OASIS: ILP-Guided Synthesis of Loop Invariants

Sahil Bhatia
Microsoft Research
Bengaluru, India
t-sab@microsoft.com

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

Nagarajan Natarajan
Microsoft Research
Bengaluru, India
nagarajn@microsoft.com

Rahul Sharma
Microsoft Research
Bengaluru, India
rahsha@microsoft.com

Prateek Jain
Microsoft Research
Bengaluru, India
prajain@microsoft.com

Abstract

Finding appropriate inductive *loop invariants* for a program is a key challenge in verifying its functional properties. Although the problem is undecidable in general, several heuristics have been proposed to handle practical programs that tend to have simple control-flow structures. However, these heuristics only work well when the space of invariants is small. On the other hand, machine-learned techniques that use continuous optimization have a high sample complexity, i.e., the number of invariant guesses and the associated counterexamples, since the invariant is required to *exactly* satisfy a specification. We propose a novel technique that is able to solve complex verification problems involving programs with larger number of variables and non-linear specifications. We formulate an invariant as a piecewise low-degree polynomial, and reduce the problem of synthesizing it to a set of integer linear programming (ILP) problems. This enables the use of state-of-the-art ILP techniques that combine enumerative search with continuous optimization; thus ensuring fast convergence for a large class of verification tasks while still ensuring low sample complexity. We instantiate our technique as the open-source OASIS tool using an off-the-shelf ILP solver, and evaluate it on more than 300 benchmark tasks collected from the annual SyGuS competition and recent prior work. Our experiments show that OASIS outperforms the state-of-the-art tools, including the winner of last year's SyGuS competition, and is able to solve 9 challenging tasks that existing tools fail on.

1 Introduction

Program verification aims to provide a strong guarantee on the correctness of an implementation by formally proving that it meets a desired property, such as termination, correctness of assertions, memory safety, and more. Given our ever-increasing dependence on complex software systems, this seems like a critical but elusive goal. The key challenge in program verification is in handling unbounded control-flow patterns, such as loops and recursive functions. A correctness proof for such patterns uses mathematical induction, and the inductive hypotheses are called *inductive invariants*, and more specifically *loop invariants* for programs with loops. Many verified-programming environments [10, 16, 29] require users to furnish the right invariants. However, it can be quite challenging, even for expert programmers, to annotate these invariants for simple practical cases (see Figure 1). In this paper, we focus on automatically synthesizing invariants that are sufficient to prove correctness.

Fully automatic synthesis of loop invariants is undecidable in general. However, the formal methods community has developed several heuristics for programs with tractable control-flow structures. One such important class is that of integer-manipulating programs which frequently arise in robotics, scientific computing and security-critical engineering applications [22]. Indeed, a major chunk of benchmarks from the annual International Competition on Software Verification (SV-Comp) [3] and the annual Syntax-Guided Synthesis Competition (SyGuS-Comp) [4] fall into this class.

This class of verification problems has been studied extensively, and several approaches have been proposed for automatically inferring loop invariants for these problems. Traditionally successful approaches for this problem are based on enumerating all possible expressions for the invariant. Broadly, the system searches for an invariant and in case of incorrect guess, is provided a proof for incorrectness. The system then uses this proof as a guiding example to refine the next candidate invariant. Although this simple heuristic works well on small problems, enumerative search fails to scale as the space of invariants grows. For example, the LOOPINVGEN [23, 24] tool, which has won the 2017 and 2018 SyGuS-Comp [6, 7] fails to handle most programs with non-linear integer arithmetic since introducing non-linearity significantly increases the space of possible expressions.

Recent prior work [28, 30] propose using machine learning (ML) and continuous optimization techniques to solve the above-mentioned search problem. However, these techniques also do not scale to challenging problems. A key issue is that while continuous optimization is highly efficient for solving a problem approximately, invariant synthesis demands finding an *exact* solution. Current methods attempt to solve the problem by "rounding" the solution, which, due to the combinatorial nature of the problem, can lead to arbitrary incorrect solutions. Furthermore, incorporating a "proof" of incorrectness is akin to adding a single constraint to the optimization problem, and it often results in an approximate solution that leads up to the same rounded solution. Hence, in general, sample complexity of these techniques can also be fairly large for reasonably hard problems.

