Data-Driven Precondition Inference with Learned Features

(Submitted to PLDI 2016)

Saswat Padhi
UCLA
padhi@cs.ucla.edu

Rahul Sharma
Stanford University
sharmar@cs.stanford.edu

Todd Millstein
UCLA
todd@cs.ucla.edu

Abstract

We present a new black-box approach to inferring likely preconditions for code. Like prior data-driven approaches, our approach employs a form of machine learning to infer a precondition that is consistent with a set of test executions. However, prior approaches require 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 in the black-box setting, we also show how to use PIE as the basis for a novel and general algorithm for inferring loop invariants when the code is available and analyzable by an automatic constraint solver. We demonstrate the capabilities of PIE by using it to infer rich preconditions for black-box OCaml library functions as well as verified loop invariants for C++ programs.

1. Introduction

Many program verification and software engineering tasks require a form of *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. The standard *weakest precondition* computation provides a solution for loop-free programs that is both sufficient and necessary [14]. Several techniques have been proposed to infer sufficient preconditions for arbitrary programs, for example in the context of particular language paradigms [28], particular kinds of logical theories [15, 45], and particular classes of program properties [5, 42].

However, none of these approaches is applicable if the code is either not available or is too complex to be analyzed by an automated constraint solver (for example due to non-linear arithmetic, subtle heap manipulations, etc.). In that setting, it is natural to employ a *data-driven* style, which uses a form of machine learning to infer likely preconditions from a set of test executions [26, 51]. The power in the data-driven style comes from its generality. The program is treated as a "black box" and simply must be executable, so its code can be arbitrarily complex. Further, data-driven inference typically employs a generic learning algorithm for separating a set of "good" test inputs from a set of "bad" ones, so the approach is applicable regardless of the complexity of the postcondition Q used to create these sets.

A key limitation of data-driven precondition inference, however, is the need to provide the learning algorithm with a set of *features*, atomic predicates that define the search space of possible preconditions. The existing data-driven precondition inference techniques [26, 51] require a fixed set of features to be specified in advance. In this paper we show how to extend the data-driven paradigm for precondition inference to iteratively *learn* features in a targeted manner, thereby eliminating the need to specify features up front and making precondition inference more expressive. We have implemented our approach in a tool called PIE (Precondition Inference Engine).

Given code C and postcondition Q, PIE executes C on a set of tests in order to partition that set into the "good" tests G that satisfy Q and the "bad" tests B that falsify Q. Suppose that at some point PIE has produced a set F of features. Similar to prior approaches, it then uses an algorithm for learning boolean formulas to infer a boolean combination P of features in F that separates G and B. However, it is possible for no such predicate P to exist. In that case, an approach with a fixed set of features 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).

Instead, we observe that a separating predicate P does not exist exactly when there is 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. We employ a form of search-based program synthesis [2, 59, 60] for this purpose, since it can automatically synthesize rich features over arbitrary datatypes. Once all conflicts are resolved in this manner, the boolean formula learner is guaranteed to produce a precondition P that is both sufficient and necessary for the given set of tests.

PIE's automatic feature learning removes the burden for users to determine the "right" set of features in advance,

and it makes the resulting preconditions more expressive and more likely to be accurate. In addition to these benefits in the "black box" setting, PIE also enables a novel approach to program verification in the setting where the code is available and analyzable by an automated constraint solver. Specifically, we have used PIE as the basis for a new algorithm for inferring provably sound loop invariants, which is an independent contribution due to its generality. For example, unlike prior loop invariant inference engines, our approach does not restrict the underlying logic of the constraint solver [16], restrict the structure of inferred invariants (e.g., the number of disjunctions) [9, 12, 13, 30, 32, 55], or require a fixed set of features to be specified in advance [24, 25, 36, 39, 55, 57].

We have implemented PIE for OCaml as well as a loop-invariant inference engine based on PIE for C++.[1] We demonstrate their capabilities in two sets of experiments. First, we have used PIE in the black-box setting to infer 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 also identified behaviors for two functions that were not mentioned in the documentation. Second, we demonstrate the generality of our PIE-based loop invariant inference engine by using it to infer loop invariants for two sets of programs that were used in the evaluation of recent approaches [16, 55]. The only prior technique that has been shown to handle the latter benchmarks [55], whose invariants combine linear arithmetic with string functions, requires both a fixed set of features and a fixed boolean structure for the desired invariant to be specified in advance.

In summary, this paper describes several contributions:

- PIE, the first data-driven approach to precondition inference that does not require features to be specified in advance.
- A novel and general approach to inferring loop invariants based on PIE.
- An implementation and experimental evaluation of PIE for both black-box precondition inference and sound loop invariant inference. We know of no other approach that has been demonstrated to be useful in both settings.

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

2. Overview

This section describes our approach to precondition inference in 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 an `Invalid_argument` exception is raised. PIE can be used to automatically infer a predicate that characterizes the set of valid arguments.

[1] Implementation URL omitted for double-blind reviewing.

