On Scaling Data-Driven Loop Invariant Inference

SAHIL BHATIA, Microsoft Research, India
SASWAT PADHI, University of California Los Angeles, USA
NAGARAJAN NATARAJAN, Microsoft Research, India
RAHUL SHARMA, Microsoft Research, India
PRATEEK JAIN, Microsoft Research, India

Automated synthesis of inductive invariants is an important problem in software verification. Once all the invariants have been specified, software verification reduces to checking of verification conditions. Although static analyses to infer invariants have been studied for over forty years, recent years have seen a flurry of data-driven invariant inference techniques which guess invariants from examples instead of analyzing program text. However, these techniques have been demonstrated to scale only to programs with a small number of variables. In this paper, we study these scalability issues and address them in our tool Oasis that improves the scale of data-driven invariant inference and outperforms state-of-the-art systems on benchmarks from the invariant inference track of the Syntax Guided Synthesis competition.

1 INTRODUCTION

Inferring inductive invariants is one of the core problems of software verification. Recently, there has been a flurry of data-driven invariant inference techniques [Alur et al. 2017; Garg et al. 2014, 2016; Li et al. 2017; Nguyen et al. 2017, 2012; Padhi et al. 2016; Sharma and Aiken 2016; Sharma et al. 2013b,a, 2012; Thakur et al. 2015; Zhu et al. 2018] that learn invariants from examples. These data-driven techniques offer attractive features such as the ability to systematically generate *disjunctive* invariants. Whereas the well-know static invariant inference techniques either fail to infer disjunctive invariants [Colón et al. 2003; Cousot and Cousot 1977; Cousot and Halbwachs 1978; Miné 2006] or require a user-provided bound on the number of disjunctions [Bagnara et al. 2006; Gulwani and Jojic 2007; Gulwani et al. 2008; Gupta et al. 2013; Sankaranarayanan et al. 2006]. At the heart of the data-driven techniques is an *active learning* [Hanneke 2009] loop: a *learner* guesses a candidate invariant from data and provides the candidate to a *teacher*. The teacher either validates that the candidate is a valid invariant or returns a counterexample. This example is added to the data and the process is repeated until the learner guesses a correct invariant. In this architecture, the more the number of program variables in the verification problem, the more the learner is likely to choose an incorrect candidate or take a long time to generate good candidates [Padhi et al. 2019]. Hence, as we discuss in our evaluation in Section 6, existing data-driven invariant inference techniques have been shown to be effective only for programs with a small number of variables.

For data-driven invariant inference to be applicable to verification of practical software, these scalability challenges must be addressed. There are two main obstacles to scalability which are related to the number of program variables: First, a program can have many variables and often only a small subset of these variables are *relevant* to the invariants. Intuitively, writing correct programs that require complicated invariants with many variables is hard for developers and prior works on invariant inference are also biased towards simple invariants [Albarghouthi and McMillan 2013]. In the absence of a technique that separates the relevant variables from the irrelevant, the learner can get bogged down by the irrelevant variables. In particular, invariant inference benchmarks in the Syntax Guided Synthesis (SyGuS) competition are provided as logic formulas where static *slicing* [Horwitz et al. 1988] fails to remove the semantically irrelevant variables. Second, data-driven

Authors' addresses: Sahil Bhatia, Microsoft Research, India, t-sab@microsoft.com; Saswat Padhi, University of California Los Angeles, USA, padhi@cs.ucla.edu; Nagarajan Natarajan, Microsoft Research, India, nagarajn@microsoft.com; Rahul Sharma, Microsoft Research, India, rahsha@microsoft.com; Prateek Jain, Microsoft Research, India, prajain@microsoft.com.

techniques also rely on some form of enumeration to generate candidate predicates. Thus, a higher number of variables causes the enumerator to take a long time to reach pertinent candidates. For example, if the enumerator exhaustively generates expressions in increasing size [Alur et al. 2017; Padhi et al. 2019] then before enumerating expressions of size s, it must enumerate all expression of size $s - 1$ over all variables.

To exemplify these scalability issues, we consider LoopInvGen [Padhi et al. 2019, 2016], the state-of-the-art data-driven invariant inference tool that won in the invariant inference track of SyGuS competition held in 2017 and 2018. It uses exhaustive enumeration to synthesize Boolean *features* (simple predicates). Then, a Boolean function learner generates a candidate invariant which is a Boolean combination of the features. As the number of program variables increases, the scalability degrades because the enumerator must explore exponentially many features and the learner needs many examples to avoid generating candidates with irrelevant variables; with a large number of variables, the learner can *overfit* on the irrelevant variables to generate incorrect candidates that will be rejected by the teacher [Padhi et al. 2019].

We explore addressing both the scalability issues, caused by enumeration and irrelevant variables, through machine learning (ML). In particular, we make the following two contributions. First, we describe a learner that can infer the relevant variables, thus ensuring that data-driven invariant inference is only applied to the simpler problem with a few or no irrelevant variables. Since the number of relevant variables is typically small, data-driven invariant inference can scale to such tasks better. Second, we show that exhaustive enumeration can be replaced by learners that are much more scalable. Instead of a *generate-and-check* approach where an enumerator generates all possible candidate features eagerly [Albarghouthi et al. 2013; Ernst et al. 2000; Padhi et al. 2016], we employ a more scalable *guess-and-check* approach where the learner intelligently guesses features from data.

We have implemented these techniques in a tool Oasis[1] that takes as input logic formulas which encode the verification of safety properties of programs over integer variables and outputs inductive invariants that are sufficient to prove the properties. To this end, Oasis employs new ML algorithms for the well-known *binary classification* problem: the learner's goal is to find a *classifier* that *separates* positive and negative examples. The classifier is a predicate that includes the positive examples and excludes the negative examples. In the context of invariant inference, an *example* is a program *state* that maps variables to integers. Oasis makes the following contributions.

First, Oasis uses binary classification to infer relevant and irrelevant variables (Section 4.3). It uses symbolic execution to generate *reachable* states (positive examples) and *bad* states (negative examples), which are backward reachable from states that violate the safety properties. Then it finds a *sparse* classifier and we classify the variables occurring in the classifier as relevant. If a variable is absent from the classifier and it is possible to separate samples of reachable states from bad states without using the variable then it is likely to be irrelevant to the invariant. The sparsity requirement ensures that we keep the number of relevant variables to minimal.

We remark that we need a custom learner for this task. Each state (reachable or bad) is a partial map from variables to integers. In particular, there are some variables that are not in the domain of the state. These variables are *don't care*, *i.e.*, they can be assigned any value without affecting the *label* (positive or negative) of the state. The (partial) models generated by SMT solvers typically have don't cares. The well-known ML classifiers learn over total maps as opposed to partial maps. Although one can extend a partial map to a total map by setting the don't care variables to zero or randomly assigned values, these alternatives are undesirable as a partial map corresponds to an infinite number of possible total maps and supplanting it with any total map loses the information

[1] The name Oasis stands for **O**ptimization **A**nd **S**earch for **I**nvariant **S**ynthesis.

encoded in the partial map. Hence, we have designed a custom learner that directly learns a classifier using partial maps and is not limited to total maps. After we obtain the set of relevant variables from the classifier, OASIS calls a modified version of LOOPINVGEN where the synthesis of features is restricted to predicates over the relevant variables.

Second, OASIS uses a learner to synthesize Boolean features from data (Section 4.4). Internally, LOOPINVGEN breaks down the problem of invariant inference into many small binary classification tasks and uses Escher [Albarghouthi et al. 2013] to find features that solve them. Specifically, Escher exhaustively enumerates all features in increasing size till it finds one that separates the positive examples from the negative examples in the small task. OASIS replaces Escher with a learner to find such features. Unlike traditional ML algorithms that have non-zero *error*, *i.e.*, they fail to separate some positive examples from some negative examples, LOOPINVGEN requires the feature synthesizer to have zero error.

OASIS uses the same learner to solve both these problems, *i.e.*, inferring relevant variables and inferring features. In particular, the learner of OASIS solves a non-standard ML problem: finding sparse classifiers with zero error in the presence of don't cares. We describe a novel learner that solves this problem (Section 5). To the best of our knowledge, all prior works on data-driven invariant inference use learners that require total maps. We show how to encode the problem of finding such a classifier as an instance of integer linear programming (ILP) which minimizes an objective function subject to linear constraints. Although linear programming has previously been used to assist invariant inference [Gupta et al. 2013], our encoding is novel. Specifically, we show how to systematically encode domain-specific heuristics as objective functions or constraints for effective learning in the context of invariant inference. Heavily optimized ILP solvers are available as off-the-shelf tools and OASIS uses them to scale data-driven invariant inference.

To demonstrate the scalability of OASIS in practice, we evaluate OASIS on over 400 benchmarks from the invariant (Inv) track of the SyGuS competion held in 2019 [syg 5 14] (Section 6). This benchmark set includes the new community provided programs that have a large number of irrelevant variables which test the scalability of invariant synthesis tools [Si et al. 2018]. Our evaluation shows that OASIS significantly improves the scalability of data-driven invariant inference on these benchmarks and solves 20% more benchmarks than LOOPINVGEN, the state-of-the-art data-driven invariant inference tool. OASIS even outperforms state-of-the-art invariant inference tools that are based on very different techniques. It solves more benchmarks than deductive synthesis implemented in CVC4 [Barrett et al. 2011; Reynolds et al. 2015] and cooperative synthesis of the recent work of DRYADSYNTH [Huang et al. 2020] that combines enumerative and deductive synthesis. Thus, our evaluation shows that OASIS significantly improves the state-of-the-art in data-driven invariant inference and makes it as scalable as deductive and cooperative techniques. OASIS solves more benchmarks than these tools and also solves benchmarks that are beyond the reach of prior work.

The rest of the paper is organized as follows. We provide an example to show the end-to-end working of OASIS (Section 2) and review the relevant background (Section 3). We describe OASIS in detail (Section 4) followed by the ILP-based learner (Section 5). We evaluate OASIS in Section 6, place it in the context of the landscape of invariant inference techniques in Section 7, and conclude with directions for future work in Section 8.

assume $(k \geq 0 \land n \geq 0)$
$i = j = 0$
while $(i \leq n)$ **do**
  $(i, j, y) \longleftarrow (i + 1, j + 1, i \times j)$
assert $(i + j + k \geq 2n \lor y \geq n^2)$

Fig. 1. The C-program of working example.

2 WORKING EXAMPLE

We use a simple (contrived) benchmark to show the working of each component of Oasis. The goal is to synthesize an inductive invariant $I(\vec{x})$, where $\vec{x} = \langle i, j, k, n, y \rangle$ that satisfies the following *verification conditions* (VCs) expressed as Horn Clauses.

$$Pre(\vec{x}) \Rightarrow I(\vec{x}) \text{ with } Pre(\vec{x}) \triangleq i = j = 0 \land k \geq 0 \land n \geq 0$$
$$I(\vec{x}) \land Trans(\vec{x}, \vec{x}') \Rightarrow I(\vec{x}') \text{ with } Trans(\vec{x}, \vec{x}') \triangleq i \leq n \land i' = i + 1 \land j' = j + 1 \land y' = i \times j$$
$$I(\vec{x}) \Rightarrow Post(\vec{x}) \text{ with } Post(\vec{x}) \triangleq i \leq n \lor i + j + k \geq 2n \lor y \geq n^2$$

These VCs encode the verification of the C-program in Figure 1. If there exists a predicate I that satisifes the VCs then for all possible inputs the assertion can never be violated. A *state* for this example is a 5-tuple that maps i, j, k, n, y to integers or don't cares (denoted by $\top$).

The first step is the identification of irrelevant variables. Oasis generates reachable states, i.e., positive examples by computing satisfying assignments of $Pre(\vec{x})$ and $Pre(\vec{x}) \land Trans(\vec{x}, \vec{x}')$. For bad states, i.e., negative examples, Oasis computes satisfying assignments of $\neg Post(\vec{x})$ and $\neg Post(\vec{x}') \land Trans(\vec{x}, \vec{x}')$. These satisfying assignments are obtained from off-the-shelf SMT solvers and result in Table 1.

i	j	k	n	y	ℓ
0	0	0	0	$\top$	1
2	3	0	1	2	1
1	-1	0	0	-1	0
6	4	0	5	15	0

Table 1. Initial symbolic execution data.

i	j	k	n	y	ℓ
0	0	-2	-1	0	0
2	3	-3	1	0	0

Table 2. Additional data from robustness checking.

i	j	k	n	ℓ
1	1	742	0	1
0	0	0	859	1
-2	-2	0	-2	0
-3	-3	1	-3	0

i	j	k	n	ℓ
0	0	21	0	1
1	1	115	38	1
5	15	0	5	1
5	0	1	4	0
6	1	0	4	0

i	j	k	n	ℓ
0	0	21	0	1
1	1	115	38	1
373	374	-3	372	0

Table 3. Classification problems generated by LoopInvGen.