In this paper, we propose a novel approach called OASIS,[1] that closely integrates the enumerative search and combinatorial optimization methods. In particular, we parameterize the invariant as a piecewise low-degree polynomials and iteratively search for invariants that satisfy the proofs of incorrectness generated for the previous guesses. We show that the problem of guessing such piecewise low-degree polynomials can be reduced to that of an integer linear program (ILP). Furthermore, our formulation allows for incorporating constraints that ensure small and natural expressions that have been shown to generalize better [23]. These ILPs can be discharged to standard solvers that ensure *optimal* solutions. This enables our technique to efficiently synthesize invariants for many challenging verification tasks with a reasonably small number of samples and iterations.

We evaluate OASIS on more than 300 verification tasks from recent prior work [24, 28]. We show that our tool consistently outperforms the state-of-the-art enumerative [23, 24] and ML techniques [28] in terms of the number of tasks solved. Our experiments indicate that our technique is efficient both in terms of time and sample complexity. OASIS solves most tasks within 300 s, unlike the recent ML-based method by Si et al. [28] that requires up to one hour per benchmark.

2 Background

In this section, we formally define the problem of verifying correctness of programs, then explain the key role of loop invariants, and conclude with a brief discussion of the prior work.

2.1 Program Verification and Loop Invariants

The first step in program verification is defining a *specification* for the desired property. Typically [3, 4] this is provided as a pair of logical formulas — **(1)** a *precondition* that constrains the initial state of the program, and **(2)** a *postcondition* that validates the final state after execution of the program. Throughout this paper, by a *program state* (or simply *state*) we denote the specific values of the program variables at some point during execution. Many programming languages support the **assume** and **assert** keywords, for stating the pre- and post-conditions respectively. For example, Figure 1 shows a program that computes the sum

```
assume (n > 0 ∧ a = 1
          ∧ t = 1 ∧ s = 1)
while (s ≤ n) do
    t ← t + 2 ;   s ← s + t
    a ← a + 1
assert (s = a^2)
```

Figure 1: A non-linear verification task proposed by Garg et al. [18]

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

of the first a positive odd numbers, and a specification that encodes the well-known result that this sum should be a^2. Given such a specification, we define the verification problem as:

Definition 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 decide whether *every* terminating execution of **P** starting from *any* state that satisfies ρ halts at a state that satisfies ϕ.

In the Floyd-Hoare logic (FHL) [17, 20], this problem is abbreviated to the formula $\{\rho\}$ **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, $\{x \geq 0\}$ `y ← 1.0 / x` $\{y \geq 0\}$ is not since $1/0$ is undefined. The FHL offers initial theoretical underpinnings for automatic verification by providing a set of inference rules that can be recursively used on the program structure.

However, the FHL inference rules are sufficient 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 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:

$\mathrm{VC}_{\mathsf{pre}}$: $\rho \implies \mathcal{I}$, i.e., $\mathcal{I}$ must hold immediately before the loop

$\mathrm{VC}_{\mathsf{ind}}$: $\{G \wedge \mathcal{I}\}\ S\ \{\mathcal{I}\}$, i.e., $\mathcal{I}$ must be inductive (hold after each iteration)

$\mathrm{VC}_{\mathsf{pos}}$: $\neg G \wedge \mathcal{I} \implies \phi$, i.e., $\mathcal{I}$ must certify the postcondition upon exiting the loop

These three properties are called the *verification conditions* (VCs) for the loop. Any predicate $\mathcal{I}$ 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).

Thanks to efficient theorem provers [9, 14], today it is possible to automatically check if a given predicate is indeed a sufficient loop invariant. However, automatically finding such a loop invariant for arbitrary loops is undecidable in general, and even simple practical instances are challenging for state-of-the-art tools. For example, to prove the correctness of our motivating example shown in Figure 1, the loop invariant $\mathcal{I} \equiv (s = a^2) \wedge (t = 2a + 1)$ is sufficient. However, our experiments indicate that all state-of-the-art tools fail to solve this verification task within reasonable time limit.

2.2 Loop Invariant Inference: The State of The Art

Static Analyses. The traditional symbolic approaches for inferring loop invariants include abstract interpretation [12, 13], predicate abstraction [8], interpolation [21], constraint solving [11, 19], and abductive inference [15]. These techniques, however, restrict the theories to which they may be applied, or the shapes of invariants that they may infer.

