Are There Good Mistakes? A Theoretical Analysis of CEGIS

Susmit Jha
Strategic CAD Labs, Intel
susmit.jha@intel.com

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

Counterexample-guided inductive synthesis (CEGIS) is used to synthesize programs from a candidate space of programs. The technique is guaranteed to terminate and synthesize the correct program if the space of candidate programs is finite. But the technique may or may not terminate with the correct program if the candidate space of programs is infinite. In this paper, we perform a theoretical analysis of counterexample-guided inductive synthesis technique. We investigate whether the set of candidate spaces for which the correct program can be synthesized using CEGIS depends on the counterexamples used in inductive synthesis, that is, whether there are *good mistakes* which would increase the synthesis power. We investigate whether the use of minimal counterexamples instead of arbitrary counterexamples expands the set of candidate spaces of programs for which inductive synthesis can successfully synthesize a correct program. We consider two kinds of counterexamples: minimal counterexamples and *history bounded* counterexamples. The history bounded counterexample used in any iteration of CEGIS is bounded by the examples used in previous iterations of inductive synthesis. We examine the relative change in power of inductive synthesis in both cases. We show that the synthesis technique using minimal counterexamples MinCEGIS has the same synthesis power as CEGIS but the synthesis technique using history bounded counterexamples HCEGIS has different power than that of CEGIS, but none dominates the other.

1 Introduction

Automatic synthesis of programs has been one of the holy grails of computer science for a long time. It has found many practical applications such as generating optimal code sequences [27, 20], optimizing performance-critical inner loops, generating general-purpose peephole optimizers [3, 4], automating repetitive programming, and filling in low-level details after the higher-level intent has been expressed [33]. A traditional view of program synthesis is that of synthesis from complete specifications. One approach is to give a specification as a formula in a suitable logic [25, 26, 13]. Another is to write the specification as a simpler, but possibly far less efficient program [27, 33, 20]. While these approaches have the advantage of completeness of specification, such specifications are often unavailable, difficult to write, or expensive to check against using automated verification techniques. This has led to proposal of oracle guided synthesis approach [19] in which the complete specification is not available. All these different variants of automated synthesis techniques share some common characteristics. They are iterative inductive synthesis techniques which require some kind of validation engines to validate candidate programs produced at intermediate iterations, and these validation engines identify counterexamples, aka mistakes, which are subsequently used for inductive synthesis in the next iteration. We collectively refer to such synthesis techniques as counterexample-guided inductive synthesis, aka CEGIS.

In this paper, we conduct a theoretical study of CEGIS by examining the impact of using different kinds of validation engines which provide different nature of counterexamples. CEGIS has been successfully used across domains and has been applied to areas such as integer program synthesis and controller

K. Chatterjee, R. Ehlers, and S. Jha (Eds.):
Third Workshop on Synthesis (SYNT 2014)
EPTCS 157, 2014, pp. 84–99, doi:10.4204/EPTCS.157.10

design where the candidate set of designs is not finite, and the synthesis technique is not guaranteed to always succeed. This raises an interesting question whether the power of CEGIS can be improved by considering validation engines which provide *better* counterexamples than any arbitrary counterexample.

We consider two kinds of counterexamples in this paper.

- First, we consider *minimal* counterexamples instead of arbitrary counterexamples. For any predefined ordering on the examples, we require that the validation engine provide a counterexample which is minimal. This defines an alternative synthesis technique: Minimal Counterexample Guided Inductive Synthesis MinCEGIS where the validation engine returns *minimal* counterexamples.

  This choice of counterexamples is motivated by literature on debugging[1]. Significant effort has been made on improving validation engines to produce counterexamples which aid debugging by localizing the error. 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. Minimal counterexamples is inspired specifically from [28, 8].

- Second, we consider history bounded counterexamples where the counterexample produced by the validation engine must be smaller than a previously seen positive example. This defines another alternative synthesis technique: History Bounded Counterexample Guided Inductive Synthesis HCEGIS where the validation engine returns *history bounded* counterexamples.

  This choice of counterexample is also motivated by literature on debugging. In particular, [12, 35] use distance of the counterexample from a correct example to help debug programs. If the counterexample is very close to a correct example, then the error localization would be more accurate. We use a similar notion and force the counterexamples produced by the validation engine to be close to some previously seen correct example.

For each of the variants of CEGIS, we analyze whether it increases the candidate spaces of programs where a synthesizer terminates with correct program. We prove the following in the paper.

1. MinCEGIS successfully terminates with correct program on a candidate space if and only if CEGIS also successfully terminates with the correct program. So, there is no increase or decrease in power of synthesis by using minimal counterexamples.
2. HCEGIS can synthesize programs from some program classes where CEGIS fails to synthesize the correct program. But contrariwise, HCEGIS also fails at synthesizing programs from some program classes where CEGIS can successfully synthesize a program. Thus, their synthesis power is not equivalent, and none dominates the other.

Thus, none of the two counterexamples considered in the paper are strictly *good* mistakes. The history bounded counterexample can enable synthesis in additional classes of programs but it also leads to loss of some synthesis power.

2 Motivating Example

In this section, we present a simple example that illustrates why it is non-intuitive to estimate the change in power of synthesis when we consider alternative kinds of counterexamples. Consider synthesizing a

[1] Practically, this would mean replacing satisfiability solving based verification engines with those using Boolean optimization such as maximum satisfiability solving techniques.

program which takes as input a tuple of two integers (x,y) and outputs 1 if the tuple lies in a specific rectangle R (defined by diagonal points $(-1,-1)$ and $(1,1)$) and 0 otherwise.

The target program is:

$$if\ ((-1 \leq x \&\& x \leq 1) \&\& (-1 \leq y \&\& y \leq 1))\ \ op = 1\ \ else\ \ op = 0$$

The candidate program space is the space of all possible rectangles in $\mathbb{Z} \times \mathbb{Z}$ where $\mathbb{Z}$ denotes the set of integers, that is,

$$if\ ((\alpha_x \leq x \&\& x \leq \beta_x) \&\& (\alpha_y \leq y \&\& y \leq \beta_y))\ \ op = 1\ \ else\ \ op = 0$$

where $\alpha_x, \alpha_y, \beta_x, \beta_y$ are the parameters that need to be discovered by the synthesis engine.

Now, consider a radial ordering of (x,y) which uses $x^2 + y^2$ as the ordering index. If we consider synthesis using minimal counter-examples, it is clear that we can learn the rectangle: starting with an initial candidate program that always outputs 1 for all (x,y) in $\mathbb{Z} \times \mathbb{Z}$; validation engine producing minimum counterexamples would discover the rectangle boundaries. One possible sequence of minimal counterexamples would be $(0,2),(0,-2),(2,0),(-2,0)$. Since the boundary points form a finite set, `MinCEGIS` will terminate with the correct program. But if the counterexamples are arbitrary as in `CEGIS`, it is not obvious whether the rectangle can be still learnt. Our paper proves that `CEGIS` can also learn such a rectangle.

