Title
A theory of formal synthesis via inductive learning

Journal
Acta Informatica, 54(7)

ISSN
0001-5903

Authors
Jha, Susmit
Seshia, Sanjit A

Publication Date
2017-11-01

DOI
10.1007/s00236-017-0294-5

Peer reviewed

A Theory of Formal Synthesis via Inductive Learning

Susmit Jha
United Technologies Research Center, Berkeley
jhask@utrc.utc.com

Sanjit A. Seshia
EECS, UC Berkeley
sseshia@eecs.berkeley.edu

Formal synthesis is the process of generating a program satisfying a high-level formal specification. In recent times, effective formal synthesis methods have been proposed based on the use of inductive learning. We refer to this class of methods that learn programs from examples as formal inductive synthesis. In this paper, we present a theoretical framework for formal inductive synthesis. We discuss how formal inductive synthesis differs from traditional machine learning. We then describe oracle-guided inductive synthesis (OGIS), a framework that captures a family of synthesizers that operate by iteratively querying an oracle. An instance of OGIS that has had much practical impact is counterexample-guided inductive synthesis (CEGIS). We present a theoretical characterization of CEGIS for learning any program that computes a recursive language. In particular, we analyze the relative power of CEGIS variants where the types of counterexamples generated by the oracle varies. We also consider the impact of bounded versus unbounded memory available to the learning algorithm. In the special case where the universe of candidate programs is finite, we relate the speed of convergence to the notion of teaching dimension studied in machine learning theory. Altogether, the results of the paper take a first step towards a theoretical foundation for the emerging field of formal inductive synthesis.

1 Introduction

The field of formal methods has made enormous strides in recent decades. Formal verification techniques such as model checking [15, 47, 16] and theorem proving (see, e.g. [45, 36, 22]) are used routinely in the computer-aided design of integrated circuits and have been widely applied to find bugs in software, analyze models of embedded systems, and find security vulnerabilities in programs and protocols. At the heart of many of these advances are computational reasoning engines such as Boolean satisfiability (SAT) solvers [41], Binary Decision Diagrams (BDDs) [13], and satisfiability modulo theories (SMT) solvers [8]. Alongside these advances, there has been a growing interest in the synthesis of programs or systems from formal specifications with correctness guarantees. We refer to this area as *formal synthesis*. Starting with the seminal work of Manna and Waldinger on deductive program synthesis [42] and Pnueli and Rosner on reactive synthesis from temporal logic [46], there have been several advances that have made formal synthesis practical in specific application domains such as robotics, online education, and end-user programming.

Algorithmic approaches to formal synthesis range over a wide spectrum, from *deductive synthesis* to *inductive synthesis*. In deductive synthesis (e.g., [42]), a program is synthesized by constructively proving a theorem, employing logical inference and constraint solving. On the other hand, inductive synthesis [19, 57, 52] seeks to find a program matching a set of input-output examples. At a high level, it is thus an instance of learning from examples, also termed as *inductive inference* or *machine learning* [6, 43]. Many current approaches to synthesis blend induction and deduction in the sense that even as they generalize from examples, deductive procedures are used in the process of generalization (see [51, 34] for a detailed exposition). Even so, the term "inductive synthesis" is typically used to refer to all of them. We will refer to these methods as *formal inductive synthesis* to place an emphasis on correctness of the synthesized artifact. These synthesizers generalize from examples by searching a

Submitted to:

restricted space of programs. In machine learning, this restricted space is called the *concept class*, and each element of that space is often called a candidate *concept*. The concept class is usually specified syntactically. It has been recognized that this *syntax guidance*, also termed as a *structure hypothesis*, can be crucial in helping the synthesizer converge quickly to the target concept [55, 51, 1].

The fields of formal inductive synthesis and machine learning have the same high-level goal: to develop algorithmic techniques for *synthesizing a concept* (function, program, or classifier) *from observations* (examples, queries, etc.). However, there are also important differences in the problem formulations and techniques used in both fields. We identify some of the main differences below:

1. *Concept Classes:* In traditional machine learning, the classes of concepts to be synthesized tend to be specialized, such as linear functions or half-spaces [61], convex polytopes [25], neural networks of specific forms [9], Boolean formulas in fixed, bounded syntactic forms [26], and decision trees [48]. However, in formal synthesis, the target concepts are general programs or automata with constraints or finite bounds imposed mainly to ensure tractability of synthesis.
2. *Learning Algorithms:* In traditional machine learning, just as concept classes tend to be specialized, so also are the learning algorithms for those classes [43]. In contrast, in formal inductive synthesis, the trend is towards using general-purpose decision procedures such as SAT solvers, SMT solvers, and model checkers that are not specifically designed for inductive learning.
3. *Exact vs. Approximate Learning:* In formal inductive synthesis, there is a strong emphasis on *exactly* learning the target concept; i.e., the learner seeks to find a concept that is consistent with all positive examples but not with any negative example. The labels for examples are typically assumed to be correct. Moreover, the learned concept should satisfy a formal specification. In contrast, the emphasis in traditional machine learning is on techniques that perform *approximate* learning, where input data can be noisy, some amount of misclassification can be tolerated, there is no formal specification, and the overall goal is to optimize a cost function (e.g., capturing classification error).
4. *Emphasis on Oracle-Guidance:* In formal inductive synthesis, there is a big emphasis on learning in the presence of an oracle, which is typically implemented using a general-purpose decision procedure or sometimes even a human user. Moreover, and importantly, the design of this oracle is part of the design of the synthesizer. In contrast, in traditional machine learning, the use of oracles is rare, and instead the learner typically selects examples from a corpus, often drawing examples independently from an underlying probability distribution. Even when oracles are used, they are assumed to be black boxes that the learner has no control over. The oracle is part of the problem definition in machine learning, whereas in formal inductive synthesis, the design of the oracle is part of the solution.

The last item, oracle-guidance, is a particularly important difference, and informs the framework we proposed in this paper.

In this paper, we take first steps towards a theoretical framework and analysis of formal inductive synthesis. Most instances of inductive synthesis in the literature rely on an oracle that answers different types of queries. In order to capture these various synthesis methods in a unifying framework, we formalize the notion of *oracle-guided inductive synthesis* (OGIS). While we defer a detailed treatment of OGIS to Section 2, we point out three dimensions in which OGIS techniques differ from each other:

1. *Characteristics of concept class:* The concept class for synthesis may have different characteristics depending on the application domain. For instance, the class of programs from which the synthesizer must generate the correct one may be finite, as in the synthesis of bitvector programs [55, 30, 24], or infinite, as in the synthesis of guards for hybrid automata [31, 33]. In the former case, termination is easily guaranteed, but it is not obvious for the case of infinite-size concept classes.
2. *Query types:* Different applications may impose differing constraints on the capabilities of the oracle. In some cases, the oracle may provide only positive examples. When verification engines are

used as oracles, as is typical in formal synthesis, the oracle may provide both positive examples and counterexamples which refute candidate programs. More fine-grained properties of queries are also possible — for instance, an oracle may permit queries that request not just any counterexample, but one that is "minimum" according to some cost function.

3. *Resources available to the learning engine:* As noted above, the learning algorithms in formal inductive synthesis tend to be general-purpose decision procedures. Even so, for tractability, certain constraints may be placed on the resources available to the decision procedure, such as time or memory available. For example, one may limit the decision procedure to use a finite amount of memory, such as imposing an upper bound on the number of (learned) clauses for a SAT solver.

We conduct a theoretical study of OGIS by examining the impact of variations along the above three dimensions. Our work has a particular focus on *counterexample-guided inductive synthesis* (CEGIS) [55], a particularly popular and effective instantiation of the OGIS framework. When the concept class is infinite size, termination of CEGIS is not guaranteed. We study the relative strength of different versions of CEGIS, with regards to their termination guarantees. The versions vary based on the type of counterexamples one can obtain from the oracle. We also analyze the impact of finite versus infinite memory available to the learning algorithm to store examples and hypothesized programs/concepts. Finally, when the concept class is finite size, even though termination of CEGIS is guaranteed, the speed of termination can still be an issue. In this case, we draw a connection between the number of counterexamples needed by a CEGIS procedure and the notion of *teaching dimension* [20] previously introduced in the machine learning literature.

To summarize, we make the following specific contributions in this paper:

1. We define the *formal inductive synthesis* problem and propose a class of solution techniques termed as *Oracle-Guided Inductive Synthesis* (OGIS). We illustrate how OGIS generalizes instances of concept learning in machine learning/artificial intelligence as well as synthesis techniques developed using formal methods. We provide examples of synthesis techniques from literature and show how they can be represented as instantiations of OGIS.
2. We perform a theoretical comparison of different instantiations of the OGIS paradigm in terms of their *synthesis power*. The synthesis power of an OGIS technique is defined as the class of concepts/programs (from an infinite concept class) that can be synthesized using that technique. We establish the following specific novel theoretical results:
   - For learning engines that can use unbounded memory, the power of synthesis engines using oracle that provides arbitrary counterexamples or minimal counterexamples is the same. But this is strictly more powerful than using oracle which provides counterexamples which are bounded by the size of the positive examples.
   - For learning engines that use bounded memory, the power of synthesis engines using arbitrary counterexamples or minimal counterexamples is still the same. The power of synthesis engines using counterexamples bounded by positive examples is not comparable to those using arbitrary/minimal counterexamples. Contrary to intuition, using counterexamples bounded by positive examples allows one to synthesize programs from program classes which cannot be synthesized using arbitrary or minimal counterexamples.
3. For finite concept classes, we prove the NP hardness of the problem of solving the formal inductive synthesis problem for finite domains for a large class of OGIS techniques. We also show that the teaching dimension [20] of the concept class is a lower bound on the number of counterexamples needed for a CEGIS technique to terminate (on an arbitrary program from that class).

The rest of the paper is organized as follows. We first present the *Oracle Guided Inductive Synthesis* (OGIS) paradigm in Section 2. We discuss related work in Section 3. We present the notation and

definitions used for theoretical analysis in Section 4 followed by the theoretical results and their proofs in Section 5 and Section 6. We summarize our results and discuss open problems in Section 7. A preliminary version of this paper appeared in the SYNT 2014 workshop [32].

2 Oracle-Guided Inductive Synthesis: OGIS

We begin by defining some basic terms and notation. Following standard terminology in the machine learning theory community [4], we define a concept c as a set of examples drawn from a domain of examples $\mathbf{E}$. In other words, $c \subseteq \mathbf{E}$. An example $x \in \mathbf{E}$ can be viewed as an input-output behavior of a program; for example, a (pre, post) state for a terminating program, or an input-output trace for a reactive program. Thus, in this paper, we ignore syntactic issues in representing concepts and model them in terms of their semantics, as a set of behaviors. The set of all possible concepts is termed the *concept class*, denoted by $\mathscr{C}$. Thus, $\mathscr{C} \subseteq 2^{\mathbf{E}}$. The concept class may either be specified in the original synthesis problem or arise as a result of a structure hypothesis that restricts the space of candidate concepts. Depending on the application domain, $\mathbf{E}$ can be finite or infinite. The concept class $\mathscr{C}$ can also be finite or infinite. Note that it is possible to place (syntactic) restrictions on concepts so that $\mathscr{C}$ is finite even when $\mathbf{E}$ is infinite.

One key distinguishing characteristic between traditional machine learning and formal inductive synthesis is the presence of an explicit formal specification in the latter. We define a specification Φ as a set of "correct" concepts, i.e., $\Phi \subseteq \mathscr{C} \subseteq 2^{\mathbf{E}}$. Any example $x \in \mathbf{E}$ such that there is a concept $c \in \Phi$ where $x \in c$ is called a *positive example*. Likewise, an example x that is not contained in any $c \in \Phi$ is a *negative example*. We will write $x \vdash \Phi$ to denote that x is a positive example. An example that is specified to be either positive or negative is termed a *labeled example*.

Note that standard practice in formal methods is to define a specification as a set of examples, i.e., $\Phi \subseteq \mathbf{E}$. This is consistent with most properties that are *trace properties*, where Φ represents the set of allowed behaviors — traces, (pre,post) states, etc. — of the program. However, certain practical properties of systems, e.g., certain security policies, are not trace properties (see, e.g., [17]), and therefore we use the more general definition of a specification.

We now define what it means for a concept to satisfy Φ. Given a concept $c \in \mathscr{C}$ we say that c satisfies Φ iff $c \in \Phi$. If we have a complete specification, it means that Φ is a singleton set comprising only a single allowed concept. In general, Φ is likely to be a partial specification that allows for multiple correct concepts.

We now present a first definition of the *formal inductive synthesis* problem:

> Given a *concept class* $\mathscr{C}$ and a domain of examples $\mathbf{E}$, the formal inductive synthesis problem is to find, using only a subset of examples from $\mathbf{E}$, a *target concept* $c \in \mathscr{C}$ that satisfies a specification $\Phi \subseteq \mathscr{C}$.

This definition is reasonable in cases where only elements of $\mathbf{E}$ can be accessed by the synthesis engine — the common case in the use of machine learning methods. However, existing formal verification and synthesis methods can use a somewhat richer set of inputs, including Boolean answers to equivalence (verification) queries with respect to the specification Φ, as well as verification queries with respect to other constructed specifications. Moreover, the synthesis engine typically does not directly access or manipulate the specification Φ. In order to formalize this richer source of inputs as well as the indirect access to Φ, we introduce the concept of an *oracle interface*.

Definition 2.1 *An* oracle interface *$\mathscr{O}$ is a subset of $\mathscr{Q} \times \mathscr{R}$ where $\mathscr{Q}$ is a set of query types, $\mathscr{R}$ is a corresponding set of response types, and $\mathscr{O}$ defines which pairs of query and response types are semantically well-formed.* ■

A simple instance of an oracle interface is one with a single query type that returns positive examples from $\mathbf{E}$. In this case, the synthesis problem is to learn a correct program from purely positive examples. The more common case in machine learning (of classifiers) is to have an oracle that supports two kinds of queries, one that returns positive examples and another that returns negative examples. As we will see in Sec. 2.1, there are richer types of queries that are commonly used in formal synthesis. For now, we will leave $\mathscr{Q}$ and $\mathscr{R}$ as abstract sets.

Implementations of the oracle interface can be nondeterministic algorithms which exhibit nondeterministic choice in the stream of queries and responses. We define the notion of *nondeterministic mapping* to represent such algorithms.

Definition 2.2 *A nondeterministic mapping $F : I \to O$ takes as input $i \in I$ and produces an output $o \in O(i) \subseteq O$ where $O(i)$ is the set of all valid outputs corresponding to input i in F.*

With this notion of an oracle interface, we now introduce our definition of formal inductive synthesis (FIS):

Definition 2.3 *Consider a* concept class *$\mathscr{C}$, a domain of examples $\mathbf{E}$, a specification Φ, and an oracle interface $\mathscr{O}$. The formal inductive synthesis problem is to find a* target concept *$c \in \mathscr{C}$ that satisfies Φ, given only $\mathscr{O}$ and $\mathscr{C}$. In other words, $\mathbf{E}$ and Φ can be accessed only through $\mathscr{O}$.* ■

Thus, an instance of FIS is defined in terms of the tuple $\langle \mathscr{C}, \mathbf{E}, \Phi, \mathscr{O} \rangle$. We next introduce a family of solution techniques for the FIS problem. A FIS problem instance defines an oracle interface and a solution technique for that problem instance can access the domain $\mathbf{E}$ and the specification Φ only through that interface.