Data-Driven Synthesis. Complementing the symbolic analyses are the data-driven "guess-and-check" approaches [18, 23, 24, 26, 27]. These techniques heuristically search for an expression consistent with observed program traces and counterexamples. Although these heuristics work well for verification tasks where the space of invariants is small, they do not scale to large spaces.

Machine Learning. Recently, ML techniques that rely on continuous optimization instead of heuristics have been proposed for scalability. Zhu et al. [30] use support-vector machines (SVMs) for learning expressions that partially satisfy the VCs, and then combine them using decision trees. Si et al. [28] propose a neural network (NN) that predicts an invariant using only the counterexamples. These techniques however have large sample complexity since an invariant must *exactly* satisfy all VCs, unlike typical scenarios for SVMs and NNs that require only an approximate solution.

Our technique OASIS combines the best of both worlds – search and optimization. The closest prior work to ours is that by Sharma and Aiken [26] which is based on Metropolis-Hastings. They formulate learning the entire loop invariant as a single optimization problem, which is in general too irregular to be amenable to exact optimization techniques. We instead build on a multi-stage learning technique proposed by Padhi et al. [23] that allows us to formulate efficiently solvable exact optimization subproblems and combine them to the final invariant. We detail this in Section 3.2.

3 Overview of Our Approach

In this section we first describe a reduction of the problem of inferring sufficient loop invariants to a classification problem, the initial inspiration for which comes from prior work by Sharma et al. [27]. We then briefly overview our classification technique, and finally show how we can incrementally improve the classifier using counterexamples generated for each VC.

3.1 Invariant Inference as Binary Classification

A sufficient loop invariant can be viewed as an *exact separator* — it should demonstrate that the set of possible states at the entry to a loop (called *loop-head states*) are disjoint from those that violate the postcondition; therefore establishing that the postcondition would be satisfied for all executions.

Consider verifying our motivating example from Figure 1. We visualize the classification problem in Figure 2. VC_{pre} and VC_{ind} from Definition 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., $(n = 10, a = t = s = 1)$ appearing before the first iteration, and the subsequent states after each iteration (indicated by the arrows), e.g., $(n = 10, a = 2, t = 3, s = 4)$, $(n = 10, a = 3, t = 5, s = 9)$ etc. The $\neg G \wedge \neg\phi$ space (red rectangle) denotes the states violating the postcondition, e.g., $(n = 10, a = 4, t = 1, s = 4)$. VC_{pos} forces $\mathcal{I}$ to be disjoint with this space. Such an invariant $\mathcal{I}$ guarantees that no execution would terminate at a state that violates the desired postcondition ϕ.

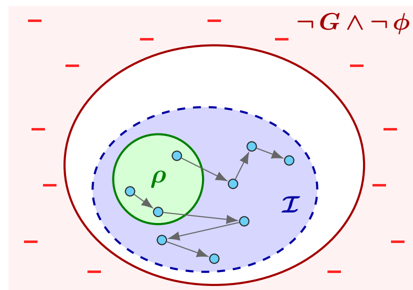

Figure 2: A sufficient loop invariant can be viewed as a classifier for states.

Note that efficiently finding the set of all possible loop-head states is undecidable in general. Indeed, if the set of all loop-head states is known, e.g., for loops with a bounded number of iterations, then this set itself describes a sufficient loop invariant and verification is always *decidable*.

3.2 Counterexample-Guided Refinement of Classifiers

We now describe our core framework for building a classifier and gradually refining it with counterexamples. Our overall approach in OASIS is largely influenced by LOOPINVGEN [23], a state-of-the-art tool which won the invariant-inference (Inv) track of 2017 and 2018 SyGuS-Comp [6, 7]. Instead of learning the entire loop invariant in a single shot, LOOPINVGEN introduced a multi-stage learning technique that composes the final invariant out of several predicates learned over smaller subproblems.

Figure 3 outlines this framework. The main INFER procedure is invoked with a specification-annotated loop $\mathcal{L} \equiv \{\rho\}$ **while** G **do** S $\{\phi\}$, and a set $\mathcal{P}$ of program states sampled randomly by running the loop for a few iterations. The framework begins with the weakest possible invariant $\mathcal{I} \equiv (\neg G \Rightarrow \phi)$, and iteratively strengthens it (line 17) for inductiveness. Note that this guarantees satisfaction of VC_{pos}. 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 state that $\mathcal{I}$ does not capture), a violation of VC_{ind} adds a *negative* example (a state that $\mathcal{I}$ captures but is not inductive on). The loop in lines 10 – 14 indicates the key learning subcomponents.