Tests	Features			
	`i1 < 0`	`i1 > 0`	`i2 < 0`	`i2 > 0`
`("pie", 0, 0)`	F	F	F	F
`("pie", 0, 1)`	F	F	F	T
`("pie", 1, 0)`	F	T	F	F
`("pie", 1, 1)`	F	T	F	T
`("pie", -1, 0)`	T	F	F	F
`("pie", 1, -1)`	F	T	T	F
`("pie", 1, 3)`	F	T	F	T
`("pie", 2, 2)`	F	T	F	T

Figure 1. Reducing precondition inference to boolean function learning.

Our OCaml implementation of precondition inference takes three inputs: an OCaml function of type `'a -> 'b`; a set of test inputs of type `'a`, which can be generated using any desired method; and a postcondition, which is simply an OCaml 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. In our running example, we simply provide the `sub` function along with some test data and the following postcondition:

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

As we show in Section 4, when given many random inputs generated by OCaml's `qcheck` library[2], PIE 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 Precondition Inference via Boolean Function Learning

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. By executing `sub` on each test input to obtain a result and then executing the above postcondition function on each input-result pair, we can partition the tests into a set G of inputs that cause `sub` to terminate normally and a set B of inputs that cause `sub` to raise an exception. In

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

the figure those tests above the horizontal line constitute the set G while those below the line constitute the set B. The goal of PIE is to learn a predicate that is satisfied by all tests in G but falsified by all tests in B.

Similar to prior data-driven approaches, PIE uses the sets G and B to produce a likely precondition by reduction to the problem of *learning a boolean formula from examples* [26, 51]. This reduction requires a set of *features*, which are atomic predicates that will be used as building blocks for the inferred precondition. As we will see below, a key innovation of PIE is the ability to automatically learn features, but PIE also accepts an optional initial set of features to use, thereby allowing the user to leverage his domain knowledge to guide the search for a useful precondition. For our example, assume that we are either given or have already generated the four features shown along the top of Figure 1.

Figure 1 illustrates the reduction to boolean function learning. Each test input induces a *feature vector*, a 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 from examples. Our PIE implementation 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 [38]. By substituting the original features for the boolean variables in the resulting logical formula, we obtain a likely precondition. This 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.* [51] learns a decision tree using the ID3 algorithm [48], which minimizes the total number of misclassified tests. The approach of Gehr *et al.* [26] 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, the approach of Gehr *et al.* must require both `i1` and `i2` to be zero in order to falsify all "bad" tests, 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 likely 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 [2, 21] to generate a feature that resolves all conflicts in a given conflict group. Given a set of base 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 set. The feature learner is invoked repeatedly until all conflicts are resolved.

In our running example of 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 set that is sufficiently large; the small conflict set 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 [2].

Observe that 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 datatypes, and generating a rich boolean structure over a fixed set of black-box features.

2.3 Loop Invariant Inference

In addition to its benefits in the black-box setting, PIE enables a novel approach to program verification in the setting where the code and postcondition are available and analyzable by an automatic constraint solver (e.g., an SMT solver). Specifically, we have used PIE as the basis for a general algorithm for inferring provably sound loop invariants. This 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, we simply ask the constraint solver whether the formula $P \Rightarrow \mathit{WP}(C, Q)$ is valid, where *WP* denotes the weakest precondition [14]. The solver 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 [8]. Given code C, a postcondition Q, and test sets G and B, VPREGEN invokes PIE 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. Note that PIE's automatic feature learning is a key enabler of this process, since feature learning is triggered whenever the addition of a test creates a conflict, thereby automatically expanding the search space. In contrast, an approach that uses a fixed set of features would fail whenever a sufficient precondition cannot be expressed with those features.

Finally, we show how to employ VPREGEN to build a loop invariant inference engine that we call LOOPINVGEN. Our approach is inspired by a prior approach to loop invariant inference that is based on an algorithm for logical abduction [16]. That approach relies on the abduction engine to generate multiple sufficient preconditions and performs a backtracking search over these candidates. Our VPREGEN algorithm generates a single sufficient precondition, but we show how to iteratively augment the set of tests given to VPREGEN in order to perform a targeted search. We have implemented LOOPINVGEN for C++ programs.

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 prove that the assertion inside the `while` loop never fails (which implies that the subsequent access `s[i]` is within bounds). However, doing so with an automatic program verifier requires an appropriate loop invariant. Specifically, the loop invariant $I(i, i_1, i_2, r, s)$ must satisfy the following three properties:

```
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[i];
    i = i + 1;
  }
  return r;
}
```

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

1. The invariant should hold when the loop is first entered:
$$(i_1 \geq 0 \wedge i_2 \geq 0 \wedge i_1 + i_2 \leq s.\mathit{length}() \wedge\ i = i_1 \wedge r = \text{“”} \Rightarrow I(i, i_1, i_2, r, s)$$
2. The invariant should be inductive:
$$I(i, i_1, i_2, r, s) \wedge i < i_1 + i_2 \Rightarrow I(i+1, i_1, i_2, r + s[i], s)$$
3. The invariant should be strong enough to prove the assertion:
$$I(i, i_1, i_2, r, s) \wedge i < i_1 + i_2 \Rightarrow 0 \leq i < s.\mathit{length}()$$

Producing an invariant that meets the above requirements requires the ability to reason about both integer and string operations. To our knowledge, the only previous technique that can infer such invariants does so via a random search over a fixed set of manually provided features and requires a manually provided template for the invariant's logical structure [55]. In contrast, LOOPINVGEN can generate a correct invariant without either of these manual inputs. For the verifier, we employ an SMT solver that has a theory of strings, such as Z3str2 [62] or CVC4 [43].