2.1 OGIS: A family of synthesizers

Oracle-guided inductive synthesis (OGIS) is an approach to solve the formal inductive synthesis problem defined above, encompassing a family of synthesis algorithms.

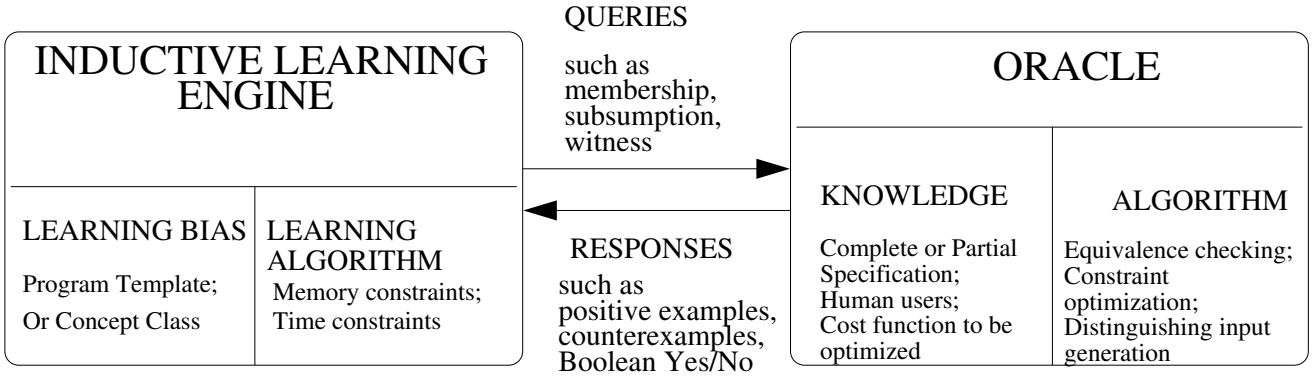

Figure 1: Oracle Guided Inductive Synthesis

As illustrated in Figure 1, OGIS comprises two key components: an *inductive learning engine* (also sometimes referred to as a "Learner") and an *oracle* (also referred to as a "Teacher"). The interaction between the learner and the oracle is in the form of a *dialogue* comprising *queries* and *responses*. The oracle is defined by the types of queries that it can answer, and the properties of its responses. Synthesis is thus an iterative process: at each step, the learner formulates and sends a query to the oracle, and the oracle sends its response. For formal synthesis, the oracle is also tasked with determining whether

the learner has found a correct target concept. Thus, the oracle implicitly or explicitly maintains the specification Φ and can report to the learner when it has terminated with a correct concept.

We first formalize the notions of learner and oracle. Let $\mathbf{Q}$ be a set of queries of types $\mathscr{Q}$, and $\mathbf{R}$ be a set of responses of types $\mathscr{R}$. We allow both $\mathbf{Q}$ and $\mathbf{R}$ to include a special element $\perp$ indicating the absence of a query or response. An element $(q,r) \in \mathbf{Q} \times \mathbf{R}$ is said to *conform* to an oracle interface $\mathscr{O}$ if q is of type q_t, r is of type r_t and $(q_t, r_t) \in \mathscr{O}$. A *valid dialogue pair* for an oracle interface $\mathscr{O}$, denoted d, is a query-response pair (q,r) such that $q \in \mathbf{Q}$, $r \in \mathbf{R}$ and (q,r) conforms to $\mathscr{O}$. The set of valid dialogue pairs for an oracle interface is denoted by $\mathbf{D}$ and $\mathbf{D}^*$ denotes the set of *valid dialogue sequences* — finite sequences of valid dialogue pairs. If $\delta \in \mathbf{D}^*$ is a valid dialogue sequence, $\delta[i]$ denotes a sub-sequence of δ of length i and $\delta(i)$ denotes the i-th dialogue pair in the sequence.

Definition 2.4 *An* oracle *is a nondeterministic mapping* $\mathbf{O} : \mathbf{D}^* \times \mathbf{Q} \to \mathbf{R}$. $\mathbf{O}$ *is consistent with a given interface* $\mathscr{O}$ *iff given a valid dialogue sequence* δ *and a query* q *of type* q_t, $\mathbf{O}(\delta, q)$ *is a response of type* r_t *where* $(q_t, r_t) \in \mathscr{O}$. *A* learner *is a nondeterministic mapping* $\mathbf{L} : \mathbf{D}^* \to \mathbf{Q} \times \mathscr{C}$. $\mathbf{L}$ *is consistent with a given interface* $\mathscr{O}$ *iff given a valid dialogue sequence* δ, $\mathbf{L}(\delta) = (q,c)$ *where* $q \in \mathbf{Q}$ *has type* q_t *such that there exists a response type* r_t *s.t.* $(q_t, r_t) \in \mathscr{O}$. ■

We will further assume in this paper that the oracle $\mathbf{O}$ is *sound*, meaning that it gives a correct response to every query it receives. For example, if asked for a positive example, $\mathbf{O}$ will not return a negative example instead. This notion is left informal for now, since a formalization requires discussion of specific queries and is orthogonal to the results in our paper.

Given the above definitions, we can now define the OGIS approach formally.

Definition 2.5 *Given a FIS* $\langle \mathscr{C}, \mathbf{E}, \Phi, \mathscr{O} \rangle$, *an* oracle-guided inductive synthesis *(*OGIS*) procedure (engine) is a tuple* $\langle \mathbf{O}, \mathbf{L} \rangle$, *comprising an* oracle $\mathbf{O} : \mathbf{D}^* \times \mathbf{Q} \to \mathbf{R}$ *and a* learner $\mathbf{L} : \mathbf{D}^* \to \mathbf{Q} \times \mathscr{C}$, *where the oracle and learner are consistent with the given oracle interface* $\mathscr{O}$ *as defined above.* ■

In other words, an OGIS engine comprises an oracle $\mathbf{O}$ that maps a "dialogue history" and a current query to a response, and a learner $\mathbf{L}$ that, given a dialogue history, outputs a hypothesized concept along with a new query. Upon convergence, the final concept output by $\mathbf{L}$ is the output of the OGIS procedure.

We also formalize the definition of when an OGIS engine solves an FIS problem.

Definition 2.6 *A dialogue sequence* $\delta \in \mathbf{D}^*$ *corresponding to* OGIS *procedure* $\langle \mathbf{O}, \mathbf{L} \rangle$ *is such that* $\delta(i)$ *is* (q,r) *where* $\mathbf{L}(\delta[i-1]) = (q,c)$ *for some query* $q \in \mathbf{Q}$ *and some concept* $c \in \mathscr{C}$, *and* $\mathbf{O}(\delta[i-1], q) = r$.

The OGIS *procedure* $\langle \mathbf{O}, \mathbf{L} \rangle$ *is said to* solve *the FIS problem* with dialogue sequence δ *if there exists an* i *such that* $\mathbf{L}(\delta[i]) = (q,c)$, $c \in \mathscr{C}$ *and* c *satisfies* Φ, *and for all* $j > i$, $\mathbf{L}(\delta[j]) = (q', c)$, *that is, the* OGIS *procedure converges to a concept* $c \in \mathscr{C}$ *that satisfies* Φ.

The OGIS *procedure* $\langle \mathbf{O}, \mathbf{L} \rangle$ *is said to* solve *the FIS problem if there exists a dialogue sequence* δ *with which it solves that problem.*

■

The convergence and computational complexity of an OGIS procedure is determined by the nature of the FIS problem along with three factors: (i) the complexity of each invocation of the learner $\mathbf{L}$; (ii) the complexity of each invocation of the oracle $\mathbf{O}$, and (iii) the number of iterations (queries, examples) of the loop before convergence. We term first two factors as *learner complexity* and *oracle complexity*, and the third as *sample complexity*. Sometimes, in OGIS procedures, oracle complexity is ignored, so that we simply count calls to the oracle rather than the time spent in each call.

An OGIS procedure is defined by properties of the learner and the oracle. Relevant properties of the learner include (i) its *inductive bias* that restricts its search to a particular family of concepts and a search strategy over this space, and (ii) *resource constraints*, such as finite or infinite memory. Relevant properties of the oracle include the types of queries it supports and of the responses it generates. We list

below the common query and response types. In each case, the query type is given in square brackets as a template comprising a query name along with the types of the formal arguments to that query, e.g., examples x or concepts c. An instance of each of these types, that is, a query, is formed by substituting a specific arguments (examples, concepts, etc.) for the formal arguments.

1. *Membership query:* $[q_{\text{mem}}(x)]$ The learner selects an example x and asks "Is x positive or negative?" The oracle responds with a label for x, indicating whether x is a positive or negative example.
2. *Positive witness query:* $[q^{+}_{\text{wit}}]$ The learner asks the oracle "Give me a positive example". The oracle responds with an example $x \vdash \Phi$, if one exists, and with $\perp$ otherwise.
3. *Negative witness query:* $[q^{-}_{\text{wit}}]$ The learner asks the oracle "Give me a negative example". The oracle responds with an example $x \nvdash \Phi$, if one exists, and with $\perp$ otherwise.
4. *Counterexample query:* $[q_{\text{ce}}(c)]$ The learner proposes a candidate concept c and asks "Does the oracle have a counterexample demonstrating that c is incorrect?" (i.e., "proof that $c \notin \Phi$?"). If the oracle can find a counterexample x to $c \notin \Phi$, the oracle provides the counterexample. Otherwise, if the oracle cannot find any counterexample, it responds with $\perp$. Such a query allows us to accurately model the working of counterexample-guided synthesis techniques such as [35] where the verification problem is undecidable but, if a counterexample is reported, it is a true counterexample.
5. *Correctness query:* $[q_{\text{corr}}(c)]$ The learner proposes a candidate concept c and asks "Is c correct?" (i.e., "does it satisfy Φ?"). If so, the oracle responds "YES" (and the synthesis can terminate). If it is not so, the oracle responds "NO" and provides the counterexample. Here x is an example such that either $x \in c$ but $x \nvdash \Phi$, or $x \notin c$ and there exists some other concept $c' \in \Phi$ containing x. This query is a stronger query than counterexample query as it is guaranteed to provide a counterexample whenever the proposed c is not correct.

   For the special case of trace properties, the correctness query can take on specific forms. One form is termed the *equivalence query*, denoted q_{eq}, where the counterexample is in the symmetric difference of the single correct target concept and c. The other is termed the *subsumption query*, denoted q_{sub}, where the counterexample is a negative example present in c, and is used when Φ is a partial specification admitting several correct concepts. It is important to note that, in the general case, a verification query does not, by itself, specify any label for a counterexample. One may need an additional membership query to generate a label for a counterexample.
6. *Crafted Correctness (Verification) query:* $[q_{\text{ccorr}}(\hat{c}, \hat{\Phi})]$ As noted earlier, oracles used in formal inductive synthesis tend to be general-purpose decision procedures. Thus, they can usually answer not only verification queries with respect to the specification Φ for the overall FIS problem, but also verification queries for specifications crafted by the learner. We refer to this class of queries as *crafted correctness/verification queries*. The learner asks "Does $\hat{c}$ satisfy $\hat{\Phi}$?" for a crafted specification $\hat{\Phi}$ and a crafted concept $\hat{c}$.

   As for q_{corr} one can define as special cases a crafted equivalence query type q_{ce} and a crafted subsumption query type q_{csub}.
7. *Distinguishing input query:* $[q_{\text{diff}}(X, c)]$ In this query, the learner supplies a finite set of examples X and a concept c, where $X \subseteq c$, and asks "Does there exist another concept c' s.t. $c \neq c'$ and $X \subseteq c'$?" If so, the oracle responds "YES" and provides both c' and an example $x \in c \ominus c'$. The example x forms a so-called "distinguishing input" that differentiates the two concepts c and c'. If no such c' exists, the oracle responds "NO".

   The distinguishing input query has been found useful in scenarios where it is computationally hard to check correctness using the specification Φ, such as in malware deobfuscation [30].

The query/response types q_{mem}, q^{+}_{wit}, q^{-}_{wit}, q_{ce}, q_{corr}, q_{ccorr} and q_{diff} listed above are not meant to be exhaustive. Any subset of such types can form an oracle interface $\mathcal{O}$. We note here that, in the machine

learning theory community, there have been thorough studies of query-based learning; see Angluin's review paper [5] for details. However, in our formalization of OGIS, new query types such as q_{ccorr} and q_{diff} are possible due to the previously-identified key differences with traditional machine learning including the general-purpose nature of oracle implementations and the ability to select or even design the oracle. Moreover, as we will see, our theoretical analysis raises the following questions that are pertinent in the setting of formal synthesis where the learner and oracle are typically implemented as general-purpose decision procedures:

- *Oracle design:* When multiple valid responses can be made to a query, which ones are better, in terms of convergence to a correct concept (convergence and complexity)?
- *Learner design:* How do resource constraints on the learner or its choice of search strategy affect convergence to a correct concept?

2.2 Examples of OGIS

We now take three example synthesis techniques previously presented in literature and illustrate how they instantiate the OGIS paradigm. These techniques mainly differ in the oracle interface that they employ.

Example 2.1 *Query-based learning of automata [3]:*
Angluin's classic work on learning deterministic finite automata (DFAs) from membership and equivalence queries [3] is an instance of OGIS with $\mathscr{O} = \{q_{\text{mem}}, q_{\text{eq}}\}$. The learner is a custom-designed algorithm called L^*, whereas the oracle is treated as a black box that answers the membership and equivalence queries; in particular, no assumptions are made about the form of counterexamples. Several variants of L^* have found use in the formal verification literature; see [18] for more information.

Example 2.2 *Counterexample-guided inductive synthesis (CEGIS) [55]:*
CEGIS was originally proposed as an algorithmic method for program synthesis where the specification is given as a reference program and the concept class is defined using a partial program, also referred to as a "sketch" [55]. It has since proved very versatile, also applying to partial specifications (see, e.g., [35]) and other ways of providing syntax guidance; see [1] for a more detailed treatment. In CEGIS, the learner (synthesizer) interacts with a "verifier' that can take in a candidate program and a specification, and try to find a counterexample showing that the candidate program does not satisfy the specification. In CEGIS, the learner is typically implemented on top of a general-purpose decision procedure such as a SAT solver, SMT solver, or model checker. The oracle (verifier) is also implemented similarly. In addition to a counterexample-generating oracle, many instances of CEGIS also randomly sample positive examples (see Sec. 5.4 of [55] and Fig. 3 of [35]). Moreover, the counterexample-generating oracle is not required to be a sound verifier that can declare correctness (e.g., see [35]). Thus, we model CEGIS as an instance of OGIS with $\mathscr{O} = \{q^{+}_{\text{wit}}, q_{\text{ce}}\}$.

