Data-Driven Precondition Inference with Learned Features

Saswat Padhi
Univ. of California, Los Angeles, USA
padhi@cs.ucla.edu

Rahul Sharma
Stanford University, USA
sharmar@cs.stanford.edu

Todd Millstein
Univ. of California, Los Angeles, USA
todd@cs.ucla.edu

Abstract

We extend the data-driven approach to inferring preconditions for code from a set of test executions. Prior work requires a fixed set of *features*, atomic predicates that define the search space of possible preconditions, to be specified in advance. In contrast, we introduce a technique for on-demand *feature learning*, which automatically expands the search space of candidate preconditions in a targeted manner as necessary. We have instantiated our approach in a tool called PIE. In addition to making precondition inference more expressive, we show how to apply our feature-learning technique to the setting of data-driven loop invariant inference. We evaluate our approach by using PIE to infer rich preconditions for black-box OCaml library functions and using our loop-invariant inference algorithm as part of an automatic program verifier for C++ programs.

Categories and Subject Descriptors D.2.1 [*Software Engineering*]: Requirements/Specifications—Tools; D.2.4 [*Software Engineering*]: Software/Program Verification—Validation; F.3.1 [*Theory of Computation*]: Specifying and Verifying and Reasoning about Programs—Invariants, Mechanical verification, Specification techniques

Keywords Precondition Inference, Loop Invariant Inference, Data-driven Invariant Inference

1. Introduction

In this work we extend the data-driven paradigm for *precondition inference*: given a piece of code C along with a predicate Q, the goal is to produce a predicate P whose satisfaction on entry to C is sufficient to ensure that Q holds after C is executed. Data-driven approaches to precondition inference [21, 42] employ a machine learning algorithm to separate a set of "good" test inputs (which cause Q to be satisfied) from a set of "bad" ones (which cause Q to be falsified). Therefore, these techniques are quite general: they can infer candidate preconditions regardless of the complexity of C and Q, which must simply be executable.

A key limitation of data-driven precondition inference, however, is the need to provide the learning algorithm with a set of *features*, which are predicates over the inputs to C (e.g., `x > 0`). The learner then searches for a boolean combination of these features that separates the set G of "good" inputs from the set B of "bad" inputs. Existing data-driven precondition inference approaches [21, 42] require a fixed set of features to be specified in advance. If these features are not sufficient to separate G and B, the approaches must either fail to produce a precondition, produce a precondition that is known to be insufficient (satisfying some "bad" inputs), or produce a precondition that is known to be overly strong (falsifying some "good" inputs).

In contrast, we show how to iteratively *learn* useful features on demand as part of the precondition inference process, thereby eliminating the problem of feature selection. We have implemented our approach in a tool called PIE (Precondition Inference Engine). Suppose that at some point PIE has produced a set F of features that is not sufficient to separate G and B. We observe that in this case there must be at least one pair of tests that *conflict*: the tests have identical valuations to the features in F but one test is in G and the other is in B. Therefore we have a clear criterion for *feature learning*: the goal is to learn a new feature to add to F that resolves a given set of conflicts. PIE employs a form of search-based program synthesis [1, 50, 51] for this purpose, since it can automatically synthesize rich expressions over arbitrary data types. Once all conflicts are resolved in this manner, the boolean learner is guaranteed to produce a precondition that is both sufficient and necessary for the given set of tests.

In addition to making data-driven precondition inference less onerous and more expressive, our approach to feature learning naturally applies to other forms of data-driven invariant inference that employ positive and negative examples. To demonstrate this, we have built a novel data-driven algorithm for inferring provably correct loop invariants. Our algorithm uses PIE as a subroutine to generate candidate invariants, thereby learning features on demand through conflict resolution. In contrast, all prior data-driven loop invariant inference techniques require a fixed set or template of features to be specified in advance [19, 20, 29, 32, 46, 48].

We have implemented PIE for OCaml as well as the loop invariant inference engine based on PIE for C++. We use these implementations to demonstrate and evaluate two distinct uses cases for PIE.[1]

First, PIE can be used in the "black box" setting to aid programmer understanding of third-party code. For example, suppose a programmer wants to understand the conditions under which a given library function throws an exception. PIE can automatically produce a likely precondition for an exception to be thrown, which is guaranteed to be both sufficient and necessary over the set of test inputs that were considered. We evaluate this use case by inferring likely preconditions for the functions in several widely used OCaml libraries. The inferred preconditions match the English documentation in the vast majority of cases and in two cases identify behaviors that are absent from the documentation.

Second, PIE-based loop invariant inference can be used in the "white box" setting, in conjunction with the standard weakest precondition computation [11], to automatically verify that a program meets its specification. We have used our C++ implementation to verify benchmark programs used in the evaluation of three recent approaches to loop invariant inference [13, 20, 46]. These programs require loop invariants involving both linear and non-linear arithmetic as well as operations on strings. The only prior techniques that have demonstrated such generality require a fixed set or template of features to be specified in advance.

The rest of the paper is structured as follows. Section 2 overviews PIE and our loop invariant inference engine informally by example, and Section 3 describes these algorithms precisely. Section 4 presents our experimental evaluation. Section 5 compares with related work, and Section 6 concludes.

2. Overview

This section describes PIE through a running example. The `sub` function in the `String` module of the OCaml standard library takes a string `s` and two integers `i1` and `i2` and returns a substring of the original one. A caller of `sub` must provide appropriate arguments, or else an `Invalid_argument` exception is raised. PIE can be used to automatically infer a predicate that characterizes the set of valid arguments.

Our OCaml implementation of precondition inference using PIE takes three inputs: a function `f` of type `'a -> 'b`; a set T of test inputs of type `'a`, which can be generated using any desired method; and a postcondition Q, which is simply a function of type `'a -> 'b result -> bool`. A `'b result` either has the form `Ok v` where `v` is the result value from the function or `Exn e` where `e` is the exception thrown by the function. By executing `f` on each test input in T to obtain a result and then executing Q on each input-result pair, T is partitioned into a set G of "good" inputs that cause Q to be satisfied and a set B of "bad" inputs that cause Q to be falsified. Finally, PIE is given the sets G and B, with the goal to produce a predicate that separates them.

[1] Our code and full experimental results are available at https://github.com/SaswatPadhi/PIE.

Tests	**Features**				**Set**
	i1<0	i1>0	i2<0	i2>0	
("pie", 0, 0)	F	F	F	F	G
("pie", 0, 1)	F	F	F	T	G
("pie", 1, 0)	F	T	F	F	G
("pie", 1, 1)	F	T	F	T	G
("pie", -1, 0)	T	F	F	F	B
("pie", 1, -1)	F	T	T	F	B
("pie", 1, 3)	F	T	F	T	B
("pie", 2, 2)	F	T	F	T	B

Figure 1: Data-driven precondition inference.

In our running example, the function `f` is `String.sub` and the postcondition Q is the following function:

```
fun arg res ->
    match res with
      Exn (Invalid_argument _) -> false
    | _ -> true
```

As we show in Section 4, when given many random inputs generated by the `qcheck` library[2], PIE-based precondition inference can automatically produce the following precondition for `String.sub` to terminate normally:

```
i1 >= 0 && i2 >= 0 && i1 + i2 <= (length s)
```

Though in this running example the precondition is conjunctive, PIE infers arbitrary conjunctive normal form (CNF) formulas. For example, if the postcondition above is negated, then PIE will produce this complementary condition for when an `Invalid_argument` exception is raised:

```
i1 < 0 || i2 < 0 || i1 + i2 > (length s)
```

2.1 Data-Driven Precondition Inference

This subsection reviews the data-driven approach to precondition inference [21, 42] in the context of PIE. For purposes of our running example, assume that we are given only the eight test inputs for `sub` that are listed in the first column of Figure 1. The induced set G of "good" inputs that cause `String.sub` to terminate normally and set B of "bad" inputs that cause `sub` to raise an exception are shown in the last column of the figure.

Like prior data-driven approaches, PIE separates G and B by reduction to the problem of *learning a boolean formula*

[2] https://github.com/c-cube/qcheck

from examples [21, 42]. This reduction requires a set of *features*, which are predicates on the program inputs that will be used as building blocks for the inferred precondition. As we will see later, PIE's key innovation is the ability to automatically learn features on demand, but PIE also accepts an optional initial set of features to use.

Suppose that PIE is given the four features shown along the top of Figure 1. Then each test input induces a *feature vector* of boolean values that results from evaluating each feature on that input. For example, the first test induces the feature vector `<F,F,F,F>`. Each feature vector is now interpreted as an assignment to a set of four boolean variables, and the goal is to learn a propositional formula over these variables that satisfies all feature vectors from G and falsifies all feature vectors from B.