To generate I, 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 sets of tests G and B that satisfy and falsify the assertion, VPREGEN generates the following precondition, which is simply a restatement of the assertion itself:

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

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

PREGEN(C: Code, Q: Predicate, T: Tests) : Predicate
Returns: A candidate precondition

1: Tests G, B := PARTITIONTESTS(C,Q,T)
2: **return** PIE(G,B)

Figure 3. Precondition generation.

While this *candidate* invariant satisfies 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 the following program:

```
assume(0 <= i && i < s.length());
assume(i < i1 + i2);
r = r + s[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. Data-Driven Precondition Inference

In this section, we present PIE's approach in more detail. After describing the overall PIE algorithm along with its components for feature learning and boolean learning, we describe how to use PIE to infer sound loop invariants when the code is available.

3.1 Core Algorithm

Figure 3 presents the algorithm for precondition generation. 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, 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

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: ϕ := BOOLLEARN(V^+, V^-)
11: **return** SUBSTITUTE(F, ϕ)

Figure 4. The core algorithm of PIE.

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 sequence 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 in more detail, we note some important aspects of the overall algorithm. First, like other 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 [26, 51] perform boolean learning but lack feature learning, which limits their expressiveness and accuracy. On the other hand, a feature learner based on program synthesis [2, 59, 60] 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 datatypes, 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 set X at line 5 in Figure 4. If the conflict sets 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.

Feature Learning Figure 5 describes our approach to feature learning. The algorithm is a simplified version of the Escher program synthesis tool [2], 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.

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 ∃f ∈ F.(∀t ∈ X+.f(t) ∧ ∀t ∈ X−.¬f(t)) then
6:       return f
7:    end if
8:    i := i + 1
9: end loop
```

Figure 5. The feature learning algorithm.

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 our overall PREGEN 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 larger features and so helps to prevent against over-fitting. 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 [2, 21].

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 [38]; 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

BOOLLEARN(V^+: Feature Vectors, V^-: Feature Vectors) : Boolean Formula
Returns: A formula ϕ 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 ≠ None then
8:        return C
9:     end if
10:    k := k + 1
11: end loop
```

Figure 6. The boolean function learning algorithm.

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.

3.2 Loop Invariant Inference

This section describes how to use PIE to infer sound loop invariants when the code is available and analyzable by an automatic constraint solver.

Verified Precondition Inference As described in Section 2.3, our loop invariant inference engine relies on an algorithm VPREGEN for generating provably sufficient preconditions given loop-free code. The VPREGEN algorithm is shown in Figure 7. We assume the existence of a verifier for loop-free programs, which is invoked after a candidate precondition is generated (line 6). If the verifier succeeds in verifying the sufficiency of precondition P, it returns `None` and we are done. Otherwise it 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 in this case we add t to the set B of "bad" tests and run PIE again, iterating until the generated precondition can be verified.

VPREGEN(C: Code, Q: Predicate, T: Tests) : Predicate
Returns: A sufficient precondition