Here, the label $\ell = 1$ corresponds to positive examples and $\ell = 0$ corresponds to negative examples. Our learner outputs $i \leq j$ as the classifier for the binary classification problem in Table 1. We run LoopInvGen with $\vec{r} = \{i, j\}$ as relevant variables and the rest of the variables marked irrelevant. Note that this set of relevant variables is incorrect and this instance of LoopInvGen will fail. In parallel to LoopInvGen, we continue improving our set of relevant variables.

The predicate $i \leq j$ *separates* the positives from the negatives: it includes, *i.e.*, is *true* for all positive examples and excludes, *i.e.*, is *false* for all negative examples in the data. Ideally, we want the classifier to *generalize* well: it should not happen that if we generate few more examples then the classifier can no longer separate the positives from the negatives. Next, we check for the *robustness* of this separator by checking for existence of positive states that it excludes or negative states that it includes. The former are generated via satisfying assignments of $Pre(\vec{x}) \wedge Trans(\vec{x}, \vec{x}') \wedge i' > j'$ and the latter from $\neg Post(\vec{x}') \wedge Trans(\vec{x}, \vec{x}') \wedge i \leq j$ and are shown in Table 2. Note that no new positive examples are added in this step as the former predicate is unsatisfiable.

Next, we use the learner to find a classifier using the data in Table 1 and Table 2. We repeat these steps till an instance of LoopInvGen succeeds. Here, these iterations end at $i \leq j + k \wedge i \leq n + 1$ which labels i, j, k, n as relevant and y as irrelevant. Note that any syntactic slicing-based technique would mark y as relevant but the semantic data guides our learner to determine the irrelevance of y. Next, we show how LoopInvGen (with our improvements) successfully infers an invariant I with this set of relevant variables.

LoopInvGen breaks down the process of finding I into two steps. First, it creates many small binary classification problems. For each such problem, a *feature synthesizer* generates a *feature* that separates the positives from the negatives. Second, the features are combined together using a Boolean function learner to generate a candidate invariant. LoopInvGen repeats these steps till a predicate that satisfies all the VCs is discovered. For our example, LoopInvGen generates the classification problems in Table 3 (See [Padhi et al. 2016] for how LoopInvGen generates these problems). Our contribution lies in using our learner to find features for each of these problems rather than using LoopInvGen's exhaustive enumeration based feature synthesizer. Here, our learner generates the following features for these three problems: $i \geq 0$, $i \leq j$, and $k \geq 0$. The Boolean function learner combines these features to generate the following candidate invariant $i \geq 0 \wedge i \leq j \wedge k \geq 0$ that satisfies all the VCs. This inductive invariant shows that the assertion in Figure 1 holds for all possible inputs.

3 BACKGROUND

In this section, we formally define the problem of verifying correctness of programs using loop invariants, describe how invariant inference can be considered as a binary classification problem, and then describe how LoopInvGen reduces this classification problem to many small binary classification problems.

3.1 Program Verification and loop invariants

The first step in program verification is defining a *specification* for the desired property. Typically [web 5 23b,c] this is provided as a pair of logical formulas — **(a)** a *precondition* that constrains the initial state of the program, and **(b)** a *postcondition* that validates the final state after execution of the program. Many programming languages support the **assume** and **assert** keywords, where **assume**(ϕ) silently halts executions that satisfy $\neg\phi$ and executing **assert**(ϕ) with a state that satisfies $\neg\phi$ raises an exception. For example, Figure 1 shows a program having a loop where the initial values are specified by initializations/**assume** statements and the postcondition is specified using the **assert** in the last line. Given such a specification, we define the verification problem as:

Definition 3.1 (Program Verification). Given a program **P** and a specification consisting of a pair of formulas — a precondition ρ and a postcondition ϕ, the verification problem is to prove that for all executions starting from states that satisfy ρ, the states obtained after executing **P** satisfy ϕ.

In Floyd-Hoare logic (FHL) [Floyd 1967; Hoare 1969], this problem is abbreviated to the formula $\{\rho\}\ \mathbf{P}\ \{\phi\}$, called a *Hoare triple*. We say that a Hoare triple is *valid* if the correctness of **P** can be provably demonstrated. For example, while $\{x < 0\}$ `y ← -x` $\{y > 0\}$ is valid, $\{x < 0\}$ `y ← x + 1` $\{y < 0\}$ is not. FHL offers initial theoretical underpinnings for automatic verification by providing a set of inference rules that can be used on the program structure. Today, state-of-the-art verification tools have mechanized these rules and apply them automatically.

However, the FHL inference rules can automatically be applied only for validating Hoare triples that are defined on loop-free programs. Applying these rules on a loop requires an additional parameter called a *loop invariant* — a predicate over the program state that is preserved across each iteration of the loop. To establish the validity of a Hoare triple, the FHL require a loop invariant to satisfy three specific properties, and a predicate that satisfies all three is called a *sufficient loop invariant*.

Definition 3.2 (Sufficient Loop Invariant). Consider a simple loop, **while** G **do** S, which executes the statement S until the condition (loop guard) G holds and then it halts. Then, for the Hoare triple $\{\rho\}$ **while** G **do** S $\{\phi\}$ to be valid, there must exist a predicate $\mathcal{I}$ that satisfies:

$$\begin{aligned} \mathrm{VC}_{\mathrm{pre}}: &\quad \rho \implies \mathcal{I}, \textit{ i.e., } \mathcal{I} \text{ must hold immediately before the loop} \\ \mathrm{VC}_{\mathrm{ind}}: &\quad \{G \wedge \mathcal{I}\}\ S\ \{\mathcal{I}\}, \textit{ i.e. } \mathcal{I} \text{ must be inductive (hold after each iteration)} \\ \mathrm{VC}_{\mathrm{post}}: &\quad \mathcal{I} \implies G \vee \phi, \textit{ i.e., } \mathcal{I} \text{ must certify the postcondition upon exiting the loop} \end{aligned}$$

These three properties are called the *verification conditions* (VCs) for the loop. Any predicate $\mathcal{I}$ that satisfies the first two VCs is called a *loop invariant*. A loop invariant that also satisfies the third VCs said to be *sufficient* (for proving the correctness of the Hoare triple). In this paper, we use *invariants* to denote sufficient loop invariants for brevity.

Thanks to efficient theorem provers [Barrett et al. 2011; de Moura and Bjørner 2008], today it is possible to automatically check if a given predicate is indeed an invariant. However, automatically finding an invariant for arbitrary loops is undecidable in general, and even small loops are challenging for state-of-the-art tools. The invariant inference track of the syntax guided synthesis competition has hundreds of benchmarks where each benchmark provides a $\mathrm{VC}_{\mathrm{pre}}$, a $\mathrm{VC}_{\mathrm{ind}}$, and a $\mathrm{VC}_{\mathrm{post}}$ as logical formulas. Different tools compete to *solve* these problems, *i.e.*, to infer the invariants every year.

3.2 Data-Drive Invariant Inference

An invariant can be viewed as a zero-error classifier — it should demonstrate that the set of possible reachable states at the entry to a loop (called *loop-head states*) are disjoint from the bad states that violate the postcondition; thus establishing that the postcondition is satisfied for all executions.

Consider verifying our motivating example from Figure 1. We visualize the classification problem in Figure 2. $\mathrm{VC}_{\mathrm{pre}}$ and $\mathrm{VC}_{\mathrm{ind}}$ from Definition 3.2 require $\mathcal{I}$ (dashed blue ellipse) to capture all possible loop-head states (cyan dots). These include states satisfying the precondition ρ (green circle), *e.g.*, $(i = j = 0, n = 2)$ appearing before the first iteration, and the subsequent states after each iteration (indicated by the arrows), *e.g.*, $(i = j = 1, n = 2, y = 0)$, $(i = j = 2, n = 2, y = 1)$ etc. The $\neg G \wedge \neg\phi$ space (red rectangle) denotes the states violating the postcondition, *e.g.*, $(i = j = 2, k = 0, n = 1, y = -1)$. $\mathrm{VC}_{\mathrm{post}}$ forces $\mathcal{I}$ to be disjoint with this space. An invariant $\mathcal{I}$ that satisfies the VCs guarantees that no execution starting from ρ would terminate at a state that violates the desired postcondition ϕ.

To infer invariants, we can label examples of loop-head states as positive and satisfying assignments of $\neg G \wedge \neg\phi$ as negative and use a classification algorithm to separate these. The output classifier is a candidate invariant. If the candidate satisfies all the VCs then we have succeeded

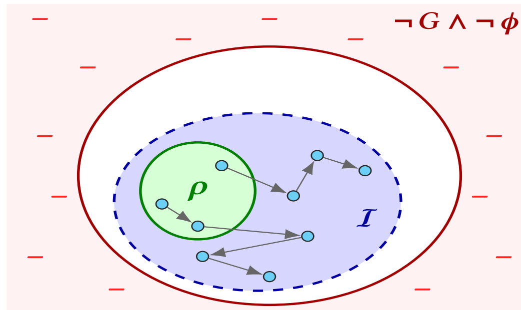

Fig. 2. A sufficient loop invariant can be viewed as a classifier for states.

in inferring an invariant. If some VC is violated then SMT solvers can produce counterexamples which can be added to positive or negative examples to generate another candidate. Since the actual invariant can be complex, prior work has explored increasingly complex learning algorithms including support vector machines [Li et al. 2017; Sharma et al. 2012], decision trees [Garg et al. 2016; Zhu et al. 2018], algorithms for learning Boolean combinations of half-spaces [Sankaranarayanan et al. 2006], Metropolis-Hastings sampling [Sharma and Aiken 2016], Gibbs sampling [Gulwani and Jojic 2007], SMT-based constraint solving [Garg et al. 2014], and, finally, neural networks [Ryan et al. 2020; Si et al. 2018]. An alternative approach was proposed by [Padhi et al. 2016] where this classification problem is decomposed into smaller more tractable classification problems that can be solved by simple learning algorithms. This approach is implemented in the tool LoopInvGen that Oasis builds upon.

3.3 LoopInvGen

LoopInvGen [Padhi et al. 2016] is a state-of-the-art data-driven invariant inference tool. It consists of a learner and a teacher that interact with each other. The teacher has access to an SMT solver and can verify loop-free programs. In particular, given a candidate invariant $\mathcal{I}$ generated by the learner, it can check the VCs and if some VC fails then it returns a program state as a counterexample. LoopInvGen uses a multi-stage learning technique that composes the candidate invariant out of several predicates, known as features, learned over smaller subproblems. Algorithm 1 outlines this framework.

The main Infer procedure is invoked with a Hoare triple $\mathcal{L} \equiv \{\rho\}$ **while** G **do** S $\{\phi\}$, and a set $\mathcal{P}$ of reachable program states. Here, we assume the loop-body S to be loop-free[2]. The program states are sampled at random by running the loop for a few iterations [Duran and Ntafos 1981]. All the states that LoopInvGen deals with are total maps that map all variables to some integers. The Check(B) procedure is a call to the teacher that invokes an SMT solver to check if B is valid. If B is valid the call returns $\perp$ otherwise it returns a (complete) satisfying assignment of $\neg B$.

Line 2 performs a sanity check: if $\rho \wedge \neg G \wedge \neg \phi$ is satisfiable then the input Hoare triple is invalid and no invariant exists. LoopInvGen starts with a weak candidate invariant $\mathcal{I} \equiv (\neg G \implies \phi)$, and iteratively strengthens it (line 17) for inductiveness. These choices ensures that all candidate invariants $\mathcal{I}$ satisfy VC_{post}. Lines 5 and 15 additionally check for VC_{pre} and VC_{ind} respectively, and add appropriate counterexamples. While a violation of VC_{pre} adds a *positive* example, a violation of VC_{ind} adds a *negative* example. Since the loop body S is loop free, VC_{ind} can be encoded as an

[2] LoopInvGen does handle multiple and nested loops [Padhi et al. 2016].