There are many algorithms for learning boolean formulas by example. PIE uses a simple but effective *probably approximately correct* (PAC) algorithm that can learn an arbitrary conjunctive normal form (CNF) formula and is biased toward small formulas [31]. The resulting precondition is guaranteed to be both sufficient and necessary for the given test inputs, but there are no guarantees for other inputs.

2.2 Feature Learning via Program Synthesis

At this point in our running example, we have a problem: there is no boolean function on the current set of features that is consistent with the given examples! This situation occurs exactly when two test inputs *conflict*: they induce identical feature vectors, but one test is in G while the other is in B. For example, in Figure 1 the tests `("pie",1,1)` and `("pie",1,3)` conflict; therefore no boolean function over the given features can distinguish between them.

Prior data-driven approaches to precondition inference require a fixed set of features to be specified in advance. Therefore, whenever two tests conflict they must produce a precondition that violates at least one test. The approach of Sankaranarayanan *et al.* [42] learns a decision tree using the ID3 algorithm [40], which minimizes the total number of misclassified tests. The approach of Gehr *et al.* [21] strives to produce sufficient preconditions and so returns a precondition that falsifies all of the "bad" tests while minimizing the total number of misclassified "good" tests.

In our running example, both prior approaches will produce a predicate equivalent to the following one, which misclassifies one "good" test:

```
!(i1 < 0) && !(i2 < 0)
          && !((i1 > 0) && (i2 > 0))
```

This precondition captures the actual lower-bound requirements on `i1` and `i2`. However, it includes an upper-bound requirement that is both overly restrictive, requiring at least one of `i1` and `i2` to be zero, and insufficient (for some unobserved inputs), since it is satisfied by erroneous inputs such as `("pie",0,5)`. Further, using more tests does not help. On a test suite with full coverage of the possible "good" and "bad" feature vectors, an approach that falsifies all "bad" tests must require both `i1` and `i2` to be zero, obtaining sufficiency but ruling out almost all "good" inputs. The ID3 algorithm will produce a decision tree that is larger than the original one, due to the need for more case splits over the features, and this tree will be either overly restrictive, insufficient, or both.

In contrast to these approaches, we have developed a form of automatic *feature learning*, which augments the set of features in a targeted manner on demand. The key idea is to leverage the fact that we have a clear criterion for selecting new features – they must resolve conflicts. Therefore, PIE first generates new features to resolve any conflicts, and it then uses the approach described in Section 2.1 to produce a precondition that is consistent with all tests.

Let a *conflict group* be a set of tests that induce the same feature vector and that participate in a conflict (i.e., at least one test is in G and one is in B). PIE's feature learner uses a form of search-based program synthesis [1, 16] to generate a feature that resolves all conflicts in a given conflict group. Given a set of constants and operations for each type of data in the tests, the feature learner enumerates candidate boolean expressions in order of increasing size until it finds one that separates the "good" and "bad" tests in the given conflict group. The feature learner is invoked repeatedly until all conflicts are resolved.

In Figure 1, three tests induce the same feature vector and participate in a conflict. Therefore, the feature learner is given these three input-output examples: `(("pie",1,1), T)`, `(("pie",1,3), F)`, and `(("pie",2,2), F)`. Various predicates are consistent with these examples, including the "right" one `i1 + i2 <= (length s)` and less useful ones like `i1 + i2 != 4`. However, overly specific predicates are less likely to resolve a conflict group that is sufficiently large; the small conflict group in our example is due to the use of only eight test inputs. Further, existing synthesis engines bias against such predicates by assigning constants a larger "size" than variables [1].

PIE with feature learning is *strongly convergent*: if there exists a predicate that separates G and B and is expressible in terms of the constants and operations given to the feature learner, then PIE will eventually (ignoring resource limitations) find such a predicate. PIE's search space is limited to predicates that are expressible in the "grammar" given to the feature learner. However, each type typically has a standard set of associated operations, which can be provided once and reused across many invocations of PIE. For each such invocation, feature learning automatically searches an unbounded space of expressions in order to produce targeted features. For example, the feature `i1 + i2 <= (length s)` for `String.sub` in our running example is automatically constructed from the operations `+` and `<=` on integers and `length` on strings, obviating the need for users to manually craft this feature in advance.

```
string sub(string s, int i1, int i2) {
  assume(i1 >= 0 && i2 >= 0 &&
         i1+i2 <= s.length());
  int i = i1;
  string r = "";
  while (i < i1+i2) {
    assert(i >= 0 && i < s.length());
    r = r + s.at(i);
    i = i + 1;
  }
  return r;
}
```

Figure 2: A C++ implementation of `sub`.

Our approach to feature learning could itself be used to perform precondition inference in place of PIE, given all tests rather than only those that participate in a conflict. However, we demonstrate in Section 4 that our separation of feature learning and boolean learning is critical for scalability. The search space for feature learning is exponential in the maximum feature size, so attempting to synthesize entire preconditions can quickly hit resource limitations. PIE avoids this problem by decomposing precondition inference into two subproblems: generating rich features over arbitrary data types and generating a rich boolean structure over a fixed set of black-box features.

2.3 Feature Learning for Loop Invariant Inference

Our approach to feature learning also applies to other forms of data-driven invariant inference that employ positive and negative examples, and hence can have conflicts. To illustrate this, we have built a novel algorithm called LOOPINVGEN for inferring loop invariants that are sufficient to prove that a program meets its specification. The algorithm employs PIE as a subroutine, thereby learning features on demand as described above. In contrast, all prior data-driven loop invariant inference techniques require a fixed set or template of features to be specified in advance [19, 20, 29, 32, 46, 48].

To continue our running example, suppose that we have inferred a likely precondition for the `sub` function to execute without error and want to verify its correctness for the C++ implementation of `sub` shown in Figure 2.[3] As is standard, we use the function `assume(`P`)` to encode the precondition; executions that do not satisfy P are silently ignored. We would like to automatically prove that the assertion inside the `while` loop never fails (which implies that the subsequent access `s.at(i)` is within bounds). However, doing so requires an appropriate loop invariant to be inferred, which involves both integer and string operations. To our knowledge, the only previous technique that has been demonstrated to infer such invariants employs a random search over a fixed set of features [46].

[3] Note that + is overloaded as both addition and string concatenation in C++.

In contrast, our algorithm LOOPINVGEN can infer an appropriate loop invariant without being given any features as input. The algorithm is inspired by the HOLA loop invariant inference engine, a purely static analysis that employs logical abduction via quantifier elimination to generate candidate invariants [13]. Our approach is similar but does not require the logic of invariants to support quantifier elimination and instead leverages PIE to generate candidates. HOLA's abduction engine generates multiple candidates, and HOLA performs a backtracking search over them. PIE instead generates a single precondition, but we show how to iteratively augment the set of tests given to PIE in order to refine its result. We have implemented LOOPINVGEN for C++ programs.

The LOOPINVGEN algorithm has three main components. First, we build a program verifier V for *loop-free* programs in the standard way: given a piece of code C along with a precondition P and postcondition Q, V generates the formula $P \Rightarrow \mathit{WP}(C, Q)$, where *WP* denotes the weakest precondition [11]. The verifier then checks validity of this formula by querying an SMT solver that supports the necessary logical theories, which either indicates validity or provides a counterexample.

Second, we use PIE and the verifier V to build an algorithm VPREGEN for generating provably sufficient preconditions for loop-free programs, via counterexample-driven refinement [5]. Given code C, a postcondition Q, and test sets G and B, VPREGEN invokes PIE on G and B to generate a candidate precondition P. If the verifier V can prove the sufficiency of P for C and Q, then we are done. Otherwise, the counterexample from the verifier is incorporated as a new test in the set B, and the process iterates. PIE's feature learning automatically expands the search space of preconditions whenever a new test creates a conflict.

Finally, the LOOPINVGEN algorithm iteratively invokes VPREGEN to produce candidate loop invariants until it finds one that is sufficient to verify the given program. We illustrate LOOPINVGEN in our running example, where the inferred loop invariant $I(i, i_1, i_2, r, s)$ must satisfy the following three properties:

1. The invariant should hold when the loop is first entered:

$$\begin{aligned}(i_1 \geq 0 \wedge i_2 \geq 0 \wedge i_1 + i_2 \leq s.\mathit{length}()\\ \wedge\ i = i_1 \wedge r = \text{``''} \Rightarrow I(i, i_1, i_2, r, s)\end{aligned}$$

2. The invariant should be inductive:

$$I(i, i_1, i_2, r, s) \wedge i < i_1 + i_2 \Rightarrow I(i+1, i_1, i_2, r+s.at(i), s)$$

3. The invariant should be strong enough to prove the assertion:

$$I(i, i_1, i_2, r, s) \wedge i < i_1 + i_2 \Rightarrow 0 \leq i < s.\mathit{length}()$$

Our example involves both linear arithmetic and string operations, so the program verifier V must use an SMT solver that supports both theories, such as Z3-Str2 [52] or CVC4 [35].