```
1: Tests G, B := PARTITIONTESTS(C,Q,T)
2: repeat
3:    P := PIE(G,B)
4:    t := VERIFY(P,C,Q)
5:    B := B ∪ {t}
6: until t = None
7: return P
```

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

Loop Invariant Inference For simplicity, we restrict the presentation of our loop invariant inference algorithm to code snippets of the form

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

where C_1 is loop-free. It is straightforward to generalize the approach to handle multiple and nested loops [16]. The goal 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:

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

The overall structure of the invariant inference engine LOOPINVGEN based on PIE is shown in Figure 8. Given a test suite T for C, we first generate a set of tests for the loop by logging the program state every time the loop head is reached. 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 by invoking VPREGEN on line 3. The inner loop (lines 4-7) then

LOOPINVGEN(C: Code, T: Tests) : Predicate
Returns: A loop invariant that is sufficient to verify the assertion in C.

Require: $C =$ `assume` P; `while` E $\{C_1\}$; `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** $t =$ `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.

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.

4. Evaluation

We have evaluated PIE's utility and generality through two experiments. First, we use PIE to infer likely preconditions for library functions without access to the source code. Second, we use PIE to infer sound loop invariants. To our knowledge, no existing approach to precondition inference has been demonstrated to be useful for both purposes.

4.1 Likely Preconditions for OCaml Code

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 two OCaml standard library modules, `List` and `String`, as well as the `BatAvlTree` module from the widely used `batteries` library[4]. Our test generator and feature learner do not currently 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 [52].

For feature learning we use a simplified version of the Escher program synthesis tool [2] that follows the algorithm described in Figure 5. Escher already supports operations on base 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* [34], 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).

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 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 if given an empty tree.

Figure 9 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 [26, 51] can only capture these pre-

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

Case	Postcondition	Learned Features
`set`(s,i,c)	exception	3
	$(i < 0) \vee (\texttt{len}(s) \leq i)$	
`sub`(s,i_1,i_2)	exception	3
	$(i_1 < 0) \vee (i_2 < 0) \vee (i_1 > \texttt{len}(s) - i_2)$	
`index`(s,c)	result $= 0$	2
	$\texttt{has}(\texttt{get}(s, 0), c)$	
`index_from`(s,i,c)	exception	4
	$(i < 0) \vee (i > \texttt{len}(s)) \vee$ $\neg\, \texttt{has}(\texttt{sub}(s, i, \texttt{len}(s) - i), c)$	

`String` module functions

`nth`$(l,\ n)$	exception	2
	$(0 > n) \vee (n \geq \texttt{len}(l))$	
`append`$(l_1,\ l_2)$	$\texttt{len}(\text{result}) = 0$	2
	$\texttt{empty}(l_1) \wedge \texttt{empty}(l_2)$	

`List` module functions

`create`$(t_1,\ v,\ t_2)$	exception	6
	$\texttt{height}(t_1) > (\texttt{height}(t_2) + 1) \vee$ $\texttt{height}(t_2) > (\texttt{height}(t_1) + 1)$	
`concat`$(t_1,\ t_2)$	$\texttt{empty}(\text{result})$	2
	$\texttt{empty}(t_1) \wedge \texttt{empty}(t_2)$	

`BatAvlTree` module functions

Figure 9. Preconditions for OCaml library functions.

conditions if they are provided with exactly the right features (e.g., $\texttt{height}(t_1) > (\texttt{height}(t_2) + 1)$) in advance, while PREGEN 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 is 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.

Finally, we evaluate our algorithm's sensitivity to the number of tests and the maximum conflict group size. The top plot in Figure 10 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 12800 tests we incur one additional timeout due to the extra overhead involved. The bottom plot in Figure 10 shows the results with varied conflict group sizes (and 6400 tests). With very small conflict groups, the tool is more likely to produce overly specific features, which leads to more errors and

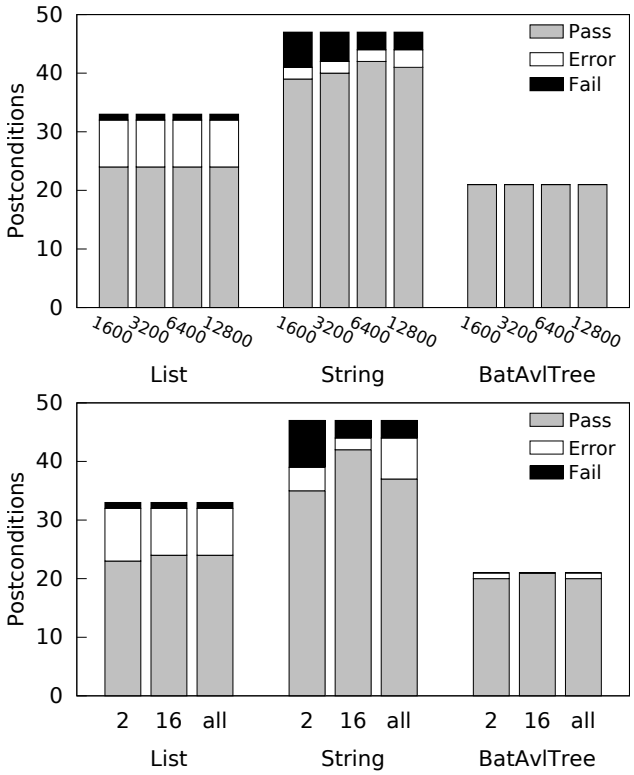

Figure 10. Comparison of PIE configurations. The top plot shows the effect of different numbers of tests. The bottom plot shows the effect of different conflict group sizes.

failures. On the other hand, with unbounded conflict group sizes, the feature learner must synthesize the entire precondition, which fails as the preconditions become larger. For example, this approach fails to generate the preconditions for `String.index_from` and `BatAvlTree.create` shown earlier. 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.

4.2 Loop Invariants for C++ Code