The question of synthesis power of different techniques using different nature of counterexamples is non-trivial when the space of programs is not finite. Even termination of inductive synthesis technique is not guaranteed when the candidate space of programs is infinite. Thus, the question of comparing the relative power of these synthesis techniques is interesting.

3 Related Work

Automated synthesis of systems using counterexamples has been widely studied in literature [32, 34, 19, 16, 7] as discussed in Section 1. While the applications of `CEGIS` to different domains have been very extensively investigated, theoretical characterization of the `CEGIS` approach independent of the application domain has received limited attention. To the best of our knowledge, this is the first attempt at a theoretical investigation into how the nature of counterexamples in `CEGIS` would impact the power of inductive synthesis technique to synthesize programs.

The inductive generalization used in `CEGIS` is similar to algorithmic learning from examples [11, 10, 21, 9, 29]. This relation between the two fields has been previously identified in [19]. A learning procedure is provided with strings from a formal language and the task of the learner is to identify the formal grammar for the language. Learning is an iterative inductive inference process. In each iteration, the learning procedure is provided a string. The string is either in the language, that is, it is a positive example, or the string is not in the language, that is, it is negative example. Based on the examples, the learning procedure proposes a formal grammar in each iteration. The learning procedure is said to be able to learn a formal language if the learner converges to the correct grammar of the formal language after a finite number of iterations. The algorithmic learning techniques can be classified across the following three dimensions:

1. Nature of examples: Examples could be restricted to only positive examples, or it could include negative examples too.

2. Memory of learner: The memory of the learner is allowed to grow infinitely or it could be bounded to a finite size.
3. Communication of examples to learner: The examples could be provided to the learner arbitrarily or as responses to specific kind of queries from the learner such as membership or subset queries.

We discuss the known theoretical results for algorithmic learning across these dimensions and identify how the results presented in this paper extend these existing results.

Gold [11] considered the problem of learning formal languages from examples. Similar inductive generalization techniques have been studied elsewhere in literature as well [18, 37, 5, 1]. 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 generalization. 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*. We examine the known results for both: *text learnable* and *informant learnable* classes of languages. None of the standard classes of formal languages are identifiable in the limit from text, that is, from only positive examples [11]. This includes regular languages, context-free languages and context-sensitive languages. It is also known that no class of language with at least one infinite language over the same vocabulary as the rest of the languages in the class, can be learnt purely from positive examples. We can illustrate this infeasibility of identifying languages from positive examples with a simple example.

Consider a vocabulary V and let V^* be all the strings that can be formed using vocabulary V. The strings in V^* are $x_1, x_2, \ldots$. Let us consider the set of languages

$$L_1 = V^* - \{x_1\}, L_2 = V^* - \{x_2\}, \ldots$$

Now a simple algorithm to learn languages from positive examples can guess the language to be L_i if x_i is the string with the smallest index not seen so far as a positive example. This algorithm can be used to inductively identify the correct language using just positive examples. But now, if we add a new language V^* which contains all the strings from vocabulary V to our class of language, that is,

$$L_2 = V^*, V^* - \{x_1\}, L_2 = V^* - \{x_2\}, \ldots$$

The above algorithm would fail to identify this class of languages.

In fact, no algorithm using positive examples would be able to inductively identify this class of languages. The key intuition is that if the data is all positive, no finite trace of positive data can distinguish whether the currently guessed language is the target language or is merely a subset of the target language. Now, if we consider the presence of negative counterexamples, the learning or synthesis algorithm can begin with the first guess as V^*. If there are no counterexamples, then V^* is the correct language. If a counterexample x_i is obtained, then the next guess is $V^* - \{x_i\}$, and this is definitely the correct language.

A detailed survey of classical results in learning from positive examples is presented in [24]. 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 [17] and [22]. In contrast to this line of work, `CEGIS` is a practical inductive generalization which restricts the memory of the synthesis engine or learner. At any step, the synthesis

engine only has the candidate design and response from the verifier which can be stored in a finite memory. Further, in contrast to learning from an infinite stream of positive and negative examples, CEGIS inductive generalization relies on using counterexamples. The positive and negative examples used in CEGIS are not arbitrary but rather they depend on the counterexample-generating verifier and the intermediate candidate programs proposed by the synthesis engine.

Another related line of work is that of techniques using iterative algorithmic learning with restricted memory of the learner [23, 36]. The learner or synthesis engine can use only finite memory but these techniques rely on availability of an infinite stream of positive examples in addition to negative examples. While the stream is not explicitly stored due to finite memory constraint, it can be used for synthesizing intermediate concepts. In contrast, CEGIS relies on using positive examples which are derived from the specification with respect to the counterexamples. These techniques differ from CEGIS in the dimension of how counterexamples are communicated to the learner or synthesis engine.

Angluin [2] considered a similar learning environment as CEGIS with respect to the communication of counterexamples to the learner or synthesis engine. Angluin's learning model consists of a teacher or oracle which provides responses to queries from the learner. The teacher in the context of Angluin is analogous to verifier in CEGIS and the learner is the synthesis engine. Similar learning models have also being proposed in [31, 14, 6, 15]. But they focus on complexity analysis of learning techniques using different kinds of queries such as membership queries, verification or equivalence queries and subset queries. In contrast, we restrict ourselves to verification queries and investigate the impact of substituting arbitrary counterexample producing verifiers with more powerful verifiers which produce counterexamples which are minimal or bounded.

Verification techniques have been adapted to provide more meaningful counterexamples [12, 28, 35, 8] for the purpose of aiding design debugging. The key idea is that these more powerful verification engines that provide not just any arbitrary counterexamples but rather a simpler counterexample with respect to some metric can be used for better debugging. These simpler or minimal counterexamples provide the most information to help localize bugs in a faulty design. If a counterexample trace is close to a correct trace and differs from a correct trace in a minimal way, then it can be used more effectively to localize the source of bug and fix it. It is natural to consider extending this use of minimal counterexamples for debugging to also enable more powerful synthesis. In this work, we conduct a theoretical analysis of using these more power verification engines and using counterexamples produced by these to aid CEGIS in synthesis.

4 Notation

In this section, we define some preliminary notation used in our definition and analysis of CEGIS and MinCEGIS. $\mathbb{N}$ represents the set of natural numbers. $\mathbb{N}_i \subset \mathbb{N}$ denotes a subset of natural numbers $\mathbb{N}_i = \{n | n < i\}$. $\min(S)$ denotes the minimal element in the set S. The union of the sets is denoted by $\cup$ and the intersection of the sets is denoted by $\cap$.

A sequence σ is a mapping from $\mathbb{N}$ to $\mathbb{N} \cup \{\perp\}$. 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 $\mathbb{N} \cup \{\perp\}$. $\sigma[0]$ is an empty sequence also denoted by σ_0. We denote the natural numbers in the range of $\sigma[i]$ by SMPL, that is, $\texttt{SMPL}(\sigma_i) = \mathit{range}(\sigma_i) - \{\perp\}$. The set of sequences is denoted by Σ.