To generate an invariant satisfying the above properties, LOOPINVGEN first asks VPREGEN to find a precondition to ensure that the assertion will not fail in the following program, which represents the third constraint above:

```
assume(i < i1 + i2);
assert(0 <= i && i < s.length());
```

Given a sufficiently large set of test inputs, VPREGEN generates the following precondition, which is simply a restatement of the assertion itself:

```
0 <= i && i < s.length()
```

While this *candidate* invariant is guaranteed to satisfy the third constraint, an SMT solver can show that it is not inductive. We therefore use VPREGEN again to iteratively strengthen the candidate invariant until it is inductive. For example, in the first iteration, we ask VPREGEN to infer a precondition to ensure that the assertion will not fail in the following program:

```
assume(0 <= i && i < s.length());
assume(i < i1 + i2);
r = r + s.at(i);
i = i+1;
assert(0 <= i && i < s.length());
```

This program corresponds to the second constraint above, but with I replaced by our current candidate invariant. VPREGEN generates the precondition `i1+i2 <= s.length()` for this program, which we conjoin to the current candidate invariant to obtain a new candidate invariant:

```
0 <= i && i < s.length() && i1+i2 <= s.length()
```

This candidate is inductive, so the iteration stops.

Finally, we ask the verifier if our candidate satisfies the first constraint above. In this case it does, so we have found a valid loop invariant and thereby proven that the code's assertion will never fail. If instead the verifier provides a counterexample, then we incorporate this as a new test input and restart the entire process of finding a loop invariant.

3. Algorithms

In this section we describe our data-driven precondition inference and loop invariant inference algorithms in more detail.

3.1 Precondition Inference

Figure 3 presents the algorithm for precondition generation using PIE, which we call PREGEN. We are given a code snippet C, which is assumed not to make any internal non-deterministic choices, and a postcondition Q, such as an assertion. We are also given a set of test inputs T for C,

```
PREGEN($C$: Code, $Q$: Predicate, $T$: Tests) : Predicate
Returns: A precondition that is consistent with all tests in $T$
1: Tests $G, B$ := PARTITIONTESTS($C$,$Q$,$T$)
2: return PIE($G$,$B$)
```

Figure 3: Precondition generation.

```
PIE($G$: Tests, $B$: Tests) : Predicate
Returns: A predicate $P$ such that $P(t)$ for all $t \in G$ and $\neg P(t)$ for all $t \in B$
1: Features $F := \emptyset$
2: repeat
3:     FeatureVectors $V^+$ := CREATEFV($F$,$G$)
4:     FeatureVectors $V^-$ := CREATEFV($F$,$B$)
5:     Conflict $X$ := GETCONFLICT($V^+, V^-, G, B$)
6:     if $X \neq$ None then
7:         $F := F \cup$ FEATURELEARN($X$)
8:     end if
9: until $X =$ None
10: $\phi$ := BOOLLEARN($V^+, V^-$)
11: return SUBSTITUTE($F$, $\phi$)
```

Figure 4: The PIE algorithm.

which can be generated by any means, for example a fuzzer, a symbolic execution engine, or manually written unit tests. The goal is to infer a precondition P such that the execution of C results in a state satisfying Q if and only if it begins from a state satisfying P. In other words, we would like to infer the weakest predicate P that satisfies the Hoare triple $\{P\}C\{Q\}$. Our algorithm guarantees that P will be both sufficient and necessary on the given set of tests T but makes no guarantees for other inputs.

The function PARTITIONTESTS in Figure 3 executes the tests in T in order to partition them into a sequence G of "good" tests, which cause C to terminate in a state that satisfies Q, and a sequence B of "bad" tests, which cause C to terminate in a state that falsifies Q (line 1). The precondition is then obtained by invoking PIE, which is discussed next.

Figure 4 describes the overall structure of PIE, which returns a predicate that is consistent with the given set of tests. The initial set F of features is empty, though our implementation optionally accepts an initial set of features from the user (not shown in the figure). For example, such features could be generated based on the types of the input data, the branch conditions in the code, or by leveraging some knowledge of the domain.

Regardless, PIE then iteratively performs the loop on lines 2-9. First it creates a *feature vector* for each test in G and B (lines 3 and 4). The i^{th} element of the sequence V^+ is a sequence that stores the valuation of the features on the i^{th} test in G. More formally,

$$V^+ = \text{CREATEFV}(F, G) \iff \forall i, j. (V_i^+)_j = F_j(G_i)$$

Here we use the notation S_k to denote the k^{th} element of the sequence S, and $F_j(G_i)$ denotes the boolean result of evaluating feature F_j on test G_i. V^- is created in an analogous manner given the set B.

We say that a feature vector v is a *conflict* if it appears in both V^+ and V^-, i.e. $\exists i, j.V_i^+ = V_j^- = v$. The function GETCONFLICT returns `None` if there are no conflicts. Otherwise it selects one conflicting feature vector v and returns a pair of sets $X = (X^+, X^-)$, where X^+ is a subset of G whose associated feature vector is v and X^- is a subset of B whose associated feature vector is v. Next PIE invokes the feature learner on X, which uses a form of program synthesis to produce a new feature f such that $\forall t \in X^+.f(t)$ and $\forall t \in X^-.\neg f(t)$. This new feature is added to the set F of features, thus resolving the conflict.

The above process iterates, identifying and resolving conflicts until there are no more. PIE then invokes the function BOOLLEARN, which learns a propositional formula ϕ over $|F|$ variables such that $\forall v \in V^+.\phi(v)$ and $\forall v \in V^-.\neg\phi(v)$. Finally, the precondition is created by substituting each feature for its corresponding boolean variable in ϕ.

Discussion Before describing the algorithms for feature learning and boolean learning, we note some important aspects of the overall algorithm. First, like prior data-driven approaches, PREGEN and PIE are very general. The only requirement on the code C in Figure 3 is that it be executable, in order to partition T into the sets G and B. The code itself is not even an argument to the function PIE. Therefore, PREGEN can infer preconditions for any code, regardless of how complex it is. For example, the code can use idioms that are hard for automated constraint solvers to analyze, such as non-linear arithmetic, intricate heap structures with complex sharing patterns, reflection, and native code. Indeed, the source code itself need not even be available. The postcondition Q similarly must simply be executable and so can be arbitrarily complex.

Second, PIE can be viewed as a hybrid of two forms of precondition inference. Prior data-driven approaches to precondition inference [21, 42] perform boolean learning but lack feature learning, which limits their expressiveness and accuracy. On the other hand, a feature learner based on program synthesis [1, 50, 51] can itself be used as a precondition inference engine without boolean learning, but the search space grows exponentially with the size of the required precondition. PIE uses feature learning only to resolve conflicts, leveraging the ability of program synthesis to generate expressive features over arbitrary data types, and then uses boolean learning to scalably infer a concise boolean structure over these features.

Due to this hybrid nature of PIE, a key parameter in the algorithm is the maximum number c of conflicting tests to allow in the conflict group X at line 5 in Figure 4. If the conflict groups are too large, then too much burden is placed on the feature learner, which limits scalability. For example, a degenerate case is when the set of features is empty, in which case all tests induce the empty feature vector and are in conflict. Therefore, if the set of conflicting tests that induce the same feature vector has a size greater than c, we choose a random subset of size c to provide to the feature learner. We empirically evaluate different values for c in our experiments in Section 4.

```
FEATURELEARN($X^+$: Tests, $X^-$: Tests) : Predicate
Returns: A feature $f$ such that $f(t)$ for all $t \in X^+$ and $\neg f(t)$ for all $t \in X^-$
1: Operations $O$ := GETOPERATIONS()
2: Integer $i$ := 1
3: loop
4:     Features $F$ := FEATURESOFSIZE($i$, $O$)
5:     if $\exists f \in F.(\forall t \in X^+.f(t) \wedge \forall t \in X^-.\neg f(t))$ then
6:         return $f$
7:     end if
8:     $i$ := $i$ + 1
9: end loop
```

Figure 5: The feature learning algorithm.

Feature Learning Figure 5 describes our approach to feature learning. The algorithm is a simplified version of the Escher program synthesis tool [1], which produces functional programs from examples. Like Escher, we require a set of *operations* for each type of input data, which are used as building blocks for synthesized features. By default, FEATURELEARN includes operations for primitive types as well as for lists. For example, integer operations include `0` (a nullary operation), `+`, and `<=`, while list operations include `[]`, `::`, and `length`. Users can easily add their own operations, for these as well as other types of data.

Given this set of operations, FEATURELEARN simply enumerates all possible features in order of the size of their abstract syntax trees. Before generating features of size $i+1$, it checks whether any feature of size i completely separates the tests in X^+ and X^-; if so, that feature is returned. The process can fail to find an appropriate feature, either because no such feature over the given operations exists or because resource limitations are reached; either way, this causes the PIE algorithm to fail.