We have implemented the invariant inference procedure described in Figure 8 for C++ code as a Clang tool[5]. As in the figure, verification condition generation is specialized to functions that contain a single loop, though this is not a limitation of the approach. We have implemented a verifier for loop-free programs using the CVC4 [43] and Z3str2 [62] SMT solvers, which support several logical theories including linear arithmetic and strings. We employ both solvers because their support for strings is incomplete, causing some queries to fail to terminate. Thus if one solver fails to terminate after 5 minutes, we try the query with the other solver.

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.

[5] http://clang.llvm.org

Case	Calls to SMT Solvers		Calls to Escher
	SAT	UNSAT	
00	2	5	1
	$(x = y)$		
01	5	8	3
	$(y > 1 - x) \wedge (y > 1)$		
02	40	8	17
	$(z \,\%\, 2 = 1) \wedge (w = 2y) \wedge (x = y)$		
03	7	10	5
	$((w > 1) \vee (w = 1)) \wedge ((k = 1) \vee (k > 1))$		
05	10	8	6
	$(w = z + 1) \wedge (x = y)$		
06	8	11	5
	$(w > z) \wedge (w \leq 2) \wedge (w = z+1) \wedge (x = z) \wedge (x \leq 1)$		
07	5	8	4
	$(x = y) \wedge (i < j \vee i = j)$		
08	3	5	3
	$(i = j) \wedge (j < n \vee n = j)$		
09	59	8	39
	$(i = x) \wedge ((x = 2y) \vee (x \,\%\, 2 = 1)) \wedge$ $((x = 2(y + 1)) \vee (x = 2y))$		
10	2	5	2
	$(y > x) \vee (0 > x)$		
11	5	8	4
	$((l < k) \vee (l = k)) \wedge (j = k)$		
12	10	8	11
	$((m < x) \vee (m = x)) \wedge (0 \leq m) \wedge ((n > m) \vee (m = 0))$		
13	10	5	9
	$((x = y) \vee (j > i) \vee (i > j))$		
14	7	8	8
	$(c_1 > 1) \wedge (c_2 > 1) \wedge ((k > i) \vee (k = i)) \wedge$ $((n > k) \vee (i \neq k))$		

Case	Calls to SMT Solvers		Calls to Escher
	SAT	UNSAT	
17	3	5	3
	$(x \leq n) \vee (x \leq 1)$		
18	7	8	6
	$(m \leq x) \wedge (0 \leq m) \wedge ((n > m) \vee (m = 0))$		
19	2	5	2
	$j > i$		
22	28	8	24
	$((i < n) \vee (n = 2j) \vee (n\%2 = 1)) \wedge ((i = n) \vee$ $(n \neq 2j)) \wedge (j \leq i - j) \wedge (j \leq n - j) \wedge (2(j+1) > i)$		
23	3	5	2
	$(i = j) \wedge (j \leq 3)$		
24	22	14	30
	$(j > (n - k)) \wedge ((n = j) \vee (k > 1)) \wedge$ $((k > 1) \vee (k = 1)) \wedge ((n = j) \vee (n = j+1) \vee (k > 2))$		
25	21	13	17
	$(x \leq y) \wedge (y \leq n) \wedge ((m < y) \vee (m = y)) \wedge$ $((y = x) \vee \neg(y > m))$		
26	26	8	20
	$(i = x) \wedge (m \,\%\, 2 = 0) \wedge (2(y + 1) > x) \wedge$ $(y \leq x - y) \wedge ((n > m) \vee (m > x) \vee (m = 2y))$		
27	70	8	24
	$(m \leq j) \wedge (y = k - x) \wedge ((n > m) \vee (m = 0)) \wedge$ $((0 = m) \vee (1 = m) \vee (1 < m))$		
28	13	8	11
	$(k = 3y) \wedge (x = z) \wedge (x = y)$		
32	4	5	5
	$(x = y) \wedge (i \neq j)$		
33	10	8	11
	$((m = x) \vee (m < x)) \wedge (0 \leq m) \wedge$ $((n > m) \vee (m = 0))$		

Figure 11. Results on the HOLA benchmarks [16].

Case	Calls to SMT Solvers			Calls to Escher
	SAT	UNSAT	UNKNOWN	
b	3	5	0	4
	$i = \texttt{len}(r)$			
c	4	8	0	4
	$\texttt{has}(r, \text{“}a\text{”}) \wedge (\texttt{len}(r) > i)$			
d	5	5	0	3
	$(i = 1) \wedge \texttt{has}(\text{“}a\text{”}, \texttt{get}(r, 0))$			

Figure 12. Results on the string benchmarks [55].

As described earlier, 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 two sets of benchmarks. First, we have run all 26 of the single-loop benchmarks that were used to evaluate the HOLA loop invariant engine [16]. The invariants in these benchmarks require only the theory of linear arithmetic. The results are shown in Table 11. We list each benchmark's number from the original benchmark set, the invariant our tool inferred, the number of calls to the SMT solvers, and the number of features generated. PIE succeeds in inferring invariants for all 26 benchmarks, including one benchmark which HOLA's technique cannot handle. By construction, these invariants are sufficient to ensure the correctness of the assertions in these benchmarks, and PIE infers the invariants fully automatically and with no initial features.

We also ran the above experiment again but with PIE replaced by our program-synthesis-based feature learner. This version succeeds for only 20 out of the 26 benchmarks. Further, for the successful cases, the average running time is 438 seconds and 1539 MB of memory, as compared to 50 seconds and 326 MB for our PIE-based approach.

Second, 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 [55]. As shown in Table 12, our approach infers loop invariants for three of these benchmarks. As mentioned earlier, the prior approach to handle these benchmarks requires both a fixed set of features and a fixed boolean structure fore the desired invariants, neither of which is required by our approach. Our approach fails on the fourth 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 queries, then our tool can succeed on this benchmark.

5. Related Work

The preconditions discussed in the literature can be classified into two categories. *Control* preconditions specify temporal properties, i.e., what other functions should be executed and in what order before invoking the function under observation [1, 3, 4, 18, 33, 35, 40, 49, 61]. In this paper, we focus on *data* preconditions that constrain the inputs to the function. Moreover, our focus is on preconditions that are sufficient to ensure safety properties; other work focuses on liveness properties such as conditional termination [10, 17].