As noted earlier, if the verifier is sound (can prove correctness of candidate concept), then q_{ce} can be substituted by q_{corr}. Moreover, general-purpose verifiers typically support not only correctness queries with respect to the original specification, but also crafted correctness queries, as well as membership queries, which are special cases of the verification problem where the specification is checked on a single input/output behavior. We term an instantiation of CEGIS with these additional query types as *generalized CEGIS*, which has an oracle interface $\mathscr{O} = \{q^{+}_{\text{wit}}, q_{\text{corr}}, q_{\text{ccorr}}, q_{\text{mem}}\}$. We will restrict our attention in this paper to the standard CEGIS.

Example 2.3 *Oracle-guided program synthesis using distinguishing inputs [30]:*
Our third example is an approach to program synthesis that uses distinguishing inputs when a complete specification is either unavailable or it is expensive to verify a candidate program against its specification [30]. In this case, distinguishing input queries, combined with witness and membership queries,

provide a way to quickly generate a corpus of examples that rule out incorrect programs. When there is only a single program consistent with these examples, only then does a correctness query need to be made to ascertain its correctness. Thus, the oracle interface $\mathscr{O} = \{q^{+}_{\text{wit}}, q_{\text{diff}}, q_{\text{mem}}, q_{\text{corr}}\}$ with q_{corr} being used sparingly. The learner and the oracle are implemented using SMT solving.

2.3 Counterexample-Guided Inductive Synthesis (CEGIS)

Consider the CEGIS instantiation of the OGIS framework. In this paper, we consider a general setting where the concept class $\mathscr{C}$ is the set of programs corresponding to the set of *recursive (decidable) languages*; thus, it is infinite. The domain $\mathbf{E}$ of examples is also infinite. We choose such an expressive concept class and domain because we want to compare how the power of CEGIS varies as we vary the oracle and learner. More specifically, we vary the *nature of responses* from the oracle to correctness and witness queries, and the *memory available* to the learner.

For the oracle, we consider four different types of counterexamples that the oracle can provide in response to a correctness query. Recall that in formal synthesis, oracles are general-purpose verifiers or decision procedures whose internal heuristics may determine the type of counterexample obtained. Each type describes a different oracle and hence, a different flavor of CEGIS. Our goal is to compare these synthesis techniques and establish whether one type of counterexample allows the synthesizer to successfully learn more programs than the other. The four kinds of counterexamples considered in this paper are as follows:

1. *Arbitrary counterexamples:* This is the "standard" CEGIS technique (denoted `CEGIS`) that makes no assumptions on the form of the counterexample obtained from the oracle. Note however that our focus is on an infinite concept class, whereas most practical instantiations of CEGIS have focused on finite concept classes; thus, convergence is no longer guaranteed in our setting. This version of CEGIS serves as the baseline for comparison against other synthesis techniques.
2. *Minimal counterexamples:* We require that the verification oracle provide a counterexample from $\mathbf{E}$ which is minimal for a given ordering over $\mathbf{E}$. The size of examples can be used for ordering. The exact definition of "size" is left abstract and can be defined suitably in different contexts. The intuition is to use counterexamples of smaller size which eliminates more candidate concepts. Significant effort has been made on improving validation engines to produce counterexamples which aid debugging by localizing the error [44, 14]. The use of counterexamples in `CEGIS` conceptually is an iterative repair process and hence, it is natural to extend successful error localization and debugging techniques to inductive synthesis.
3. *Constant-bounded counterexamples:* Here the "size" of the counterexamples produced by the verification oracle is bounded by a constant. This is motivated by the use of bounds in formal verification such as bounded model checking [10] and bug-finding in concurrent programs [7] using bounds on context switches.
4. *Positive-bounded counterexamples:* Here the counterexample produced by the validation engine must be smaller than a previously seen positive example. This is motivated from the industrial practice of validation by simulation where the system is often simulated to a finite length to discover bugs. The length of simulation often depends on the traces which illustrate known positive behaviors. It is expected that errors will show up if the system is simulated up to the length of the largest positive trace. Mutation-based software testing and symbolic execution also has a similar flavor, where a sample correct execution is mutated to find bugs.

In addition to the above variations to the oracle, we also consider two kinds of learners that differ based on their ability to store examples and counterexamples:

1. *Infinite memory:* In the typical setting of CEGIS, the learner is not assumed to have any memory bound, allowing the learner to store as many examples and counterexamples as needed. Note that, for an infinite domain, this set of examples can grow unbounded.
2. *Finite memory:* A more practical setting is one where the learner only has a finite amount of memory, and therefore can only store a finite representation of examples or hypothesized programs. This notion of finite memory is similar to that used classically for language learning from examples [62]. We give the first theoretical results on the power of CEGIS and its variants, for general program synthesis, in this restricted setting.

We introduce notation to refer to these variants in a more compact manner. The synthesis engine using arbitrary counterexamples and with infinite memory is denoted as $T_{\texttt{CEGIS}}$. The variant of the synthesis engine which is restricted to use finite memory is referred to as $T_{\texttt{cegis}}$. Similarly, the synthesis engine using minimal counterexamples and infinite memory is called minimal counterexample guided inductive synthesis ($T_{\texttt{MINCEGIS}}$). The variant of this engine using finite memory is referred to as $T_{\texttt{mincegis}}$. The synthesis engine using counterexamples which are smaller than a fixed constant is called a constant bounded counterexample guided inductive synthesis, and is denoted as $T_{\texttt{CBCEGIS}}$ if the memory is not finite and $T_{\texttt{cbcegis}}$ if the memory is finite. The synthesis engine using counterexamples which are smaller than the largest positive examples is called positive-history bounded counterexample guided inductive synthesis, and is denoted as $T_{\texttt{PBCEGIS}}$ if the memory is not finite and $T_{\texttt{pbcegis}}$ if the memory is finite.

For the class of programs corresponding to the set of recursive languages, our focus is on *learning in the limit*, that is, whether the synthesis technique converges to the correct program or not (see Definition 4.14 in Sec. 4 for a formal definition). This question is non-trivial since our concept class is not finite. In this paper, we do not discuss computational complexity of synthesis, and the impact of different types of counterexamples on the speed of convergence. Investigating the computational complexity for concept classes for which synthesis is guaranteed to terminate is left as a topic for future research.

We also present an initial complexity analysis for OGIS in case of finite concept classes. The decidability question for finite class of programs is trivial since convergence is guaranteed as long as the queries provide new examples or some new information about the target program. But the speed at which the synthesis approach converges remains relevant even for finite class of programs. We show that the complexity of these techniques is related to well-studied notions in learning theory such as the *Vapnik-Chervonenkis dimension* [12] and the *teaching dimension* [20].

3 Background and Related Work

In this section, we contrast the contributions of this paper with the most closely related work and also provide some relevant background.

3.1 Formal Synthesis

The past decade has seen an explosion of work in program synthesis (e.g. [54, 55, 30, 56, 37, 58]. Moreover, there has been a realization that many of the trickiest steps in formal verification involve synthesis of artifacts such as inductive invariants, ranking functions, assumptions, etc. [51, 23]. Most of these efforts have focused on solution techniques for specific synthesis problems. There are two main unifying characteristics across most of these efforts: (i) syntactic restrictions on the space of programs/artifacts to be synthesized in the form of templates, sketches, component libraries, etc., and (ii) the use of inductive synthesis from examples. The recent work on *syntax-guided synthesis* (SyGuS) [1] is an attempt to capture these disparate efforts in a common theoretical formalism. While SyGuS is about formalizing the

synthesis *problem*, the present paper focuses on formalizing common ideas in the *solution techniques*. Specifically, we present OGIS as a unifying formalism for different solution techniques, along with a theoretical analysis of different variants of CEGIS, the most common instantiation of OGIS. In this sense, it is complementary to the SyGuS effort.

3.2 Machine Learning Theory

Another related area is the field of machine learning, particularly the theoretical literature. In Section 1, we outlined some of the key differences between the fields of formal inductive synthesis and that of machine learning. Here we focus on the sub-field of *query-based learning* that is the closest to the OGIS framework. The reader is referred to Angluin's excellent papers on the topic for more background [4, 5].

A major difference between the query-based learning literature and our work is in the treatment of oracles, specifically, how much control one has over the oracle that answers queries. In query-based learning, the oracles are treated as black boxes that answer particular types of queries and only need to provide one valid response to a query. Moreover, it is typical in the query-based learning literature for the oracle to be specified a priori as part of the problem formulation. In contrast, in our OGIS framework, designing a synthesis procedure involves also designing or selecting an oracle. The second major difference is that the query-based learning literature focuses on specific concept classes and proves convergence and complexity results for those classes. In contrast, our work proves results that are generally applicable to programs corresponding to recursive languages.

3.3 Learning of Formal Languages

The problem of learning a formal language from examples is a classic one. We cover here some relevant background material.

Gold [19] considered the problem of learning formal languages from examples. Similar techniques have been studied elsewhere in literature [29, 63, 11, 2]. The examples are provided to learner as an infinite stream. The learner is assumed to have unbounded memory and can store all the examples. This model is unrealistic in a practical setting but provides useful theoretical understanding of inductive learning of formal languages. Gold defined a class of languages to be *identifiable in the limit* if there is a learning procedure which identifies the grammar of the target language from the class of languages using a stream of input strings. The languages learnt using only positive examples were called *text learnable* and the languages which require both positive and negative examples were termed *informant learnable*. None of the standard classes of formal languages are identifiable in the limit from text, that is, from only positive examples [19]. This includes regular languages, context-free languages and context-sensitive languages.

A detailed survey of classical results in learning from positive examples is presented by Lange et al. [39]. The results summarize learning power with different limitations such as the inputs having certain noise, that is, a string not in the target language might be provided as a positive example with a small probability. Learning using positive as well as negative examples has also been well-studied in literature. A detailed survey is presented in [27] and [38]. Lange and Zilles [40] relate Angluin-style query-based learning with Gold-style learning. They establish that any query learner using superset queries can be simulated by a Gold-style learner receiving only positive data. But there are concepts learnable using subset queries but not Gold-style learnable from positive data only. Learning with equivalence queries coincides with Gold's model of limit learning from positive and negative examples, while learning with membership queries equals finite learning from positive data and negative data. In contrast to this line of work, we present a general framework OGIS to learn programs or languages and Angluin-style or Gold-

style approaches can be instantiated in this framework. Our theoretical analysis focusses on varying the oracle and the nature of counterexample produced by it to examine the impact of using different types of counterexamples obtainable from verification or testing tools.

3.4 Learning vs. Teaching

We also study the complexity of synthesizing programs from a finite class of programs. This part of our work is related to previous work on the complexity of *teaching* in exact learning of concepts by Goldman and Kearns [20]. Informally, the teaching dimension of a concept class is the minimum number of instances a teacher must reveal to uniquely identify any target concept from the class. Exact bounds on teaching dimensions for specific concept classes such as orthogonal rectangles, monotonic decision trees, monomials, binary relations and total orders have been previously presented in literature [20, 21]. Shinohara et al. [53] also introduced a notion of teachability in which a concept class is teachable by examples if there exists a polynomial size sample under which all consistent learners will exactly identify the target concept. Salzberg et al. [50] also consider a model of learning with a helpful teacher. Their model requires that any teacher using a particular algorithm such as the nearest-neighbor algorithm learns the target concept. This work assumes that the teacher knows the algorithm used by the learner. We do not make any assumption on the inductive learning technique used by the OGIS synthesis engine. Our goal is to obtain bounds on the number of examples that need to be provided by the oracle to synthesize the correct program by relating our framework to the literature on teaching.

4 Theoretical Analysis of CEGIS: Preliminaries

Our presentation of formal inductive synthesis and OGIS so far has not used a particular representation of a concept class or specification. In this section, we begin our theoretical formalization of the counterexample-guided inductive synthesis (CEGIS) technique, for which such a choice is necessary. We precisely define the formal inductive synthesis problem for concepts that correspond to recursive languages. We restrict our attention to the case when the specification is partial and is a trace property — i.e., the specification is defined by a single formal language. This assumption, which is the typical case in formal verification and synthesis, also simplifies notation and proofs. Most of our results extend to the case of more general specifications; we will make suitable additional remarks about the general case where needed. For ease of reference, the major definitions and frequently used notation are summarized in Table 1.

4.1 Basic Notation

We use $\mathbb{N}$ to denote the set of natural numbers. $\mathbb{N}_i \subset \mathbb{N}$ denotes a subset of natural numbers $\mathbb{N}_i = \{n | n < i\}$. Consider a set $S \subset \mathbb{N}$. $\min(S)$ denotes the minimal element in S. The union of the sets is denoted by $\cup$ and the intersection of the sets is denoted by $\cap$. $S_1 \setminus S_2$ denotes set minus operation with the resultant set containing all elements in S_1 and not in S_2.

We denote the set $\mathbb{N} \cup \{\perp\}$ as $\mathbb{N}_\perp$. A sequence σ is a mapping from $\mathbb{N}$ to $\mathbb{N}_\perp$. We denote a prefix of length k of a sequence by $\sigma[k]$. So, $\sigma[k]$ of length k is a mapping from $\mathbb{N}_k$ to $\mathbb{N}_\perp$. $\sigma[0]$ is an empty sequence also denoted by σ_0 for brevity. The set of natural numbers appearing in the sequence $\sigma[i]$ is defined using a function SAMPLE, where $\texttt{SAMPLE}(\sigma[i]) = \mathit{range}(\sigma[i]) - \{\perp\}$. The set of sequences is denoted by Σ.

Languages and Programs: We also use standard definitions from computability theory which relate languages and programs [49]. A set L of natural numbers is called a computable or recursive language if

there is a program, that is, a computable, total function P such that for any natural number n,

$$P(n) = 1 \texttt{ if } n \in L \texttt{ and } P(n) = 0 \texttt{ if } n \notin L$$

We say that P *identifies* the language L. Let $L_{map}(P)$ denote the language L identified by the program P. The mapping L_{map} is not necessarily one-to-one and hence, syntactically different programs might identify the same language. In formal synthesis, we do not distinguish between syntactically different programs that satisfy the specification. Additionally, in this paper, we restrict our discussion to recursive languages because it includes many interesting and natural classes of languages that correspond to programs and functions of various kinds, including regular, context free, context sensitive, and pattern languages.