We extend natural numbers to pairs. Let $\langle n_1, n_2 \rangle$ be any bijective computable function from $\mathbb{N} \times \mathbb{N} \to \mathbb{N}$ which is monotonically increasing in both of its arguments. Similarly, pairs can be extended to n-tuples. Assuming existence of such a bijective mapping, tuples can also be used in place of natural

numbers as elements of a language. A language in this case would be a subset of such tuples.

We also use standard definitions from computability theory [30]. A set L_i of natural numbers is called computable or recursive language if there is an program, that is, a computable, total function P_i such that for any natural number n, $P_i(n) = 1$ if $n \in L_i$ and $P_i(n) = 0$ if $n \notin L_i$. We denote the complement of language L_i by $\overline{L_i}$. We denote the union of two languages L_i and L_j by $L_i \cup L_j$, and the intersection of two languages L_i and L_j by $L_i \cap L_j$. Also for convenience, we use $L(P_i)$ to denote L_i using the one to one mapping between languages and programs that identify them. Thus, we distinguish only between semantically different programs and not the syntactically different programs which identify the same language.

The languages are sets of natural numbers. The natural numbers correspond to indexed elements of the language or valid input-output traces of the program. Without loss of generality, the natural ordering of natural numbers is used as an ordering of elements in the set. In practice, this will correspond to some user-provided ordering on the elements of the language. For example, for a program manipulating strings, we can choose alphabetical ordering and for program operating on numerical tuples, we can choose lexicographical ordering. We define a minimum operator $\min(L)$ which uses this natural ordering to report the minimum element in the language L. If the ordering is not total, $\min(L)$ denotes one of the minimal elements in the language L with respect to the given partial ordering.

Given a sequence $\mathscr{L}$ of non-empty languages $L_0, L_1, L_2, \ldots$, $\mathscr{L}$ is said to be an indexed family of languages if and only if there exists a recursive function `TEMPLATE` such that $\texttt{TEMPLATE}(i,n) = P_i(n)$. We denote the corresponding set of programs $P_0, P_1, P_2, \ldots$ by $\mathscr{P}$. For brevity, we refer to $\texttt{TEMPLATE}(i,n)$ also as $P_i(n)$. Intuitively, `TEMPLATE` defines the encoding of candidate program space similar to sketches in [33] and the component interconnection encoding in [19]. The index i is used to index into this encoding to select a particular program P_i. $P_i(n)$ denotes the output of the program on input n.

Inductive synthesis consists of synthesis engines T each of which identify the correct program P_i using a set of examples from the target language L_i from a given indexed family of languages $\mathscr{L}$. So, the overall synthesis problem is as follows. Let $\mathscr{P}$ be the class of candidate programs corresponding to indexed family of languages $\mathscr{L}$. No, given some target language L_i from $\mathscr{L}$, the synthesis engine receives a set of examples $\{n_1, n_2, \ldots\}$. The synthesis task is to identify P_i corresponding to L_i from the candidate programs $\mathscr{P}$. `CEGIS` is a particular kind of inductive synthesis techniques in which examples are obtained using counterexamples produced through iterative validation of inductively produced intermediate conjecture programs. We use the notations developed in this section to formally define concepts useful for theoretical analysis of `CEGIS` in the next section.

5 Definitions

In this section, we present some definitions. Trace is a sequence of examples from the target language L. The formal definition of trace is as follows:

Definition Trace τ: A trace τ for a language L is a sequence with $\texttt{SMPL}(\tau) = L$. $\tau[i]$ denotes the prefix of the trace τ of length i. $\tau(i)$ denotes the i-th element of the trace.

Counterexample guided inductive synthesis (`CEGIS`) techniques employ a verifier to provide counterexamples. So, we define verifiers for a language formally below and then, give a formal definition of a `CEGIS` engine denoted by $T_{\texttt{CEGIS}}$. Intuitively, the verifier returns a counterexample if the languages are different and returns $\perp$ if they are equivalent. We use one way difference instead of the symmetric difference between sets for ease of presentation.

Definition A verifier $\texttt{CHECK}_L$ for a language L is a non-deterministic mapping from $\mathscr{L}$ to $\mathbb{N} \cup \perp$ such that $\texttt{CHECK}_L(L_i) = \perp$ if and only if $L_i \subseteq L$, and $\texttt{CHECK}_L(L_i) \in L_i \cap \overline{L}$ otherwise.

Definition A CEGIS engine $T_{\texttt{CEGIS}} : \sigma \times \sigma \to \mathscr{P}$ is defined recursively below.
$T_{\texttt{CEGIS}}(\tau[n], \texttt{cex}[n]) = F(T_{\texttt{CEGIS}}(\tau[n-1], \texttt{cex}[n-1]), \tau(n), \texttt{cex}(n))$
where F is a recursive function $\mathscr{P} \times \mathbb{N} \times \mathbb{N} \to \mathscr{P}$ that characterizes the engine and how it eliminates counterexamples, $\tau[n]$ is a trace for language L and cex is a counterexample sequence such that
$\texttt{cex}(i) = \texttt{CHECK}_L(L(T_{\texttt{CEGIS}}(\tau[i-1], \texttt{cex}[i-1])))$.
$T_{\texttt{CEGIS}}(\sigma_0, \sigma_0)$ is a predefined constant representing an initial guess P_0 of the program, which for example, could be program corresponding to the universal language $\mathbb{N}$.

Intuitively, CEGIS is provided with a trace along with a counterexample trace formed by counterexamples to the latest conjectured languages. Thus, CEGIS receives two inputs. The counterexample is generated through a subset query.

Definition We say that $T_{\texttt{CEGIS}}$ converges to P_i if and only if for all, but finitely many prefixes $\tau[n]$ of τ, $T_{\texttt{CEGIS}}(\tau[n], \texttt{cex}[n]) = P_i$. We denote this by $T_{\texttt{CEGIS}}(\tau, \texttt{cex}) \to P_i$. In other words, $T_{\texttt{CEGIS}}(\tau, \texttt{cex}) \to P_i$ if and only if there exists k such that for all $n \geq k$, $T_{\texttt{CEGIS}}(\tau[n], \texttt{cex}[n]) = P_i$.

Definition $T_{\texttt{CEGIS}}$ identifies a language L_i if and only if for all traces τ of the language L_i and counterexample sequences cex, $T_{\texttt{CEGIS}}(\tau, \texttt{cex}) \to P_i$. $T_{\texttt{CEGIS}}$ identifies a language family $\mathscr{L}$ if and only if $T_{\texttt{CEGIS}}$ identifies every $L_i \in \mathscr{L}$.

We now define the set of language families that can be identified by the counterexample guided synthesis engines as CEGIS formally below.

Definition $\texttt{CEGIS} = \{\, \mathscr{L} \mid \exists T_{\texttt{CEGIS}} \,.\, T_{\texttt{CEGIS}} \text{ identifies } \mathscr{L} \,\}$

Now, we consider a variant of counterexample guided inductive synthesis where we use minimal counterexamples instead of arbitrary counterexamples MinCEGIS. We define a minimal counterexample generating verifier before defining MinCEGIS. This requires an ordering of the elements in the language.