The closest work to ours is that of Sankaranarayanan *et al.* [51], which uses a decision-tree 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 [26]. As discussed in Section 2, the key innovation of PIE over these works is its support for on-demand feature learning, instead of using a fixed set of features that is specified in advance. Indeed Sankaranarayanan *et al.* explicitly mention the need for feature learning as part of future work.

There are also several static approaches to infer provably sufficient preconditions. Unlike PIE, these techniques all require the source code to be available and analyzable. The standard weakest precondition computation infers preconditions for loop-free programs [14]. For programs with loops, a backward symbolic analysis with search heuristics can yield preconditions [6, 12]. Other approaches leverage properties of particular language paradigms [28], require logical theories that support quantifier elimination [15, 45], and employ counterexample-guided abstraction refinement (CEGAR) with domain-specific refinement heuristics [53, 54]. Finally, some static approaches to precondition inference target specific program properties, such as predicates about the heap structure [5, 42] or about function equivalence [37].

We have shown how PIE can be used to build a sound loop invariant inference engine. Prior work also combines data-driven inference with a program verifier or SMT solver to infer sound loop and other invariants. For example, some work has leveraged Daikon [19] to generate likely program specifications and then used a program verifier to remove the invariants that are not provable [23, 46, 47, 52]. These approaches only employ good examples, so they cannot incorporate counterexamples to refine candidate invariants. Other approaches employ both good and bad examples [24, 25, 36, 39, 41, 55–58], as in our approach.

The key distinguishing feature of PIE relative to this work is its support for feature learning. Except for one of the above works [56], which uses support vector machines (SVMs) [11] to learn new numerical features, all prior works employ a fixed set or template of features. Further, the approaches based on Daikon as well as the work of Sharma *et al.* [55–58] cannot learn arbitrary boolean formulas. The ICE framework [24, 25] infers loop invariants by fixing the numerical features to octagons ($\pm x \pm y \leq c$) and learning from "implication counterexamples" (in addition to good and bad examples). In contrast, PIE can directly employ any off-the-shelf boolean learner that uses only good and bad examples.

There are many other approaches to invariant inference. The HOLA [16] loop invariant generator is based on an algorithm for logical abduction [15]; we employed a similar technique to turn PIE into a loop invariant generator. Standard invariant generation tools that are based on abstract interpretation [12, 13], constraint solving [9, 32], or probabilistic inference [30] require the number of disjunctions to be specified manually. Other approaches [20, 22, 27, 29, 31, 44, 50] can handle disjunctions, but they restrict the number of disjunctions by trace-based heuristics, custom built abstract domains, or widening. In contrast, our invariant inference engine places no *a priori* bound on the number of disjunctions.

Finally, the work of Cheung *et al.* [7] combines program synthesis and machine learning in order to provide event recommendations to users of social media. They use the SKETCH system [59] 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. In contrast, PIE 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 shown how to extend 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 makes "black box" precondition inference more expressive by ensuring that produced preconditions are both sufficient and necessary for the given set of tests. We also show how to leverage this expressiveness to build a general approach for inferring loop invariants when the code is available and analyzable. Our experimental results demonstrate the benefits of our approach in both of these settings.

References

[1] M. Acharya, T. Xie, J. Pei, and J. Xu. Mining API patterns as partial orders from source code: from usage scenarios to specifications. In *Proceedings of the 6th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2007, Dubrovnik, Croatia, September 3-7, 2007*, pages 25–34, 2007.

[2] A. Albarghouthi, S. Gulwani, and Z. Kincaid. Recursive program synthesis. In *Computer Aided Verification - 25th International Conference, CAV 2013, Saint Petersburg, Russia, July 13-19, 2013. Proceedings*, pages 934–950, 2013.

[3] R. Alur, P. Cerný, P. Madhusudan, and W. Nam. Synthesis of interface specifications for java classes. In *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2005, Long Beach, California, USA, January 12-14, 2005*, pages 98–109, 2005.

[4] G. Ammons, R. Bodík, and J. R. Larus. Mining specifications. In *Conference Record of POPL 2002: The 29th SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Portland, OR, USA, January 16-18, 2002*, pages 4–16, 2002.

[5] 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.

[6] 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*, PLDI '09, pages 363–374, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-392-1. doi: 10.1145/1542476.1542517. URL http://doi.acm.org/10.1145/1542476.1542517.