Algorithm 1 The LoopInvGen algorithm [Padhi et al. 2016]. The teacher is Check and the learner is Learn.

```
    function Infer({ρ} while G do S {ϕ}, 𝒫)
1       if Check(ρ ⟹ (G ∨ ϕ)) ≠ ⊥ then return False
2       ℐ ← (¬G ⟹ ϕ)
3       while True do
4           c ← Check(ρ ⟹ ℐ)
5           if c ≠ ⊥ then return Infer({ρ} while G do S {ϕ}, 𝒫 ∪ {c})
6           𝒩 ← {}
7           while True do
8               ℱ ← {}
9               while True do
10                  (P, N) ← Conflict(𝒫, 𝒩, ℱ)
11                  if P = N = {} then break
12                  else ℱ ← ℱ ∪ Learn(P, N)
13              δ ← BoolCombine(ℱ)
14              c ← Check({δ ∧ G ∧ ℐ} S {ℐ})
15              if c ≠ ⊥ then 𝒩 ← 𝒩 ∪ {c}
16          ℐ ← (ℐ ∧ δ)
17          if δ = True then return ℐ
```

SMT formula (through a weakest precondition computation) whose validity ensures the validity of $\mathrm{VC}_{\mathrm{ind}}$. The lines 10 – 14 indicates the key learning subcomponents.

In line 11, the Conflict procedure selects two sets $P \subseteq \mathcal{P}$ and $N \subseteq \mathcal{N}$ that are *conflicting*, *i.e.*, these positive and negative examples are indistinguishable modulo $\mathcal{F}$, the set of current features. That is for all features $f \in \mathcal{F}.\forall x, y \in P \cup N. f(x) = f(y)$. For such P and N, line 13 learns a feature that separates P and N by invoking the learner Learn. In LoopInvGen, the learner is implemented using Escher [Albarghouthi et al. 2013] that exhaustively enumerates all predicates over all variables in increasing size till it finds a feature f that separates P and N, and this f is added to $\mathcal{F}$. The loop in lines 10–13 has the following postcondition: $\forall x \in \mathcal{P}.\forall y \in \mathcal{N}.\exists f \in \mathcal{F}.f(x) \neq f(y)$, *i.e.*, for every positive example x and every negative example y, there is a feature f that separates x and y. Once $\mathcal{F}$ has enough features, line 14 uses a standard Boolean-function learner [Mitchell et al. 1997; Padhi et al. 2016] BoolCombine to learn δ, a Boolean combination of these features, that separates $\mathcal{P}$ and $\mathcal{N}$. Then LoopInvGen logically strengthens the candidate invariant $\mathcal{I}$ by conjoining it with δ. For more details on this framework, we refer to the LoopInvGen paper [Padhi et al. 2016]. In particular, [Padhi et al. 2016] shows that breaking the binary classification problem of separating $\mathcal{P}$ and $\mathcal{N}$ by a candidate invariant into the two step approach of first inferring features that separate $P \subseteq \mathcal{P}$ and $N \subseteq \mathcal{N}$ and then combining the features is an effective approach to invariant inference. The features are usually much simpler than the invariants, which makes inferring features much more tractable than inferring candidate invariants.

Next, we discuss our contributions: the inference of relevant variables and the changes Oasis makes to LoopInvGen followed by our ILP-based learning (Section 5).

4 OASIS FRAMEWORK

In this section, we overview our approach for accelerating invariant inference using a set of *relevant* variables. First, we define the state space for programs and describe our encoding of the verification conditions described in Definition 3.2. We then describe the notion of relevant variables for a program verification problem, and present our approach for inferring sufficient loop invariants using these relevant variables.

4.1 Notation

Given a program **P** we write $\vec{x}_{\mathbf{P}}$, to denote the sequence $\langle x_1, \ldots, x_n \rangle$ of variables appearing in it. We omit the subscript **P** and simply write $\vec{x}$ when the program is clear from context. A program state for **P**, denoted $\vec{\sigma} = \langle v_1, \ldots, v_n \rangle$, is a sequence of values assigned to the program variables — any subset of these values may be *irrelevant* (denoted $\top$). A program state $\vec{\sigma}$ is said to be *total* if it does not contain $\top$, and is said to be *partial* otherwise. Finally, we use the shorthand $(\vec{x} \mapsto \vec{\sigma})$ to denote the value assignment predicate $(x_1 = v_1 \wedge \cdots \wedge x_n = v_n)$, where irrelevant values ($\top$) are simply dropped, *e.g.*, $(\langle x_1, x_2, x_3 \rangle \mapsto \langle v_1, \top, v_3 \rangle) \equiv (x_1 = v_1 \wedge x_3 = v_3)$.

Although, the techniques described in this work can be easily extended to programs containing multiple and nested loop, for simplicity, we consider verifying our single-loop program from the previous section: $\mathcal{L} \equiv \{\rho\}$ **while** G **do** S $\{\phi\}$. We formally model the loop in our program **P** as a transition relation $\textsf{TRANS}_{\mathbf{P}}$ over program states. Two states $\vec{\sigma}_1$ and $\vec{\sigma}_2$ are related by $\textsf{TRANS}_{\mathbf{P}}$ iff a single iteration of the loop body (S) transitions the state $\vec{\sigma}_1$ to $\vec{\sigma}_2$. We need $\textsf{TRANS}_{\mathbf{P}}$ to be a relation as programs can have non-determinism. Formally,

$$\textsf{TRANS}_{\mathbf{P}}(\vec{\sigma}_1, \vec{\sigma}_2) \iff \big\{G \wedge (\vec{x} \mapsto \vec{\sigma}_1)\big\} \; S \; \big\{(\vec{x} \mapsto \vec{\sigma}_2)\big\}$$

Similarly, we model the precondition and postcondition as unary predicates on program states:

$$\textsf{PRE}_{\mathbf{P}}(\vec{\sigma}) \iff (\vec{x} \mapsto \vec{\sigma}) \wedge \rho \qquad\qquad \textsf{POST}_{\mathbf{P}}(\vec{\sigma}) \iff (\vec{x} \mapsto \vec{\sigma}) \wedge \neg G \implies \phi$$

We omit the subscript **P** and simply write $\textsf{PRE}$, $\textsf{TRANS}$ and $\textsf{POST}$ when the program **P** is clear from context. These relations together define a program verification (or equivalently, a sufficient loop invariant inference) problem. Indeed, this encoding of program verification problems is a commonly used language-agnostic intermediate representation. Moreover, existing program analysis tools can automatically generate these relations from high-level programs and their formal specifications. This representation facilitates the use of off-the-shelf SMT solvers.

A sufficient loop invariant that establishes the correctness of the Hoare triple $\mathcal{L}$ is also a predicate defined over states of the program **P**. Such an invariant is required to satisfy the following verification conditions from Definition 3.1 in terms of the $\textsf{PRE}$, $\textsf{TRANS}$ and $\textsf{POST}$ relations above:

$$\underset{(\mathrm{VC_{pre}})}{\forall \vec{\sigma}.\, \textsf{PRE}(\vec{\sigma}) \implies \mathcal{I}(\vec{\sigma})} \qquad \underset{(\mathrm{VC_{ind}})}{\forall \vec{\sigma}, \vec{\sigma}'.\, \mathcal{I}(\vec{\sigma}) \wedge \textsf{TRANS}(\vec{\sigma}, \vec{\sigma}') \implies \mathcal{I}(\vec{\sigma}')} \qquad \underset{(\mathrm{VC_{post}})}{\forall \vec{\sigma}.\, \mathcal{I}(\vec{\sigma}) \implies \textsf{POST}(\vec{\sigma})}$$

Example 4.1. Consider again our motivating example from Figure 1 where $\vec{x} = \langle i, j, k, n, y \rangle$. We use $\vec{\sigma}$ and $\vec{\sigma}'$ to denote the tuples $\langle v_i, v_j, v_k, v_n, v_y \rangle$ and $\langle v_i', v_j', v_k', v_n', v_y' \rangle$ of values respectively. The following $\textsf{PRE}$, $\textsf{TRANS}$ and $\textsf{POST}$ relations encode the verification problem:

$$\begin{aligned}
\textsf{PRE}(\vec{\sigma}) &\triangleq (v_i = v_j = 0) \wedge (v_k \geq 0) \wedge (v_n \geq 0) \\
\textsf{TRANS}(\vec{\sigma}, \vec{\sigma}') &\triangleq (v_k' = v_k) \wedge (v_n' = v_n) \wedge \Big((v_i \leq v_n) \implies \big(v_i' = v_i + 1 \wedge v_j' = v_j + 1 \wedge v_y' = v_i \cdot v_j\big)\Big) \\
\textsf{POST}(\vec{\sigma}) &\triangleq \neg(v_i \leq v_n) \implies \big((v_i + v_j + v_k \geq 2 \cdot v_n) \vee (v_y \geq v_n^2)\big)
\end{aligned}$$

4.2 Relevant Variables

Scalability is a major challenge for existing data-driven invariant inference techniques. As number of variables increases the performance of these techniques degrades rapidly, although in many cases a sufficient invariant for verifying these programs contains only a small number of variables. We propose a novel technique that first identifies a small subset of variables over which a sufficient loop

Algorithm 2 Oasis framework for scaling loop invariant inference

```
   function Oasis(⟨PRE , TRANS , POST⟩ : Verification Problem, σ⃗+ : States, σ⃗− : States)
1      Classifier C ← Learn(σ⃗+, σ⃗−)
2      if C = ⊥ then return ⊥
3      Variables r⃗ ← FilterVariables(C)
4      do parallel
5          in thread 1 do
6              σ⃗ ← FindPosCounterExample(⟨PRE , TRANS , POST⟩, C)
7              if σ⃗ ≠ ⊥ then return Oasis(⟨PRE , TRANS , POST⟩, σ⃗+ ∪ {σ⃗}, σ⃗−)
8          in thread 2 do
9              σ⃗ ← FindNegCounterExample(⟨PRE , TRANS , POST⟩, C)
10             if σ⃗ ≠ ⊥ then return Oasis(⟨PRE , TRANS , POST⟩, σ⃗+, σ⃗− ∪ {σ⃗})
11         in thread 3 do
12             I ← RelInfer(⟨PRE , TRANS , POST⟩, σ⃗+, σ⃗−, r⃗) |timeout = τ
13             if I ≠ ⊥ then return I
```

invariant is likely to exist. Then, it simultaneously refines this subset and searches for a sufficient invariant till one is found.

Our core framework, called Oasis, is outlined in Algorithm 2. Oasis accepts the standard set of arguments for a data-driven verification technique (discussed in Section 3.3) — a verification problem (encoded as a triple $\langle \text{PRE}, \text{TRANS}, \text{POST} \rangle$), and some sampled positive ($\vec{\sigma}_+$) and negative ($\vec{\sigma}_-$) program states typically sampled randomly. We first invoke the Learn function with these sampled states to learn a predicate C that separates $\vec{\sigma}_+$ and $\vec{\sigma}_-$, *i.e.*,

$$\big(\forall \vec{\sigma} \in \vec{\sigma}_+ .\, C(\vec{\sigma})\big) \land \big(\forall \vec{\sigma} \in \vec{\sigma}_- .\, \neg C(\vec{\sigma})\big)$$

We detail the Learn function in Section 5, which utilizes machine-learning techniques to efficiently find a sparse separator for $\vec{\sigma}_+$ and $\vec{\sigma}_-$. In line 3 we drop irrelevant variables, those that do not affect the prediction of the classifier over $\vec{\sigma}_+ \cup \vec{\sigma}_-$, and consider the remaining variables $\vec{r} \subseteq \vec{x}$ to be a candidate set of relevant variables. In Section 2 we show some examples of classification problems, the learned classifiers and relevant variables.

After a set $\vec{r}$ of relevant variables is identified, in lines 3 – 12, we try to refine the set of relevant variables and find a sufficient invariant over them in parallel. In particular, we execute the following three threads in parallel:

(1) one that attempts to find a *positive* state misclassified by the classifier
(2) one that attempts to find a *negative* state misclassified by the classifier
(3) one that runs invariant inference using the currently identified relevant variables

Schohn and Cohn [2000] showed that classifiers can be refined by sampling near the classification boundary. This idea is used in Li et al. [2017], which showed that when compared to random sampling, active learning improves the quality of sampled program states and accelerates the search for sufficient invariants. Threads 1 and 2 are responsible for an active-learning-based refinement of the relevant variables set, and thread 3 attempts to find a sufficient invariant over these variables, if there exists one. Next, we detail our active learning strategy (Section 4.3) and our relevance-aware invariant inference algorithm RelInfer (Section 4.4). Note that the RelInfer thread is run with a timeout of τ so that long-running inference threads are automatically cleaned up as we spin up more threads with refined sets of relevant variables.