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

Definition A MinCEGIS engine $T_{\texttt{MinCEGIS}} : \sigma \times \sigma \to \mathscr{P}$ is defined recursively below.
$T_{\texttt{MinCEGIS}}(\tau[n], \texttt{cex}[n]) = F(T_{\texttt{MinCEGIS}}(\tau[n-1], \texttt{cex}[n-1]), \tau(n), \texttt{cex}(n))$
where F is a recursive function $\mathscr{P} \times \mathbb{N} \times \mathbb{N} \to \mathscr{P}$ that characterizes the engine and how it eliminates counterexamples, $\tau[n]$ is a trace for language L and cex is a counterexample sequence such that
$\texttt{cex}(i) = \texttt{MINCHECK}_L(L(T_{\texttt{MinCEGIS}}(\tau[i-1], \texttt{cex}[i-1])))$.
$T_{\texttt{MinCEGIS}}(\sigma_0, \sigma_0)$ is a predefined constant representing an initial guess P_0 of the program, which for example, could be program corresponding to the language $\mathbb{N}$.

The convergence of the MinCEGIS synthesis engine to a language and family of languages is defined in similar way as CEGIS.

Definition We say that $T_{\texttt{MinCEGIS}}$ converges to P_i, that is, $T_{\texttt{MinCEGIS}}(\tau, \texttt{cex}) \to P_i$ if and only if there exists k such that for all $n \geq k$, $T_{\texttt{MinCEGIS}}(\tau[n], \texttt{cex}[n]) = P_i$.

Definition $T_{\texttt{MinCEGIS}}$ identifies a language L_i if and only if for all traces τ of the language L_i and counterexample sequences cex, $T_{\texttt{MinCEGIS}}(\tau, \texttt{cex}) \to P_i$. $T_{\texttt{MinCEGIS}}$ identifies a language family $\mathscr{L}$ if and only if $T_{\texttt{MinCEGIS}}$ identifies every $L_i \in \mathscr{L}$.

Definition $\texttt{MinCEGIS} = \{\ \mathscr{L} \mid \exists T_{\texttt{MinCEGIS}}\ .\ T_{\texttt{MinCEGIS}}$ identifies $\mathscr{L}\}$

Next, we consider another variant of counterexample guided inductive synthesis HCEGIS where we use *history bounded* counterexamples instead of arbitrary counterexamples. We define a history bounded counterexample generating verifier before defining HCEGIS. Unlike the previous cases, the verifier for generating history bounded counterexample is also provided with the trace seen so far by the synthesis engine. The verifier generates a counterexample smaller than the largest element in the trace. If there is no counterexample smaller than the largest element in the trace, then the verifier does not return any counterexample. From the definition below, it is clear that we need to only order elements in the language and do not need to define an ordering of $\perp$ with respect to the language elements since the comparison is done between an element in non-empty $L \cap L_i$ and elements $\tau[j]$ in the trace.

Definition A verifier $\texttt{HCHECK}_L$ for a language L is a mapping from $\mathscr{L} \times \sigma$ to $\mathbb{N} \cup \perp$ such that $\texttt{HCHECK}_L(L_i, \tau[n]) = m$ where $m \in \overline{L} \cap L_i \wedge m < \tau(j)$ for some $j \leq n$, and $\texttt{HCHECK}_L(L_i, \tau[n]) = \perp$ otherwise.

Definition A HCEGIS engine $T_{\texttt{HCEGIS}} : \sigma \times \sigma \to \mathscr{P}$ is defined recursively below. $T_{\texttt{HCEGIS}}(\tau[n], \texttt{cex}[n]) = F(T_{\texttt{HCEGIS}}(\tau[n-1], \texttt{cex}[n-1]), \tau(n), \texttt{cex}(n))$
where F is a recursive function $\mathscr{P} \times \mathbb{N} \times \mathbb{N} \to \mathscr{P}$ that characterizes the engine and how it eliminates counterexamples, $\tau[n]$ is a trace for language L and cex is a counterexample sequence such that $\texttt{cex}(i) = \texttt{HCHECK}_L(L(T_{\texttt{MinCEGIS}}(\tau[i-1], \texttt{cex}[i-1])), \tau[i-1])$.
$T_{\texttt{HCEGIS}}(\sigma_0, \sigma_0)$ is a predefined constant representing an initial guess P_0 of the program, which for example, could be program corresponding to the language $\mathbb{N}$.

The convergence of the HCEGIS synthesis engine to a language and family of languages is defined in similar way as CEGIS and MinCEGIS.

Definition We say that $T_{\texttt{HCEGIS}}$ converges to P_i, that is, $T_{\texttt{HCEGIS}}(\tau, \texttt{cex}) \to P_i$ if and only if there exists k such that for all $n \geq k$, $T_{\texttt{HCEGIS}}(\tau[n], \texttt{cex}[n]) = P_i$.

Definition $T_{\texttt{HCEGIS}}$ identifies a language L_i if and only if for all traces τ of the language L_i and counterexample sequences cex, $T_{\texttt{HCEGIS}}(\tau, \texttt{cex}) \to P_i$. $T_{\texttt{HCEGIS}}$ identifies a language family $\mathscr{L}$ if and only if $T_{\texttt{HCEGIS}}$ identifies every $L_i \in \mathscr{L}$.

Definition $\texttt{HCEGIS} = \{\ \mathscr{L} \mid \exists T_{\texttt{HCEGIS}}\ .\ T_{\texttt{HCEGIS}}$ identifies $\mathscr{L}\}$

6 Main Result

In this section, we present the main results of the paper. We first compare MinCEGIS and CEGIS in the first part of the section followed by HCEGIS and CEGIS in the second part of the section. Since the focus of our work is to analyze the impact of change in the power of counterexample providing verification engine, we fix the inductive generalization function F that eliminates counterexamples. So, we vary the counterexample generating verifier $\texttt{CHECK}_L$, $\texttt{MINCHECK}_L$ and $\texttt{HCHECK}_L$ but F is constant in our definitions of CEGIS, MinCEGIS and HCEGIS. In the rest of the section, we present the two central results of this paper:

1. $\texttt{MinCEGIS} = \texttt{CEGIS}$
2. $\texttt{HCEGIS} \neq \texttt{CEGIS}$

6.1 Synthesis Using Minimal Counterexamples

We investigate whether $\texttt{MinCEGIS} = \texttt{CEGIS}$ and prove that it is in fact true. So, replacing a verification engine which returns arbitrary counterexamples with a verification engine which returns minimal counterexamples does not increase the power of inductive synthesis system. The main result regarding this non-intuitive fact that there is no change in the power of synthesis technique by using minimal counterexamples is summarized in Theorem 6.1.

Theorem 6.1 *The power of synthesis techniques using arbitrary counterexamples and those using minimal counterexamples are equivalent, that is,* $\texttt{MinCEGIS} = \texttt{CEGIS}$.

Proof $\texttt{CEGIS} \subseteq \texttt{MinCEGIS}$ trivially. $\texttt{MINCHECK}_L$ is a special case of $\texttt{CHECK}_L$ and minimal counterexample reported by $\texttt{MINCHECK}_L$ can be treated as arbitrary counterexample to simulate CEGIS using MinCEGIS . Intuitively, using minimal counterexample is not worse than using arbitrary counterexamples.