[7] A. Cheung, A. Solar-Lezama, and S. Madden. Using program synthesis for social recommendations. In *21st ACM International Conference on Information and Knowledge Management, CIKM'12, Maui, HI, USA, October 29 - November 02, 2012*, pages 1732–1736, 2012.

[8] E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement. In *Computer Aided Verification, 12th International Conference, CAV 2000, Chicago, IL, USA, July 15-19, 2000, Proceedings*, pages 154–169, 2000.

[9] M. Colón, S. Sankaranarayanan, and H. Sipma. Linear invariant generation using non-linear constraint solving. In *Computer Aided Verification, 15th International Conference, CAV 2003, Boulder, CO, USA, July 8-12, 2003, Proceedings*, pages 420–432, 2003.

[10] B. Cook, S. Gulwani, T. Lev-Ami, A. Rybalchenko, and M. Sagiv. Proving conditional termination. In *Computer Aided Verification, 20th International Conference, CAV 2008, Princeton, NJ, USA, July 7-14, 2008, Proceedings*, pages 328–340, 2008.

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

[12] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference Record of the Fourth ACM Symposium on Principles of Programming Languages, Los Angeles, California, USA, January 1977*, pages 238–252, 1977.

[13] P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. In *Conference Record of the Fifth Annual ACM Symposium on Principles of Programming Languages, Tucson, Arizona, USA, January 1978*, pages 84–96, 1978.

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

[15] I. Dillig and T. Dillig. Explain: A tool for performing abductive inference. In N. Sharygina and H. Veith, editors, *Computer Aided Verification - 25th International Conference, CAV 2013, Saint Petersburg, Russia, July 13-19, 2013. Proceedings*, volume 8044 of *Lecture Notes in Computer Science*, pages 684–689. Springer, 2013.

[16] 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, OOPSLA 2013, part of SPLASH 2013, Indianapolis, IN, USA, October 26-31, 2013*, pages 443–456, 2013.

[17] V. D'Silva and C. Urban. Conflict-driven conditional termination. In *Computer Aided Verification - 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18-24, 2015, Proceedings, Part II*, pages 271–286, 2015.

[18] D. R. Engler, D. Y. Chen, and A. Chou. Bugs as inconsistent behavior: A general approach to inferring errors in systems code. In *Proceedings of the 18th ACM Symposium on Operating System Principles, SOSP 2001, Chateau Lake Louise, Banff, Alberta, Canada, October 21-24, 2001*, pages 57–72, 2001.

[19] 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.

[20] M. Fähndrich and F. Logozzo. Static contract checking with abstract interpretation. In *Formal Verification of Object-Oriented Software - International Conference, FoVeOOS 2010, Paris, France, June 28-30, 2010, Revised Selected Papers*, pages 10–30, 2010.

[21] 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, Portland, OR, USA, June 15-17, 2015*, pages 229–239, 2015.

[22] G. Filé and F. Ranzato. Improving abstract interpretations by systematic lifting to the powerset. In *Logic Programming, Proceedings of the 1994 International Symposium, Ithaca, New York, USA, November 13-17, 1994*, pages 655–669, 1994.

[23] 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 Verifica-*

tion Conference, HVC 2014, Haifa, Israel, November 18-20, 2014. Proceedings, pages 48–53, 2014.

[24] P. Garg, C. Löding, P. Madhusudan, and D. Neider. ICE: A robust framework for learning invariants. 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 69–87, 2014.

[25] P. Garg, C. Löding, P. Madhusudan, and D. Neider. Learning invariants using decision trees and implication counterexamples. In *POPL*, 2016.

[26] T. Gehr, D. Dimitrov, and M. T. Vechev. Learning commutativity specifications. In D. Kroening and C. S. Pasareanu, editors, *Computer Aided Verification - 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18-24, 2015, Proceedings, Part I*, volume 9206 of *Lecture Notes in Computer Science*, pages 307–323. Springer, 2015. ISBN 978-3-319-21689-8. URL http://dx.doi.org/10.1007/978-3-319-21690-4.

[27] 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, VMCAI 2012, Philadelphia, PA, USA, January 22-24, 2012. Proceedings*, pages 235–250, 2012.

[28] R. Giacobazzi. Abductive analysis of modular logic programs. *Journal of Logic and Computation*, 8(4):457–483, 1998. URL http://dx.doi.org/10.1093/logcom/8.4.457.

[29] D. Gopan and T. W. Reps. Guided static analysis. In *Static Analysis, 14th International Symposium, SAS 2007, Kongens Lyngby, Denmark, August 22-24, 2007, Proceedings*, pages 349–365, 2007.

[30] S. Gulwani and N. Jojic. Program verification as probabilistic inference. In *Proceedings of the 34th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2007, Nice, France, January 17-19, 2007*, pages 277–289, 2007.

[31] 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, Tucson, AZ, USA, June 7-13, 2008*, pages 281–292, 2008.

[32] 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, TACAS 2009, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009, York, UK, March 22-29, 2009. Proceedings*, pages 262–276, 2009.

[33] T. A. Henzinger, R. Jhala, and R. Majumdar. Permissive interfaces. In *Proceedings of the 10th European Software Engineering Conference held jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2005, Lisbon, Portugal, September 5-9, 2005*, pages 31–40, 2005.