Essentially, in line 13 we first learn a set $\mathcal{F}$ of predicates over subproblems by invoking LEARN, and then use a standard Boolean-function learner [23] BOOLCOMBINE in line 14 to learn a Boolean combination of these predicates and update the candidate

```
1  func INFER (ℒ ≡ {ρ} while G do S {φ}, 𝒫)
2    if ρ ⇏ (¬G ⇒ φ) then return False
3    ℐ ← (¬G ⇒ φ)
4    while True do
5      c ← CHECK(ρ ⇒ ℐ)
6      if c ≠ ⊥ then return INFER(ℒ, 𝒫 ∪ {c})
7      𝒩 ← {}
8      while True do
9        ℱ ← {}
10       while True do
11         (σ₊, σ₋) ← CONFLICT(𝒫, 𝒩, ℱ)
12         if σ₊ = σ₋ = {} then break
13         else ℱ ← ℱ ∪ LEARN(σ₊, σ₋)
14       δ ← BOOLCOMBINE(ℱ)
15       c ← CHECK(δ ⇒ {G ∧ ℐ} S {ℐ})
16       if c ≠ ⊥ then 𝒩 ← 𝒩 ∪ {c}
17     ℐ ← (ℐ ∧ δ)
18     if δ = True then return ℐ
```

Figure 3: A framework for loop-invariant inference using automatic feature learning [23]

invariant. The predicates in $\mathcal{F}$ are learned in a demand-driven manner. In line 11, the CONFLICT procedure selects two sets $(\sigma_+, \sigma_-) \subseteq \mathcal{P} \times \mathcal{N}$ that are *conflicting*, i.e., these positive and negative examples are indistinguishable modulo $\mathcal{F}$. The LEARN procedure then generates a distinguishing predicate for (σ_+, σ_-). For more details on this framework, we refer to the LOOPINVGEN paper [23].

The primary difference between OASIS and LOOPINVGEN is the LEARN procedure — while LOOPINVGEN uses exhaustive enumeration, which does not scale to large search spaces, OASIS uses integer-linear programming as we discuss in the next section.

4 The ILP Formulation

In this section, we formulate the problem of generating candidate predicates for synthesizing invariants, given a set of labeled program states, i.e., the LEARN call in line 13 of Figure 3. Let $\mathcal{V}$ denote the set of program variables, and $p(\sigma)$ denote a polynomial of degree at most r defined over the program states $\sigma \in \Sigma \subseteq \mathbb{Z}^{|\mathcal{V}|}$. We model the problem of inferring loop invariants $\mathcal{H} : \mathbb{Z}^{|\mathcal{V}|} \to \{\mathsf{True}, \mathsf{False}\}$ as a search problem over piecewise thresholded polynomials:

$$\mathcal{H} = \left\{ \bar{p}(\sigma) = \mathbb{1}\left\{\sigma \in \Sigma_i\right\} \prod_j \mathbb{1}\left\{p_{ij}(\sigma) > 0\right\}, \bigcup_i \Sigma_i = \Sigma \right\}.$$

Note that $p(\sigma) = \langle \mathbf{w}, \Phi(\sigma) \rangle + b$, where $\Phi(\sigma)$ is the vector with coordinates corresponding to the terms in the multinomial expansion of $\langle \mathbf{1}, \sigma \rangle^r$. The membership predicates (i.e., $\mathbb{1}\left\{\sigma \in \Sigma_i\right\}$) themselves can be written as polyhedral constraints, and therefore can be subsumed in the product term. Rewriting the resulting logical formulae in conjunctive normal form, with C denoting the number of conjuncts and D the number of disjuncts in each conjunct, we have:

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

Given a set of program states with corresponding labels, our task is to find a candidate invariant/predicate $\in \mathcal{H}_{\text{CNF}}$ such that **(1)** it accurately classifies the given program states, and **(2)** 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. 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. Integer-manipulation programs imply polynomial coefficients $\mathbf{w}$ and b should be integers as well. Furthermore, the polynomial coefficients are governed by program variables themselves, which are often bounded. 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.