Despite the simplicity of this algorithm, it works well in practice, as we show in Section 4. Enumerative synthesis is a good match for learning features, since it biases toward small features, which are likely to be more general than large features and so helps to prevent against overfitting. Further, the search space is significantly smaller than that of traditional program synthesis tasks, since features are simple expressions rather than arbitrary programs. For example, our algorithm does not attempt to infer control structures such as conditionals, loops, and recursion, which is a technical focus of much program-synthesis research [1, 16].

```
BOOLLEARN($V^+$: Feature Vectors, $V^-$: Feature Vectors) : Boolean Formula
Returns: A formula $\phi$ such that $\phi(v)$ for all $v \in V^+$ and $\neg\phi(v)$ for all $v \in V^-$
1: Integer $n$ := size of each feature vector in $V^+$ and $V^-$
2: Integer $k$ := 1
3: loop
4:     Clauses $C$ := ALLCLAUSESUPTOSIZE($k$, $n$)
5:     $C$ := FILTERINCONSISTENTCLAUSES($C$, $V^+$)
6:     $C$ := GREEDYSETCOVER($C$, $V^-$)
7:     if $C \neq$ None then
8:         return $C$
9:     end if
10:    $k$ := $k$ + 1
11: end loop
```

Figure 6: The boolean function learning algorithm.

Boolean Function Learning We employ a standard algorithm for learning a small CNF formula that is consistent with a given set of boolean feature vectors [31]; it is described in Figure 6. Recall that a CNF formula is a conjunction of *clauses*, each of which is a disjunction of *literals*. A literal is either a propositional variable or its negation. Our algorithm returns a CNF formula over a set $x_1, \ldots, x_n$ of propositional variables, where n is the size of each feature vector (line 1). The algorithm first attempts to produce a 1-CNF formula (i.e., a conjunction), and it increments the maximum clause size k iteratively until a formula is found that is consistent with all feature vectors. Since BOOLLEARN is only invoked once all conflicts have been removed (see Figure 4), this process is guaranteed to succeed eventually.

Given a particular value of k, the learning algorithm first generates a set C of all clauses of size k or smaller over $x_1, \ldots, x_n$ (line 4), implicitly representing the conjunction of these clauses. In line 5, all clauses that are inconsistent with at least one of the "good" feature vectors (i.e., the vectors in V^+) are removed from C. A clause c is inconsistent with a "good" feature vector v if v falsifies c:

$$\forall 1 \leq i \leq n.(x_i \in c \Rightarrow v_i = \texttt{false}) \wedge (\neg x_i \in c \Rightarrow v_i = \texttt{true})$$

After line 5, C represents the strongest k-CNF formula that is consistent with all "good" feature vectors.

Finally, line 6 weakens C while still falsifying all of the "bad" feature vectors (i.e., the vectors in V^-). In particular, the goal is to identify a minimal subset C' of C where for each $v \in V^-$, there exists $c \in C'$ such that v falsifies c. This problem is equivalent to the classic *minimum set cover* problem, which is NP-complete. Therefore, our GREEDYSETCOVER function on line 6 uses a standard heuristic for that problem, iteratively selecting the clause that is falsified by the most "bad" feature vectors that remain, until all such feature vectors are "covered." This process will fail to cover all "bad" feature vectors if there is no k-CNF formula consistent with V^+ and V^-, in which case k is incremented; otherwise the resulting set C is returned as our CNF formula.

Because the boolean learner treats features as black boxes, this algorithm is unaffected by their sizes. Rather, the search space is O(n^k), where n is the number of features and k is the maximum clause size, and in practice k is a small constant. Though we have found this algorithm to work well in practice, there are many other algorithms for learning boolean functions from examples. As long as they can learn arbitrary boolean formulas, then we expect that they would also suffice for our purposes.

```
VPREGEN($C$: Code, $Q$: Predicate, $G$: Tests) : Predicate
Returns: A precondition $P$ such that $P(t)$ for all $t$ in $G$ and $\{P\}C\{Q\}$ holds
1: Tests $B$ := $\emptyset$
2: repeat
3:     $P$ := PIE($G$,$B$)
4:     $t$ := VERIFY($P$,$C$,$Q$)
5:     $B$ := $B \cup \{t\}$
6: until $t$ = None
7: return $P$
```

Figure 7: Verified precondition generation for loop-free code.

Properties As described above, the precondition returned by PIE is guaranteed to be both necessary and sufficient for the given set of test inputs. Furthermore, PIE is *strongly convergent*: if there exists a predicate that separates G and B and is expressible in terms of the constants and operations given to the feature learner, then PIE will eventually (ignoring resource limitations) find and return such a predicate.

To see why PIE is strongly convergent, note that FEATURELEARN (Figure 5) performs an exhaustive enumeration of possible features. By assumption a predicate that separates G and B is expressible in the language of the feature learner, and that predicate also separates any sets X^+ and X^- of conflicting tests, since they are respectively subsets of G and B. Therefore each call to FEATURELEARN on line 7 in Figure 4 will eventually succeed, reducing the number of conflicting tests and ensuring that the loop at line 2 eventually terminates. At that point, there are no more conflicts, so there is some CNF formula over the features in F that separates G and B, and the boolean learner will eventually find it.

3.2 Loop Invariant Inference

As described in Section 2.3, our loop invariant inference engine relies on an algorithm VPREGEN that generates provably sufficient preconditions for loop-free code. The VPREGEN algorithm is shown in Figure 7. In the context of loop invariant inference (see below), VPREGEN will always be passed a set of "good" tests to use and will start

```
LOOPINVGEN(C: Code, T: Tests) : Predicate
Returns: A loop invariant that is sufficient to verify that C's assertion never fails.
Require: $C = \texttt{assume } P; \texttt{while } E\ \{C_1\}; \texttt{assert } Q$
 1: $G$ := LOOPHEADSTATES($C, T$)
 2: loop
 3:     $I$ := VPREGEN( [assume $\neg E$], $Q$, $G$)
 4:     while not $\{I \wedge E\}\, C_1 \{I\}$ do
 5:         $I'$ := VPREGEN( [assume $I \wedge E$; $C_1$], $I$, $G$)
 6:         $I := I \wedge I'$
 7:     end while
 8:     $t$ := VALID($P \Rightarrow I$)
 9:     if $t = \texttt{None}$ then
10:         return $I$
11:     else
12:         $G := G \cup$ LOOPHEADSTATES($C, \{t\}$)
13:     end if
14: end loop
```

Figure 8: Loop invariant inference using PIE.

with no "bad" tests, so we specialize the algorithm to that setting. The VPREGEN algorithm assumes the existence of a verifier for loop-free programs. If the verifier can prove the sufficiency of a candidate precondition P generated by PIE (lines 3-4), it returns None and we are done. Otherwise the verifier returns a counterexample t, which has the property that $P(t)$ is true but executing *C* on t ends in a state that falsifies Q. Therefore we add t to the set *B* of "bad" tests and iterate.

The LOOPINVGEN algorithm for loop invariant inference is shown in Figure 8. For simplicity, we restrict the presentation to code snippets of the form

$$C = \texttt{assume } P; \texttt{while } E\ \{C_1\}; \texttt{assert } Q$$

where C_1 is loop-free. Our implementation also handles code with multiple and nested loops, by iteratively inferring invariants for each loop encountered in a backward traversal of the program's control-flow graph.

The goal of LOOPINVGEN is to infer a loop invariant I which is sufficient to prove that the Hoare triple $\{P\}(\texttt{while } E\{C_1\})\{Q\}$ is valid. In other words, we must find an invariant I that satisfies the following three constraints:

$$\begin{array}{ccc} P & \Rightarrow & I \\ \{I \wedge E\} & C_1 & \{I\} \\ I \wedge \neg E & \Rightarrow & Q \end{array}$$

Given a test suite T for C, LOOPINVGEN first generates a set of tests for the loop by logging the program state every time the loop head is reached (line 1). In other words, if $\vec{x}$ denotes the set of program variables then we execute the following instrumented version of C on each test in T:

$$\texttt{assume } P; \texttt{log } \vec{x}; \texttt{while } E\ \{C_1; \texttt{log } \vec{x}\}; \texttt{assert } Q$$

If the Hoare triple $\{P\}(\texttt{while } E\{C_1\})\{Q\}$ is valid, then all test executions are guaranteed to pass the assertion, so all logged program states will belong to the set G of passing tests. If a test fails the assertion then no valid loop invariant exists so we abort (not shown in the figure).

