to be of interest. Conversely, customizing an effective test generator requires insights into the root cause of the error which is difficult to ascertain given just a single buggy input.

In order to enable developers to quickly explore a space of related buggy inputs, we propose to automatically *learn* a test generator that is biased towards triggering a particular fault. Figure 2 presents the high-level architecture of Murphy, an automated tool for error generalization we have developed for this purpose. To use Murphy, the user provides:

(1) A blackbox functional program that takes algebraic datatypes like lists, trees, and heaps as input.
(2) Pre- and post-conditions that characterize the expected behavior of the method(s) under test.
(3) *Perturbation operators* that manipulate input datatypes. Intuitively, these operators can be used to describe the "siblings" (semantically-related elements) of a data value. In our example, these operators might include `Stack.head`, `Stack.tail`, and `Stack.snoc`.
(4) One or more initial *buggy inputs*. A buggy input satisfies the method's precondition but causes the method to terminate in a state violating the postcondition. These inputs are given to Murphy's perturbation learning component.
(5) The number of generators to be synthesized for each bug. Each generator is expected to produce distinct buggy inputs.

Using these ingredients, we employ a Markov-Chain Monte-Carlo (MCMC)-based [Hastings 1970] generative learning method to synthesize a set of specialized test generators that collectively produce a (potentially infinite) set of inputs, each of which is guaranteed to cause the method to violate the supplied postcondition and which are all derived from the initial buggy input(s) provided by the user. As we show in Figure 2, the user can add more operators to improve the learned result, or use the generators to sample against the inputs provided to glean insight into program misbehavior. Murphy assumes no access to the method(s) under test (or the libraries that it uses), instead reasoning about program behavior solely by observing its inputs and outputs.

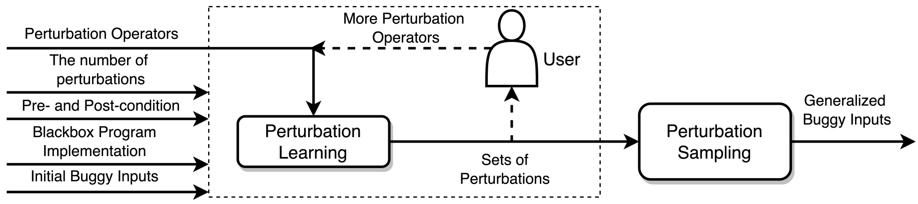

Fig. 2. Murphy pipeline.

Our experimental results show that Murphy is able to synthesize effective and interesting error generators for the applications in our benchmark suite in a few minutes and that it can help improve the efficacy of data-driven specification inference and verification tools that benefit from additional data samples.

Although our methodology might superficially appear to be similar to an example-based program synthesis technique that uses the initial buggy inputs as examples from which to synthesize a test generator, we note some crucial differences. In our context, constructing input-output examples is challenging, requiring a deep understanding of libraries and the functions used by the client program. Moreover, programming-by-example-based synthesis approaches [Gulwani et al. 2017] do not naturally apply since there are very few (perhaps only one) examples to guide synthesis. On the other hand, a typical verification-based program synthesis problem requires users to provide some form of specification that describes the desired result. However, because we do not have a functional specification of the buggy inputs we seek (indeed, that is what we are trying to learn!), synthesis approaches that rely on a verifier like Sketch [Solar-Lezama 2008] will not work out-of-the-box. Instead, our approach leverages a learning-based framework to generalize from a very small set of known buggy inputs using user-provided perturbation operators as a form of inductive bias to guide the learner [Baxter 2000].

In summary, our key contributions are:

- A framing of error generalization as a learning problem in which the learnt generators, which we refer to as *perturbations*, generalize a small set of initial buggy inputs to a *family* of unique buggy inputs, all derivable from this initial set.
- The application of an MCMC-based generative model framework to explore the hypothesis space of potential perturbations.
- A detailed evaluation of our approach using a tool, Murphy, that applies these ideas to a comprehensive set of realistic and challenging functional (OCaml) data structure programs.

The remainder of the paper is structured as follows. We begin with an overview of our solution to the error generalization problem, using a detailed example to motivate its key ideas. A formal characterization of the problem is then given in Section 3. Section 4 describes how a MCMC-based learning strategy can be used to synthesize a desired perturbation for given target programs and initial buggy inputs. 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

To illustrate the problem and our approach in more detail, let us reconsider the buggy `sort` program from the previous section, which was intended to always return a sorted stack. As we saw, however,

```
1 let rec merge s1 s2 =
2   if Stack.is_empty s1 then s2 else
3   if Stack.is_empty s2 then s1 else
4   let (h1, t1) = Stack.pop s1 in
5   let (h2, t2) = Stack.pop s2 in
6   if h1 < h2 then Stack.push h1 (Stack.push h2 (merge t1 t2))
7   else if h1 > h2 then Stack.push h2 (Stack.push h1 (merge t1 t2))
8   else Stack.push h1 (merge t1 t2)
```

Fig. 3. A function that merges two sorted stacks. If either of the input stacks are empty, `merge` returns the other (lines 2 – 3). Otherwise, the function pops the top element off of each stack, and appends both elements in order to the result of merging the tails of the two inputs. (line 6 – 8).

this assumption does not hold when when `sort` is applied to the stack `[1;2;3;4]`:

$$\mathsf{sort}([1;2;3;4]) \equiv [1;3;2;4]$$

Notably, the following call to the external function `merge` yields

$$\mathtt{merge}([1;2],[3;4]) \equiv [1;3;2;4] \qquad (\alpha_{\mathtt{merge}})$$

Now, a developer debugging this error is likely to assume (although they may not be sure) that `merge` should preserve uniqueness and sorted-ness, i.e., `merge` should produce a strictly sorted stack when applied to strictly sorted stacks or, more formally[1]:

$$\mathit{sorted}(\mathtt{s}_1) \wedge \mathit{sorted}(\mathtt{s}_2) \implies \mathit{sorted}(\mathtt{merge}(\mathtt{s}_1,\mathtt{s}_2)) \qquad (\Sigma_{\mathtt{merge}} \implies \Phi_{\mathtt{merge}})$$

Since the resulting list ($[1;3;2;4]$) in the observed call is not sorted, the developer might conclude that `merge` is triggering the fault. It is not clear, though, what specific characteristics of the input are causing the function to behave incorrectly.

Given the implementation of `merge` shown in Figure 3, as well as specifications for the `Stack` methods used by `merge`, the developer would be able to conclude that the implementation of `merge` is incorrect. Indeed, it is possible to define a predicate that precisely captures all the inputs that would trigger a violation of $\Sigma_{\mathtt{merge}} \implies \Phi_{\mathtt{merge}}$, but that nonetheless satisfy the precondition that the input stacks be sorted:

$$\begin{aligned}&\forall(\mathtt{s}_1,\mathtt{s}_2 : \mathtt{SortedStack}), \exists i, 0 \le i \wedge i < |\mathtt{s}_1| \wedge i < |\mathtt{s}_2| \wedge \\ &\qquad ((\mathtt{s}_1[i] > \mathtt{s}_2[i] \implies (i+1 < |\mathtt{s}_2| \wedge \mathtt{s}_1[i] \ge \mathtt{s}_2[i+1])) \vee \\ &\qquad (\mathtt{s}_2[i] > \mathtt{s}_1[i] \implies (i+1 < |\mathtt{s}_1| \wedge \mathtt{s}_2[i] \ge \mathtt{s}_1[i+1])))\end{aligned} \qquad (E_{\mathtt{merge}})$$

This formula precisely captures the salient features of *every* buggy input to `merge`: (a) it must satisfy the precondition of the function, namely the input is a pair of sorted stacks; and either (b) the i^{th} element in the first stack, s_1, is greater than the i^{th} element in the second one, s_2, and also greater than or equal to the element at index $i+1$ in s_2; (c) *or* the i^{th} element in s_2 is greater than i^{th} element in s_1 and also greater than or equal to the element at index $i+1$ in s_1. These conditions certainly hold for our failed input ($\alpha_{\mathtt{merge}}$) since both stacks are sorted and the first element of the second list (3) is greater than the first and second elements of the first list (1 and 2). Automatically deriving this specification of buggy inputs is non-trivial, however. Beyond requiring specifications of the library functions (e.g. `Stack.is_empty` and `Stack.pop`), we also need to establish an inductive invariant that captures the recursive behavior of `merge` in order to relate how elements in the input stacks relate to elements in the output stack. If `merge`'s source code is unavailable, however, it is unclear how to even begin soundly generating such a specification.

[1]We assume the $\mathit{sorted}(\cdot)$ predicate holds for lists whose elements are *strictly* increasing.

To account for these scenarios, we instead treat `merge` as a blackbox program. As E_{merge} suggests, the key *structural* property that triggers an error can be quite complicated, and we have only a single buggy input from which to reason about `merge`'s behavior. One way to expand our knowledge is to identify other inputs to `merge` that also trigger the error. For example, knowing that the pair of input stacks $([2; 4], [6; 8])$ and $([1; 2; 4], [3; 4])$ are also problematic provides stronger evidence that the issue is somehow tied to the pairwise ordering of elements over the two stacks (and not to any specific stack value), providing insight into a potential root cause for the fault. This is exactly the goal of *error generalization*. Our challenge is to devise a mechanism to discover these inputs equipped with only the buggy input α_{merge} and the ability to observe the input-output behavior of `merge`. Since it is hard to pluck additional buggy inputs out of thin air, we propose to instead pluck them from the thin errors we have in hand.

Our key insight is that while directly discovering an intricate property like E_{merge} is difficult, it is comparatively easier to identify input properties that are *not* relevant to the bug. As one simple example, observe that every stack we get by uniformly incrementing the elements of α_{merge} by an arbitrary constant still violates the safety condition since such a transformation preserves the constraints expressed in E_{merge}. We further note that the set of stacks in E_{merge} is *closed* under this transformation: it is impossible to break out of this set of buggy inputs by repeated application of the transformation. This suggests that the bug is not sensitive to this change and that all the elements reachable by this transformation share the same key structural property that caused α_{merge} to trigger the error.

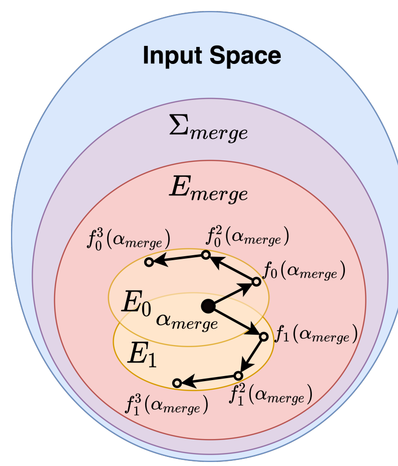

Fig. 4. Sets of buggy inputs generated by *perturbations*.

Figure 4 illustrates this principle. Buggy inputs to `merge` that are similar to the initial buggy input α_{merge} are captured by an implicit property E_{merge} on lists; these buggy inputs must also satisfy Σ_{merge}, the precondition of `merge`. The families of these buggy inputs are represented by the corresponding labeled ovals shown in Figure 4. We can approximate our understanding of E_{merge} by identifying a set that is *closed* under the property "the first input stack has value $i + 1$ at position i while the second input stack is $[3; 4]$" that also contains α_{merge}; this set is shown as the oval labelled E_0 in Figure 4. Since this set is closed, we can also think of it as defining a "neighborhood" around α_{merge} that evinces a bug (and is thus contained in E_{merge}). On the other hand, E_1 might represent the property that "two strictly increasing stacks that have the same length and have prefix $[1; 2]$ and $[3; 4]$, resp."; this set defines a neighborhood different from E_0.

More formally, the *ideal* goal of error generalization is to identify a set of inputs $E = \cup_{1\leq\ i\leq n} E_i$ such that $\forall i, 1 \leq\ i \leq n, \alpha_{\text{merge}} \in E_i$ and E is *maximal* - the set of buggy inputs for `merge` found in any other set is contained within E. For our running example, this means representing E by a function whose domain is a pair of stacks and whose codomain is precisely E_{merge}. However, since we do not have access to E_{merge}, realizing this ideal is problematic. We seek instead to intelligently sample from E_{merge}'s neighborhoods using α_{merge} as an initial guide. Our approach sacrifices completeness (i.e., discovering the actual set representing E_{merge}) for efficiency (i.e., quickly finding a diverse representative collection of E_{merge}'s elements). Our key observation is that any neighborhood around α_{merge} can be compactly represented as an error-preserving transformation or *perturbation*.