Consider the search problem **(1)** above: formally, we want to find a predicate $\bar{p} \in \mathcal{H}_{\text{CNF}}$ 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 $\bar{p}$ as a tree of depth 3: the program variables form the input layer to thresholded polynomials, 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 first layer. The reduction of the search problem to ILP is given as follows. Let $|\Phi|$ denote the number of features, i.e., length of $\Phi(\sigma)$. Bold letters denote vectors (e.g. $\mathbf{1}$ is a vector of all ones).

(**Input layer: Thresholded polynomials**) Write $z_{ncd} = \mathbb{1}\left\{\langle \mathbf{w}_{cd}, \Phi(\sigma_n) \rangle + b_{cd} > 0\right\} \in \{0, 1\}$. This is captured by the following constraints, for a sufficiently large integer M:

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

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

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

(**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:

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

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), (3) and (4) combined.* This ILP satisfies the following properties, the proofs of which are straightforward and can be found in the appendix.

Claim 1. *Any feasible solution to the above ILP problem is a member of* $\mathcal{H}_{\text{CNF}}$.

Claim 2. *Given a set of labeled program states* $\{\sigma_n, y_n\}, n \in [N]$*, if there exists a* $\bar{p} \in \mathcal{H}_{\text{CNF}}$ *s.t.* $\bar{p}(\sigma_n) = y_n$ *for all* $n \in [N]$*, then the ILP problem above has at least one 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 have been shown to generalize better [23]. Simplicity in our case can be characterized by the size of the predicate clauses and the magnitude of the polynomials. One way to achieve this is by constraining the L_1-norm of the coefficients $\mathbf{w}$. Note that L_1-norm can be expressed using linear constraints:[1] $\|\mathbf{w}\|_1 = \langle \mathbf{1}, \mathbf{w}^+ + \mathbf{w}^- \rangle$, where $\mathbf{w}^+ \geq 0$ and $\mathbf{w}^- \geq 0$ (componentwise inequality) such that $\mathbf{w} = \mathbf{w}^+ - \mathbf{w}^-$.

However, focussing only on the magnitude may lead to poor solutions. For example, consider the Hoare triple: $\{i = 0 \wedge s = 0\}$ **while** `(i` $\neq$ `n)` **do** `{s` $\leftarrow$ `s+i; i++;}` $\{s = n(n+1)/2\}$. Here, it is easy to verify that the loop invariant $s = i(i+1)/2$ is sufficient to assert VC_{pos}. The equivalent predicate in $\mathcal{H}_{\text{CNF}}$, $2s - i^2 - i \geq 0 \wedge i^2 + i - 2s \geq 0$, 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 [23, 24] employ heuristics such as preferring equality to inequality. We handle this by explicitly penalizing the *inclusion* of features in the solution. Our final objective function combines both the penalties:

$$\begin{aligned} &\min_{\mathbf{w}, \mathbf{w}^+, \mathbf{w}^-, b, z, y^{\vee}, \mu} \sum_{c \in [C], d \in [D]} \langle \mathbf{1}, \mathbf{w}^+_{cd} + \mathbf{w}^-_{cd} \rangle \;+\; \lambda \langle \mathbf{1}, \mu \rangle \\ &\quad \text{subject to Equations (2), (3) and (4), and} \\ &\mathbf{1} - M(\mathbf{1} - \mu) \;\leq\; \sum_{c \in [C], d \in [D]} \mathbf{w}^+_{cd} + \mathbf{w}^-_{cd} \;\leq\; M\mu, \\ &\qquad \forall c \in [C], d \in [D], \quad \mathbf{w}_{cd} \;=\; \mathbf{w}^+_{cd} - \mathbf{w}^-_{cd}, \\ &\qquad \mathbf{w}^+_{cd} \geq 0\,, \;\; \mathbf{w}^-_{cd} \geq 0\,, \;\; \mu \;\in\; \{0,1\}^{|\Phi|}. \end{aligned} \tag{5}$$

The key advantages of the above ILP formulation are **(1)** we can get the optimal predicate, **(2)** we can leverage continuous and integer optimization techniques to solve the problem. This enables efficient, scalable search compared to existing enumerative techniques.

Remark 1. 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 following section, we detail the parameters used in our evaluation.

5 Experimental Evaluation

We have implemented OASIS using the LOOPINVGEN [23] framework in OCaml, and using Z3 [14] as the theorem prover for checking validity of the verification conditions. We implemented our logic for reducing the classification problem to ILPs in a Python script, which discharges the ILP subproblems to the OR-Tools [2] optimization package from Google. We evaluate OASIS on commodity hardware — CPU-only machines with up to 32 GB RAM running Ubuntu Linux 18.04.