Given a sequence of non-empty languages $\mathscr{L} = L_0, L_1, L_2, \ldots$, $\mathscr{L}$ is said to be an indexed family of languages if and only if for all languages L_i, there exists a recursive function TEMPLATE such that $\texttt{TEMPLATE}(j,n) = P(n)$ and $L_{map}(P) = L_i$ for some j. Practical applications of program synthesis often consider a family of candidate programs which contain syntactically different programs that are semantically equivalent, that is, they have the same set of behaviors. Formally, in practice program synthesis techniques permit picking j such that $\texttt{TEMPLATE}(j,n) = P(n)$ and $L_{map}(P) = L_i$ for all $j \in I_j$ where the set I_j represents the syntactically different but semantically equivalent programs that produce output 1 on an input if and only if the input natural number belongs to L_i. Intuitively, a function TEMPLATE defines an encoding of the space of candidate programs similar to encodings proposed in the literature such as those on program sketching [55] and component interconnection encoding [30]. In the case of formal synthesis where we have a specification Φ, we are only interested in finding a single program satisfying Φ. In the general case, Φ comprises a set of allowed languages, and the task of synthesis is to find a program identifying some element of this set. In the case of partial specifications that are trace properties, Φ comprises subsets of a single target language L_c. Any program P_c identifying some subset of L_c is a valid solution, and usually positive examples are used to rule out programs identifying "uninteresting" subsets of L_c. Thus, going forward, we will define the task of program synthesis as one of identifying the corresponding correct language L_c.

Ordering of elements in the languages: A language corresponds to a set of program behaviors. We model this set in an abstract manner, only assuming the presence of a total order over this set, without prescribing any specific ordering relation. Thus, languages are modeled as sets of natural numbers. While such an assumption might seem restrictive, we argue that this is not the case in the setting of CEGIS, where the ordering relation is used specifically to model the oracle's preference for returning specific kinds of counterexamples. For example, consider the case where elements of a language are input/output traces. We can construct a totally ordered set of all possible input/output traces using the length of the trace as the primary ordering metric and the lexicographic ordering as the secondary ordering metric. Thus, an oracle producing smallest counterexample would produce an unique trace which is shortest in length and is lexicographically the smallest. The exact choice of ordering is orthogonal to results presented in our paper, and using the natural numbers allows us to greatly simplify notation.

4.2 CEGIS Definitions

We now specialize the definitions from Sec. 2 for the case of CEGIS. An indexed family of languages (also called a language class) $\mathscr{L}$ defines the concept class $\mathscr{C}$ for synthesis. The domain $\mathbf{E}$ for synthesis is the set of natural numbers $\mathbb{N}$ and the examples are $i \in \mathbb{N}$. Recall that we restrict our attention to the special case where the specification Φ is captured by a single target language, i.e., L_c comprising all permitted program behaviors. Therefore, the formal inductive synthesis (FIS) problem defined in Section 2 (Definition 2.3) can be restricted for this setting as follows:

Definition 4.1 *Given a* language class $\mathscr{L}$*, a domain of examples* $\mathbb{N}$*, the specification* Φ *defined by a target language* L_c*, and an oracle interface* $\mathscr{O}$*, the problem of formal inductive synthesis of languages (and the associated programs) is to identify a language in* Φ *using only the oracle interface* $\mathscr{O}$*.*

Counterexample-guided inductive synthesis (CEGIS) is a solution to the problem of formal inductive synthesis of languages where the oracle interface $\mathscr{O}$ is defined as follows.

Definition 4.2 *A* counterexample-guided inductive synthesis (CEGIS) oracle interface *is* $\mathscr{O} = \mathscr{Q} \times \mathscr{R}$ *where* $\mathscr{Q} = \{q^+_{wit}, q_{ce}(L)\}$ *with* $L \in \mathscr{L}$*,* $\mathscr{R} = \mathbb{N}_\perp$*, and the specification* Φ *is defined as subsets of a target language* L_c*. The positive witness query* q^+_{wit} *returns a positive example* $i \in L_c$*, and the counterexample query* q_{ce} *takes as argument a candidate language* L *and either returns a counterexample* $i \in L \setminus L_c$ *showing that the candidate language* L *is incorrect or returns* $\perp$ *if it cannot find a counterexample.* [1]

Symbol	Meaning	Symbol	Meaning
$\mathbb{N}$	natural numbers	$\mathbb{N}_i$	natural numbers less than i
$\min(S)$	minimal element in set S	$S_1 \setminus S_2$	set minus
$S_1 \cap S_2$	set intersection	$S_1 \cup S_2$	set union
σ	sequence of numbers	σ_0	empty sequence
$\sigma[i]$	sequence of length i	$\sigma(i)$	ith element of sequence σ
L_i	language (a subset of $\mathbb{N}$)	$\overline{L}_i$	complement of language
P_i	program for L_i	$L_{map}(P_i) = L_i$	language corresponding to P_i
$\texttt{SAMPLE}(\sigma)$	natural numbers in σ	Σ	set of sequences
$\mathscr{L}$	family of languages	$\mathscr{P}$	family of programs
τ	transcript	`cex`	counterexample transcript
T	synthesis engine	`learn`	inductive learning engine
$\texttt{CHECK}_L$	verification oracle for L	$\texttt{MINCHECK}_L$	minimal counterexample oracle
$\texttt{CBCHECK}_{B,L}$	bounded counterexample oracle	$\texttt{PBCHECK}_L$	positive bounded counterexample oracle
`CEGIS`	set of language families identified by inf memory cegis engine	`cegis`	set of language families identified by finite memory cegis engine
`MINCEGIS`	`CEGIS` with `MINCHECK`	`mincegis`	`cegis` with `MINCHECK`
`CBCEGIS`	`CEGIS` with `CBCHECK` for a given constant B	`cbcegis`	`cegis` with `CBCHECK` for a given constant B
`PBCEGIS`	`CEGIS` with `PBCHECK`	`pbcegis`	`cegis` with `PBCHECK`

Table 1: Frequently used notation in the paper

The sequence τ of responses of the positive witness q^+_{wit} query is called the *transcript*, and the sequence `cex` of the responses to the counterexample queries q_{ce} is called the *counterexample sequence*. The positive witness queries can be answered by the oracle sampling examples from the target language. Our work uses the standard model for language learning in the limit [19], where the learner has access to an infinite stream of positive examples from the target language. This is also realistic in practical CEGIS settings for infinite concept classes (e.g. [35]) where more behaviors can be sampled over time. We formalize these terms below.

Definition 4.3 *A* transcript τ *for a specification language* L_c *is a sequence with* $\texttt{SAMPLE}(\tau) = L_c$*.* $\tau[i]$ *denotes the prefix of the transcript* τ *of length* i*.* $\tau(i)$ *denotes the* i*-th element of the transcript.*

[1] CEGIS techniques in literature [55, 35] initiate search for correct program using positive examples and use specification to obtain positive examples corresponding to counterexamples.

Definition 4.4 *A* counterexample sequence $\texttt{cex}$ *for a specification language L_c from a counterexample query q_{ce} is a sequence with $\texttt{cex}(i) = q_{ce}(L_{cand_i})$, where $\texttt{cex}[i]$ denotes the prefix of the counterexample sequence* $\texttt{cex}$ *of length i, $\texttt{cex}(i)$ denotes the i-th element of the counterexample sequence, and L_{cand_i} is the argument of the i-th invocation of the query q_{ce}.*

We now define the verification oracle in CEGIS that produces arbitrary counterexamples, as well as its three other variants which generate particular kinds of counterexamples.

Definition 4.5 *A verifier $\texttt{CHECK}_L$ for language L is a nondeterministic mapping from $\mathscr{L}$ to $\mathbb{N}_\perp$ such that $\texttt{CHECK}_L(L_i) = \perp$ if and only if $L_i \subseteq L$, and $\texttt{CHECK}_L(L_i) \in L_i \setminus L$ otherwise.*

Remark: For more general specifications Φ that are a set of languages, the definition of $\texttt{CHECK}_L$ changes in a natural way: it returns $\perp$ if and only if $L_i \in \Phi$ and otherwise returns an example j that is in the intersection of the symmetric differences of each language $L \in \Phi$ and the candidate language L_i.

We define a minimal counterexample generating verifier below. The counterexamples are minimal with respect to the total ordering on the domain of examples.

Definition 4.6 *A verifier $\texttt{MINCHECK}_L$ for a language L is a nondeterministic mapping from $\mathscr{L}$ to $\mathbb{N}_\perp$ such that $\texttt{MINCHECK}_L(L_i) = \perp$ if and only if $L_i \subseteq L$, and $\texttt{MINCHECK}_L(L_i) = \min(L_i \setminus L)$ otherwise.*

Next, we consider another variant of counterexamples, namely (constant) bounded counterexamples. Bounded model-checking [10] returns a counterexample trace for an incorrect design if it can find a counterexample of length less than the specified constant bound. It fails to find a counterexample for an incorrect design if no counterexample exists with length less than the given bound. Verification of concurrent programs by bounding the number of context switches [7] is another example of the bounded verification technique. This motivates the definition of a verifier which returns counterexamples bounded by a constant B.

Definition 4.7 *A verifier $\texttt{CBCHECK}_{B,L}$ is a nondeterministic mapping from $\mathscr{L}$ to $\mathbb{N}_\perp$ such that $\texttt{CBCHECK}_{B,L}(L_i) =$ m where $m \in L_i \setminus L \wedge m < B$ for the given bound B, and $\texttt{CBCHECK}_{B,L}(L_i) = \perp$ if such m does not exist.*

The last variant of counterexamples is *positive bounded* counterexamples. The verifier for generating positive bounded counterexample is also provided with the transcript seen so far by the synthesis engine. The verifier generates a counterexample smaller than the largest positive example in the transcript. If there is no counterexample smaller than the largest positive example in the transcript, then the verifier does not return any counterexample. This is motivated by the practice of mutating correct traces to find bugs in programs and designs. The counterexamples in these techniques are bounded by the size of positive examples (traces) seen so far.[2]

Definition 4.8 *A verifier $\texttt{PBCHECK}_L$ is a nondeterministic mapping from $\mathscr{L} \times \Sigma$ to $\mathbb{N}_\perp$ such that $\texttt{PBCHECK}_L(L_i, \tau[n]) =$ m where $m \in L_i \setminus L \wedge m < \tau(j)$ for some $j \leq n$, and $\texttt{PBCHECK}_L(L_i, \tau[n]) = \perp$ if such m does not exist.*

We now define the oracle for counterexample guided inductive synthesis. We drop the queries in dialogue since there are only two kind of queries and instead only use the sequence of responses: transcript τ and the counterexample sequence $\texttt{cex}$. The oracle also receives as input the current candidate language L_{cand} to be used as the argument of the q_{corr} query. The overall response of the oracle is a pair of elements in $\mathbb{N}_\perp$.

[2]Note that we can extend this definition to include counterexamples of size bounded by that of the largest positive example seen so far plus a constant. The proof arguments given in Sec. 5 continue to work with only minor modifications.

Definition 4.9 *An oracle* $\mathbf{O}$ *for counterexample-guided inductive synthesis (*CEGIS oracle*) is a nondeterministic mapping* $\Sigma \times \Sigma \times \mathscr{L} \to \mathbb{N}_\perp \times \mathbb{N}_\perp$ *such that* $\mathbf{O}(\tau[i-1], \mathtt{cex}[i-1], L_{cand}) = (\tau(i), \mathtt{cex}(i))$ *where* $\tau(i)$ *is the nondeterministic response to positive witness query* q^+_{wit} *and* $\mathtt{cex}(i)$ *is the nondeterministic response to counterexample query* $q_{ce}(L_{cand})$*. The oracle can use any of the four verifiers presented earlier to generate the counterexamples. An oracle using* $\mathtt{CHECK}_L$ *is called* $\mathbf{O}_{\mathtt{cegis}}$*, one using* $\mathtt{MINCHECK}_L$ *is called* $\mathbf{O}_{\mathtt{mincegis}}$*, one using* $\mathtt{PBCHECK}_L$ *is called* $\mathbf{O}_{\mathtt{pbcegis}}$ *and one using* $\mathtt{CBCHECK}_{B,L}$ *is called* $\mathbf{O}_{\mathtt{cbcegis}}$*.*

We make the following reasonable assumption on the oracle. The oracle is assumed to be *consistent*: it does not provide the same example both as a positive example (via a positive witness query) and as a negative example (as a counterexample). Second, the oracle is assumed to be *non-redundant*: it does not repeat any positive examples that it may have previously provided to the learner; for a finite target language, once the oracle exhausts all positive examples, it will return $\perp$.

The learner is simplified to be a mapping from the sequence of responses to a candidate program.

Definition 4.10 *An* infinite memory learner $\mathtt{LEARN}$ *is a function* $\Sigma \times \Sigma \to \mathscr{L}$ *such that* $\mathtt{LEARN}(\tau[n], \mathtt{cex}[n]) = L$ *where* L *includes all positive examples in* $\tau[n]$ *and excludes all examples in* $\mathtt{cex}[n]$*.*[3] $\mathtt{LEARN}(\sigma_0, \sigma_0)$ *is a predefined constant representing an initial guess* L_0 *of the language, which, for example, could be* $\mathbb{N}$*.*

We now define a finite memory learner which cannot take the unbounded sequence of responses as argument. The finite memory learner instead uses the previous candidate program to summarize the response sequence. We assume that languages are encoded in terms of a finite representation (index of the language since the language class is an indexed family of languages and we assume that every index needs unit memory) such as a program that identifies that language. Such an iterative learner only needs finite memory.

Definition 4.11 *A* finite memory learner $\mathtt{learn}$ *is a recursive function* $\mathscr{L} \times \mathbb{N}_\perp \times \mathbb{N}_\perp \to \mathscr{L}$ *such that for all* $n \geq 0$*,* $\mathtt{learn}(L_n, \tau(n), \mathtt{cex}(n)) = L_{n+1}$*, where* L_{n+1} *includes all positive examples in* $\tau[n]$ *and excludes all examples in* $\mathtt{cex}[n]$*. We define* $L_0 = \mathtt{LEARN}(\sigma_0, \sigma_0)$ *to be the initial guess of the language, which for example, could be* $\mathbb{N}$*. For ease of presentation, we omit the finite memory available to the learner in its functional representation above. The learner can store additional finite information.*

The synthesis engine using infinite memory can now be defined as follows.

Definition 4.12 *An infinite memory* $\mathtt{CEGIS}$ *engine* $T_{\mathtt{CEGIS}}$ *is a pair* $\langle \mathbf{O}_{\mathtt{cegis}}, \mathtt{LEARN} \rangle$ *comprising a CEGIS oracle* $\mathbf{O}_{\mathtt{cegis}}$ *and an infinite memory learner* $\mathtt{LEARN}$*, where, there exists* τ *and* $\mathtt{cex}$ *such that for all* $i \geq 0$*,* $\mathbf{O}_{\mathtt{cegis}}(\tau[i], \mathtt{cex}[i], L_i) = (\tau(i+1), \mathtt{cex}(i+1))$ *and* $L_i = \mathtt{LEARN}(\tau[i], \mathtt{cex}[i])$*. Since the oracle* $\mathbf{O}_{\mathtt{cegis}}$ *is nondeterministic,* $T_{\mathtt{CEGIS}}$ *can have multiple transcripts* τ *and counterexample sequences* $\mathtt{cex}$*.*

A synthesis engine with finite memory cannot store unbounded infinite transcripts. So, the bounded memory $\mathtt{cegis}$ synthesis engine $T_{\mathtt{cegis}}$ uses a finite memory learner $\mathtt{learn}$.