With this new set *G* of tests, LOOPINVGEN first generates a candidate invariant that meets the third constraint above by invoking VPREGEN on line 3. The inner loop (lines 4-7) then strengthens I until the second constraint is met. If the generated candidate also satisfies the first constraint (line 8), then we have found an invariant. Otherwise we obtain a counterexample t satisfying $P \wedge \neg I$, which we use to collect new program states as additional tests (line 12), and the process iterates. The verifier for loop-free code is used on lines 3 (inside VPREGEN), 4 (to check the Hoare triple), and 5 (inside VPREGEN), and the underlying SMT solver is used on line 8 (the validity check).

We note the interplay of strengthening and weakening in the LOOPINVGEN algorithm. Each iteration of the inner loop strengthens the candidate invariant until it is inductive. However, each iteration of the outer loop uses a larger set G of passing tests. Because PIE is guaranteed to return a precondition that is consistent with all tests, the larger set G has the effect of weakening the candidate invariant. In other words, candidates get strengthened, but if they become stronger than P in the process then they will be weakened in the next iteration of the outer loop.

Properties Both the VPREGEN and LOOPINVGEN algorithms are *sound*: VPREGEN(*C*, Q, G) returns a precondition P such that $\{P\}C\{Q\}$ holds, and LOOPINVGEN(*C*, T) returns a loop invariant I that is sufficient to prove that $\{P\}(\texttt{while } E\ \{C_1\})\{Q\}$ holds, where $C = \texttt{assume } P; \texttt{while } E\ \{C_1\}; \texttt{assert } Q$. However, neither algorithm is guaranteed to return the weakest such predicate.

VPREGEN(*C*, *Q*, *G*) is *strongly convergent*: if there exists a precondition P that is expressible in the language of the feature learner such that $\{P\}C\{Q\}$ holds and $P(t)$ holds for each $t \in G$, then VPREGEN will eventually find such a precondition.

To see why, first note that by assumption each test in G satisfies P, and since $\{P\}C\{Q\}$ holds, each test that will be put in B at line 5 in Figure 7 falsifies P (since each such test causes Q to be falsified). Therefore P is a separator for G and B, so each call to PIE at line 3 terminates due to the strong convergence result described earlier. Suppose P has size s. Then each call to PIE from VPREGEN will generate features of size at most s, since P itself is a valid separator for any set of conflicts. Further, each call to PIE produces a logically distinct precondition candidate, since each call includes a new test in B that is inconsistent with the previous candidate. Since the feature learner has a finite number of operations for each type of data, there are a finite number of features of size at most s and so also a finite number of logically distinct boolean functions in terms of such features.

Hence eventually P or another sufficient precondition will be found.

LoopInvGen is not strongly convergent: it can fail to terminate even when an expressible loop invariant exists. First, the iterative strengthening loop (lines 4-7 of Figure 8) can generate a VPreGen query that has no expressible solution, causing VPreGen to diverge. Second, an adversarial sequence of counterexamples from the SMT solver (line 9 of Figure 8) can cause LoopInvGen's outer loop to diverge. Nonetheless, our experimental results below indicate that the algorithm performs well in practice.

4. Evaluation

We have evaluated PIE's ability to infer preconditions for black-box OCaml functions and LoopInvGen's ability to infer sufficient loop invariants for verifying C++ programs.

4.1 Precondition Inference

Experimental Setup We have implemented the PreGen algorithm described in Figure 3 in OCaml. We use PreGen to infer preconditions for all of the first-order functions in three OCaml modules: `List` and `String` from the standard library, and `BatAvlTree` from the widely used `batteries` library[4]. Our test generator and feature learner do not handle higher-order functions. For each function, we generate preconditions under which it raises an exception. Further, for functions that return a list, string, or tree, we generate preconditions under which the result value is empty when it returns normally. Similarly, for functions that return an integer (boolean) we generate preconditions under which the result value is `0` (`false`) when the function returns normally. A recent study finds that roughly 75% of manually written specifications are predicates like these, which relate to the presence or absence of data [43].

For feature learning we use a simplified version of the Escher program synthesis tool [1] that follows the algorithm described in Figure 5. Escher already supports operations on primitive types and lists; we augment it with operations for strings (e.g., `get`, `has`, `sub`) and AVL trees (e.g., `left_branch`, `right_branch`, `height`). For the set T of tests, we generate random inputs of the right type using the `qcheck` OCaml library. Analogous to the *small scope hypothesis* [28], which says that "small inputs" can expose a high proportion of program errors, we find that generating many random tests over a small domain exposes a wide range of program behaviors. For our tests we generate random integers in the range $[-4, 4]$, lists of length at most 5, trees of height at most 5 and strings of length at most 12.

In total we attempt to infer preconditions for 101 function-postcondition pairs. Each attempt starts with no initial features and is allowed to run for at most one hour and use up to 8GB of memory. Two key parameters to our algorithm are the number of tests to use and the maximum size of conflict groups to provide the feature learner. Empirically we have found 6400 tests and conflict groups of maximum size 16 to provide good results (see below for an evaluation of other values of these parameters).

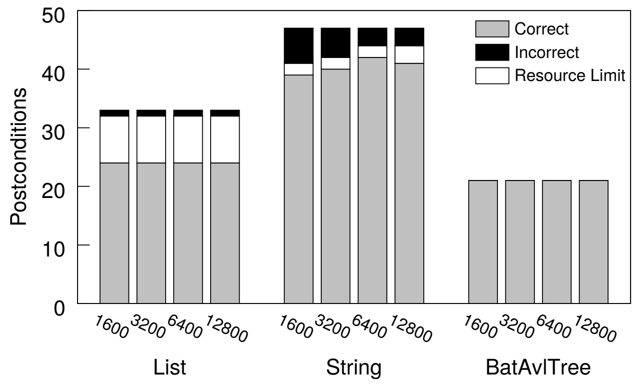
[4] `http://batteries.forge.ocamlcore.org`

Table 1: A sample of inferred preconditions for OCaml library functions.

Case	Postcondition	Learned Features
`String` module functions		
`set`(s, i, c)	throws exception	3
	$(i < 0) \lor (\texttt{len}(s) \leq i)$	
`sub`(s, i_1, i_2)	throws exception	3
	$(i_1 < 0) \lor (i_2 < 0) \lor (i_1 > \texttt{len}(s) - i_2)$	
`index`(s, c)	result $= 0$	2
	$\texttt{has}(\texttt{get}(s, 0), c)$	
`index_from`(s, i, c)	throws exception	4
	$(i < 0) \lor (i > \texttt{len}(s)) \lor \neg\, \texttt{has}(\texttt{sub}(s, i, \texttt{len}(s) - i), c)$	
`List` module functions		
`nth`$(l,\ n)$	throws exception	2
	$(0 > n) \lor (n \geq \texttt{len}(l))$	
`append`$(l_1,\ l_2)$	`empty`(result)	2
	$\texttt{empty}(l_1) \land \texttt{empty}(l_2)$	
`BatAvlTree` module functions		
`create` $(t_1,\ v,\ t_2)$	throws exception	6
	$\texttt{height}(t_1) > (\texttt{height}(t_2) + 1) \lor \texttt{height}(t_2) > (\texttt{height}(t_1) + 1)$	
`concat`$(t_1,\ t_2)$	`empty`(result)	2
	$\texttt{empty}(t_1) \land \texttt{empty}(t_2)$	

Results Under the configuration described above, PreGen generates correct preconditions in 87 out of 101 cases. By "correct" we mean that the precondition fully matches the English documentation, and possibly captures actual behaviors not reflected in that documentation. The latter happens for two `BatAvlTree` functions: the documentation does not mention that `split_leftmost` and `split_rightmost` will raise an exception when passed an empty tree.

Table 1 shows some of the more interesting preconditions that PreGen inferred, along with the number of synthesized features for each. For example, it infers an accurate precondition for `String.index_from`(s,i,c), which returns the index of the first occurrence of character c in string s after position i, through a rich boolean combination of arithmetic and string functions. As another example, PreGen automatically discovers the definition of a balanced tree, since `BatAvlTree.create` throws an exception if the resulting tree would not be balanced. Prior approaches to precondition inference [21, 42] can only capture these preconditions if they are provided with exactly the right features (e.g.,



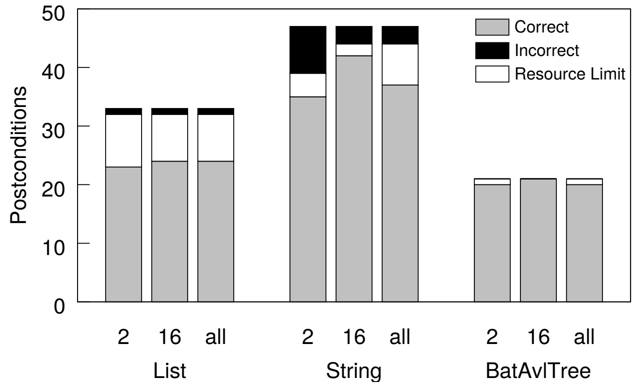


Figure 9: Comparison of PIE configurations. The top plot shows the effect of different numbers of tests. The bottom plot shows the effect of different conflict group sizes.