Algorithm 3 Procedures for refinement of candidate relevant variables

function FindPosCounterExample($\langle$PRE , TRANS , POST$\rangle$: Verification Problem, C : Predicate)
1 **for** $k = 0$ **to** ∞ **do**
2 Predicate Reachable(k) $\triangleq$ PRE $(\vec{\sigma}_0) \land$ TRANS $(\vec{\sigma}_0, \vec{\sigma}_1) \land \cdots \land$ TRANS $(\vec{\sigma}_{k-1}, \vec{\sigma}_k)$
3 Counterexample $\boldsymbol{c} \leftarrow$ Check($\forall \vec{\sigma}_0, \ldots, \vec{\sigma}_k$. Reachable($k$) $\implies C(\vec{\sigma}_k)$)
4 **if** $\boldsymbol{c} \neq \bot$ **then return** $\boldsymbol{c}[\vec{\sigma}_k]$

function FindNegCounterExample($\langle$PRE , TRANS , POST$\rangle$: Verification Problem, C : Predicate)
5 **for** $k = 0$ **to** ∞ **do**
6 Predicate Bad(k) $\triangleq \neg$POST$(\vec{\sigma}_k) \land$ TRANS $(\vec{\sigma}_{k-1}, \vec{\sigma}_k) \land \cdots \land$ TRANS $(\vec{\sigma}_0, \vec{\sigma}_1)$
7 Counterexample $\boldsymbol{c} \leftarrow$ Check($\forall \vec{\sigma}_0, \ldots, \vec{\sigma}_k$. Bad($k$) $\implies \neg C(\vec{\sigma}_0)$)
8 **if** $\boldsymbol{c} \neq \bot$ **then return** $\boldsymbol{c}[\vec{\sigma}_0]$

4.3 Refining Relevant Variables

We now detail our procedures for refining a set of relevant variables. The FindPosCounterExample and FindNegCounterExample procedures, which run in threads 1 and 2 respectively, are outlined in Algorithm 3. Each of these procedures returns a program state that is misclassified by the current classifier C, which is then used to learn a new classifier, and thus a new set of relevant variables.

The FindPosCounterExample procedure identifies *positive* misclassifications — a reachable program state $\vec{\sigma}$ that the classifier labels as a negative state, *i.e.*, $\neg C(\vec{\sigma})$. To identify such states, we gradually expand the frontier of reachable states starting from the precondition PRE and then repeatedly applying the transition relation TRANS . In line 2, we construct the predicate Reachable(k) that captures all states that are reachable in exactly k applications of the transition relation, *i.e.*, k iterations of the loop. In line 3, we check if all such states are subsumed by the current classifier. Upon finding a counterexample, in line 4, we return the misclassified state.

The FindNegCounterExample procedure works in a very similar manner and identifies *negative* misclassifications — a bad program state $\vec{\sigma}$ (one that would lead to violation of the final assertion) that the classifier labels as a positive state, *i.e.*, $C(\vec{\sigma})$. To identify such states, we gradually expand the frontier of known bad states starting from those that violate the postcondition POSTand then repeatedly reversing the transition relation TRANS . In line 6, we construct the predicate Bad(k) that captures all states that lead to state to an assertion violation in exactly k applications of the transition relation, *i.e.*, k iterations of the loop. In line 7, we check if all such states are excluded by the current classifier. Upon finding a counterexample, in line 8, we return the misclassified state.

Although these procedures can be computationally expensive, our implementation caches the results of intermediate queries for reuse. In particular, unsatisfiable paths are generated at most once.

4.4 Invariant Inference with Relevant Variables

Once we have a set of relevant variables from the learned classifier, we run our invariant inference algorithm (in thread 3) with these variables together with all the positive states ($\vec{\boldsymbol{\sigma}}_+$) and negative states ($\vec{\boldsymbol{\sigma}}_-$) sampled so far. In Algorithm 4 we outline this algorithm. They key difference with respect to Algorithm 1 is the use of $\vec{r}$ — the set of relevant variables. While Algorithm 1 learns features over all variables $\vec{x}$ in the program, Algorithm 4 only learns features over $\vec{r}$, which is provided to the Learn procedure in line 11. In the next section, we detail this relevance-aware learning procedure.

Algorithm 4 A loop invariant inference algorithm that utilizes relevant variable information

function RELINFER($\langle \textsf{PRE}, \textsf{TRANS}, \textsf{POST} \rangle$: Verification Problem, $\vec{\boldsymbol{\sigma}}_{\boldsymbol{+}}$: States, $\vec{\boldsymbol{\sigma}}_{\boldsymbol{-}}$: States, $\vec{r}$: Variables)
1 **if** CHECK($\forall \vec{\sigma}.\, \textsf{PRE}(\vec{\sigma}) \implies \textsf{POST}(\vec{\sigma})$) $\neq \bot$ **then throw** "No Solution!"
2 Predicate $\mathcal{I} \leftarrow \textsf{POST}$
3 **while** True **do**
4   **if** CHECK($\forall \vec{\sigma}, \vec{\sigma}'.\, \mathcal{I}(\vec{\sigma}) \wedge \textsf{TRANS}(\vec{\sigma}, \vec{\sigma}') \implies \mathcal{I}(\vec{\sigma}')$) $= \bot$ **then return** $\mathcal{I}$
5   States $\mathcal{P}, \mathcal{N} \leftarrow \vec{\boldsymbol{\sigma}}_{\boldsymbol{+}}, \vec{\boldsymbol{\sigma}}_{\boldsymbol{-}}$
6   **while** True **do**
7   Features $\mathcal{F} \leftarrow \{\}$
8   **while** True **do**
9   States $(P, N) \leftarrow$ CONFLICT($\mathcal{P}, \mathcal{N}, \mathcal{F}$)
10   **if** $P = N = \{\}$ **then break**
11   **else** $\mathcal{F} \leftarrow \mathcal{F} \cup$ LEARN($P, N, \vec{r}$)
12   Predicate $\delta \leftarrow$ BOOLCOMBINE($\mathcal{F}$)
13   Counterexample $\boldsymbol{c} \leftarrow$ CHECK($\forall \vec{\sigma}, \vec{\sigma}'.\, \delta(\vec{\sigma}) \wedge \mathcal{I}(\vec{\sigma}) \wedge \textsf{TRANS}(\vec{\sigma}, \vec{\sigma}') \implies \mathcal{I}(\vec{\sigma}')$)
14   **if** $\boldsymbol{c} = \bot$ **then break**
15         $\mathcal{N} \leftarrow \mathcal{N} \cup \{c[\vec{\sigma}]\}$
16     $\mathcal{I} \leftarrow (\mathcal{I} \wedge \delta)$
17   Counterexample $\vec{\sigma} \leftarrow \forall \vec{\sigma}.\, \textsf{PRE}(\vec{\sigma}) \implies \mathcal{I}(\vec{\sigma})$
18   **if** $\vec{\sigma} \neq \bot$ **then**
19         $\mathcal{I} \leftarrow \textsf{POST}$
20         $\vec{\boldsymbol{\sigma}}_{\boldsymbol{+}} \leftarrow \vec{\boldsymbol{\sigma}}_{\boldsymbol{+}} \cup \{\vec{\sigma}\}$

5 CLASSIFIER LEARNING

In this section, we formulate the problem of generating a classifier that separates positive program states from negative program states. By default, the output classifier predicate can use any of the program variables. If we restrict the classifier to use only a subset $\vec{r}$ of variables (*e.g.*, the call to LEARN procedure in line 11 of Algorithm 4) then we first project the examples to $\vec{r}$ and then learn a classifier over the projected states. Let $\mathbf{x}$ denote a vector of program variables that can occur in the classifier. In this section, we use the standard notation that bold letters denote vectors (*e.g.*, $\mathbf{0}$ is a vector of all zeros). We model the problem of inferring a classifier $h : \mathbb{Z}^{|\mathbf{x}|} \longrightarrow \{\textsf{True}, \textsf{False}\}$ as a search problem over the following class of CNF predicates with C denoting the number of conjuncts and D the number of disjuncts in each conjunct:

$$\mathcal{H}_{\text{CNF}} = \left\{ \bigwedge_{c \in [C]} \bigvee_{d \in [D]} \langle \mathbf{w}_{cd}, \mathbf{x} \rangle + b_{cd} > 0 \right\}. \tag{1}$$

where $b \in \mathbb{Z}$ and $\langle \mathbf{w}, \mathbf{x} \rangle + b$ is an inner product between a vector $\mathbf{w} \in \mathbb{Z}^{|\mathbf{x}|}$ and $\mathbf{x}$. We use $[n]$ to denote the list $\{0, 1, \ldots, n-1\}$.

Given a set of program states with corresponding labels, our task is to find a classifier $h \in \mathcal{H}_{\text{CNF}}$ such that **(a)** it separates the positive states from the negative states, and **(b)** it *generalizes* to unseen program states The first part is a *search* question, whereas the second part suggests learning to choose simple and *natural* predicates. Note that the class of invariants $\mathcal{H}_{\text{CNF}}$ is very powerful – one can trivially fit any given set of examples. We make the following observations. The search problem becomes meaningful on a given set of program states, if we restrict the predicate sizes (*i.e.*, C and D) to be small. Furthermore, the coefficients are often bounded by the constants occurring in the program. Finally, and most importantly, we are not dealing with arbitrary predicate formulas, but ones that have a nice conjunction-of-disjunctions structure. These observations enable reformulating the search problem as an integer-linear programming (ILP) problem that can be *efficiently* solved in practice for our benchmarks by off-the-shelf ILP solvers.

Consider the search problem **(1)** above: formally, we want to find a predicate $h \in \mathcal{H}_{\text{CNF}}$ that accurately classifies a given set of labeled program states $\{\sigma_n, y_n\}_{n=1}^{N}$, where $y_n \in \{0,1\}$. It is convenient to think of h as a tree of depth 3: the program variables form the input layer to the linear inequalities, which are grouped by $\bigvee$ operators to yield disjunctive predicates. The root node is the $\bigwedge$ operator that represents conjunction of the predicates represented by the second layer. The reduction of the search problem to ILP is given as follows.

(**Input layer: linear inequalities**) Write $z_{ncd} = \mathbb{1}\{\langle \mathbf{w}_{cd}, \sigma_n \rangle + b_{cd} > 0\} \in \{0,1\}$, where the indicator function $\mathbb{1}\{p\}$ of a predicate p maps True to 1 and False to 0. This is captured by the following constraints, for a sufficiently large integer M:

$$\forall n \in [N], c \in [C], d \in [D], \quad -M(1 - z_{ncd}) \; < \; \langle \mathbf{w}_{cd}, \sigma_n \rangle + b_{cd} \; \leq \; M z_{ncd},$$
$$\mathbf{w}_{cd} \in \mathbb{Z}^{|\mathbf{x}|}, b_{cd} \in \mathbb{Z}, z_{ncd} \; \in \; \{0,1\}. \tag{2}$$

(**Middle layer: Disjunctions**) Note that the value of the c-th conjunct on a given input σ_n corresponds to summing $z_{ncd} = \mathbb{1}\{p_{ncd}\}$ over d, *i.e.*, write $y_{nc}^{\vee} = \mathbb{1}\left\{\bigvee_{d \in [D]} p_{ncd}\right\}$. This is captured by the constraint:

$$\forall n \in [N], c \in [C], \quad -M(1 - y_{nc}^{\vee}) \; < \; \sum_{d \in [D]} z_{ncd} \; \leq \; M y_{nc}^{\vee},$$
$$y_{nc}^{\vee} \; \in \; \{0,1\}. \tag{3}$$

(**Final layer: Conjunction**) The predicted label on a given input state is given by a conjunction of the above disjunctions. Requiring that the predicted label match the observed label for each example is equivalent to the following constraints:

$$\text{for } n \in [N] \text{ s.t. } y_n = 1, \sum_{c \in [C]} y_{nc}^{\vee} \; \geq \; C \,,$$
$$\text{for } n \in [N] \text{ s.t. } y_n = 0, \sum_{c \in [C]} y_{nc}^{\vee} \; \leq \; C - 1 \,. \tag{4}$$

The search problem can now be stated as the ILP problem: *find a feasible integral solution* $\{z, y^{\vee}, \mathbf{w}, b\}$ *subject to the constraints Equations* (2) *to* (4) *combined.* Note that the problem formulation naturally handles partial states — if $(\sigma_n)_j$ is $\top$ then $(\mathbf{w}_{cd})_j$ is set to zero. Equation (2) is applied over only the variables that don't map to $\top$. This ILP satisfies the following properties, the proofs of which are straightforward and presented below for completeness. We abuse notation by using 0 (resp. 1) and False (resp. True) interchangeably.