The more interesting case to prove is $\texttt{MinCEGIS} \subseteq \texttt{CEGIS}$. For a language L, let MinCEGIS converge to P on trace τ. We show that $T_{\texttt{CEGIS}}$ can simulate $T_{\texttt{MinCEGIS}}$ and also converge to P on trace τ.

The proof idea is to simulate $T_{\texttt{MinCEGIS}}$ in two phases. In one phase, $T_{\texttt{CEGIS}}$ finds the minimal counterexample for a candidate language L_j by iteratively calling $\texttt{CHECK}_L$ on $L_j \cap \{i\}$ where $i = 0,1,2,3\ldots$. The minimum i for which $\texttt{CHECK}_L$ returns a counterexample for $L_j \cap \{i\}$ is the minimum counterexample. In the second phase, $T_{\texttt{CEGIS}}$ consumes the next elements from the trace. While searching for minimum counterexample, $T_{\texttt{CEGIS}}$ needs to store the backlog of the traces as well as cache the minimum counterexample for candidate languages.

We now present the formal description of the proof. For this simulation, we use some auxiliary variables maintained by $T_{\texttt{CEGIS}}$ which store some finite information required for simulating $T_{\texttt{MinCEGIS}}$. The key idea is for $T_{\texttt{CEGIS}}$ to iteratively guess the minimal counterexample in multiple micro-steps and then use that to simulate one step of $T_{\texttt{MinCEGIS}}$. But simulating each step of $T_{\texttt{MinCEGIS}}$ takes finite number of micro-steps for $T_{\texttt{CEGIS}}$ and uses finite storage.

The first auxiliary component for this simulation is a minimal counterexample map

$$\texttt{lce} : \mathscr{P} \to \mathbb{N} \cup \{\top\} \cup \{\bot\}$$

Intuitively, this maps a candidate program P_i (language L_i) to minimal counterexample as known to $T_{\texttt{CEGIS}}$ so far in simulating $T_{\texttt{MinCEGIS}}$. If minimal counterexample is not known for a given program, lce maps the program to $\top$. If there is no counterexample to a given program, lce maps the program to $\bot$. At any given step, only finite number of programs have their minimal counterexamples known, and the rest are mapped to $\top$.

Next, we define a mapping $T_{\texttt{lce}}$ from $\mathscr{P} \times \sigma \to \mathscr{P}$ which simulates $T_{\texttt{MinCEGIS}}$ based on the known lce so far, that is,

$$T_{\texttt{lce}}(P, \tau[n]) = P_n \texttt{ where } P_i = F(P_{i-1}, \tau(i), \texttt{lce}(P_{i-1})) \texttt{ for } i = 1,2,\ldots \texttt{ and } P_0 = P$$

if $\texttt{lce}(P_i)$ is defined for $i = 1,2,\ldots$ and it is undefined if $\texttt{lce}(P_i)$ is $\top$ for any i.

$T_{\texttt{lce}}$ simulates $T_{\texttt{MinCEGIS}}$ using the same counterexamples and intermediate candidate programs for the known history `lce`. If `lce` is $\top$ for any of the intermediate programs, $T_{\texttt{MinCEGIS}}$ is undefined. Further, we record the program proposed by $T_{\texttt{MinCEGIS}}$ into the variable $P^m_{\texttt{sim}}$ and the last program which initiated search for minimal counter example in P_{last}. $\tau^m_{\texttt{sim}}$ records the part of the trace already simulated by $T_{\texttt{CEGIS}}$ and μ is the candidate minimal counterexample while searching for minimal counterexample.

Initialization: All the internal auxiliary variables are initialized as follows. $P^0_{\texttt{sim}} = P_0$ which is the same initialization as $T_{\texttt{MinCEGIS}}$ being simulated, $\mu = 0$, $P_{last} = P_0$, and $\tau^0_{\texttt{sim}} = \sigma_0$. `lce` is initialized to map all P to $\top$ as no minimal counterexamples are known at the beginning.

Update: We describe the updates made in each iteration m. One of the following cases is true in each iteration.
Case 1: If $P_{last} = P^m_{\texttt{sim}}$, that is, we are in lock-step with the `MinCEGIS` synthesis algorithm with the same candidate program.
Case 1.1: If there is any counterexample for $P^m_{\texttt{sim}}$ (found using the verifier for $T_{\texttt{CEGIS}}$), that is, the candidate program has a counterexample and we need to find the corresponding minimal counterexample.
Case 1.1.1: If $\texttt{lce}(P^m_{\texttt{sim}})$ is not $\top$, that is, the minimal counterexample for candidate program is already part of `lce`.
Let τ_{done} be the longest prefix for $\tau^m_{\texttt{sim}}\tau(m+1)$ such that $T_{\texttt{lce}}(P^m_{\texttt{sim}}, \tau_{done})$ is defined. $\tau_{done}\tau^{m+1}_{\texttt{sim}} = \tau^m_{\texttt{sim}}\tau(m+1)$, $P^{m+1}_{\texttt{sim}} = T_{\texttt{lce}}(P^m_{\texttt{sim}}, \tau_{done})$, $P_{last} = P^{m+1}_{\texttt{sim}}$
We use the minimal counterexample from `lce` and then advance the simulation τ_{done} traces ahead if $T_{\texttt{lce}}$ can simulate the trace using minimal counterexamples from `lce` for all the intermediate candidate programs.
Case 1.1.2: If $\texttt{lce}(P^m_{\texttt{sim}})$ is $\top$, that is, the minimal counterexample for candidate program is not known.
$\tau^{m+1}_{\texttt{sim}} = \tau^m_{\texttt{sim}}\tau(m+1)$, $P^{m+1}_{\texttt{sim}} = P^m_{\texttt{sim}} \cap \{0\}$
We initialize the candidate language $P^{m+1}_{\texttt{sim}}$ for searching for minimal counterexample to $P^m_{\texttt{sim}} \cap \{0\}$, that is, it is either the language consisting only of the minimal element $\{0\}$ or is empty. Since our verifier uses a subset query, empty language will return no counterexamples.
Case 1.2: If there is no counterexample for $P^m_{\texttt{sim}}$,
Let τ_{done} be the longest prefix for $\tau^m_{\texttt{sim}}\tau(m+1)$ such that $T_{\texttt{lce}}(P^m_{\texttt{sim}}, \tau_{done})$ is defined. $\tau_{done}\tau^{m+1}_{\texttt{sim}} = \tau^m_{\texttt{sim}}\tau(m+1)$, $P^{m+1}_{\texttt{sim}} = T_{\texttt{lce}}(P^m_{\texttt{sim}}, \tau_{done})$, $P_{last} = P^{m+1}_{\texttt{sim}}$ and $\texttt{lce}(P^m_{\texttt{sim}}) = \bot$,
The candidate program seen so far is subset of the target language and we consume as much of the trace τ_{done} as possible for which $T_{\texttt{lce}}$ is defined.