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

Tool	Total Solved (out of 133)	Unique Solved (out of 92 unique)
CODE2INV	106 *	—
CVC4	104	70
LOOPINVGEN	97	65
LOOPINVGEN+HE	109	74
OASIS	**121**	**81**

Table 1: Results on 133 instances studied by Si et al. [28] requiring reasoning over linear arithmetic. We allow a timeout of 300 s for each instance with each tool. * See Remark 2.

Solvers and Benchmarks. We compare our tool, OASIS, against four state-of-the-art techniques: **(1)** CODE2INV [28] which uses machine learning, **(2)** LOOPINVGEN [23] which uses enumerative search, **(3)** LOOPINVGEN+HE [24] which is a recent extension to LOOPINVGEN for efficiently enumerating large spaces, and **(4)** CVC4 [9, 25] which uses a refutation-based approach. LOOPINVGEN and CVC4 are respectively the winner and the runner-up of the invariant-synthesis (Inv) track of both SyGuS-Comp'17 [6] and SyGuS-Comp'18 [7].

We evaluate our OASIS tool on a comprehensive suite of benchmarks used in recent studies [24, 28], which subsumes those curated by the Syntax-Guided Synthesis Competition (SyGuS-Comp) [4]. All our benchmarks are encoded in the extended SyGuS input format [5], and are publicly available.

Exclusions. Recently, Zhu et al. [30] have proposed another approach based on solving constrained Horn clauses (CHCs), but their implementation only handles C/C++ benchmarks. We exclude a comparison with their tool since a SyGuS $\mapsto$ C translation is not uniquely defined. Moreover, most benchmarks used in their evaluation are subsumed by the SyGuS-Comp'18 benchmarks [7]. We also exclude a performance comparison with the CODE2INV [28] tool since it is orders of magnitude slower than all other tools we evaluate and compare with.

Loop Invariant Inference. We evaluate the tools on 133 instances published by Si et al. [28], and report the results in Table 1. OASIS synthesizes the sufficient loop invariants on 12 more instances than the second best tool (also see Remark 2). On examining these instances, we found that there are 41 duplicates. We therefore also present a comparison on the 92 unique instances in the third column. We find a similar trend in the performance of the tools in this column as well.

These 133 instances have relatively simple, (piecewise) linear invariants, i.e., they only involve polynomials of degree $r = 1$. Moreover, it suffices to use only two disjuncts, i.e., $C = 1$ and $D = 2$ in Equation (5), and constrain the polynomial coefficients to integers within $[-6, 6]$ for OASIS.[2] We set $M = 100,000$ in Equation (5) for linear benchmarks. The median number of variables in the 92 unique instances is 10 (max is 22). As we discuss next, this sharply contrasts with linear instances of Table 2(a), where the median is 3 (max is 9). This indicates that OASIS scales better with an increasing number of variables, as compared to state-of-the-art tools.

Remark 2. This is the number Si et al. [28] report, but their results are not comparable since the reported experiments had discrepancies. For comparing against existing solvers, the authors translated the C/C++ instances, which their tool consumes, to the SyGuS input format [5]. However, some of these translations were incorrect. We use the updated translations [1] for our evaluation.

Next, we evaluate the tools on all the (piecewise) linear instances used in evaluating the recent LOOPINVGEN+HE tool [24]. Results for these additional 162 instances (not including the ones already covered in the above 133) are presented in Table 2(a). We use the same hyperparameter choices for OASIS as in Table 1, but constrain the polynomial coefficients to integers within $[-5000, 5000]$ since several of these instances contain large constants. We observe that while the existing tools are competitive on this set, OASIS performs slightly better.

Finally, we evaluate 18 challenging instances curated by Padhi et al. [24], that involve (piecewise) non-linear invariants involving polynomials of degree up to 3. We set $M = 1,000,000$ in Equation (5),[3] and constrain the polynomial coefficients to integers within $[-10, 10]$ for non-linear benchmarks. For evaluating OASIS, we start with degree $r = 1$, and increase r by 1 after every 300 s timeout. The

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

[3] Theoretically M can be set to any sufficiently large integer and need not be tuned.