Theorem 5.1. *Any feasible solution to the ILP problem* (5) *is a member of* $\mathcal{H}_{CNF}$.

Proof. Let $\{z, y^{\vee}, \mathbf{w}, b\}$ denote a feasible solution. First, note that for any fixed c, $y_{nc}^{\vee} = 0$ iff $\sum_{d \in [D]} z_{ncd} = 0$ because the conditions (3) hold. As $y_{nc}^{\vee} \in \{0,1\}$ and $z_{ncd} \in \{0,1\}$, it follows that $y_{nc}^{\vee} = \bigvee_{d \in [D]} z_{ncd}$, for each c. Next, it is immediate that $y_n = 1$ iff every $y_{nc}^{\vee} = 1$ because the conditions (4) hold. So, we have $y_n = \bigwedge_{c \in [C]} y_{nc}^{\vee}$. Finally, notice that for any n, c, d, $z_{ncd} = 1$ iff $\langle \mathbf{w}_{cd}, \sigma_n \rangle + b_{cd} > 0$ because the conditions (2) hold. In other words, $z_{ncd} = \mathbb{1}\{\langle \mathbf{w}_{cd}, \sigma_n \rangle + b_{cd} > 0\}$. Putting these together, we have, $y_n = \bigwedge_{c \in [C]} y_{nc}^{\vee} = \bigwedge_{c \in [C]} \bigvee_{d \in [D]} \langle \mathbf{w}_{cd}, \sigma_n \rangle + b_{cd} > 0 \in \mathcal{H}_{\text{CNF}}$. □

Theorem 5.2. *Given a set of labeled program states* $\{\sigma_n, y_n\}, n \in [N]$*, if there is a* $h \in \mathcal{H}_{CNF}$ *s.t.* $h(\sigma_n) = y_n$ *for all* $n \in [N]$*, then the ILP problem* (5) *has at least one feasible solution.*

Proof. This direction is easier to show. We can read off the integral coefficients $\mathbf{w}$ and b for all the polynomials from h, and obtain $z_{ncd} = \mathbb{1}\{\langle \mathbf{w}_{cd}, \sigma_n \rangle + b_{cd} > 0\}$ so that (2) hold. Then assign $y^{\vee}$ variables as in the proof of Claim 5.1, so that (3) hold. Finally, because $h(\sigma_n) = y_n$ holds for all $n \in [N]$, it follows that (4) also hold. We have a feasible solution. □

Now, consider the problem of learning *generalizable* predicates **(2)**. To this end, we follow the Occam's razor principle – seeking predicates that are "simple" and hence generalize better [Albarghouthi and McMillan 2013]. Simplicity in our case can be characterized by the size of the predicate clauses and the magnitude of the coefficients. One way to achieve this is by constraining the L_1-norm of the coefficients $\mathbf{w} = [w_1, \ldots, w_n]$, *i.e.*, by minimizing $\sum_{i \in [n]} |w_i|$. Note that L_1-norm can be expressed using linear constraints:[3] $\|\mathbf{w}\|_1 = \langle \mathbf{1}, \mathbf{w}^+ + \mathbf{w}^- \rangle$, where $\mathbf{w}^+ \geq 0$ and $\mathbf{w}^- \geq 0$ (componentwise inequality) such that $\mathbf{w} = \mathbf{w}^+ - \mathbf{w}^-$.

However, focusing only on the magnitude may lead to poor solutions. For example, consider the Hoare triple: $\{n \geq 0 \land x = n \land y = 0\}$ **while** `(x > 0)` **do** `{s ← y++; x--;}` $\{y = n\}$. Here, it is easy to verify that the loop invariant $x + y = n$ is sufficient to assert $\mathrm{VC}_{\mathrm{post}}$. The equivalent predicate in $\mathcal{H}_{\mathrm{CNF}}$, $x + y - n \geq 0 \land n - x - y \geq 0$, however has a larger L_1-norm though the invariant is a simple equality. So, simply minimizing the L_1-norm is not sufficient. Existing solvers [Padhi et al. 2019, 2016] employ heuristics such as preferring equality to inequality. We handle this by explicitly penalizing the *inclusion* of variables in the solution by using a penalty μ where $\mu_j = 0$ iff $\forall c \in [C]. \forall d \in [D]. (\mathbf{w}_{cd})_j = 0$. Intuitively, the more the number of variables with non-zero coefficients in the classifier, the more the penalty. Our final objective function combines both μ and L_1-norm penalties:

$$
\begin{aligned}
&\min_{\mathbf{w}, \mathbf{w}^+, \mathbf{w}^-, b, z, y^{\vee}, \mu} \sum_{c \in [C], d \in [D]} \langle \mathbf{1}, \mathbf{w}^+_{cd} + \mathbf{w}^-_{cd} \rangle \quad + \quad \lambda \langle \mathbf{1}, \mu \rangle \\
&\qquad \text{subject to Equations (2) to (4), and} \\
&\mathbf{1} - M(\mathbf{1} - \mu) \quad \leq \sum_{c \in [C], d \in [D]} \mathbf{w}^+_{cd} + \mathbf{w}^-_{cd} \quad \leq \quad M\mu, \\
&\qquad \forall c \in [C], d \in [D], \quad \mathbf{w}_{cd} \quad = \quad \mathbf{w}^+_{cd} - \mathbf{w}^-_{cd}, \\
&\qquad \mathbf{w}^+_{cd} \geq 0\,, \quad \mathbf{w}^-_{cd} \geq 0\,, \quad \mu \quad \in \quad \{0, 1\}^{|\mathbf{x}|}.
\end{aligned} \tag{5}
$$

The key advantage of the above ILP formulation is that it can be solved optimally by off-the-shelf solvers that leverage continuous and integer optimization techniques to solve such problems. This enables efficient and scalable search compared to enumerative techniques.

We now formally give the implementation of Learn procedure in Figure 3. GenerateExpression (in Step 3) executed on the solution $\{z, y^{\vee}, \mathbf{w}, b\}$ to the ILP problem Equation (5) outputs the expression $\bigwedge_{c \in [C]} \bigvee_{d \in [D]} \langle \mathbf{w}_{cd}, \mathbf{x} \rangle + b_{cd} > 0$.

[3] Integrality constraints on $\mathbf{w}^+$, $\mathbf{w}^-$ aren't needed, so the problem as stated is technically a mixed ILP.

function LEARN(*data*)
$\{z, y^{\vee}, \mathbf{w}, b\} \leftarrow$ Solve Equation (5) with labeled program states $\{\sigma_n, y_n\}_{n=1}^{N}$ from *data*
expr $\leftarrow$ GENERATEEXPRESSION($\{z, y^{\vee}, \mathbf{w}, b\}$) ▷ returns a CNF expression over vars
return *expr*

Fig. 3. Implementation of LEARN procedure using the ILP formulation.

In practice, it suffices to restrict $\mathbf{w}$ and b to a small set of integers in Equation (2), and M to be a very large integer. In the evaluation below, we use only two disjuncts[4], *i.e.*, $C = 1$ and $D = 2$ in Equation (5), constrain the coefficients to integers within $[-1000, 1000]$ and use $M = 100,000$ in Equation (5). In Figure 4, we show the constraints generated by our ILP formulation for the Hoare triple $\{x = y = 0\}$ **while** `(x ≥ 0)` **do** `{x ← x + y}` $\{False\}$ from Gulavani et al. [2006]. We use the data in Table 4 to list our constraints.

x	y	ℓ
0	0	1
-1	⊤	0

Table 4. Execution data.

Constraints for Equation (2)

$$-M(1 - z_{111}) < 0 * \mathbf{w}_{111} + 0 * \mathbf{w}_{112} \leq z_{111}$$
$$-M(1 - z_{121}) < 0 * \mathbf{w}_{122} + 0 * \mathbf{w}_{122} \leq z_{112}$$
$$-M(1 - z_{111}) < -1 * \mathbf{w}_{112} \leq z_{211}$$
$$-M(1 - z_{121}) < -1 * \mathbf{w}_{111} \leq z_{212}$$
$$z_{111}, z_{112}, z_{211}, z_{212} \in \{0, 1\}$$

Constraints for Equation (3)

$$-M(1 - y_{11}^{\vee}) < z_{111} + z_{112} \leq M y_{11}^{\vee}$$
$$-M(1 - y_{21}^{\vee}) < z_{211} + z_{212} \leq M y_{21}^{\vee}$$
$$y_{11}^{\vee}, y_{21}^{\vee} \in \{0, 1\}$$

Constraints for Equation (4)

$$y_{11}^{\vee} \geq 1$$
$$y_{21}^{\vee} \leq 0$$

Constraints for Equation (5)

$$\mathbf{w}_{111} = \mathbf{w}_{111}^{+} + \mathbf{w}_{111}^{-}$$
$$\mathbf{w}_{112} = \mathbf{w}_{112}^{+} + \mathbf{w}_{112}^{-}$$
$$\mathbf{w}_{121} = \mathbf{w}_{121}^{+} + \mathbf{w}_{121}^{-}$$
$$\mathbf{w}_{122} = \mathbf{w}_{122}^{+} + \mathbf{w}_{122}^{-}$$
$$1 - M(1 - \mu_1) < \mathbf{w}_{111}^{+} + \mathbf{w}_{111}^{-} \leq M\mu_1$$
$$1 - M(1 - \mu_1) < \mathbf{w}_{121}^{+} + \mathbf{w}_{121}^{-} \leq M\mu_1$$
$$1 - M(1 - \mu_2) < \mathbf{w}_{112}^{+} + \mathbf{w}_{112}^{-} \leq M\mu_2$$
$$1 - M(1 - \mu_2) < \mathbf{w}_{122}^{+} + \mathbf{w}_{122}^{-} \leq M\mu_2$$
$$\mu_1, \mu_2 \in \{0, 1\}$$

Objective function to minimize

$$\min_{\mathbf{w}, \mathbf{w}^{+}, \mathbf{w}^{-}, z, y^{\vee}, \mu} \mathbf{w}_{111}^{+} + \mathbf{w}_{111}^{-} + \mathbf{w}_{121}^{+} + \mathbf{w}_{121}^{-}$$
$$+\mathbf{w}_{112}^{+} + \mathbf{w}_{112}^{-} + \mathbf{w}_{122}^{+} + \mathbf{w}_{122}^{-} + \lambda(\mu_1 + \mu_2)$$

Fig. 4. Example to show the constraints generated by our ILP formulation.

6 EXPERIMENTAL EVALUATION

We have implemented OASIS using the LOOPINVGEN [Padhi et al. 2016] framework in OCaml, and using Z3 [de Moura and Bjørner 2008] as the theorem prover for checking validity of the

[4] An equality requires two disjuncts: we flip the labels (y_n) in Equation (5) and negate the optimal predicate.

verification conditions. We implement our technique for reducing the classification problem to ILPs in a Python script, which discharges the ILP subproblems to the OR-Tools [web 5 23a] optimization package. FindPosCounterExample and FindNegCounterExample procedures in Algorithm 3 are implemented in a python script and we use Z3 [de Moura and Bjørner 2008] to solve the constraints. We evaluate Oasis on commodity hardware — a CPU-only machines with 2.5GHz Intel Xeon processor, 32 GB RAM, and running Ubuntu Linux 18.04.

Solvers. We compare Oasis, against three tools: **(a)** LoopInvGen [Padhi et al. 2016] which uses data-driven invariant inference, **(b)** CVC4 [Barrett et al. 2011; Reynolds et al. 2015] which uses a refutation-based approach, and **(c)** DradSynth [Huang et al. 2020] which uses a combination of enumerative and deductive synthesis (cooperative synthesis). CVC4 and LoopInvGen are respectively the winners of the invariant-synthesis (Inv) track of SyGuS-Comp'19 [syg 5 14] and SyGuS-Comp'18 [Alur et al. 2019]. Recently, Huang et al. [2020] showed that their cooperative synthesis technique is able to perform better than LoopInvGen and CVC4 on invariant synthesis tasks.

Benchmark	**# Variables**			**# Instances**
	Median	**Average**	**Maximum**	
Sygus 2018	3	4	9	127
Unconfounded	10	11	22	92
Confounded1	15	16	32	92
Confounded5	25	26	44	92

Table 5. Statistics of the 403 SyGuS instances used for evaluation.