Case 2: If $P_{last} \neq P^m_{\texttt{sim}}$, that is, the simulation is trying to find the minimum counterexample as a result of case 1.1.2.
Case 2.1: If there is any counterexample $\texttt{cex}_{\texttt{sim}}$ for $P^m_{\texttt{sim}}$ (found using the verifier for $T_{\texttt{CEGIS}}$),
Update $\texttt{lce}(P_{last}) = \texttt{cex}_{\texttt{sim}}$. Let τ_{done} be the longest prefix for $\tau^m_{\texttt{sim}}\tau(m+1)$ such that $T_{\texttt{lce}}(P_{last}, \tau_{done})$ is defined. $\tau_{done}\tau^{m+1}_{\texttt{sim}} = \tau^m_{\texttt{sim}}\tau(m+1)$, $P^{m+1}_{\texttt{sim}} = T_{\texttt{lce}}(P_{last}, \tau_{done})$, $P_{last} = P^{m+1}_{\texttt{sim}}$, $\mu = 0$
If there is a counterexample, since the candidate language was a single element set or empty, and verification engine checks for containment in the target language, the only element in the language has to be the counterexample. Further, starting from step 1.1.2 and with possible increments in step 2.2, we stop with the minimal counterexample in this step and add it to the `lce`.
Case 2.2: If there is no counterexample for $P^m_{\texttt{sim}}$, that is, we have not yet found the minimal counterexample.
$\mu = \mu + 1$, $P^{m+1}_{\texttt{sim}} = P_{last} \cap \{\mu\}$, and $\tau^{m+1}_{\texttt{sim}} = \tau^m_{\texttt{sim}}\tau(m+1)$.

We increment μ and search for whether $P_{last} \cap \{\mu\}$ is in the target language. This is either empty or is a language consisting of a single element $\{\mu\}$.

Progress: Now, we first show progress of the simulation in parsing trace $\tau[m]$. For any m, there exists $m' > m$ such that $\tau[m] = \tau^m_{done}\tau^m_{\texttt{sim}}$, $\tau[m'] = \tau^{m'}_{done}\tau^{m'}_{\texttt{sim}}$ and τ^m_{done} is a proper prefix of $\tau^{m'}_{done}$. This follows from the observation that Case 2.2 can not be repeated infinitely after Case 1.1.2 since P_{last} has at least one counterexample. So, case 2.1 would eventually become true and since `lce` is extended, $T_{\texttt{MinCEGIS}}$ would be defined for a longer prefix.

Correctness: Let $T_{\texttt{MinCEGIS}}$ converge on τ after reading prefix $\tau[n]$. From progress, after some m, $\tau[n]$ would be a prefix of τ^m_{done}. Since $T_{\texttt{MinCEGIS}}$ converges after reading $\tau[n]$, $F(P_n, \tau(n'), \texttt{cex}(n')) = P_n$ for $n' > n$. Now, `lce` is not $\top$ for all intermediate programs $P_{m''}$ in $T_{\texttt{MinCEGIS}}$ for $m'' \leq m$. So, $P^m_{\texttt{sim}} = T_{\texttt{lce}}(P^0_{\texttt{sim}}, \tau[m]) = T_{\texttt{lce}}(P_0, \tau[m]) = P_n$ and for all $m' > m$, $P^{m'}_{\texttt{sim}} = F(P_n, \tau(n'), \texttt{cex}(n')) = P_n$ So, $T_{\texttt{CEGIS}}$ also converges to P_n, that is, `MinCEGIS` $\subseteq$ `CEGIS`.

Thus, `MinCEGIS` $=$ `CEGIS`.

Thus, `MinCEGIS` successfully terminates with correct program on a candidate space if and only if `CEGIS` also successfully terminates with the correct program. So, there is no increase or decrease in power of synthesis by using minimal counterexamples.

6.2 Synthesis Using History Bounded Counterexamples

We investigate whether `HCEGIS` $=$ `CEGIS` or not, and prove that they are not equal. So, replacing a verification engine which returns arbitrary counterexamples with a verification engine which returns counterexamples bounded by history has impact on the power of the synthesis technique. But this does not strictly increase the power of synthesis. Instead, the use of history bounded counterexamples does allow programs from new classes to be synthesized but at the same time, program from some program classes which were amenable to `CEGIS` can no longer be synthesized using history bounded counterexamples. The main result regarding the power of synthesis techniques using history bounded counterexamples is summarized in Theorem 6.2.

Theorem 6.2 *The power of synthesis techniques using arbitrary counterexamples and those using history bounded counterexamples are not equivalent, and none is more powerful than the other.* `HCEGIS` $\neq$ `CEGIS`. *In fact,* `HCEGIS` $\not\subseteq$ `CEGIS` *and* `CEGIS` $\not\subseteq$ `HCEGIS`.

We prove this using the following two lemma. The first lemma 6.3 shows that there is a family of languages from which a program recognizing a language can be synthesized by `CEGIS` but, this can not be done by `HCEGIS`. The second lemma 6.4 shows that there is another family of languages from which a program recognizing a language can be synthesized by `HCEGIS` but not by `CEGIS`.

Lemma 6.3 *There is a family of languages $\mathscr{L}$ such that for the candidate programs $\mathscr{P}$ corresponding to $\mathscr{L}$,* `HCEGIS` *can not synthesize a program P in $\mathscr{P}$ recognizing some language L in $\mathscr{L}$ but* `CEGIS` *can synthesize P, that is,* `CEGIS` $\not\subseteq$ `HCEGIS`

Proof Consider the languages formed by upper bounding the elements by some fixed constant, that is,

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

Now, consider the family of languages consisting of these, that is, $\mathscr{L} = \{L_i | i \in \mathbb{N}\}$. Given this family $\mathscr{L}$, let the target language L (for which we want to synthesize a recognizing program P) be L_i.

If we obtain a trace $\tau[j]$ at any point in synthesis using history bounded counterexamples, then for any intermediate program P_j proposed by $T_{\texttt{HCEGIS}}$, $\texttt{HCHECK}_L$ 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, $T_{\texttt{HCEGIS}}$ can not synthesize P corresponding to 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, $\texttt{CEGIS} \not\subseteq \texttt{HCEGIS}$.

This shows that CEGIS can be used to identify programs when HCEGIS 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 the nonintuitive result that this restriction enables synthesis of programs which can not be synthesized by CEGIS. The proof uses a diaganolization argument similar to the argument used in [24] for showing the increase in inductive synthesis power when negative examples are introduced in addition to the positive examples. This argument is presented in Section 3. Recall that the set of languages considered in that case were $L_1 = V^* - \{x_1\}, L_2 = V^* - \{x_2\}, \ldots$ and the language V^*. The argument relies on indistinguishability of $V^* - \{x_1\}$ and V^* with respect to finite traces of positive examples.

In the proof below, we similarly 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. We use the tuple notation introduced in Section 4 to clearly identify the diagnolization.