Definition 4.13 *A finite memory* $\mathtt{cegis}$ *engine* $T_{\mathtt{cegis}}$ *is a tuple* $\langle \mathbf{O}_{\mathtt{cegis}}, \mathtt{learn} \rangle$ *comprising a CEGIS oracle* $\mathbf{O}_{\mathtt{cegis}}$ *and a finite memory learner* $\mathtt{learn}$ *where, there exists* τ *and* $\mathtt{cex}$ *such that for all* $i \geq 0$*,* $\mathbf{O}_{\mathtt{cegis}}(\tau[i], \mathtt{cex}[i], L_{i+1}) = (\tau(i+1), \mathtt{cex}(i+1))$ *and* $L_i = \mathtt{learn}(L_i, \tau(i), \mathtt{cex}(i))$*. Since the oracle* $\mathbf{O}_{\mathtt{cegis}}$ *is nondeterministic,* $T_{\mathtt{CEGIS}}$ *can have multiple transcripts* τ *and counterexample sequences* $\mathtt{cex}$*.*

A pair $(\tau, \mathtt{cex})$ is a valid transcript and counterexample sequence for $T_{\mathtt{cegis}}$ if the above definitions hold for that pair. We denote this by $(\tau, \mathtt{cex}) \models T_{\mathtt{cegis}}$. Similar to Definition 2.5, the convergence of the counterexample-guided synthesis engine is defined as follows:

[3] This holds due to the specialization of Φ to a partial specification, and as a trace property. For general Φ, the learner need not exclude all counterexamples.

Definition 4.14 *We say that* $T_{\texttt{cegis}} : \langle \mathbf{O}_{\texttt{cegis}}, \texttt{learn} \rangle$ *identifies L, that is, it converges to L, written* $T_{\texttt{cegis}} \rightarrow L$ *if and only if there exists k such that for all* $n \geq k$, $\texttt{learn}(L_n, \tau[n], \texttt{cex}[n]) = L$ *for all valid transcripts* τ *and counterexample sequences* cex *of* $T_{\texttt{cegis}}$.

This notion of convergence is standard in language learning in the limit [19]. For the case of general specifications Φ, as given in Definition 2.5, the synthesizer must converge to *some* language in Φ. As per Definition 4.3, a transcript is an infinite sequence of examples which contains all the elements in the target language. Definition 4.14 requires the synthesis engine to converge to the correct language after consuming a *finite* part of the transcript and counterexample sequence. This notion of convergence is standard in the literature on language learning in the limit [19] [4].

We extend Definition 4.14 to general specifications Φ as follows: $T_{\texttt{cegis}}$ identifies a specification Φ if it identifies *some* language in Φ. As noted before, this section focuses on the case of a partial specification that is a trace property. In this case, Φ comprises all subsets of a target language L_c. Since Definition 4.3 defines a transcript as comprising all positive examples in L_c and Definition 4.14 requires convergence for all possible transcripts, the two notions of identifying Φ and identifying L_c coincide. We therefore focus in Sec. 5 purely on language identification with the observation that our results carry over to the case of "specification identification".

Definition 4.15 $T_{\texttt{cegis}} = \langle \mathbf{O}_{\texttt{cegis}}, \texttt{learn} \rangle$ *identifies a language family* $\mathscr{L}$ *if and only if* $T_{\texttt{cegis}}$ *identifies every language* $L \in \mathscr{L}$.

The above definition extends to families of specifications in an exactly analogous manner. We now define the set of language families that can be identified by the inductive synthesis engines as cegis formally below.

Definition 4.16 $\texttt{cegis} = \{ \mathscr{L} \mid \exists \texttt{learn}\, \forall \mathbf{O}_{\texttt{cegis}} \,.$ *the engine* $T_{\texttt{cegis}} = \langle \mathbf{O}_{\texttt{cegis}}, \texttt{learn} \rangle$ *identifies* $\mathscr{L} \}$.

The convergence of synthesis engine to the correct language, identification condition for a language, and language family identified by a synthesis engine are defined similarly as listed in Table 2.

Learner / Oracle	$\mathbf{O}_{\texttt{cegis}}$	$\mathbf{O}_{\texttt{mincegis}}$	$\mathbf{O}_{\texttt{pbcegis}}$
Finite memory learn	$T_{\texttt{cegis}}$, cegis	$T_{\texttt{mincegis}}$, mincegis	$T_{\texttt{pbcegis}}$, pbcegis
Infinite memory LEARN	$T_{\texttt{CEGIS}}$, CEGIS	$T_{\texttt{MINCEGIS}}$, MINCEGIS	$T_{\texttt{PBCEGIS}}$, PBCEGIS

Table 2: Synthesis engines and corresponding sets of language families

The constant bounded counterexample-guided inductive synthesis oracle $\mathbf{O}_{\texttt{cbcegis}}$ uses the verifier $\texttt{CBCHECK}_{B,L}$. It takes an additional parameter B which is the constant bound on the maximum size of a counterexample. If the verifier cannot find a counterexample below this bound, it will respond with $\perp$.

Definition 4.17 *Given a bound B,* $T_{\texttt{cbcegis}} = \langle \mathbf{O}_{\texttt{cbcegis}}, \texttt{learn} \rangle$ *where* $\mathbf{O}_{\texttt{cbcegis}}$ *uses* $\texttt{CBCHECK}_{B,L}$*, we say that* $T_{\texttt{cbcegis}}$ *identifies a language family* $\mathscr{L}$ *if and only if* $T_{\texttt{cbcegis}}$ *identifies every language* $L \in \mathscr{L}$.

Note that the values of B for which a language family $\mathscr{L}$ is identifiable can be different for different $\mathscr{L}$. The overall class of language families identifiable using $\mathbf{O}_{\texttt{cbcegis}}$ oracles can thus be defined as follows:

Definition 4.18 $\texttt{cbcegis} = \{ \mathscr{L} \mid \exists B \;\; \exists \texttt{learn} \,.\, \forall \mathbf{O}_{\texttt{cbcegis}}\, s.t. \mathbf{O}_{\texttt{cbcegis}}$ *uses* $\texttt{CBCHECK}_{B,L}$ *. the engine* $T_{\texttt{cegis}} = \langle \mathbf{O}_{\texttt{cbcegis}}, \texttt{learn} \rangle$ *identifies* $\mathscr{L} \}$

[4] In this framework, a synthesis engine is only required to converge to the correct concept without requiring it to recognize it has converged and terminate. For a finite concept or language, termination can be trivially guaranteed when the oracle is assumed to be non-redundant and does not repeat examples.

5 Theoretical Analysis of CEGIS: Results

In this section, we present the theoretical results when the class of languages (programs) is infinite. We consider two axes of variation. We first consider the case in which the inductive learning technique has finite memory in Section 5.1, and then the case in which it has infinite memory in Section 5.2. For both cases, we consider the four kinds of counterexamples mentioned in Section 1 and Section 4; namely, arbitrary counterexamples, minimal counterexamples, constant bounded counterexamples and positive bounded counterexamples.

For simplicity, our proofs focus on the case of partial specifications that are trace properties, the common case in formal verification and synthesis. Thus, Φ comprises subsets of a target specification language L_c. However, many of the results given here extend to the case of general specifications. Most of our theorems show differences between language classes for CEGIS variants — i.e., theorems showing that there is a specification on which one variant of CEGIS converges while the other does not — and for these, it suffices to show such a difference for the more restricted class of partial specifications. The results also extend to the case of equality between language classes (e.g., Theorem 5.1) in certain cases; we make suitable remarks alongside.

5.1 Finite Memory Inductive Synthesis

We investigate the four language classes $\texttt{cegis}, \texttt{mincegis}, \texttt{cbcegis}$ and $\texttt{pbcegis}$ identified by the synthesis engines $T_{\texttt{cegis}}$, $T_{\texttt{mincegis}}$, $T_{\texttt{cbcegis}}$ and $T_{\texttt{pbcegis}}$ and establish relations between them. We show that $\texttt{cbcegis} \subset \texttt{mincegis} = \texttt{cegis}$, $\texttt{pbcegis} \not\subseteq \texttt{cegis}$ and $\texttt{pbcegis} \not\supseteq \texttt{cegis}$.

5.1.1 Minimal vs. Arbitrary Counterexamples

We begin by showing that replacing a deductive verification engine which returns arbitrary counterexamples with a deductive verification engine which returns minimal counterexamples does not change the power of counterexample-guided inductive synthesis. The result is summarized in Theorem 5.1.

Theorem 5.1 *The power of synthesis techniques using arbitrary counterexamples and those using minimal counterexamples are equivalent, that is,* $\texttt{mincegis} = \texttt{cegis}$.

Proof $\texttt{MINCHECK}_L$ is a special case of $\texttt{CHECK}_L$ in that a minimal counterexample reported by $\texttt{MINCHECK}_L$ can be treated as arbitrary counterexample to simulate $T_{\texttt{cegis}}$ using $T_{\texttt{mincegis}}$. Thus, $\texttt{cegis} \subseteq \texttt{mincegis}$.

The more interesting case to prove is $\texttt{mincegis} \subseteq \texttt{cegis}$. For a language L, let $\texttt{mincegis}$ converge to the correct language L on transcript τ. We show that $T_{\texttt{cegis}}$ can simulate $T_{\texttt{mincegis}}$ and also converge to L on transcript τ. The proof idea is to show that a finite learner can simulate $\texttt{MINCHECK}_L$ by making a finite number of calls to $\texttt{CHECK}_L$. Therefore, the learner sees the same counterexample sequence with $\texttt{CHECK}_L$ as with $\texttt{MINCHECK}_L$ and thus converges to the same language in both cases.

Consider an arbitrary step of the dialogue between learner and verifier when a counterexample is returned. Let the arbitrary counterexample returned by the verifier for a candidate language L_i be c, that is $\texttt{CHECK}_L(L_i) = c$. Thus, c is an upper bound on the minimal counterexample returned by $\texttt{MINCHECK}_L$. The latter can be recovered using the following characterization:

$$\texttt{MINCHECK}_L(L_i) = \text{ minimum } j \text{ such that } \texttt{CHECK}_L(\{j\}) \text{ is not } \perp \text{ for } 0 \leq j \leq \texttt{CHECK}_L(L_i)$$

The learner can thus perform at most c queries to $\texttt{CHECK}_L$ to compute the minimal counterexample that would be returned by $\texttt{MINCHECK}_L$. In case of totally ordered set (such as $\mathbb{N}$), we could do this more efficiently using binary search. At each stage of the iteration, the learner needs to store the smallest

counterexample returned so far. Thus, the work performed by the learner in each iteration to craft queries to $\texttt{CHECK}_L$ can be done with finite memory. $\texttt{MINCHECK}_L(L_i)$ can be computed using finite memory and using at most $c = \texttt{CHECK}_L(L_i)$ calls of $\texttt{CHECK}_L$.

Thus, $T_{\texttt{cegis}}$ can simulate $T_{\texttt{mincegis}}$ by finding the minimal counterexample at each step using the verifier CHECK iteratively as described above. This implies that mincegis = cegis. ■

Thus, mincegis successfully converges to the correct language if and only if cegis also successfully converges to the correct language. So, there is no increase or decrease in power of synthesis by using the deductive verifier that provides minimal counterexamples.

Remark: The above result (and its analog in Sec. 5.2) also holds in the case of general specifications when CEGIS is replaced by Generalized CEGIS. In particular, if either crafted correctness (q_{ccorr}) or membership queries (q_{mem}) are introduced, then it is easy to show that cegis can simulate mincegis by mimicking each step of mincegis by recovering the same counterexample it used with suitable q_{mem} or q_{ccorr} queries. In this case, cegis can converge to every language that mincegis converges to, and hence identifies the same class of specifications.

5.1.2 Bounded vs. Arbitrary Counterexamples

We next investigate cbcegis and compare its relative synthesis power to cegis. As intuitively expected, cbcegis is strictly less powerful than cegis as summarized in Theorem 5.2 which formalizes the intuition.

Theorem 5.2 *The power of synthesis techniques using bounded counterexamples is less than those using counterexamples, that is,* cbcegis $\subset$ cegis.

Proof Since bounded counterexample is also a counterexample, we can easily simulate a bounded verifier CBCHECK using a CHECK by ignoring counterexamples from CHECK if they are larger than a specified bound B which is a fixed parameter and can be stored in the finite memory of the inductive learner. Thus, cbcegis $\subseteq$ cegis.

We now describe a language class for which the corresponding languages cannot be identified using bounded counterexamples.

Language Family 1 *: $\mathscr{L}_{notcb} = \{L_i | i > B \text{ and } L_i = \{n | n \in \mathbb{N} \wedge n > i\}\}$ where B is a constant bound.*

We provide this by contradiction. Let us assume that there is a $T_{\texttt{cbcegis}}$ that can identify languages in $\mathscr{L}_{notcb}$. Let the verifier used by $T_{\texttt{cbcegis}}$ be CBCHECK and B' be the constant bound on the counterexamples produced by CBCHECK. Let us consider the languages $\mathscr{L}_{notcbfail} = \{L_j | L_j \in \mathscr{L}_{notcb} \wedge j > B'\} \subseteq \mathscr{L}_{notcb}$, the set of counterexamples that can be produced by CBCHECK is the same for these languages (that is, $\{n | n \in \mathbb{N} \wedge n \leq B'\}$) since the counterexamples produced by CBCHECK cannot be larger than B'. Thus, a synthesis engine $T_{\texttt{cbcegis}}$ cannot distinguish between languages in $\mathscr{L}_{notcbfail}$ which is a contradiction. Thus, $T_{\texttt{cbcegis}}$ cannot identify all languages in $\mathscr{L}_{notcb}$. $T_{\texttt{cegis}}$ can identify all languages in $\mathscr{L}_{notcb}$ using a simple learner which proposes L_i as the hypothesis language if i is the smallest positive example seen so far. So, cbcegis $\subset$ cegis. ■

We next analyze pbcegis, and show that it is not equivalent to cegis or contained in it. So, replacing a deductive verification engine which returns arbitrary counterexamples with a verification engine which returns counterexamples bounded by history of positive examples has impact on the power of the synthesis technique. But this does not strictly increase the power of synthesis. Instead, the use of positive history bounded counterexamples allows languages from new classes to be identified but at the same time, language from some language classes which could be identified by cegis can no longer be identified using positive bounded counterexamples. The main result regarding the power of synthesis techniques using positive bounded counterexamples is summarized in Theorem 5.3.

Theorem 5.3 *The power of synthesis techniques using arbitrary counterexamples and those using positive bounded counterexamples are not equivalent, and none is more powerful than the other.* $\texttt{pbcegis} \neq \texttt{cegis}$. *In fact,* $\texttt{pbcegis} \not\subseteq \texttt{cegis}$ *and* $\texttt{cegis} \not\subseteq \texttt{pbcegis}$.

We prove this using the following two lemmas. The first lemma 5.4 shows that there is a family of languages from which a language can be identified by `cegis` but, this cannot be done by `pbcegis`. The second lemma 5.5 shows that there is another family of languages from which a language can be identified by `pbcegis` but not by `cegis`.