Benchmarks. We evaluate our technique on 403 instances which were part of the SyGuS-Comp'19 [syg 5 14] and also studied by Huang et al. [2020]. All these instances require reasoning over linear arithmetic. Out of these 403 instances, 276 were published by Si et al. [2018] and 127 were part of the SyGuS-Comp'18. The 276 instances are divided into three groups of 92 instances each **(a)** Unconfounded, **(b)** Confounded1, and **(c)** Confounded5. Confounded1 and Confounded5 instances [Si et al. 2018] were obtained by adding irrelevant variables to each of the Unconfounded instances. The number of irrelevant variables ranges from 4-9 in Confounded1 and from 12-23 in Confounded5. In Table 5, we give key statistics of these benchmarks. These instances are provided as a collection of logic formulas representing the VCs (Section 3.1) in the SyGuS grammar [Raghothaman et al. 2019].

Tool	**Solved (out of 403)**
CVC4	287
DradSynth	346
LoopInvGen	272
Oasis	**353**

Table 6. Comparison of Oasis with SyGuS tools on the 403 instances which were part of the SyGuS-Comp'19 [syg 5 14] invariant synthesis track.

Tool	**Solved**
CVC4	0
DradSynth	11
LoopInvGen	1
Oasis	**13**

Table 7. Number of uniquely solved instances by each tool.

6.1 Results on SyGuS Benchmarks

Comparison with SyGuS Competitors. We report the number of instances each tool solves with a timeout of 30 minutes[5] in Table 6. Oasis synthesizes sufficient loop invariants on **353** instances, **7** more than the second best tool and **66** more than CVC4, the winner of invariant-synthesis (Inv) track of SyGuS-Comp'19. Oasis is able to solve **13** instances which no other tool can solve. In Table 7, we list the number of unique benchmarks each tool solves. Out of the 353 instances that Oasis solves, 262 instances had disjunctive invariants.

Tool	Unconfounded (out of 92)	Confounded1 (out of 92)	Confounded5 (out of 92)	Solved (out of 276)
LoopInvGen	62	59	44	165
Oasis	84	84	67	**235**

Table 8. Comparison of Oasis and LoopInvGen, the state-of-the-art data driven tool, on the 276 instances which were part of the SyGuS-Comp'19 [syg 5 14] and studied by Si et al. [2018]. These 276 instance are more complex than the remaining 127 instances in terms of the number of variables present in them. These results indicate Oasis scales than better LoopInvGen on program with large number of variables.

Comparison with Data-Driven Tools. Oasis solves 81 more benchmarks than LoopInvGen, which is the state-of-the-art data-driven invariant inference tool, the winner of SyGuS-Comp'18 [Alur et al. 2019], and runner up of SyGuS-Comp'19 [syg 5 14]. In Table 8, we give break down of the number of instances LoopInvGen and Oasis solves in each category of the 276 instances from Si et al. [2018] to show how the complexity of benchmarks affects the performance of data-driven tools. Oasis solves 70 more benchmarks than LoopInvGen, indicating that Oasis scales better to programs with large number of variables. Recently, Si et al. [2018] (code2inv) and Ryan et al. [2020] (cln2inv) propose neural network based approaches for inferring invariants. We evaluate these two tools on Unconfounded instances[6], code2inv solves 64 and cln2inv solves 86 instances within 30 minutes. Garg et al. [2016] and Zhu et al. [2018] are two other data-driven invariant inference tools. From Table 5, the complexity of the 127 instances from SyGuS-Comp'18 is lower than that of the 276 instances from Si et al. [2018]. The 127 instances from SyGuS-Comp'18 subsumes the benchmarks these two tools were evaluated on and Oasis can solve all the instances that they succeeded on.

In Figure 5, we plot and compare the solving time for Oasis and LoopInvGen on the 403 instances. Oasis is slower that LoopInvGen on most benchmarks because it runs many Z3 queries to refine the set of relevant variables. Identifying relevant variables helps Oasis scale to programs with large number of variables and Oasis solves fewer instances without it (Section 6.2).

[5] [Huang et al. 2020] uses a timeout of 30 minutes and we keep the same timeout.

[6] Confounded1 and Confounded5 instances are only available as logic formulas and code2inv/cln2inv require C files as input. It is not straightforward to translate the constraints to C while maintaining a fair comparison.

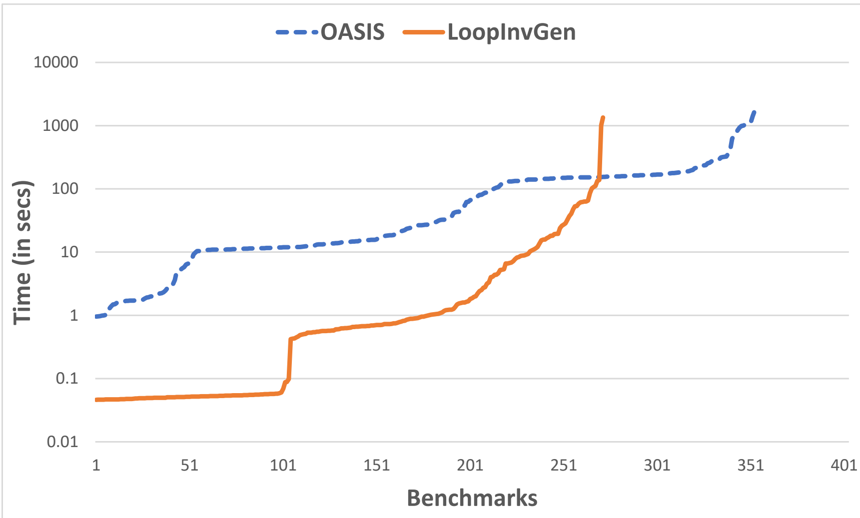

Fig. 5. Solving time comparison of Oasis and LoopInvGen on the 403 SyGuS instances.

Table 9. Details of Oasis invariant synthesis time, LoopInvGen invariant synthesis time, total variables in the instance, number of relevant variable used by Oasis in the successful run of RelInfer, number of variables in the invariant of the corresponding Unconfounded instance (Gold Solution) and size of the synthesized invariant for Confounded1 and Confounded5 instances solved by Oasis. A '-' indicates that LoopInvGen times out on that instance.

Benchmark	Oasis Time	LoopInvGen Time	# Variables	# Relevant Variables	Gold Solution	Size
1_conf1	32.46	-	15	10	2	28
2_conf1	20.83	-	15	6	2	28
3_conf1	142.35	-	18	9	3	11
5_conf1	747.97	-	20	11	3	11
10_conf1	54.31	0.68	14	6	1	24
11_conf1	13.8	0.75	14	6	1	24
12_conf1	41.67	0.69	14	6	1	24
13_conf1	15.58	0.73	14	6	1	24
15_conf1	156.51	-	20	9	2	32
16_conf1	191.54	-	20	9	2	7
17_conf1	322.36	-	20	14	2	32
18_conf1	364.25	-	20	9	2	7
23_conf1	76.15	-	15	6	2	60
24_conf1	26.3	-	15	6	2	60
28_conf1	67.35	-	11	5	2	7
29_conf1	272.63	-	11	8	2	11
38_conf1	140.04	12.01	16	7	1	3
40_conf1	158.92	0.97	16	6	1	3
41_conf1	159.94	0.96	16	7	2	19
42_conf1	620.74	11.46	16	6	1	3
43_conf1	158.4	4.39	16	5	1	3
44_conf1	157.74	5.23	16	5	1	3
45_conf1	163.14	18.16	16	7	1	3
46_conf1	272.05	-	16	7	2	18
47_conf1	159.41	19.41	16	7	1	3
48_conf1	158	11.01	16	7	1	3

49_conf1	158.05	15.35	16	7	1	3
56_conf1	234.26	-	16	6	2	7
57_conf1	168.95	0.82	16	6	2	7
65_conf1	11.46	0.05	14	6	2	22
71_conf1	140.38	110.69	21	4	1	3
77_conf1	165.67	0.75	16	8	2	7
78_conf1	141.53	37.24	16	6	1	3
79_conf1	140.24	32.57	16	6	1	3
91_conf1	14.82	-	12	5	2	7
94_conf1	80.46	-	19	10	4	14
95_conf1	177.43	1.36	20	10	3	53
96_conf1	186.7	0.89	20	10	2	53
97_conf1	15.91	0.69	20	5	1	3
98_conf1	14.77	0.68	20	5	1	3
99_conf1	86.83	1.86	17	8	3	39
100_conf1	26.67	-	17	8	3	30
103_conf1	10.91	0.05	9	3	1	19
107_conf1	281.15	-	21	9	3	11
108_conf1	140.97	1.08	23	5	2	7
109_conf1	324.32	-	23	11	3	11
110_conf1	54.29	8.79	17	8	2	63
111_conf1	144.44	9.14	17	8	2	63
114_conf1	13.79	1.06	16	6	1	19
115_conf1	15.71	0.86	16	6	1	19
118_conf1	15.71	8.75	17	8	2	63
119_conf1	18.63	8.86	17	8	2	63
120_conf1	21.32	-	15	6	2	56
121_conf1	25.12	-	15	6	2	56
124_conf1	101.6	-	19	10	4	47
125_conf1	32.25	-	19	10	4	47
130_conf1	114.27	-	32	12	3	11
131_conf1	104.58	-	32	12	3	11
132_conf1	1087.6	1.59	26	14	1	3
1_conf5	62.36	-	24	9	2	14
2_conf5	25.22	-	24	6	2	14
10_conf5	23.32	1.97	23	6	1	7
11_conf5	27.18	1.82	23	6	1	7
12_conf5	73.78	2.52	23	6	1	7
13_conf5	27.07	2.78	22	6	1	7
16_conf5	1131.16	-	29	9	2	7
18_conf5	215.43	-	29	9	2	7
24_conf5	849.02	-	24	8	2	41
25_conf5	14.68	-	17	3	1	3
28_conf5	1001.83	-	19	10	2	7
30_conf5	13.48	-	17	3	1	3
36_conf5	189.78	1.03	24	6	1	3
38_conf5	152.22	28.63	26	7	1	3
40_conf5	166.24	15.81	26	6	1	3
41_conf5	174.61	2.74	26	7	2	7
42_conf5	165.92	40.38	26	6	1	3
43_conf5	163.34	13.57	26	5	1	3
44_conf5	180.09	58.54	24	5	1	3
45_conf5	171.74	62.23	26	7	1	3
46_conf5	178.62	-	26	7	2	27
47_conf5	167.27	52.93	26	7	1	3
48_conf5	169.15	63.3	26	7	1	3
49_conf5	165.85	53.48	24	7	1	3
50_conf5	164.16	1.54	24	5	1	3
51_conf5	199.07	1.23	24	7	1	3
56_conf5	177.1	26.27	26	6	2	7
57_conf5	700.96	1.49	24	6	2	7
63_conf5	90.45	-	23	6	2	7
64_conf5	84.12	-	23	8	2	7
65_conf5	122.15	-	23	9	2	7
67_conf5	88	-	25	8	2	7
68_conf5	289.74	-	25	8	3	14
70_conf5	122.22	-	25	8	3	11

71_conf5	146.19	61.43	29	4	1	3
77_conf5	171.97	1.23	24	8	2	7
78_conf5	151.23	102.33	24	6	1	3
79_conf5	183.15	-	24	6	1	3
83_conf5	236.27	-	23	6	2	7
84_conf5	32.48	-	23	6	2	7
85_conf5	162.42	-	23	6	2	7
91_conf5	24.29	1.02	20	5	2	7
94_conf5	185.83	-	27	10	4	17
95_conf5	259.72	24.03	29	10	3	53
96_conf5	215.47	47.57	29	10	3	53
97_conf5	35.41	0.8	29	5	1	3
98_conf5	26.88	0.7	29	5	1	3
99_conf5	42.8	2.01	26	8	3	39
100_conf5	180.4	-	26	8	3	27
101_conf5	147.21	-	19	5	2	11
102_conf5	106.3	-	19	5	2	11
103_conf5	18.53	0.95	17	6	1	3
107_conf5	162.78	-	30	9	3	11
108_conf5	149.05	3.31	32	5	2	7
114_conf5	23.81	1.05	25	6	1	31
115_conf5	22.33	0.93	25	6	1	31
120_conf5	66.52	-	24	6	2	37
124_conf5	1068.72	-	28	10	4	47
128_conf5	21.2	18.11	19	5	1	3
132_conf5	1681.38	5.35	38	14	1	3
133_conf5	17.97	27.28	19	5	2	7

Analysis. In Table 9, we give detailed statistics of 120 instances which are Confounded1 and Confounded5 instances that Oasis solves[7]. Oasis takes 183.77 seconds on average for invariant synthesis. LoopInvGen times out on 51 instances and averages 15.36 seconds on the 69 instances it solves. We give the total number of variables in the instance, the number of relevant variables Oasis uses to synthesize the invariant and the number of variables in the gold solution, *i.e.*, the number of variables appearing in the invariant of the corresponding Unconfounded instance. Oasis reduces the number of variables it uses to solve the problem by 3× on average. Size of the invariant is computed as the number of nodes in the SyGuS AST [Raghothaman et al. 2019] of the invariant.