Lemma 6.4 *There is a family of languages $\mathscr{L}$ such that for the candidate programs $\mathscr{P}$ corresponding to $\mathscr{L}$,* CEGIS *can not synthesize a program P in $\mathscr{P}$ recognizing some language L in $\mathscr{L}$ but* HCEGIS *can synthesize P, that is,* $\texttt{HCEGIS} \not\subseteq \texttt{CEGIS}$

Proof Consider the following languages $L_i^{01} = \{\langle j, n\rangle | j \in \{0,1\}, n \in \mathbb{N}\}$. We now construct a family of languages in L_i^{01} which are finite and have atleast one element of the form $\langle 1, .\rangle$, that is,

$$\mathscr{L}^{fin} = \{L_i^{01} | i \in \mathbb{N} \wedge |L_i^{01}| \text{is finite} \wedge \exists k \, s.t. \, \langle 1,k\rangle \in L_i^{01}\}$$

Now consider the languages L_i which are subsets of $\mathbb{N}$. We consider only those languages L_i such that the index i of the language is also the smallest element in the language, that is, $i = \min(L_i)$. We now build a language of pairs as follows: $L_i^{diag} = \{\langle 0, n\rangle | n \in L_i\}$ if $i = \min(L_i)$ and undefined, otherwise We construct a second family of languages using these languages. $\mathscr{L}^{diag} = \{L_i^{diag}\}$ if L_i^{diag} is defined for

index i. Now, we consider the following family of languages

$$\mathscr{L} = \mathscr{L}^{fin} \cup \mathscr{L}^{diag}$$

We show that there is a language L in $\mathscr{L}$ such that the program P recognizing L can not be synthesized by CEGIS but HCEGIS can synthesize all programs recognizing any language in $\mathscr{L}$.

The key intuition is as follows. If the examples seen by synthesis algorithm are all of the form $\langle 0,.\rangle$, then any synthesis technique can not differentiate whether the language belongs to $\mathscr{L}^{fin}$ or $\mathscr{L}^{diag}$. If the language belongs to $\mathscr{L}^{fin}$, the synthesis engine would eventually obtain an example of the form $\langle 1,.\rangle$ (since each language in $\mathscr{L}^{fin}$ has atleast one element of this kind and these languages are finite). While the synthesis technique using arbitrary counterexamples can not recover the previous examples, the techniques with access to the verifier which produces history bounded counterexamples can recover all the previous examples.

We can easily specify a $T_{\texttt{HCEGIS}}$ which can synthesize programs that correspond to languages in $\mathscr{L}$. $T_{\texttt{HCEGIS}}$ works as follows. If all the elements seen so far are of the form $\langle 0,.\rangle$, then the synthesis algorithm and picks the minimum j such that $\langle 0,j\rangle$ has been seen as an example by the synthesis engine. The proposed program is P_j corresponding to L_j. If the proposed program is not the correct program, HCHECK returns $\langle 0,j_{ce}\rangle$ such that $j_{ce} < j$. This is guaranteed since HCHECK returns counterexamples smaller than the examples seen so far, and we have assumed that P_j is not correct. So, iteratively, the algorithm would discover a language from $\mathscr{L}^{diag}$ eventually. But if the language is from $\mathscr{L}^{fin}$, then we know that all languages in $\mathscr{L}^{fin}$ are finite and have at least one element of the form $\langle 1,.\rangle$. After $T_{\texttt{HCEGIS}}$ sees $\langle 1,.\rangle$, for every future positive example x, it queries HCHECK with the singleton language having only one element $\{\langle x+2,0\rangle\}$. Clearly, $\langle x+2,0\rangle$ is not in the language since it only contains elements of the form $\langle 0,.\rangle$ and $\langle 1,.\rangle$. But HCHECK returns no counterexample for $\{\langle x_{max}+2,0\rangle\}$ if x_{max} is the largest positive example seen so far. At this point, we can recover all positive examples seen previously by enumerating all $x' < x_{max}$ and testing the candidate language $\{x'\}$ with HCHECK. We get a counterexample if and only if x' is not in the target language. Further, the target language is finite and hence, enumerating members of the language is sufficient to identify the target language after consuming a finite trace. Thus, $T_{\texttt{HCEGIS}}$ can synthesize programs corresponding to any language in $\mathscr{L}$.

We now prove the infeasibility of CEGIS for this class of languages. Let us assume that $\mathscr{L} \in$ CEGIS. So, there is a synthesis engine $T_{\texttt{CEGIS}}$ which can synthesize programs corresponding to languages in $\mathscr{L}$. Let us consider trace τ and counterexample sequence cex such that $T_{\texttt{CEGIS}}$ converges in n steps that is $T_{\texttt{CEGIS}}(\tau,\texttt{cex}) \rightarrow T_{\texttt{CEGIS}}(\tau[n],\texttt{cex}[n])$. Now, $\texttt{cex}[n]$ is a valid counterexample sequence of any language L such that $\texttt{SMPL}(\tau[n]) \subseteq L \subseteq \mathbb{N} - \texttt{SMPL}(\texttt{cex}[n])$. Since $T_{\texttt{CEGIS}}$ must recognize a language from any trace and any arbitrary counterexample sequence, we choose a trace and counterexample sequence as follows. Let us consider a trace τ' of the form $\tau[n](\langle 1,z_1\rangle)^{\infty}$. The corresponding counterexample trace discovered by $T_{\texttt{CEGIS}}$ is $\texttt{cex}[n]$ followed by minimal counterexamples, if any, after observing $\langle 1,z_1\rangle$. Now, we pick an element $\langle 0,z_2\rangle$ such that $\langle 0,z_2\rangle \notin \texttt{cex}[n]$ and $\langle 0,z_2\rangle \notin \tau[n]$. Since $\texttt{cex}[n]$ is a valid counterexample sequence of any language L such that $\texttt{SMPL}(\tau[n]) \subseteq L \subseteq \mathbb{N} - \texttt{SMPL}(\texttt{cex}[n])$, the behavior of $T_{\texttt{CEGIS}}$ is same for $\tau[n](\langle 1,z_1\rangle)^{\infty}$ as it is for $\tau[n]\langle 0,z_2\rangle(\langle 1,z_1\rangle)^{\infty}$. Thus, $T_{\texttt{CEGIS}}$ can not distinguish between the two languages: $L^d = \texttt{SMPL}(\tau[n]) \cup \{\langle 1,z_1\rangle\}$ and $L^{d'} = \texttt{SMPL}(\tau[n]) \cup \{\langle 0,z_2\rangle,\langle 1,z_1\rangle\}$. Intuitively, $T_{\texttt{CEGIS}}$ can forget some positive examples seen before observing $\langle 1,z_1\rangle$ and there is no way to regenerate these as it can be done with $T_{\texttt{HCEGIS}}$.

Thus, HCEGIS $\not\subseteq$ CEGIS.

Hence, HCEGIS can synthesize programs from some program classes where CEGIS fails to synthesize the correct program. But contrariwise, HCEGIS also fails at synthesizing programs from some program

classes where CEGIS can successfully synthesize a program. Thus, their synthesis power is not equivalent, and none dominates the other.