$\texttt{height}(t_1) > (\texttt{height}(t_2) + 1)$) in advance, while PRE-GEN learns the necessary features on demand.

The 14 cases that either failed due to time or memory limits or that produce an incorrect or incomplete precondition were of three main types. The majority (10 out of 14) require universally quantified features, which are not supported by our feature learner. For example, `List.flatten`(l) returns an empty list when each of the inner lists of l is empty. In a few cases the inferred precondition is incomplete due to our use of small integers as test inputs. For example, we do not infer that `String.make`(i,c) throws an exception if i is greater than `Sys.max_string_length`. Finally, a few cases produce erroneous specifications for list functions that employ physical equality, such as `List.memq`. Our tests for lists only use primitives as elements, so they cannot distinguish physical from structural equality.

Configuration Parameter Sensitivity We also evaluated PIE's sensitivity to the number of tests and the maximum conflict group size. The top plot in Figure 9 shows the results with varied numbers of tests (and conflict group size of 16). In general, the more tests we use, the more correct our results. However, with 12,800 tests we incur one additional case that hits resource limits due to the extra overhead involved.

Table 2: Comparison of PIE with an approach that uses eager feature learning. The size of a feature is the number of nodes in its abstract syntax tree. Each Q_i indicates the i^{th} quartile, computed independently for each column.

	Size of Features	Number of Features	
		EAGER	PIE
`Min`	2	13	1
Q_1	3	29.25	1
Q_2	4	55	1
Q_3	5.25	541.50	2
`Max`	13	18051	5
`Mean`	4.54	1611.80	1.50
`SDev`	2.65	4055.50	0.92

The bottom plot in Figure 9 shows the results with varied conflict group sizes (and 6400 tests). On the one hand, we can give the feature learner only a single pair of conflicting tests at a time. As the figure shows, this leads to more cases hitting resource limits and producing incorrect results versus a conflict group size of 16, due to the higher likelihood of synthesizing overly specific features. On the other hand, we can give the feature learner all conflicting tests at once. When starting with no initial features, all tests are in conflict, so this strategy requires the feature learner to synthesize the entire precondition. As the figure shows, this approach hits resource limitations more often versus a conflict group size of 16. For example, this approach fails to generate the preconditions for `String.index_from` and `BatAvlTree.create` shown in Table 1. Further, in the cases that do succeed, the average running time and memory consumption are 11.7 second and 309 MB, as compared to only 1.8 seconds and 66 MB when the conflict group size is 16.

Comparison With Eager Feature Learning PIE generates features *lazily* as necessary to resolve conflicts. An alternative approach is to use Escher up front to eagerly generate every feature for a given program up to some maximum feature size s. These features can then simply all be passed to the boolean learner. To evaluate this approach, we instrumented PIE to count the number of candidate features that were generated by Escher each time it was called.[5] For each call to PIE, the maximum such number across all calls to Escher is a lower bound, and therefore a best-case scenario, for the number of features that would need to be passed to the boolean learner in an eager approach. It's a lower bound for two reasons. First, we are assuming that the user can correctly guess the maximum size s of features to generate in order to produce a precondition that separates the "good" and "bad" tests. Second, Escher stops generating features as soon as it finds one that resolves the given conflicts, so in

[5] All candidates generated by Escher are both type-correct and exception-free, i.e. they do not throw exceptions on any test inputs.

general there will be many features of size s that are not counted.

Table 2 shows the results for the 52 cases in our experiment above where PIE produces a correct answer and at least one feature is generated. Notably, the minimum number of features generated in the eager approach (13) is more than double the maximum number of features selected in our approach (5). Nonetheless, for functions that require only simple preconditions, eager feature learning is reasonably practical. For example, 25% of the preconditions (`Min` to Q_1 in the table) require 29 or fewer features. However, the number of features generated by eager feature learning grows exponentially with their maximum size. For example, the top 25% of preconditions (from Q_3 to `Max`) require a minimum of 541 features to be generated and a maximum of more than $18,000$. Since boolean learning is in general super-linear in the number n of features (the algorithm we use is $O(n^k)$ where k is the maximum clause size), we expect an eager approach to hit resource limits as the preconditions become more complex.

4.2 Loop Invariants for C++ Code

We have implemented the loop invariant inference procedure described in Figure 8 for C++ code as a Clang tool[6]. As mentioned earlier, our implementation supports multiple and nested loops. We have also implemented a verifier for loop-free programs using the CVC4 [35] and Z3-Str2 [52] SMT solvers, which support several logical theories including both linear and non-linear arithmetic and strings. We employ both solvers because their support for both non-linear arithmetic and strings is incomplete, causing some queries to fail to terminate. We therefore run both solvers in parallel for two minutes and fail if neither returns a result in that time.

We use the same implementation and configuration for PIE as in the previous experiment. To generate the tests we employ an initial set of 256 random inputs of the right type. As described in Section 3.2, the algorithm then captures the values of all variables whenever control reaches the loop head, and we retain at most 6400 of these states.

We evaluate our loop invariant inference engine on multiple sets of benchmarks; the results are shown in Table 3 and a sample of the inferred invariants is shown in Table 4. First, we have used LoopInvGen on all 46 of the benchmarks that were used to evaluate the HOLA loop invariant engine [13]. These benchmarks require loop invariants that involve only the theory of linear arithmetic. Table 3 shows each benchmark's name from the original benchmark set, the number of calls to the SMT solvers, the number of calls to the feature learner, the size of the generated invariant, and the running time of LoopInvGen in seconds. LoopInvGen succeeds in inferring invariants for 43 out of 46 HOLA benchmarks, including three benchmarks which HOLA's technique cannot handle (cases 15, 19, and 34). By construction, these invariants are sufficient to ensure the correctness of the assertions in these benchmarks. The three cases on which LoopInvGen fails run out of memory during PIE's CNF learning phase.

Second, we have used LoopInvGen on 39 of the benchmarks that were used to evaluate the ICE loop invariant engine [19, 20]. The remaining 19 of their benchmarks cannot be evaluated with LoopInvGen because they use language features that our program verifier does not support, notably arrays and recursion. As shown in Table 3, we succeed in inferring invariants for 35 out of the 36 ICE benchmarks that require linear arithmetic. LoopInvGen infers the invariants fully automatically and with no initial features, while ICE requires a fixed template of features to be specified in advance. The one failing case is due to a limitation of the current implementation — we treat boolean values as integers, which causes PIE to consider many irrelevant features for such values.

We also evaluated LoopInvGen on the three ICE benchmarks whose invariants require non-linear arithmetic. Doing so simply required us to allow the feature learner to generate non-linear features; such features were disabled for the above tests due to the SMT solvers' limited abilities to reason about non-linear arithmetic. LoopInvGen was able to generate sufficient loop invariants to verify two out of the three benchmarks. Our approach fails on the third benchmark because both SMT solvers fail to terminate on a particular query. However, this is a limitation of the solvers rather than of our approach; indeed, if we vary the conflict-group size, which leads to different SMT queries, then our tool can succeed on this benchmark.

Third, we have evaluated our approach on the four benchmarks whose invariants require both arithmetic and string operations that were used to evaluate another recent loop invariant inference engine [46]. As shown in Table 3, our approach infers loop invariants for all of these benchmarks. The prior approach [46] requires both a fixed set of features and a fixed boolean structure for the desired invariants, neither of which is required by our approach.

Finally, we ran all of the above experiments again, but with PIE replaced by our program-synthesis-based feature learner. This version succeeds for only 61 out of the 89 benchmarks. Further, for the successful cases, the average running time is 567 seconds and 1895 MB of memory, versus 28 seconds and 128 MB (with 573 MB peak memory usage) for our PIE-based approach.

5. Related Work

We compare our work against three forms of specification inference in the literature. First, there are several prior approaches to inferring preconditions given a piece of code and a postcondition. Closest to PIE are the prior data-driven approaches. Sankaranarayanan *et al.* [42] uses a decision-tree

[6] `http://clang.llvm.org/docs/LibTooling.html`

Table 3: Experimental results for LoopInvGen. An invariant's size is the number of nodes in its abstract syntax tree. The analysis time is in seconds.