Equipped with a perturbation, we can generate a neighborhood of buggy inputs by iteratively applying the perturbation to α_{merge}.

Note that there may be many perturbations that can be constructed from α_{merge}, each of which characterize a different neighborhood. For example, appending a new element to the tail of s_1 in α_{merge} also evinces a safety violation. The input:

$$([1;2],[3;4]) \xrightarrow{\text{append 3 to the end of } \mathsf{s}_1} ([1;2;3],[3;4])$$

produces an output $[1;3;2;4;3]$ that is also not sorted. Perturbations can also be quite rich, e.g., they can depend on both input lists:

$$([1;2;3],[3;4]) \xrightarrow{\text{append the last element of } \mathsf{s}_2 \text{ to the end of } \mathsf{s}_1} ([1;2;3;4],[3;4])$$

The identify function is also a valid, albeit uninteresting, perturbation:

$$([1;2],[3;4]) \xrightarrow{\text{id}} ([1;2],[3;4])$$

By iteratively applying these perturbations, we can generate different neighborhoods of E_{merge}; by combining these sets together, we can approximate the full set of buggy inputs. Given that the space of possible perturbations is quite large, the critical question is how to identify perturbations that, in combination, can generate as many elements of (the unknown) E_{merge} as possible. We now present our solution to this challenge.

2.1 Learning Perturbations

The first challenge to finding desirable perturbations is defining a *hypothesis space* of potential solutions. We need some way of identifying which transformations may be useful when generalizing an error, and we must do so without access to external functions or libraries used by the program we are trying to test. Our solution is to have the user suggest a set of interesting actions to use when building perturbations. In the case of merge, for example, the user may believe that adding an element to the end of a stack is interesting, and so might suggest using `Stack.snoc` to explore the input space. In general, users will supply a set of potentially useful operators for each of the data types in the input domain of the target program. A candidate perturbation is a well-typed, loop-free, sequential composition of suggested operators.

$$\{+0, -1, 0\} \cup \{\mathsf{Stack.cons}, \mathsf{Stack.snoc}, \mathsf{Stack.head}, \mathsf{Stack.last}, \mathsf{Stack.min}, \mathsf{Stack.lower\_bound}, \mathsf{Stack.max}, \mathsf{Stack.upper\_bound}\}$$

Fig. 5. A set of perturbation operators Θ_{merge}.

For example, given the set of operations Θ_{merge} shown in Figure 5, we can build the example perturbation `f0`, which is shown in the left-hand side of Figure 6. The two example perturbations discussed above can be similarly implemented using Θ_{merge}. Iteratively applying `f0` produces the following sequence of buggy inputs:

$$([1;2],[3;4]) \xrightarrow{\mathtt{f0}} ([1;2;3],[3;4]) \xrightarrow{\mathtt{f0}} ([1;2;3;4],[3;4]) \xrightarrow{\mathtt{f0}} ([1;2;3;4;5],[3;4]) \xrightarrow{\mathtt{f0}} \dots$$

We call the set of all inputs reachable from α_{merge} through repeated applications of `f0` its *perturbation closure*. All the inputs in the closure of `f0` satisfy the invariant:

$$|\mathsf{s}_1| \geq 2 \land (\forall i, 0 \leq i < |\mathsf{s}_1| \implies \mathsf{s}_1[i] = i + 1) \land l_2 = [3;4] \qquad (E_0)$$

This is exactly the property E_0 shown in Figure 4. This property is stronger than the actual buggy precondition of merge, i.e. $E_0 \subset E_{\text{merge}}$, as `f0` does not change the second input stack s_2. In contrast,

```
let f0 (s1: Stack.t) (s2: Stack.t) =
    let (e1: int) = Stack.last s1 in
    let (e2: int) = e1 + 1 in
    let (l3: Stack.t) = Stack.snoc s1 e2 in
    (l3, s2)
```

```
let f1 (s1: Stack.t) (s2: Stack.t) =
    let (e1: int) = Stack.upper_bound s2 in
    let (l3: Stack.t) = Stack.snoc e1 s1 in
    let (l4: Stack.t) = Stack.snoc e1 s2 in
    (l3, l4)
```

Fig. 6. Two perturbations that collectively explore an error neighborhood $E_0 \cup E_1$.

the perturbation f1 in the right-hand side of Figure 6 generalizes α_{merge} in the following way:

$$([1;2],[3;4]) \xrightarrow{\texttt{f1}} ([1;2;5],[3;4;5]) \xrightarrow{\texttt{f1}} ([1;2;5;6],[3;4;5;6]) \xrightarrow{\texttt{f1}} \dots$$

By taking the union of the closures of these two perturbations, we have a new closure $E_0 \cup E_1$ that satisfies the following property:

$$\begin{aligned} &|\mathsf{s}_1| \geq 2 \wedge (\forall i, 0 \leq i < |\mathsf{s}_1| - 1 \implies \mathsf{s}_1[i] < \mathsf{s}_1[i+1]) \vee \\ &|\mathsf{s}_2| \geq 2 \wedge (\forall i, 0 \leq i < |\mathsf{s}_2| \implies \mathsf{s}_2[i] = i + 3) \end{aligned} \tag{$E_0 \cup E_1$}$$

which is a better approximation of the optimal generalization of α_{merge} that is defined by E_{merge}. We can thus approximate E_{merge} by collecting the results of iteratively applying f0 and f1 to α_{merge} a bounded number of times: $\bigcup_{0 \leq i \leq n} \texttt{f0}^i(\alpha_{\text{merge}}) \cup \texttt{f1}^i(\alpha_{\text{merge}})$.

Since automatically discovering an optimal error generalization may not always be possible, it is not immediately clear how to judge how close a perturbation is to the optimal solution. Thankfully, there is a natural ranking between potential solutions: we say that a perturbation p_1 is "better" than another p_2 precisely when all the buggy inputs in the closure of p_2 are included in the closure of p_1. Thus, our goal is to find a perturbation whose closure covers as many buggy inputs as possible for a given exploration budget.

Even in the simplest setting that only searches for a single perturbation (e.g., f0), the space of possible (fixed-length) solutions is exponential in the number of perturbation operators, so a naïve enumerate-and-compare strategy is unlikely to scale. Instead, we adopt a search strategy that makes a crucial observation about our hypothesis space: perturbations that are *syntactically* similar are also likely to be *semantically* similar. Put another way: the closures of similar sequences of operations are likely to include similar elements when applied to the same initial buggy inputs. Given a candidate solution, we can see which of its syntactic 'neighbors' improve on it, choose one of the improved perturbations as a new candidate solution, and then recurse. Since there are no guarantees on which of these solutions will lead to the optimal solution, we instead adopt a *sampling* approach to try to explore the space of potential solutions. The challenge, of course, is that we want to do this efficiently, in the absence of any knowledge about the true error region.

Our approach works as follows. Given a set of perturbation operators, a buggy input α, and a bound on the size of perturbations, (1) we randomly choose a perturbation p_1 as our initial solution; (2) we change p_1 slightly to yield a new perturbation p_2; (3) if the number of buggy inputs in p_2's perturbation closure is larger than what in p_1's with respect to α, we continue our search by considering further transformations from p_2; (4) otherwise, we discard p_2 as a viable candidate and consider other perturbations derivable from p_1 by applying other transformations. We repeat this process until we encounter a perturbation that cannot jump to a better one, or we reach a time limit. Since there are potentially many possible transformations for a given perturbation, we induce a distribution over these possibilities. In the simplest case, this distribution would be uniform, but in practice we can employ heuristics that exploit particular semantic features of the perturbation operators comprising the hypothesis space or that are biased towards particular kinds of transformations. For example, we might encourage transformations to favor operators that permute a list (e.g., `List.reverse`) over operators that change the value of a list's elements (e.g., +1).

Framed in this way, our approach can be naturally modeled as an instance of Monte-Carlo Markov-Chain (MCMC) sampling. Each possible transformation (or "jump" in MCMC parlance) produces a new element in a Markov chain whose transition probabilities characterize the likelihood that the jumps that leads from one perturbation to another other are beneficial; a perturbation p' that is derivable from another p using a small number of jumps would thus have a higher transition probability from p than one that is syntactically very dissimilar. For example, assume p_0 in Figure 7 is a good perturbation for the method under test; the immediately adjacent nodes p_1-p_4 are expected to cover a similar set of buggy inputs. The further from p_0 we jump, the less the expected overlap: the perturbations p_{12} and p_{13}, for example, are expected to cover parts of the error region that are more dissimilar from p_0 than p_7. Even starting from these distant nodes, however, our MCMC-method is expected to eventually explore perturbations closer to the center of the figure, until it finally settles on p_0.

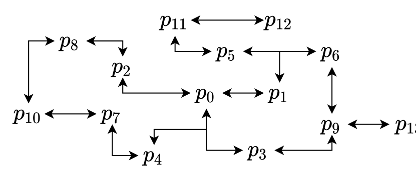

Fig. 7. Sampling the hypothesis space of perturbations as a Markov chain.

In order to illustrate how this idea manifests in practice, we conducted an experiment that tries to find a perturbation for the buggy `merge` function in a hypothesis space consisting of perturbations built from at most four of the perturbation operators shown in Figure 5. We run our search function in parallel to learn three distinct perturbations under a 200 MCMC step-bound, and generate a set of inputs by iterating the learnt perturbations 100 times as an approximation to their closure, and union these closures as the final result. The quality of these results are evaluated by determining the coverage percentage in terms of all feasible buggy inputs this union covers. Our result, shown in Figure 8, demonstrates that even with a relatively small number of steps, MCMC sampling is able to explore 94.4% of all feasible buggy inputs while exploring only a small fraction - $\frac{200\times3}{7.8\times10^5} \approx 0.1\%$ - of the total number of possible perturbations that can be constructed.

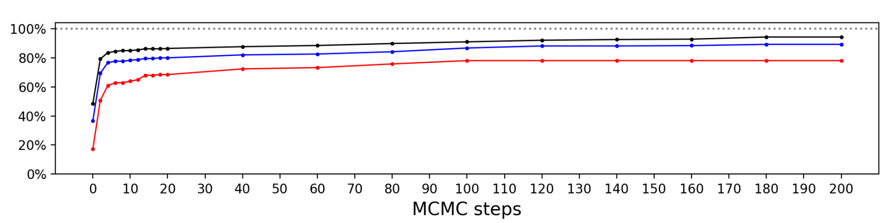

Fig. 8. Experimental results of MCMC-based perturbation learning. We run the experiment 20 times. In each run, we learn 1, 2 or 3 perturbations. The figure shows how the coverage of learned perturbations change as the search progresses. The x-axis indicates the number of MCMC steps taken, and the y-axis indicates the percentage of the buggy inputs that can be generated by *any* perturbation in the hypothesis space that can also be generated by the candidate perturbations at that point in the search. The red (blue, black) lines indicate the average coverage rate with 1 (2, 3) learnt perturbations.

It may be the case that the provided perturbation operators are simply not expressive enough to represent a reasonably complete set of buggy inputs. For example, it is impossible for a perturbation that only uses Θ_{merge} to cover all of E_{merge} since these operators can not "modify" existing elements in α_{merge}. In other words, the buggy input $([1;2],[3;5])$ cannot be generated using Θ_{merge} since perturbations derived from this operator set can only add to α_{merge} and not change its values. By examining both the learned perturbations and subsets of their closures, users may hypothesize that additional operators may be relevant to a particular bug. In such cases, they can refine the set of perturbation operators to yield a closure closer to E_{merge} (e.g., adding a `Stack.replace_last` operator that replaces the last element of a stack).

2.2 Perturbation Learning in Action

We now illustrate the details of our approach to perturbation learning on our running example. Suppose that we start from the randomly generated perturbation `f2` shown on the right.

```
let f2 (s1: Stack.t) (s2: Stack.t) =
    let (e1: int) = Stack.head s2 in
    let (l2: Stack.t) = Stack.snoc s1 e1 in
    (l2, s2)
```

This perturbation simply appends the head of `s2` to the end of `s1`. This is not a particularly good perturbation since its closure only contains 2 buggy inputs:

$$([1;2],[3;4]) \xrightarrow{\texttt{f2}} ([1;2;3],[3;4]) \xrightarrow{\texttt{f2}} ([1;2;3;3],[3;4])$$

Even from this relatively poor starting point, however, our algorithm can eventually find the perturbation `f0` shown in Figure 6, a significantly more desirable generator. To see how, consider the jumps depicted in Figure 9. For the first jump, we use a class of transformations that insert a new penultimate statement to `f2`. Two possible programs that can be produced by this modification are shown in Figure 10.