6.2 Ablation Study

In Section 6.1, we saw that Oasis solves more instances than any other tool. Now, through this study, we try to answer the following questions about Oasis:

(1) *Does identifying relevant variables really help?*
(2) *Is our ILP formulation better than other techniques?*
(3) *Is our relevant variable identifying algorithm by itself sufficient to guess invariants?*

Tool	**Solved (out of 403)**
Oasis, No Vars Select	326
Oasis	**353**

Table 10. Comparison of Oasis and Oasis without our relevant variable identifying algorithm. Oasis, No Vars Select uses all the variables appearing in the program as relevant variables.

To show that inference of relevant variables helps in solving more instances, we compare our tool, Oasis, against the following configuration:

7 We don't include the instances where the post-condition is a sufficient invariant.

function OASIS, NAIVE VARS SELECT($\langle$PRE , TRANS , POST$\rangle$: Verification Problem, $\vec{\sigma}_+$: States, $\vec{\sigma}_-$: States, Vars: Variables in Problem)

  for each subset s of VARS **do**

    Variables $\vec{r} \leftarrow s$

    $\mathcal{I} \leftarrow$ RELINFER($\langle$PRE , TRANS , POST$\rangle$, $\vec{\sigma}_+, \vec{\sigma}_-, \vec{r})\big|_{\text{timeout} = \tau}$

    if $\mathcal{I} \neq \bot$ **then return** $\mathcal{I}$

Fig. 6. Implementation of OASIS with naive enumeration strategy. $\vec{r}$ is the set of relevant variables.

(1) **OASIS, NO VARS SELECT.** We don't use our relevant variables identifying algorithm, *i.e.*, we don't execute lines 1-3, thread 1 and thread 2 in Algorithm 2. We use all the variables appearing in the program to infer the invariant. We still use our ILP formulation (Section 5) for the LEARN function in Algorithm 4.

We observe from Table 10 that inferring relevant variables helps solve 27 more instances. Moreover, these results also show that out of the 81 benchmarks that OASIS solves more than LOOPINVGEN, 54 are because of replacing the exhaustive enumeration-based feature synthesizer in LOOPINVGEN by ILP and 27 are because of the ILP-based relevant variable inference.

Tool	**Solved (out of 403)**
OASIS, NAIVE VARS SELECT	258
OASIS+DT	338
OASIS, COMPLETE MAPS	336
OASIS, NO OPTIMIZATION	297
OASIS	**353**

Table 11. Comparison of OASIS and OASIS with our ILP formulation replaced with naive enumeration, decision tree, ILP without objective function and ILP with complete maps instead of partial maps.

Next, to show that our ILP formulation is better suited for inferring invariants, we compare OASIS to the following configurations:

(1) **OASIS, NAIVE VARS SELECT.** We use an exhaustive enumeration strategy to find the set of relevant variables instead of using our ILP formulation (Section 5) to identify this set. We replace our Algorithm 2 with the implementation in Figure 6.
(2) **OASIS+DT.** We use scikit-learn [web 5 14] implementation of decision tree in place of our ILP formulation (Section 5) for the LEARN function in Algorithm 2 to find the set of relevant variables. There is no constraint on the height of the tree that the decision tree algorithm can learn.
(3) **OASIS, COMPLETE MAPS.** Instead of using the partial maps returned by Z3 while sampling states in Algorithm 3, we use complete maps. Partial maps are completed by replacing the don't care values with random integers.
(4) **OASIS, NO OPTIMIZATION.** We ignore the objective function used in Equation (5), which biases our learner towards simple classifiers, and instead only search for solutions that satisfy the constraints generated by our ILP formulation. This learner with no optimization is used for the LEARN function in both Algorithm 2 and Algorithm 4.

Again, in the configurations 1, 2 and 3, we still use our ILP formulation (Section 5) for the LEARN function in Algorithm 4.

We observe from Table 11 that using a simple strategy like enumerating over combinations of variables is not adequate to find the set of relevant variables. Also, this strategy is not scalable for problems with a large number of variables. Decision trees have been widely used for classification tasks [Garg et al. 2016; Zhu et al. 2018]. Our ILP formulation performs slightly better than decision trees because of the objective function which has specific penalties to learn simple and generalizable expressions. We also observe that running our ILP formulation without the objective function results in expressions without any constraints on the coefficients of the variables and the size of expression and we solve far less number of instances than running with the objective function. This shows that solving only the search problem in our ILP formulation is insufficient for generalization and learning expressions consistent with the Occam's razor principle helps in solving more instances.

Tool	**Solved (out of 403)**
Oasis, No RelInfer	82
Oasis	**353**

Table 12. Comparison of Oasis and Oasis without our relevance-aware invariant inference algorithm (thread 3) in Algorithm 2.

Finally, one might think that the classifier that separates reachable and bad states during the inference of relevant variables might be a good guess for a sufficient loop invariant. To show that this classifier is usually not an invariant, we compare Oasis against the following configuration:

(1) **Oasis, No RelInfer.** We don't run RelInfer, *i.e.*, thread 3 in Algorithm 2. We use the classifier C returned by the Learn function in Algorithm 2 as our invariant guess.

From Table 12, we observe that the classifier learnt between the good and bad states in Algorithm 2 is not usually a sufficient invariant. However, it is still an indicator of the relevant variables which appear in the sufficient invariant.

7 RELATED WORK

Loop invariant inference is a challenging problem with a long history. Although, we focus on numerical invariants in this paper, invariant inference over practical programs can be reduced to numerical reasoning [Ball et al. 2001]. The existing techniques for numerical loop invariants can be classified in two categories: those that are purely static and infer invariants from program text and data-driven approaches that guess invariants from examples of program states. The traditional static approaches for inferring loop invariants include abstract interpretation [Cousot and Cousot 1977; Cousot et al. 2005], predicate abstraction [Ball et al. 2001; Godefroid et al. 2010], interpolation [Henzinger et al. 2004; Jhala and McMillan 2006], constraint solving [Colón et al. 2003], and abductive inference [Dillig et al. 2012, 2013]. Although these approaches are mature and can scale to large programs, the data-driven approaches are more recent and the scalability is currently limited. However, data-driven invariant inference techniques (*e.g.*, Garg et al. [2014]; Padhi et al. [2016]) have been shown to outperform static approaches for verification of small but non-trivial loops.

Oasis reduces the problem of invariant inference to solving a constrained ILP problem, where a solver minimizes a penalty while maintaining the feasibility of constraints. Similar to us, [Colón et al. 2003; Gulwani et al. 2008; Sankaranarayanan et al. 2006] also reduce invariant inference to constraint solving. However, their constraints are non-linear and much harder to solve. Subsequently, Gupta et al. [2013] use data to make these constraints linear. However, this line of work either doesn't support disjunctive invariants or requires the number of disjunctions to be fixed by a user-provided template. Oasis has no such restrictions and can generate invariants that are arbitrary Boolean combinations of linear inequalities. Moreover, these techniques only solve the feasibility problem and do not have penalty terms. Although we can encode the search component of our ILP problem as an SMT constraint [Garg et al. 2014], our penalty terms are effective at generalization (Table 11). Thus, we use an optimization framework instead of a constraint solving framework like prior works.

Unlike [Nguyen et al. 2017, 2012; Sharma et al. 2013b,a], which are data-driven techniques for non-linear invariants, Oasis focuses on linear invariants. This design is primarily motivated by the presence of hundreds of benchmarks from the SyGuS competition. There are only a few benchmarks for non-linear invariants and these can already be solved well by existing techniques [Nguyen et al. 2017; Yao et al. 2020]. Additionally, since Oasis is implemented on top of LoopInvGen, it inherits the capabilities to infer invariants for multiple loops and nested loops from Padhi et al. [2016].

Prior work on data-driven techniques to infer arbitrary Boolean combinations of linear inequalities have all been evaluated on benchmarks at the scale of the SyGuS'18 benchmarks (Table 5) that have less than ten variables. These include techniques that use SMT solvers directly [Garg et al. 2014], PAC-learning [Sharma et al. 2013a], decision trees [Garg et al. 2016], SVMs [Li et al. 2017], and combinations of SVMs and decision trees [Zhu et al. 2018]. Techniques based on neural networks [Ryan et al. 2020; Si et al. 2018] have been evaluated at the scale of Unconfounded benchmarks (Table 5). Oasis uses ILP to scale data-driven inference to succeed on benchmarks with even more variables.

8 CONCLUSIONS

Oasis makes the following contributions. Conceptually, Oasis reduces the problem of invariant inference to learning relevant variables and learning features. Technically, Oasis provides a novel ILP-based learner which generates sparse classifiers and solves both these problems effectively. Practically, Oasis outperforms the state-of-the-art tools, including the most recent work of Huang et al. [2020], on benchmarks from the invariant inference track of the Syntax Guided Synthesis competition. Oasis both solves more benchmarks and can solve benchmarks that no other tool could solve before. We are working towards integrating Oasis with a full-fledged verification system for effective verification of complete applications.

Stepping back, the inference of loop invariants is an old problem with a rich history. Many techniques have been applied to this problem and they all have their strengths and weaknesses. Data-driven invariant inference techniques can handle challenging loops with confusing program text by applying ML techniques to mine patterns directly from data. However, these techniques have been evaluated only on loops with a small number of variables. This weakness is clear on benchmarks with large number of irrelevant variables. Oasis uses ML to infer the relevant variables which leads to simpler verification problems with fewer variables. We believe that this idea of simplifying the verification problems using ML is generally applicable. Oasis demonstrates that ML-based simplification is effective for data-driven invariant inference and we will explore it in other contexts in the future.

REFERENCES

Accessed: 2019-05-23a. OR-Tools – Google Optimization Tools. https://github.com/google/or-tools.
Accessed: 2019-05-23b. The International Competition on Software Verification. https://sv-comp.sosy-lab.org.
Accessed: 2019-05-23c. The Syntax-Guided Synthesis Competition. https://sygus.org.
Accessed: 2020-05-14. Decision Tree - Scikit-Learn. https://scikit-learn.org/stable/modules/tree.html.
Accessed: 2020-05-14. The Syntax-Guided Synthesis Competition 2019. https://sygus.org/comp/2019/.

Aws Albarghouthi, Sumit Gulwani, and Zachary Kincaid. 2013. Recursive Program Synthesis. In *Computer Aided Verification - 25th International Conference, CAV 2013, Saint Petersburg, Russia, July 13-19, 2013. Proceedings (Lecture Notes in Computer Science, Vol. 8044)*, Natasha Sharygina and Helmut Veith (Eds.). Springer, 934–950.

Aws Albarghouthi and Kenneth L. McMillan. 2013. Beautiful Interpolants. In *Computer Aided Verification - 25th International Conference, CAV 2013, Saint Petersburg, Russia, July 13-19, 2013. Proceedings (Lecture Notes in Computer Science, Vol. 8044)*, Natasha Sharygina and Helmut Veith (Eds.). Springer, 313–329.

Rajeev Alur, Dana Fisman, Saswat Padhi, Rishabh Singh, and Abhishek Udupa. 2019. SyGuS-Comp 2018: Results and Analysis. *CoRR* abs/1904.07146 (2019). http://arxiv.org/abs/1904.07146

Rajeev Alur, Arjun Radhakrishna, and Abhishek Udupa. 2017. Scaling Enumerative Program Synthesis via Divide and Conquer. In *Tools and Algorithms for the Construction and Analysis of Systems - 23rd International Conference, TACAS 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings, Part I (Lecture Notes in Computer Science, Vol. 10205)*, Axel Legay and Tiziana Margaria (Eds.). 319–336.

Roberto Bagnara, Patricia M. Hill, and Enea Zaffanella. 2006. Widening operators for powerset domains. *Int. J. Softw. Tools Technol. Transf.* 8, 4-5 (2006), 449–466.