Lemma 5.4 *There is a family of languages $\mathscr{L}$ such that* `pbcegis` *cannot identify every language L in $\mathscr{L}$ but* `cegis` *can do so, that is,* $\texttt{cegis} \not\subseteq \texttt{pbcegis}$.

Proof Now, consider the language family 2 formed by upper bounding the elements by some fixed constant. Let the target language L (for which we want to identify L_i. In rest of the proof, we also refer to this family as $\mathscr{L}$ for brevity.

Language Family 2 $\mathscr{L}_{notpb} = \{L_i | i \in \mathbb{N}\}$ *such that* $L_i = \{n | n \in \mathbb{N} \wedge n \leq i\}$.

If we obtain a transcript $\tau[j]$ at any point in synthesis using positive bounded counterexamples, then for any intermediate language L_j proposed by $T_{\texttt{pbcegis}}$, $\texttt{PBCHECK}_L$ would always return $\perp$ since all the counterexamples would be larger than any element in $\tau[j]$. This is the consequence of the chosen languages in which all counterexamples to the language are larger than any positive example of the language. So, $T_{\texttt{pbcegis}}$ cannot identify the target language L.

But we can easily design a synthesis engine $T_{\texttt{cegis}}$ using arbitrary counterexamples that can synthesize P corresponding to the target language L. The algorithm starts with L_0 as its initial guess. If there is no counterexample, the algorithm next guess is L_1. In each iteration j, the algorithm guesses L_{j+1} as long as there are no counterexamples. When a counterexample is returned by $\texttt{CHECK}_L$ on the guess L_{j+1}, the algorithm stops and reports the previous guess L_j as the correct language.

Since the elements in each language L_i is bounded by some fixed constant i, the above synthesis procedure $T_{\texttt{cegis}}$ is guaranteed to terminate after i iterations when identifying any language $L_i \in \mathscr{L}$. Further, $\texttt{CHECK}_L$ did not return any counterexample up to iteration $j-1$ and so, $L_j \subseteq L_i$. And in the next iteration, a counterexample was generated. So, $L_{j+1} \not\subseteq L_i$. Since, the languages in $\mathscr{L}$ form a monotonic chain $L_0 \subset L_1 \ldots$. So, $L_j = L_i$. In fact, $j = i$ and in the i-th iteration, the language L_i is correctly identified by $T_{\texttt{cegis}}$. Thus, $\texttt{cegis} \not\subseteq \texttt{pbcegis}$. ■

This shows that `cegis` can be used to identify languages when `pbcegis` will fail. Putting a restriction on the verifier to only produce counterexamples which are bounded by the positive examples seen so far does not strictly increase the power of synthesis.

We now show that this restriction enables identification of languages which cannot be identified by `cegis`.

In the proof below, we construct a language which is not distinguishable using arbitrary counterexamples and instead, it relies on the verifier keeping a record of the largest positive example seen so far and restricting counterexamples to those below the largest positive example.

Lemma 5.5 *There is a family of languages $\mathscr{L}$ such that,* `cegis` *cannot identify a language L in $\mathscr{L}$ but* `pbcegis` *can identify L, that is,* $\texttt{pbcegis} \not\subseteq \texttt{cegis}$.

Proof Consider the language

$$L^{32} = \{3^j.2^i | j \in \{0,1\}, i \in \mathbb{N}\}$$

where $3^j.2^i$ is a natural number obtained by taking the product of 3 raised to the power of j and 2 raised to the power of i. L^{32} is a set of these natural numbers. We now construct a family of languages which are finite subsets of L^{32} and have at least one member of the form 3.2^i, that is,

$$\mathscr{L}^{32} = \{L_i^{32} | i \in \mathbb{N}, L_i^{32} \subset L^{32}, L_i^{32} \text{ is finite and } \exists k \; s.t. \; 3.2^k \in L_i^{32}\}$$

We now consider the language

$$L^2 = \{2^i | i \in \mathbb{N}\}$$

Now, let $\mathscr{L}^2$ be the family of languages such that the smallest element member in the language is the same as the index of the language, that is,

$$\mathscr{L}^2 = \{L_i^2 | i \in \mathbb{N}, L_i^2 \subseteq L^2, L_i^2 \text{ is infinite and } \min(L_i^2) = 2^i\}$$

Now, we consider the following family of languages below.

Language Family 3

$$\mathscr{L}_{pb} = \mathscr{L}^{32} \cup \mathscr{L}^2$$

We refer to this language as $\mathscr{L}$ in rest of the proof for brevity. We show that there is a language L in $\mathscr{L}$ such that the language L cannot be identified by `cegis` but `pbcegis` can identify any language in $\mathscr{L}$.

The key intuition is as follows. If the examples seen by synthesis algorithm till some iteration i are all of the form 2^j, then any synthesis technique cannot differentiate whether the language belongs to $\mathscr{L}^{32}$ or $\mathscr{L}^2$. If the language belongs to $\mathscr{L}^{32}$, the synthesis engine would eventually obtain an example of the form 3.2^j (since each language in $\mathscr{L}^{32}$ has at least one element of this kind and these languages are finite). While the synthesis technique using arbitrary counterexamples cannot recover the previous examples, the techniques with access to the verifier which produces positive bounded counterexamples can recover all the previous examples.

We now specify a $T_{\texttt{pbcegis}}$ which can identify languages in $\mathscr{L}$. The synthesis approach works in two possible steps.

- Until an example 3.2^j is seen by the synthesis engine, let 2^i be the smallest member element seen so far in the transcript, the learner proposes L_i as the language. If the target language $L \in \mathscr{L}^2$, the learner would eventually identify the language since the minimal element will show up in the transcript. If the target language $L \in \mathscr{L}^{32}$, then eventually, an example of the form 3.2^j will be seen since L must have one such member element. And after such an example is seen in the transcript, the synthesis engine moves to second step.
- After an example of the form 3.2^j is seen, the synthesis engine can now be sure that the language belongs to $\mathscr{L}^{32}$ and is finite. Now, the learner can discover all the positive examples seen so far using the following trick. We first discover the upper bound B_p on positive examples seen so far.

  $$B_p = \text{ minimum k such that } \texttt{PBCHECK}_L(\{3^k\}, \tau[n]) \text{ returns } \perp \text{ for } k = 2, 3, \ldots$$

  Recall that $3^k, k = 2, 3, \ldots$ are not in the target language since they are not in any of the languages in the $\mathscr{L}$ to which the target language belongs. $\texttt{PBCHECK}_L$ will return the only element 3^k in the proposed candidate language as a counterexample as long as there is some positive example 2^i seen previously such that $2^i \geq 3^k$. So, 3^{B_p} is the upper bound on all the positive examples seen so far. The learner can now construct singleton languages $\{2^j\}$ for $j = 0, 1, \ldots l$ such that $2^l < 3^{B_p}$. If a counterexample is returned by $\texttt{PBCHECK}_L(\{2^i\}, \tau[n])$ then 2^i is not in the target language. If no counterexample is

returned, then 2^i is in the target language. This allows the synthesis engine to recover all the positive examples seen previously in finite steps. As we recover the positive examples, we run a Gold style algorithm for identifying finite languages [28] to converge to the correct language. Thus, the learner would identify the correct language using finite memory.

We now prove that cegis does not identify this family of languages. Let us assume that $\mathscr{L} \in$ cegis. So, there is a synthesis engine $T_{\texttt{cegis}}$ which can identify all languages in $\mathscr{L}$. So, $T_{\texttt{cegis}}$ must converge to any language $L_1 \in \mathscr{L}^2$ after some finite transcript τ_s. Let us consider an extension $\tau_s 2^m$ of τ_s such that $2^m \in L_1$ and $2^m \notin \texttt{SAMPLE}(\tau_s)$. Such an element 2^m exists since τ_s is a finite transcript and L_1 is an infinite language. Since the learner converges to L_1 starting from the initial language L_0 after consuming τ_s, $\texttt{learn}(L_0, \tau_s 2^m, \texttt{cex}') = \texttt{learn}(L_0, \tau_s, \texttt{cex})$.

Let us consider two transcripts $\tau_s 2^m (3.2^p)\perp^\omega$ and $\tau_s (3.2^p)\perp^\omega$ where $\perp^\omega$ denotes repeating $\perp$ infinitely in the rest of the transcript. We know that $\texttt{learn}(L_0, \tau_s 2^m, \texttt{cex}') = \texttt{learn}(L_0, \tau_s, \texttt{cex}) = L_1$ and thus, $\texttt{learn}(\tau_s 2^m (3.2^p)\perp^\omega, \texttt{cex}') = \texttt{learn}(\tau_s (3.2^p)\perp^\omega, \texttt{cex}) = \texttt{learn}(L_1, (3.2^p)\perp^\omega, \texttt{cex}'')$. So, the synthesis engine would behave exactly the same for both transcripts, and if it converges to a language L_2 on one transcript, it would converge to the same language on the other transcript. But the two transcripts are clearly from two different languages in $\mathscr{L}^{32}$. One of the transcripts corresponds to the finite language $\texttt{SAMPLE}(\tau_s) \cup \{3.2^p\}$ and the other corresponds to $\texttt{SAMPLE}(\tau_s) \cup \{2^m, 3.2^p\}$. This is a contradiction and hence, there is no synthesis engine using arbitrary counterexamples $T_{\texttt{cegis}}$ that can identify all languages in $\mathscr{L}$.

■

5.1.3 Different Flavors of Bounded Counterexamples

Finally, we compare pbcegis and cbcegis and show that they are not contained in each other.

Theorem 5.6 *The power of synthesis techniques using bounded counterexamples is neither less nor more than the techniques using positive bounded counterexamples, that is,* cbcegis $\not\subseteq$ pbcegis *and* pbcegis $\not\subseteq$ cbcegis.

Proof We consider two languages considered in previous proofs and show that the languages corresponding to one of them can only be identified by pbcegis while the languages corresponding to the other can only be identified by cbcegis.

Consider the language family 1 ($\mathscr{L}_{notcb}$) formed by lower bounding the elements by some fixed constant, that is, $\mathscr{L}_{notcb} = \{L_i | i > B \text{ and } L_i = \{n | n \in \mathbb{N} \wedge n > i\}\}$ where B is a fixed integer constant. We have proved in Theorem 5.2 that a synthesis engine $T_{\texttt{cbcegis}}$ cannot identify all languages in $\mathscr{L}_{notcb}$. On the other hand, any counterexample is smaller than all positive examples in any language in $\mathscr{L}_{notcb}$. So, a verifier producing positive bounded counterexample behaves similar to an arbitrary counterexample verifier since any positive example is larger than all negative examples. Thus, $T_{\texttt{cegis}}$ can identify languages in this language class. So, pbcegis $\not\subseteq$ cbcegis.

Now, consider the family of languages consisting of these, that is,

Language Family 4 $\mathscr{L}_{cbnotpb} = \{L_i | i < B\}$ *where* $L_i = \{n | n \in \mathbb{N} \wedge n \leq i\}$

This is a slight variant of the language class considered in proving $T_{\texttt{cegis}}$ to be more powerful than $T_{\texttt{pbcegis}}$ where we have restricted the class of languages to be a finite set. As stated earlier, PBCHECK does not produce any counterexample for these languages since all positive examples are smaller than any counterexample. But CBCHECK can be used to identify languages in this class by selecting the bound of the counterexamples to be B. Since, the counterexamples are at most of size B for these languages, a bounded counterexample verifier behaves exactly like an arbitrary counterexample producing verifier. Thus, cbcegis $\not\subseteq$ pbcegis. ■

5.2 Infinite Memory Inductive Synthesis

We now consider the case where the inductive learning engine has infinite unbounded memory. This case is simpler than the one considered earlier with finite memory bound on the inductive learning engine and most of the results presented here follow from the results proved for the finite memory case. For brevity of space, we only give proof sketches highlighting the difference from the finite memory case.

1. The proof of Theorem 5.1 works even when we replace the inductive learning engine using finite memory with the one using infinite memory. Further, the minimal counterexample can still be used as an aribitrary counterexample. And so, MINCEGIS = CEGIS.

2. Next, we show that CBCEGIS $\subseteq$ CEGIS. Consider an arbitrary but fixed constant B. For this B, consider all verifiers CBCHECK that only produce counterexamples bounded by B. We wish to argue that any infinite memory learner LEARN that can converge to a target language L_c using any CBCHECK can also do so using CHECK. The basic idea is as follows: since LEARN has infinite memory, it can make extra queries to CHECK to obtain counterexamples bounded by B and learns only from those. Suppose at some step it received a counterexample x bigger than B for candidate language L. Then LEARN constructs a new candidate language L' that excludes x but otherwise agrees with L.[5] It then queries CHECK with this new candidate L', and iterates the process until a counterexample less than B is received (which must happen if such a counterexample exists). LEARN uses its infinite-size memory to construct candidate languages that keep track of a potentially unbounded number of counterexamples bigger than B. Thus, LEARN uses this procedure to convert any CHECK into some CBCHECK. Since CBCEGIS comprises all language families learnable by LEARN given *any* CBCHECK, these language families are also learnable by LEARN using CHECK. Therefore, CBCEGIS $\subseteq$ CEGIS.

3. We now sketch the proof for PBCEGIS $\subseteq$ CEGIS. The argument is similar to the previous case. Since the learner has infinite memory, it can store all the positive examples seen so far. Moreover, similar to the case of CBCEGIS, it can construct a stream of candidate languages to query CHECK so as to obtain positive history bounded counterexamples, as follows. It queries CHECK to obtain an arbitrary counterexample. If this is smaller than the largest positive example in stored positive examples, then the learner uses this example for proposing the next hypothesis language. If this counterexample is larger that the largest positive example, it constructs a new candidate language by excluding this counterexample from the previous candidate language, and again queries CHECK to obtain a new counterexample. This continues until the learner can get a positive history bounded counterexample or there is no such counterexample. Thus, the learner now uses only positive history bounded counterexamples, and hence, $T_{\texttt{CEGIS}}$ can identify any language that $T_{\texttt{PBCEGIS}}$ can identify.

We now present three languages used previously in proofs for inductive learning engines using finite memory, and show how these languages allow us to distinguish relative power of synthesis engines.

1. Consider the language family 1: $\mathscr{L}_{notcb} = \{L_i | i > B \text{ and } L_i = \{n | n \in \mathbb{N} \wedge n > i\}\}$ where B is a constant bound. The argument in Theorem 5.2 also holds for the infinite memory synthesis engines, and so, $\mathscr{L}_{notcb} \in \overline{\texttt{CBCEGIS}} \cap \texttt{CEGIS}$.

   Further, a positive history bounded verifier will always return a counterexample if one exists since all counterexamples are smaller than any positive example in the language. Thus, $T_{\texttt{PBCEGIS}}$ can also identify languages in $\mathscr{L}_{notcb}$. Thus, $\mathscr{L}_{notcb} \in \overline{\texttt{CBCEGIS}} \cap \texttt{PBCEGIS}$.