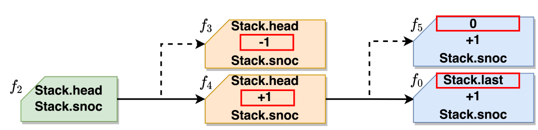

Fig. 9. Jumps from bad perturbation to a good one.

```
let f3 (s1: Stack.t) (s2: Stack.t) =
  let (e1: int) = Stack.head s2 in
  let (e2: int) = e1 - 1 in
  let (l3: Stack.t) =
    Stack.snoc s1 e2 in
  (l3, s2)
```

```
let f4 (s1: Stack.t) (s2: Stack.t) =
  let (e1: int) = Stack.head s2 in
  let (e2: int) = e1 + 1 in
  let (l3: Stack.t) =
    Stack.snoc s1 e2 in
  (l3, s2)
```

Fig. 10. Modified perturbations built from `f2`.

The new perturbation `f3` subtracts one from the head element of $\mathtt{s}_2$ and then appends it to the tail of $\mathtt{s}_1$. The elements in its closure satisfy the following property:

$$|\mathtt{s}_1| \geq 2 \land \mathtt{s}_1[0] = 1 \land \mathtt{s}_1[1] = 2 \land (\forall i, 2 \leq i < |\mathtt{s}_1| \implies \mathtt{s}_1[i] = 2) \land l_2 = [3;4] \qquad (E_3)$$

Elements in the closure for the other modified perturbation, `f4`, satisfy:

$$|\mathtt{s}_1| \geq 2 \land \mathtt{s}_1[0] = 1 \land \mathtt{s}_1[1] = 2 \land (\forall i, 2 \leq i < |\mathtt{s}_1| \implies \mathtt{s}_1[i] = 4) \land l_2 = [3;4] \qquad (E_4)$$

Notice that

$$E_3 \cap E_{\text{merge}} \equiv \{([1;2],[3;4])\}$$
$$E_4 \cap E_{\text{merge}} \equiv \{([1;2],[3;4]),([1;2;4],[3;4])\}$$

Our algorithm ascribes a transition probability from `f2` to both `f3` and `f4`; initially, both `f3` and `f4` are regarded as equally good candidates. However, note `f4`, whose closure includes 2 buggy inputs, is intuitively better than that of `f3` which contains only one. Since the number of buggy inputs in `f3`'s closure is no bigger than what in `f2`'s, the MCMC algorithm eventually transitions from `f2` to `f4`.

For the next jump, we apply a class of jumps that replace one of the statements in `f4` with a new one. Two possible results of this transformation are shown in Figure 11. Elements in the closure for

```
let f5 (s1: Stack.t) (s2: Stack.t) =
  let (e1: int) = 0 in
  let (e2: int) = e1 + 1 in
  let (l3: Stack.t) =
    Stack.snoc s1 e2 in
  (l3, s2)
```

```
let f0 (s1: Stack.t) (s2: Stack.t) =
  let (e1: int) = Stack.last s1 in
  let (e2: int) = e1 + 1 in
  let (l3: Stack.t) =
    Stack.snoc s1 e2 in
  (l3, s2)
```

Fig. 11. Modified perturbations built from `f4`.

perturbation `f5` satisfy the invariant:

$$|\mathsf{s}_1| \geq 2 \land \mathsf{s}_1[0] = 1 \land \mathsf{s}_1[2] = 2 \land (\forall i, 2 \leq i < |\mathsf{s}_1| \implies \mathsf{s}_1[i] = 1) \land l_2 = [3; 4] \qquad (E_5)$$

The second new perturbation, on the other hand, is exactly `f0` from the start of this section. Observe that E_5 contains only one buggy input, namely α_{merge}, making it a poor candidate for continued exploration.

We pause to note that the actual learning algorithm in Murphy compares the quality of perturbations by a heuristic evaluation procedure instead of simply comparing the total number of buggy inputs two perturbations can generate in a fixed number of iterations. Although this algorithm is more complicated than what we describe above, the intuition remains the same. The details of our learning procedure are introduced in Section 4 and Section 5.

Our search gets stuck at `f0` because none of the adjacent perturbations cover more buggy inputs. Murphy then returns `f0` as the perturbation that maximizes the number of covered buggy inputs among the candidates that have been explored. From this point, we can restart the search from a new starting perturbation, relying on the probabilistic nature of MCMC to explore a different region of the hypothesis space that can presumably reach, for example, `f1`. Once this process has terminated, the resulting perturbations are passed to a sampling component (the right-hand box in Figure 2), which generates a subset of their closure (up to a bound) to generalize the initial set of buggy inputs.

3 PROBLEM FORMALIZATION

We begin by formally defining the *error generalization* problem and our proposed solution based on perturbations. Our focus is on errors in functional programs that manipulate values of structured datatypes like lists, stacks, trees, and heaps. In both this section and the rest of the paper, we restrict the discussion to well-typed programs like merge and `f0` from the previous section. Given a program $\mathsf{P} : \tau_I \longrightarrow \tau_O$, we write $\mathsf{P}(\alpha) \Downarrow \beta$ to denote that applying P to input $\alpha : \tau_I$ produces the value $\beta : \tau_O$.

Definition 3.1. An instance of the error generalization problem is defined by three components:

- The *target program* that is being tested, $\mathsf{P} : \tau_I \longrightarrow \tau_O$. Like other sampling-based testers [Claessen and Hughes 2000; Dénès et al. 2014], we assume that P is a blackbox program, i.e., the only way we can inspect the behavior of P is by observing its inputs and outputs.
- A *specification* of the expected behaviors of P as a pair of pre- (Σ) and post- (Φ) conditions, denoted $\Sigma \implies \Phi$.
- A (possibly singleton) set of *initial inputs* $A_{\texttt{init}}$ that cause P to violate its specification.

Definition 3.2 (Buggy Inputs). For a given error generalization problem, we say that an input $\alpha : \tau_I$ is *buggy* if

(1) α satisfies the precondition: $\alpha \in \Sigma$, and
(2) applying P to α produces a value that violates the postcondition: $\exists \beta, \mathsf{P}(\alpha) \Downarrow \beta \wedge \beta \notin \Phi$.

Definition 3.3 (Error Generalization). We say that a set of inputs A is a *generalization* of (or A generalizes) the initial buggy inputs $A_{\texttt{init}}$ if all the elements of A are buggy and A is a superset of $A_{\texttt{init}}$ ($A_{\texttt{init}} \subseteq A$).

Example 3.4. For the target program merge with specification $\Sigma_{\texttt{merge}} \implies \Phi_{\texttt{merge}}$ from Section 2, E_{merge} generalizes the (singleton) set of initial buggy inputs $\{\alpha_{\mathsf{merge}}\}$.

According to this definition, $A_{\texttt{init}}$ is a valid generalization of itself, albeit one that is not very satisfying. In general, our goal is to generalize $A_{\texttt{init}}$ as much as possible, in the sense that our solution should contain every other generalization that we can identify. E_{merge} is an example of such a maximal generalization, in that it precisely captures *every* input that triggers a violation of $\Phi_{\texttt{merge}}$. Since we assume that the target program is blackbox, however, it is not obvious how to infer a specification for $E_{\texttt{merge}}$. Given that we can probe the behavior of P on specific inputs, one naïve solution to error generalization is to exhaustively test P on all inputs, recording every buggy input we find, but this is computationally infeasible. Instead, we adopt of the strategy of constructing a *generator* which generalizes an initial input by intelligently enumerating sets of additional buggy inputs. We construct these generators using transformations on the input datatypes of P.

Definition 3.5 (Perturbation). Given an instance of the error generalization problem, a *perturbation* is a function $f : \tau_I \to \tau_I$. A perturbation is *sound* with respect to a buggy input α iff for all natural numbers n, $f^n(\alpha)$ is also buggy, where f^n is the composition of f applied n times, i.e., $f^n = \overbrace{f \circ f \ldots \circ f}^{n}$.

Example 3.6. The function $\texttt{f0}$ from Section 2 produces only buggy inputs when repeatedly applied to α_{merge}, and is thus a sound perturbation for this input. In contrast, the perturbation $\texttt{f2}$ is not sound with respect to α_{merge}, as $\texttt{f2}^2(\alpha_{\mathsf{merge}}) \equiv ([1;2;3;3],[3;4])$ violates the precondition Σ_{merge}, since $[1;2;3;3]$ is not strictly increasing.

We lift a perturbation f to operate on sets of values in the neighborhood of a given buggy input $\alpha \in A_{\texttt{init}}$ as follows: $f{\uparrow}_{\alpha}(A) \triangleq \{\alpha\} \cup \{f(a) \mid a \in A\}$. (We omit the α subscript on $\uparrow$ when α is clear from context.) Iteratively applying this *perturbation functor* generates all buggy inputs reachable from α via f.

Example 3.7. Iteratively applying the perturbation functor for $\texttt{f0}$ from Section 2 produces the following sequence of generalizations of the buggy input α_{merge}.

$$\emptyset \xrightarrow{\texttt{f0}\uparrow} \{([1;2],[3;4])\} \xrightarrow{\texttt{f0}\uparrow} \left\{\begin{matrix}([1;2],[3;4]),\\ ([1;2;3],[3;4])\end{matrix}\right\} \xrightarrow{\texttt{f0}\uparrow} \left\{\begin{matrix}([1;2],[3;4]),\\ ([1;2;3],[3;4]),\\ ([1;2;3;4],[3;4])\end{matrix}\right\} \xrightarrow{\texttt{f0}\uparrow} \ldots$$

Observe that the set produced at each iteration in the previous example is included in the set of each subsequent iteration. In general, the sequence of error generalizations produced by a perturbation functor never shrinks:

Lemma 3.8 (Perturbation Functors are Monotone). *A perturbation functor $f{\uparrow}_{\alpha}$ built from a buggy input α and perturbation f is always (but necessarily strictly) monotone.*[2]

A direct corollary of this theorem is that the least fixed-point of a perturbation functor is guaranteed to exist:

[2]The proofs of the lemmas and theorems of all sections can be found in the supplementary material.

Corollary 3.9 (Perturbation Closure). *For a given instance of the error generalization problem, the least fixed-point of a perturbation functor $f{\uparrow}_\alpha$, denoted as $\mathtt{lfp}(f{\uparrow}_\alpha)$, for a given buggy initial input α exists. Furthermore, if f is sound, $\mathtt{lfp}(f{\uparrow}_\alpha)$ is a generalization of α.*

We call this fixed-point the *α-closure* of a buggy input α and perturbation f (we again omit α in cases where it is clear from context). Intuitively, when f is sound, this closure contains all the buggy inputs that can be explored by f from α. Observe that taking the union of the closures of multiple sound perturbations also produces a valid generalization of an error. The resulting generalization is also guaranteed to be at least as "good" a solution as the individual closures, in the sense that it is a superset of each of them. Thus, we can reduce the problem of error generalization to that of constructing sound perturbations for each buggy input in $A_{\mathtt{init}}$.

Theorem 3.10 (Error Generalization Via Perturbations). *Given an instance of the error generalization problem, and a non-empty set of sound perturbations F_α for each α in $A_{\mathtt{init}}$, we can build a generalization of $A_{\mathtt{init}}$ by taking the union of the closures of the perturbations for each buggy input, i.e. $\bigcup\limits_{\alpha \in A_{\mathtt{init}}} \bigcup\limits_{f \in F_\alpha} \mathtt{lfp}(f{\uparrow}_\alpha)$ is a valid generalization of $A_{\mathtt{init}}$.*

The full closure of a perturbation functor can be approximated by applying it to the empty set some fixed number of times: $f{\uparrow}^n_\alpha(\emptyset) \subseteq \mathtt{lfp}(f{\uparrow}_\alpha)$. We exploit this observation to construct a generalization from a set of sound perturbations for each buggy input in $A_{\mathtt{init}}$. If these perturbations are also *strictly* monotonic, we can grow the corresponding error generalization by increasing the number of functor compositions.