Thomas Ball, Rupak Majumdar, Todd D. Millstein, and Sriram K. Rajamani. 2001. Automatic Predicate Abstraction of C Programs. In *Proceedings of the 2001 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. ACM. https://doi.org/10.1145/378795.378846

Clark Barrett, Christopher L. Conway, Morgan Deters, Liana Hadarean, Dejan Jovanovic, Tim King, Andrew Reynolds, and Cesare Tinelli. 2011. CVC4. In *Computer Aided Verification - 23rd International Conference (CAV) (Lecture Notes in Computer Science, Vol. 6806)*. Springer. https://doi.org/10.1007/978-3-642-22110-1_14

Michael Colón, Sriram Sankaranarayanan, and Henny Sipma. 2003. 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 (Lecture Notes in Computer Science, Vol. 2725)*, Warren A. Hunt Jr. and Fabio Somenzi (Eds.). Springer, 420–432.

Patrick Cousot and Radhia Cousot. 1977. 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 (POPL)*. ACM. https://doi.org/10.1145/512950.512973

Patrick Cousot, Radhia Cousot, Jérôme Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. 2005. The ASTREÉ Analyzer. In *Programming Languages and Systems, 14th European Symposium on Programming (ESOP) (Lecture Notes in Computer Science, Vol. 3444)*. Springer. https://doi.org/10.1007/978-3-540-31987-0_3

Patrick Cousot and Nicolas Halbwachs. 1978. 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*, Alfred V. Aho, Stephen N. Zilles, and Thomas G. Szymanski (Eds.). ACM Press, 84–96.

Leonardo Mendonça de Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference (TACAS) (Lecture Notes in Computer Science, Vol. 4963)*. Springer. https://doi.org/10.1007/978-3-540-78800-3_24

Isil Dillig, Thomas Dillig, and Alex Aiken. 2012. Automated error diagnosis using abductive inference. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '12, Beijing, China - June 11 - 16, 2012*, Jan Vitek, Haibo Lin, and Frank Tip (Eds.). ACM, 181–192.

Isil Dillig, Thomas Dillig, Boyang Li, and Kenneth L. McMillan. 2013. Inductive Invariant Generation via Abductive Inference. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications OOPSLA)*. ACM. https://doi.org/10.1145/2509136.2509511

Joe W. Duran and Simeon C. Ntafos. 1981. A Report on Random Testing. In *Proceedings of the 5th International Conference on Software Engineering, San Diego, California, USA, March 9-12, 1981*, Seymour Jeffrey and Leon G. Stucki (Eds.). IEEE Computer Society, 179–183.

Michael D. Ernst, Adam Czeisler, William G. Griswold, and David Notkin. 2000. Quickly detecting relevant program invariants. In *Proceedings of the 22nd International Conference on on Software Engineering, ICSE 2000, Limerick Ireland, June 4-11, 2000*, Carlo Ghezzi, Mehdi Jazayeri, and Alexander L. Wolf (Eds.). ACM, 449–458.

Robert W. Floyd. 1967. Assigning Meanings to Programms. In *Proccedings of the AMS Symposium on Appllied Mathematics*, Vol. 19. American Mathematical Society. http://www.cs.virginia.edu/~weimer/2007-615/reading/FloydMeaning.pdf

Pranav Garg, Christof Löding, P. Madhusudan, and Daniel Neider. 2014. 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 (Lecture Notes in Computer Science, Vol. 8559)*, Armin Biere and Roderick Bloem (Eds.). Springer, 69–87.

Pranav Garg, Daniel Neider, P. Madhusudan, and Dan Roth. 2016. Learning invariants using Decision Trees and Implication Counterexamples. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming*

Language (POPL). ACM. https://doi.org/10.1145/2837614.2837664

Patrice Godefroid, Aditya V. Nori, Sriram K. Rajamani, and SaiDeep Tetali. 2010. Compositional may-must program analysis: unleashing the power of alternation. In *Proceedings of the 37th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2010, Madrid, Spain, January 17-23, 2010*, Manuel V. Hermenegildo and Jens Palsberg (Eds.). ACM, 43–56.

Bhargav S. Gulavani, Thomas A. Henzinger, Yamini Kannan, Aditya V. Nori, and Sriram K. Rajamani. 2006. SYNERGY: a new algorithm for property checking. In *Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2006, Portland, Oregon, USA, November 5-11, 2006*, Michal Young and Premkumar T. Devanbu (Eds.). ACM, 117–127.

Sumit Gulwani and Nebojsa Jojic. 2007. 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*, Martin Hofmann and Matthias Felleisen (Eds.). ACM, 277–289.

Sumit Gulwani, Saurabh Srivastava, and Ramarathnam Venkatesan. 2008. 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*, Rajiv Gupta and Saman P. Amarasinghe (Eds.). ACM, 281–292.

Ashutosh Gupta, Rupak Majumdar, and Andrey Rybalchenko. 2013. From tests to proofs. *Int. J. Softw. Tools Technol. Transf.* 15, 4 (2013), 291–303.

Steve Hanneke. 2009. *Theoretical foundations of active learning.* Technical Report. CARNEGIE-MELLON UNIV PITTSBURGH PA MACHINE LEARNING DEPT.

Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Kenneth L. McMillan. 2004. Abstractions from proofs. In *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2004, Venice, Italy, January 14-16, 2004*, Neil D. Jones and Xavier Leroy (Eds.). ACM, 232–244.

C. A. R. Hoare. 1969. An Axiomatic Basis for Computer Programming. *Commun. ACM* 12, 10 (1969). https://doi.org/10.1145/363235.363259

Susan Horwitz, Thomas W. Reps, and David W. Binkley. 1988. Interprocedural Slicing Using Dependence Graphs. In *Proceedings of the ACM SIGPLAN'88 Conference on Programming Language Design and Implementation (PLDI), Atlanta, Georgia, USA, June 22-24, 1988*, Richard L. Wexelblat (Ed.). ACM, 35–46.

Kangjing Huang, Xiaokang Qiu, Peiyuan Shen, and Yanjun Wang. 2020. Reconciling Enumerative and Deductive Program Synthesis. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation* (London, UK) *(PLDI 2020)*. Association for Computing Machinery, New York, NY, USA, 1159–1174. https://doi.org/10.1145/3385412.3386027

Ranjit Jhala and Kenneth L. McMillan. 2006. A Practical and Complete Approach to Predicate Refinement. In *Tools and Algorithms for the Construction and Analysis of Systems, 12th International Conference (TACAS) (Lecture Notes in Computer Science, Vol. 3920)*. Springer. https://doi.org/10.1007/11691372_33

Jiaying Li, Jun Sun, Li Li, Quang Loc Le, and Shang-Wei Lin. 2017. Automatic loop-invariant generation and refinement through selective sampling. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering, ASE 2017, Urbana, IL, USA, October 30 - November 03, 2017*, Grigore Rosu, Massimiliano Di Penta, and Tien N. Nguyen (Eds.). IEEE Computer Society, 782–792.

Antoine Miné. 2006. The octagon abstract domain. *High. Order Symb. Comput.* 19, 1 (2006), 31–100.

Tom M Mitchell et al. 1997. Machine learning. 1997. *Burr Ridge, IL: McGraw Hill* 45, 37 (1997), 870–877.

ThanhVu Nguyen, Timos Antonopoulos, Andrew Ruef, and Michael Hicks. 2017. Counterexample-Guided Approach to Finding Numerical Invariants. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering (ESEC/FSE)*. ACM. https://doi.org/10.1145/3106237.3106281

ThanhVu Nguyen, Deepak Kapur, Westley Weimer, and Stephanie Forrest. 2012. Using dynamic analysis to discover polynomial and array invariants. In *34th International Conference on Software Engineering, ICSE 2012, June 2-9, 2012, Zurich, Switzerland*, Martin Glinz, Gail C. Murphy, and Mauro Pezzè (Eds.). IEEE Computer Society, 683–693.

Saswat Padhi, Todd Millstein, Aditya Nori, and Rahul Sharma. 2019. Overfitting in Synthesis: Theory and Practice. In *Computer Aided Verification - 30th International Conference (CAV) (Lecture Notes in Computer Science)*. Springer (To Appear). https://arxiv.org/pdf/1905.07457

Saswat Padhi, Rahul Sharma, and Todd D. Millstein. 2016. Data-Driven Precondition Inference with Learned Features. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. ACM. https://doi.org/10.1145/2908080.2908099

Mukund Raghothaman, Andrew Reynolds, and Abhishek Udupa. 2019. The SyGuS Language Standard Version 2.0. (2019).

Andrew Reynolds, Morgan Deters, Viktor Kuncak, Cesare Tinelli, and Clark W. Barrett. 2015. Counterexample-Guided Quantifier Instantiation for Synthesis in SMT. In *Computer Aided Verification - 27th International Conference (CAV) (Lecture Notes in Computer Science, Vol. 9207)*. Springer. https://doi.org/10.1007/978-3-319-21668-3_12

Gabriel Ryan, Justin Wong, Jianan Yao, Ronghui Gu, and Suman Jana. 2020. CLN2INV: Learning Loop Invariants with Continuous Logic Networks. In *8th International Conference on Learning Representations, ICLR 2020, Addis Ababa, Ethiopia, April 26-30, 2020*. OpenReview.net.

Sriram Sankaranarayanan, Franjo Ivancic, Ilya Shlyakhter, and Aarti Gupta. 2006. Static Analysis in Disjunctive Numerical Domains. In *Static Analysis, 13th International Symposium, SAS 2006, Seoul, Korea, August 29-31, 2006, Proceedings (Lecture Notes in Computer Science, Vol. 4134)*, Kwangkeun Yi (Ed.). Springer, 3–17.

Greg Schohn and David Cohn. 2000. Less is More: Active Learning with Support Vector Machines. In *Proceedings of the Seventeenth International Conference on Machine Learning (ICML 2000), Stanford University, Stanford, CA, USA, June 29 - July 2, 2000*, Pat Langley (Ed.). Morgan Kaufmann, 839–846.

Rahul Sharma and Alex Aiken. 2016. From Invariant Checking to Invariant Inference Using Randomized Search. *Formal Methods in System Design* 48, 3 (2016). https://doi.org/10.1007/s10703-016-0248-5

Rahul Sharma, Saurabh Gupta, Bharath Hariharan, Alex Aiken, Percy Liang, and Aditya V. Nori. 2013b. A Data Driven Approach for Algebraic Loop Invariants. 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 (Lecture Notes in Computer Science, Vol. 7792)*, Matthias Felleisen and Philippa Gardner (Eds.). Springer, 574–592.

Rahul Sharma, Saurabh Gupta, Bharath Hariharan, Alex Aiken, and Aditya V. Nori. 2013a. Verification as Learning Geometric Concepts. In *Static Analysis - 20th International Symposium (SAS) (Lecture Notes in Computer Science, Vol. 7935)*. Springer. https://doi.org/10.1007/978-3-642-38856-9_21

Rahul Sharma, Aditya V. Nori, and Alex Aiken. 2012. Interpolants as Classifiers. In *Computer Aided Verification - 24th International Conference, CAV 2012, Berkeley, CA, USA, July 7-13, 2012 Proceedings (Lecture Notes in Computer Science, Vol. 7358)*, P. Madhusudan and Sanjit A. Seshia (Eds.). Springer, 71–87.

Xujie Si, Hanjun Dai, Mukund Raghothaman, Mayur Naik, and Le Song. 2018. Learning Loop Invariants for Program Verification. In *Advances in Neural Information Processing Systems 31: Annual Conference on Neural Information Processing Systems (NeurIPS)*. http://papers.nips.cc/paper/8001-learning-loop-invariants-for-program-verification

Aditya V. Thakur, Akash Lal, Junghee Lim, and Thomas W. Reps. 2015. PostHat and All That: Automating Abstract Interpretation. *Electron. Notes Theor. Comput. Sci.* 311 (2015), 15–32.

Jianan Yao, Gabriel Ryan, Justin Wong, Suman Jana, and Ronghui Gu. 2020. Learning nonlinear loop invariants with gated continuous logic networks. In *Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation, PLDI 2020, London, UK, June 15-20, 2020*, Alastair F. Donaldson and Emina Torlak (Eds.). ACM, 106–120.

He Zhu, Stephen Magill, and Suresh Jagannathan. 2018. A Data-Driven CHC Solver. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. ACM. https://doi.org/10.1145/3192366.3192416