Case	Calls to Solvers	Calls to Escher	Sizes of Invariants	Analysis Time
HOLA benchmarks [13]				
01	7	3	11	21
02	43	17	15	27
03	31	22	3,7,15	46
04	4	2	7	18
05	7	3	11	23
06	51	26	9,9	54
07	111	45	19	116
08	4	2	7	18
09	27	18	3,15,7,22	40
10	14	7	28	21
11	21	18	15	22
12	33	19	13,15	45
13	46	30	33	54
14	9	9	31	22
15	26	27	33	39
16	9	10	11	22
17	22	17	15,7	31
18	6	5	15	20
19	18	14	27	32
20	61	24	33	115
21	23	10	19	23
22	16	11	13	22
23	10	7	11	21
24	29	19	1,7,11	40
25	83	47	11,19	142
26	90	32	9,9,9	71
27	32	20	7,3,7	44
28	7	2	3,3	20
29	66	19	11,11	47
30	18	12	35	29
31	-	-	-	-
32	-	-	-	-
33	-	-	-	-
34	30	20	37	25
35	5	4	11	18
36	128	36	11,15,19,11	113
37	13	11	19	22
38	44	38	29	36
39	10	5	11	20
40	30	24	19,17	40
41	17	11	15	27
42	25	15	50	37
43	4	2	7	19
44	14	14	20	26
45	60	33	11,9,9	64
46	12	5	21	24

Case	Calls to Solvers	Calls to Escher	Sizes of Invariants	Analysis Time
Linear arithmetic benchmarks from ICE [20]				
afnp	12	7	11	22
cegar1	16	11	12	30
cegar2	13	11	19	23
cggmp	34	22	143	32
countud	6	4	9	17
dec	4	2	3	17
dillig01	7	3	11	19
dillig03	14	7	15	29
dillig05	7	3	11	21
dillig07	8	4	11	21
dillig12	32	18	13,11	44
dillig15	31	35	55	42
dillig17	24	22	19,11	31
dillig19	19	13	31	32
dillig24	29	18	1,3,11	40
dillig25	57	31	11,19	74
dillig28	7	2	3,3	19
dtuc	9	2	3,3	22
fig1	4	2	7	17
fig3	7	5	7	17
fig9	7	2	7	19
formula22	12	11	16	25
formula25	11	5	15	21
formula27	23	5	19	25
inc2	4	2	7	18
inc	4	2	7	17
loops	19	12	7,7	28
sum1	16	14	21	22
sum3	6	1	3	20
sum4c	41	21	38	32
sum4	6	3	9	17
tacas6	9	8	11	22
trex1	6	2	7,1	19
trex3	9	7	7	23
w1	4	2	7	17
w2	-	-	-	-
Non-linear arithmetic benchmarks from ICE [20]				
multiply	25	19	15	41
sqrt	-	-	-	-
square	11	7	5	24
String benchmarks [46]				
a	66	22	110	45
b	6	5	4	10
c	7	4	8	11
d	7	3	9	11

Table 4: A sample of inferred invariants for C++ benchmarks.

`(HOLA) 07`	$I : (b = 3i - a) \land (n > i \lor b = 3n - a)$
`(HOLA) 22`	$I : (k = 3y) \land (x = y) \land (x = z)$
`(ICE linear) dillig12`	$I_1 : (a = b) \land (t = 2s \lor \texttt{flag} = 0)$ $I_2 : (x \leq 2) \land (y < 5)$
`(ICE linear) sum1`	$I : (i = sn + 1) \land (sn = 0 \lor sn = n \lor n \geq i)$
`(Strings) c`	$I : \mathsf{has}(r, \text{“}a\text{”}) \land (\mathsf{len}(r) > i)$
`(ICE non-linear) multiply`	$I : (s = y * j) \land (x > j \lor s = x * y)$

learner to infer preconditions from good and bad examples. Gehr *et al.* also uses a form of boolean learning from examples, in order to infer conditions under which two functions commute [21]. As discussed in Section 2, the key innovation of PIE over these works is its support for on-demand feature learning, instead of requiring a fixed set of features to be specified in advance. In addition to eliminating the problem of feature selection, PIE's feature learning ensures that the produced precondition is both sufficient and necessary for the given set of tests, which is not guaranteed by the prior approaches.

There are also several static approaches to precondition inference. These techniques can provide provably sufficient (or provably necessary [10]) preconditions. However, unlike data-driven approaches, they all require the source code to be available and statically analyzable. The standard weakest precondition computation infers preconditions for loop-free programs [11]. For programs with loops, a backward symbolic analysis with search heuristics can yield preconditions [3, 8]. Other approaches leverage properties of particular language paradigms [23], require logical theories that support quantifier elimination [12, 37], and employ counterexample-guided abstraction refinement (CEGAR) with domain-specific refinement heuristics [44, 45]. Finally, some static approaches to precondition inference target specific program properties, such as predicates about the heap structure [2, 34] or about function equivalence [30].

Second, we have shown how PIE can be used to build a novel data-driven algorithm LOOPINVGEN for inferring loop invariants that are sufficient to prove that a program meets its specification. Several prior data-driven approaches exist for this problem [18–20, 29, 32, 33, 46–49]. As above, the key distinguishing feature of LOOPINVGEN relative to this work is its support for feature learning. Other than one exception [47], which uses support vector machines (SVMs) [7] to learn new numerical features, all prior works employ a fixed set or template of features. In addition, some prior approaches can only infer restricted forms of boolean formulas [46–49], while LOOPINVGEN learns arbitrary CNF formulas. Finally, the ICE approach [19] requires a set of “implication counterexamples” in addition to good and bad examples, which necessitates new algorithms for learning boolean formulas [20]. In contrast, LOOPINVGEN can employ any off-the-shelf boolean learner. Unlike LOOPINVGEN, ICE is strongly convergent [19]: it restricts invariant inference to a finite set of candidate invariants that is iteratively enlarged using a dovetailing strategy that eventually covers the entire search space.

There are also many static approaches to invariant inference. The HOLA [13] loop invariant generator is based on an algorithm for logical abduction [12]; we employed a similar technique to turn PIE into a loop invariant generator. HOLA requires the underlying logic of invariants to support quantifier elimination, while LOOPINVGEN has no such restriction. Standard invariant generation tools that are based on abstract interpretation [8, 9], constraint solving [6, 27], or probabilistic inference [25] require the number of disjunctions to be specified manually. Other approaches [15, 17, 22, 24, 26, 36, 41] can handle disjunctions but restrict their number via trace-based heuristics, custom built abstract domains, or widening. In contrast, LOOPINVGEN places no *a priori* bound on the number of disjunctions.

Third, there has been prior work on data-driven inference of specifications given only a piece of code as input. For example, Daikon [14] generates likely invariants at various points within a given program. Other work leverages Daikon to generate candidate specifications and then uses an automatic program verifier to validate them, eliminating the ones that are not provable [38, 39, 43]. As above, these approaches employ a fixed set or template of features. Unlike precondition inference and loop invariant inference, which require more information from the programmer (e.g., a postcondition), general invariant inference has no particular goal and so no notion of “good” and “bad” examples. Hence these approaches cannot obtain counterexamples to refine candidate invariants and cannot use our conflict-based approach to learn features.

Finally, the work of Cheung *et al.* [4], like PIE, combines machine learning and program synthesis, but for a very different purpose: to provide event recommendations to users of social media. They use the SKETCH system [50] to generate a set of recommendation functions that each classify all test inputs, and then they employ SVMs to produce a linear combination of these functions. PIE instead uses program synthesis for feature learning, and only as necessary to resolve conflicts, and then it uses machine learning to infer boolean combinations of these features that classify all test inputs.

6. Conclusion

We have described PIE, which extends the data-driven paradigm for precondition inference to automatically learn features on demand. The key idea is to employ a form of program synthesis to produce new features whenever the current set of features cannot exactly separate the "good" and "bad" tests. Feature learning removes the need for users to manually select features in advance, and it ensures that PIE produces preconditions that are both sufficient and necessary for the given set of tests. We also described LOOPINVGEN, which leverages PIE to provide automatic feature learning for data-driven loop invariant inference. Our experimental results indicate that PIE can infer high-quality preconditions for black-box code and LOOPINVGEN can infer sufficient loop invariants for program verification across a range of logical theories.

Acknowledgments

Thanks to Miryung Kim, Sorin Lerner, Madan Musuvathi, Guy Van den Broeck, and the anonymous reviewers for helpful feedback on this paper and research; Sumit Gulwani and Zachary Kincaid for access to the Escher program synthesis tool; Isil Dillig for access to the HOLA benchmarks; and Yang He for extensions to the PIE implementation. This research was supported by the National Science Foundation under award CCF-1527923 and by a Microsoft fellowship.

References

[1] A. Albarghouthi, S. Gulwani, and Z. Kincaid. Recursive program synthesis. In *Computer Aided Verification - 25th International Conference*, pages 934–950, 2013.

[2] C. Calcagno, D. Distefano, P. W. O'Hearn, and H. Yang. Compositional shape analysis by means of bi-abduction. *Journal of the ACM*, 58(6), 2011.

[3] S. Chandra, S. J. Fink, and M. Sridharan. Snugglebug: A powerful approach to weakest preconditions. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 363–374, 2009.