4 SYNTHESIZING PERTURBATIONS VIA MCMC-BASED LEARNING

We now turn to the task of how to construct perturbations in order to generalize buggy inputs. Unfortunately, we lack both formal specifications and the source of the program under test, and have only a limited number of buggy inputs from which to generalize. Thus, traditional program synthesis techniques are not a good fit for generating perturbations. Our solution is to instead use an MCMC-based learning algorithm to search for a collection of perturbations. An *MCMC-based sampler* draws elements from a probability destiny function in proportion to their value; in the limit, a sampler will model the (unknown) underlying distribution with vanishingly small errors. Intuitively, MCMC can be understood an intelligent hill-climbing technique that is robust to distributions with local minima. Formally, it is a mechanism to compute the posterior distribution ($p(\theta|x)$) in a Bayesian inference problem by approximating the normalization factor $p(x) = \int_\theta p(x|\theta)p(\theta)d\theta$ in Bayes' rule: $p(\theta|x) = \frac{p(x|\theta)\,p(\theta)}{p(x)}$. Because this factor may be intractable to compute, exact inference is often not possible, hence the need for approximation methods. We can frame the search for perturbations as a Bayesian inference problem in which ascertaining the quality of a perturbation is conditioned on prior knowledge that characterizes the probability distribution of buggy inputs in a program. In our setting, we can think of x as denoting a set of all possible inputs to a program; θ denoting a candidate perturbation; $p(\theta|x)$, the posterior, denoting the likelihood that θ correctly identifies elements in x as buggy or correct; $p(x|\theta)$ denoting the likelihood that x is actually buggy if it is claimed to be by θ; $p(\theta)$, the prior, denoting the likelihood that θ is actually a valid perturbation, absent any observations on its behavior with respect to x; and, $p(x)$ denoting the likelihood that an arbitrary element in x is buggy. Under this view, we seek to find a sampler for $p(\theta|x)$ that identifies perturbations that accurately differentiate buggy from non-buggy inputs.

MCMC methods are model-free techniques that repeatedly perform Monte Carlo sampling from an unknown posterior distribution by first setting up a Markov Chain whose stationary distribution is the one we sample from. It then simulates a random sequence of states (candidate perturbations in our setting) long enough to approximate (to some small error) a valid perturbation.

Implementations of MCMC sampling have three key components: a *sample* or *hypothesis* space, a *jump proposal*, and a *cost function*. The sample space simply defines the set of possible solutions that the algorithm can return. Jump proposals define how MCMC explores the sample space by specifying how to transition from one sample to another. The cost function assigns a real number *cost* to the elements of the sample space. MCMC algorithms compare two states using their costs, biasing the search towards samples with lower costs. Since it is only used to compare two samples, the cost function does not need to know which states in the hypothesis space are minimal.

As discussed in Section 3, the ultimate generalization of a set of buggy inputs is built from a collection of perturbations. In the rest of this section, we describe our adaptation of MCMC-based sampling to synthesize these functions individually. Our first challenge is to find a sample space that is amenable to exploration by MCMC. Akin to component-based synthesis approaches [Feng et al. 2017; Gulwani et al. 2011], we encode perturbations as a sequence of transformations over the datatypes of the inputs of the program under test.

This strategy affords us considerable flexibility, as it places no restrictions on the datatypes or the operations they support. Our set of solutions is thus parameterized over a set of basic components, which we call *perturbation operators*.

Definition 4.1 (Perturbation Operators). A perturbation operator $\theta : \tau_1 \to \tau_2$ is a terminating (possibly nondeterministic) function over a datatype τ_1. We denote the set of available perturbation operators as Θ.

Example 4.2. One possible set of perturbation operators for a target program that inserts an integer into a binary search tree includes operations for finding the upper bound of all elements in a binary tree (`upper_bound`); adding a new node to the leftmost leaf of a tree (`append_right`); rotating a tree counter-clockwise (`rotate_left`); dropping all the nodes on the lowest level of a tree (`drop_bottom`); as well as operations over integers (e.g., `max` and `min`).

Definition 4.3 (Hypothesis Space). For a program under test $P : \tau_I \to \tau_O$, perturbation operators Θ, and a bound on the number of statements m, the syntax of candidate solutions is:

$$\begin{aligned}
&x, y, z, f \in \text{variables} \qquad b \in \mathbb{B} \qquad \theta \in \Theta \\
&c ::= b \mid x \mid c \wedge c \mid c \vee c \mid \neg c \\
&s ::= \textbf{let}\ \overline{x} \coloneqq\ \textbf{if}\ c\ \textbf{then}\ \theta(\overline{y})\ \textbf{else}\ \theta(\overline{y}) \\
&p ::= \textbf{def}\ f(\overline{x : \tau}) \coloneqq s_0; \ldots; s_m; \textbf{return}\ \overline{z}
\end{aligned}$$

where $\bar{\cdot}$ indicates a tuple of variables, and $\overline{z}$ has type τ_I. We use $Hyp(\tau_I, \Theta, m)$ to denote the corresponding *hypothesis space* of candidate perturbations. Every perturbation in such a hypothesis space has a corresponding perturbation closure, per Theorem 3.9.

Example 4.4. As a revision of the bad perturbation `f2` we introduced in Section 2, `f2'` appends the head element of s_1 to s_2 only when it is greater than the upper bound of s_2. Note that the statement `Stack.head s2` is syntactic sugar for: `if true then Stack.head s2 else Stack.head s2`.

```
let f2' (s1: Stack.t) (s2: Stack.t) =
  let (e1: int) = Stack.head s2 in
  let (e2: int) = Stack.upper_bound s2 in
  let (b3: bool) = e1 > e2 in
  let (l4: Stack.t) = if b3
    then Stack.snoc s1 e1
    else Stack.snoc s1 e2  in
  (l4, s2)
```

With this hypothesis space in hand, we next present the details of our MCMC-based algorithm for exploring this space, including our choice of jump proposal and cost function.

Algorithm 1: Perturbation Learning

```
Input : Target program P : τ_I → τ_O, pre- and post-condition Σ and Φ, perturbation operators Θ,
        initial buggy inputs A_init, number of instructions m, termination condition C, the number of
        perturbations to be learned n, and the cost function iteration count t.
Output : Mapping from each initial buggy input to a set of perturbations.
1  foreach α ∈ A_init do
2      F_α ← ∅;
3      repeat n times
4          do
5              F ← ArgAssign(ChooseM(Θ, m), τ_I);
6          while F = ∅;
7          f ← ChooseM(F, 1);
8          cost_f ← CalculateCost(f, α_i, Σ, Φ, t);
9          (f_best, cost_best) ← (f, cost_f);
10         while C do
11             f′ ← Jump(Θ, f);
12             cost_f′ ← CalculateCost(f′, α_i, Σ, Φ, t);
13             f ← Judge(f, cost_f, f′, cost_f′);
14             if cost_f < cost_best then
                   (f_best, cost_best) ← (f, cost_f);
15         F_α ← F_α ∪ f_best;
16 return [α ↦ F_α];
17 Procedure Judge(f, cost_f, f′, cost_f′)
18     if cost_f′ < cost_f then
19         return f′
20     else if ChooseM([0.0, 1.0], 1) < cost_f / cost_f′ then
21         return f′
22     else
23         return f
24 Procedure Sample(A_init, 𝓕, k)
25     E ← ∅;
26     foreach α ∈ A_init do
27         foreach f ∈ 𝓕(α) do
28             for i ← 0 to k do
29                 E ← E ∪ {f^i(α)};
30     return E;
```

5 ALGORITHM

The details of our MCMC-based approach to *perturbation synthesis* are presented in Algorithm 1. This algorithm takes as input the components of the error generalization problem, the parameters that define the hypothesis space, and a termination criterion C. The algorithm returns a map from each initial buggy input to a collection of n perturbations. The main loop of the algorithm implements the random walk described in Section 4 using three key subprocedures: the jump proposal *Jump*, cost function *CalculateCost*, and the MCMC judgement *Judge*. The algorithm is nondeterministic, which manifests as calls to a function $ChooseM(A, m)$. This function nondeterministically selects m elements from the set A at random, and can select the same element multiple times.

For each initial buggy input α, the algorithm starts by initializing the corresponding set of perturbations F_α to the empty set (line 2) and then constructs n perturbations for α (lines $4-15$). The main perturbation learning algorithm begins by building an initial candidate solution (lines $4-9$). This candidate is built by picking m perturbation operators from Θ at random until a set is found that can be used to construct at least one well-typed program (line 5). One of these functions is then randomly selected as the initial solution, and both this perturbation and its cost are assigned to a pair of variables, f_{best} and $cost_{best}$, that keep track of the best known solution for the current buggy input. The algorithm then enters a loop that explores the solution space (lines $10-15$) using the MCMC-based strategy outlined in Section 4.

The body of this loop first uses *Jump* to propose a new solution based on the current one, and then calculates its cost using *CalculateCost*. The loop then calls the *Judge* subroutine to decide whether to adopt the newly proposed function as the current candidate. *Judge* uses the well-known Metropolis–Hastings algorithm [Hastings 1970] to compare two perturbations f and f' (lines $18-23$). If f' has a lower cost than f, it is always selected. Otherwise, f' is chosen when $\frac{cost_f}{cost_{f'}}$ is greater than a random number between 0.0 and 1.0. This strategy ensures that the likelihood the algorithm jumps to a worse solution is governed by its cost relative to the cost of the current candidate. The random walk ends when the input termination condition C is satisfied. In our implementation, this is either a bound on the number of loop iterations or the total execution time

The algorithm returns a mapping from each initial buggy inputs to its corresponding set of learned perturbations. Murphy then uses the *Sample* subroutine (lines 24 – 30) to construct an error generalization from this result. *Sample* begins by initializing the generated buggy inputs E to the empty set (line 25), then iteratively applies each learned perturbation to its corresponding initial buggy input k times (lines 26 – 29), and finally returns E as the final set of generalized buggy inputs. We observe that as k approaches ∞, E grows closer to $\bigcup_{\alpha \in A_{\texttt{init}}} \bigcup_{f \in F_\alpha} \texttt{lfp}(f{\uparrow}_\alpha)$.

5.1 Jump Proposal

The *Jump* subroutine takes the set of perturbation operators and the current solution f, and proposes a new candidate solution by slightly modifying f. Inspired by stochastic approaches to compiler superoptimization [Schkufza et al. 2016; Sharma et al. 2015], *Jump* nondeterministically applies a type-safety-preserving transformation drawn from one of four classes:

(1) `ReArgAssign`: randomly reassign the arguments of the perturbation operators;
(2) `SwapStatements`: swap two statements;
(3) `ReplaceOperator`: replace the perturbation operator in a statement with another drawn from Θ;
(4) `ReplaceGuard`: replace the guard in a statement with another one built from the current variable context.

We add constant operators that do not need any inputs (e.g. `true` : `bool`) in Θ to ensure *Jump* can (probabilistically) explore every state in the hypothesis space defined by Θ and m.

Theorem 5.1 (Soundness of Jump Proposal). *For an input type τ, set of perturbation operators Θ, and bound n, there exists a finite path between any pair of perturbations in the hypothesis space $Hyp(\tau, \Theta, m)$ via Jump.*

5.2 Cost Function

The "true" cost of a perturbation is determined by the number of perturbations that improve on it, i.e., those which contain its closure. As it is computationally expensive (and likely impossible) to directly calculate this ideal cost, the *CalculateCost* subroutine, presented in Algorithm 2, instead tries to approximate it. The final cost of f is largely determined by generating a subset of its closure. This subset is initialized to α and is then iteratively extended by applying f to all of its elements a fixed number of times t, adding any newly discovered inputs to A at each step. After generating this set, *CalculateCost* records the ratio of buggy inputs in A to its total number of elements (line 9).

This ratio is not the final cost, however. In order to achieve good coverage of the hypothesis space, we also prioritize three properties of a perturbation: it should not fail, it should produce

Algorithm 2: The *CalculateCost* subroutine.

Input : A perturbation f, buggy input $\alpha_{\texttt{init}}$, program under test P, specification $\Sigma \implies \Phi$, and iteration bound t.
Output : The cost of f.