Tool	Solved
CVC4	135
LoopInvGen	142
LoopInvGen+HE	144
Oasis	**145**

(a) Results on 162 linear instances

Tool	Solved
CVC4	6
LoopInvGen	7
LoopInvGen+HE	8
Oasis	**10**

(b) Results on 18 non-linear instances

Table 2: Results on 180 instances studied by Padhi et al. [24] requiring reasoning over linear and non-linear arithmetic. We allow a total timeout of 300 s for linear, and 600 s for non-linear instances.

Benchmark	Statistic	Iteration Count	Running Time (s)
45 out of 92 linear instances [28]	Min	1	1.4
	Median	2	5.1
	Max	164	135.3
79 out of 162 linear instances [24]	Min	1	1.1
	Median	3	2.8
	Max	236	225.4
8 out of 18 non-linear instances [24]	Min	1	2.6
	Median	1	9.7
	Max	9	41.5

Table 3: Number of iterations and the wall-clock time required by Oasis to solve the benchmarks. The statistics are computed on the instances that required at least one invocation of the ILP solver, i.e., at least one Learn call from the Infer algorithm in Figure 3.

results on this benchmark with a total timeout of 600 s is presented in Table 2(b). Oasis outperforms state-of-the-art by a clear margin in these challenging instances.

We observe that one failed instance (named `s9.desugared.sl`) requires at least degree 3 polynomials — upon allowing a 900 s total timeout, Oasis is able to generate a sufficient invariant. We note that Oasis is also able to solve another failed instance, `sqrt-more-rows-swap-columns.sl`, with a 10 min timeout per r, whereas other tools still fail to solve this.

Impact of Subproblem Size. A key hyperparameter to Oasis is the number of examples N elicited to generate a candidate predicate (by solving Equation (5)) per iteration. For the linear benchmarks (Tables 1 and 2(a)) we use $N \in \{10, 50\}$, and for the non-linear ones (Table 2(b)) we use $N \in \{100, 1000\}$ and record the best setting per instance (we perform the same sweep for LoopInvGen+HE as well). When we fix the size N and evaluate Oasis across the 92 linear instances from Table 1 (third column), we find that the number of solved instances are **68** ($N = 10$), **69** ($N = 30$), **75** ($N = 50$), and **64** ($N = 100$) respectively. This indicates that the performance of the tool is not monotonic in N and suggests a trade-off in choosing N — in a deployment scenario, one may start with small N, fail fast and restart with larger N.

Time and Sample Complexity. Our sample complexity is proportional to the number of iterations required and time complexity is proportional to the wall-clock time required by Oasis to synthesize a sufficient invariant. Note that the total number of examples required is at most number of iterations times N (number of examples per round). The number of iterations and running time for the three benchmarks, corresponding to Tables 1, 2(a) and 2(b), are shown in Table 3.

Table 3 shows that out of all solved instances from Si et al. [28], half of the instances required at most 2×50 examples, and the remaining at most $164 \times 50 = 8200$. Thus, our sample complexity is orders of magnitude better than the ML technique proposed by Si et al. [28, Figure 5 (b)]. Furthermore, the absolute time needed to solve an instance was at most three minutes (not reported by Si et al. [28]). These observations are consistent over the other two benchmarks as well.

6 Conclusions

We present a novel synthesis approach for loop invariants, a key requirement for program verification, that combines the best of traditionally successful enumerative search with optimization techniques used in machine-learning. Our solution carefully captures the structure in the problem — we give a simple and efficient reduction of a complex search problem to a set of ILPs. Further exploiting the structure in the parameter space, we solve these ILPs optimally with off-the-shelf solvers on commodity hardware. Our tool, OASIS, outperforms competitive state-of-the-art solvers by a significant margin on several challenging verification tasks. We believe that this combination of search and optimization techniques would help scale verification tools to handle complex real-world tasks.

References

[1] Code2Inv: Source Code and Benchmarks. https://github.com/PL-ML/code2inv, Accessed: 2019-05-23.

[2] OR-Tools – Google Optimization Tools. https://github.com/google/or-tools, Accessed: 2019-05-23.

[3] The International Competition on Software Verification. https://sv-comp.sosy-lab.org, Accessed: 2019-05-23.

[4] The Syntax-Guided Synthesis Competition. https://sygus.org, Accessed: 2019-05-23.