[5] We can do this as we have a finite representation of L (e.g., in the form of its characteristic function) and can modify this to initially check if the input is x, and if so, to report that this is not in the modified language.

2. Consider the language family 2: $\mathscr{L}_{notpb} = \{L_i | i \in \mathbb{N}\}$ where

$$L_i = \{n | n \in \mathbb{N} \wedge n \leq i\}$$

As argued in the proof of Theorem 5.3, the verifier producing positive bounded counterexamples will not report any counterexample for any of the languages in $\mathscr{L}_{notpb}$ because all counterexamples are larger than any positive example. So, languages in this family cannot be identified by $T_{\texttt{PBCEGIS}}$ but these can be identified using $T_{\texttt{CEGIS}}$. So, $\mathscr{L}_{notpb} \in \overline{\texttt{PBCEGIS}} \cap \texttt{CEGIS}$.

3. Consider the finite language family 4: $\mathscr{L}_{cbnotpb} = \{L_i | i < B\}$ where

$$L_i = \{n | n \in \mathbb{N} \wedge n \leq i\}$$

As argued in proof of Theorem 5.6, the verifier PBCHECK does not produce any counterexample for these languages since all positive examples are smaller than any counterexample. But CBCHECK can be used to identify languages in this class by selecting the bound to be B. Since, the counterexamples are at most of size B for these languages, a bounded counterexample verifier behaves exactly like an arbitrary counterexample producing verifier. Thus, $\mathscr{L}_{cbnotpb} \in \overline{\texttt{PBCEGIS}} \cap \texttt{CBCEGIS}$.

We now summarize the results described in this section below. For finite memory learners, cbcegis $\subset$ mincegis $=$ cegis, pbcegis and cegis are not comparabale, that is, pbcegis $\not\subseteq$ cegis and pbcegis $\not\supseteq$ cegis. cbcegis and pbcegis are also not comparable. In case of infinite memory learners, CBCEGIS $\subset$ MINCEGIS $=$ CEGIS, and PBCEGIS $\subset$ CEGIS $=$ MINCEGIS. CBCEGIS and PBCEGIS are again not comparable. The results are summarized in Figure 2.

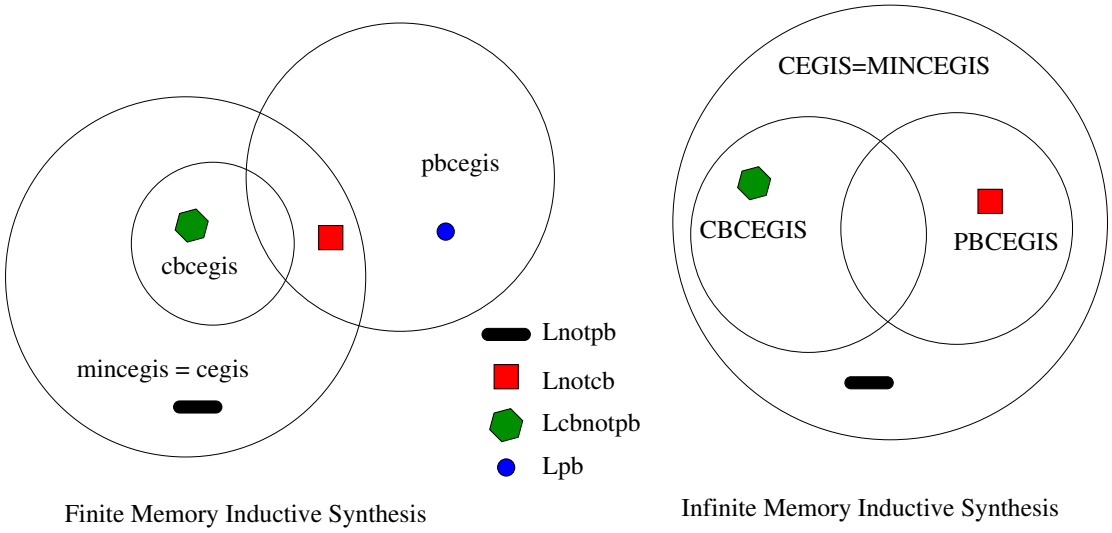

Figure 2: Summary of Results on Decidability of Synthesis for Infinite Language Classes

6 Analysis of OGIS for Finite Language Classes

We now discuss the case when the class of candidate programs (languages) has finite cardinality. As in Sec. 4, rather than referring to programs we will refer to synthesizing the languages identified by those programs. If the language class is finite then there exists a terminating OGIS procedure, e.g., one that

simply enumerates languages from this class until one satisfying the specification Φ is obtained. Moreover, any implementation of OGIS which uses an oracle that provides new (positive/negative) examples in every iteration ruling out at least one candidate language will terminate with the correct language. The counterexample guided inductive synthesis approach [55] for bitvector sketches and oracle guided inductive synthesis using distinguishing inputs [30] for programs composed of a finite library of components are examples of OGIS synthesis techniques applied to finite language classes. We analyze the complexity of synthesis for finite language classes and discuss its relation to the notion of *teaching dimension* from the concept learning literature [20]. This connection between synthesis of languages from finite classes and teaching of concepts was first discussed in [30]. Here we establish that the size of the smallest set of examples for language (program) synthesis is bounded below by the teaching dimension of the concept class corresponding to the class of languages.

6.1 NP-hardness

We measure efficiency of an OGIS synthesis engine using the notion of *sample complexity* mentioned in Sec. 2 — the number of queries (and responses) needed to correctly identify a language. In order to analyze sample complexity, we need to fix the nature of queries to the oracle. We focus on queries to which the oracle provides an example or counterexample in response. We show that finding the minimal set of examples to be provided by the oracle such that the synthesis engine converges to the correct language is NP-hard.

Theorem 6.1 *Solving the formal inductive synthesis problem $\langle \mathscr{C}, \mathbf{E}, \Phi, \mathscr{O} \rangle$ for a finite $\mathscr{C}$ and finite $\mathbf{E}$ with the minimum number of queries is NP-hard for any oracle interface $\mathscr{O}$ comprising the correctness query q_{corr} (and possibly qposwit and qnegwit).*

Proof We prove NP-hardness through reduction from the minimum set cover problem. Consider the minimum set cover problem with k sets $S_1, S_2, \ldots, S_k$ and a universe comprising m elements $x_1, x_2, \ldots, x_m$ which needs to be covered using the sets. We reduce it to a formal inductive synthesis problem $\langle \mathscr{C}, \mathbf{E}, \Phi, \mathscr{O} \rangle$ where $\mathscr{C} = \{L_1, L_2, \ldots, L_m, L_{m+1}\}$ is a set of $m+1$ languages, $\mathbf{E} = \{e_1, e_2, \ldots, e_k\}$ is the domain comprising k examples over which the languages are defined and $\Phi = \{L_{m+1}\}$ is the specification. Intuitively, the m languages $L_1, \ldots, L_m$ are associated to the m elements in the set cover problem. The k examples correspond to the k sets. The sets $L_1, L_2, \ldots, L_{m+1}$ are constructed as follows: For all $1 \leq i \leq k$ and $1 \leq j \leq m$, example e_i belongs to the symmetric difference of L_j and L_{m+1} if and only if the set S_i contains element x_j. We can do this, for instance, by including e_i in L_j but not in L_{m+1}.

Consider the operation of an OGIS procedure implementing an $\mathscr{O}$ containing q_{corr}. Every unsuccessful correctness query returns a counterexample which is an element of $\mathbf{E}$ in the symmetric difference of the proposed L_j and L_{m+1}. Let $e_{i_1}, e_{i_2}, \ldots, e_{i_n}$ be the smallest set of counterexamples that uniquely identifies the correct language L_{m+1}. So, for all $1 \leq j \leq m$, there exists some i_l such that either $e_{i_l} \in L_j$ or $e_{i_l} \in L_{m+1}$ but not both. And so, for all $1 \leq j \leq m$, there exists some i_l such that $x_j \in S_{i_l}$ where $i_l \in \{i_1, i_2, \ldots, i_n\}$. Moreover, dropping i_l results in some x_j not being covered (the corresponding L_j is not distinguished from L_{m+1}). Thus, $S_{i_1}, S_{i_2}, \ldots, S_{i_n}$ is a solution to the minimum set cover problem which is known to be NP-complete. Similarly, it is easy to see that any solution to the minimum set cover problem also yields a minimum counterexample set.

We can therefore conclude that solving the formal inductive synthesis problem $\langle \mathscr{C}, \mathbf{E}, \Phi, \mathscr{O} \rangle$ with the minimum number of queries is NP-hard.

■

We note that this proof applies to any FIS problem with an oracle interface $\mathscr{O}$ containing the correctness query q_{corr}. Moreover, this proof can be easily extended to other oracle interfaces as well,

such as the version of the distinguishing input method that does not use the correctness query, with $\mathscr{O} = \{q^{+}_{\text{wit}}, q_{\text{diff}}, q_{\text{mem}}\}$. In this latter case, the combined use of q_{diff} and q_{mem} yields the desired mapping.

6.2 Relation to Teaching Dimension

Goldman et al. [20, 21] proposed *teaching dimension* as a measure to study computational complexity of learning. They consider a teaching model in which a helpful teacher selects the examples of the concept and provides it to the learner. Informally, the teaching dimension of a concept class is the minimum number of examples that a teacher must reveal to uniquely identify any target concept chosen from the class.

For a domain $\mathbf{E}$ and concept class $\mathscr{C}$, a concept $c \in \mathscr{C}$ is a set of examples from $\mathbf{E}$. So, $\mathscr{C} \subseteq 2^{\mathbf{E}}$. In the learning model proposed by Goldman et al. [20, 21], the basic goal of the teacher is to help the learner identify the target concept $c^* \in \mathscr{C}$ by providing an example sequence from $\mathbf{E}$. We now formally define the teaching dimension of a concept class.

Definition 6.1 *(adapted from [20]) An example sequence is a sequence of labeled examples from* $\mathbf{E}$*, where the labels are given by some underlying specification. For concept class* $\mathscr{C}$ *and target concept* $c \in \mathscr{C}$*, we say* T *is a teaching sequence for* c *(in* $\mathscr{C}$*) if* T *is an example sequence that uniquely identifies* c *in* $\mathscr{C}$ *- that is,* c *is the only concept in* $\mathscr{C}$ *consistent with* T*. Let* $T(c)$ *denote the set of all teaching sequences for* c*. Teaching dimension* $TD(\mathscr{C})$ *of the concept class is defined as follows:*

$$TD(\mathscr{C}) = \max_{c \in \mathscr{C}} \left(\min_{\tau \in T(c)} |\tau| \right)$$

Consider an FIS problem where the specification is complete, i.e., $\Phi = \{L_c\}$. Consider an instance of OGIS using any combination of witness, equivalence, subsumption, or distinguishing input queries. Each of these queries, if it does not terminate the OGIS loop, returns a new example for the learner. Thus, the number of iterations of the OGIS loop, its sample complexity, is the number of examples needed by the learner to identify a correct language. Suppose the minimum such number of examples, for any specification (target language $L_c \in \mathscr{C}$), is $M_{\mathsf{OGIS}}(\mathscr{C})$. Then, the following theorem must hold.

Theorem 6.2 $M_{\mathsf{OGIS}}(\mathscr{C}) \geq TD(\mathscr{C})$

The theorem can be obtained by a straightforward proof by contradiction: if $M_{\mathsf{OGIS}}(\mathscr{C}) < TD(\mathscr{C})$, then for each target concept to be learned, there is a shorter teaching sequence than $TD(\mathscr{C})$, viz., the one used by the OGIS instance for that target, contradicting the definition of teaching dimension.

Now, given that the teaching dimension is a lower bound on the sample complexity of OGIS, it is natural to ask how large $TD(\mathscr{C})$ can grow in practice. This is still a largely open question for general language classes. However, results from machine learning theory can help shed more light on this question. One of these results relates the teaching dimension to a second metric for measuring complexity of learning, namely the *Vapnik-Chervonenkis* (VC) dimension [60]. We define this below.

Definition 6.2 *[60] Let* $\mathbf{E}$ *be the domain of examples and* c *be a concept from the class* $\mathscr{C}$*. A finite set* $\mathbf{E}' \subseteq \mathbf{E}$ *is shattered by* $\mathscr{C}$ *if* $\{c \cap \mathbf{E}' | c \in \mathscr{C}\} = 2^{\mathbf{E}'}$*. In other words,* $\mathbf{E}' \subseteq \mathbf{E}$ *is shattered by* $\mathscr{C}$ *if for each subset* $\mathbf{E}'' \subseteq \mathbf{E}'$*, there is a concept* $c \in \mathscr{C}$ *which contains all of* $\mathbf{E}''$*, but none of the instances in* $\mathbf{E}' - \mathbf{E}''$*. The Vapnik-Chervonenskis (VC) dimension is defined to be smallest* d *for which no set of* $d+1$ *examples is shattered by* $\mathscr{C}$*.*

Blumer et al. [12] have shown that the VC dimension of a concept class characterizes the number of examples required for learning any concept in the class under the distribution-free or probably approximately correct (PAC) model of Valiant [59]. The differences between teaching dimension and

Vapnik-Chervonenkis dimension are discussed at length by Goldman and Kearns [20]. The following theorems from [20] provides lower and upper bound on the teaching dimension of a finite concept class in terms of the size of the concept class and its VC-dimension.

Theorem 6.3 *[20] The teaching dimension $TD(\mathscr{C})$ of any concept class $\mathscr{C}$ satisfies the following upper and lower bounds:*

$$VC(\mathscr{C})/\log(|\mathscr{C}|) \leq TD(\mathscr{C}) \leq |\mathscr{C}| - 1$$

where $VC(\mathscr{C})$ is the VC dimension of the concept class $\mathscr{C}$ and $|\mathscr{C}|$ denotes the number of concepts in the concept class.

Moreover, Goldman and Kearns [20] exhibit a concept class for which the upper bound is tight. This indicates that without restrictions on the concept class, one may not be able to prove very strong bounds on the sample complexity of OGIS.

To summarize, we have shown that solving the formal inductive synthesis problem for finite domains and finite concept classes with the minimum number of queries is NP-hard. Further, we showed that the combinatorial measure of teaching dimension captures the smallest number of examples required to identify the correct language.

7 Conclusion

We presented a theoretical framework and analysis of formal inductive synthesis by formalizing the notion of *oracle-guided inductive synthesis* (OGIS). We illustrated how OGIS generalizes instances of concept learning in machine learning as well as synthesis techniques developed using formal methods. We focus on counterexample-guided inductive synthesis (CEGIS) which is an OGIS implementations that uses the verification engine as the oracle. We presented different variations of `cegis` motivated by practice, and showed that their synthesis power can be different, especially when the learning engine can only store a bounded number of examples. There are several directions for future work. We discuss some open problems below that would further improve the theoretical understanding of formal inductive synthesis.