1 $(A, c_E, c_D, c_{Op}) \leftarrow (\{\alpha_{\texttt{init}}\}, 0, 0, 0)$;
2 **for** $i \leftarrow 1$ **to** t **do**
3   **for** $\alpha \in f(A)$ **do**
4   **if** $\mathsf{P}(\alpha)$ *raises exception* **then**
5             $c_E \leftarrow c_E + ExcptnPenalty(i)$;
6   **if** $\alpha \in A$ **then**
7             $c_D \leftarrow c_D + DupValuePenalty(i)$;
8         $A \leftarrow A \cup \{\alpha\}$;
9 $c \leftarrow |\{\alpha \in \Sigma \wedge \exists \beta \in \mathsf{P}(\alpha).\beta \notin \Phi \mid \alpha \in A\}| / |A|$;
10 $c_{Op} \leftarrow c_{Op} - DupOpPenalty(f, x)$;
11 **return** $WeightedSum(c, c_E, c_D, c_{Op})$;

(mostly) unique elements, and it should use the perturbation operators in interesting ways. Our cost calculation includes three additional costs (c_E, c_D, and c_I) to encourage the discovery of perturbations with these features (line 11). To penalize exceptions inside the body of a perturbation, which happens when, e.g., the candidate perturbation tries to take the head of an empty list, we add an additional penalty for each generated input that triggers an exception (line 5). We also penalize any duplicate tests (line 7) to encourage the generation of unique inputs. We assume that the exceptions and duplicates that occur in later iterations are less harmful, and so we discount the corresponding penalties based on which iteration of f generated them.

Finally, we want to bias our search towards candidate perturbations that use a diverse set of perturbation operators. To see why, recall the target program merge and its corresponding perturbation `f0` from Section 2. The perturbation `f0'` shown to the right is almost the same as `f0`, but it adds an additional element to `s1`. The sequence of inputs generated by these perturbations are as follows:

```
let f0' (s1: Stack.t) (s2: Stack.t) =
  let (e1: int) = Stack.last s1 in
  let (e2: int) = e1 + 1 in
  let (l3: Stack.t) = Stack.snoc s1 e2 in
  let (e4: int) = e2 + 1 in
  let (l5: Stack.t) = Stack.snoc l3 e4 in
  (l5, s2)
```

$$([1;2],[3;4]) \xrightarrow{\texttt{f0}} ([1;2;3],[3;4]) \xrightarrow{\texttt{f0}} ([1;2;3;4],[3;4]) \xrightarrow{\texttt{f0}} ([1;2;3;4;5],[3;4]) \xrightarrow{\texttt{f0}} ([1;2;3;4;5;6],[3;4]) \ldots$$

$$([1;2],[3;4]) \xrightarrow{\texttt{f0'}} ([1;2;3;4],[3;4]) \xrightarrow{\texttt{f0'}} ([1;2;3;4;5;6],[3;4]) \ldots$$

Observe that applying `f0'` is equivalent to applying `f0` twice, so the full closure of `f0` includes that of `f0'`. To avoid this sort of redundancy, we analyze the sequence of perturbation operations were used to build each output value in a perturbation, and assess a penalty when some are repeated. As an example, the sequence for $\texttt{l}_5$ in `f0'` is `Stack.last`; `+1`; `Stack.snoc`; `+1`; `Stack.snoc` which uses the operator `+1` and `Stack.snoc` twice; causing `f0'` to have a greater cost than `f0`.

6 EVALUATION

To evaluate our approach, we have implemented an automated error generalization framework, called Murphy, that targets functional OCaml programs which manipulate rich abstract datatypes (ADTs) like stacks, heaps, and trees. Murphy takes five parameters: (1) a black-box target program, (2) the program's specification in the form of a pre- and postcondition, (3) a (possibly singleton) set of buggy inputs, (4) a set of perturbation operators, and (5) bounds on training time, input generation time, and the size and number of perturbations. Given these inputs, Murphy learns a generator that produces a family of tests which generalize the original buggy inputs. Murphy comes equipped with a standard library of transformations on common algebraic datatypes. This library provides 20 operations for binary trees, for example, including functions like `root`, `rotate_left`, and `max`.

Our experimental evaluation considers four key questions[3]:

- **Q1**: Is Murphy *effective*? How does it generalize the provided buggy inputs compared to other automated test frameworks?
- **Q2**: Is Murphy *efficient*? Does it produce this family of tests in a reasonable amount of time?
- **Q3**: Does Murphy *generalize well*? Does Murphy learn perturbations that explain most of the buggy inputs representable in a given hypothesis space?
- **Q4**: Is Murphy *useful*? Can it generate results that improve the results of other black-box program analysis tools?

[3]The supplementary material includes an evaluation of how sensitive Murphy is to the number of perturbation operators provided.

All reported data was collected on a Linux server with an Intel(R) Core(TM) i7-8700 CPU @ 3.20GHz and 64GB of RAM.

Table 1. Experimental results. The $|\Theta|$ column indicates the number of perturbation operators used in each benchmark. There are two main groupings of columns. The first presents the baseline results using the default generators provided by QuickCheck to generate inputs for 500s. The second presents the results of using Murphy to learn two perturbations using a 200s bound on training time and then generating inputs for 50s each. Each group characterizes the quality of the tests generated with respect to the total number of tests produced: the percentage of generated inputs that satisfy the precondition (Σ), the percentage of generated inputs that are buggy, i.e. that produce an output violating the postcondition ($\Sigma \wedge \neg\Phi$), and the percentage of buggy inputs which are unique ($\Sigma \wedge \neg\Phi \wedge \exists!$). As Murphy always generates unique buggy inputs, we omit Σ, $\Sigma \wedge \neg\Phi$ and $\Sigma \wedge \neg\Phi \wedge \exists!$ in its columns. Each group includes the average time each tool needs to find a single buggy input (t). This average is computed by dividing the total execution time (500s) by the number of unique buggy inputs generated. We also present the percentage of all feasible buggy inputs that are explained by the learned perturbations ($\cup^2/\cup^{Hyp}$).

		QuickCheck				Murphy	
Benchmark	$\lvert\Theta\rvert$	Σ	$\Sigma \wedge \neg\Phi$	$\Sigma \wedge \neg\Phi \wedge \exists!$	t (ms)	t (ms)	$\cup^2/\cup^{Hyp}$
BankersQ	16	0.022%	0.022%	0.022%	387.914	0.230	95.52%
BatchedQ	16	11.970%	11.822%	11.510%	0.727	0.180	80.15%
BinomialHp	17	7.523%	0.003%	0.003%	5891.519	0.143	91.48%
CustomStk	16	1.985%	0.524%	0.514%	18.397	0.216	97.90%
SortedL	16	1.987%	0.212%	0.211%	56.743	0.211	87.43%
SplayHp	20	47.321%	0.028%	0.028%	48.649	0.248	78.01%
Stream	17	1.467%	0.445%	0.425%	28.688	0.253	98.38%
Trie	30	4.293%	3.619%	3.616%	2.199	0.239	61.84%
UnbSet	20	62.029%	0.735%	0.723%	1.984	0.239	96.59%
UniqueL	16	13.460%	7.568%	6.137%	0.731	0.182	70.36%
RbSet	16	0.250%	0.129%	0.129%	16.495	0.169	87.29%
LeftistHp	19	21.867%	0.002%	0.002%	133.875	0.226	93.36%
PhysicistsQ	17	0.002%	0.001%	0.001%	20146.334	0.197	99.60%
RealtimeQ	17	0.025%	0.001%	0.001%	1219.821	0.214	97.16%
SkewHp	16	7.382%	0.001%	0.001%	35877.397	0.164	92.26%
PairingHp	16	12.272%	0.000%	0.000%	∞	0.198	95.81%

Efficiency and Effectiveness. To address the first two questions, we constructed a corpus[4] of abstract data type (ADT) implementations drawn from Okasaki [1999], the OCaml standard library [Leroy et al. 2014], Verified Functional Algorithms [Appel 2018], and Software Foundations [Pierce et al. 2010]. We then conducted two sets of experimental evaluations of Murphy using sixteen benchmarks, each of which consists of the following components:

- **Target Program**: A single ADT operation into which we have manually injected a fault.
- **Specification**: A pair of pre- and post-conditions built from the representation invariant of the ADT and the intuitive specification of the faulty operation.
- **Initial Input**: A single, manually written buggy input.
- **Perturbation Operators**: The operators provided by Murphy's standard library for the algebraic data type used as the representation type of the ADT.

The baseline point of comparison for our first set of experiments is the set of errors generated by a blackbox automated testing framework that generates datatypes with zero knowledge of

[4]All the benchmarks and results from our evaluation are provided in the supplementary material.

the error. To establish this baseline, we chose the popular QuickCheck framework [Claessen and Hughes 2000], and used the generators provided by the library to randomly sample values of the underlying representation types used by each benchmark. The generator for lists of integers used by CustomStk, SortedL and UniqueL, for example, uses the built-in `small_nat` generator for list elements and a uniform distribution to select either `nil` or `cons` when generating a list. We use a 500s time limit when generating tests with both Quickcheck and Murphy. Per Algorithm 1, Murphy has two phases: it first synthesizes perturbations for the initial inputs, and then uses the learned function to generate additional test inputs. In order to ensure a fair comparison, our evaluation includes the time for both phases: we first spend 200 + 200s synthesizing two perturbations and the next 50 + 50s generating test inputs using each perturbation.

To judge the effectiveness of Murphy against our baseline (**Q1** and **Q2**), we define the following three criteria for measuring the quality of a set of tests:

- Σ How many of the tests represent *valid inputs*, i.e. satisfy the precondition of the program under test?
- $\neg\Phi$ How many of the tests are *buggy*, i.e. cause the program under test to produce an output that violates the postcondition?
- $\exists!$ How well does the set generalize the initial input, i.e., how many *unique* elements does the set contain?

The subset of unique buggy inputs (i.e., those tests that satisfy all three criteria) represents the set of *high-quality* tests generated by an automated testing framework. The detailed results of our experiments are shown in Table 1. We note that Murphy *always* generates high-quality tests for each of the benchmarks, i.e. it was able to successfully learn a sound perturbation for each benchmark. The baseline approach, in contrast, generates fewer (at most 62.0%) inputs satisfying the precondition; after filtering out non-buggy tests, this decreases substantially to at most 11.5%. There are two primary reasons behind the poor performance of the generators based on purely random sampling:

(1) Some ADTs (e.g. real time queue, physicist's queue) have strict representation invariants, making it unlikely that a random generator will choose values satisfying the precondition of the function (Σ). Equipped with perturbation operators for the underlying representation type, on the other hand, Murphy is able to learn perturbations that preserve the representation invariant of each ADT.
(2) Even if the representation invariant is relatively permissive, it is difficult for a zero-knowledge random generator to trigger an error when the program under test behaves correctly on most inputs ($\Sigma \land \neg\Phi$).

Taken together, these results demonstrate that Murphy can effectively generalize buggy inputs.

This set of experiments also provides evidence that Murphy can efficiently generalize an error (**Q2**). To demonstrate this, we calculated the average time needed to generate a unique buggy input using both QuickCheck and Murphy (column t in Table 1). Even accounting for the 200s of training time per perturbation, Murphy is able to quickly generate a large family of high-quality tests, averaging less than a tenth of a millisecond per unique buggy input. Even in the worst case, this is more than 4x faster than the baseline Quickcheck implementation.

Our second set of experiments was designed to evaluate how well Murphy can generalize from a small set of buggy inputs to find perturbations that capture properties relevant to the error in the program under test (**Q3**), e.g. E_{merge}. Since the generalizations Murphy can build are dictated by the hypothesis space of possible perturbations, our point of comparison is the best solution in that space: the generalization built from closures of *every* perturbation in the solution space, which we call the set of *feasible* buggy inputs. We attempt to quantify how well a set of learned perturbations covers the feasible buggy inputs by measuring how many of the inputs can be “explained” by the learned perturbations. To do so, we first approximate the set of feasible bugs by exhaustively enumerating every well-typed candidate perturbation, generating (bounded) sets of buggy inputs from this enumeration, filtering out any non-buggy inputs from the resulting sets, and taking the union of those sets.