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

[35] G. Jiang, H. Chen, C. Ungureanu, and K. Yoshihira. Multi-resolution abnormal trace detection using varied-length n-grams and automata. In *Second International Conference on Autonomic Computing (ICAC 2005), 13-16 June 2005, Seattle, WA, USA*, pages 111–122, 2005.

[36] 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, VMCAI 2010, Madrid, Spain, January 17-19, 2010. Proceedings*, pages 180–196, 2010.

[37] M. Kawaguchi, S. K. Lahiri, and H. RebÂŘlo. Conditional equivalence. Technical report, MSR, 2010.

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

[39] 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, APLAS 2010, Shanghai, China, November 28 - December 1, 2010. Proceedings*, pages 328–343, 2010.

[40] T. Kremenek, P. Twohey, G. Back, A. Y. Ng, and D. R. Engler. From uncertainty to belief: Inferring the specification within. In *7th Symposium on Operating Systems Design and Implementation (OSDI '06), November 6-8, Seattle, WA, USA*, pages 161–176, 2006.

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

[42] 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.

[43] 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, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18-22, 2014. Proceedings*, pages 646–662, 2014.

[44] L. Mauborgne and X. Rival. Trace partitioning in abstract interpretation based static analyzers. In *ESOP*, 2005.

[45] Y. Moy. Sufficient preconditions for modular assertion checking. In F. Logozzo, D. Peled, and L. D. Zuck, editors, *Verification, Model Checking, and Abstract Interpretation, 9th International Conference, VMCAI 2008, San Francisco, USA, January 7-9, 2008, Proceedings*, volume 4905 of *Lecture Notes in Computer Science*, pages 188–202. Springer, 2008. ISBN 978-3-540-78162-2. URL http://dx.doi.org/10.1007/978-3-540-78163-9_18.

[46] J. W. Nimmer and M. D. Ernst. Automatic generation of program specifications. In *Proceedings of the International Symposium on Software Testing and Analysis, July 22-24, Roma, Italy. ACM, 2002*, pages 229–239, 2002.

[47] 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*, SIGSOFT '02/FSE-10, pages 11–20, New York, NY, USA, 2002. ACM.

ISBN 1-58113-514-9. doi: 10.1145/587051.587054. URL http://doi.acm.org/10.1145/587051.587054.

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

[49] M. K. Ramanathan, A. Grama, and S. Jagannathan. Static specification inference using predicate mining. In *Proceedings of the ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation, San Diego, California, USA, June 10-13, 2007*, pages 123–134, 2007.

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

[51] S. Sankaranarayanan, S. Chaudhuri, F. Ivancic, and A. Gupta. Dynamic inference of likely data preconditions over predicates by tree learning. In *Proceedings of the ACM/SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2008, Seattle, WA, USA, July 20-24, 2008*, pages 295–306, 2008.

[52] 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*, ICSE 2014, pages 596–607, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2756-5. doi: 10.1145/2568225.2568285. URL http://doi.acm.org/10.1145/2568225.2568285.

[53] M. N. Seghir and D. Kroening. Counterexample-guided precondition inference. In *Programming Languages and Systems - 22nd European Symposium on Programming, ESOP 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013, Rome, Italy, March 16-24, 2013. Proceedings*, pages 451–471, 2013.

[54] M. N. Seghir and P. Schrammel. Necessary and sufficient preconditions via eager abstraction. In *Programming Languages and Systems - 12th Asian Symposium, APLAS 2014, Singapore, November 17-19, 2014, Proceedings*, pages 236–254, 2014.

[55] 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.

[56] R. Sharma, A. V. Nori, and A. Aiken. Interpolants as classifiers. In *Computer Aided Verification - 24th International Conference, CAV 2012, Berkeley, CA, USA, July 7-13, 2012 Proceedings*, pages 71–87, 2012.

[57] R. Sharma, S. Gupta, B. Hariharan, A. Aiken, and A. V. Nori. Verification as learning geometric concepts. In *Static Analysis - 20th International Symposium, SAS 2013, Seattle, WA, USA, June 20-22, 2013. Proceedings*, pages 388–411, 2013.

[58] R. Sharma, E. Schkufza, B. R. Churchill, and A. Aiken. Conditionally correct superoptimization. In *OOPSLA*, 2015.

[59] 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*, ASPLOS XII, pages 404–415, New York, NY, USA, 2006. ACM. ISBN 1-59593-451-0. doi: 10.1145/1168857.1168907. URL http://doi.acm.org/10.1145/1168857.1168907.

[60] S. Srivastava, S. Gulwani, and J. S. Foster. From program verification to program synthesis. In *Proceedings of the 37th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2010, Madrid, Spain, January 17-23, 2010*, pages 313–326, 2010.

[61] J. Whaley, M. C. Martin, and M. S. Lam. Automatic extraction of object-oriented component interfaces. In *ISSTA*, pages 218–228, 2002.

[62] 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, ESEC/FSE'13, Saint Petersburg, Russian Federation, August 18-26, 2013*, pages 114–124, 2013.