- Teaching dimension of concept classes such as decision trees and axis parallel rectangles have been well-studied in literature. But teaching dimension of formal concept classes such as programs in the *while* [64] language with only linear arithmetic over integers is not known. Finding teaching dimensions for these classes would help in establishing bounds on the number of examples needed for synthesizing programs from these classes.
- We investigated the difference in synthesis power when the learning engine has finite memory vs when the learning engine has infinite memory. Another important question to consider is how the power of the synthesis engine changes when we restrict the time complexity of learning engine such as the learning engines which take time polynomial in the number of examples.
- We have not analyzed the impact of different learning strategies that may traverse the space of possible programs (languages) in various ways. This is also an interesting avenue for future work.

In summary, our paper is a first step towards a theory of formal inductive synthesis, and much remains to be done to improve our understanding of this emerging area with several practical applications.

References

[1] Rajeev Alur, Rastislav Bodik, Garvit Juniwal, Milo M. K. Martin, Mukund Raghothaman, Sanjit A. Seshia, Rishabh Singh, Armando Solar-Lezama, Emina Torlak & Abhishek Udupa (2013): *Syntax-Guided Synthe-*

sis. In: *Proceedings of the IEEE International Conference on Formal Methods in Computer-Aided Design (FMCAD)*.

[2] Dana Angluin (1980): *Inductive Inference of Formal Languages from Positive Data*. Information and Control 45, pp. 117–135, doi:10.1016/S0019-9958(80)90285-5.

[3] Dana Angluin (1987): *Learning regular sets from queries and counterexamples*. Information and computation 75(2), pp. 87–106.

[4] Dana Angluin (1988): *Queries and concept learning*. Machine Learning 2(4), pp. 319–342, doi:10.1023/A:1022821128753.

[5] Dana Angluin (2004): *Queries revisited*. Theoretical Computer Science 313(2), pp. 175 – 194, doi:http://dx.doi.org/10.1016/j.tcs.2003.11.004. Available at `http://www.sciencedirect.com/science/article/pii/S030439750300608X`. Algorithmic Learning Theory.

[6] Dana Angluin & Carl H. Smith (1983): *Inductive Inference: Theory and Methods*. ACM Computing Surveys 15, pp. 237–269.

[7] Mohamed Faouzi Atig, Ahmed Bouajjani & Shaz Qadeer (2011): *Context-Bounded Analysis For Concurrent Programs With Dynamic Creation of Threads*. Logical Methods in Computer Science 7(4), doi:10.2168/LMCS-7(4:4)2011. Available at `http://dx.doi.org/10.2168/LMCS-7(4:4)2011`.

[8] Clark Barrett, Roberto Sebastiani, Sanjit A. Seshia & Cesare Tinelli (2009): *Satisfiability Modulo Theories*. In Armin Biere, Hans van Maaren & Toby Walsh, editors: Handbook of Satisfiability, chapter 8, 4, IOS Press.

[9] Yoshua Bengio, Ian J. Goodfellow & Aaron Courville (2015): *Deep Learning*. Available at `http://www.iro.umontreal.ca/~bengioy/dlbook`. Book in preparation for MIT Press.

[10] Armin Biere (2009): *Bounded Model Checking*. In: Handbook of Satisfiability, pp. 457–481, doi:10.3233/978-1-58603-929-5-457. Available at `http://dx.doi.org/10.3233/978-1-58603-929-5-457`.

[11] L. Blum & M. Blum (1975): *Toward a mathematical theory of inductive inference*. Information and Control 28(2), pp. 125–155, doi:10.1016/s0019-9958(75)90261-2.

[12] Anselm Blumer, A. Ehrenfeucht, David Haussler & Manfred K. Warmuth (1989): *Learnability and the Vapnik-Chervonenkis Dimension*. J. ACM 36(4), pp. 929–965, doi:10.1145/76359.76371. Available at `http://doi.acm.org/10.1145/76359.76371`.

[13] Randal E. Bryant (1986): *Graph-based algorithms for Boolean function manipulation*. IEEE Transactions on Computers C-35(8), pp. 677–691.

[14] Yibin Chen, Sean Safarpour, Joo Marques-Silva & Andreas G. Veneris (2010): *Automated Design Debugging With Maximum Satisfiability*. IEEE Trans. on CAD of Integrated Circuits and Systems 29(11), pp. 1804–1817, doi:10.1109/TCAD.2010.2061270.

[15] Edmund M. Clarke & E. Allen Emerson (1981): *Design and Synthesis of Synchronization Skeletons Using Branching-Time Temporal Logic*. In: Logic of Programs, pp. 52–71.

[16] Edmund M. Clarke, Orna Grumberg & Doron A. Peled (2000): *Model Checking*. MIT Press.

[17] Michael R. Clarkson & Fred B. Schneider (2010): *Hyperproperties*. Journal of Computer Security 18(6), pp. 1157–1210.

[18] Dimitra Giannakopoulou and Corina S. Pasareanu, eds. (2008): *Special issue on learning techniques for compositional reasoning*. Formal Methods in System Design 32(3), pp. 173–174.

[19] E. Mark Gold (1967): *Language identification in the limit*. Information and Control 10(5), pp. 447–474, doi:10.1016/S0019-9958(67)91165-5.

[20] Sally A. Goldman & Michael J. Kearns (1992): *On the Complexity of Teaching*. Journal of Computer and System Sciences 50, pp. 303–314. Available at `http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.55.3652`.

[21] Sally A. Goldman, Ronald L. Rivest & Robert E. Schapire (1993): *Learning Binary Relations and Total Orders*. SIAM J. Comput. 22(5), pp. 1006–1034, doi:10.1137/0222062. Available at `http://dx.doi.org/10.1137/0222062`.

[22] M. J. C. Gordon & T. F. Melham (1993): *Introduction to HOL: A Theorem Proving Environment for Higher-Order Logic*. Cambridge University Press.

[23] Sergey Grebenshchikov, Nuno P Lopes, Corneliu Popeea & Andrey Rybalchenko (2012): *Synthesizing software verifiers from proof rules*. In: *ACM SIGPLAN Notices*, 47, ACM, pp. 405–416.

[24] Sumit Gulwani, Susmit Jha, Ashish Tiwari & Ramarathnam Venkatesan (2011): *Synthesis of loop-free programs*. In: *PLDI*, pp. 62–73, doi:10.1145/1993498.1993506.

[25] Tibor Hegedűs (1994): *Geometrical Concept Learning and Convex Polytopes*. In: *Proceedings of the Seventh Annual Conference on Computational Learning Theory*, COLT '94, ACM, New York, NY, USA, pp. 228–236, doi:10.1145/180139.181124. Available at `http://doi.acm.org/10.1145/180139.181124`.

[26] Jeffrey C Jackson (1997): *An Efficient Membership-Query Algorithm for Learning {DNF} with Respect to the Uniform Distribution*. *Journal of Computer and System Sciences* 55(3), pp. 414 – 440, doi:http://dx.doi.org/10.1006/jcss.1997.1533. Available at `http://www.sciencedirect.com/science/article/pii/S0022000097915336`.

[27] Sanjay Jain (1999): *Systems that learn: an introduction to learning theory*. MIT press.

[28] Sanjay Jain & Efim Kinber (2007): *Iterative learning from positive data and negative counterexamples*. *Information and Computation* 205(12), pp. 1777 – 1805, doi:http://dx.doi.org/10.1016/j.ic.2007.09.001.

[29] Klaus P. Jantke & Hans-Rainer Beick (1981): *Combining Postulates of Naturalness in Inductive Inference*. *Elektronische Informationsverarbeitung und Kybernetik* 17(8/9), pp. 465–484.

[30] Susmit Jha, Sumit Gulwani, Sanjit A. Seshia & Ashish Tiwari (2010): *Oracle-guided Component-based Program Synthesis*. ICSE '10, ACM, New York, NY, USA, pp. 215–224, doi:10.1145/1806799.1806833.

[31] Susmit Jha, Sumit Gulwani, Sanjit A. Seshia & Ashish Tiwari (2010): *Synthesizing Switching Logic for Safety and Dwell-Time Requirements*. In: *Proceedings of the International Conference on Cyber-Physical Systems (ICCPS)*, pp. 22–31.

[32] Susmit Jha & Sanjit A. Seshia (2014): *Are There Good Mistakes? A Theoretical Analysis of CEGIS*. In: *3rd Workshop on Synthesis (SYNT)*.

[33] Susmit Jha, Sanjit A. Seshia & Ashish Tiwari (2011): *Synthesis of Optimal Switching Logic for Hybrid Systems*. In: *Proceedings of the International Conference on Embedded Software (EMSOFT)*, pp. 107–116.

[34] Susmit Kumar Jha (2011): *Towards Automated System Synthesis Using SCIDUCTION*. Ph.D. thesis, EECS Department, University of California, Berkeley. Available at `http://www.eecs.berkeley.edu/Pubs/TechRpts/2011/EECS-2011-118.html`.

[35] Xiaoqing Jin, Alexandre Donzé, Jyotirmoy Deshmukh & Sanjit A. Seshia (2013): *Mining Requirements from Closed-Loop Control Models*. In: *Proceedings of the International Conference on Hybrid Systems: Computation and Control (HSCC)*.

[36] Matt Kaufmann, Panagiotis Manolios & J. Strother Moore (2000): *Computer-Aided Reasoning: An Approach*. Kluwer Academic Publishers.

[37] Viktor Kuncak, Mikaël Mayer, Ruzica Piskac & Philippe Suter (2012): *Software synthesis procedures*. *Commun. ACM* 55(2), pp. 103–111.

[38] S. Lange (2000): *Algorithmic Learning of Recursive Languages*. Mensch-und-Buch-Verlag.

[39] Steffen Lange, Thomas Zeugmann & Sandra Zilles (2008): *Learning Indexed Families of Recursive Languages from Positive Data: A Survey*. *Theor. Comput. Sci.* 397(1-3), pp. 194–232, doi:10.1016/j.tcs.2008.02.030.

[40] Steffen Lange & Sandra Zilles (2004): *Formal language identification: Query learning vs gold-style learning*. *Information Processing Letters*, p. 2004.

[41] Sharad Malik & Lintao Zhang (2009): *Boolean Satisfiability: From Theoretical Hardness to Practical Success*. *Communications of the ACM (CACM)* 52(8), pp. 76–82. Available at `http://doi.acm.org/10.1145/1536616.1536637`.

[42] Zohar Manna & Richard Waldinger (1980): *A Deductive Approach to Program Synthesis*. *ACM Trans. Program. Lang. Syst.* 2(1), pp. 90–121, doi:10.1145/357084.357090.

[43] Thomas M. Mitchell (1997): *Machine Learning*, first edition. McGraw-Hill, Inc., New York, NY, USA.

[44] Antonio Morgado, Mark Liffiton & Joao Marques-Silva (2013): *MaxSAT-Based MCS Enumeration*. In Armin Biere, Amir Nahir & Tanja Vos, editors: *Hardware and Software: Verification and Testing, Lecture Notes in Computer Science* 7857, Springer Berlin Heidelberg, pp. 86–101, doi:10.1007/978-3-642-39611-3_13.

[45] S. Owre, J. M. Rushby & N. Shankar (1992): *PVS: A Prototype Verification System*. In Deepak Kapur, editor: *11th International Conference on Automated Deduction (CADE), Lecture Notes in Artificial Intelligence* 607, Springer-Verlag, pp. 748–752.

[46] Amir Pnueli & Roni Rosner (1989): *On the Synthesis of a Reactive Module*. In: *ACM Symposium on Principles of Programming Languages (POPL)*, pp. 179–190.

[47] Jean-Pierre Queille & Joseph Sifakis (1982): *Specification and Verification of Concurrent Systems in CESAR*. In: *Symposium on Programming, LNCS* 137, pp. 337–351.

[48] J. R. Quinlan (1986): *Induction of Decision Trees*. *Mach. Learn.* 1(1), pp. 81–106, doi:10.1023/A:1022643204877. Available at `http://dx.doi.org/10.1023/A:1022643204877`.

[49] Hartley Rogers, Jr. (1987): *Theory of Recursive Functions and Effective Computability*. MIT Press, Cambridge, MA, USA.

[50] S. Salzberg, A.L. Delcher, D. Heath & S. Kasif (1995): *Best-case results for nearest-neighbor learning*. *Pattern Analysis and Machine Intelligence, IEEE Transactions on* 17(6), pp. 599–608, doi:10.1109/34.387506.

[51] Sanjit A. Seshia (2012): *Sciduction: Combining Induction, Deduction, and Structure for Verification and Synthesis*. In: *Proceedings of the Design Automation Conference (DAC)*, pp. 356–365.

[52] Ehud Y Shapiro (1982): *Algorithmic Program Debugging*. MIT Press.

[53] Ayumi Shinohara & Satoru Miyano (1990): *Teachability in Computational Learning*. In: *ALT*, pp. 247–255.

[54] Armando Solar-Lezama, Rodric Rabbah, Rastislav Bodík & Kemal Ebcioglu (2005): *Programming by sketching for bit-streaming programs*. In: *PLDI*.

[55] Armando Solar-Lezama, Liviu Tancau, Rastislav Bodk, Sanjit A. Seshia & Vijay A. Saraswat (2006): *Combinatorial sketching for finite programs*. In: *ASPLOS*, pp. 404–415, doi:10.1145/1168857.1168907.

[56] Saurabh Srivastava, Sumit Gulwani & Jeffrey S. Foster (2010): *From program verification to program synthesis*. In: *POPL '10: Proceedings of the 37th annual ACM Symposium on Principles of Programming Languages*, pp. 313–326.

[57] Phillip D. Summers (1977): *A Methodology for LISP Program Construction from Examples*. *J. ACM* 24(1).

[58] Abhishek Udupa, Arun Raghavan, Jyotirmoy V. Deshmukh, Sela Mador-Haim, Milo M.K. Martin & Rajeev Alur (2013): TRANSIT*: Specifying Protocols with Concolic Snippets*. In: *Proceedings of the* 34^{th} *ACM SIGPLAN conference on Programming Language Design and Implementation*, pp. 287–296.

[59] Leslie G. Valiant (1984): *A Theory of the Learnable*. *Communications of the ACM* 27, pp. 1134–1142.

[60] V. N. Vapnik & A. Ya. Chervonenkis (1971): *On the Uniform Convergence of Relative Frequencies of Events to Their Probabilities* 16(2), pp. 264–280. doi:10.1137/1116025.

[61] Sanford Weisberg (2005): *Applied Linear Regression*, third edition. Wiley, Hoboken NJ. Available at `http://www.stat.umn.edu/alr`.

[62] Rolf Wiehagen (1976): *Limit Detection of Recursive Functions by Specific Strategies*. *Electronic Information Processing and Cybernetics* 12(1/2), pp. 93–99.

[63] Rolf Wiehagen (1990): *A Thesis in Inductive Inference*. In Jrgen Dix, Klaus P. Jantke & Peter H. Schmitt, editors: *Nonmonotonic and Inductive Logic, Lecture Notes in Computer Science* 543, Springer, pp. 184–207, doi:10.1007/BFb0023324.

[64] Glynn Winskel (1993): *The Formal Semantics of Programming Languages: An Introduction*. MIT Press, Cambridge, MA, USA.