Next, we use a specification inference tool [Zhou et al. 2021] to infer specifications for each of the perturbations for the benchmarks in Table 1. The inferred specifications act as an explanation of the sorts of bugs covered by each perturbation. As an example, the specification inferred for `f2` from Section 2 is “all the elements of the first stack are less than or equal to all the elements of the second stack”, a specification that fails to explain a feasible bug like $([1; 2; 3; 4], [3; 4])$. Alternatively, the specification inferred for `f0` is “the head element in the first stack is less than or equal to all the elements of the second stack”, which does explain the aforementioned buggy input. To measure the quality of the learned perturbation, we look at the feasible buggy inputs we sampled from all possible perturbations and calculate the percentage which satisfy the inferred specifications. The results of these experiments are shown in Table 1. With two exceptions, the perturbations learned by Murphy cover at least 75% of all sampled feasible buggy inputs. The first of these exceptions, Trie, has the largest hypothesis space, since it requires perturbation operators for both lists and trees, suggesting that Murphy needs more training time to fully explore the space. The other exception is the UniqueL benchmark, which has one of the largest sets of buggy inputs, making it hard for just two perturbations to cover the full region.

```
1  let rec insert (x: int) (s: int unbset) =
2    match s with
3    | Leaf -> Node (x, Leaf, Leaf)
4    | Node (y, a, b) ->
5      if x < y
6      then Node (x, a, insert y b)
7      else if y < x
8      then Node (y, a, insert x b)
9      else s
```

```
1  let f_unbset (x: int) (s: int unbset) =
2    let (lb: int) = upper_bound s in
3    let (s1: int unbset) = append_right lb s in
4    let (s2: int unbset) = rotate_left s1 in
5    let (s3: int unbset) = drop_bottom s2 in
6    (x, s3)
```

Fig. 12. Target program and synthesized perturbations for the unbalanced set benchmark (UnbSet).

Interesting Functions. Not surprisingly, our machine learning-based approach allows Murphy to synthesize interesting and non-obvious generators. As one example, consider the UnbSet benchmark, which targets a set ADT backed by an unbalanced binary tree. The operations of this ADT assume the tree is sorted: $l < n$ for all left children l of node n, and $n < r$ for all right children r. The `insert` operation that is shown on the left-hand side of Figure 12 fails to maintain this invariant due to a bug on line 6, which recursively inserts the current node `y` instead of `x`. When given an integer 0 and the tree shown in Figure 13a, for example, `insert` produces the unsorted tree shown in Figure 13a'. Equipped with this buggy input and its stock set of tree perturbation operators, which includes the `upper_bound`, `append_right`, `rotate_left`, and `drop_bottom` functions described in Section 4, Murphy infers the `f_unbset` function shown on the right-hand side of Figure 12. Although these stock tree perturbation operators do not know anything about the sorted tree invariant, the learned perturbation nevertheless respects this invariant. Figure 13 gives an example execution of `f_unbset`. This function first finds the upper bound of the elements of the input tree, which it then appends to the right-most leaf of the tree (lines 2-3), producing the tree in Figure 13b. Next, `f_unbset` rotates that tree (line 4) to construct the tree in Figure 13c (the rotated nodes are boxed in the figure), before dropping all the nodes on the lowest level. Notice that the resulting tree, shown in Figure 13d, is also sorted, but inserting 0 into it produces the buggy tree shown in Figure 13d'.

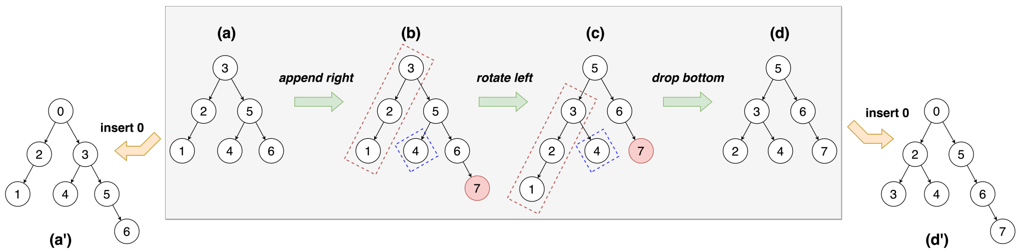

Fig. 13. An example execution of the learned perturbation (`f_unbset`) for the unbalanced set benchmark.

Utility. In order to evaluate the usefulness of the tests generated by Murphy, we used it to augment the training data used by two learning-based blackbox program analyses (**Q4**). The quality of results for any data-driven program analysis [Miltner et al. 2020; Padhi et al. 2016; Zhou et al. 2021] fundamentally depends on the data they are given. Granted access to the program source, these analyses can improve their results by augmenting this data set by inspecting the program, e.g., by querying a verifier to find witnesses of unsafe behavior. When analyzing blackbox programs, however, it is not always clear how to gather additional examples of, e.g., safety violations. Since Murphy learns to generate precisely these sorts of inputs, we hypothesized that it could be used to improve the performance of these tools. To investigate this hypothesis, we targeted two existing data-driven specification inference tools, PIE [Padhi et al. 2016] and Elrond [Zhou et al. 2021].

The first of these, PIE, is a tool for inferring a precondition under which it is safe to execute a program. Like Murphy, PIE does not assume access to the source code of the program; instead it learns an assertion that accepts a set of "good" tests satisfying some property while also rejecting a set of "bad" tests that violate the property. PIE is biased towards learning the weakest consistent assertion, so if the set of bad tests is too small, it may produce a precondition that is unsafe. We thus investigated whether augmenting the set of bad tests with additional buggy inputs produced by Murphy can help PIE to avoid this pitfall.

To do so, we looked for existing applications of PIE that align with the expected use case for Murphy, namely programs that manipulate algebraic data types and that have at least one safe input and one buggy input. We could identify two such programs in the original benchmark suite of PIE [Padhi et al. 2016]. Both programs are accompanied by a postcondition used for distinguishing between good and bad tests, and a precondition that represents the "correct" precondition for the program. For each benchmark, we used PIE to generate a precondition from each of the following test suites: a baseline set of 100 random tests produced by Quickcheck, the baseline set enhanced with 100 additional randomly-generated tests, and the baseline set augmented with 5 bad tests generated by a perturbation synthesized by Murphy. These three sets are respectively labelled ϕ_π, ϕ_{π^2}, and $\phi_{\pi+\mu}$ in Table 2. To control for the randomness of automated test generation, we repeated each experiment 60 times. The entry under each category reports the probability that PIE infers a precondition that is logically equivalent to the precondition provided by the original benchmark.

As Table 2 shows, the accuracy of PIE on both benchmarks improves when augmented with tests from Murphy. For the failing cases for the first benchmark (RevInv) PIE inferred the incorrect precondition $len(l) \neq 0 \wedge len(l) \neq 1$. Here, QuickCheck failed to produce a test with a list that was not a non-trivial palindrome (i.e. one with more than one element). Using the "trivial" singleton lists as buggy inputs, Murphy was able to find more "interesting" negative samples (e.g., palindromes with more than a single element), which in turn guided PIE towards the correct precondition, $rev(l) \neq l$. This example shows how the cost function's use of heuristics to identify "interesting" inputs can help improve the coverage of the learned perturbation. For the 10% of cases in LenApp

Table 2. Experimental results of augmenting PIE's test suite with additional tests from Murphy. The first three columns report the name of the original benchmark, the desired postcondition, and the "correct" safe precondition that should be inferred (Σ_{safe}). The last three columns indicate the probability that PIE infers a precondition equivalent to Σ_{safe} across 60 experimental runs using tests drawn from one of three categories: ϕ_π is a baseline set of 100 tests produced by zero-knowledge generators, ϕ_{π^2} augments that set with 100 additional tests produced by zero-knowledge generators, and $\phi_{\pi+\mu}$ augments ϕ_π with 5 bad tests generated by a perturbation synthesized by Murphy.

Benchmark	Postcondition	Σ_{safe}	ϕ_π	ϕ_{π^2}	$\phi_{\pi+\mu}$
RevInv	$\texttt{list\_rev}(l) \neq l$	$rev(l) \neq l$	41.7%	75.0%	100.0%
LenApp	$\texttt{list\_append}(l_1, l_2) = []$	$len(l_1) = 0 \land len(l_2) = 0$	3.3%	5.0%	90.0%

where using Murphy did not help PIE find the correct precondition, we observed that the set of randomly-generated *good* tests was insufficient; in this benchmark, the tests did not include an example in which both input lists were empty. We additionally observed that Murphy was effective at helping PIE *refine* an initial precondition: for each run, it was always the case that $\phi_{\pi+\mu} \implies \phi_\pi$, i.e. Murphy always produced an input that strengthened the baseline precondition inferred by PIE.

Our second set of experiments targeted Elrond [Zhou et al. 2021], a data-driven tool that infers specifications of library functions which are sufficient to ensure the safety of a given client of the library. As with PIE, Elrond does not assume access to the source code of the library, and instead probes blackbox implementations of its methods in order to infer their specifications. Elrond uses data from Quickcheck to provide tests that are used to construct a candidate specification; these tests are effectively treated by Elrond as counter-examples to refine its current proposed specification. An additional SMT-based logical refinement phase is then used to further safely weaken (or generalize) this specification. We hypothesized that Murphy could find additional useful inputs that QuickCheck could not, helping Elrond to infer better initial candidates. Doing so would reduce the number of SMT queries needed by the second phase, and in turn, improve the overall execution time needed to perform specification inference.

To test this hypothesis, we inserted an intermediate phase between Elrond's initial learning and weakening phases. This intermediate phase uses Murphy to learn two perturbations within a 200 + 200s time bound, and then iterates each perturbation 2000 times to generate additional tests. Those tests were then used to refine the candidate specifications one final time before the specifications were handed off to the SMT-based weakening phase.

The results of this experiment for each of the 6 benchmarks that timed out in the original evaluation of Elrond are shown in Table 3. In every benchmark, Murphy generates a significant amount of additional data, which in each case helps generalize at least one of the inferred library specifications. This corresponds to at least a 70.3% (at least 250 minutes) reduction in the time spent in the weakening phase across all benchmarks. This significant improvement happens because many potential weakenings were covered by the additional data generated by Murphy, before the beginning of the SMT-intensive weakening phase.

7 RELATED WORK

Stochastic Search for Programs. The idea of using statistical sampling methods like MCMC to explore a space of programs has also been investigated in STOKE [Schkufza et al. 2013; Sharma et al. 2013], a stochastic superoptimizer that uses MCMC sampling to explore the space of possible programs in search of one that is an optimization of the given target program. Like Murphy, the sampling-based framework employed by STOKE makes it necessarily incomplete; its experimental results, however, demonstrate that it is capable of constructing programs that far outperform those

Table 3. Experimental results of the Elrond case study, which includes the six benchmarks whose weakening phase timed out in the original evaluation of Elrond. The first group of columns lists the benchmark name, the number of distinct library functions it uses ($|F|$), and the total number of library function calls in the client program ($|R|$). The next group of columns presents results from the unmodified version of Elrond. The two columns report the total number of counterexamples generated by Quickcheck during the initial learning ($|cex|$) phase and the total weakening time in minutes (t_w). An entry of 1000+ for t_w indicates the benchmark failed to finish after 16+ hours. The third group reports the results for the version of Elrond augments with counterexamples generated by Murphy. The last three columns report the number of additional counterexamples found by Murphy ($|cex|^\mu$), the number of inferred specifications weaker than the initial solutions produced by the initial learning phase ($|F|^\mu$), and the total weakening time needed for the refined set of specifications (t_w^μ).

Benchmark	$\|F\|$	$\|R\|$	$\|cex\|$	$t_w(min)$	$\|cex\|^\mu$	$\|F\|^\mu$	$t_w^\mu(min)$
Stack	4	5	37	453.84	1764	2	126.91
Heap	3	8	78	1000+	1885	2	94.47
	2	21	155	1000+	779	1	119.32
	2	21	178	308.94	1161	1	56.02
Set	1	21	110	1000+	910	1	94.10
	2	8	85	351.64	1285	2	104.36

produced by traditional superoptimizers. Unlike Murphy, however, which uses MCMC to explore the space of perturbations for the purposes of error generalization, STOKE's sampling algorithm is defined in terms of a cost function that integrates correctness of a proposed transformation (in terms of results over test cases) and runtime performance improvement (statically approximated); its search procedure therefore lacks any notion of generalization in determining the utility of a candidate program. Liang et al. [2010] present an MCMC-based framework for synthesizing programs in combinatory logic whose terms are defined in a probabilistic context-free grammar and whose learning objective is determined from training examples. Their goals and methodology are notably different from Murphy's.

There has been much recent work in the program synthesis community on using machine learning methods to synthesize and reason about programs [Allamanis et al. 2018; Alon et al. 2019; Bielik et al. 2016; Murali et al. 2018; Pradel and Chandra 2021; Raychev et al. 2019], attempting to generalize semantically-relevant properties by learning from (often very large) corpora. In contrast, our approach relies on a probabilistic sampling method to construct a random walk over the space of candidate perturbations, and crucially makes no assumptions on the availability of training data, allowing it to generalize from a small (possibly single) set of buggy inputs. An et al. [2019] explored how programming-by-example systems could leverage the hypothesis that candidate programs are robust to user-specified semantic properties in order to generalize from a small set of examples.