7 Discussion and Conclusion

The paper presents formal analysis of the impact of counterexample selection on what programs can be synthesized, without any restriction on the type of program other than it be from a countable set. We have shown that the use of minimal counterexamples does not enable synthesizing programs from newer space of candidate programs. In practice, this means that any domain where MinCEGIS can be used, use of CEGIS would also be possible since MinCEGIS successfully terminates with correct program on a candidate space if and only if CEGIS also successfully terminates with the correct program. So, there is no increase or decrease in the power of synthesis by using minimal counterexamples. But HCEGIS can synthesize programs from some program classes where CEGIS fails to synthesize the correct program. Contrariwise, HCEGIS also fails at synthesizing programs from some program classes where CEGIS can successfully synthesize a program. Thus, their synthesis power is not equivalent, and none dominates the other. This paper is a first step towards the theoretical characterization of Counterexample Guided Inductive Synthesis technique: CEGIS.

Further analysis of CEGIS is pertinent given the widespread adoption of CEGIS as one of the standard paradigms for automated synthesis. We envision the following directions in which further work can be done to better understand the power of CEGIS techniques.

- Speed of convergence: MinCEGIS and CEGIS have equal synthesis power and if one of the techniques successfully identifies a program from a given program class, the other would also be able to successfully synthesize this program. But would both techniques need the same number of counterexamples for successfully synthesizing the program ? If we measure the complexity of automated synthesis using the number of counterexamples needed to synthesize a program, the comparison of the complexity of MinCEGIS and CEGIS is open.

  Similarly, for the program spaces on which both HCEGIS and CEGIS terminate, can we compare the number of counterexamples needed by the two techniques to synthesize a program.

- Newer variants of counterexamples: The two new variants of counterexamples considered in this paper; namely, the minimal counterexamples and the history bounded counterexamples are not the only variants that can be used in CEGIS. The question of whether there are other variants of counterexamples which would enable synthesis in program spaces beyond the power of conventional CEGIS is open.

  In particular, consider another new variant of counterexamples which are minimal counterexamples among all the counterexamples which are larger than the largest positive examples seen so far. This counterexample captures another notion of being *close to correct* counterexample, and it would be interesting to investigate whether it increases the power of CEGIS.

In summary, we presented variants of CEGIS using different kinds of counterexamples and compared the power of these variant synthesis techniques with CEGIS. This is a first step towards a better theoretical understanding of the synthesis power of CEGIS technique.

References

[1] 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.

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

[3] Sorav Bansal & Alex Aiken (2006): *Automatic Generation of Peephole Superoptimizers*. In: *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XII, ACM, New York, NY, USA, pp. 394–403, doi:10.1145/1168857.1168906.

[4] Sorav Bansal & Alex Aiken (2008): *Binary Translation Using Peephole Superoptimizers*. In: *OSDI*, pp. 177–192.

[5] 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.

[6] A Blumer, A Ehrenfeucht, D Haussler & M Warmuth (1986): *Classifying Learnable Geometric Concepts with the Vapnik-Chervonenkis Dimension*. In: *Proceedings of the Eighteenth Annual ACM Symposium on Theory of Computing*, STOC '86, ACM, New York, NY, USA, pp. 273–282, doi:10.1145/12130.12158.

[7] Krishnendu Chatterjee, Thomas A. Henzinger, Ranjit Jhala & Rupak Majumdar (2005): *Counterexample-guided Planning*. In: *UAI*, AUAI Press, pp. 104–111.

[8] 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.

[9] A. Cornuejols (1993): *Getting Order Independence in Incremental Learning*. In: *Proc. of the 1993 AAAI Spring Symposium on Training Issues in Incremental Learning*, Stanford, California, pp. 42–52.

[10] Jerome A. Feldman (1972): *Some Decidability Results on Grammatical Inference and Complexity*. *Information and Control* 20(3), pp. 244–262, doi:10.1016/S0019-9958(72)90424-X.

[11] 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.

[12] Alex Groce, Sagar Chaki, Daniel Kroening & Ofer Strichman (2006): *Error Explanation with Distance Metrics*. *Int. J. Softw. Tools Technol. Transf.* 8(3), pp. 229–247, doi:10.1007/s10009-005-0202-0.

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

[14] David Haussler (1986): *Quantifying the Inductive Bias in Concept Learning (Extended Abstract)*. In Tom Kehler, editor: *AAAI*, Morgan Kaufmann, pp. 485–489. Available at http://www.aaai.org/Library/AAAI/1986/aaai86-081.php.

[15] Tibor Hegedűs (1994): *Geometrical Concept Learning and Convex Polytopes*. COLT '94, ACM, New York, NY, USA, pp. 228–236, doi:10.1145/180139.181124.

[16] Thomas A. Henzinger, Ranjit Jhala & Rupak Majumdar (2003): *Counterexample-guided Control*. ICALP'03, Springer-Verlag, Berlin, Heidelberg, pp. 886–902.

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

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

[19] 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.

[20] Rajeev Joshi, Greg Nelson & Keith Randall (2002): *Denali: A Goal-directed Superoptimizer*. In: *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, PLDI '02, ACM, New York, NY, USA, pp. 304–314, doi:10.1145/512529.512566.

[21] Michael J. Kearns, Robert E. Schapire & Linda M. Sellie (1992): *Toward Efficient Agnostic Learning*. In: *Proceedings of the Fifth Annual Workshop on Computational Learning Theory*, COLT '92, ACM, New York, NY, USA, pp. 341–352, doi:10.1145/130385.130424.

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

[23] Steffen Lange & Thomas Zeugmann (1996): *Incremental Learning from Positive Data*. *Journal of Computer and System Sciences* 53(1), pp. 88 – 103, doi:10.1006/jcss.1996.0051.

[24] 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.

[25] 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.

[26] Zohar Manna & Richard Waldinger (1992): *Fundamentals Of Deductive Program Synthesis*. *IEEE Transactions on Software Engineering* 18, pp. 674–704, doi:10.1109/32.153379.

[27] Henry Massalin (1987): *Superoptimizer: A Look at the Smallest Program*. *SIGARCH Comput. Archit. News* 15(5), pp. 122–126, doi:10.1145/36177.36194.

[28] 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.

[29] Sara Porat & JeromeA. Feldman (1991): *Learning automata from ordered examples*. *Machine Learning* 7(2-3), pp. 109–138, doi:10.1007/BF00114841.

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

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

[32] Armando Solar Lezama (2008): *Program Synthesis By Sketching*. Ph.D. thesis, EECS Department, University of California, Berkeley.

[33] 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.

[34] Saurabh Srivastava, Sumit Gulwani & Jeffrey S. Foster (2010): *From Program Verification to Program Synthesis*. *SIGPLAN Not.* 45(1), pp. 313–326, doi:10.1145/1707801.1706337.

[35] Chao Wang, Zijiang Yang, Franjo Ivančić & Aarti Gupta (2006): *Whodunit? Causal Analysis for Counterexamples*. In Susanne Graf & Wenhui Zhang, editors: *Automated Technology for Verification and Analysis*, *Lecture Notes in Computer Science* 4218, Springer Berlin Heidelberg, pp. 82–95, doi:10.1007/11901914_9.

[36] Rolf Wiehagen (1976): *limit detection of recursive functions by specific strategies*. *Electronic Information Processing and Cybernetics* 12(1/2), pp. 93–99.

[37] 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.