[4] A. Cheung, A. Solar-Lezama, and S. Madden. Using program synthesis for social recommendations. In *21st ACM International Conference on Information and Knowledge Management*, pages 1732–1736, 2012.

[5] E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement. In *Computer Aided Verification - 12th International Conference*, pages 154–169, 2000.

[6] M. Colón, S. Sankaranarayanan, and H. Sipma. Linear invariant generation using non-linear constraint solving. In *Computer Aided Verification - 15th International Conference*, pages 420–432, 2003.

[7] C. Cortes and V. Vapnik. Support-vector networks. *Machine Learning*, 20(3):273–297, 1995.

[8] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Fourth ACM Symposium on Principles of Programming Languages*, pages 238–252, 1977.

[9] P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. In *Fifth Annual ACM Symposium on Principles of Programming Languages*, pages 84–96, 1978.

[10] P. Cousot, R. Cousot, M. Fähndrich, and F. Logozzo. Automatic inference of necessary preconditions. In *Verification, Model Checking, and Abstract Interpretation - 14th International Conference*, pages 128–148, 2013.

[11] E. W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, Englewood Cliffs, New Jersey, 1976.

[12] I. Dillig and T. Dillig. Explain: A tool for performing abductive inference. In *Computer Aided Verification - 25th International Conference*, pages 684–689. Springer, 2013.

[13] I. Dillig, T. Dillig, B. Li, and K. L. McMillan. Inductive invariant generation via abductive inference. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object-Oriented Programming Systems Languages & Applications*, pages 443–456, 2013.

[14] M. D. Ernst, J. H. Perkins, P. J. Guo, S. McCamant, C. Pacheco, M. S. Tschantz, and C. Xiao. The daikon system for dynamic detection of likely invariants. *Sci. Comput. Program.*, 69(1-3):35–45, 2007.

[15] M. Fähndrich and F. Logozzo. Static contract checking with abstract interpretation. In *Formal Verification of Object-Oriented Software - International Conference*, pages 10–30, 2010.

[16] J. K. Feser, S. Chaudhuri, and I. Dillig. Synthesizing data structure transformations from input-output examples. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 229–239, 2015.

[17] G. Filé and F. Ranzato. Improving abstract interpretations by systematic lifting to the powerset. In *Logic Programming, Proceedings of the 1994 International Symposium*, pages 655–669, 1994.

[18] J. P. Galeotti, C. A. Furia, E. May, G. Fraser, and A. Zeller. Dynamate: Dynamically inferring loop invariants for automatic full functional verification. In *Hardware and Software: Verification and Testing - 10th International Haifa Verification Conference*, pages 48–53, 2014.

[19] P. Garg, C. Löding, P. Madhusudan, and D. Neider. ICE: A robust framework for learning invariants. In *Computer Aided Verification - 26th International Conference*, pages 69–87, 2014.

[20] P. Garg, D. Neider, P. Madhusudan, and D. Roth. Learning invariants using decision trees and implication counterexamples. In *Proceedings of the 43rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 499–512, 2016.

[21] T. Gehr, D. Dimitrov, and M. T. Vechev. Learning commutativity specifications. In *Computer Aided Verification - 27th International Conference*, pages 307–323, 2015.

[22] K. Ghorbal, F. Ivancic, G. Balakrishnan, N. Maeda, and A. Gupta. Donut domains: Efficient non-convex domains for

abstract interpretation. In *Verification, Model Checking, and Abstract Interpretation - 13th International Conference*, pages 235–250, 2012.

[23] R. Giacobazzi. Abductive analysis of modular logic programs. *Journal of Logic and Computation*, 8(4):457–483, 1998.

[24] D. Gopan and T. W. Reps. Guided static analysis. In *Static Analysis, 14th International Symposium*, pages 349–365, 2007.

[25] S. Gulwani and N. Jojic. Program verification as probabilistic inference. In *Proceedings of the 34th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 277–289, 2007.

[26] S. Gulwani, S. Srivastava, and R. Venkatesan. Program analysis as constraint solving. In *Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation*, pages 281–292, 2008.

[27] A. Gupta, R. Majumdar, and A. Rybalchenko. From tests to proofs. In *Tools and Algorithms for the Construction and Analysis of Systems, 15th International Conference*, pages 262–276, 2009.

[28] D. Jackson. *Software Abstractions: Logic, Language, and Analysis*. MIT Press, 2006.

[29] Y. Jung, S. Kong, B. Wang, and K. Yi. Deriving invariants by algorithmic learning, decision procedures, and predicate abstraction. In *Verification, Model Checking, and Abstract Interpretation, 11th International Conference*, pages 180–196, 2010.

[30] M. Kawaguchi, S. K. Lahiri, and H. Rebelo. Conditional equivalence. Technical Report MSR-TR-2010-119, Microsoft Research, October 2010.

[31] M. J. Kearns and U. V. Vazirani. *An Introduction to Computational Learning Theory*. The MIT Press, Cambridge, Massachusetts, 1994.

[32] S. Kong, Y. Jung, C. David, B. Wang, and K. Yi. Automatically inferring quantified loop invariants by algorithmic learning from simple templates. In *Programming Languages and Systems - 8th Asian Symposium*, pages 328–343, 2010.

[33] S. Krishna, C. Puhrsch, and T. Wies. Learning invariants using decision trees. *CoRR*, abs/1501.04725, 2015.

[34] T. Lev-Ami, M. Sagiv, T. Reps, and S. Gulwani. Backward analysis for inferring quantified preconditions. Technical Report TR-2007-12-01, Tel Aviv University, 2007.

[35] T. Liang, A. Reynolds, C. Tinelli, C. Barrett, and M. Deters. A DPLL(T) theory solver for a theory of strings and regular expressions. In *Computer Aided Verification - 26th International Conference*, pages 646–662, 2014.

[36] L. Mauborgne and X. Rival. Trace partitioning in abstract interpretation based static analyzers. In *European Symposium on Programming*, 2005.

[37] Y. Moy. Sufficient preconditions for modular assertion checking. In *Verification, Model Checking, and Abstract Interpretation, 9th International Conference*, pages 188–202, 2008.

[38] J. W. Nimmer and M. D. Ernst. Automatic generation of program specifications. In *Proceedings of the International Symposium on Software Testing and Analysis*, pages 229–239, 2002.

[39] J. W. Nimmer and M. D. Ernst. Invariant inference for static checking:. In *Proceedings of the 10th ACM SIGSOFT Symposium on Foundations of Software Engineering*, pages 11–20, 2002.

[40] J. R. Quinlan. Induction of decision trees. *Machine Learning*, 1(1):81–106, 1986.

[41] S. Sankaranarayanan, F. Ivancic, I. Shlyakhter, and A. Gupta. Static analysis in disjunctive numerical domains. In *Static Analysis Symposium*, pages 3–17, 2006.

[42] S. Sankaranarayanan, S. Chaudhuri, F. Ivancic, and A. Gupta. Dynamic inference of likely data preconditions over predicates by tree learning. In *ACM/SIGSOFT International Symposium on Software Testing and Analysis*, pages 295–306, 2008.

[43] T. W. Schiller, K. Donohue, F. Coward, and M. D. Ernst. Case studies and tools for contract specifications. In *Proceedings of the 36th International Conference on Software Engineering*, pages 596–607, 2014.

[44] M. N. Seghir and D. Kroening. Counterexample-guided precondition inference. In *22nd European Symposium on Programming*, pages 451–471, 2013.

[45] M. N. Seghir and P. Schrammel. Necessary and sufficient preconditions via eager abstraction. In *Programming Languages and Systems - 12th Asian Symposium*, pages 236–254, 2014.

[46] R. Sharma and A. Aiken. From invariant checking to invariant inference using randomized search. In *Computer Aided Verification - 26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18-22, 2014. Proceedings*, pages 88–105, 2014.

[47] R. Sharma, A. V. Nori, and A. Aiken. Interpolants as classifiers. In *Computer Aided Verification - 24th International Conference*, pages 71–87, 2012.

[48] R. Sharma, S. Gupta, B. Hariharan, A. Aiken, and A. V. Nori. Verification as learning geometric concepts. In *Static Analysis - 20th International Symposium*, pages 388–411, 2013.

[49] R. Sharma, E. Schkufza, B. R. Churchill, and A. Aiken. Conditionally correct superoptimization. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object-Oriented Programming Systems Languages & Applications*, pages 147–162, 2015.

[50] A. Solar-Lezama, L. Tancau, R. Bodik, S. Seshia, and V. Saraswat. Combinatorial sketching for finite programs. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 404–415, 2006.

[51] S. Srivastava, S. Gulwani, and J. S. Foster. From program verification to program synthesis. In *37th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 313–326, 2010.

[52] Y. Zheng, X. Zhang, and V. Ganesh. Z3-str: a z3-based string solver for web application analysis. In *Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 114–124, 2013.