Genetic Programming. Our technique also bears some similarity to genetic programming [Forrest et al. 2009; Goues et al. 2012] and evolutionary search [Mendelson et al. 2021] methods insofar as they all involve exploring a high-dimensional space of candidate programs. While Murphy performs this search in service of error generalization, synthesizing functions that generate a family of inputs guaranteed to trigger a bug from provided buggy inputs, genetic programming uses tests that reflect both positive and negative executions in service of program repair or patch generation tasks in order to prevent the defects that triggered these negative executions. This difference in goals also leads to differences in approach - genetic programming methods rely on a predefined set of heuristics that govern program evolution while Murphy leverages a statistical sampling technique to search for high-quality perturbations.

Automated Test Generation. Murphy shares superficially similar goals to fuzzing [Godefroid 2020], a commonly-used automated testing mechanism that seeks to improve test coverage by mutating inputs. These techniques are generally categorized according to how much access they are given to the program under test. Whitebox fuzzing approaches use program analysis [Bounimova et al. 2013; Godefroid et al. 2012] and symbolic execution [Cadar et al. 2008; Godefroid et al. 2005] to guide the construction of new inputs that result in execution of new program paths. Greybox techniques such as AFL [Zalewski 2015] leverage instrumentation and dynamic execution to drive subsequent fuzzing actions. DeepFuzz [Liu et al. 2019] is a blackbox technique that learns a generative recurrent neural network which can generate syntactically well-formed C programs for fuzz testing C compilers. The closure of a perturbation can be thought of a set of fuzzed inputs derived from the original buggy input, each of which is guaranteed to generate an error, constrained by the method's precondition. Notably, however, the inputs generated by perturbations are not tailored for improving program coverage, since the approach is fully blackbox, but for error generalization. For similar reasons, Murphy is also distinguished from methods that explicitly synthesize test cases in the form of client programs [Samak and Ramanathan 2015; Samak et al. 2015] used to drive execution through libraries; these techniques also rely on some form of static analysis to guide the synthesis process.

Besides property-based random testing frameworks like QuickCheck [Claessen and Hughes 2000], metamorphic testing [Chen et al. 2018] uses the notion of metamorphic relations to define properties that drive the generation of new test cases from existing ones, without the need for a test oracle to ascertain the utility of the newly generated test. In contrast, Murphy generates inputs via a test generator synthesis procedure that avoids the need for users to supply metamorphic relations, using only the method's pre- and post-conditions and MCMC sampling to drive its search. While metamorphic testing is a general property-based testing technique, imposing few constraints on the structure of supplied relations that are used to define the properties of interest, Murphy's approach is lighter-weight and fully automatable, and, as demonstrated here, offers significant utility for error generalization tasks in functional programs.

8 CONCLUSION

This paper addresses the problem of *error generalization*, generalizing a small (possibly singleton) set of buggy inputs into a large family of similar bugs. Error generalization can help developers document, diagnose, and test fixes to software faults, as well as aid data-driven reasoning techniques which rely on a body of error-generating inputs. Our proposed solution uses an MCMC-based learning technique to synthesize perturbations, specialized test generators for these sorts of buggy inputs. We have built a tool based on our approach, called Murphy, and shown that it is highly effective at generalizing small sets of buggy inputs to blackbox functional programs that manipulate structured datatypes. We have also demonstrated that Murphy can help improve the efficacy of data-driven specification inference and verification tools by supplying additional useful data to these tools.

REFERENCES

Miltiadis Allamanis, Earl T. Barr, Premkumar T. Devanbu, and Charles Sutton. 2018. A Survey of Machine Learning for Big Code and Naturalness. *ACM Comput. Surv.* 51, 4 (2018), 81:1–81:37. https://doi.org/10.1145/3212695

Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. 2019. code2vec: Learning Distributed Representations of Code. *Proc. ACM Program. Lang.* 3, POPL (2019), 40:1–40:29. https://doi.org/10.1145/3290353

Shengwei An, Rishabh Singh, Sasa Misailovic, and Roopsha Samanta. 2019. Augmented Example-Based Synthesis Using Relational Perturbation Properties. *Proc. ACM Program. Lang.* 4, POPL, Article 56 (dec 2019), 24 pages. https://doi.org/10.1145/3371124

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

Jonathan Baxter. 2000. A Model of Inductive Bias Learning. *J. Artif. Intell. Res.* 12 (2000), 149–198. https://doi.org/10.1613/jair.731

Pavol Bielik, Veselin Raychev, and Martin T. Vechev. 2016. PHOG: Probabilistic Model for Code. In *Proceedings of the 33nd International Conference on Machine Learning, ICML 2016, New York City, NY, USA, June 19-24, 2016 (JMLR Workshop and Conference Proceedings, Vol. 48)*, Maria-Florina Balcan and Kilian Q. Weinberger (Eds.). JMLR.org, 2933–2942. http://proceedings.mlr.press/v48/bielik16.html

Ella Bounimova, Patrice Godefroid, and David A. Molnar. 2013. Billions and Billions of Constraints: Whitebox Fuzz Testing in Production. In *35th International Conference on Software Engineering, ICSE '13, San Francisco, CA, USA, May 18-26, 2013*, David Notkin, Betty H. C. Cheng, and Klaus Pohl (Eds.). IEEE Computer Society, 122–131. https://doi.org/10.1109/ICSE.2013.6606558

Cristian Cadar, Daniel Dunbar, and Dawson R. Engler. 2008. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. In *8th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2008, December 8-10, 2008, San Diego, California, USA, Proceedings*, Richard Draves and Robbert van Renesse (Eds.). USENIX Association, 209–224. http://www.usenix.org/events/osdi08/tech/full_papers/cadar/cadar.pdf

Tsong Yueh Chen, Fei-Ching Kuo, Huai Liu, Pak-Lok Poon, Dave Towey, T. H. Tse, and Zhi Quan Zhou. 2018. Metamorphic Testing: A Review of Challenges and Opportunities. *ACM Comput. Surv.* 51, 1 (2018), 4:1–4:27. https://doi.org/10.1145/3143561

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

Maxime Dénès, Catalin Hritcu, Leonidas Lampropoulos, Zoe Paraskevopoulou, and Benjamin C Pierce. 2014. QuickChick: Property-based Testing For Coq. In *The Coq Workshop*.

Yu Feng, Ruben Martins, Yuepeng Wang, Isil Dillig, and Thomas W. Reps. 2017. Component-Based Synthesis for Complex APIs. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages* (Paris, France) *(POPL 2017)*. Association for Computing Machinery, New York, NY, USA, 599–612. https://doi.org/10.1145/3009837.3009851

Stephanie Forrest, ThanhVu Nguyen, Westley Weimer, and Claire Le Goues. 2009. A Genetic Programming Approach to Automated Software Repair. In *Genetic and Evolutionary Computation Conference, GECCO 2009, Proceedings, Montreal, Québec, Canada, July 8-12, 2009*, Franz Rothlauf (Ed.). ACM, 947–954. https://doi.org/10.1145/1569901.1570031

Patrice Godefroid. 2020. Fuzzing: Hack, Art, and Science. *Commun. ACM* 63, 2 (2020), 70–76. https://doi.org/10.1145/3363824

Patrice Godefroid, Nils Klarlund, and Koushik Sen. 2005. DART: Directed Automated Random Testing. In *Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation, Chicago, IL, USA, June 12-15, 2005*, Vivek Sarkar and Mary W. Hall (Eds.). ACM, 213–223. https://doi.org/10.1145/1065010.1065036

Patrice Godefroid, Michael Y. Levin, and David A. Molnar. 2012. SAGE: Whitebox Fuzzing for Security Testing. *Commun. ACM* 55, 3 (2012), 40–44. https://doi.org/10.1145/2093548.2093564

Claire Le Goues, ThanhVu Nguyen, Stephanie Forrest, and Westley Weimer. 2012. GenProg: A Generic Method for Automatic Software Repair. *IEEE Trans. Software Eng.* 38, 1 (2012), 54–72. https://doi.org/10.1109/TSE.2011.104

Sumit Gulwani, Susmit Jha, Ashish Tiwari, and Ramarathnam Venkatesan. 2011. Synthesis of Loop-Free Programs. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation* (San Jose, California, USA) *(PLDI '11)*. Association for Computing Machinery, New York, NY, USA, 62–73. https://doi.org/10.1145/1993498.1993506

Sumit Gulwani, Oleksandr Polozov, and Rishabh Singh. 2017. Program Synthesis. *Foundations and Trends® in Programming Languages* 4, 1-2 (2017), 1–119. https://doi.org/10.1561/2500000010

W Keith Hastings. 1970. Monte Carlo Sampling Methods using Markov Chains and their Applications. (1970).

Leonidas Lampropoulos, Diane Gallois-Wong, Cătălin Hriţcu, John Hughes, Benjamin C. Pierce, and Li-yao Xia. 2017. Beginner's Luck: A Language for Property-Based Generators. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*. Association for Computing Machinery, New York, NY, USA, 114–129. https://doi.org/10.1145/3009837.3009868

Leonidas Lampropoulos, Michael Hicks, and Benjamin C. Pierce. 2019. Coverage Guided, Property Based Testing. *Proc. ACM Program. Lang.* 3, OOPSLA, Article 181 (oct 2019), 29 pages. https://doi.org/10.1145/3360607

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).

Percy Liang, Michael I. Jordan, and Dan Klein. 2010. Learning Programs: A Hierarchical Bayesian Approach. In *Proceedings of the 27th International Conference on Machine Learning (ICML-10), June 21-24, 2010, Haifa, Israel*, Johannes Fürnkranz and Thorsten Joachims (Eds.). Omnipress, 639–646. https://icml.cc/Conferences/2010/papers/568.pdf

Xiao Liu, Xiaoting Li, Rupesh Prajapati, and Dinghao Wu. 2019. DeepFuzz: Automatic Generation of Syntax Valid C Programs for Fuzz Testing. In *The Thirty-Third AAAI Conference on Artificial Intelligence, AAAI 2019*. AAAI Press, 1044–1051. https://doi.org/10.1609/aaai.v33i01.33011044

Jonathan Mendelson, Aaditya Naik, Mukund Raghothaman, and Mayur Naik. 2021. GENSYNTH: Synthesizing Datalog Programs without Language Bias. In *Thirty-Fifth AAAI Conference on Artificial Intelligence, AAAI 2021*. AAAI Press, 6444–6453. https://ojs.aaai.org/index.php/AAAI/article/view/16799

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

Vijayaraghavan Murali, Letao Qi, Swarat Chaudhuri, and Chris Jermaine. 2018. Neural Sketch Learning for Conditional Program Generation. In *6th International Conference on Learning Representations, ICLR 2018, Vancouver, BC, Canada, April 30 - May 3, 2018, Conference Track Proceedings*. OpenReview.net. https://openreview.net/forum?id=HkfXMz-Ab

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

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

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).

Michael Pradel and Satish Chandra. 2021. Neural Software Analysis. *Commun. ACM* 65, 1 (dec 2021), 86–96. https://doi.org/10.1145/3460348

Veselin Raychev, Martin T. Vechev, and Andreas Krause. 2019. Predicting Program Properties from 'Big Code'. *Commun. ACM* 62, 3 (2019), 99–107. https://doi.org/10.1145/3306204

Malavika Samak and Murali Krishna Ramanathan. 2015. Synthesizing Tests for Detecting Atomicity Violations. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2015, Bergamo, Italy, August 30 - September 4, 2015*, Elisabetta Di Nitto, Mark Harman, and Patrick Heymans (Eds.). ACM, 131–142. https://doi.org/10.1145/2786805.2786874

Malavika Samak, Murali Krishna Ramanathan, and Suresh Jagannathan. 2015. Synthesizing Racy Tests. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, June 15-17, 2015*, David Grove and Stephen M. Blackburn (Eds.). ACM, 175–185. https://doi.org/10.1145/2737924.2737998

Eric Schkufza, Rahul Sharma, and Alex Aiken. 2013. Stochastic Superoptimization. In *Architectural Support for Programming Languages and Operating Systems, ASPLOS 2013, Houston, TX, USA, March 16-20, 2013*, Vivek Sarkar and Rastislav Bodík (Eds.). ACM, 305–316. https://doi.org/10.1145/2451116.2451150

Eric Schkufza, Rahul Sharma, and Alex Aiken. 2016. Stochastic Program Optimization. *Commun. ACM* 59, 2 (jan 2016), 114–122. https://doi.org/10.1145/2863701

Dana Scott. 1976. Data Types as Lattices. *SIAM J. Comput.* 5 (09 1976), 522–587. https://doi.org/10.1137/0205037

Rahul Sharma, Eric Schkufza, Berkeley Churchill, and Alex Aiken. 2013. Data-Driven Equivalence Checking. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages and Applications*. Association for Computing Machinery, New York, NY, USA, 391–406. https://doi.org/10.1145/2509136.2509509

Rahul Sharma, Eric Schkufza, Berkeley Churchill, and Alex Aiken. 2015. Conditionally Correct Superoptimization. *SIGPLAN Not.* 50, 10 (oct 2015), 147–162. https://doi.org/10.1145/2858965.2814278

Armando Solar-Lezama. 2008. *Program Synthesis by Sketching*. Ph.D. Dissertation. University of California, Berkeley.