[5] Rajeev Alur, Dana Fisman, Rishabh Singh, and Armando Solar-Lezama. Results and Analysis of SyGuS-Comp'15. In *Proceedings Fourth Workshop on Synthesis (SYNT)*, volume 202 of *EPTCS*, 2015. URL https://doi.org/10.4204/EPTCS.202.3.

[6] Rajeev Alur, Dana Fisman, Rishabh Singh, and Armando Solar-Lezama. SyGuS-Comp 2017: Results and Analysis. In *Proceedings Sixth Workshop on Synthesis (SYNT@CAV)*, volume 260 of *EPTCS*, 2017. URL https://doi.org/10.4204/EPTCS.260.9.

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

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

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

[10] Edwin Brady. Idris, a general-purpose dependently typed programming language: Design and implementation. *Journal of Functional Programming*, 23(5), 2013. URL https://doi.org/10.1017/S095679681300018X.

[11] Michael Colón, Sriram Sankaranarayanan, and Henny Sipma. Linear Invariant Generation Using Non-linear Constraint Solving. In *Computer Aided Verification, 15th International Conference (CAV)*, volume 2725 of *Lecture Notes in Computer Science*. Springer, 2003. URL https://doi.org/10.1007/978-3-540-45069-6_39.

[12] Patrick Cousot and Radhia Cousot. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Conference Record of the Fourth ACM Symposium on Principles of Programming Languages (POPL)*. ACM, 1977. URL https://doi.org/10.1145/512950.512973.

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

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

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

[16] Jean-Christophe Filliâtre and Andrei Paskevich. Why3 - Where Programs Meet Provers. In *Programming Languages and Systems - 22nd European Symposium on Programming (ESOP)*, volume 7792 of *Lecture Notes in Computer Science*. Springer, 2013. URL https://doi.org/10.1007/978-3-642-37036-6_8.

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

[18] Pranav Garg, Daniel Neider, P. Madhusudan, and Dan Roth. 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, 2016. URL https://doi.org/10.1145/2837614.2837664.

[19] Ashutosh Gupta, Rupak Majumdar, and Andrey Rybalchenko. From Tests to Proofs. *STTT*, 15(4), 2013. URL https://doi.org/10.1007/s10009-012-0267-5.

[20] C. A. R. Hoare. An Axiomatic Basis for Computer Programming. *Communications of the ACM*, 12(10), 1969. URL https://doi.org/10.1145/363235.363259.

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

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

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

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

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

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

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

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

[29] Nikhil Swamy, Juan Chen, Cédric Fournet, Pierre-Yves Strub, Karthikeyan Bhargavan, and Jean Yang. Secure Distributed Programming with Value-Dependent Types. *Journal of Functional Programming*, 23 (4), 2013. URL https://doi.org/10.1017/S0956796813000142.

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

Proofs of Claims

Claim 3. *Any feasible solution to the ILP problem in Equation (5) is a member of* $\mathcal{H}_{\text{CNF}}$.

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

$$y_n \;=\; \bigwedge_{c\in[C]} y^{\vee}_{nc} \;=\; \bigwedge_{c\in[C]} \bigvee_{d\in[D]} z_{ncd} \;=\; \bigwedge_{c\in[C]} \bigvee_{d\in[D]} \mathbb{1}\{\langle \mathbf{w}_{cd}, \Phi(\sigma_n)\rangle + b_{cd} > 0\} \;\in\; \mathcal{H}_{\text{CNF}}.$$

This completes the proof. □

Claim 4. *Given a set of labeled program states* $\{\sigma_n, y_n\}, n \in [N]$, *if there exists a* $\bar{p} \in \mathcal{H}_{\text{CNF}}$ *s.t.* $\bar{p}(\sigma_n) = y_n$ *for all* $n \in [N]$, *then the ILP problem above has at least one feasible solution.*

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

Verification Tasks Solved Only by OASIS

- **1 / 162 From LOOPINVGEN+HE [24] Linear Benchmarks:**
  1. `pldi18_seahorn_Fig4.sl`

- **5 / 133 From CODE2INV [28] Linear Benchmarks:**
  1. `1.c.sl`
  2. `2.c.sl`
  3. `16.c.sl`
  4. `46.c.sl`
  5. `94.c.sl`

- **3 / 18 From LOOPINVGEN+HE [24] Non-Linear Benchmarks:**
  1. `s10.desugared.sl`
  2. `s11.desugared.sl`
  3. `sqrt-more-rows-swap-columns.sl` (the motivating example from Figure 1)