M. Zalewski. 2015. American Fuzzy Lop.

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 (oct 2021), 29 pages. https://doi.org/10.1145/3485493

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*. 491–507.

A BENCHMARKS AND EXPERIMENT RESULTS

Our benchmark suite and experiment results are available on the following anonymous link:
https://anonymous.4open.science/r/Murphy-Supplementary-Material-3511

B PROOFS OF LEMMAS AND THEOREMS

LEMMA B.1 (PERTURBATION FUNCTORS ARE MONOTONE). *A perturbation functor $f{\uparrow}_\alpha$ built from a buggy input α and perturbation f is always (non-strictly) monotone.*

Proof. In order to show $f{\uparrow}_\alpha$ is monotone, it is sufficient to show:

$$\forall A_1, A_2, A_1 \subseteq A_2 \implies f{\uparrow}_\alpha(A_1) \subseteq f{\uparrow}_\alpha(A_2)$$

According to the definition of $f{\uparrow}_\alpha$:

$$f{\uparrow}^\alpha(A) \triangleq \{\alpha\} \cup \{f(a) \mid a \in A\}$$

and so:

$$\begin{aligned} f{\uparrow}_\alpha\,(A_1) &\triangleq \{\alpha\} \cup \{f(a) \mid a \in A_1\} \\ &\subseteq \{\alpha\} \cup \{f(a) \mid a \in A_2\} \qquad \text{as } (A_1 \subseteq A_2) \\ &\triangleq f{\uparrow}_\alpha\,(A_2) \end{aligned}$$

Qed.

COROLLARY B.2 (PERTURBATION CLOSURE). *For a given instance of the error generalization problem, the least fixed-point of a perturbation functor $f{\uparrow}_\alpha$, denoted as $\mathtt{lfp}(f{\uparrow}_\alpha)$, for a given buggy initial input α exists. Furthermore, if f is sound, $\mathtt{lfp}(f{\uparrow}_\alpha)$ is a generalization of α.*

Proof. First we prove the least fixed-point of a perturbation functor $f{\uparrow}_\alpha$ exists via Scott's fixed-point theorem. The input space of the target program $\mathsf{P} : \tau_1 \times \cdots \times \tau_n \longrightarrow \tau_O$ is no larger than ω, and so the following theorem [Scott 1976] applies: *all Scott-continuous functions F have least fixed point $\bigcup_{i<\omega} F^i(\emptyset)$*. Thus, to show $\mathtt{lfp}(f{\uparrow}_\alpha)$ exists, it is sufficient to show $f{\uparrow}_\alpha$ is Scott-continuous, that is, $f{\uparrow}_\alpha$ is monotone and preserves the all directed supremum. By the above lemma, we know $f{\uparrow}_\alpha$ is monotone. For arbitrary directed supremum $\bigsqcup A_i$ of directed subset $\{A_i\}$ of P_ω:

$$\begin{aligned} f{\uparrow}_\alpha\,(\bigsqcup A_i) &\triangleq \{\alpha\} \cup \{f(a) \mid a \in \bigsqcup A_i\} \\ &= \{\alpha\} \cup \{f(a) \mid a \in \bigcup A_i\} \qquad \text{as } (\{A_i\} \subseteq P_\omega) \\ &= \{\alpha\} \cup \bigcup \{f(a) \mid a \in A_i\} \\ &= \bigcup (\{\alpha\} \cup \{f(a) \mid a \in A_i\}) \\ &\triangleq \bigcup f{\uparrow}_\alpha\,(A_i) \\ &= \bigsqcup f{\uparrow}_\alpha\,(A_i) \qquad \text{as } (\{f{\uparrow}_\alpha\,(A_i)\} \subseteq P_\omega) \end{aligned}$$

Thus $f{\uparrow}_\alpha$ preserves the all directed supremum, and so the $\mathtt{lfp}(f{\uparrow}_\alpha)$ exists.

Scott's fixed pointed theorem additionally states:

$$\mathtt{lfp}(f{\uparrow}_\alpha) = \bigcup_{i<\omega} f{\uparrow}^i_\alpha\,(\emptyset)$$

To show $\mathtt{lfp}(f{\uparrow}_\alpha)$ is a generalization of α when f is sound, according to Definition 3.3, we first prove:

$$\alpha \in \{\alpha\} = f{\uparrow}_\alpha\,(\emptyset) \subseteq \bigcup_{i<\omega} f{\uparrow}^i_\alpha\,(\emptyset) = \mathtt{lfp}(f{\uparrow}_\alpha)$$

Moreover, we prove every element in the fixed point is reachable from α via f by induction. For the initial case, $f{\uparrow_\alpha}(\emptyset) = \{\alpha\}$ is reachable from α; assume $f{\uparrow_\alpha^i}(\emptyset) = \{\alpha\}$ is reachable from α,

$$\begin{aligned} f{\uparrow_\alpha^{i+1}}(\emptyset) &= f{\uparrow_\alpha}(f{\uparrow_\alpha^i}(\emptyset)) \\ &= \{\alpha\} \cup \{f(a) \mid a \in f{\uparrow_\alpha^i}(\emptyset)\} \quad \text{where } f{\uparrow_\alpha^i}(\emptyset) \text{ is reachable} \end{aligned}$$

which is also reachable from α via f. Then every element in the fixed point is reachable from α via f, and is buggy as f is sound. Thus $\mathtt{lfp}(f{\uparrow_\alpha})$ is a generalization of α. **Qed.**

Theorem B.3 (Error Generalization Via Perturbations). *Given an instance of the error generalization problem and a non-empty set of sound perturbations F_α for each α in $A_{\mathtt{init}}$, we can build a generalization of $A_{\mathtt{init}}$ by taking the union of the closures of the perturbations for each buggy input, i.e.* $\bigcup_{\alpha\in A_{\mathtt{init}}} \bigcup_{f\in F_\alpha} \mathtt{lfp}(f{\uparrow_\alpha})$ *is a valid generalization of $A_{\mathtt{init}}$.*

Proof. According to Definition 3.3, we first show $A_{\mathtt{init}}$ is a subset of $\bigcup_{\alpha\in A_{\mathtt{init}}} \bigcup_{f\in F_\alpha} \mathtt{lfp}(f{\uparrow_\alpha})$. According to Corollary B.2,

$$\forall f \in F_\alpha, \alpha \in \mathtt{lfp}(f{\uparrow_\alpha})$$

thus,

$$A_{\mathtt{init}} \subseteq \bigcup_{\alpha\in A_{\mathtt{init}}} \{\alpha\} \subseteq \bigcup_{\alpha\in A_{\mathtt{init}}} \bigcup_{f\in F_\alpha} \mathtt{lfp}(f{\uparrow_\alpha})$$

On the other hand, as $\mathtt{lfp}(f{\uparrow_\alpha})$ is a generalization of α, all elements in it are buggy, and so every element in the union $\bigcup_{\alpha\in A_{\mathtt{init}}} \bigcup_{f\in F_\alpha} \mathtt{lfp}(f{\uparrow_\alpha})$ is also buggy. Therefore, this union is a valid generalization of $A_{\mathtt{init}}$. **Qed.**

Theorem B.4 (Soundness of Jump Proposal). *For an input type τ, set of perturbation operators Θ, and bound n, there exists a finite path between any pair of perturbations in the hypothesis space $Hyp(\tau, \Theta, m)$ via Jump.*

Proof. As we introduced the "constant" operators which do not need any input (e.g. `true : bool`) in Θ, a perturbation that only uses these operators can be a "hub" between two arbitrary perturbations. More precisely, there is a "hub" perturbation f^* that uses constant perturbation operator p for each statement and returns input variables as result. As all perturbations are endo-functions (the input and output types are the same), this f^* is always type safe. We can then perform the following transformation to jump to an arbitrary perturbation f from f^*:

(1) Apply `ReplaceGuard` and `ReplaceOperator` to make the first statement of f^* the same as f. Notice that, as there are no previous operators applied, all variables required by the operators used in the first statement of f can be found in f^*. After that, we apply `ReplaceGuard` and `ReplaceOperator` to make the second statement of f^* the same as f. As now f and the current perturbation has the same first statement, all variables required by the operators used in the second statement of f are also available in current perturbation. By induction, we can repeat these jumps until all statements in the current perturbation the same with f.
(2) Apply `ReArgAssign` to make the variables returned by current perturbation the same as f. As all previous statements of two perturbations are the same, this step is also type safe.

We can reverse the above steps to jump from arbitrary perturbation f' back to f^*. Thus there exists a finite path between any pair of perturbations in the hypothesis space $Hyp(\tau, \Theta, m)$ via Jump. **Qed.**

C ROBUSTNESS

The set of programs considered by Murphy is ultimately defined by the set Θ of perturbations provided to it. In order to synthesize useful perturbations, Θ should be large enough to include meaningful operators that can guide Murphy towards a helpful solution. However, simply adding more operators to Θ increases the size of the search space the learner must navigate, complicating convergence. To investigate how sensitive Murphy is to the selection of perturbation operators (**Q4**), we evaluated how the choice of Θ impacts the percentage of high-quality tests the tool generates. Our evaluation examines three of the ADT benchmarks that use lists for their underlying representation (CustomStk, UniqueL, and SortedL), as lists are the datatype with the highest number of perturbation operators in our experiments (16). For our experimental setup, we limit the number of statements in the perturbation to 4, then select 3 perturbation operators at random[5] to include in Θ. We next run Murphy for 500 MCMC steps[6], and then generate 10 tests. We then randomly choose a new perturbation operator to add to Θ, and repeat the experiment, stopping once all 16 list transformations have been included in Θ.

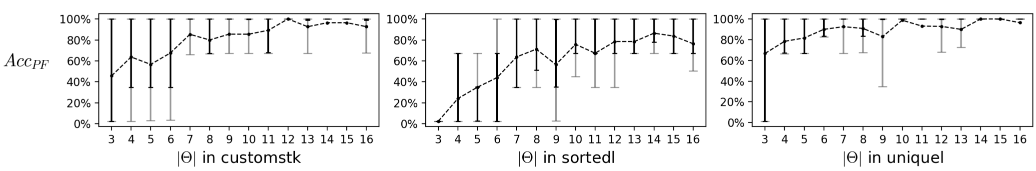

Fig. 14. Experimental results. These figures show the average accuracy of perturbations synthesized within a 500 step bound using different numbers of perturbation operators. In each sub-figure, the x-axis indicates the size of Θ, and the y-axis indicates the percentage of high-quality tests (unique buggy inputs) (Acc_{PF}). The dashed line indicate the average accuracy of all tests, and the black (grey) bar covers the range of accuracy of 50% (80%) tests.

Figure 14 reports the results of this process for each of the three benchmarks, averaged across 8 runs. For each benchmark, we observe that the ratio of high-quality tests (the dashed line) increases rapidly as the number of perturbation operators grows, suggesting that Murphy is able to learn a fairly good (e.g., over 50% accurate) perturbations even when given relatively few (e.g., 7) perturbation operators. With fewer perturbation operators, the quality of the resulting tests can vary wildly, depending on whether the right operators are included. As the size of Θ grows, however, the variance in the accuracy of the learned function decreases. We also observe that that the quality of the perturbation synthesized by Murphy does not decrease much as this set grows large, even when it has to consider the full set of candidate solutions ($O(4^{16})$), suggesting that Murphy is able to focus on the relevant operators when learning a perturbation function. Taken together, these experiments suggest that Murphy is fairly robust to the inclusion of extraneous perturbation operators, even when Θ is relatively large.

[5]We begin with 3 operators to ensure that there are several well-typed candidate solutions in the hypothesis space.
[6]We do not use a time bound because the execution time of the target program is different for each benchmark.